RICE UNIVERSITY

Portable Programming Models for Heterogeneous Platforms

by

Deepak Majeti

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Vivek Sarkar, Chair
E.D. Butcher Chair of Engineering.
Professor of Computer Science

John Mellor-Crummey
Professor of Computer Science

Timothy C. Warburton
Professor of Computational and Applied Mathematics

Rajkishore Barik
Senior Research Scientist at Intel Labs

Houston, Texas

May, 2015

ABSTRACT

Portable Programming Models for Heterogeneous Platforms

by

Deepak Majeti

With the end of Dennard scaling and emergence of dark silicon, the bets are high on heterogeneous architectures to achieve both application performance and energy efficiency. However, diversity in heterogeneous architectures poses severe programming challenges in terms of data layout, memory coherence, task partitioning, data distribution, and sharing of virtual addresses. Existing high-level programming languages are inadequate to address these new architectural features since they lack the necessary abstractions to address the challenges mentioned above. It is necessary for existing languages to be extended minimally with high-level constructs while maintaining existing standards of portability, performance, and productivity. The compiler and runtime together must efficiently map these constructs to a target architecture.

We introduce *Concord*, a C++ based programming model that extends the Intel Threading Building Blocks onto integrated heterogeneous CPU+GPU architectures that do not share the same virtual address between CPU and GPU. *Concord* supports many C++ features including virtual functions. We implement Shared Virtual Memory to map applications with pointer intensive data structures onto heterogeneous architectures that do not share the same virtual address.

We introduce Heterogeneous Habanero-C (*H2C*), an implementation of the Habanero execution model targeting modern heterogeneous architectures with multiple

devices. *H2C* provides high-level constructs to specify the computation, communication and synchronization in a given application. The *H2C* compiler and runtime frameworks efficiently map these high-level constructs onto underlying heterogeneous hardware. The highlights of *H2C* include: a data layout framework to generate code with best data layout suited for a given memory hierarchy; constructs to specify a task partition, leaving the complex analysis of determining the resultant data distribution to the compiler; and a unified event framework that allows a programmer to implement applications with a macro data-flow model for current heterogeneous architectures.

Experimental results show that *Concord* and *H2C* provide good portability, productivity, and performance. We believe that programming systems like *H2C* and *Concord* that have a tight integration of language, compiler and runtime are the right way to target current and future heterogeneous systems.

Acknowledgments

I am grateful to my thesis advisor Prof. Vivek Sarkar for his guidance, encouragement, and patience. His constant feedback and suggestions made my journey through graduate school very memorable and worthwhile. He motivated me in all my academic and personal endeavors. It is an honor to have worked with him. I would like to thank my other thesis committee members Prof. Timothy C. Warburton and Prof. John Mellor-Crummey for their time and feedback. I am very thankful to my thesis co-chair, mentor, and friend, Rajkishore Barik for his guidance and collaboration throughout my doctoral studies in both research and personal fronts. He motivated me to think beyond my threshold and helped me overcome my shortcomings.

I am fortunate to have worked with many collaborators and friends during my graduate career including Mauricio Breternitz at AMD, Kuldeep Meel, Dragos Sbîrlea, Alina Sbîrlea, Shams Imam, Jisheng Zhao, Karthik Murthy, Rishi Surendran, Milind Chabbi, Nikita Kozin, Prasanth Chatarasi, Vivek Kumar and Max Grossman at Rice University. I am thankful to all the members of the Habanero group for their support and technical discussions. During my graduate program, I am fortunate to have worked on projects in collaboration with various organizations including Intel, AMD, IBM, Texas Instruments, Halliburton, UCLA, LLNL and Cray. The experience gained from these projects is invaluable.

I would like to thank the Rice University Computer Science department for providing a scholarly environment. I would also like to thank the administrative staff who promptly helped me in various administrative works. I am also thankful to the members of the Indian Students at Rice (ISAR) and Graduate Student Association for organizing various activities and ensuring a stress-free graduate life. My role as

the president for ISAR during 2012 - 2013 is a memorable experience.

Finally, I am indebted to my parents Jagan Mohan Majeti and Shanthi Mohan Majeti for their patience and support. My brother Karthik Majeti played an important role in encouraging me through the doctoral program. I am more than lucky to have met my wife Priyanka Raja at Rice. She stood by me during my worst times and was always there to cheer me up. I would also like to thank my other friends who made my stay in Houston enjoyable including Rajesh Gandham, Aarthi Muthuswamy, Reshmy Mohanan, Shruti Kashinath, Gaurav Patel, Rajoshi Biswas and Rahul Kumar.

This work was supported in part by the Ken Kennedy Institute for Information Technology 2014/15 ExxonMobil Graduate Fellowship.

Contents

Illustrations

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Over the years, we have observed Moore's law [1], which states that the "*number of transistors per square inch double every two years*". Robert Dennard, in his classic 1974 paper [2] formulated MOSFET scaling which showed that as transistors get smaller, they can switch faster and consume less power. As a result from Moore and Dennard's work, when transistors shrank (along with changes to the doping and lithography process), we enjoyed faster processors. However, Dennard did not consider leakage currents that were insignificant at the micrometer scale. Around 2005, these leakage currents came to dominate the total power consumption, and Dennard scaling came to an end. A reduction in transistor size could no longer lead to faster processors. A direct consequence of the end of Dennard scaling is the emergence of Dark Silicon [3]. Dark Silicon describes the growing gap between how many transistors fit into a chip with each lithography shrink vs. how many transistors can be used simultaneously for a given power budget. Essentially, Dark Silicon prevents all the transistors on a chip from being operational at the same instance. To take advantage of the abundant transistors available, chip manufacturers are adopting heterogeneous architectures to achieve increased performance and energy efficiency.

In the last decade, various heterogeneous architectures have become pervasive from

mobile phones to supercomputers. *We define a heterogeneous architecture to be a system that has more than one kind of processor.* Few examples include CellBE [4] from IBM; CPU+GPU architectures [5], [6], [7] from Intel, AMD, NVIDIA; CPU+DSP architectures [8] from Texas Instruments; CPU+FPGA architectures [9] from Altera, Xilinx and other custom processors [10] from vendors like Cadence. These heterogeneous systems with their superior computational capabilities have opened opportunities to solve massive computational problems that include DNA sequencing, medical imaging, big-data analytics, human brain simulation, and particle simulations. The current top two supercomputers: Tianhe-2 from China, and Titan from United States have heterogeneous hardware [11]. Most mobile smartphones today are increasingly becoming heterogeneous. For instance, the Apple iPhone6 has a motion co-processor, a multi-core CPU + GPU processor and various sensors including a barometer, accelerometers, gyroscopes and compasses.

We claim that heterogeneous architectures are here to stay. There are at least two reasons why we believe heterogeneous architectures will remain for many years to come. The first reason is that the relationship between Moore's law and Dennard scaling has come to an end. This is due to the leakage currents introduced by the transistors at the nanometer scale. As a result, decreasing transistor sizes along with lithographic changes fail to increase the clock frequency. Hardware manufacturers are resorting to heterogeneous multicore architectures to deliver increased performance and energy savings. Dark Silicon is another potential source of heterogeneity as hardware designers have begun to utilize the extra transistors to implement different kind of cores [12].

The second reason we believe heterogeneous architectures will prevail is because of Internet of Things (IoT) [13]. IoT is fast catching up and has the potential to disrupt

	Classification Of Heterogeneous Architectures	Examples
1	Non-coherent Memory + Non-shared Virtual Memory	Intel CPU + Discrete GPU
2	Coherent Memory + Non-shared Virtual Memory	Intel Ivy-Bridge
3	Coherent Memory + Shared Virtual Memory	HSA, NVIDIA UVA

Table 1.1 : Classification of heterogeneous architectures

the processor ecosystem and influence the way we build our next super-computer. IoT not only consists of "big data" generated from various sensors, but also involves a constant feedback mechanism. Not many years from now, when you wake up, your pillow will communicate your sleep pattern to the coffee machine and this will in turn decide the strength of the coffee you are served. Your wardrobe will suggest your clothes based on the weather forecast and your calendar schedule for the day. This feedback mechanism requires a constant interaction with heterogeneous devices. Your home could be the next heterogeneous architecture!

Classification of Heterogeneous Architectures

Recent heterogeneous hardware can be classified into three main categories as described in Table 1.1. We briefly describe some terminology below to better understand this classification.

- Virtual Memory Sharing: If two processors have the same virtual address to physical memory mapping, then the system is termed as a Shared Virtual Memory (otherwise Non-shared Virtual Memory).

- Memory Coherence: If a write to a memory location by one processor can be immediately seen by another processor without any additional programming,

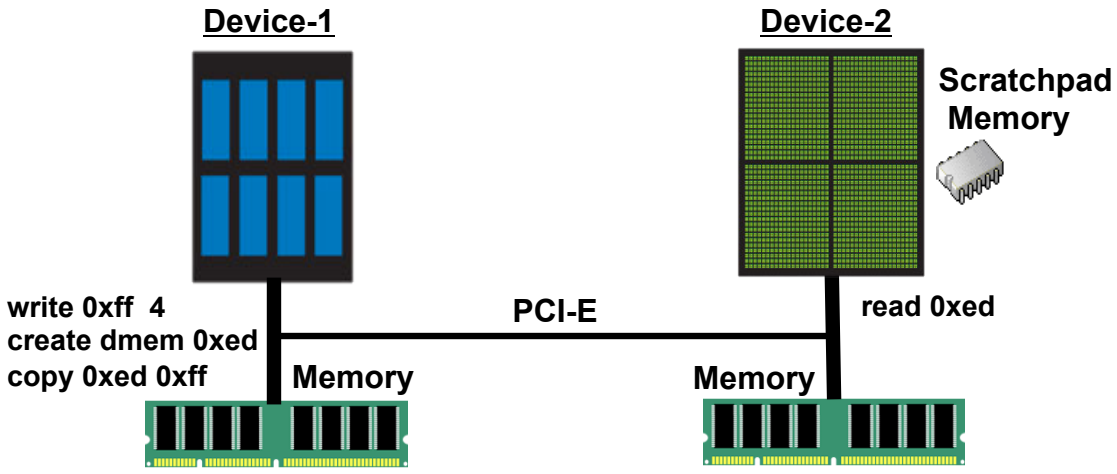


Figure 1.1 : Non-coherent memory + Non-shared virtual memory

the memory sub-system is termed a Coherent Memory (otherwise Non-coherent Memory).

We now describe the classification of heterogeneous architectures in more detail. Figure 1.1 depicts the first category of heterogeneous systems where each device has its own physical memory, and the underlying memory subsystem is not coherent and the processors do not share a virtual address space. To communicate data, the two devices must perform a series of commands that include: *creating buffers* on remote memory locations, *copying data*, and *a mechanism to convert the virtual address mapping*. In this scenario, for Device-1 to communicate a value `4` to Device-2, it must first write the value to a local memory location say `0xff`. It has to then create a memory location `0xed` on the Device-2 memory, and then copy the memory location `0xff` to `0xed`. Device-2 can now read the value `4` from its local memory `0xed`. Most generic host+accelerator systems belong to this category including Intel CPU + Discrete AMD/NVIDIA GPU.

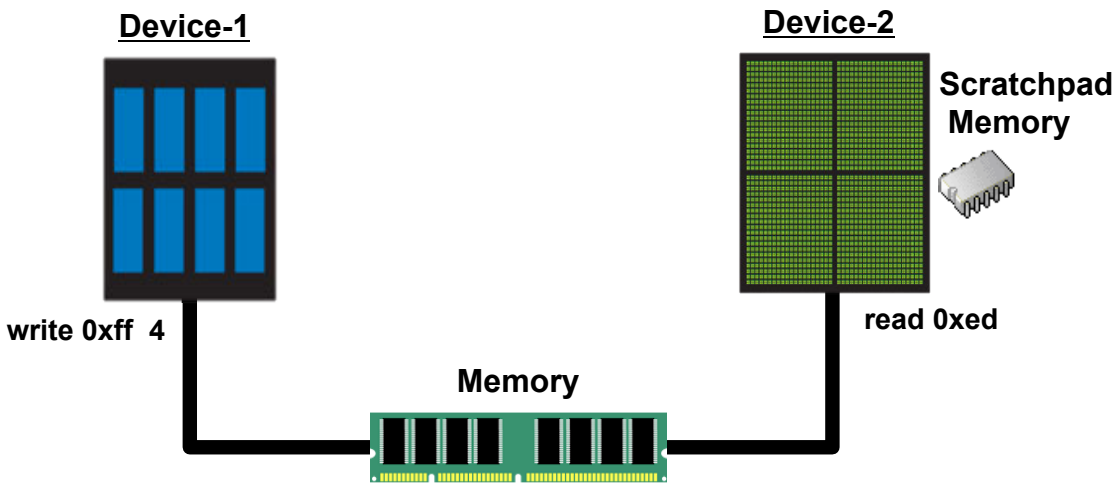


Figure 1.2 : Coherent memory + Non-shared virtual memory

Figure 1.2 depicts the second category of heterogeneous systems where both the devices share a single physical memory, and the underlying memory subsystem is coherent and the processors do not share a virtual address space. In this scenario, for Device-1 to communicate a value `4` to Device-2, it must first write the value to a virtual memory location say `0xff`. A mechanism such as a system driver API is used to find the virtual address mapping of `0xff` on Device-2 say `0xed`. Device-2 can now read the value `4` from its virtual memory location `0xed`. To communicate data, the programmer is responsible to map a virtual address from one device to another. Examples of these systems include Intel Ivy-Bridge (CPU + Integrated GPU), TI-KeystoneII (CPU + DSP).

Figure 1.3 depicts the third category of heterogeneous systems where all the devices share a single physical memory, and the underlying memory subsystem is fully-coherent and the processors share a virtual address space. In this scenario, for Device-1 to communicate a value `4` to Device-2, it must first write the value to a virtual memory location say `0xff`. Device-2 can now read the value `4` from the same

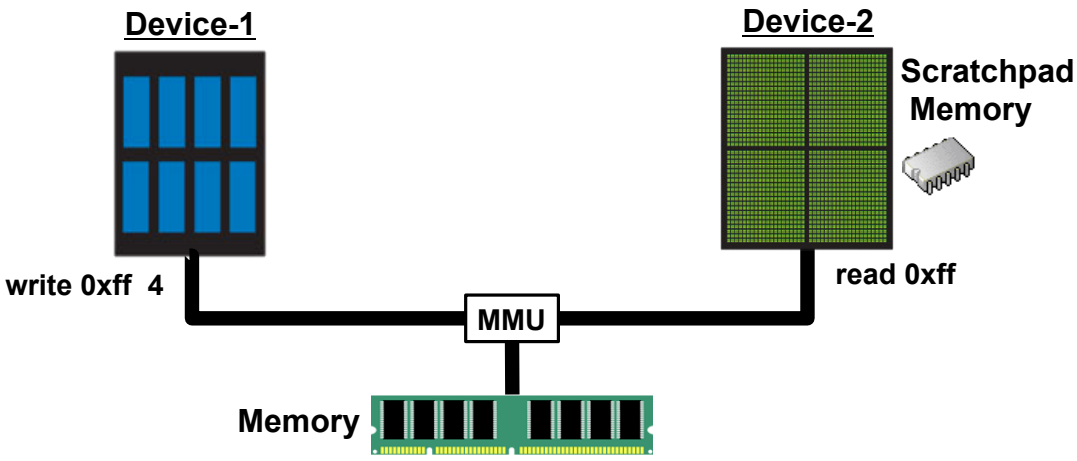


Figure 1.3 : Coherent memory + Shared virtual memory

virtual memory location `0xff`. These systems are easy to program since they do not require any explicit data movement or virtual memory mapping. However, the drawback of these systems is that they incur a significant cost for managing coherency, which has energy implications. Examples of systems include Intel Broadwell, AMD HSA, Multicore processors.

Figure 1.4 is similar to the third category of heterogeneous systems where each device has its own physical memory, and the underlying memory subsystem is coherent and the processors share a virtual address space. However, the coherence and sharing of virtual memory is achieved with the help of drivers provided by the vendor. In this scenario, for Device-1 to communicate a value `4` to Device-2, it must first write the value to a virtual memory location say `0xff`. Device-2 can now read the value `4` from the same virtual memory location `0xff`. The device driver automatically handles the memory coherence and virtual memory management. Examples of these systems include NVIDIA CUDA Unified Virtual Addressing (UVA). The implementation of UVA from NVIDIA is not publicly available. However, similar systems have been

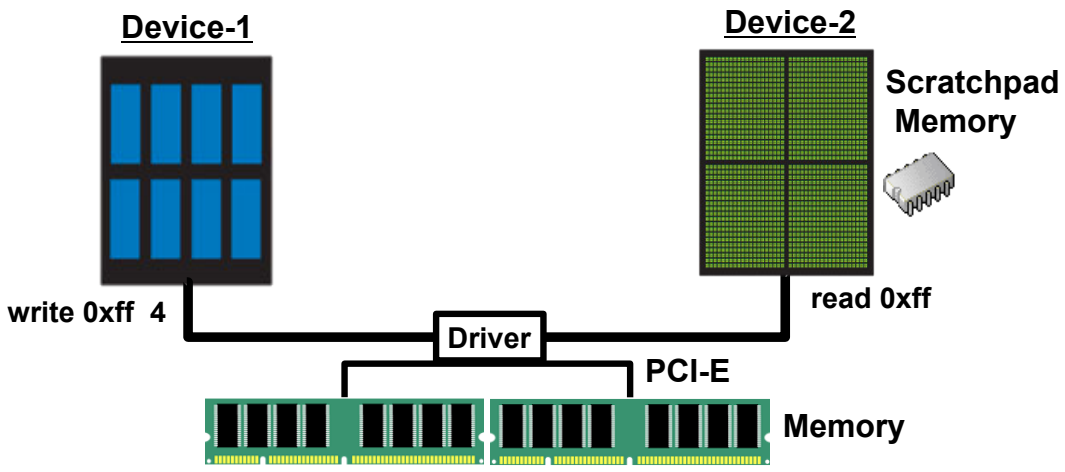


Figure 1.4 : Driver managed: Coherent memory + Shared virtual memory

implemented in literature [14] [15].

From the above description, it is evident that today's heterogeneous architectures differ in their architectural features. The memory hierarchy and cache structures are very different on these heterogeneous devices. With such diverse characteristics, it is not only hard to program these systems in a portable manner, but also very challenging to optimize them. The implication now is that both current and future software must run on these newer heterogeneous hardware. At the same time, the current standards of productivity, performance and portability must be met. A key challenge is that existing programming languages used to develop software applications are not able to utilize the full potential of these newer and faster processors. Consequently, application programmers have to deal with low-level programming languages; furthermore, these languages involve non-trivial learning and training. Extensive training has always been a barrier to the adoption of any new language. Furthermore, legacy software applications as well as libraries need re-targeting for these newer hardware causing a portability challenge.

Execution Mode	**Time (sec/step)**	**× Power = Energy (J/Step)**	**MFLOPS/Watt**
Single Core CPU	2.40	× 20W = **48.0**	**100**
CPU OpenCL	1.30	× 30W = **39.0**	**123**
GPU OpenCL	0.87	× 13W = **11.3**	**425**

Table 1.2 : Energy and performance comparison of heterogeneous architectures

Heterogeneous architectures also provide an interesting trade-off with respect to energy. Application performance can be tuned either for execution time or energy consumption. To study the extent of energy savings, we perform an experiment on a standard laptop, which has an Intel(R) Core(TM)2 Duo CPU device running at 2530 MHz and a GeForce 9400M GPU device running at 1100 MHz. The energy measurements were performed using a watt-meter, which reports the power consumption of the whole system including CPU, GPU, memory and I/O subsystem. Precautions were taken to avoid extraneous readings. For instance, the battery was fully charged before performing the experiments.

We ran an NBody application with $15K$ bodies on three execution modes shown in Table 1.2. NBody is a version of the molecular dynamics which iterates over a time-stepping loop updating the position, acceleration and velocities of the bodies. We see that the single core CPU version runs for 2.4 seconds per time step and consumes an average of 20 watts of power (as observed on the watt meter) with a total energy consumption of 48 joules per time step. The NBody application for the given input computes a total of $20 \times 15K \times 15K$ floating point operations and this translates to 100 mega-flops per watt on the single core CPU. The CPU OpenCL (using both

the CPU cores) runs faster at 1.3 seconds, but at the same time consumes 30 watts of power on average with a total energy consumption of 39 joules per time step. This translates to a total of 123 mega-flops per watt on the CPU. Finally, the GPU OpenCL (using the GPU) not only runs faster at 0.87 seconds, but also consumes less power at an average of 13 watts with a total energy consumption of 11.3 joules per time step. This translates to a total of 425 mega-flops per watt on the GPU. For this application, GPU is best in terms of both energy consumed and execution time.

Energy consumption is particularly critical for embedded and mobile systems. Programming systems for heterogeneous architectures must also provide features to optimize energy consumption along with performance.

1.2 Challenges Programming Heterogeneous Architectures

Some of the key programming challenges facing today's heterogeneous hardware include task partitioning, data distribution and coherence, data layout and handling devices with non-shared virtual memory some of which are explained below.

Data Layout

Data layout refers to the storage pattern of data in main memory. There are many storage choices including row-major [16], column-major [16], array-of-structures and structure-of-arrays [17]. Data layout is critical for application performance. An improper layout can lead to poor cache utilization resulting in performance degradation. Data layout is a well-studied problem in the context of single and multi-core CPUs [17–20]. However, in a heterogeneous environment, data layout becomes challenging since different processors may expect a different layout. For instance, CPUs prefer array-of-structures layout since they can benefit from prefetching and spatial

locality. On the other hand, GPUs prefer a structure-of-arrays layout since they can benefit from coalescing of memory loads. Data layout also impacts mapping of tasks.

Task Partitioning and Data Management

Task partitioning involves mapping a set of tasks (in an application) onto the available heterogeneous processors. Each of these processors vary in number and the kind of computation units, memory hierarchy, and other hardware limitations present. Some challenges involved in mapping a task onto heterogeneous processors are listed below.

- Each task can run on all or a subset of the available processors.
- The performance of each task varies for each processor. The mapping is based on purely execution time, total energy consumed or a combination of both.
- Mapping influences data locality and data movement.
- Data layout and mapping are dependent.

Sophisticated tuning and heuristics are required to optimize all the above parameters and automatically map a given program. Auto-tuning approaches so far have not been very successful. Alternatively, these tasks can also be mapped adaptively at runtime. However, runtime techniques are limited since they have to make mapping decisions during program execution and evaluating all the parameters and restrictions incur considerable overhead. These challenges can be simplified by enabling the programmer to make high-level decisions about the mapping and let the compiler/runtime efficiently map the tasks onto the specified processor units.

Once the task mapping is specified, the data required to execute these tasks has to be distributed among the different memory locations. The challenge now is to deter-

mine the optimal distribution of data among the devices. Data transfer technologies such as PCI-E are used to move data from one device to another. However, these current technologies suffer from low bandwidth and high latency and are far behind the memory bandwidths within a device. This latency gap leads to an imbalance making data movement between devices very expensive. Thus, programmers have to minimize the data movement to get the maximum performance. Alternatively, the programmers can choose to overlap the data movement with computation to reduce the communication latency. In the presence of complex data access patterns, the amount of data that needs to be moved can be hard to determine. For large programs, the problems of task mapping, data layout, and data distribution must be handled together since decisions at one portion of the program might influence other portions of the programs.

Non-shared Virtual Memory and Memory Coherence

Other challenges in programming these heterogeneous architectures include devices that do not have the same virtual memory and managing the coherence of data across the memory hierarchy. The problem of non-shared virtual memory becomes severe when programs use data-structures that include pointers to data. Such applications are very common today and in order to execute them, the programmer has to translate these pointers to different virtual addresses for each device. The data coherence problem can occur at various levels in the memory hierarchy. Some heterogeneous processors like integrated CPU+GPU processors share the same physical memory but have semi-coherent caches. The programmer has to determine where in a program the data needs to be consistent and has to manually flush the data. The memory coherence can also occur in the DRAM memory across the devices. When the programmer

distributes data, it is possible that the data is duplicated across these devices. It is the responsibility of the programmer to maintain copies only when it is legal to do so.

1.3 Thesis Statement

Existing programming systems are not productive for targeting current heterogeneous architectures. The thesis of this dissertation is that minimal extensions to existing programming languages can yield programming models that target modern heterogeneous architectures with portability, productivity, and performance We establish this thesis by demonstrating extensions to C and C++ that enable a user to write productive machine-independent portable programs, from which the compiler and runtime maximize program performance and energy efficiency by generating executables tuned towards both multi-core CPUs and heterogeneous hardware.

Thesis Overview

Figure 1.5 shows the high-level overview of this dissertation, which explores extensions to existing programming languages such as C/C++ with minimal high-level constructs to target heterogeneous architectures. These extensions should be suitable to target both current and future applications. However, these extensions should meet existing standards of productivity, portability and performance, and at the same time handle some of the challenges including data layout, task partitioning, data distribution, event management, exploiting hardware specific resources and virtual memory sharing which modern heterogeneous architectures pose.

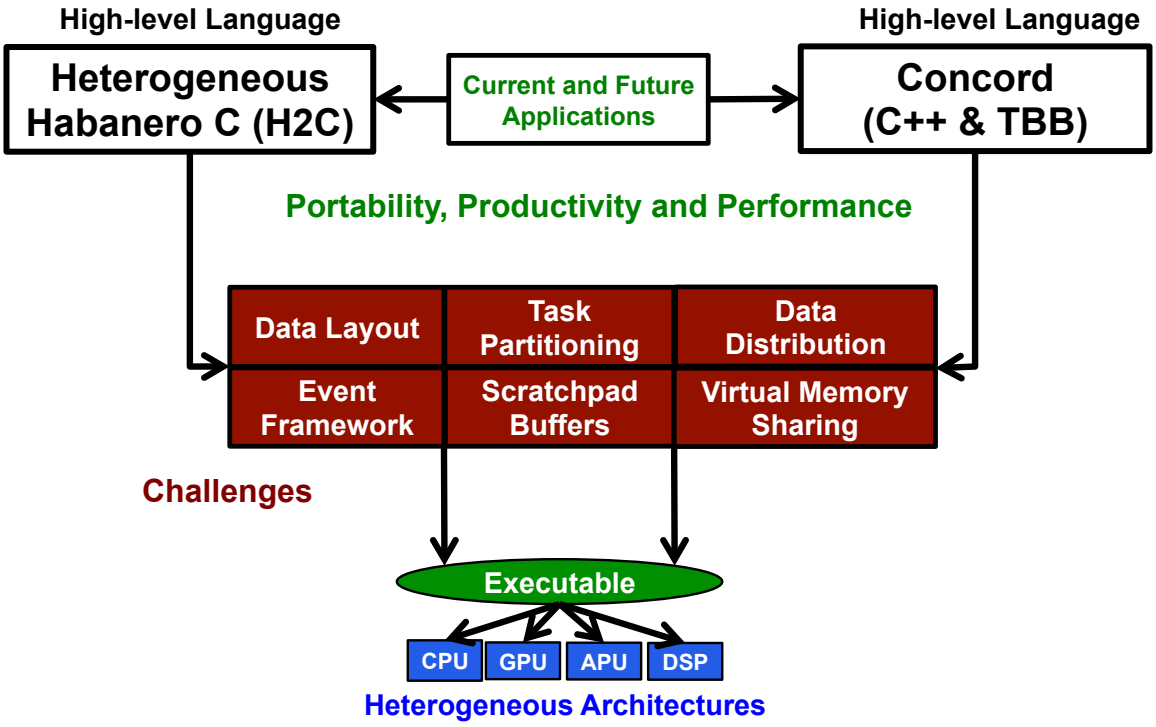


Figure 1.5 : Thesis overview

1.4 Thesis Contributions

This dissertation, makes the following contributions:

Concord

Integrated CPU+GPU heterogeneous processors became pervasive in the last decade. Intel introduced heterogeneous systems starting with the Sandy-Bridge series and are now widely available on most desktops and server computers. The GPU occupies a significant portion of the die and is also power efficient. However, it is very challenging to take advantage of the GPU for computation due to restrictions on its programmability and high development effort. The goal of *Concord* is to take advantage of the GPU

seamlessly without any additional programming effort. We design and implement a C++ based programming model (aka *Concord*) for integrated CPU+GPU heterogeneous architectures based on the Intel Threading Building Block [21] library interface. *Concord* is open-source and is available at https://github.com/IntelLabs/iHRC. It is in use by various product groups at Intel, and in various academic and research institutions.

Challenges

Processors such as GPUs pose restrictions on programmability. For instance, both the CPU cores and GPUs can access the same physical memory, but some GPUs lack hardware and OS support for a shared virtual memory across the CPU+GPU cores. The lack of shared virtual memory restricts execution of applications with recursive data structures and pointers on the GPU. Traditional approaches use techniques that incur high-overheads such as serialization and un-serialization [22] of the data structures to overcome this limitation. Further, GPUs do not support function calls. Due to these limitations, supporting C++ virtual functions on a GPU becomes challenging.

Highlights of Concord

The key contribution of *Concord* is to efficiently implement Shared Virtual Memory (SVM) in software to enable traversals of the same pointer-based data structure from the CPU and the GPU. We implement C++ virtual functions on the GPU. We use a "de-virtualization + inline" scheme to support virtual functions. De-virtualization on GPUs is complicated because, it requires copying the virtual tables to the SVM regions supported by *Concord* and make the corresponding code changes inside the

GPU kernel. We also provide an efficient scheme to implement parallel reductions on a GPU from a high-level language. Finally, we port many regular and irregular C++ CPU applications to *Concord* and report their performance on these integrated CPU+GPU devices.

Heterogeneous Habanero-C (H2C)

The philosophy of *H2C* is to provide a machine-independent programming model on today's diverse heterogeneous hardware. The idea is similar to High Performance Fortran (HPF) [23], which was introduced in the early 90s with a goal of supporting a single program, multiple target programming system, for distributed cluster machines. HPF introduced many high-level constructs including FORALL, data alignment, and data distributions. A single HPF program can compile to any distributed cluster. However, to get good performance, the programmer might have to tune the source code for a specific target cluster.

Challenges

Todays heterogeneous processors on a single node pose software challenges analogous to those of the distributed cluster machines. Some of these challenges include data layout and coherence, "heterogeneous" data distribution and task partitioning, and point-to-point synchronization across heterogeneous devices. Also, these architectures are diverse. One has to generate different versions of the program for each device.

H2C Programming Model

Analogous to HPF, the goal of *H2C* is to provide a machine-independent programming model for today's heterogeneous architectures. *H2C* is an extension of the Habanero-

C [24] programming model with a target of achieving productivity and performance portability on these devices. The programming model of *H2C* is a mix of task-based programming model [24] (across devices) and SPMD [25] (within a device). Currently, *H2C* is implemented to support heterogeneous devices present on a single node. Extensions to support a distributed heterogeneous cluster are discussed in the future work section.

OpenCL [26] is provided by many vendors today to program heterogeneous devices in a portable manner. However, OpenCL is too low-level for easy adoption. *H2C* combines the high-level language features of the Habanero model with the ubiquity of OpenCL. The result is that *H2C* can be used to program a variety of heterogeneous devices including CPU+GPU integrated and discrete devices, DSPs, and even FPGAs in some cases.

The data layout framework pushes *H2C* a step further with extensions to manage the data layout of a program in a portable manner. Our meta-data layout framework allows programmers to specify different data layouts for different devices. The automatic data layout framework in *H2C* extends the data layout formulation in HPF [16]. The extensions include mapping of tasks onto heterogeneous devices and handling data duplication on devices with local memories. A key novelty of the automatic data layout approach in *H2C* is that it unifies the two problems of task mapping and data layout into a single problem.

H2C provides partitioning constructs to specify "heterogeneous" task partitions and data distributions. Our implementation takes advantage of a polyhedral [27] framework and explicit parallel semantics of *H2C* constructs to efficiently implement task partitions and data distributions. These partitioning constructs highlight another novelty of *H2C*, which is a first step to enable a PGAS [28] like programming

model for heterogeneous systems with discrete memories.

Apart from flat synchronization offered by "forasync-finish" constructs, *H2C* allows implementation of point-to-point dependencies across multiple devices using the "forasync-await" constructs. The most efficient way to manage events on top of heterogeneous architectures is to use the "event" implementation provided by the vendor OpenCL library. However, these events created are only limited to a single device. *H2C* overcomes this limitation with the help of a novel and efficient light-weight Unified Event (UE) framework created as part of this work. UE consists of a combination of compiler analysis and runtime implementation to manage dependencies across devices. The novelty of UE is that it enables macro-dataflow [29] programs to run unchanged on heterogeneous devices.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 describes the *Concord* programming model which is a C++ TBB extension that targets heterogeneous processors, which do not share the same virtual addressing. Chapter 3 describes Heterogeneous Habanero-C(*H2C*) that is an extension of the Habanero model for heterogeneous architectures. Chapter 4 describes the data layout framework implemented in *H2C*. Chapter 5 describes the related work and finally, Chapter 6 summarizes our conclusions and describes future extensions to this work.

Chapter 2

Concord Programming Model

2.1 Introduction

Today's heterogeneous systems are tightly coupled or loosely coupled that is they may not share the virtual address to physical address mappings. Examples of loosely coupled heterogeneous architectures include integrated CPU+GPU and CPU+DSP architectures. In these architectures, the CPU and the accelerator (GPU/DSP) share the same physical memory but use different virtual addresses to access the physical memory (Classification 2 in Table 1.1). Loosely coupled systems tend to offer more benefits in terms of energy consumption due to lack of full coherency (including sharing page tables) mechanism. Maintaining full CPU-GPU coherency increases processor complexity and die area, and it is unrealistic to expect all GPUs, including low-end GPUs for mobile devices, to support page table sharing in hardware. Example: Intel Bay Trail tablet.

However, lack of virtual memory page sharing restricts the programmability since applications with recursive data structures (data pointers) cannot be executed easily across different processors on the same system. The application developer has to serialize and de-serialize those data structures with pointers to overcome this limitation. This is also a source of portability problems since the programmer has to maintain two versions of the program: one for architectures where the virtual memory is shared and another for those it isn't. Hence, if the programming framework can handle the

virtual memory sharing automatically, one can implement applications in a portable manner, thereby improving programmer productivity.

The interest to support virtual memory sharing on such architectures is also fueled by the ubiquity of *integrated CPU+GPU* processors from major hardware vendors such as Intel and AMD. These processors integrate a CPU and GPU onto the same die where they share resources like physical memory and the last-level cache. The advantage of integrated GPUs is that they benefit from low-latency communication and eliminate data copying, which significantly lowers the cost of offloading work to the GPU. However, integrated GPUs are limited by the power and size budget allocated for the integrated processor.

One way to reduce the complexity of GPU programming is to use the same data-parallel programming models that are already used for programming multi-core CPUs. The question, though, remains whether benefits of GPU execution can be extended to irregular applications written in an object-oriented programming style that features object references, virtual functions, and functor-based parallel constructs.

In this chapter, we describe *Concord*, a heterogeneous C++ programming framework for processors with integrated GPUs designed to allow general-purpose, object-oriented, data-parallel programs to take advantage of GPU execution. *Concord* supports most C++ features, including namespaces, templates, multiple inheritance, operator and function overloading, as well as virtual functions. It supports two parallel constructs for offloading computation to the GPU: a *parallel-for* loop and a *parallel-reduce* loop. These constructs are modeled after ones provided by Intel's Threading Building Blocks (TBB) [21], and are similar to those provided by other CPU parallelism frameworks such as OpenMP, TPL [30], and Cilk [31]. Most importantly, *Concord* supports seamless sharing of data between the CPU and GPU via an

efficient software implementation of shared virtual memory (SVM) augmented with compiler optimizations to reduce the overhead of shared pointer translations. SVM enables programs to directly share pointer-containing data structures between the CPU and GPU. Since object-oriented programs make heavy use of objects that point to other objects, SVM is a prerequisite for GPU execution of object-oriented C++ programs. Our SVM solution is implemented purely in software and targets integrated GPUs with no virtual pages shared between CPU and GPU such as processors readily available today from Intel and AMD.

We evaluate *Concord* using seventeen realistic regular and irregular C++ applications running on two computer systems with Intel 4^{th} Generation Core processors. Some of these applications are pointer-intensive as they operate on irregular data structures (trees and graphs) represented in the traditional C/C++ fashion using pointers. *Concord* is now an open source project [32] and has been used by many researchers including the Galois group [33] at UT-Austin to evaluate their irregular applications.

The rest of this chapter is organized as follows. Section 2.2 presents some background for the the *Concord* programming model. Section 2.3 presents the *Concord* programming language constructs and restrictions. Section 2.4 then describes the details of our prototype implementation. Sections 2.5 provides experimental results.

2.2 Background

In this section, we briefly summarize important frameworks that are used to implement the *Concord* programming model.

2.2.1 OpenCL

OpenCL [26] is an open standard to program modern heterogeneous hardware. An OpenCL implementation provides a low-level API to compile, execute and also map a program on a heterogeneous architecture. The API also provides constructs to specify asynchronous computations and communication along with synchronization. OpenCL follows the *offload* model where the main program is executed on a "host" which launches tasks onto "devices". Many vendors today including Intel(CPU/CPU/Xeon Phi), AMD(CPU/GPU/APU), NVIDIA(GPU), Texas Instruments(CPU/DSP), Xilinx(FPGA) and Altera(FPGA) provide implementations of OpenCL to program their hardware. OpenCL is increasingly being adopted by various developers to write applications for current heterogeneous hardware. However, OpenCL is challenging for average programmers to learn, thereby limiting its rate of adoption onto newer architectures.

2.2.2 LLVM/Clang

LLVM [34] is an open source compiler tool-chain designed to provide modern static and dynamic compilation strategies. A source language is compiled down to an SSA based intermediate representation called "LLVM byte-code". All optimizations are performed on this byte-code. This byte-code is further lowered down by a back-end to target specific assembly/binary code. Clang is the front-end parser module of LLVM. LLVM has recently gained a lot of popularity due to its modular and reusable features and is widely adopted by industry and academic institutions.

2.3 Programming Model

Concord supports most C++ features with some exceptions. It provides two API functions for data-parallel iterations and reductions and has SVM support that enables programs to transparently share pointer-containing data structures.

2.3.1 Programming Constructs

Concord's template API functions for data-parallel computation are modeled after the corresponding ones in Intel Threading Building Blocks (TBB).

```
template <class Body>
void parallel_for_hetero(int n, const Body &b, bool on_GPU);
template <class Body>
void parallel_reduce_hetero(int n, const Body &b, bool on_GPU);
```

Both template functions take a parameter `n` that specifies the iteration space, $[0..n)$ to be done in parallel. For both functions, the second parameter `b` must be an instance of a class `Body` that defines a method `void operator(int i)` specifying the body of the parallel loop or reduction. The third parameter controls whether execution should be on the CPU or GPU. For `parallel_reduce_hetero`, the `Body` class must define an additional method `join` to combine the results for two `Body` objects. The programming model ensures mutual exclusion for the `join` method.

Concord does not guarantee that different loop iterations will be executed in parallel. Also, as in TBB, programmers should make no assumption about the order in which different iterations are done. Similarly, floating point determinism in reductions is not guaranteed.

```
1  class LoopBody {
2      Node * nodes;  // array of nodes
3      public:
4      LoopBody(Node *arr) : nodes(arr) {}
5      void operator()(int i) { // executed in parallel
6          nodes[i].next = &(nodes[i+1]);
7      }
8  };
9  void convertToLinkedListFromArray(Node * array, int N) {
10     LoopBody *b = new LoopBody(array);
11     parallel_for_hetero(N, *b, GPU);
12 }
```

Figure 2.1 : Concord program to convert an array of node objects to a linked list in parallel.

An example showing the use of `parallel_for_hetero` appears in Figure 2.1. This example illustrates how it might be used to convert an array of pointers to a singly-linked list data structure in parallel. The main kernel is written in the `operator()` of `class LoopBody`. An instance of this class is passed as an argument to the `parallel_for_hetero` along with the number of iterations N and the target device (`GPU`).

To illustrate the use of `parallel_reduce_hetero`, Figure 2.2 shows how it can be used to compute the sum over the result of applying a function to each element of an array. The `operator()` computes the result of the function applied to each array index $A[i]$ and the `join()` reduces the result in parallel.

```
1  class Body {
2      float *A, result;
3      public:
4      Body(float *aa): A(aa), result(0.0f) { }
5      void operator()(int i) {  // executed in parallel
6          result = f(A[i]);     // compute local result
7      }
8      void join(Body &rhs) {
9          result += rhs.result; // reduction: sum results
10     }
11 };
12 ...
13 Body *body = new Body(A);
14 parallel_reduce_hetero(vector_size, *body, GPU);
```

Figure 2.2 : parallel_reduce_hetero example

2.3.2 Shared Virtual Memory(SVM) Support

In order to make existing C++ programs portable on integrated processor with non-shared virtual memory, *Concord* provides SVM. This allows programs running on the CPU and GPU to directly share complex, pointer-containing data structures such as trees and linked lists. SVM also eliminates the need to marshal data between the CPU and GPU.

2.3.3 Support for C++

Concord supports most C++ features in the GPU code including classes, virtual functions, multiple inheritance, operator and function overloading, templates, and namespaces. However, due to compiler and GPU hardware limitations, there are restrictions to its C++ support, violations of which result in compile-time warnings and `parallel_for_hetero` or `parallel_reduce_hetero` code being executed on the CPU. In particular, *Concord* does not support recursion (except for tail-recursion that can be eliminated at the compile time), function calls via a function pointer, taking the address of a local variable, memory allocation on GPU, and exceptions. We plan to lift the last two restrictions as part of the future work. Note that although *Concord* does not support function calls via a function pointer, it supports virtual and externally defined functions.

2.4 Implementation

Figure 2.3 depicts the components of our *Concord* framework along with their interaction with other components. We use the Clang and LLVM infrastructure to compile *Concord* C++ programs. A compiler pass identifies the heterogeneous loop body functions (*i.e.,* the `operator()` and `join` methods of a body class) and generates CPU code as well as GPU OpenCL kernel code for them. We generate a host-side executable that embeds the generated OpenCL. Later, to execute a heterogeneous loop, the runtime extracts its OpenCL code, just-in-time compiles it to GPU ISA if necessary via the vendor-specific OpenCL compiler, and then, based on the `on_GPU` flag, decides whether to execute it on the CPU or GPU.

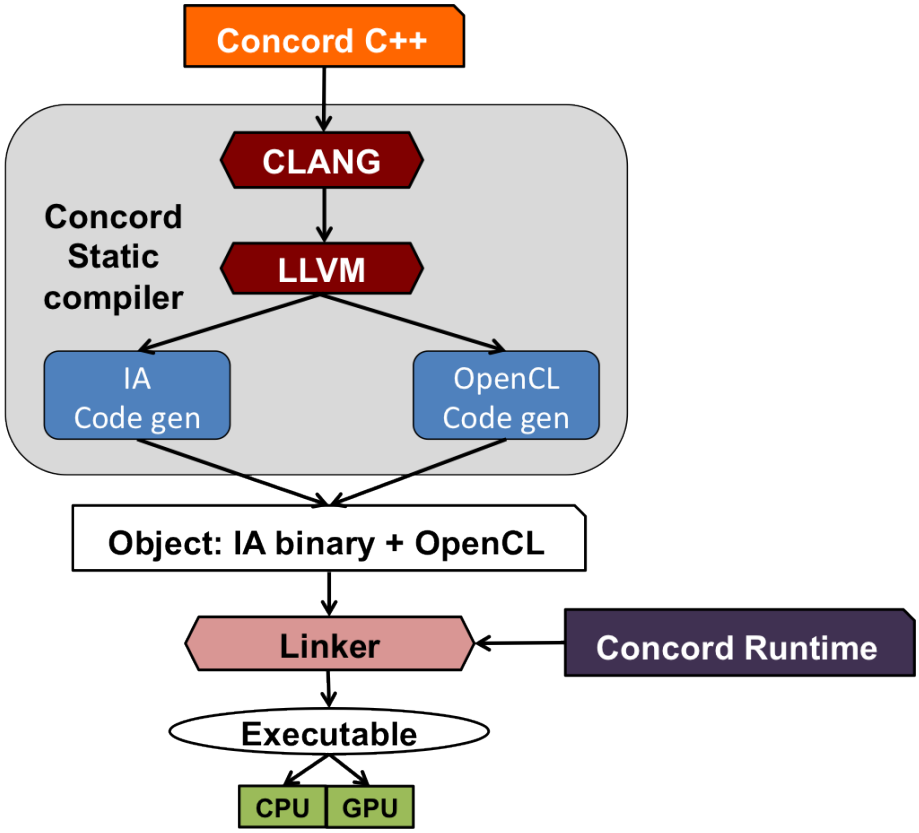


Figure 2.3 : Overall flow diagram of the Concord framework

2.4.1 CPU-GPU Shared Pointers (SVM)

Concord's SVM support allows the GPU to share the same pointers as the CPU. *Concord* represents a shared pointer using the CPU virtual memory address, and the compiler generates code to translate virtual addresses on the GPU at runtime. The challenge of implementing this translation is that the CPU and GPU may have separate virtual-to-physical mappings and different pointer representations. These details differ greatly from one processor architecture to the next. The remainder of this section describes our implementation on Intel's 4^{th} Generation Core processor.

On this processor, the GPU and CPU use separate page tables. The GPU's virtual

address space is segmented into *surfaces* and each surface is referenced by a binding table entry. A GPU pointer is represented as a binding table index plus an offset. To access memory, the offset is added to the surface's base address obtained by looking up that surface's binding table entry. Thus, when we dereference a shared pointer on the GPU, we must translate that CPU virtual address so that it refers to the same physical memory location on both GPU and CPU.

To do this translation, we create a virtual memory region at program startup that is shared between the CPU and GPU*. Any shared pointer that the GPU needs to dereference must be allocated in this shared memory region. We achieve this by redirecting malloc and free to specialized routines that allocate and free memory in the shared memory region. The shared memory region is pinned during GPU kernel execution and has a backing GPU surface with a binding table entry that is constant during runtime. This approach substantially reduces the cost of *Concord*'s shared pointer translation.

Figure 2.4 depicts the compiler transformation necessary to synchronize the virtual addresses of shared pointers between CPU and GPU. Given the base addresses of CPU and GPU for the shared region as *cpu_base* and *gpu_base* respectively, a pointer *ptr_p* in the CPU virtual address space has a corresponding GPU virtual address *gpu_ptr_p* where

$$gpu\_ptr\_p = gpu\_base + (ptr\_p - cpu\_base)$$

. This address translation can be optimized by using the runtime constant

$$svm\_const = gpu\_base - cpu\_base$$

*On Intel's 4^{th} Generation Core processor, all *physical* memory is shared between CPU and GPU.

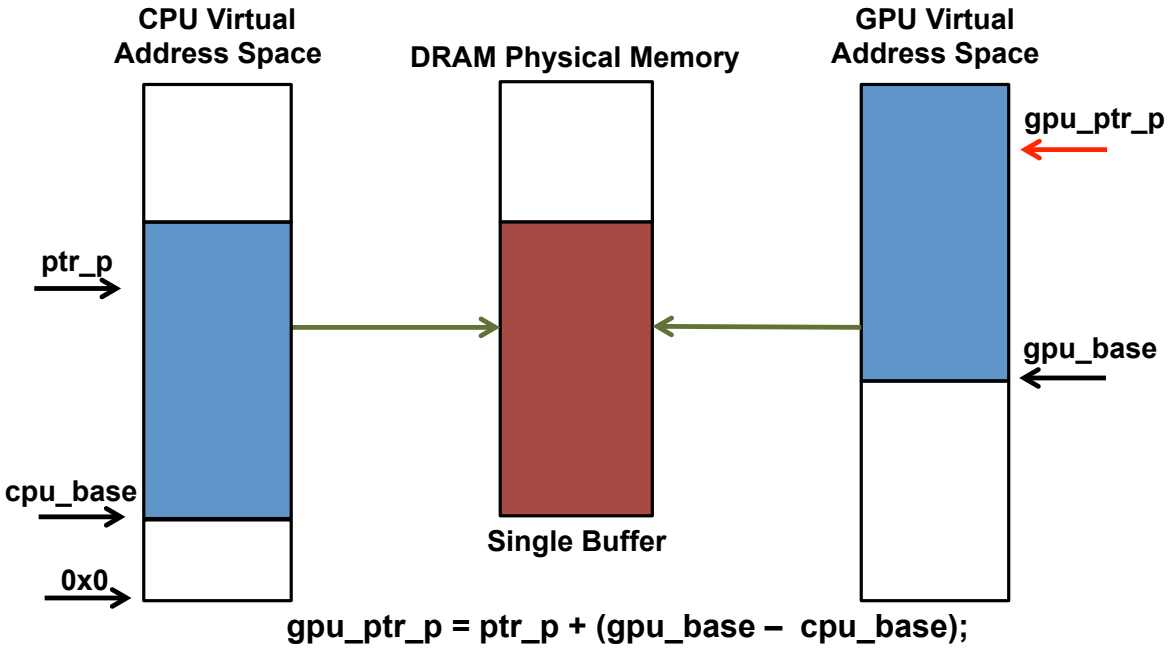


Figure 2.4 : CPU and GPU shared pointer transformation

that is computed only once. Then, before dereferencing *ptr_p* on the GPU, it can be translated to *gpu_ptr_p* by simply adding the runtime constant *svm_const*.

Figure 2.5 presents the compiler generated OpenCL code for the `operator(int i)` method in Figure 2.1 using the pointer transformation described in this section. The OpenCL kernel `offload` takes additional arguments for *gpu_base*, *cpu_base*, and the pointer *cpu_ptr* to the `Body` object (which is same as *b* in the source program). The shared pointers, *cpu_ptr* and *gpu_ptr_p*[*i*].*next* are translated from the CPU address space to the GPU address space using the `GPU_PTR` macro.

Our pointer translation technique can be generalized to scenarios where CPU and GPU use different encoding schemes and lengths. For example, if CPU memory is addressed using 64-bits and GPU memory uses 32-bits, we can apply the same pointer arithmetic as long as the shared region does not exceed 4GB.

```
1  typedef unsigned long CpuPtr;
2  #define GPU_PTR(T,p) (__global T *)(&svm_const[(p)])
3  #define CPU_PTR(T,p) (__global T *)(&svm_rconst[(p)])
4  __kernel void offload(__global char *gpu_base,
5                                  CpuPtr cpu_base, CpuPtr cpu_ptr) {
6            uint i = get_global_id(0);
7          __global char *svm_const = (gpu_base - cpu_base);
8          __global char *svm_rconst = (cpu_base - gpu_base);
9          __global Node *gpu_ptr = GPU_PTR(Node, cpu_ptr);
10          *(GPU_PTR(Node, gpu_ptr[i].next)) = &gpu_ptr[i+1];
11 }
```

Figure 2.5 : OpenCL generated by Concord compiler for `operator()`.

In this implementation, we restrict the SVM framework to a single buffer to reduce the overhead of SVM. Multiple buffers can be handled using a scheme similar to cache associativity implementations, where the bits of the original address can be used to identify the buffer. Once the buffer has been identified, the final address can be computed using the offset and base address corresponding to the identified buffer.

2.4.2 Virtual Functions

One of the most widely used dynamic features of C++ is its virtual function support. Although there are a variety of different ways to implement virtual functions, the *vtable* (virtual table) approach is common in modern C++ compilers. In this approach, a compiler creates a separate vtable for each class and when creating an instance of that class (an object), adds to that object a pointer to the class's vtable. A call to a virtual function is then handled by dereferencing the underlying runtime

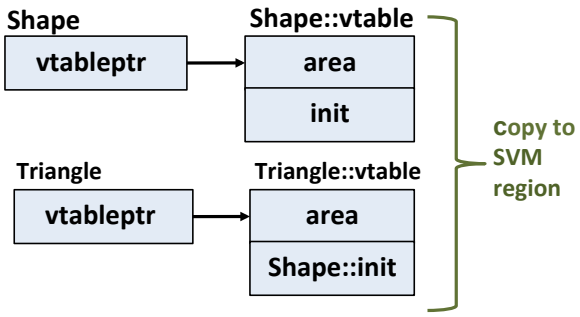
object's vtable pointer, locating the corresponding virtual function entry and finally dereferencing that pointer to call the function. To implement virtual functions on the GPU, vtables need to be allocated in the shared region and more importantly, function pointers are required on the GPU. Current integrated GPU hardware designs are not yet capable of supporting function pointers, so we use a compiler-based solution.

To support virtual functions on the GPU, the *Concord* compiler implements three key operations: a) move necessary vtables and runtime-type information to the shared region; b) share the global symbols of relevant virtual functions between the CPU and GPU using shared memory; c) translate a virtual function call into an inline sequence of tests of the call target against the possible target function pointer values for that call. The compiler implements global symbol sharing between CPU and GPU by allocating a new structure in the shared memory region that encapsulates all global symbols needed for the virtual function calls executed by a GPU function. It also determines the set of call targets for a given virtual function using class hierarchy analysis and alias analysis.

Figure 2.6 shows the implementation of virtual functions in *Concord*. Figure 2.6(a) shows a C++ program with virtual function *area()* (called on line 12). Figure 2.6(b) shows the virtual table layout generated by the compiler. These virtual table data structures have to be copied to the SVM region. Figure 2.6(c) shows the code generated for the *area()* virtual call. The code generated for the CPU virtual call (lines 2-3) is the standard function pointer call. However, the GPU virtual call (lines 5-8) is implemented by de-virtualizing. Essentially, all possible targets of a virtual function call are identified and a jump table is built for these targets with the corresponding functions in-lined.

```
1  // original hierarchy
2  class Shape {
3    virtual void init();
4    virtual int area();
5  };
6  class Triangle:Shape {
7    virtual int area();
8  };
9
10 int foo(){
11 //virtual function call
12      Shape *s;
13      int val;
14      val = s->area();
15      return val;
16 }
```

(a) C++ Program with Virtual Functions



(b) Object Layout with Virtual Tables

```
1 Shape *s;
2 int val;
3 //CPU Virtual Function call:
4 (s->vtableptr[1])();
5 //GPU Virtual Function call:
6 if (s->vtableptr[1] == Ctx->Shape::area)
7      val = Shape::area(); //inline
8 else
9      val =  Triangle:area(); //inline
```

(c) Generated code

Figure 2.6 : Example showing handling of virtual functions by Concord

2.4.3 Reduction

When using `parallel_reduce_hetero`, the `Body` object's `join` method contains reduction code that combines two `Body` objects. We modified our compiler and runtime to perform hierarchical reduction of the body objects on the GPU using *local memory*, the high-speed on-GPU memory that is shared among all work-items of a work-group in OpenCL.

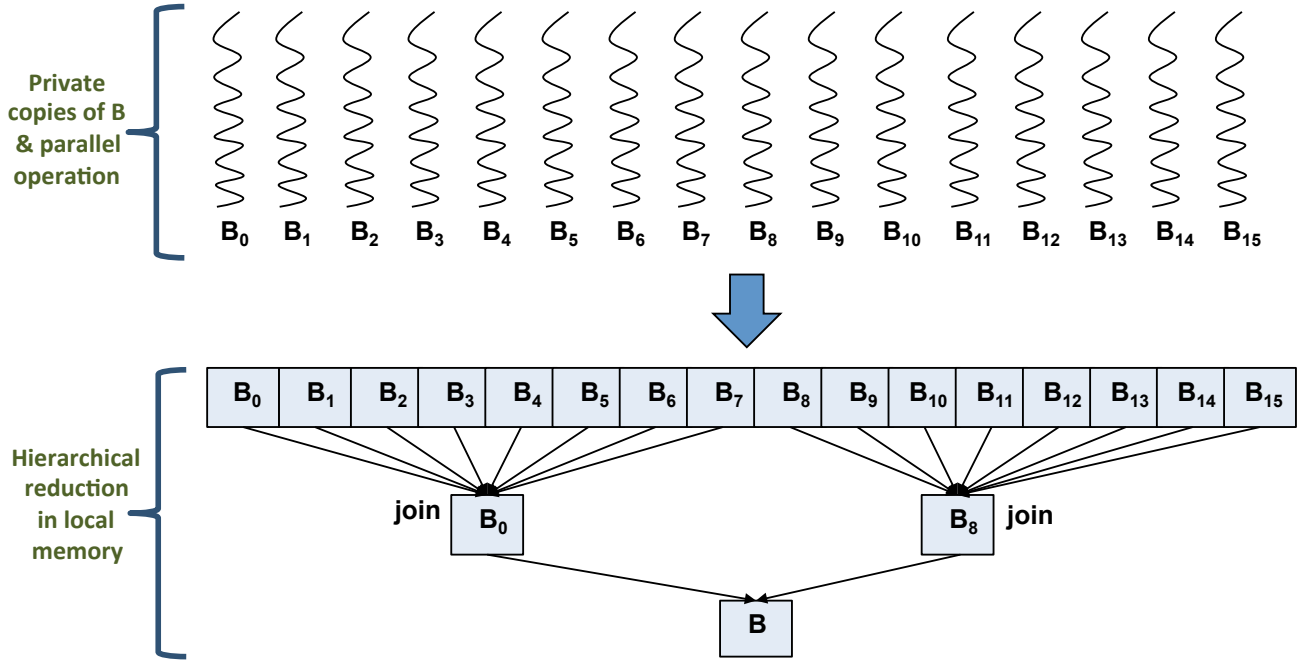


Figure 2.7 : Reduction on GPU: private reduction followed by hierarchical reduction in local memory

The compiler generates OpenCL code for the `join` method similar to the code generation technique for `operator()`. We generate additional wrapper OpenCL code that makes multiple copies of the shared `Body` object in each thread's private memory, invokes the `operator()` function to compute the thread's value that participates in reduction, moves the private objects to local memory, and finally, iteratively performs reduction using local memory until a single value is left. The local memory copies hold intermediate reduction results. The final reduced value is copied back to the original shared `Body` object. Figure 2.7 describes this process. The original sequential `join` function pointer is also passed to the runtime to perform sequential reduction if local memory is insufficient or if the GPU is busy.

2.4.4 Code Generation

The *Concord* compiler translates `parallel_for_hetero` and `parallel_reduce_hetero` to the runtime API functions `offload` and `offload_reduce` respectively. These runtime functions take additional compiler-generated arguments: (1) a `gpu_program_t` structure for the entire program to hold the OpenCL code and its cached JIT-compiled GPU binary; (2) a `gpu_function_t` structure to cache per-function GPU binary code in order to reuse the JIT-compiled code. The `gpu_function_t` also carries the user-specified device information per kernel as specified in the third argument of `parallel_for_hetero` and `parallel_reduce_hetero`.

Concord compiler performs standard compiler optimization techniques like loop-unrolling and scratchpad memory optimizations. Apart for these optimizations, we devise another optimization in *Concord* to reduce the S/W-based SVM implementation overheads. This optimization is described in detail below.

2.4.5 Reducing SVM Implementation Overhead

The pointer arithmetic operations inserted as described in Section 2.4.1 must be minimized by the compiler whenever possible. Depending on how shared pointers are used on the GPU, it may be beneficial to retain the CPU virtual address representation for a shared pointer instead of eagerly translating it to GPU address space. For example, if the GPU code loads a shared pointer and stores it into a memory location without dereferencing it, then it is better never to convert the CPU virtual address. Hence, there are some situations when it is better to translate eagerly CPU to GPU addresses, and other situations when lazy translation is better. For example, consider the code sample shown in Figure 2.8.

```
//Pointer conversion required on GPU
int **a = data->a, **b = data->b;
for(int i=0; i<N; i++)
    //Pointer conversion required on GPU
        b[i] = a[i];
    // a is not used on GPU after this
```

Figure 2.8 : Illustration of **lazy** vs. **eager** compiler transformation of shared pointers

In this code fragment, pointer $a[i]$ is loaded from memory and written into $b[i]$ at each iteration of the loop. With eager translation (*i.e.,* convert to GPU virtual memory representation as soon as the pointer is loaded), we need pointer arithmetic operations to translate the array addresses a and b only immediately after their definitions, which are outside the `for`-loop.

Using lazy translation (*i.e.,* keep the CPU virtual memory representation as is and translate to GPU representation just before dereferencing it), we must add pointer arithmetic to translate a and b from the CPU to the GPU representation on every loop iteration. The eager approach is clearly beneficial in this case.

On the other hand, eagerly converting the address of an array element $a[i]$ to a GPU virtual address results in wasted work because $a[i]$ is never dereferenced on the GPU. It would convert all $a[i]$ pointers to GPU addresses only to immediately convert them back to CPU addresses in order to store them in array b. The lazy approach is preferable in this case.

Both eager and lazy approaches have their advantages and disadvantages and can

```
1 int **a = data->a;
2 for ( int i=0; i<N; i++)
3     = GPU_PTR(int *, a)[i];
4 //Overhead: N
```

(a) Lazy Translation

```
1 int **a = GPU_PTR(int *, data->a);
2 for ( int i=0; i<N; i++)
3     = CPU_PTR(int, GPU_PTR(int, a[i]));
4 //Overhead: 2N + 1
```

(b) Eager Translation

```
1 int **a = AS_GPU_PTR(int *, data->a);
2 for ( int i=0; i<N; i++)
3       = a[i];
4 //Overhead: 1
```

(c) Best (Eager + Lazy) Translation

Figure 2.9 : Example showing **eager**, **lazy** and **best** approaches

perform better or worse depending on the code patterns in a program. We devise a strategy where we keep both the CPU representation and GPU representation for every pointer. The GPU representation is obtained by converting the pointer eagerly when it is loaded from memory. If at a later use the pointer is stored into a memory location (as $a[i]$ in Figure 2.8), we replace the use by the CPU representation. Otherwise, we use GPU representation. If a pointer is never dereferenced on the GPU, a standard dead code elimination pass eliminates the redundant conversion to GPU address space. Figure 2.9 describes the translation overheads due to these three approaches when applied to the code in Figure 2.8.

We optimize the placement of GPU pointer conversion operations using standard live-range shrinking techniques used in optimal code motion [35].

2.5 Experimental Evaluation

This section evaluates the *Concord* system using a set of regular and irregular data-parallel C++ programs. We first present the overhead comparison between *Concord* and Intel TBB library implementations. We then present comprehensive execution time performance and energy measurements for these workloads using the GPU as well as CPU-only execution. Finally, we demonstrate that our software-based SVM implementation has minimal overhead.

2.5.1 Experimental Setup

We evaluated our *Concord* framework on two systems with integrated Intel 4^{th} generation Core processors running the Windows 7 64-bit operating system: (1) a 1.7GHz Dual-Core i7-4650U Ultrabook with 4GB memory, and (2) a 3.4GHz Quad-Core i7-4770 desktop with 8GB memory. The processor in (2) targets high-performance desktops and servers whereas the processor in (1) is a mobile processor that targets laptops and other mobile devices. While the desktop processor has a higher TDP (Thermal Design Power) budget of 84W, the Ultrabook operates at a low TDP budget of 15W. Energy efficiency is particularly important for mobile systems such as the Ultrabook as it increases battery life. The integrated GPUs on the two systems each have seven hardware threads, each of which is 16-wide SIMD. The desktop GPU is an Intel HD Graphics 4600 with 20 execution units (EUs) and runs at a turbo-mode controlled frequency from 350MHz to 1.25GHz. On the other hand, the Ultrabook GPU is an Intel HD Graphics 5000 with 40 EUs and runs at a turbo-mode controlled frequency from 200MHz to 1.1GHz. We compiled all workloads using CLANG and LLVM version 3.3 with *Concord* extensions and using optimization level -O2. We performed energy measurements by using an internal tool to Intel that measures package

energy by sampling the machine-specific register MSR_PKG_ENERGY_STATUS.

Our evaluation used several regular and irregular data-parallel C++ workloads most of which use pointers extensively. Most of these were ported from existing TBB or multi-core C++ programs. Some were taken from the Rodinia benchmark suite [37], and OpenCV [40], while others were written manually. The origins and static characteristics of the workloads are presented in Table 2.1. The benchmark are summarized below:

1. *Barnes-Hut:* This program uses the efficient Barnes-Hut algorithm for n-body simulation. It partitions the bodies into subregions using an *octree* so that forces from nearby bodies are computed exactly while forces from far-away particles are approximated. We target force calculations to the GPU. Since the octree is unbalanced and traversed recursively to compute the force on each body, the code is highly irregular.

2. *Breadth-first search (BFS):* This program does a breadth-first search in a graph that computes the distance of each node from a specified source node. It uses a compressed row representation and exhibits memory irregularity that depends on the input graph. Our results are for the Western USA road network.

3. *BTree:* This workload uses an n-ary search tree with records stored on leaves of the tree. Searching is targeted to the GPU. Since the search tree is unbalanced, the search process is irregular.

4. *Black-Scholes:* This program calculates the option prices using the Black-Scholes Partial Differential Equation (PDE). It uses a data parallel kernel to iterate over the five features for each option and analytically computes the final price of the option.

Benchmarks	Origin	Input size	LoC	Device LoC	Parallel Construct
Barnes-Hut	In-house	1000000 bodies	828	105	PFH
BFS	Galois [36]	$\|V\|$=6.2M, $\|E\|$=15.2M	866	19	PFH
BTree	Rodinia [37]	command.txt	3111	84	PFH
Blackscholes	PARSEC [38]	64K	713	163	PRH
ClothPhysics	Intel [39]	50K nodes & 200K connections	9234	411	PRH
CC	Galois [36]	$\|V\|$=6.2M, $\|E\|$=15.2M	473	36	PFH
Facedetect	OpenCV [40]	3000x2171	3691	378	PFH
GameOfLife	TBB [21]	10sec	2438	181	PFH
Mandelbrot	TBB [41]	1920x1080	1375	41	PFH
Matmult	In-house	2048x2048	113	11	PFH
NBody	Intel [42]	4096	501	41	PFH
PetMe	Intel [39]	2563 nodes & 10242 connections	9234	411	PRH
Raytracer	In-house [43]	SP=256, MA=3, LI=5	843	134	PFH
Seismic	TBB [21]	1950x1326	733	16	PFH
Skip_list	In-house	50000000 keys	467	21	PFH
StringFinder	TBB [21]	17711 (N=22)	208	14	PFH
SSSP	Galois [36]	$\|V\|$=6.2M, $\|E\|$=15.2M	1196	19	PFH

Table 2.1 : Concord C++ workloads and their characteristics.

parallel_for_hetero(PFH), parallel_reduce_hetero(PRH)

5. *ClothPhysics:* This application models cloth soft-body using a graph consisting of distinct points (nodes) joined by springs (edges). As the cloth moves, new tension and torsion forces are computed for every node by traversing the neighboring nodes.

6. *Connected Component (CC):* This program executes a topology-driven search in a connected-component graph. The search depends on the input graph and so is irregular.

7. *Facedetect:* This program detects faces using Haar-like features that encode information about the faces. A cascade of classifiers is first trained and then applied to an input image. That cascade data structure is traversed during face detection process. The workload comes from OpenCV [40] computer vision library.

8. *GameOfLife:* This program runs two simultaneous instances of the classic Conway's "Game of Life". One of these instances uses serial calculations to update the board while the other executes the games in parallel.

9. *Mandelbrot:* This application computes a Mandelbrot set, a set of points in the complex plane that forms a fractal.

10. *Matmult:* Standard matrix multiplication application.

11. *NBody:* NBody is a popular molecular dynamics code that simulates the movement of particles in a 3D space. The algorithm has two kernels executed in a time-step loop. The first kernel computes the acceleration of the particles and has a $O(N^2)$ complexity where N is the number of particles, and the second

kernel updates the position of the particles based on the new acceleration and has $O(N)$ complexity.

12. *PetMe:* This application simulates entire soft-body characters using cloth simulation techniques. Soft body physics is an increasingly popular feature in video games. Due to their computational intensity, soft body physics is presently used sparingly to depict the movement of cloth, hair, and other flexible elements.

13. *Raytrace:* The key data structure used in raytracing algorithms is a *scene graph* consisting of objects and lights, each represented using a pointer vector. The program uses a parallel version of the algorithm in [43]. During each pixel's color computation, scene graph components are intersected several times. Virtual function dispatch is used to intersect objects.

14. *Seismic:* This application simulates a seismic wave in parallel.

15. *Skip_list:* A skip list stores a sorted list of values using a hierarchy of linked lists, which enables efficient searches in $\mathcal{O}(log\ n)$ steps. While searching for values, this program traverses the intermediate linked-list structures that depend on the input data.

16. *StringFinder:* This example uses parallel_for construct to find a substring match. For each position in a string, the program displays the length of the largest matching substring elsewhere in the string and displays the location of a largest match for each position.

17. *Single source shortest path (SSSP):* This application uses the Bellman-Ford algorithm to compute the shortest path of all nodes from a fixed start node

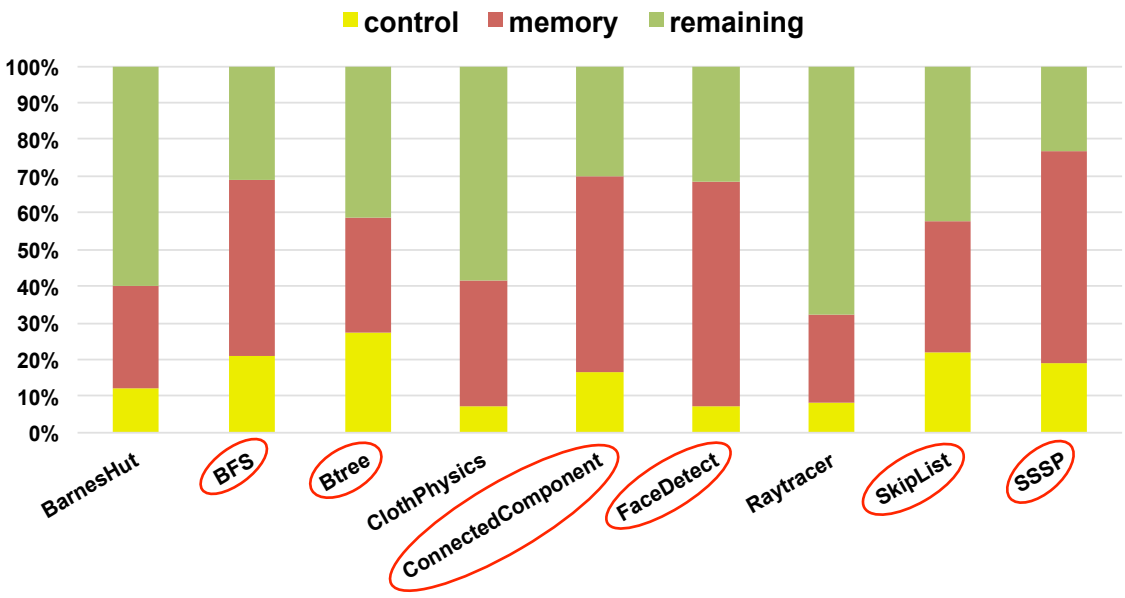


Figure 2.10 : Dynamic estimates of irregularity for each application. Benchmarks which have more than 50% irregularity have been circled in red.

in a directed graph with weighted edges. It exhibits irregular memory access patterns that depend on the input graph.

Irregular workloads tend to show large amount of control flow divergence or uncoalesced memory accesses. To understand the irregularities in our workloads, we collected static measurements of irregularity at the IR level. Figure 2.10 shows the dynamic estimated of ir-regularity in the program. Each bar shows the amount of control flow(yellow), memory operations such as read/write(red) and rest of the computations(green) for a given application.

Pointer-based data-parallel programs are traditionally difficult to port to GPUs without significant software engineering effort. However, since *Concord* supports pointer sharing using SVM and provides TBB-like APIs, We ported most bench-

marks with little effort. Once ported, the *same* C++ code could run transparently on either the CPU or GPU. As an example of the effort involved, one of us ported the `ClothPhysics` application, which consists of 9234 lines of TBB code, to *Concord* in one day without any prior experience with it.

We report execution times for all benchmarks without any hand optimization after the port to *Concord*. We averaged the runtime performance of five runs. Our CPU execution times do not include compilation whereas our GPU execution times include a one-time compilation for each kernel. That is; multiple invocations of a kernel use the cached GPU binary as described in Section 2.4.4.

2.5.2 Performance and Energy Efficiency

Figure 2.11 shows the runtime performance of *Concord* on CPU on the desktop system compared to the TBB library version on CPU. Note that the *Concord* on CPU is compiled down to OpenCL which internally is implemented on top of the TBB library. Hence, the comparison between the CPU version and the TBB version gives us an indication of the overhead due to our *Concord* framework. The CPU version in certain cases beats the TBB version because the back-end compiler is able to better vectorize the OpenCL kernel(generated from *Concord*) due to its parallel semantics.

Figure 2.12 shows the overall GPU speedup and energy savings compared to multicore CPU execution on the ultrabook system. With GPU execution, we found performance improvement ranged from 1.11× to 9.88× with a geometric mean improvement of 2.5× compared to multi-core CPU execution. It is not surprising to see that all workloads show performance improvement from offloading work to the GPU since the integrated 40 EU GPU on this system is more powerful than the dual-core CPU. `Raytracer`, in particular, achieves the best performance improvement

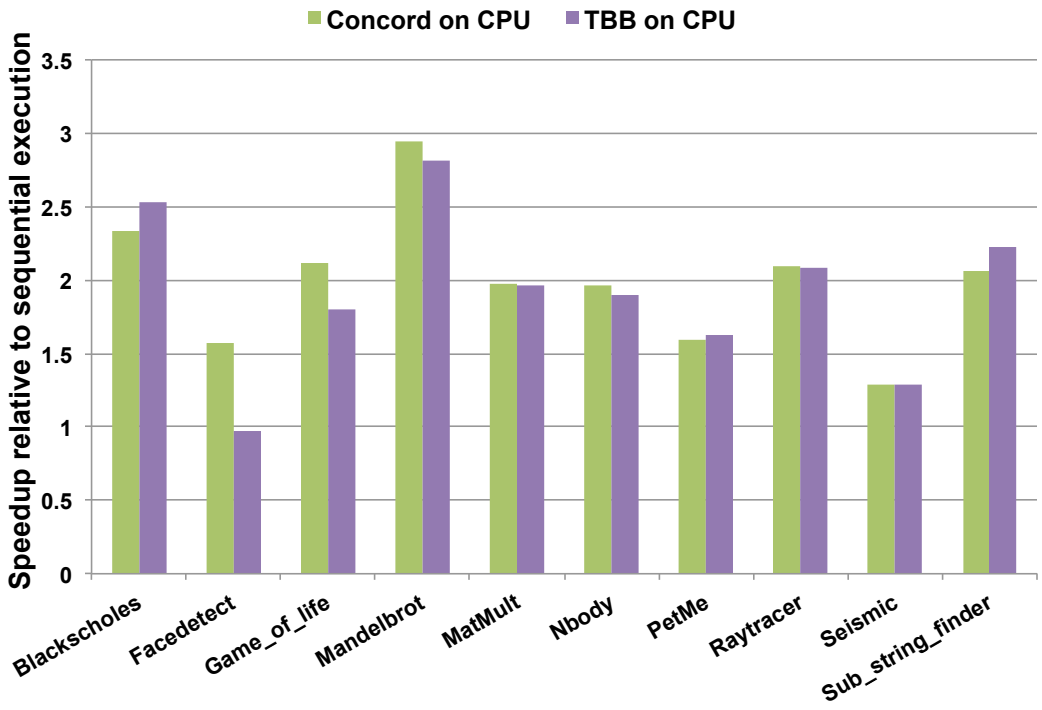


Figure 2.11 : Runtime performance of Concord CPU on the desktop system compared to TBB Library on CPU

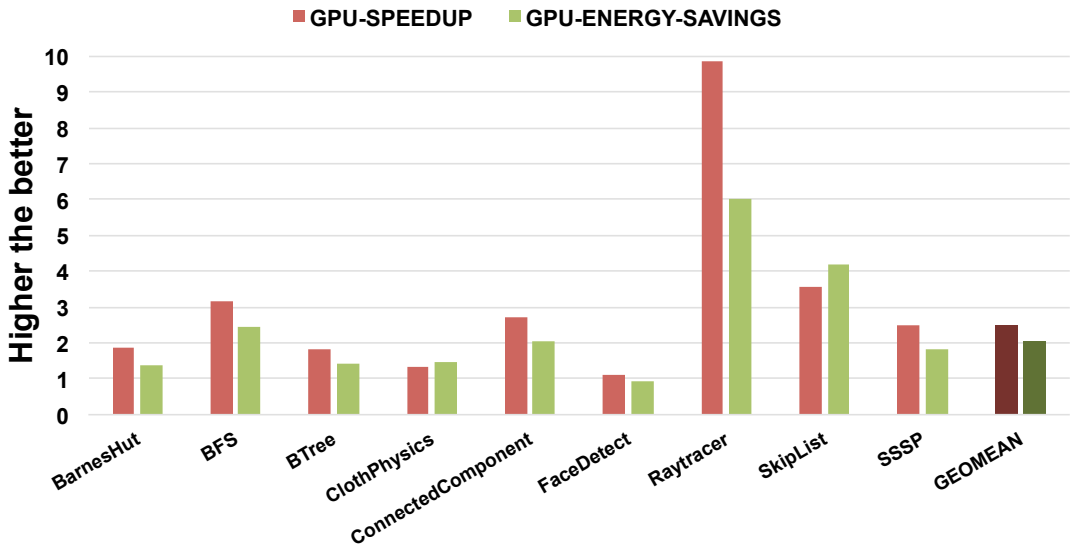


Figure 2.12 : Runtime and energy performance relative to multi-core CPU execution on the ultrabook system

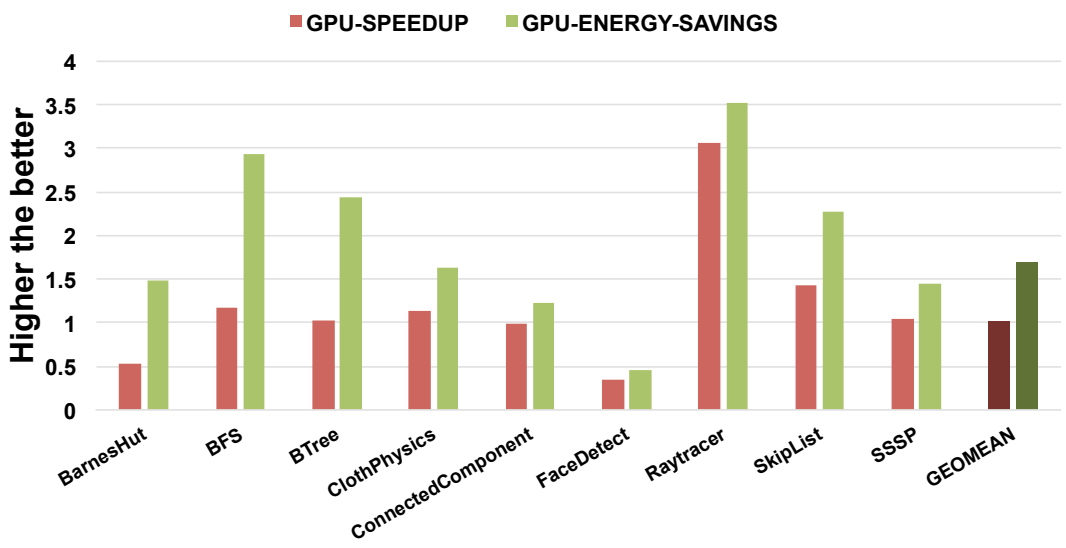


Figure 2.13 : Runtime and energy performance relative to multi-core CPU execution on the desktop system

of 9.88× as it exhibits the least amount of irregularity compared to other workloads (as shown in Figure 2.10). We also observe energy savings ranged from 0.93× to 6.03× with an geometric mean savings of 2.04× using GPU execution compared to multi-core CPU execution. All workloads except `FaceDetect` show energy savings from GPU execution. `Raytracer` has the highest energy savings of 6.04×, which is primarily due to its high performance on the GPU.

Figure 2.13 shows the overall speedup and energy savings compared to multi-core execution on the desktop system. With GPU execution, we found a geometric mean energy savings of 1.69× compared to multi-core CPU execution. All workloads except `FaceDetect` show energy savings from offloading work to the GPU. GPU execution of `BFS`, `Raytracer`, `SkipList`, and `BTree` yield especially significant energy savings–2.94×, 3.52×, 2.27×, and 2.43×, respectively–compared to multi-core

CPU execution. Interestingly, GPU execution results in significant energy savings even though it gives on geometric mean only 1% performance benefit (as shown in Figure 2.13) compared to multi-core CPU on the desktop system. The discrepancy between performance and energy efficiency on GPU vs. CPU is especially pronounced for `Barnes-Hut`, a tree traversal algorithm where the memory coalescing opportunity for two neighboring iterations of the `parallel_for_hetero` loop may depend on the input data. This workload is 47% slower on the GPU than the multi-core CPU, and yet it is 48% more energy efficient.

For the desktop systems, the similar performance on the CPU and GPU for irregular workloads is not surprising since (1) the CPU cores have much higher main memory bandwidth than the integrated GPU cores, and (2) the CPU cores are equipped with highly accurate branch predictors that handle control flow divergence very well. Thus, even though there is a large amount of parallelism on the GPU, GPU performance is hindered by application irregularity.

Overhead of our SW-based SVM

To study the overhead of our SVM implementation, we took one pointer-intensive *Concord* workload, `Raytracer`. We implemented an equivalent OpenCL 1.2 program. Since OpenCL 1.2 doesn't support pointer sharing between the CPU and GPU, the OpenCL Raytracer's host CPU program had to flatten the pointer-based scene graph data structure, convert its embedded vectors into linear arrays, and create OpenCL buffer objects in order to share that scene graph with the GPU. In addition, the *Concord* `Raytracer` code executing on the GPU had to be translated to OpenCL C and modified to traverse the flattened scene graph representation using integer offsets. We found negligible overhead for small images while, for even the

largest image size, We observed only a 6% overhead.

2.6 Summary

A number of specialized languages have been developed for offloading work to GPUs, but their use has been restricted by their complexity and required architectural understanding. Furthermore, these languages are targeted at accelerating regular data-parallel applications operating on array-based data structures, not the kind of pointer-based applications typical in multi-core C++ programming that operate on irregular data structures such as trees and graphs.

This thesis chapter describes the *Concord* C++ programming framework for processors with integrated GPUs. With its support for SVM and most C++ constructs, *Concord* is designed to allow object-oriented C++ data-parallel programs to take advantage of GPU execution. Its compiler optimizations reduce the cost of software-based SVM. Using seventeen realistic regular and irregular C++ applications, we demonstrate that C++ applications using pointers and other object-oriented features can be automatically mapped to GPUs. Furthermore, we demonstrate that GPU execution can bring significant energy benefits to irregular applications even without sophisticated algorithm or data restructuring changes: our results show an average energy savings of 2.04× on an Ultrabook and 1.69× on a desktop over multi-core CPU execution.

Much research has gone into improving the performance of regular data-parallel GPU applications. Our work on accelerating irregular C++ programs is complementary to this research, and could be combined with it for even better results.

Chapter 3

Heterogeneous Habanero-C (H2C)

3.1 Introduction

Today's heterogeneous architectures are diverse and pose severe programmability challenges. The optimization challenges include minimizing the overheads due to communication of data, mapping and scheduling of tasks and maximizing the utilization of the available resources. Current approaches use heuristics to automatically solve some of these challenges, but these approaches are limited to only certain applications or to a single architecture. For example, automatic approaches to mapping of tasks assume that all the processors resources are available for a given task. However, in practice the resources could be shared or are limited. For instance, the memory of a GPU is limited to at-most 12GB in state-of-art devices, and the automatic mapper must now be aware of these constraints and make decisions at runtime which could result in performance degradation. In general, a productive approach is to provide the programmer with high-level constructs to specify the parallelism and possible optimizations in a program, and enable the compiler and runtime to map the specification efficiently. Such an approach has worked well in the case of SMPs where automatic parallelization schemes were limited, but enabling programmers with high-level constructs (like OpenMP) to specify the parallelism eliminated these limitations.

We develop Heterogeneous Habanero-C (*H2C*) by extending Habanero-C [24] (an extension of the popular C programming language) to target multiple heterogeneous

CPU, GPU and APU architectures. The idea is to enable a common programming platform for domain experts, software developers and "ninja" parallel programmers while also providing portability, performance, and productivity. Our main goal is that *"the user writes a machine independent program in* H2C*, and the compiler and runtime generates an executable tuned to the particular hardware".* The principles are similar to High Performance Fortran (HPF), which was introduced in the early 90s for distributed cluster machines. A single HPF program can compile to any distributed cluster. The programming model of *H2C* combines task-based programming model (across devices) and SPMD (within a device).

Some of the constructs introduced in *H2C* handle programmer-specified task partitioning, based on which the compiler and runtime automatically determine the data distributions and necessary data transfers. This approach allows the programmer to use multiple heterogeneous devices with minimal additional programming effort. We also implement compiler optimizations for locality by taking advantage of scratchpad buffers available on heterogeneous hardware. Finally, we implement a lightweight Uniform Event framework that supports point to point synchronization across multiple heterogeneous devices and enabling programming of data-flow applications. We used *H2C* to implement a variety of benchmarks, and observed that *H2C* is more productive, portable and achieves performance similar to expert written programs in low-level languages that target heterogeneous processors.

3.2 Background

In this section, we briefly summarize key tools components/frameworks that are used to implement the *H2C* programming model.

3.2.1 ROSE Compiler Framework

The ROSE compiler framework [44], being developed at Lawrence Livermore National Laboratory, is an open source compiler capable of generating source-to-source code translators and analyzers. ROSE supports multiple languages including C, C++, and Fortran and represents them in a common intermediate representation consisting of an Abstract Syntax Tree, symbol tables, and other data structures. The simple interfaces it provides to modify the IR allow quick development. ROSE uses Edison Design Group's (EDG) C++ front-end to parse C and C++ applications. One can add new language constructs to EDG to extend the base C/C++ languages. The original Habanero-C implementation is based on ROSE.

3.2.2 PolyOpt (Polyhedral Framework)

PolyOpt [45] is a polyhedral loop optimization framework that interfaces with ROSE. The philosophy of polyhedral optimizations is to use mathematical abstractions to analyze and optimize programs. Polyhedral analysis enables many loop optimizations like loop-reversal, loop skewing and can also be applied to data locality optimizations, memory management optimizations, communication optimizations, etc. However, one of the drawbacks is that polyhedral optimizations are limited to loop bounds, array accesses and conditionals that are affine functions of the loop iterators. Program regions that are amenable to polyhedral optimizations are called *static control parts* (SCoP) [46]. Figure 3.1 shows a sample output SCoP that is in a matrix format.

```
//      i    N    K    M    const
//# Iteration domain
//1     1    0    0    0     0     ## i >= 0
//1    -1    1    0    0    -1     ## -i + N - 1 >= 0
//# Read access informations
//1     1    0    1    0     0     ## B[i + K]
//1     1    0    0   -1     0     ## B[i - M]
//# Write access informations
//2     1    0    0    0     0     ## A[i]

for(i = 0; i < N; i++)
   A[i] = B[i + K] + B[i - M];
```

Figure 3.1 : Sample output SCoP for a vector add program

3.3 Programming Model

H2C is a high-level programming language that targets heterogeneous architectures by building on Habanero-C constructs. The high-level parallel constructs in *H2C* are as follows:

Language Extensions

- **comm_async** *copyin*⟨*args*⟩ *copyout*⟨*args*⟩ *at*⟨*device*⟩: Asynchronously copy data specified by the arguments "from" and "to" the *device*. These communication operations are assumed to be initiated by the host. The *at* clause is used to specify the device where the data is located.

- **forasync** *point*⟨*args*⟩ *range*⟨*args*⟩ (*optional clauses*) *at*⟨*device*⟩{Body}: Multi-

dimensional data parallel loop. The loop indices are specified by the *point* clause. The loop bounds are specified by the *range* clause and *at* clause is used to specify the mapping of the kernel to the available devices. There is no implicit barrier at the end of the forasync construct. The programmer is responsible for ensuring that the loop iterations are logically independent and can be executed in parallel (no ordering is assumed even in the presence of floating-point computations).

Optional forasync Clauses:

seq$\langle args \rangle$: forasync is compiled down to multiple tasks based on the underlying architecture. The granularity of each task can be specified using the *seq* clause.

scratchpad$\langle args \rangle$: Specify the variables to take advantage of the available scratchpad buffers. Example: On the GPU this clause could be used to promote the specified variables to take advantage of local shared memory buffer.

partition$\langle args \rangle$: Specify the mapping of the tasks onto the available processors.

- **finish** $\langle Body \rangle$: Ensures that comm_async and forasync tasks spawned inside *Body* are completed.

- **await** $\langle events \rangle$: Wait until the specified events are completed.

- **phased-next**: Enables point-to-point synchronization. The default is a flat barrier.

- **single**{Body}: Used inside a parallel region to ensure only a single thread executes the *Body*.

Compiler Framework

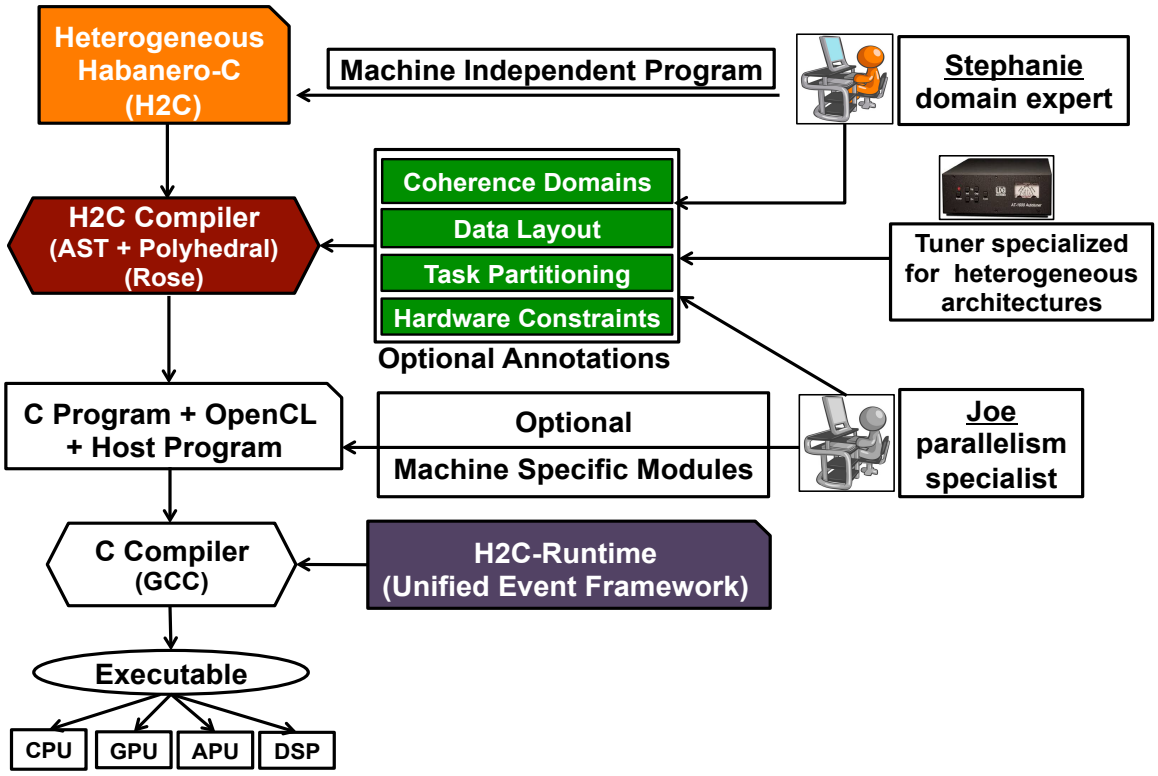


Figure 3.2 : Overall compilation flow

The compilation framework consists of a static compiler based on the ROSE infrastructure. The given *H2C* program is translated to a C program. The static compiler automatically generates host-side binary with embedded OpenCL code (for the comm_async and forasyncs). It also uses PolyOpt to extract the polyhedral information (SCoP) from forasync constructs.

Figure 3.2 shows the overall compilation framework of *H2C*. Following our goal of a single source and multiple targets, the *H2C* compiler takes a machine independent program and generates OpenCL tuned to a particular processor. To generate target specific OpenCL code, the compiler uses information such as "*coherence domains*"

to avoid communication where possible. Different "*data layouts*" can be specified for each processor and a set of arrays. "*Scheduling details*" indicate the type of executions units present such as SIMT(GPU) or SMP(CPU). Additional "*hardware constraints*" such as lack of function call support or availability of special instructions can be specified either by the programmer or an auto-tuner specialized for heterogeneous architectures. These optional annotations are specified in a file or can be inferred from the *H2C* utility tool described later. Optionally, expert programmers can now add hand-coded and "ninja" optimized OpenCL modules separately. The generated C program is linked with the *H2C* runtime, and a target specific executable is built.

Figure 3.3 shows a sample program written in *H2C*. The program performs a vector addition on a *gpu* device. Lines 1-4 initialize parameters used for the kernel execution. Lines 5-7 allocate memory for arrays A, B, C. The memory is allocated via a special memory allocator introduced in Section 3.3. Line 8 initializes these arrays. Line 9 creates a new finish scope. Line 11 launches a comm_async(data movement) task that copies arrays B, C from host to device *gpu*. The parent task continues to execute method call $foo1$ (line 12) in parallel with the communication task. The end of the finish scope on line 13 ensures both $foo1$ method call execution and the communication task are completed. Line 14 creates a new finish scope. Lines 16-18 show a forasync construct that specifies an array addition kernel task to be launched asynchronously. The kernel will be executed on device *gpu* if all the device constraints are satisfied defaulting to host otherwise. The parent task continues to execute method call $foo2$. The end of the finish scope on line 20 ensures the kernel task and method $foo2$ are completed. Finally, line 21-25 copy the data back from the device to host and execute method call $foo3$. The data movement is overlapped with the execution of method call $foo3$. The end of the finish scope on line 25 ensures

```
1  int dev_cnt = 1, M = 1024, N=1024;
2  int size = N * M * sizeof(float);
3  int gpu=2; //dev id 2 is TESLA M2050 GPU
4  int dev_lst[dev_cnt] = {gpu};
5  float *A = hc_malloc(size, dev_cnt, dev_lst);
6  float *B = hc_malloc(size, dev_cnt, dev_lst);
7  float *C = hc_malloc(size, dev_cnt, dev_lst);
8  initialize(A,B,C);
9  finish{
10 //asynchronously copy data from host to device
11     comm_async copyout(B,C) at(gpu);
12     foo1();
13 }//wait for the copies to complete
14 finish{
15 //asynchronously execute the kernel
16     forasync point(i,j) range(0:M,0:N) at(gpu){
17         A[i * N + j] = B[i * N + j] + C[i * N + j];
18     }
19     foo2();
20 }//wait for the kernel execution
21 finish{
22 //asynchronously copy data from device to host
23     comm_async copyin(A) at(gpu);
24     foo3();
25 }//wait for the copy to complete
```

Figure 3.3 : Example H2C vector add program

```
1  __kernel void kernel_1(__global float *a,  __global float *b,
2                                  __global float *c, int N) {
3    i = get_global_id(1);j = get_global_id(0);
4    if(i<M && j<N)//Padding for body
5       a[i * N + j] = b[i * N + j] + c[ i * N + j];
6  }
```

Figure 3.4 : Generated OpenCL kernel

```
1  void offload(float *a, float *b, float *c,int N, char *kernel_name,
2                                          int dev,domain r){
3    kl = get_kernel(kernel_name,dev);
4    ind0 = get_buffer(C,dev);
5    clSetKernelArg(kl, 0, sizeof(cl_mem), &ind0);
6    ..........
7    clEnqueueNDRangeKernel(cmmnd, kl, 2, r.offset, r.global, r.local,
8                                           0, NULL, NULL);
9  }
```

Figure 3.5 : Generated host program

both $foo3$ method call execution and the communication task are completed.

Figures 3.4 shows the generated OpenCL kernel. The pseudo-code for the kernel generation is described in Algorithm 1 described later.

Figure 3.5, shows the generated host program. The host program contains the necessary OpenCL glue code required for kernel execution and data movement. This is auto-generated by the compiler.

```
1  hc_runtime_init();//added by compiler
2  ..........//allocate memory
3  hc_start_finish(gpu);
4    hc_start_async(gpu);
5      copy_to_device(b,gpu);
6      copy_to_device(c,gpu);
7      foo1();
8    hc_stop_async(gpu);
9  hc_stop_finish(gpu);
10 hc_start_finish(gpu);
11   hc_start_async(gpu);
12     domain range = .....;
13     offload(a, b, c, N, "kernel_1", gpu, range);
14     bar();
15   hc_stop_async(gpu);
16 hc_stop_finish(gpu);
17 ................
18     copy_from_device(a,gpu);
19     foo2();
20   hc_stop_async(gpu);
21 hc_stop_finish(gpu);
```

Figure 3.6 : Generated C program

Figure 3.6 shows the generated C program. The C program contains the original application code along with *H2C* runtime API (*H2C* constructs are lowered to *H2C* runtime API).

The novelty of *H2C* comes from the support of the partition clause. This clause

```
forasync point(i) range(0:N) at(dev1,dev2) partition(N/3, 2*N/3){
    A[i] = B[i + K] + B[i - M];
}
```

Figure 3.7 : Iteration partition example

is used with the forasync construct to specify the iteration partitions. The partition clause helps utilize all the available processors and also enables the programmer to specify a heterogeneous distribution of tasks. Figure 3.7 shows an example *H2C* program that uses the partition clause. In this example, the partition semantics mean that the N iterations of the forasync clause are to be partitioned such that, the first $N/3$ iterations are to be executed on device *dev1*, and the remaining $\frac{2*N}{3}$ iterations are to be executed on the device *dev2*. The compiler now automatically determines the *data distributions* for each device. The arguments to the at clause are device ids that are an integer type. The device ids are generated by an H2C *utility tool* for a given heterogeneous architecture. Figure 3.8 shows a sample list of devices and their device ids along with some architectural details generated by the tool.

H2C supports two compilation modes: "*implicit*" and "*explicit*", with the later as the default mode. In the implicit mode, the compiler automatically generates code for the data communication and coherence using a dependency analysis. In the explicit mode, the user is responsible for data movement and coherence across device. The implicit mode makes conservative assumptions based on the dependency analysis and may not generate the optimal code. On the other hand, the "explicit" mode gives more freedom to the programmer and allows the communication to move beyond function calls and file modules.

ID	Name	Type	#Cores	Memory(MB)
0	Intel Xeon	CPU	12	48255.9
1	Tesla M2050	GPU	14	2687.4
2	Radeon HD 5970	GPU	20	4087.1

Figure 3.8 : Output from H2C utility tool containing device IDs and architectural information

Runtime Framework

The memory allocation in *H2C* is via *hc_malloc*, that is a special wrapper over standard *malloc*.

```
void* hc_malloc(size_t size, int dev_count, int *dev_list);
```

The arguments include the size of the memory requested, the number of devices and the corresponding device ids where the memory is to be allocated. The allocator creates buffers on the host and on the devices specified. All the data movement between these memory buffers happen via the host. Note: If the host and device share the same physical memory (e.g.,: Integrated GPU), there is no need for data transfer.

Events (*hc_event*) form a powerful component of the *H2C* runtime as they provide point-to-point synchronization support. Events also enable applications to run efficiently by overlapping the computation and communication. The *await* clause is used to specify a dependency between two tasks. We implement unified event framework

to manage events across multiple heterogeneous devices.

3.4 Implementation

In this section, we describe our *H2C* compiler and runtime implementation. As described earlier, the compiler generates C program from a given *H2C* program and inserts calls to the *H2C* runtime. The *H2C* runtime is implemented on top of OpenCL. Each device vendor provides an OpenCL implementation, and the *H2C* runtime acts as a uniform layer on top of them. At the beginning of an *H2C* program, a call to the "*hc_runtime_init()*" is made that initializes the *H2C* runtime. During initialization, the *H2C* runtime instantiates available OpenCL devices for each processor and assigns device ids for each processor similar to the *H2C* utility tool described in Figure 3.8. For each device, it creates contexts, command queues, builds OpenCL kernels and stores few hardware specific details such a workgroup size limits and available DRAM memory. These entities are accessed via the device id corresponding to the device.

3.4.1 Asynchronous Computation and Communication

The comm_async construct is used to communicate the data between devices asynchronously. The "*copyout*" construct copies the data from the host to the device and is translated to "copy_to_device(void *host_ptr, int dev)" runtime call. The "*copyin*" construct copies the data from the device to the host and is translated to "copy_from_device(void *host_ptr, int dev)" runtime call. These calls are implemented on top of the OpenCL read/write buffer API.

The forasync construct is a multi-dimensional data parallel loop construct. The body of the forasync construct is translated to an OpenCL kernel specialized to a particular device. The devices are specified by the at clause. The iteration domain

Algorithm 1 Generate OpenCL kernel

Input: F::forasync body
Output: *body*::OpenCL kernel

```
1: body = F.outline();
2: Append_OpenCL_attr(body);
3: InsertPad(body);
4: InsertGlobalDecls(body);
5: if MetaFile().present then
6:     PerformDataLayout(body);
7: end if
8: if (CheckReuse(body)&&IsScratchpad()) then
9:     Tile(body)
10: else if (CheckStencil(body)&&IsScratchpad()) then
11:     StencilTile(body)
12: else if (CheckReduce(body)&&IsScratchpad()) then
13:     ReduceTile(body)
14: end if
15: return body
```

is derived from the clauses specified to the forasync. The algorithm to generate the OpenCL kernel is shown in Algorithm 1. Line 1 outlines the body of the forasync. Line 2 appends OpenCL specific attributes such as "*__kernel*", "*__global*", etc. to variables and parameters of the outlined body. Line 3 inserts padding to the body of the outlined call. The padding is required to handle cases when the global work-group size is not a multiple of the local work-group size. Line 4 then inserts global constants, structure declaration, and other globals used by the body. Line 5 performs the data layout transformation of the array accesses if specified. The data layout transformation framework is described in another paper [47]. Lines 8-14 check for reuse, stencil, and reduction patterns in order to take advantage of scratchpad buffers like shared memory on the GPU. This optimization is explained in section 3.4.4.

The forasync construct is replaced by a call to "*offload(...)*" with the corresponding arguments including kernel name and domain of the kernel. The programmer specifies the work-group size via the range clauses. "*offload(...)*" executes the corresponding OpenCL kernel using the "clEnqueueNDRangeKernel" API call. A runtime call to "*hc_start_async(int dev)*" is made at the start of the forasync, comm_async scopes. A call to "*hc_stop_async(int dev)*" is made at the end of the forasync, comm_async scopes. Each asynchronous task creates a new command queue.

Finally, the finish construct is a synchronization point (barrier) and ensures all the tasks (communication + computation) executed within its scope are completed. A runtime call to "*hc_start_finish()*" is made at the start of the finish scope and a call to "*hc_stop_finish()*" is made at the end of the finish scope. The "*hc_stop_finish()*" calls the OpenCL "clFinish()" API. The command queues are derived from the at clause arguments provided in the finish scope.

Each call to *hc_start_async* creates a new command queue for the specified device. All the communication and computation tasks in the scope of the async (until *hc_stop_async* is reached) are now enqueued into this command queue. Two command queues can logically execute in parallel, and this follows the async semantics. A call to the *hc_start_finish* begins a new finish scope and all the commands queues created within this scope are recorded. When the execution reaches the corresponding *hc_stop_finish*, a *cl_finish* for the recorded command queues ensures all the tasks in the scope are completed.

3.4.2 Iteration Partitioning

The partition clause is used to specify the forasync iteration partitions. Figure 3.9 shows the partition clause used for the forasync construct with integer values as ar-

```
forasync point(i) range(6:1030) at(dev1,dev2) partition(512,512){
    A[i] = B[i + 8] + B[i - 6];
}
```

Figure 3.9 : Partition example to determine the amount of data to be copied

guments. The arguments to the partition pragma are the iteration domain, one for each device specified in the at clause. In the task partitioning scheme, the programmer provides a partition of the iteration space (applies to forasync construct) and the compiler determines the data distribution. The data distribution is used to determine the amount of memory that needs to be allocated and communicated to each device. We only consider block distributions in our work.

For example, the data parallel loop in Figure 3.9 has an iteration domain ranging from 6 to 1030, a total of 1024 iterations (1030 is excluded). The data domain of array A varies from $[6, 1030)$. Array B has two data domains, $[14, 1038)$ and $[0, 1024)$ for the two data references. Let's assume the programmer decides to partition the iteration domain onto two devices(dev1, dev2) with 512 iterations each. Array A is distributed into two blocks of 512 elements each. However, the data domain of array B for the dev1 is $[14, 526)$, $[0, 512)$ which when combined are data elements $[0, 526)$ while that of the dev2 device is $[526, 1038)$, $[512, 1024)$ which when combined are data elements $[512, 1038)$. This optimal data distribution is copied to the corresponding devices.

H2C determines the data distribution with the help of the SCoP information generated by the Polyopt framework and PIP [48] library. At compile-time, PIP takes as input, a matrix of linear inequalities(loop bounds, array references) obtained from the

Algorithm 2 Forasync partitioning

Input: f::forasync

Output: $MapRead$:: Read data partition, $MapWrite$:: Write data partition

```
1: PartList = f.partition();
2: SCoP B = PolyOpt(f);
3: for R ϵ ArrayReferences(B) do
4:     //min/max value for each array affine index in terms of the partition values
5:     Quast = ComputeBounds(R,B,PartList);
6:     MinDom = Quast.Min();
7:     MaxDom = Quast.Max();
8:     if WRITE(R) then
9:         MapWrite.add(R,(MinDom,MaxDom));
10:    else if READ(R) then
11:        MapRead.add(R,(MinDom,MaxDom));
12:    end if
13: end for
14: CoalesceDomains(MapRead);
15: GenerateCopies(MapRead);
```

polyhedral SCoP format. This matrix is used to generate the lexicographic minimum and maximum expressions of the given affine array references. These minimum and maximum expressions are used to generate the code for data movement. Note that these expressions might contain symbolic variables and constants. The actual data movement is performed at runtime when the symbolic parameter values are known.

Algorithm 2 describes the steps involved in the forasync partition. The partitions and the SCoP are extracted from the forasync construct on lines 2-3. On lines 4-15, the corresponding read/write data distributions are inferred from the PIPLib. These values are in terms of the partition parameters. Finally, on lines 17-18 the read domains that overlap are merged into one domain, and the corresponding code for data copies is generated. The domain merging is similar to the communication coalescing

```
forasync point(i) range(0:1024) at(cpu,gpu){
    if(i%2==0)
        A[2*i] = 1;
    else
        A[2*i+1] = 2;
}
```

Figure 3.10 : Disjoint but overlapping partition

work by Chavarria-Miranda [49]. The semantics of the forasync construct ensure that the write references do not overlap. Note that we approximate the distributions to be copied by computing the rectangular and cubic domains (convex hull), and these are efficiently supported in OpenCL.

We duplicate the read reference where necessary. Essentially, all the read data is local to a processor before the execution begins. Once the execution completes, the write data is merged at the synchronization point. Merging the output buffers after the kernel execution might be non-trivial. It is possible that the write indices are disjoint, but the domains overlap with each other. Figure 3.10 shows an example where the even iterations write to even indices and vice-versa. If we partition the kernel into two 512 iteration domains, the output buffers will have to be manually combined elements by element on the host side. This problem also occurs in runtimes that automatically manage the coherence among heterogeneous processors. However, in our work we only partition the iteration domain if the data distributions of the write references are non-overlapping. A compiler error is generated in this case of overlapping write references.

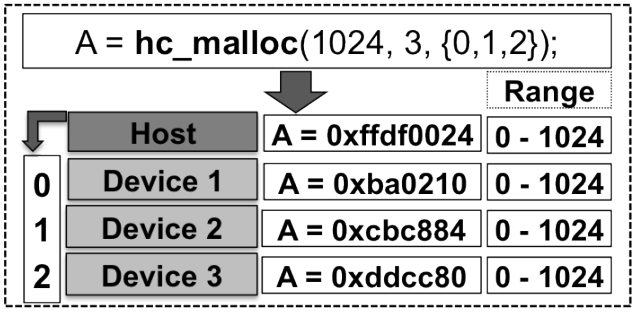


Figure 3.11 : hc_malloc implementation

3.4.3 Memory Management

The buffer management module implements the *H2C* memory allocator described in Section 3.3. A call to the memory allocator instantiates a fat pointer that contains the information of the base addresses of each device and the range of the buffer. This fat pointer is indexed using the host base address and the device id. Figure 3.11 show an example of *hc_malloc* for buffer allocation size of 1024 for three devices. The call creates buffers of sizes 1024 on each of the devices specified and also the host. The *hc_malloc* call now returns a fat pointer with the device-id and device buffer base address. This fat pointer is stored on the host. Now, for a given device-id, the base address can be retrieved in constant time.

3.4.4 Compiling for Scratchpad Buffers

Some processors have low-latency memory buffers (local shared, constant memory on GPU and MSMC on a DSP). One can copy frequently used data to these buffers to improve the access time. *H2C* is capable of taking advantage of these buffers by looking for patterns in the forasync body. Figure 3.12 shows three sample patterns that can be re-written to exploit data locality. These patterns can also be considered

```
1  //Locality Reuse Pattern
2  forasync point(i) range(0:N) at(gpu){
3     for(int j = 0 ; j < N; j++){
4        A[i] = (B[i] + B[j])/2;
5     }
6  }
7  //Stencil Reuse Pattern
8  forasync point(i,j) range(0:M,0:N) at(gpu){
9     A[(i*N) + j] = B[(i*N) + j] + B[(i*N) + j + 1]
10       + B[(i*(N + 1)) + j] + B[(i*(N - 1)) + j];
11 }
12 //Reduction Pattern
13 forasync point(i,j) range(0:M,0:N) at(gpu){
14     A[i] += B[i*N + j];
15 }
```

Figure 3.12 : Reuse patterns for scratchpad optimization

as an "*embedded domain specific language*". Optionally, the "*scratchpad*" clause can be used as a hint to the compiler to promote only the specified variables. *H2C* tiles the code when it is legal do so and generates code to copy the data to these scratchpad buffers. It then changes the corresponding accesses via these scratch pad buffers. Figure 3.13 shows the OpenCL code generated for the locality reuse pattern in Figure 3.12.

```
1  #define LOCAL_SZ 256
2  __kernel void kernel_1(__global float *A, __global float *B,int N){
3      int i = get_global_id(0),out_j,in_j;
4      int local_id = get_local_id(0);
5      __local float loc_B[LOCAL_SZ];
6      for(out_j=0; out_j < N/LOCAL_SZ;++out_j){
7        int j = out_j*LOCAL_SZ + local_id;
8        loc_B[local_id] = B[j];
9        barrier(CLK_LOCAL_MEM_FENCE);
10       for(in_j = 0;in_j < LOCAL_SZ;++in_j){
11          A[i] = (B[i] + loc_B[j])/2;
12       }
13       barrier(CLK_LOCAL_MEM_FENCE);
14     }
15 }
```

Figure 3.13 : OpenCL code generated for locality reuse

3.4.5 Unified Event Framework

A programmer can use *hc_event* to specify dependencies between 2 tasks. The *await* clause is used to specify the sink of the dependency. *H2C* runtime implements the *hc_event* on top of OpenCL events. However, dependencies in OpenCL are associated only with a single context. The complexity of the event management arises from the fact that the programmer can now specify a dependency between events in two different contexts (devices). An event in one context cannot be resolved from another context. To overcome this limitation, we implement a unified event (UE) framework on top of different OpenCL contexts. UE is implemented on the host with the help

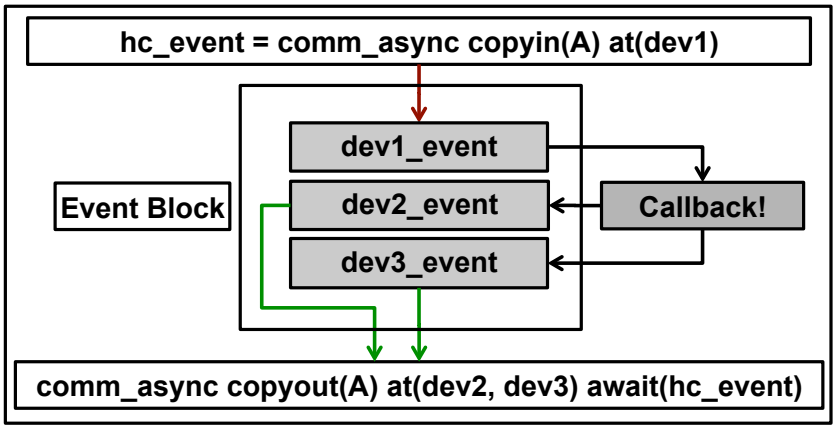


Figure 3.14 : Unified Event framework using Event Blocks

of an *Event Block*. Figure 3.14 describes the Event Block (hc_event) implementation. Each Event Block consists of a set of events (input event, output events), one per each context (device). An input event, (dev1_event) is registered with the context of device1. The output events, *dev2_event, dev3_event* are registered with contexts of device2 and device3. A callback is implemented such that output events are satisfied when the input event is satisfied.

3.5 Experimental Evaluation

In this section, we evaluate the **productivity**, **portability**, and **performance** of the *H2C* programming language.

We use an Intel X5660 Xeon CPU with 6 cores (each 2 HT), running at 2.8 GHz, and an NVIDIA Tesla M2050 GPU, with 14 SMs (each 32 cores) running at 1.1 GHz, to evaluate the performance of *H2C*. The compiler used to compile the generated C versions of each application is GCC 4.4.6 (with the flags -g -O2). All OpenCL kernels were compiled with their default optimizations enabled. Intel CPU tests were

Name	Description	# Kernels	Data Type	Input
`Seismic` [21]	Seismic Wave Simulation	2	Float	10K × 10K
`LBM` [50] [51]	CFD Simulation	2	Float	300×300×300
`NBody` [21]	Molecular Dynamics	2	Float	100K
`Jacobi1D`	Smoothening Algorithm	1	Float	102400K

Table 3.1 : Characteristics of benchmarks used in the evaluation.

performed using 2011 Release of Intel OpenCL SDK, v1.5. NVIDIA GPU tests were performed using NVIDIA SDK v5.0. Table 3.1 summarizes the benchmarks we use in the evaluation including their description and compile-time characteristics. The *H2C* implementation have been extended from OpenMP and sequential versions of the programs. OpenCL implementations have been hand-written.

Productivity Evaluation

We measure the productivity using software productivity metrics. Lines of code (LoC) is used to compare the ease of programming. Cyclomatic Complexity (CC) metric [52] measures the control flow structure of programs and indicates the divergence in a given program. Halstead's metrics [53] help evaluate software complexity. The Mental Effort (ME) is computed using a set of Halstead metrics and represents the effort required to develop and understand a program in a specific programming language. A lesser value is desired for all three metrics. We use these metrics and evaluate the ease of programming with *H2C* compared to OpenCL. Table 3.2 shows the comparison of these productivity metrics between *H2C* and OpenCL. We observe that *H2C* requires lesser lines of code (LoC), involves less complex control flow(CC) and also requires

Name	LoC		CC		ME ($\times 10^3$)	
	H2C	**OCL**	**H2C**	**OCL**	**H2C**	**OCL**
`Seismic`	114	210	12	14	864.9	1111.4
`LBM`	1395	3426	147	578	2981.8	10164.9
`NBody`	101	197	9	10	861.4	1874.2
`Jacobi1D`	30	117	3	6	71.9	390.5

Table 3.2 : Comparison of Lines of Code (LOC), Cyclomatic Complexity (CC), Mental Effort (ME) for *H2C* and OpenCL (OCL) (Lower is better for all metrics)

less mental effort (ME) for development. The difference for LBM is notably high since the corresponding OpenCL program was written in a more generic manner resulting in higher code size. However, this generalized code does not affect the execution time.

Portability and Performance Evaluation

We show the portability by compiling the same *H2C* program onto multiple architectures. The OpenCL GPU versions of Jacobi1D and NBody require explicit communication from the host and custom kernel modifications to take advantage of scratchpad buffers. *H2C* language and compiler can generate different kernels from the same source. We evaluate the performance using the partitioning constructs of [*H2C*]. The timings reported are the average of five runs and include both computation and communication time. We partition the forasync loops of each application onto a single CPU, single GPU, two GPUs, and a combination of two GPUs + single CPU. The partition sizes for each of these configurations have been determined based on the individual timings on each device. We also measure the execution time of the

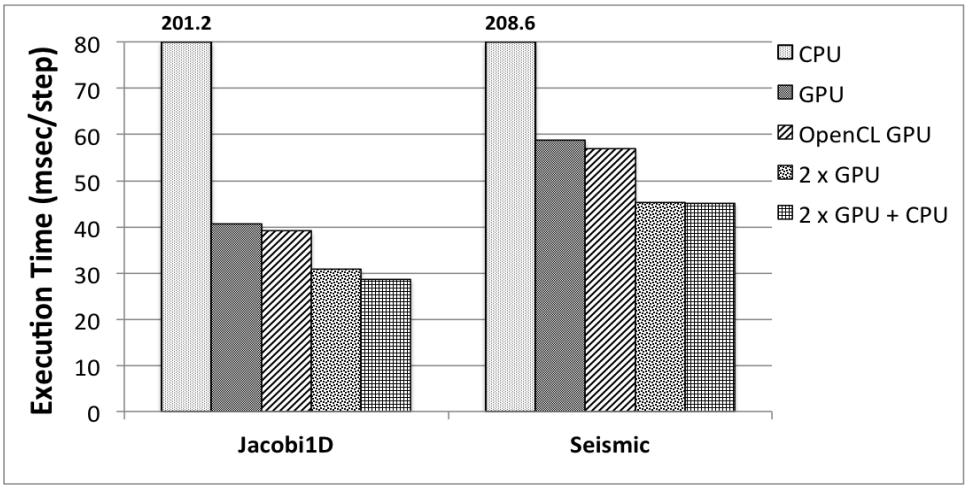


Figure 3.15 : Execution time (msec/step) of Jacobi1D and Seismic due to iteration partition on multiple devices

hand-coded OpenCL implementations on a single GPU device.

Figure 3.15 shows the execution time for Jacobi1D and Seismic applications when partitioned onto multiple platforms. The performance of these programs increases when partitioned onto multiple GPUs but tends to flatten out when including the CPU device. This is because the additional communication overhead involved with the boundary regions negates any potential performance benefit from adding a CPU. Both Jacobi1D and Seismic are memory-bound applications. Bars 2,3 show that the performance of *H2C* programs is similar to the corresponding hand-coded OpenCL versions.

Figure 3.16 shows the performance of two versions of the NBody program when partitioned onto multiple platforms. The first version (NBody) does not take advantage of locality optimizations. We observe an improvement in performance when executed onto multiple heterogeneous devices. NBody application is compute bound, and overhead due to communication is low. The second version (NBody_Opt) takes

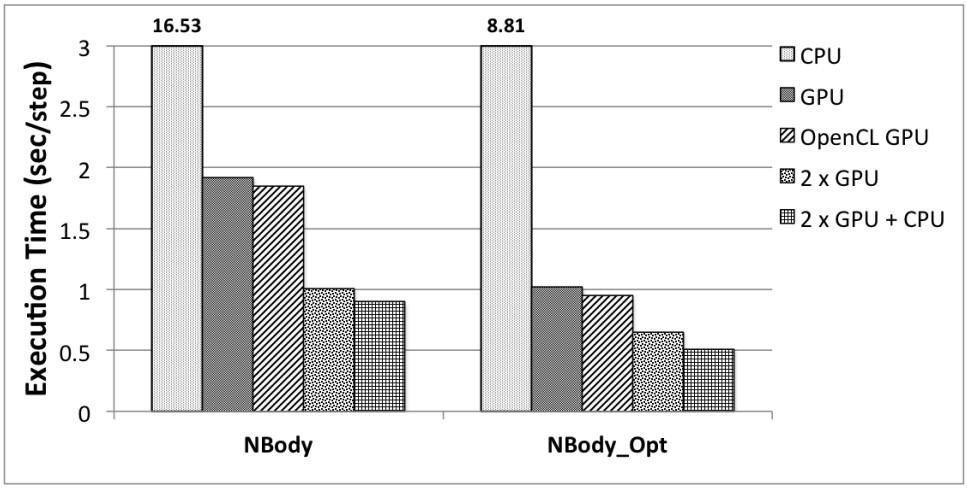


Figure 3.16 : Execution time (sec/step) of NBody and NBody_opt(locality optimized)

advantage of the locality optimizations (scratchpad buffers). We observe almost a 2× improvement over the unoptimized version. This version also scales well over multiple heterogeneous devices. Bars 2, 3 show that the performance of NBody written in *H2C* is similar to the corresponding OpenCL versions. Note that the *H2C* version automatically generates the locality optimized code. The compile-time overhead from PIPlib to evaluate the data distribution parameters from the task partitions is insignificant in all the benchmarks.

We evaluate our unified event framework by implementing two versions of the Lattice Boltzmann Method (LBM) application. LBM simulation is widely used in the oil and gas industry to identify the porosity of rocks. It has the common stencil pattern where the first kernel computes the grid; then the ghost regions are exchanged and finally merged locally. However, LBM can suffer from dynamic load imbalance based on the input data. The first version uses comm_async, forasync and finish constructs to implement the LBM application. This version restricts the overlap between communication and computation because the finish acts as a flat barrier. The second

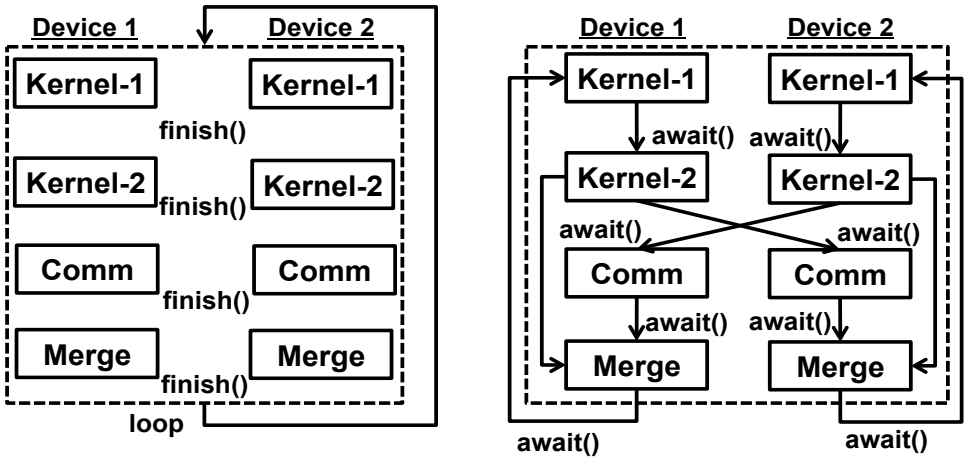


Figure 3.17 : LBM implementation of "finish" (left) and "await"(right)

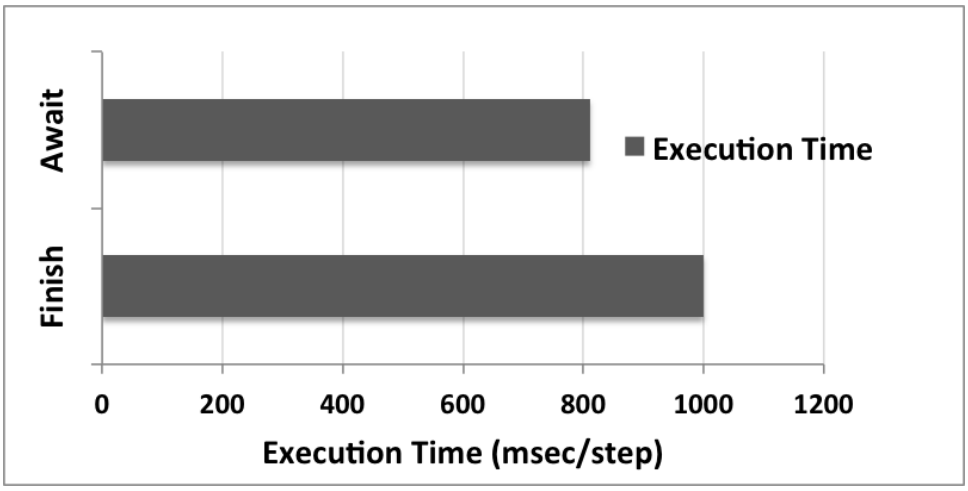


Figure 3.18 : LBM execution time (msec) for "finish" (bottom) and "await"(top)

version uses comm_async, forasync and await constructs to implement the LBM application. This version allows the overlap between communication and computation because the await construct enables point to point synchronization support between various tasks. Figure 3.17 shows two implementations of LBM. On the left is the finish implementation and on the right is the await implementation. The await clause

uses the *H2C* unified event framework to synchronize across devices.

Figure 3.18 shows the benefits of using events in the LBM application. The finish version incurs high communication overhead due to the flat barrier. However, the await version can overlap the communication with the computation. We observe a speedup of 1.23× for the await version relative to the finish version.

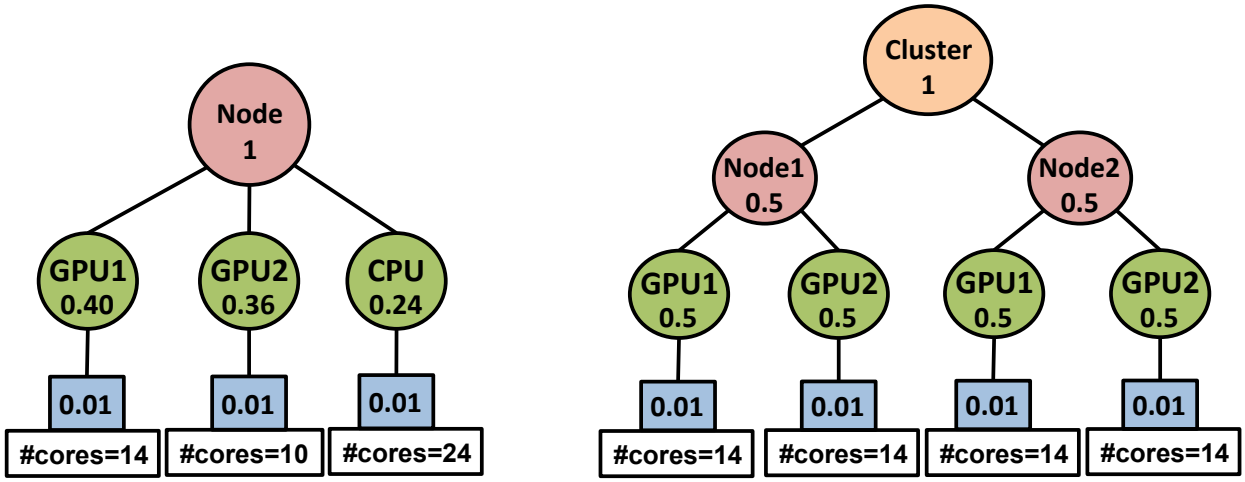
3.6 Extensions

H2C currently support only a single node (*H2C* can handle multiple devices on a single node). Extending them to a distributed cluster of heterogeneous nodes will expand its scope. We describe extensions to *H2C* to program a distributed heterogeneous cluster.

We propose Hierarchical Device Trees (HDT) to achieve this. HDT has been influenced from Hierarchical Place Trees (HPT) [54]. The idea is to specify the at and the corresponding partition arguments in an XML file. The programmer can now maintain different HDT topologies for different clusters and nodes.

Hierarchical Device Trees (HDT) Model

In the Hierarchical Device Trees (HPT) model, each core of a CPU, GPU, APU (Integrated CPU+GPU) or an FPGA is abstracted as a single leaf node, and the heterogeneous system is abstracted as a device tree. The device tree abstracts the underlying hardware (cluster, node, device, thread). Each tree node has a value that can be used to specify a partition at a given device node level. For instance, the leaf level nodes can be used to specify the chunk-size (CPU) or work-group size (GPU). Essentially, each level of the tree represents a hierarchy of the parallelism available on the underlying hardware.



(a) HDT: 2 GPUs and 1 CPU

(b) HDT: Cluster of 2 nodes with 2 GPUs each

Figure 3.19 : Hierarchy of devices for a single node and a two node cluster. The value in each node represents the partition of the work on each device.

Figure 3.19 shows two sample HDTs. Figure 3.19(a) shows an HDT for a single node with two GPU devices and a single CPU device. The values of each node specify the maximum partition. For example, if the total number of tasks are $100K$, GPU1 gets $40K$, GPU2 gets $36K$ and CPU gets $24K$ tasks each. Further the programmer can specify another level of parallelism on each device. Essentially, the work-group sizes become 400 and 360 for the GPUs and chunk size becomes 240 for the CPU. Similarly Figure 3.19(b) shows an HDT for a cluster with two nodes and two GPUs devices each. If the total number of tasks are $200K$, both the nodes get $100K$ each and each GPU gets $50K$ each. We denote each node in the HDT as a *device place*. Figure 3.20 show a sample *H2C* program that uses an HDT.

```
1 HDT *topology = Input_HDT("input_top.xml");
2 forasync point(i) range(0:100000) at(topology){
3       A[i] = B[i] + C[i];
4 }
```

Figure 3.20 : H2C program with HDT

3.7 Summary

The contributions of this thesis chapter are as follows:

- Introduce Heterogeneous Habanero-C (*H2C*), a high-level programming language that can be used to program heterogeneous processors and achieve productivity, portability and performance.
- The highlights of *H2C* include high-level constructs to overlap communication and computation, task partitioning, data distributions and a unified event framework.
- The *H2C* compiler takes advantage of both AST and polyhedral optimizations to generate code tuned to a particular hardware.
- Evaluation of four benchmarks shows *H2C* to be portable, productive and also achieve performance similar to hand-coded low-level OpenCL implementations.
- Propose extension for *H2C* to target a distributed heterogeneous cluster.

Chapter 4

Data Layout for Heterogeneous Architectures

4.1 Introduction

An important aspect of heterogeneous systems is that different devices have different kinds of memory hierarchies. For example, NVIDIA GPUs have L1 and L2 caches that are connected to the system memory via PCIe whereas the integrated GPUs from Intel (e.g., Ivy Bridge and Haswell) have an L3 cache that is connected to the system memory on the same die with a last-level cache (LLC) that is shared between the CPU and GPU. On the host side, the CPU cores have memory hierarchy consisting of L1, L2, L3, and LLC. Recent studies [55–57] have shown that data layouts play a major role in determining application performance on both the CPU and GPU. Determining the optimal data layout, however, remains a challenging task since the performance of a data layout depends on factors such as (a) number of parallel hardware threads/contexts available; (b) memory hierarchy; (c) data access pattern in the program; (d) input size of the program. For example, CPU usually performs well with an Array-Of-Struct (AoS) layout because an AoS layout can help improve pre-fetching and cache sharing on CPUs. On the other hand, GPU performs well with a Struct-Of-Array (SoA) layout in general case since an SoA layout can improve the performance on GPUs due to coalescing of memory accesses. The GPU memory performance depends upon the number of coalesced accesses, whereas the host CPU memory performance depends on factors such as false sharing and data reuse. Hence,

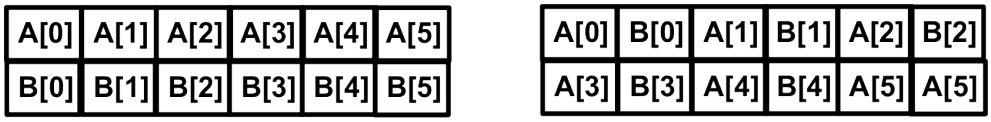


Figure 4.1 : SoA layout (left) and AoS layout (right) for arrays $A[0-5], B[0-5]$

the data layout impacts performance and is different for different architectures. Given the proliferation of device technologies on heterogeneous architectures and their differing memory hierarchies, it is best to provide the programmer a high-level framework to specify the data layout and leave the code generation to an optimizing compiler. However, none of the existing languages that target heterogeneous architectures provide mechanisms to specify the data layout. We believe that a compiler-driven data layout transformation framework can help bridge this gap. Figure 4.1 shows the AoS and SoA layouts for two arrays A and B of six elements each.

We present a meta-data framework that allows both programmers and tuning experts to specify architecture specific and domain-specific information for *parallel-for* loops of a program. A meta-data file is created for an application and is populated with entries on the data layout to be used for a device on the heterogeneous system. The data layout We focus on in this paper include structure-of-array (SOA) and array-of-structure (AOS). Any high-level language, which has *parallel-for* loops can be extended to accommodate the meta-data framework. In our work, we target the data-parallel **forasync** construct in *H2C* programming language and integrate our meta-data framework with the *H2C* compiler and runtime. The meta-data information is very useful in guiding our compiler optimization passes for the generation of efficient code for a device.

Using the metadata framework, the programmer can only specify a single data lay-

out to the entire program. However, in programs with multiple kernels, a single layout may not be optimal for the entire program. To manage the data layout automatically, we designed ADHA: a two-level compiler based automatic data layout framework and a reference implementation of the same in the Heterogeneous Habanero-C (*H2C*) programming system. The lower level formulation deals with the data layout problem for a parallel code region and provides a greedy algorithm that uses an *affinity graph* to obtain approximate solutions. The higher level formulation targets data layouts for the entire program, for which we provide a graph-based shortest path algorithm that uses the data layouts for the code regions computed in the lower level. The final data layout could be a single layout for the entire program or multiple layouts for different code regions with layout re-mapping in between kernels.

Overall, in this thesis chapter, we present a meta-data framework in *H2C* that allows both the programmer and the tuning expert to specify the underlying architecture and domain-specific knowledge for parallel-for loops; A compiler and runtime framework to automatically generate efficient code based on the meta-data information. We also introduce ADHA: a two-level compiler based automatic data layout framework and a reference implementation of the same in *H2C* programming system. The lower level formulation deals with the data layout problem for a parallel code region, and provides a greedy algorithm that uses an *affinity graph* to obtain approximate solutions. The higher level formulation targets data layouts for the entire program, for which we provide a graph-based shortest path algorithm that uses the data layouts for the code regions computed in the lower level.

We currently focus on AoS-to-SoA and SoA-to-AoS transformations in the *H2C* compiler. Note that an exponential AoS layouts are possible for a given number of fields.

arch_name –> Arch **name meta_data** **meta_data** –> (**struct_def**)∗ (**scratchpad_def**)∗ **struct_def** –> Struct **name** (**field_def**)∗ **scratchpad_def** –> Scratchpad **name** (**field_def tile_size line_num**)∗ **field_def** –> Field **type name** length **type** = fp | dp | ip length –> (**digit**)∗ **tile_size** –> (**digit**)∗ **line_num** –> (**digit**)∗ **name** = (**letter**)(**letter**|**digit**)∗ **letter** –> _ |A|B|C| . . . |Z|a|b|c| . . . |z| **digit** –> 1|2|3|4|5|6|7|8|9|0	Arch Intel_GPU **Struct** bodypos **Field fp** posx **Field fp** posy **Field fp** posz **Struct** bodyacc **Field fp** accx **Field fp** accy **Field fp** accz **Scratchpad Field fp** posx 256 **Scratchpad Field fp** posy 256 **Scratchpad Field fp** posz 256 Arch AMD_GPU **Struct** bodypos **Field fp** posx **Field fp** posy **Field fp** posz **Field fp** accx **Field fp** accy **Field fp** accz **Scratchpad Field fp** accx 1024

Figure 4.2 : Meta-data grammar (left) and Meta-data file example (right)

4.2 Meta-data Layout Framework

Our meta-data framework is built on top of Heterogeneous Habanero-C (*H2C*) compiler and runtime infrastructure. For each device on a heterogeneous system, it is possible to specify the desired data layout for an array-based or structure-based data structures of a given *forasync* loop. The data layouts that we focus on are: (1) AOS: array-of-structure; and (2) SOA: structure-of-array.

The grammar for the meta-data and an example is shown in Figure 4.2. The meta-data file consists of a set architecture specific optimization information. The architectural details consist of the data layout information and scratchpad memory allocation information for a given program. Each struct definition has a label *Struct*, a name for the struct and a set of fields. Each field in turn has a label *Field*, the type of the field and the name of the field. The type of fields can be *fp*: a pointer to an

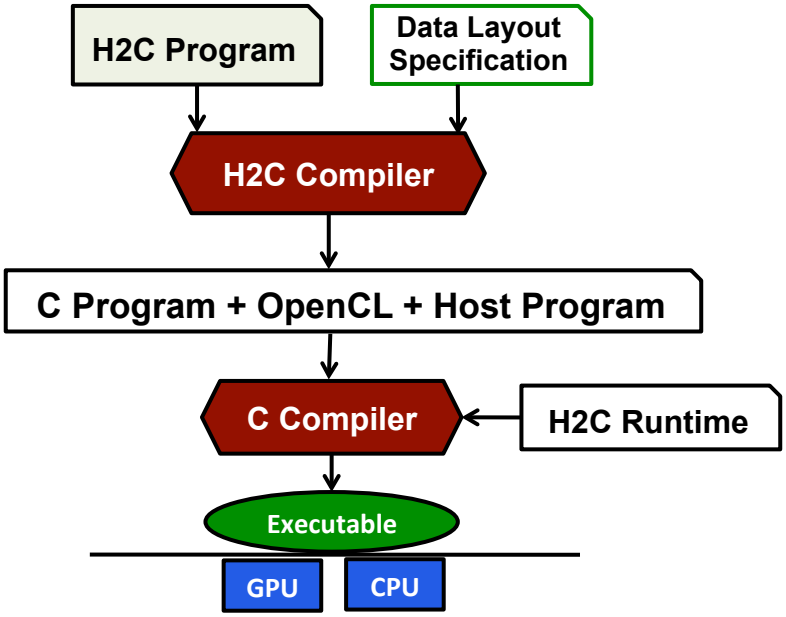


Figure 4.3 : Compilation flow of Meta-data framework

array of float values, *dp*: a pointer to an array of double values or *ip*: a pointer to an array of integer values. The scratchpad memory allocation information consists of a set of buffer descriptions. It begins with a label *Scratchpad*, the field type, the field name and the buffer size to be cached. Optionally, the programmer can choose to pass the scratchpad variables through the *scratchpad* clause of the *forasync* construct.

Our overall meta-data framework is shown in Figure 4.3. The application user writes a program in *H2C* using the *forasync* construct. Followed by which, either the developer or the tuning expert specifies the data layout specification for the application in a file. We extend the *H2C* compiler infrastructure to (1) perform data layout transformation based on the meta information; (2) generate OpenCL host and device code for a given meta-data specification. The original *H2C* program and the generated OpenCL code are linked together to provide a single executable, which runs

```
1  forasync point(i) range(0:N) at(gpu){
2      for(int j = 0 ; j < N; j++){
3          A[i] = (C[i] + B[j])/2;
4      }
5  }
```

(a) H2C program with re-use pattern

```
Arch Intel_CPU
  Struct AC
     Field fp a
     Field fp c
  Scratchpad Field fp b
```

(b) Meta-data file for Intel CPU

Figure 4.4 : H2C program + meta-data file with data layout specification

```
1  struct AC{float a, float c};
2  __kernel void kernel(__global struct AC *ac, __global float *B,int N){
3      int i = get_global_id(0),out_j,in_j;
4      int local_id = get_local_id(0);
5      __local float loc_B[LOCAL_SZ];
6      for(out_j=0; out_j < N/LOCAL_SZ;++out_j){
7          int j = out_j*LOCAL_SZ + local_id;
8          loc_B[local_id] = B[j];
9          barrier(CLK_LOCAL_MEM_FENCE);
10         for(in_j = 0;in_j < LOCAL_SZ;++in_j)
11                 ac[i].A = (ac[i].C + loc_B[j])/2;
12         barrier(CLK_LOCAL_MEM_FENCE);
13     }
14 }
```

Figure 4.5 : Generated OpenCL kernel with AoS layout specified in Figure 4.4 and re-use optimization of H2C

on the target architecture.

Figure 4.4 shows a sample *H2C* program and a sample meta-data file with data layout specification for an Intel CPU device. Figure 4.5 shows the corresponding OpenCL kernel code generated.

4.2.1 Data Layout Transformation

The compiler pass first parses the specified meta-data file, and it creates a meta-data map for each architecture. The mapping is between the fields and the struct name they belong to, and is done for each such struct meta-data information. If it finds any scratchpad meta-data information, it records them in the IR. The data layout transformation (DLT) compiler pass then generates the code based on the specified data layout in the meta-data file. It generates code that includes the new struct definitions and the code that operates on it.

Algorithm 3 shows the algorithm for transforming the program with a given data layout. `DLT` takes the input program and a meta-data file. `createStructDefinitions(M)` adds the struct definitions as specified in the meta-data file to the AST. These structs are defined only once in the global scope. The DLT pass then iterates over all the functions and performs the steps described in lines 3-7.

`tryAddStructInstances(f)` analyzes the function parameters. If any of the parameter names appear in the meta file, an instance of the corresponding struct is declared in the function scope. If we abstract the struct as a group of fields names, then one struct instance is declared per group. In next step, `updateInst(I)` checks all pointer or array references in the function body. If any of those references are via any of the fields in the meta-data file, then the access is replaced with the

Algorithm 3 Meta-data layout transformation

Input: Meta-data file M and input program P

Output: Transformed program P'

```
1: createStructDefinitions(M);
2: for each function F in P do
3:     for each formal f in function parameter list do
4:         tryAddStructInstances(f);
5:     end for
6: end for
7: for each instruction I in function body do
8:     updateInst(I);
9: end for
```

corresponding struct instance.

An important factor here is that the type of the function in the original program remains the same. Keeping the function types intact will avoid rewriting the direct and indirect calls to the function.

4.2.2 Memory Management

The memory allocation of *H2C* is described in section 3.3. We extend the *H2C* allocator to support our meta-data layout framework. The name of the field is passed as an additional argument to the allocator. The syntax of the extended memory allocator is shown below.

```
void *hc_meta_malloc(char *name, size_t sz, int dev_ct, int *dev_lst );
```

We implement a memory manager to handle the data layouts and device buffer management. The memory manager has two important components, the memory allocator and the layout handler. During the program initialization phase, the layout handler reads the meta-data file and creates a map of the data layout. The memory

manager with the help of the field name looks at the layout map and allocates the memory based on the following simple rules.

1. If the field does not belong to any struct layout in the meta-data file, it means that the programmer wishes it to retain the original layout.

2. If the field belongs to a struct layout group the the allocation happens as follows. Memory is allocated only once per struct group. If memory to the group has already been allocated, then a pointer to the chunk, offset by the field position is returned. If the memory is not allocated to the group, then memory for the whole struct group is allocated. The amount of memory chunk is equal to the number of fields times the number of bytes requested during the memory allocation. Then a pointer to the chunk, offset by the field position is returned.

Restrictions of our meta-data framework

The user cannot alias the fields specified in the meta-data file. We plan to resolve this issue with the help of an alias analysis. Another limitation in the programming model is that a variable name cannot be repeated in the whole program in different scopes. A clever variable renaming mechanism can overcome this limitation. Also, all fields in a struct must be of the same type. We currently do not support more complex data layouts such as AoSoA (Array-of-structure-of-arrays) and leave it for future work.

4.3 ADHA: Automatic Data layout framework for Heterogeneous Architectures

The metadata framework described earlier enables the programmer to specify only a single data layout for the entire program. However, in programs with multiple kernels, a single layout may not be optimal for the entire program. The best data layout could be a single layout for the entire program or different layouts for different parts of the program and data remapping between the parts.

4.3.1 Motivating Example

In this section, we consider a heterogeneous CPU+GPU architecture and show the performance impact of various data layouts. The example also illustrates the complexity and intricacies in selecting the best data layout for a given architecture. Let us consider a micro-benchmark with two data-parallel loops as illustrated in Figure 4.6. The first data-parallel kernel implements a stencil-like computation involving 5 arrays, x, y, z, w, & e, and the second kernel executes a simple multiply and add computation involving 3 arrays x, y, & e.

We use *H2C* forasync syntax for the data-parallel loops (details of *H2C* are given in Section 3.3). The clauses in the forasync loops are as follows: *index* specifies the loop's index variable, *range* describes the iteration domain (M = 10240×10240 in this example) and *at* specifies the target device ("NVIDIA Kepler K40C" in our case).

We execute the program on the NVIDIA GPU with two different layouts: AoS with x,y,z, & w in a structure and SoA where each of these fields are independent structures. Kernel-1 takes 5.3 msec with the AoS layout and 11.5 msec with SoA layout. Kernel-2 takes 8.4 msec with the AoS layout and 3.0 msec with SoA layout.

```
struct ABCD{float x; float y; float z;float w;};
float *x, *y, *z, *w, *e;
init(x, y, z, w, e);
//Kernel−1 on GPU with AoS layout: 5.3 msec
//Kernel−1 on GPU with SoA layout: 11.5 msec
forasync point(i) range(0:M) at(dev){
 if(.....){
   e[j] = ((x[j] + y[j] + z[j]) / w[j])
        + ((x[j+1] + y[j+1] + z[j+1]) / w[j+1])
        + ((x[j+2] + y[j+2] + z[j+2]) / w[j+2])
        + ((x[j+3] + y[j+3] + z[j+3]) / w[j+3]);
 }
}
//Remap from AoS to SoA: 3.3 msec
remap(xyzw, x, y, z, w);
//Kernel−2 on GPU with AoS layout: 8.4 msec
//Kernel−2 on GPU with SoA layout: 3.0 msec
forasync point(i) range(0:M) at(dev){
    x[j] = (y[j] + e[j] * 1.432);
}
```

Figure 4.6 : Microbenchmark in H2C. Best mapping is obtained when Kernel-1 executes with AoS layout, followed by data remapping from AoS to SoA and then Kernel-2 executes with SoA layout.

Remapping from the AoS layout to SoA layout takes 3.3 msec on the same machine. If the data layout choice is left to the programmer, the programmer will be forced to choose either of SoA or AoS. In that case, the best performance programmer can obtain would be 14.8 msec by choosing SoA. The optimal mapping is to execute the first kernel on the GPU with the AoS layout, and then remap the data layout from the AoS to the SoA and then execute the second kernel on the GPU with the SoA layout. The application now takes the best execution time of 11.9 msec resulting a speedup of 1.24. Therefore, a single data layout is not optimal in this case.

It is interesting to observe that while popular practice is to use a SoA to achieve coalesced memory accesses, we instead discover that AoS layout on GPU is more beneficial in this case. On the GPU, the AoS layout is specified using aligned structures such as *float2* and *float4* types. When we profiled the above code using an NVIDIA profiler [58], we observed that the compiler was generating **128-bit** loads for float4 types, **64-bit** loads for float2 types and **32-bit** loads for *float* types. The benefit from 128-bit loads comes from the fact that there are fewer instructions to issue (compared to 4 32-bit loads). Therefore, we noticed that as long as the fields are always accessed together, it is better to arrange them in an AoS layout, which was also observed in [59].

4.3.2 Problem Formulation

In this section, we formalize the optimal data layout problem and provide corresponding complexity results. The objective of an automatic data layout framework is to automatically determine the best data layout(s) for a given architecture and generate the corresponding executable. As illustrated in the previous section, due to the variations in data access patterns across code regions in a program, a single layout for the

entire program may not be always optimal. In the following subsections we propose a scheme that produces different data layouts for different parts of the program.

To assign different data layouts at different points in a program, we need a mechanism to partition the program. To this end, we treat data parallel kernels as the smallest unit of the program and partition the program into disjoint *sections*[*] and initially assign a single data-parallel kernel for each *section*. In our theoretical analysis, we assume all *sections* lie in a single control flow path (i.e. there are no branches). We use the superblock technique [60] to handle the case where there is control-flow between parallel *sections* of a program.

Let $\mathsf{S} = \{S_1, S_2, \cdots S_n\}$ be the set of *sections* for a program P. We denote the set of fields of P by F such that $F = \{f_1, ..., f_r\}$. To avoid notational clutter, we use *field* to refer to the fields in both AoS and SoA (which are actually arrays). Accordingly, the data layout $\mathsf{D} = \{d_1, d_2, \cdots d_n\}$ represent the corresponding data layouts of fields for each *section*. We assume that the set of fields in data layout d_i for *section* S_i is subset F^i over the fields.

Problem Statement

We use $\mathsf{Cf}(S_i, d_i)$ to denote the cost of executing *section* S_i with data layout d_i and $C(d_i, d_{i-1})$ to denote the cost to obtain data layout d_i from d_{i-1}. Finally, we formulate the optimal data layout problem as finding the data layout D for program P such that following is minimum.

$$\sum_{i=1}^{N} (\mathsf{Cf}(S_i, d_i) + C(d_i, d_{i-1}))$$

[*]We apologize to the reader for overloading the word "*section*". We henceforth use "Sec." refer to a Section in the chapter organization structure

Hierarchical Approach

The above formulation is similar to the formulations in previous related works and therefore, it is easy to extend past complexity results for High Performance Fortran [61] and show that the complexity of finding an optimal data layout is NP-hard [17, 61]. These approaches use expensive approaches such as Integer Linear Programming to determine the best layout. Previous formulations only provide complexity results and fails to provide more insight into designing an efficient algorithm. It is, therefore, important to ask if the problem can be formulated differently, which might provide better insight?

In this chapter, we answer the above question affirmatively and propose a novel two-level hierarchical formulation of the data layout problem. The bottom level formulation, Section Data Layout (*SDL*), deals with the data layout selection for a *section* based on interactions within a *section*. On the other hand, the top level formulation, Program Data Layout (*PDL*), takes in data layouts computed at the *SDL* level and computes the optimal data layout for the overall program.

The plan for the rest of this chapter is as follows: we first discuss *PDL* and prove that *PDL* can be computed in polynomial time. Then we move on to the bottom level and show that *SDL* is NP-hard. To address the intractability of *SDL* in practice, we propose a greedy algorithm that is later employed in our experiments.

Program Data Layout

The problem of Program Data Layout (*PDL*) is concerned with selecting of data layout for the entire program while considering inter-*section* interactions. *PDL* takes in the data layouts returned by *SDL* for each *section* and returns the data layout for the entire program.

The control flow (limited to structured control flow with a single-level nesting) among *sections* allows us to construct a directed acyclic graph with in-degree and out-degree of nodes restricted to at most one. We later describe the conversion of programs with loops into acyclic graphs. As discussed above, the data layout for a *section* can consist of fields accessed by its predecessors. To felicitate this, we introduce an operation combine_section that takes in optimal data layouts d_i, d_j for *sections* S_i, S_j such that S_j is successor of S_i and returns the data layout by merging d_i, d_j. We use $\mathsf{cost}(\mathsf{combine\_section}(d_i, d_j))$ to represent the cost of combine operation for data layouts d_i and d_j.

Another possible operation is remap_layout, which remaps the data layout from d_i to d_j. The cost for remap_layout is directly proportional to the number of fields between data layouts that are remapped. We use

$$\mathsf{cost}(d_1^f, d_2^i, d_2^f) = \begin{cases} \mathsf{cost}(\mathsf{combine\_section}(d_1^f, d_2^i)) & , \text{if}(d_2^f = \mathsf{combine\_section}(d_1^f, d_2^i)) \\ \mathsf{cost}(\mathsf{remap\_layout}(d_1^f, d_2^f)) & , otherwise \end{cases} \tag{4.1}$$

to denote the cost of transformation of d_2^i to d_2^f where d_1^f is the data layout of the preceding section. Therefore, using the notation introduced in Sec. 4.3.2 we have $C(d_1^f, d_2^f) = \mathsf{cost}(d_1^f, d_2^i, d_2^f)$.

We formulate the Program Data Layout (PDL) problem as follows: PDL takes in the set of data layouts $\{d_1^i, d_2^i, \cdots d_n^i\}$ computed from *SDL* and returns a set of data layouts $\{d_1^f, d_2^f, \cdots d_n^f\}$ such that $d_n^f = d_n^i$ or $d_n^f = \mathsf{combine\_section}(d_{n-1}^f, d_n^i)$ and the cost computed as $\sum_{i=1}^{n-1} C(d_i^f, d_{i+1}^f) + \sum_{i=1}^{n} \mathsf{Cf}(d_i^f, S_i)$ is minimum. For example, let $n = 4$ and we have *sections* S_1, S_2, S_3, S_4 and data layouts returned by *SDL* is $\{d_1^i, d_2^i, d_3^i, d_4^i\}$, where subscript i is used to denote the input to to *PDL* (We use superscript f to denote the "final" data layout returned by *PDL*). One possible final

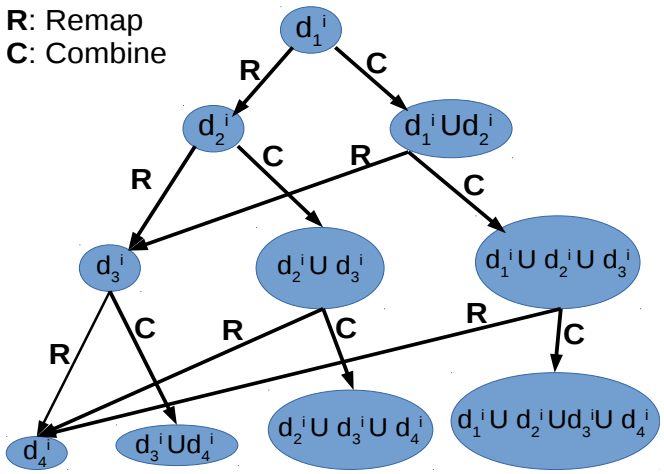


Figure 4.7 : Possible configurations for PDL

configuration is

$$d_1^f = d_1^i;$$

$$d_2^f = \mathsf{combine\_section}(d_1^i, d_2^i)$$

$$d_3^f = d_3^i$$

$$d_4^f = \mathsf{combine\_section}(d_3^i, d_4^i)\}$$

and the cost associated with it is $\mathsf{cost}(\mathsf{combine\_section}(d_1^i, d_2^i)) + \mathsf{cost}(\mathsf{remap\_layout}(d_2^f, d_3^i)) + \mathsf{cost}(\mathsf{remap\_layout}(d_3^i, d_4^i)) + \sum_{i=1}^{4} \mathsf{Cf}(S_i, d_i^f)$. Figure 4.7 illustrates all the possible configuration for this case. Note that there are only four different layouts possible for *section* 4: d_4^i, $\mathsf{combine\_section}(d_3^i, d_4^i)$, $\mathsf{combine\_section}(\mathsf{combine\_section}(d_2^i, d_3^i), d_4^i)$, $\mathsf{combine\_section}(\mathsf{combine\_section}(\mathsf{combine\_section}(d_1^i, d_2^i), d_3^i), d_4^i)$. We also note that every possible data layout can be specified by the last remap_layout operation. For example, in case of d_4^i, the last remap_layout was applied at the *section* 3 and for

combine_section(d_3^i, d_4^i), the last remap_layout operation was applied at the *section* 2. The following theorem presents the complexity analysis of PDL.

Theorem 4.3.1 PDL is in PTIME.

Proof To prove PDL is in PTIME, we reduce PDL to finding the shortest path over a graph. To this end, we construct a DAG for every $G = (V, E)$ where a node represents a possible data layout for a Section. We call this DAG the PDL-DAG. From above we know that for *section* S_i there are only i possible data layouts. In our DAG, an edge represents either combine_section or remap_layout operation. Let $D_{i,j}(i > j)$ represent the final data layout for *section* i obtained such that the last remap_layout operation was at *section* j. Also, we obtain $D_{i+1,j}$ and $D_{i+1,i}$ by applying combine_section and remap_layout operations respectively. Therefore in our DAG G, $V = \{D_{i,j|0\leq j<i<n}\} \cup D_{\text{dest}}$, where n is the total number of *sections* and D_{dest} is an extra node we introduce for technical reasons explained later. We construct all the combine_section and remap_layout edges such that the weight of combine_section edge $(D_{i,j}, D_{i+1,j})$ is sum of the cost of combine_section edge and $\mathsf{Cf}(D_{i+1,j}, S_{i+1})$. The edges from $D_{n,j|0<j<n}$ to D_{dest} are added with weight 0. Therefore, $E = \{(D_{i,j}, D_{i+1,j})\} \cup \{(D_{i,j}, D_{i+1,i})\} \cup \{(D_{n,j}, D_{\text{dest}})\}$ for $0 \leq j < i < n$. With this formulation, the problem PDL reduces to finding the shortest (weighted) path from $D_{1,0}$ to D_{dest}. The shortest path for this graph can be computed in $\mathcal{O}(|E|+|V|\log|V|)$.

We now compute the cardinalities of sets V and E. For *section* S_i we have i nodes in G. Therefore summing up all the nodes and adding 1 for D_{dest} node we have $|V| = 1 + \sum_{i=1}^{n} i = 1 + n(n+1)/2$. Also, for every node $D_{i,j}(i < n)$, we have 2 outgoing edges and for nodes $D_{n,j}$ we have one outgoing edges. Thus summing up all the edges, we have $|E| = n + \sum_{i=1}^{n-1}(2 * i) = \mathcal{O}(n^2)$. Therefore, the shortest path

for G can be computed in $\mathcal{O}(n^2 + n^2 \log n) \in \mathcal{O}(n^2 \log n)$. Hence, the problem *PDL* can be computed in PTIME.

Section Data Layout

The objective of *SDL* is to find the optimal data layout for a given *section* considering only Array of Structure (AoS) and Structure of Array (SoA) layouts. In any instance of a data layout, there is a single SoA but multiple AoS possible.

Lemma 4.3.2 The number of possible data layouts D_i for a section *S_i with n fields follows the Bell number*

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_n \tag{4.2}$$

$$B_1 = 1 \tag{4.3}$$

Proof The number of candidate layouts ψ, is based on the number of fields F^i in that section S_i. Now the fields in F^i can be arranged into different structs or independent arrays. Let us assume each array as struct of size 1. If we assume each struct to be a set, then the number of layouts is nothing but the number of ways n elements can be partitioned is exactly the **partition set problem**. The number of partitions in the partition set problem follows the Bell number.

Figure 4.8 show an instance of the data layout possible for a *section*, which uses 7 fields $\{*a, *b, *c, *d, *e, *f, *g\}$. Based on the code and the target architecture, affinity values are associated with every pair of fields. The computation of affinity values is discussed in detail in Sec 4.3.3. The fields and the affinity values can be represented as a weighted complete graph $G_{\text{cluster}} = (V, E)$, where $V = F$ and $(v_1, v_2) \in E$ for every $v_1, v_2 \in V$. Let $\mathsf{W}(e) \in N$ denote the weight of edge e and $\mathsf{W}(G = (V, E))$ denote

```
struct SOA{float *a; float *b;}ab;
struct AOS1{float c;float d;}cd[100];
struct AOS2{float e;float f;float g;}efg[100];
```

Figure 4.8 : Example data layout instance

sum of weights for all the edges $e \in E$. An optimal data layout would combine fields into cluster such that the sum of weights of inter-cluster edges would be minimum, therefore sum of weights of clusters edges to be maximum. This stems from the observation that sum of weights of inter-cluster edges is proportional to cache misses. Due to factors such as pre-fetch size, the size of every cluster is bounded to a given constant, henceforth denoted as k. Therefore, optimal data layout problem for a section, denoted as *SDL*, can be formulated as follows:

SDL(G, k): Given a weighted complete graph $G_{\text{cluster}} = (V, E)$ with integer weights, find a partition $OC = \{C_1, C_2, ...C_i\}$ such that $|C_i| < k$ and $\sum \mathsf{W}(C_i)$ is maximum.

The following decision problem formulation, denoted as SDLD, comes handy in analyzing complexity of *SDL*.

SDLD(G, k, c): Given a weighted complete graph $G_{\text{cluster}} = (V, E)$ with integer weights, does there exist a partition $OC = \{C_1, C_2, \cdots C_i\}$ such that $|C_i| < k$ and $\sum \mathsf{W}(C_i) = c$.

Our complexity analysis of *SDL* and SDLD uses the reduction from following problem, denoted as PART.

PART: Given a Graph $G = (V, E)$ with $|V| = 3q$ for some integer q, can G be partitioned into q disjoint sets $V_1, V_2, \cdots, V_n$, each containing exactly 3 vertices such that

for each $V_i = u_i, v_i, w_i$, $1 \leq i \leq q$, all three of the edges $\{u_i, v_i\}, \{u_i, w_i\}$ and $\{v_i, w_i\}$ belong to E.

The following theorem provides the complexity analysis of PART (stated as GT11 in [62]).

Theorem 4.3.3 PART *is* NP*-complete.*

The following theorem provides the complexity analysis of *SDL*.

Theorem 4.3.4 SDL(G, k) *is* NP*-hard.*

Proof We prove $SDL(G, k)$ is NP-hard by proving that SDLD(G, k, c) is NP-Complete for $k = 3$.

To prove SDLD is NP-complete for $k = 3$, we reduce PART to SDLDas follows: For a given PART instance $G_p = (V, E)$ with $|V| = 3q$, we construct a complete graph $G_c = (V', E')$ such that $V = V'$ and for every $v_i, v_j \in V', (v_i, v_j) \in E'$. We assign weights for edges in G_c as follows: for $(v_i, v_j) \in E, \mathsf{W}((v_i, v_j)) = 1$, otherwise 0. Now we claim that for G_c and $k = 3$, there exists a partition $OC = \{C_1, C_2, \cdots C_n\}$ with $\sum \mathsf{W}(C_i) = 3q$ iff G_p can be partitioned into q disjoint sets. If G_p can be partitioned into q disjoint sets, then weight of each subset is 3, so the total weight is $3q$. The proof is now completed by proving the "only if" part of the claim below by contradiction.

We assume that there exists a partition OC such that $\sum \mathsf{W}(C_i) = 3q$ and G_p can not be partitioned into q disjoint sets. The set of clusters in OC can be divided into six categories: (1) clusters with 3 vertices and all the edges of the cluster belong to E (2) clusters with 3 vertices and two edges belong to E, (3) clusters with 3 vertices and one edge belong to E, (4) clusters with three vertices and no edge belongs to E, (5) cluster with two vertices and the edge belongs to E and (6) cluster with two

vertices and the edge does not belong to E. Let x, y, z, u, v, w denote the number of clusters in the order described above. Since the total number of vertices is $3q$, we have $3x+3y+3z+3u+2v+2w = 3q$. Since G_p can not be partitioned into q disjoint sets, we have $y + z + u + v + w > 0$. Summing up the weights contributed by each type of cluster, we have $3x + 2y + z + v + w$. Since $x < q$ and $y + z + u + v + w > 0$, $3x + 2y + z + v + w < 3q$, which is a contradiction to our assumption. Hence, there exists a partition $OC = \{C_1, C_2, \cdots C_n\}$ with $\sum \mathsf{W}(C_i) = 3q$ iff G_p can be partitioned into q disjoint sets.

Remark: $SDL(G, k)$ is PTIME computable for k=2. In this case, the problem can be reduced to minimum edge weight cover set problem, which can be computed in PTIME [63]).

Greedy Strategy

While the NP-hardness of *SDL* motivates us to ask if approximation to *SDL* is any easier, the complexity analysis of approximation to *SDL* is beyond the scope of this thesis and requires a further study. We instead propose an algorithm, SGML, based on greedy-heuristic strategies. On a high level, the algorithm sorts the edges according to their weights and has flavor of the union-find algorithm. The pseudo-code for the algorithm is presented in Algorithm 4. SGML takes in two parameters as input: an affinity graph $G = (V, E)$ and an integer k, which bounds the maximum size of a cluster. SGML assumes access to three subroutines: (1) CreateNewCluster takes as input a pair of two nodes (u, v) and returns a new cluster that contains u and v, (2) AddToCluster takes as inputs a cluster c_u and a node v and adds node v to the cluster c_u, (3) MergeClusters takes as inputs two cluster c_u and c_v, and merges cluster c_v into c_u. SGML chooses the edges in decreasing order of their weights. For every edge (u, v)

Algorithm 4 Determine clustering

Input: $G = (V, E)$: Affinity graph , k: Maximum cluster size
Output: Clustering C

```
1: E^S ← WeightSorted(E)
2: C = {}
3: for edge e = (u, v) in E^S do
4:     c_u = FindCluster(u); c_v = FindCluster(v)
5:     if (c_u == NULL && c_v == NULL ) then
6:         c = CreateNewCluster(u, v); C = C ∪ c
7:     else if (c_v == NULL && |c_u| ≤ k − 2) then
8:         AddToCluster(c_u, v))
9:     else if (c_u == NULL && |c_v| ≤ k − 2) then
10:        AddToCluster(c_v, u)
11:    else if (c_u! = c_v && |c_u| + |c_v| < k) then
12:        MergeClusters(c_u, c_v)
13:    end if
14: end for
15: return C
```

chosen, there are five possibilities: (1) u and v do not belong to any of the clusters: in this case, a new cluster with the vertices u and v is created, (2) u does not belong to any cluster and the size of cluster for $v(c_v)$ is less than k-1: in this case, we add u to c_v, (3) v does not belong to any cluster and the size of the cluster for $u(c_u)$ is less than $k-1$: in this case, we add u to c_v, (4) u and v belong to different clusters (c_u and c_v respectively) such that $|c_u| + |c_v| < k$, in this case we merge clusters c_u and c_v and (5) for cases not covered above, we ignore the edge and proceed to the next edge.

4.3.3 ADHA Implementation

We discuss the implementation details of our automatic data layout framework in the Heterogeneous Habanero-C (*H2C*) programming system. The overall automatic

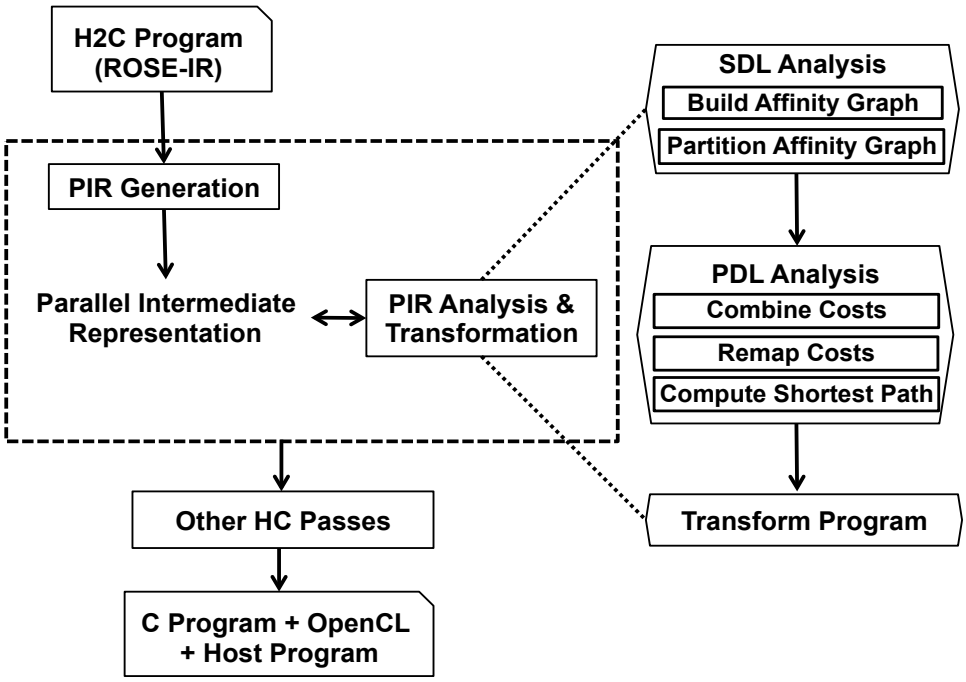


Figure 4.9 : Compiler framework for automatic data layout

data layout framework consists of a set of analysis passes followed by the data layout transformation pass. We also describe the details of how affinity graphs are constructed including how remapping of data layout(remap_layout) and combining of sections(combine_section) costs are computed for a *H2C* program. Since data layout impacts only data-parallel kernels that target various devices, we only consider forasync and finish constructs of *H2C* in this work.

Figure 4.9 shows a diagrammatic description of our data layout transformation framework. From ROSE IR, we generate the parallel intermediate representation (*PIR*) [64]. Once the *PIR* is constructed, we perform data layout analysis for each data-parallel section (*SDL*). During *SDL* analysis, we build an *affinity graph* for each *section* and then employ the algorithm SGML described in Sec. 4.3.2 to partition the

affinity graph. Subsequently, we perform data layout (*PDL*) analysis for the entire program. During this phase, we compute the remap_layout and combine_section costs for kernels and then apply the shortest path algorithm described in Section 4.3.2 to obtain the best data layout.

Finally, the program is transformed to use the data layout determined above. The placement of the remap operations is done carefully using code motion techniques described in [65]. We later discuss the construction of *PIR*, affinity graph, and computation of remap_layout/combine_section costs in more detail.

Handling Loops in PDL

The *PDL* pass requires the program control flow to be a DAG. Cycles are introduced if the *sections* in a program are involved in a loop. Loops complicate the layout selection because the layout now depends on two *sections*, one from the forward edge and the other from the backward edge. We handle this by peeling the first iteration and last iteration of the loop. We now have a program structure where the forward and backward edge come from the same code block. We can now ignore the backward edge. The resultant graph is now acyclic. The loop is further unrolled L times, where L is the number of sections in the loop to obtain the "steady state" optimal data layout for the remaining loop iterations.

Figure 4.10 describes how structured *loops* involving *sections* are handled. The left side shows a program control flow where sections S_1, S_2, S_3 are in a loop. The layout of S_1 is now dependent on S_0(forward edge) and S_3(backward edge). We now peel the first and last iteration of the loop. We also unroll the loop three times to determine the "steady state" data layout for the remaining loop iterations.

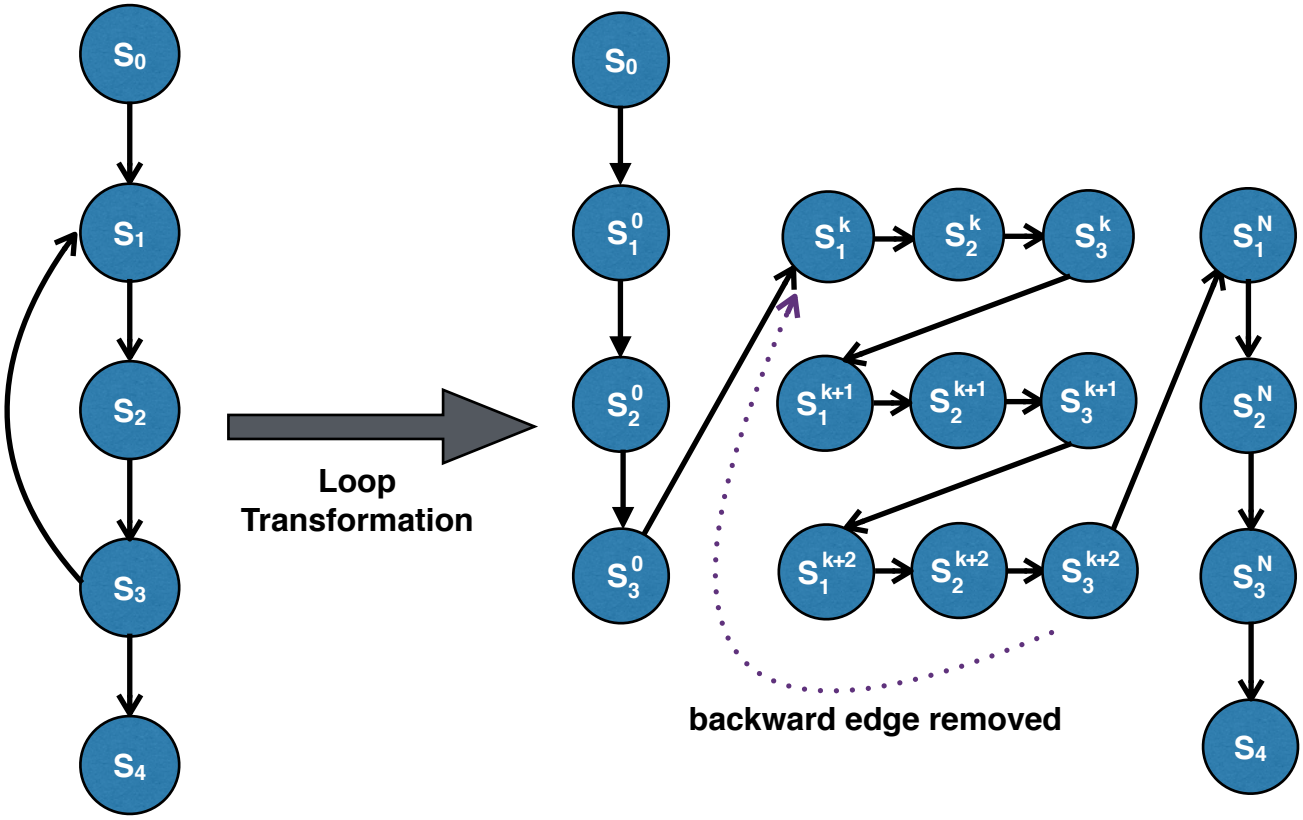


Figure 4.10 : PIR control flow transformation

PIR

The *PIR* is a common intermediate language for explicitly-parallel programs such as *H2C*. For every function in a program, the *PIR* for that method consists of three key data structures: 1) a Region Structure Tree (*RST*); 2) a set of Region Control Flow Graphs (*RCFG*); and 3) a set of Region Dictionaries (*RD*). The *RST* represents the region nesting structure of the method being compiled, analogous to the Loop

Structure Tree (LST) introduced in [66]. Each region in the *RST* has an associated control flow graph (*RCFG*) that encapsulates control flow for the immediate children of the region. Additionally, each region stores summary information, such as upwards-exposed uses and downwards-exposed defs, in an associated dictionary (*RD*).

For *H2C*, the single-entry regions considered in this work include `FINISH`, `FORASYNC`, and loop regions. Two special empty regions `START` and `END` are added to designate the start and end of a function. The other IR nodes considered in the *RCFG* are array load `ALOAD`, array store `ASTORE`, object field load `FLOAD`, and object field store `FSTORE`.

Affinity Graph Construction

The affinity graph construction is an important component of our framework that captures how close a group of data items are accessed together in the program. We build the affinity graph for each *section*. The affinity graph is a weighted undirected graph where the nodes represent individual data items (a statement of the form `ALOAD`, `ASTORE`, `FLOAD`, `FSTORE`) and edges represent the co-access pattern of two data items. The weight on an edge reflects the frequency of accessing them together and also the amount of memory accessed in between them. Following past approaches for static cost estimation, the frequency of array access inside a loop-nest is estimated as 10^d, where d denotes loop depth.

To reduce the size of the resulting affinity graph, the body of a *section* is heavily optimized before the construction of the affinity graph. In particular, scalar replacement is performed aggressively to eliminate accesses to $a[i-1]$ where a prior iteration loads $a[i]$ with no killing dependency in between them in a loop region. Similarly, variable renaming is performed in such a way that loops iterating over the same it-

eration space (exactly same lower and upper bounds) are assigned the same index variable name.

For *sections* consisting of accesses to both arrays and object fields, we build two separate affinity graphs: one focusing on arrays and another focusing on object fields. Note that the affinity graph for arrays must capture information about the amount of memory needed by the object fields accessed in between and vice versa. This information is conservatively computed. For the rest of the discussion, we will only focus on building the affinity graph for array accesses.

We now describe a flow-insensitive algorithm to build affinity graph as shown in Algorithm 5. We start by scanning a basic block from top to bottom. If we visit an `ALOAD` $a[i]$ or `ASTORE` $a[i]$ instruction, we create a node for $a[i]$, if it is not there already in the affinity graph. We count the number of memory accesses, mem_usage$(a[i], b[i])$, from the previous `ALOAD` $b[i]$ or `ASTORE` $b[i]$ instruction (takes into account object field accesses). We add an edge between $a[i]$ to $b[i]$ with the edge weight $w(e(a[i], b[i]))$ as:

$$w(e(a[i], b[i])) = \begin{cases} 0 & , \text{if}(\text{mem}(a[i], b[i]) > \text{cache\_size}) \\ \text{freq}(B) * \frac{1}{log_2(\text{mem}(a[i], b[i]))} & , otherwise \end{cases} \tag{4.4}$$

where freq(B) denote the frequency of basic block B. If the memory usage, $mem(a[i], b[i])$ is greater than the cache size, then we assign 0 as weight indicating there is no point combining them. Otherwise, the weight is computed as the product of the basic block frequency and the inverse of the logarithm of the memory usage. It is important to emphasize the freq(B) component since frequently executed blocks will contribute significantly to the overall data layout. If the edge already exists, we accumulate the edge weights to account for aggregated frequency counts.

Algorithm 5 Affinity graph construction from a parallel section

Input: *PIR* for the parallel section

Output: $G(V, E)$:: graph with node set V and edge set E for the parallel section

```
1: V := {}; E := {};
2: for each loop region L in PIR do
3:     for each basic block B in the RCFG(L) do
4:         mem := 0;
5:         prev_I := {};
6:         for each instruction I in B do
7:             if I is an FLOAD a.f or FSTORE a.f then
8:                 mem += sizeof(a.f);
9:             end if
10:            if I is an ALOAD a[i] or ASTORE a[i] then
11:                Create a node for a[i], if not already in V;
12:                if prev_I is of the form ALOAD b[i] or ASTORE b[i] then
13:                    Add an edge e between nodes for a[i] and b[i], if not present;
14:                    Assign/Update edge weight, w(e) using the Eq. 4.4;
15:                end if
16:                prev_I := I;
17:                mem := 0;
18:            end if
19:        end for
20:    end for
21: end for
22: return
```

Remap Layout Cost Estimation

The remap_layout cost estimation not only depends on the amount of data being remapped but also depends on the type of remapping used. Different types of remapping operations are:

- *Local Data Remapping (LDR)*: remaps the data in blocks.

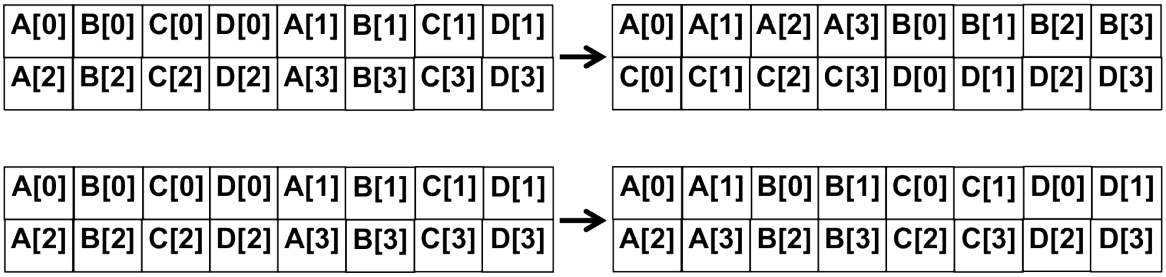


Figure 4.11 : Global data remapping (Top), Local data remapping (Bottom)

- *Out-of-place Global Data Remapping (OGDR)*: remaps the entire data from one data layout to another but uses an additional buffer.
- *In-place Global Data Remapping (IGDR)*: remaps the entire data from one data layout to another without any additional buffer.

Although IGDR saves space, it is computationally inefficient as it requires several synchronization operations when performed in parallel. In contrast, OGDR does not require any synchronization. We focus on OGDR and LDR remappings for the rest of the discussion. OGDR transforms the entire data from AoS to SoA. LDR on the other hand regroups the data to a local SoA data layout in blocks.

Figure 4.11 demonstrates how LDR and OGDR are constructed with the help of four arrays, $a[0:3]$, $b[0:3]$, $c[0:3]$, and $d[0:3]$. The data layout on the left-hand side is in AoS. The top-right shows the GDR version of SoA whereas the bottom-right shows the LDR version of SoA (uses a block size of 2): two elements of arrays a, b, c, d are mapped to SoA layout followed by the remaining two elements in each array.

A remap_layout operation can be parallelized to reduce its impact on execution time. We empirically determine the remap_layout cost for LDR and OGDR with the help of micro-benchmarks on a given hardware platform (this operation is performed

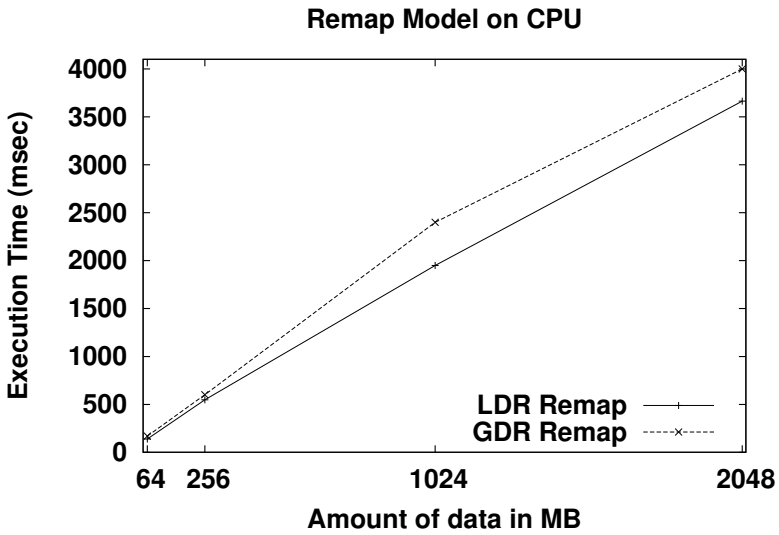


Figure 4.12 : Remapping costs on an Intel Xeon CPU

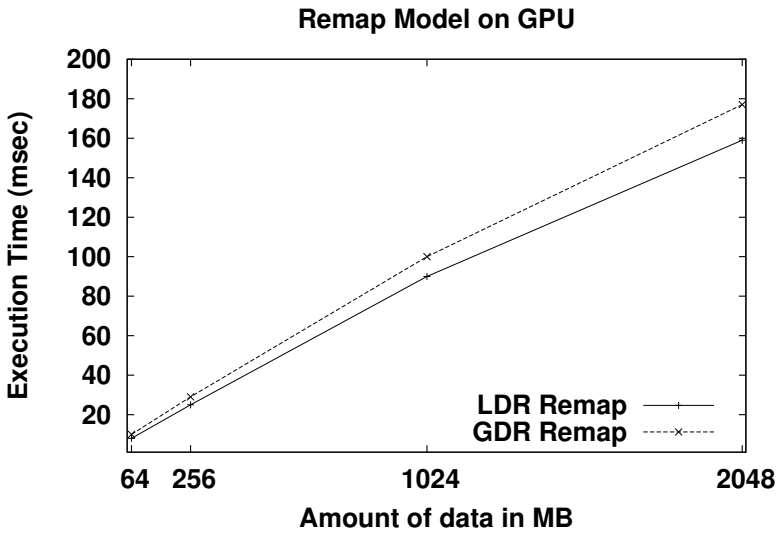


Figure 4.13 : Remapping costs on an NVIDIA Tesla GPU

once per platform and stored in a table).

Figure 4.12 depicts the data remapping costs on a Tesla M1050 GPU and Figure 4.13 depicts the remapping costs on an Intel Xeon CPU. On the X-axis we use the amount of data being remapped. The charts show that it is always beneficial to

perform remapping on the GPU as opposed to the CPU. Additionally, LDR is always faster than OGDR on both the CPU and the GPU. This is because LDR benefits from data locality on the CPU where as on the GPU, it performs remapping by taking advantage of scratchpad memory and *local barrier*. On the other hand, LDR is feasible only when the same partition of data items are remapped across multiple kernels. In our evaluation, by default we use LDR to remap the data except for the case where two consecutive kernels remap data from different partitions (as computed using SGML algorithm), at which point we switch to OGDR.

Algorithm 6 Compute remap cost

```
1:  procedure RemapSections(S1: section, S2: section)
2:      fieldsize ← 0
3:      for f ϵ S1.Fields do //for each field or array
4:          if f.getLayout(S2)==NULL then //if f is not accessed in another
5:              continue;
6:          end if
7:          if f.getLayout(S1) neq f.getLayout(S2) then // the layouts are different
8:              // combine the frequency of the basic block containing the field or array
9:              fieldsize+ = f.size ∗ freq(basicblock(f));
10:         end if
11:     end for
12:     return remap_model(fieldsize) using the Figures 4.12, 4.13;
13: end procedure
```

Algorithm 6 presents the remap_layout cost estimation. It takes two parallel *sections* as input and outputs the estimated cost of remapping. The algorithm it-

erates over the fields (both object fields and array accesses) in both *sections*, checks if a field appears in only one of the *section*'s data layout (and not in both) and accordingly updates the counter fieldsize, which counts the amount of data that needs to be remapped. This value is passed to the remap_model (as shown in Figures 4.12 for CPU, 4.13 for GPU) which then returns the cost of remap.

Combine Sections Cost Estimation

The combine_section cost is estimated as the loss in performance by assigning the same data layout for two *sections* instead of the previously assigned individual data layouts. If the layouts of both the *sections* are the same, then the combine cost is 0. If the layouts are different, then an intermediate layout $DL12$ is obtained by combining the two sections, $S1$ and $S2$, and running the SGML algorithm on the combined affinity graph $S12$.

The combine cost is the predicted performance loss and is the sum of difference between running the *sections* with the original layouts $DL1, DL2$ compared to running them using the new layout $DL12$. The pseudo-code for the procedure CombineSections is presented in Algorithm 7.

In Algorithm 7, we build a $Perf\_model$ function that takes a *section* $S1$ and a combined data layout $DL12$. It then uses the *combine_model* to return the estimated cost. The *combine_model* is determined using a set of micro-benchmarks mimicking different kernel characteristics. We classify a kernel into either compute-bound or memory-bound. A kernel is classified statically as compute-bound if the ratio of the compute instruction to the total number of instructions is greater than a threshold (0.6 used in our evaluation), otherwise it is memory-bound. The *combine_model* takes two layouts, the data size (computed similar to Algorithm 6), the memory-

Algorithm 7 Compute combine cost

```
1: procedure CombineSections(S1:a parallel section, S2:a parallel section)
2:     // Merge affinity graphs for S1 and S2, and perform partitioning using SGML
   algorithm
3:     DL12 = SGML(merge(S1.affinity_graph,S2.affinity_graph))
4:     // Find the cost of executing S1 using the combined layout DL12
5:     cost1 = PERF_MODEL(S1,DL12)
6:     // Find the cost of executing S2 using the combined layout DL12
7:     cost2 = PERF_MODEL(S2,DL12)
8:     // return the sum of the costs
9:     return (cost1 + cost2);
10: end procedure

1: procedure Perf_model(S1, DL12)
2:     // classify S1 to memory bound or compute-bound
3:     T = classify_kernel(S1);
4:     combine_cost ← 0
5:     for f ← S1.Fields do // for all field accesses and arrays
6:         D1 = f.getLayout(S1);// find the current layout of f in S1
7:         D2 = DL12.getLayout(f); // find the current layout of f in DL12
8:         if  D1 neq D2  then
9:             combine_cost+=combine_model(S1.datasize,D1,D2,T);
10:        end if
11:    end for
12:    return combine_cost
13: end procedure
```

boundedness of the kernel and returns the performance loss. It is possible that the two affinity graphs cannot be combined due a conflicting affinity value between two fields. In such a case, we use the default layout specified by the programmer.

We wrote a memory-bound micro-benchmark that randomly updates memory locations in a loop inside a kernel. We run this micro-benchmark for varying amount

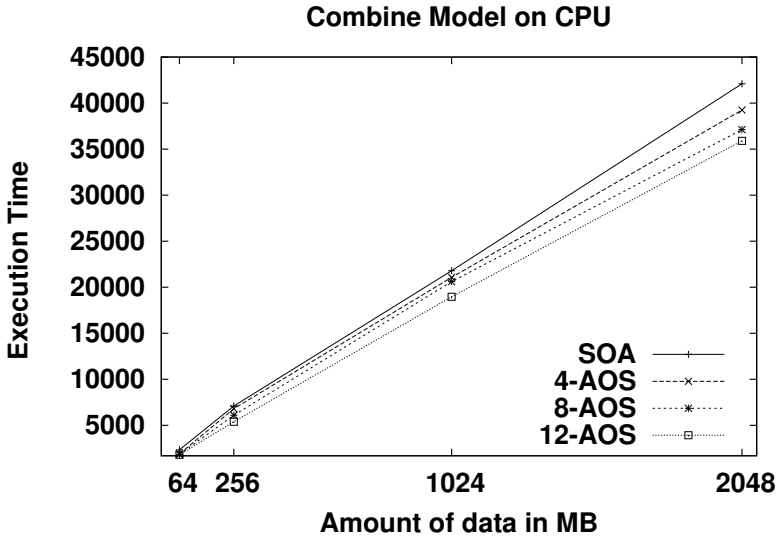


Figure 4.14 : Combine cost model on an Intel Xeon CPU for a memory-bound kernel with varying partition size

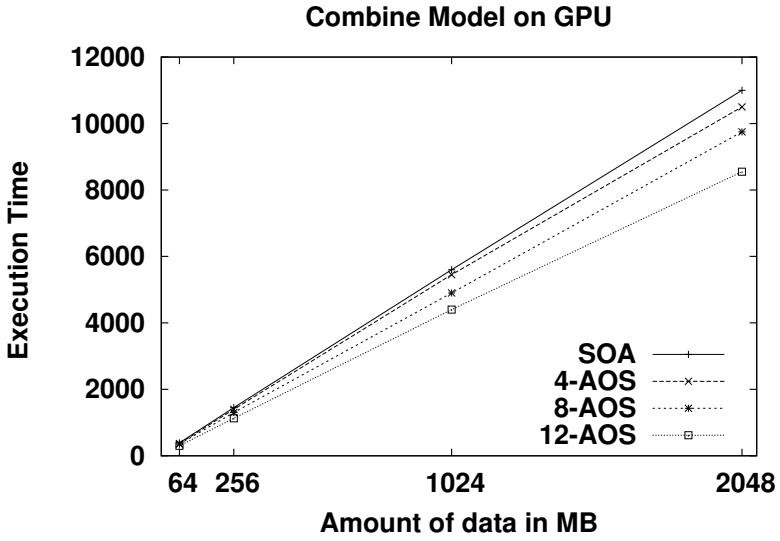


Figure 4.15 : Combine cost model on an NVIDIA Tesla GPU for a memory-bound kernel with varying partition size

of data and different partition sizes. Figure 4.14 shows the effect of the data layout on a CPU and Figure 4.15 shows the effect of the data layout on a GPU for a memory

bound kernel. The x-axis represents the total amount of data being accessed inside the kernel, and the y-axis represents the execution time in milliseconds. Each curve represents the execution time for different partition sizes varying from 1 to 12 in this graph. The effect of data layout on a CPU or GPU becomes prominent when the amount of data being accessed increases. We use this curve to determine the *combine_cost* in Algorithm 7.

Our implementation of ADHA automatically compiles forasync loops down to OpenCL with the corresponding data layout output by the *SDL* + *PDL* pass. ADHA can be employed to efficiently run a H2C program on modern CPU+GPU platforms that support OpenCL.

4.4 Evaluation

The goal of the experimental evaluation is to prove our meta-data framework's ability to extract maximum performance from a given architecture. We compare the impact of data layout on each benchmark on GPUs and multi-core CPUs.

4.4.1 Experimental Setup

Table 4.1 describes the benchmarks used in this evaluation. We chose a set of applications whose performance will be most impacted by data layout transformations.

The *Medical* Imaging benchmark includes phases from a medical imaging pipeline used to analyze different types of medical images for defects or abnormalities [67]. This application consists of three main phases: *denoising*, *registration*, and *segmentation*. For our evaluation, we focus on the most computationally significant phase of the three, *registration*. The registration phase consists of seven kernels and six fields.

The Lattice Boltzmann Method (*LBM*) simulation benchmark was provided to

Name	Description	Original Layout	Num of Kernels	Num of Fields	Input
Medical	Medical Image Registration	SoA	7	6	256×256×256
LBM	CFD Simulation	SOA	2	19	300×300×300
NBody	Molecular Dynamics	SOA	2	10	10000
K-Means	Clustering Algorithm	SOA	2	16	8388608
Seismic	Seismic Wave Simulation	SOA	2	6	4096 × 4096
SRAD	Speckle Reducing Anisotropic Diffusion	SOA	2	4	4096 × 4096
MRIQ	Matrix Q for 3D Magnetic Resonance Imaging	SOA	1	6	64 × 64 × 64
GESUMMV	Linear Algebra Kernel	SOA	1	5	10000
GEMVER	Linear Algebra Kernel	SOA	1	9	10240
SYR2K	Linear Algebra Kernel	SOA	1	4	2048×2048

Table 4.1 : Compile-time statistics for the benchmarks used in the evaluation.

us by Halliburton Services. A related benchmark is also available in the Parboil benchmark suite [51]. It is a computational fluid dynamics simulator. It applies a set of collision and propagation operations on the lattice points. The benchmark uses nineteen fields and has two kernels.

The *NBody* particle simulation benchmark was written from scratch for this work. A sample program is available in the TBB benchmarks [21]. It has two kernels: *force update* and *velocity update*. The NBody application uses a total of ten fields.

The *K-Means* benchmark is a clustering workload from the Rodinia benchmark suite. The benchmarks consists of two kernels: the first kernel is a parallel loop, while the second kernel purely performs a reduction over all the features. The second kernel

is executed sequentially in the original OpenMP version of the benchmark. Since our current implementation does not support reduction, we port this loop as a sequential loop by employing the *s*eq clause in `forasync` construct. The number of fields is equal to the number of features, which is sixteen in our case.

The *Seismic* benchmark suite was created based on the example included in the Intel TBB benchmark suite [21]. Seismic simulates the propagation of waves during seismic activity. The benchmark uses six fields and has two kernels.

The *SRAD* benchmark from the Rodinia benchmark suite [68] is also used. SRAD is used to "remove locally correlated noise" in "ultrasonic and radar imaging applications based on partial differential equations". SRAD has two kernels and uses a total of four fields in the main data structure *N, S, E, W*.

The *MRIQ* benchmark from the Parboil benchmark suite [51] computes a Q matrix. The Q matrix represents the scanner configuration used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space. The MRIQ code has been converted to SOA layout by hand. The benchmark uses six fields and has a single kernel.

GESUMMV, *GEMVER*, and *SYR2K* are linear algebra kernels from the Polybench benchmark suite [69]. They have one kernel with five, nine and four fields respectively.

Table 4.2 lists the hardware architectures used in our evaluation. We use a variety of CPU and GPU systems with differing memory hierarchies in order to demonstrate the benefit of our data layout transformation. The compiler used for the sequential versions of each application GCC 4.4.6 (with the flags `-g -O2`). All OpenCL kernels were compiled with their default optimizations enabled. Intel GPU tests were run using the 2013 Release of the Intel OpenCL SDK [70]. Intel CPU tests were performed using 2011 Release of Intel OpenCL SDK, v1.5 [70]. NVIDIA GPU tests were

Vendor	Type	Model	Freq (GHz)	Cores	Local Mem(KB)	L1$ (KB)	L2$ (MB)
Intel	CPU	X5660	2.8	6	N.A	192	1.5
Intel	IGPU	i7-3770U	1.1	14	64	N.A	N.A
NVIDIA	DGPU	Tesla M2050	0.6	8	8x48	16	0.8
AMD	CPU	A10-5800K	1.4	2	N.A.	16	32
AMD	IGPU	HD 7660	0.8	6	6x32	N.A	4

Table 4.2 : Hardware architectures. IGPU: Integrated GPU, DGPU: Discrete GPU

performed using NVIDIA SDK v5.0 [71]. AMD GPU and GPU tests were performed using AMD APP SDK v2.8 [72].

Table 4.3 shows the different data layouts used for each benchmark. The fields without any curly braces belong to the SoA layout. All the fields within a curly brace belong to a AoS layout. For the meta-data layout framework, we specify meta-data files corresponding to each layout. All OpenCL kernels, glue code, and different layouts for each of these applications were generated from a *H2C* array-based implementation.

4.4.2 Meta-data Layout Evaluation

We compare relative execution time for array and struct data layouts on different CPU and GPU platforms. For all the architectures, we compare the SoA layout with AoS$*$ layouts. The execution time also contains the data copy (communication) time and is obtained from the OpenCL API. The communication time is negligible for Intel GPU because of its integrated GPU and shared memory architecture. As a result, there is no copying overhead.

Medical Imaging			
SoA	V1, V2, V3, U1, U2, U3, S, T, interpT		
AoSU	V1, V2, V3, {U1, U2, U3}, S, T, interpT		
AoSV	{V1, V2, V3}, U1, U2, U3, S, T, interpT		
AoSUV	{V1, V2, V3}, {U1, U2, U3}, S, T, interpT		
LBM			
SoA	19 Fields belong to SoA		
AoS	1 AoS of size 16, 1 AoS of size 3		
NBody			
SoA	px, py, pz, vx, vy, vz, ax, ay, az, mass		
AoS	{px, py, pz, ax, ay, az}, {vx, vy, vz}, mass		
AoSP	{px, py, pz}, {vx, vy, vz}, {ax, ay, az}, mass		
Seismic		**KMeans**	
SoA	D, L, V, M, S, T	SoA	16 Fields belong to SoA
AoS	D, L, V, M, { S, T}	AoS	1 AoS of size 16
Seismic		**SRAD**	
SoA	16 Fields belong to SoA	SoA	N, S, E, W
AoS	1 AoS of size 16	AoS	{N, S, E}, W
		AoSE	{N, S}, {E, W}
MRIQ		**GESUMMV**	
SoA	kx , ky , kz, phiMag	SoA	a, b, x, y, tmp
AoS	{kx , ky , kz}, phiMag	AoS	{a, b}, x, y, tmp
GEMMVER		**SYR2K**	
SoA	u1, u2, v1, v2	SoA	a, b, c
AoS	{u1, u2}, {v1, v2}	AoS	{a, b}, c

Table 4.3 : Data layouts description

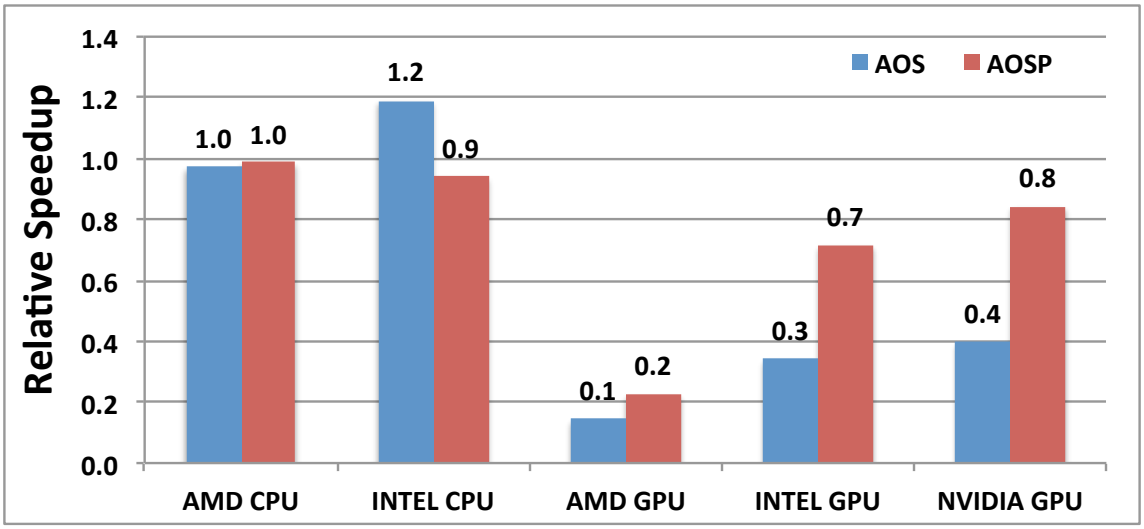


Figure 4.16 : Performance of NBody with AoS and AoSP relative to SoA layout on various devices

Figure 4.16 shows the performance of the N-Body benchmark for various layouts. We see that the AoS and AoSP versions performs well on the CPU. The SoA layout performs better on GPUs due to memory coalescing.

Figure 4.17 shows the performance of the Seismic benchmark. The SoA layout shows better performance on AMD CPU, whereas the AoS layout is better on Intel CPU. This can be attributed to the difference in cache associativity and sizes between AMD and Intel. On the GPU side, the array layout performs well on all 3 GPU hardware as expected.

Figure 4.18 shows the performance of the SRAD benchmark for different layouts. SRAD shows improved performance for the AoS and AoSE layouts relative to the SoA layout for all the architectures. Surprisingly even on the GPU the struct layout performs better than the array layout. This is contrary to GPU best practices. The memory access functions in the SRAD kernel are non-affine and irregular. It is dif-

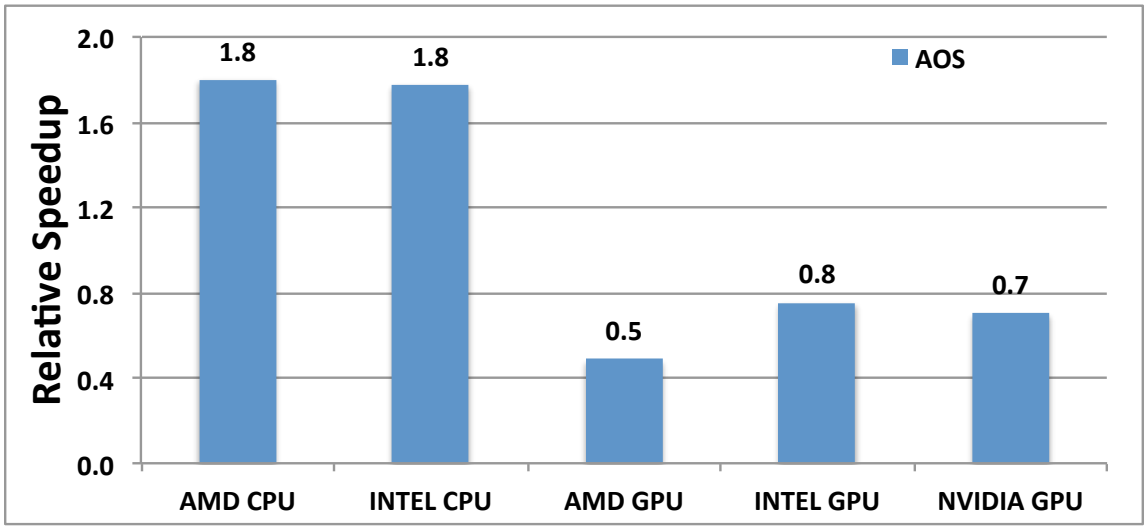


Figure 4.17 : Performance of Seismic with AoS relative to SoA layout on various devices

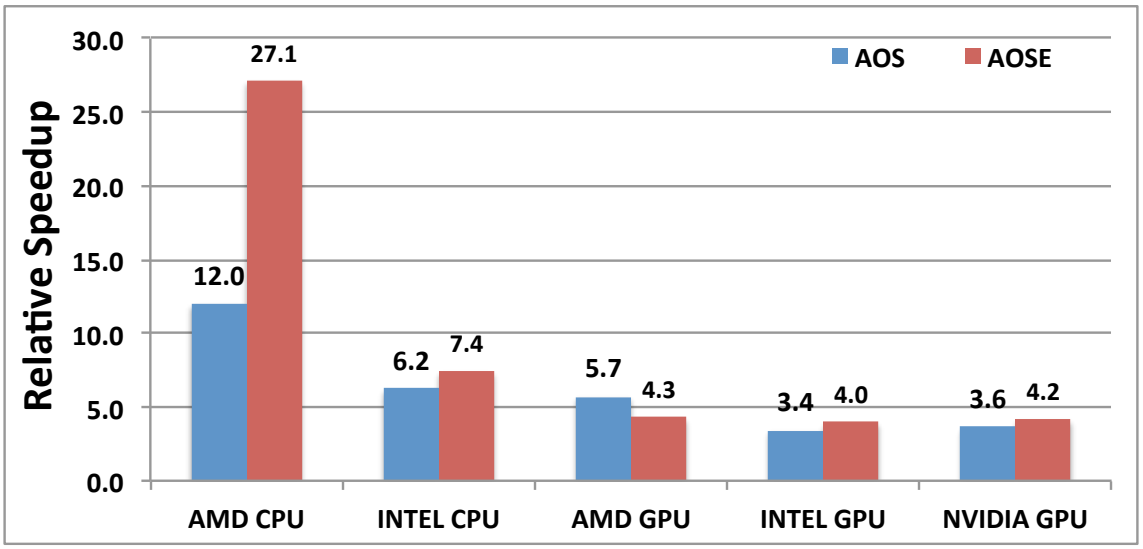


Figure 4.18 : Performance of SRAD with AoS and AoSE relative to SoA layout on various devices

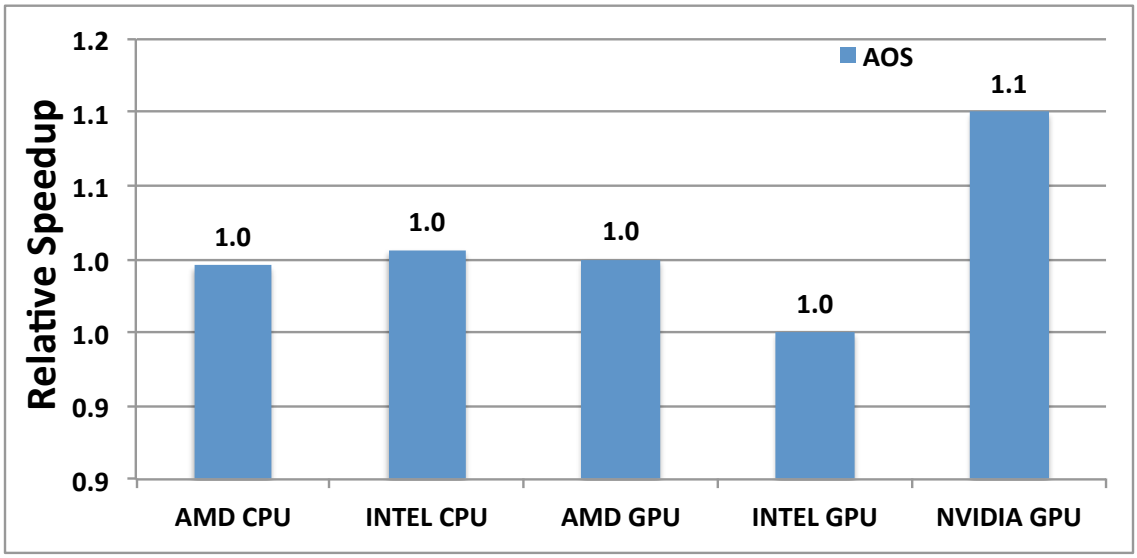


Figure 4.19 : Performance of MRIQ with AoS relative to SoA layout on various devices

ficult for a compiler or programmer to analyze and determine the right layout. Our framework enables rapid prototyping and testing of different layouts for performance on multiple architectures.

Figure 4.19 shows the performance of MRIQ benchmark. the NVIDIA GPU performs slightly better on the AoS layout. This is because a single device copy due to the AoS layout is efficient compared to multiple copies resulting from SoA layout. For the other architectures, MRIQ exhibits little or no variation across layouts. Data layout does not play a role in MRIQ performance since it is compute bound. If an application is compute bound, then the data layout does not make a significant difference in performance because the memory latency is hidden by the computation.

Figure 4.20 shows the performance of medical image benchmark for different layouts. The AoS and {AoSU} layouts are better on the CPU whereas the SoA layout is better on the GPU. Medical image kernel is similar to a 3D Jacobi (stencil) computa-

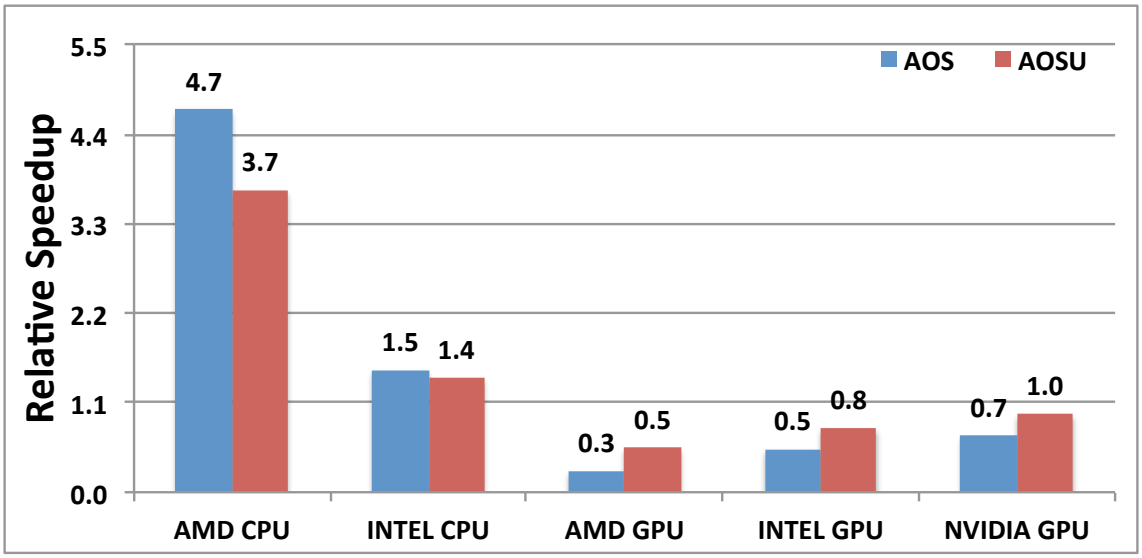


Figure 4.20 : Performance of Medical with AoS and AoSU relative to SoA layout on various devices

tion. The stencil computation is performed separately on three input buffers, and the results are written into corresponding output buffers. Keeping the input buffers in a single struct is helpful for the CPU because when a point for one stencil is loaded, the points for the other two stencils are implicitly loaded (multiple points fit in a cache line) The array layout would have caused three loads for the same point, one in each of the three stencils. On the GPU side, the SoA layout is better as expected due to memory coalescing.

Best practices generally dictate the use of SoA data layouts on GPUs due to improved coalescence of global memory accesses. However, our SRAD and MRIQ results contradict this knowledge. Our meta-data framework enables rapid prototyping and optimization of different data layouts, allowing tuning experts to rapidly discover optimal layouts for complex and irregular applications. For the CPU, the layout often

depends upon the kernel features and memory access patterns. Our programming model can easily port such applications to different architectures.

4.4.3 ADHA Evaluation

SDL evaluation

We report *SDL* results for all the data-parallel kernels in our benchmarks. Figure 4.21 shows the speedups obtained for CPU-*SDL*, GPU-*SDL* for our benchmarks. The default layout specified by the programmer is used as the baseline (as shown in Table 4.1 and Table 4.3). 13 out of the total 16 kernels show speedup using our SGML greedy heuristic (Algorithm 4) compared to the baseline layout. It is not surprising to see that many data-parallel kernels show performance improvement from data layout optimization since it results in better cache utilization. We observe performance benefits of up to 2.87× with a geometric mean improvement of 1.35× on the CPU. With the GPU execution using the GPU-*SDL*, we found performance improvements of up to 2.21× with a geometric mean improvement of 1.31×. It is important to note that we take advantage of the better load instructions available on most GPU hardware as described in Section 4.3.1. These benefits can be attributed to the decreased instruction pressure and better memory bandwidth due to the generation of better loads by the NVIDIA backend compiler.

We now discuss the results for each data-parallel kernel: The first seven kernels (Medical-1 to Medical-7) in Figure 4.21 are from the Medical imaging registration benchmark. Kernels numbered 1, 2, 3, and 7, access $V1, V2, V3$ fields and are grouped together by our *SDL* algorithm and is named as AoSV layout. Kernels numbered 4, and 5 access fields $U1, U2, U3$ and $V1, V2, V3$ as two groups, but have complementary access patterns (Read, Write). These two groups are kept independently and is named

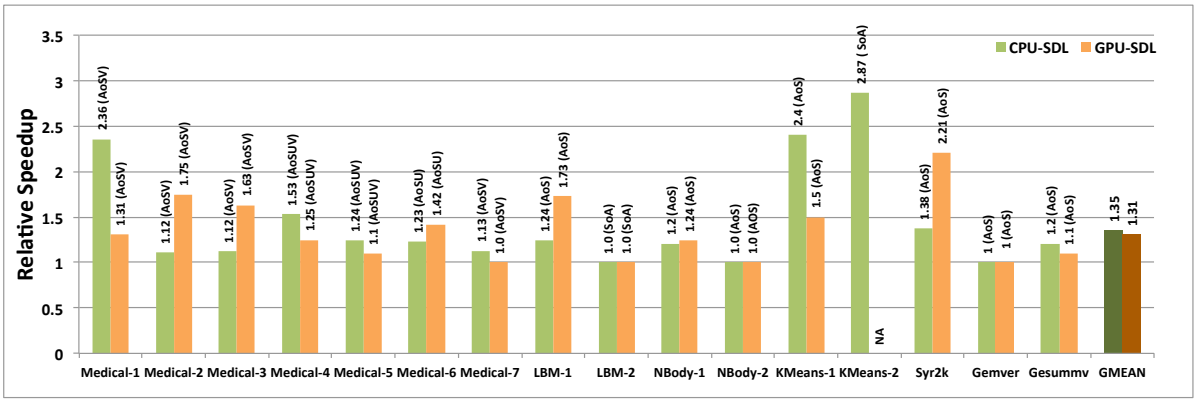


Figure 4.21 : Speedup for all data-parallel kernels on the CPU and GPU by using our *SDL* algorithm compared to the programmer specified default layout

as AoSUV layout. Finally, Kernel 6 accesses the $U1, U2, U3$ fields together and are grouped with the name AoSU layout. We obtain speedup ranging from $1\times$ to $2.36\times$ for all the kernels by using CPU-*SDL* and GPU-*SDL*.

The bars for LBM-1 and LBM-2 in Figure 4.21 show the speedups obtained for the two kernels in the LBM benchmark. This benchmark has 19 fields for each lattice point in a 3-D space. The first kernel access all the 19 fields for each lattice point while the second kernel access the lattice points for each field. We observe that both the kernels require complementary data layouts. The *SDL* pass combines all the 19 fields in an AoS layout for the first kernel and gives SoA layout for the second kernel. The first kernel gives a speedup of $1.24\times$ on the CPU and $1.73\times$ on the GPU. The second kernel does not benefit from our layout transformation since it uses original baseline SoA layout.

The bars for NBody-1 and NBody-2 in Figure 4.21 show the speedups obtained for the NBody benchmark. The position and acceleration fields occur together in

the first kernel but with different access frequencies. The position, acceleration and velocity fields occur together in the second kernel. The *SDL* pass groups them into individual groups as AoS layout. This application spends 99% of its execution time in the first kernel. We observe a speedup of 1.2× on the CPU and 1.24× on the GPU for the first kernel.

The bars for KMeans-1 and KMeans-2 in Figure 4.21 show the speedups obtained for the KMeans benchmark. The first kernel finds the cluster index for each of the input. Hence the *SDL* pass groups all the clustering features in an AoS layout. We limit the AoS size to 16 which is based on the cache line size. The second kernel is a reduction kernel which is not supported by our current GPU implementation. The first kernel results in a speedup of 2.4× on the CPU and 1.5× on the GPU. The second kernel achieves a speedup of 2.87× on the CPU.

SYR2K kernel reads from the arrays a, b, and writes to array c. Some accesses to the arrays a and b are strided. Hence AoS layout benefits from improved cache utilization compared to SoA. We get a speedup of 1.38× on the CPU and 2.21× on the GPU.

GEMVER kernel reads from the fields $u1$, $u2$, $v1$, and $v2$. However the sizes of these fields are very small. Hence the AoS layout performs similar to SoA layout as observed in our combine_section cost model.

GESUMMV reads from the fields a, b, x, and writes to fields tmp, y. However the sizes of x and y differ from that of a and b, and hence only a and b are combined by *SDL*. We observe a speedup of 1.2× on the GPU and 1.1× on the CPU.

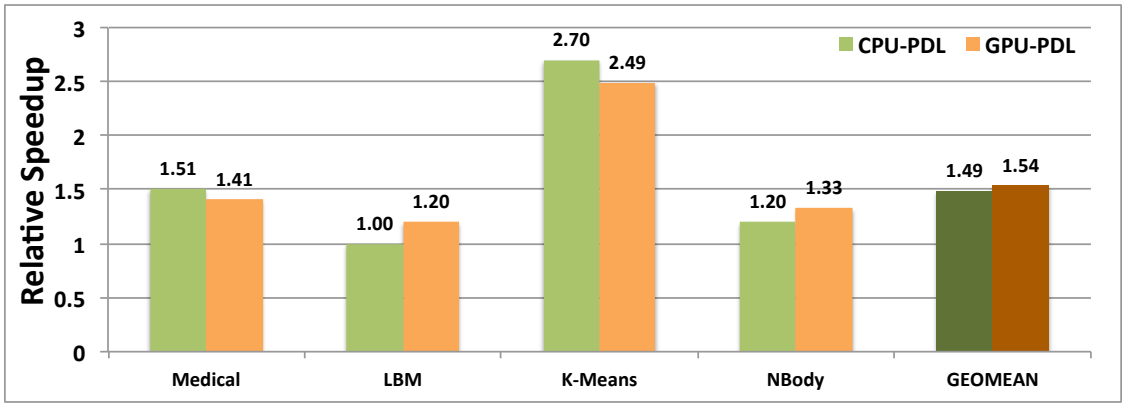


Figure 4.22 : Speedup for multi-kernel benchmarks on the CPU and GPU by using our *PDL* algorithm compared to the programmer specified default layout

PDL evaluation

We now report results for our multi-kernel benchmarks: Medical Imaging, LBM, K-Means, and NBody using the *PDL*. None of the benchmarks have any control flow between the individual data-parallel kernels. We use the combine_section costs and remap_layout costs as shown in Figures 4.12, 4.13 and Figures 4.14, 4.15. We can observe from these graphs that the combine_section cost is $\sim$ 1000 msec between 8-AoS and SoA configurations for the CPU for 1024 MB of data. This means that if a kernel has 8-AoS layout and is combined to an intermediate SoA layout, we estimate the performance loss as $\sim$ 1000 msec. This combine cost is less than the remap cost of $\sim$ 2000 msec (via LDR) on the CPU . On the other hand, the remap cost of $\sim$ 90 msec for LDR is less than the combine_section cost of $\sim$ 800 msec for 8-AoS and SoA on the GPU for 1024 MB. Figure 4.22 shows the PDL speedup of the benchmarks we evaluated.

The first three kernels of the medical imaging benchmark have the same layout

AoSV as shown in Figure 4.21. Kernels 4 and 5 have a different layout AoSUV. Now we either have to combine_section or remap these two layouts. The combine_section cost between AoSU and AoSUV is 0 because they donot have any common fields. Hence we combine_section these *sections*. Similarly, AoSUV and AoSU donot have any common fields and hence we combine_section kernels 5 and kernel 6. Finally, AoSU and AoSV layouts do not have any common fields and hence kernel 6 and kernel 7 are combined. The overall layout from the *PDL* pass is AoSUV and a speedup of 1.51× on the CPU and 1.41× on the GPU.

LBM is interesting for *PDL* because both of it's kernels prefer complementary layouts as explained in the *SDL* results. *PDL* has decide if it is beneficial to use combine_section or remap_layout. This benchmark uses a total grid size of approximately 1024 MB. *PDL* computes the corresponding costs from the combine_section and remap_layout models described in Sections 4.3.3 & 4.3.3. As explained earlier, it is more beneficial to perform combine_section on the CPU and also to perform remap_layout on the GPU. The combine_section model for the CPU uses the programmer specified SoA layout because the two kernels cannot be combined. Hence the speedup compared to the baseline layout is 1. On the GPU, the two kernels are remapped using LDR and we observed a speedup of 1.2×.

Both kernels in K-Means have been written with a default layout of SoA to enable coalescing on the GPU. That is, all the features of the input data are independent arrays. Both the kernels can take advantage of AoS layout since each kernel is iterating on all the features for every data item. The *SDL* pass assigned an AoS layout for each of the kernels. As mentioned on the *SDL* results, the second kernel is executed sequentially on the CPU. In the *PDL* pass, both the kernels will keep the layout as AoS. We observed a speedup of 2.7× on the CPU and 2.49× on the GPU. The overall

speedups obtained are dominated by the speedup from the second kernel which gets executed on the CPU with AoS layout (shown in Figure 4.21).

The combine_section cost of the NBody kernels is 0. This is because their corresponding layouts are independent. Hence the *PDL* output is the same AoS layout. Since the first kernel dominates the majority execution time, the speedup is similar to *SDL*, which is 1.2× on the CPU and 1.23× on the GPU.

Overall, we observe a geometric mean speedup of 1.49× on the CPU and 1.54× on the GPU.

4.5 Extensions

We describe the data layout problem in section 4.3.2. The mapping problem described below is similar to the layout problem and deals with finding the optimal mapping of a given program on a given heterogeneous system. One can choose to map the entire program on a single device, or choose to map different parts of the program to different devices with data copying in between. The mapping problem is described as follows.

Mapping Problem

We use $\mathsf{Cf}(S_i, e_i)$ to denote the cost of executing *section* S_i on device e_i and $C(e_i, e_{i-1})$ to denote the cost of moving data from device e_{i-1} to device e_i. Finally, the mapping problem can be formulated as finding the device mapping E for program P such that following is minimum.

$$\sum_{i=1}^{N} (\mathsf{Cf}(S_i, e_i) + C(e_i, d_{e-1}))$$

The above formulation essentially finds the best mapping of various forasyncs in a program such the total cost of execution and data movement is minimized.

We plan to extend our ADHA framework formulation and combine the mapping problem with the data layout problem as follows.

Mapping + Data Layout Problem

We use $\mathsf{Cf}(S_i, d_i, e_i)$ to denote the cost of executing *section* S_i with data layout d_i on device e_i where $e_i \epsilon E_i$ and $d_i \epsilon D_i$. Let $\mathsf{Cl}(d_i, d_j)$ denote the cost to obtain data layout d_j from d_i and $\mathsf{Ct}(e_i, e_j)$ be the cost of transferring data from device e_i to device e_j.

Finally, the data layout and mapping problem can be formulated as finding the data layout D and device mapping E for program P such that

$$\sum_{i=1}^{N}(\mathsf{Cf}(d_i, S_i, e_i) + \mathsf{Cl}(d_i, d_{i-1}) + \mathsf{Ct}(e_i, e_{i-1}))$$

is minimum.

The *PDL* approach can still be used to find the best data layout and device mapping for a program on a given heterogeneous platform. The complexity of the overall algorithm will now change based on the number of devices, but the overall complexity of *PDL* will remain polynomial.

4.6 Summary

In this thesis chapter, we provide two solutions to the data layout problem on heterogeneous architectures. We first present a compiler-driven data layout transformation that is applicable to any data parallel programming model. The data layout transformation uses a "meta-file" approach which enables the same source code to be compiled with different layouts without involving the programmer worrying about it. We then present ADHA, an automatic two-level hierarchical data layout framework for heterogeneous architectures that can dramatically improve programmer produc-

tivity and portability for current heterogeneous architectures. We show that this formulation helps separate kernels running on a CPU and GPU, and uses an optimal PTIME algorithm to determine the overall data layout given the data layouts for each kernels computed by greedy search. We provide a reference implementation of the formulation in the Heterogenous Habanero-C compiler framework. The framework uses a parallel intermediate representation to build the affinity graph and a model to estimate the combine_section and remap_layout costs which are used in determining the overall data layout of the program. Our experimental results show significant benefits from these two approaches and demonstrate that the best data layout for a given program can be different for CPU vs. GPU execution. We finally propose extensions to ADHA by combining the mapping problem with the data layout problem.

Chapter 5

Related Work

In this section, we compare the Heterogeneous Habanero-C(*H2C*) programming model, compiler, and runtime implementation with previous work. Section 5.1 compares existing languages that target heterogeneous architectures. Section 5.2 discusses techniques that handle the layout of data and compare them with the meta-data layout framework of *H2C*. Section 5.3 describes software techniques for data coherence on heterogeneous architectures. Section 5.4 discusses techniques to map kernels onto heterogeneous processors. Section 5.5 discusses some advanced features implemented on heterogeneous architectures similar to *Concord*.

5.1 Languages for Heterogenous Architectures

Languages for heterogeneous architectures can be classified as high-level and low-level. Low-level languages that target heterogeneous architectures are OpenCL [26] and CUDA [73]. OpenCL is an open standard to program modern heterogeneous hardware. An OpenCL implementation provides low-level API to compile, execute and also map a program on a heterogeneous architecture. The API also provides constructs to specify asynchronous computations and communication along with synchronization. OpenCL follows the *offload* model where the main program is executed on a "host", which launches tasks onto "devices". Many vendors today including Intel(cpu/gpu/xeon phi), AMD(cpu/cpu/apu), NVIDIA(gpu), Texas Instru-

ments(cpu/dsp), Xilinx(fpga) and Altera(fpga) provide implementations of OpenCL to program their hardware. OpenCL is increasingly being adopted by various developers to write applications for current heterogeneous hardware.

CUDA is introduced by NVIDIA in 2006 as a general purpose compute platform for their GPUs. It's programming model is similar to OpenCL. The key abstractions include a hierarchy of thread groups, shared memories, and barrier synchronization. The hierarchy is divided into blocks of work-groups where each work-group is further partitioned into a set of co-operative parallel threads. CUDA also provides a rich set of mathematical libraries that are tuned to their GPU hardware. A major drawback of CUDA is that it is limited to only NVIDIA GPUs and hence is not portable onto GPU from other vendors. However, both OpenCL and CUDA are challenging for average programmers to learn, thereby limiting the rate of their adoption on newer architectures. They are also not portable in the sense that the same program will not give the best performance on all the architectures.

To overcome these limitations, various existing languages have been extended, and new high-level programming languages have been developed to program current heterogeneous architectures. Grand Central Dispatch (GCD) [74] is another low-level language approach that supports concurrent execution on multicore hardware running iOS and Mac OSX.

Baskaran et al. automatically generate CUDA code from regular C programs [75]. They leverage the polyhedral model to enable efficient memory loads, find thread-block level parallelism and also to take advantage of the on-chip memory. However, the polyhedral model is limited to only affine programs and is constrained by dependency analysis. The high-level constructs of *H2C* enable the programmer to specify both affine and non-affine expressions.

SnuCL [76] extends OpenCL to a cluster of heterogeneous CPU-GPU processors. On the CPU, they emulate work-group coalescing by converting to a sequential loop. The OpenCL extensions include collectives for data.

OpenACC [77] and OpenMP-4.0 [78] provide a directive based approach to target heterogeneous architectures. The programmer annotates code regions using pragmas that are compiled and executed on a particular device. The annotations include constructs for both communication and computation. Both OpenMP and OpenACC are based on the "host+accelerator" model. OpenACC is targeted towards accelerators while OpenMP targets both shared memory CPUs and accelerators. The compiler directives are just hints from the programmer, and different compilers may choose to implement a certain directive differently leading to performance variations across compiler implementations. Mint [79] targets a domain specific problem: stencil computations. Mint is a pragma-based model that automatically generates CUDA from C code for heterogeneous computing. Mint identifies patterns of the stencil and generates code to take advantage of the local memory available on GPUs. Mint is specific to only stencil computations. The optimizer pass in *H2C* also identifies re-use patterns like stencil computations and is capable of code generation similar to Mint. A major limitation of pragma based approaches is that the operations are only limited to the pragma begin/end regions. This is a severe limitation because, the programmer cannot start a pragma in one module(file) and end the same pragma in another module. This forces the entire region constrained to a single module and could result in cluttering to a single file (usually main()). On the other hand, language constructs like in *H2C* are not restricted to any regions. A programmer or the compiler can explicitly manage the lifetime of the data across multiple modules.

Grewe et al. [80] developed a compiler to automatically generate optimized

OpenCL code from data-parallel OpenMP programs. It automatically determines whether to run OpenCL code on the GPU or to run OpenMP code on the multi-core host. C++ Accelerated Massive Parallelism (C++ AMP) [81] is a C++ specification to take advantage of heterogeneous processors such as a GPU. C++ AMP provides some nice abstractions like *array_views*, which are helpful for productivity. It also provides the *tile* and *barrier* constructs to take advantage of the thread group structure on the GPUs. C++ AMP does not currently support hybrid CPU-GPU computing. Dubach et al. introduce Lime [82] programming language for heterogeneous CPU + GPU architectures. Lime is an extension of the Java language. They introduce two operators namely *task* and *connect*. *Task* is mapped to an OpenCL kernel while *connect* represents the flow of data. Lime uses the *finish* construct to ensure completion of the tasks. It also uses a simple pattern matching scheme to take advantage of the various memory hierarchies on the GPU. Lime is a streaming programming model suitable for streaming applications. CnC-CUDA [83] uses CUDA to support heterogenous platforms. CnC is a graph-based programming language, which consists of three main constructs namely step collections, data item collections, and control tag collections. One drawback of CnC-CUDA is that the user has to manually write CUDA code. Cunningham et al. [84] at IBM extend X10 to generate CUDA. X10 follows the APGAS programming model. APGAS model is based on the principles of locality, asynchrony, conditional atomicity and order. X10 and *H2C* have a similar programming model. X10 is a new language based on Java-like object-oriented design.

H2C automatically generates OpenCL code from high-level extension to the Habanero programming model. *H2C* supports some important features like *SVM*, *metadata layout framework* and task distributions which none of the above high-level lan-

guages support. The asynchronous task parallelism enabled by high-level constructs, compiler, and runtime make *H2C* a portable, productive and performant programming language for heterogeneous computing. The other advantage of *H2C* is that *H2C* is an extension of the C programming language, and both existing and new applications can take advantage of the heterogeneous processors.

5.2 Data Layout

The data layout problem has been well studied for more than a decade in various contexts. The goal of data layout optimization techniques is to reduce memory latency, by taking advantage of prefetch streams and exploiting the memory hierarchy. Data layout problem was studied in High Performance Fortran(HPF) to automatically determine the alignment and distribution of global arrays. Ulrich and Kennedy extensively worked on automatic data alignment and distribution framework for HPF [16, 61, 85, 86] at Rice University. The data layout considers the alignment of each dimension of the multidimensional arrays so as to reduce the cost of communicating data across a cluster of distributed processors. The optimal alignment depends upon the access patterns of the array dimensions. Ulrich et al. also show that finding the optimal data is an NP-Complete [85] problem in the absence of control flow. Anderson et al. [87] proved that the problem of dynamic remapping in the presence of control flow is NP-hard. Their work divides a program into *phases*. Each phase consists of a loop nest that covers all the induction variables occurring inside the loop body. Ulrich [85] proves that finding an optimal data alignment is an NP-Complete problem in the absence of control flow. Lam et al. [87] proved that the problem of dynamic remapping in the presence of control flow in NP-hard. Wu et al. [20] have proved that finding the optimal data layout to maximize the number of coalesced

accesses on a GPU is NP-complete.

The number of layouts possible is exponential, and Ulrich presents heuristics to prune these layouts. He also proposes an optimal integer programming solution when the number of kernels is less. Chen Ding later worked on another version of the data layout problem i.e., *array regrouping and structure splitting*. The problem statement is to find the optimal grouping of array fields in a program to efficiently take advantage of cache-reuse on CPUs. Chen et al. [17,88] extended the proof from Ulrich and claims that the problem of array regrouping is also NP-complete but does not provide any proof. We provide a complete proof of NP-completeness for the array regrouping problem. Chen and Ulrich partition the programs into phases (parts of a program which access data more than a cache line) and find the grouping where it is profitable to do so. Their profitability heuristic is to find the sets of arrays that: 1) Always occur together in the entire program, 2) The set is the largest possible set. They further propose extensions(no implementation) to the formulation to allow for useless data and dynamic remapping of layouts. The array regrouping heuristics have been further extended to handle irregular programs. Zhong et. al. [18, 19] use profile information to build reference graphs and use clustering heuristics to determine the best layout. Luz et al. [89] showed the benefits of array regrouping on embedded systems.

Sung et al. [56] use data layout transformation to enable memory level parallelism on structured grid applications. They look into AoS and SoA for GPU memory coalescing. Their framework increases the memory level parallelism by distributing the data access by a thread to different banks. The meta-data framework in *H2C* considers other important factors like prefetching, TLB miss rate, and cache miss effects to figure the optimal layout.

DL [90]. Uses in-place transposition to remap data via cyclic copying. The user

has to write a different version of the code for array layout and ASTA(AOSOA). ASTA will help in avoiding camping of the memory channels on GPU due to large slides compared to arrays. Dymaxion [55] provides an API, which is a set of remapping functions from one layout to another. *maprow2col*, *mapdiagonal*, *mapindirect* are some of the mapping functions provided by the API. The remapping of the data is done along with the PCI-E transfer of data. The runtime chunks the data and launches a transformation kernel for each chunk. This allows overlap of remapping and transfer of data. The authors evaluate the performance of hybrid CPU-GPU execution of the k-means application. They use one layout for the CPU and another layout for the GPU with the help of their API. Dymaxion uses a runtime approach which the authors show could be prohibitive. Our *H2C* compiler uses compile-time techniques to change the data layout and leverages its asynchronous features to reduce the overhead of data remapping.

TALC [91] uses a meta-file and an input program to generate code with the corresponding layout. Our meta-data layout framework has been inspired from TALC. TALC, however, is limited only to CPUs. We extend TALC by generalizing it for heterogeneous processors.

5.3 Data Management among Heterogeneous devices

There are two schemes to manage data coherence in a heterogeneous environment. The first scheme uses static analysis to determine the coherence points, and the second scheme proposes runtime framework to dynamically handle the coherence on a need basis. Static analysis techniques are limited because the compiler is forced to make conservative assumption resulting in redundant communication. However, static analysis can benefit from the dependency information to overlap the communication

and computation. On the other hand, dynamic techniques are more precise because much of the dependency information is resolved at runtime and there by minimizing the communication. However, dynamic approached suffer from overheads due to runtime management of coherence information.

Jablin et al. [92] developed a runtime framework *DyManD*, to dynamically manage data for CPU-GPU architectures. Their previous work CGCM [93], uses a static analysis to show that acyclic communication between CPU and GPU if present can lead to good performance. DyManD relies on the modified memory allocators. It allocates numerically identical addresses on CPU and GPU by allocating data on the GPU first and then using mmap to get the corresponding CPU address. Any address on the GPU is just a masked version of the CPU address. Use interrupts to change the state of memory to one of GPUEx. CPUEx and shared. It also introduces glue kernels. The idea is to convert a short CPU kernel between two GPU kernels into a single threaded GPU kernel to avoid cyclic communication. Pai et al. [14] improve over DyManD for X10 language. By checking if data is stale on the GPU, their framework avoids transfer of data. They use compiler analysis to insert coherence checks at the optimal point. Amini et al. [94] in their work design a static analysis to optimize the communication between a host-accelerator system. Their automatic approach focuses on transferring the data to the device from the host as early as possible to delay the transfer from the host to the device as late as possible. Use LRU policy to evict the data on GPU.

5.4 Hybrid CPU-GPU Execution

Chau-Wen [95] in his thesis work designed the Fortran D compiler to automate and optimize the communication, data layout and partitioning the work for Fortran pro-

grams. Qilin [96] provides wrappers for heterogeneous computing and uses adaptive mapping to schedule the work between CPU and GPU. Boyle et al. [97] use machine learning techniques to statically partition the work between CPU and GPU. They execute a suite of benchmarks to build a code feature vector. This feature vector is built using **raw kernel features** like the number of compute operations, accesses to global memory, accesses to local memory, coalesced memory accesses, average number of data transfers and work-items per kernel. The future derive some combined code features like communication to computation ratio, % coalesced memory accesses, the ratio of local to global memory accesses × avg. # work-items per kernel, computation-memory ratio. Lee et al. [98] address the issue of partitioning data-parallel kernels with irregular memory access patterns over multiple devices. Merging discontinuous data is done by copying the device memory to host buffer. Then they launch a CPU kernel without the compute part just to copy the data to the corresponding locations. The partitioning decision becomes more complicated when systems are equipped with several types of devices. The performance of a GPU is often not constant to the amount of data that it operates upon, and this variation will affect the partitioning decision. To handle this problem, they introduce a performance variation-aware partitioning scheme that builds a profile for each device with copy costs and then decides the profitability of offloading using a recursive tree-based approach. Petabricks extension [99] to support GPUs via OpenCL includes high-level algorithmic choices. The compiler divides these choices according to the ease of mapping them efficiently onto a CPU and GPU. Alina et al. [100] in the Habanero team map a data-flow programming model onto heterogeneous processors. They build a work-stealing runtime to schedule the work on different processors.

Hierarchical Place Trees (HPT) [54] is a programming abstraction for task place-

ment and data movement. The memory hierarchy of a machine is modeled as a hierarchical tree and each memory location is denoted as a place. The programmer can now schedule tasks, which use similar data on a particular place thereby exploiting locality. HPTs support GPUs memories by adding an *acc* clause to the program indicating that the data is not implicitly accessed outside its place. HITMAP [101,102] is a library-based approach that provides an API for user-defined distributions. It adopts and SPMD model and provides certain high reusable communication patterns such as point-to-point communications, paired exchanges for neighbors, shifts along a virtual axis.

Automatic mapping approaches are limited to only certain applications or a single architecture. For example, automatic approaches to mapping of tasks assume that all the processors resources are available for a given task. However, in practice the resources could be shared or limited. For instance, the memory of a GPU is limited to at most 12GB in state-of-art devices, and the automatic mapper must now be aware of these constraints and make decisions at runtime that could result in performance degradation. *H2C* provides the high-level *at* and *partition* constructs to the programmer to specify the mapping. The user chooses the mapping of tasks based on the resources available and the compiler automatically determines the data distribution.

5.5 Advanced GPU Support

Both CUDA and the next major release of OpenCL, OpenCL 2.0 [26], support pointer sharing (SVM) between the CPU and GPU. However, CUDA's SVM support requires hardware support (it is limited to NVIDIA's Fermi-class and later GPUs), while OpenCL 2.0's SVM typically requires special hardware or operating system support. AMD APU A10-7650K (code name Kaveri) is the first integrated GPU with full

hardware support for SVM.

GMAC [15] provides shared memory support between CPU and GPU. It uses a coherence protocol to maintain the coherence between CPU+GPU. It handles the non-virtual addressing by requesting the same virtual address space on the CPU side (using mmap) that is generated on the GPU side. However, GMAC is limited only to a single device since multiple devices can generate overlapping virtual address ranges. *Concord* on the other side uses a compiler approach to handle the mapping and can support any number of devices.

There has been past work on implementing shared virtual memory in software [103] on distributed-memory processors via distributed shared memory (DSM) schemes. DSMs are implemented in a couple of ways including memory management software, Operating System extensions and language runtime systems. However, implementing such a system is complex and also not feasible in many heterogeneous systems today due to restrictions imposed either by the vendor or the particular hardware. Some of these restrictions include lack of OS support and closed nature of the hardware. In the modern heterogeneous setting, virtual memory sharing in software is only achieved (in some cases) by vendor-provided drivers. For example, CUDA Unified Virtual Addressing is restricted only to NVIDIA discrete GPUs. Other efforts that simplify programming heterogeneous systems include using custom hardware approaches like Merge [104] and Exochi [105], but these approaches are limited to a particular hardware. The techniques applied in *Concord* can be implemented on top any heterogeneous hardware with a coherent memory subsystem.

Chapter 6

Conclusions and Future Work

Heterogeneous architectures are pervasive today and will be in the future. However, programming these architectures is non-trivial, and this poses constraints on portability, productivity, and performance. The diverse architectural features of these heterogeneous architectures make it challenging to achieve the optimal performance and energy efficiency. It is necessary to be able to program these architectures in a machine independent manner. In this dissertation, we have implemented two programming models, *Concord* and Heterogeneous Habanero-C (*H2C*) that address the above portability and productivity challenges for heterogeneous architectures.

Concord is a C++ programming framework for processors with integrated GPUs. With support for SVM and most C++ constructs, *Concord* is designed to allow object-oriented C++ data-parallel programs to seamlessly take advantage of GPU execution in addition to multi-core execution. Additionally, its compiler optimizations reduce the cost of software-based SVM implementation. Using seventeen realistic regular and irregular C++ applications, we demonstrate that C++ applications that use recursive data structures, and object-oriented features can be automatically mapped to the GPU. Furthermore, we demonstrate that GPU execution can bring significant energy benefits to irregular C++ applications even without sophisticated algorithm or data restructuring changes. This is in contrast to the large literature on GPU execution that show benefit for regular applications.

H2C programming model targets multiple heterogeneous devices and provides features that make programming these devices very simple. The highlights of *H2C* include high-level constructs to overlap communication and computation, partition tasks, distribute data, and a unified event framework. The *H2C* compiler takes advantage of both AST and polyhedral optimizations to generate code tuned to a particular heterogeneous hardware. Evaluation of four benchmarks shows *H2C* to be portable, productive and also achieve performance similar to hand-coded low-level OpenCL implementations on a system with CPU and more than one GPU.

Memory latency is a major source of performance degradation in today's applications. With memory hierarchies becoming deeper, data layout plays an important role in reducing these latencies. The current trend of programming systems is to leave the data layout to the programmer. We introduce two data layout frameworks in *H2C*: First, A *meta-data layout* framework enables a programmer to specify a high-level specification of the layout. The compiler automatically generates a code executable with the specified layout; Second, Automatic Data layout for Heterogeneous Architectures (ADHA), automatically determines the best layout for a given application. The best layout for a given program is dependent on the kernel mapping, data transpose and communication costs. ADHA formulates the layout + mapping problem into two phases and determines the best layout for the entire program. Experimental results show data layout is crucial in application performance. Our two data layout frameworks extend the *H2C* programming system in achieving higher performance portability.

We believe these programming systems will be of tremendous value in the upcoming years where heterogeneous systems will be a lot more ubiquitous and pervasive.

Future Work

As part of our future work, we plan to implement and evaluate our proposed extensions to *H2C* and ADHA data layout framework described in section 3.6 and section 4.5 respectively. We also plan to extend ADHA to compute the optimal layout and mapping in terms of energy efficiency.

Bibliography

[1] G. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan 1998.

[2] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, pp. 256–268, Oct 1974.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, no. 3, pp. 122–134, 2012.

[4] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, pp. 589–604, July 2005.

[5] Intel, "Ivy Bridge," 2012. http://ark.intel.com/products/codename/29902/Ivy-Bridge.

[6] AMD, "Heterogeneous System Architecture (HSA)," 2014. http://www.amd.com/en-us/innovations/software-technologies/processors-for-business/hsa.

[7] NVIDIA, "High Performance Computing (HPC)," 2015. http://www.nvidia.com/page/products.html.

[8] Texas Instruments, "66AK2L06 Multicore DSP+ARM KeyStone II System-on-Chip (SoC)," 2015.

[9] Altera, " FPGA," 2015. https://www.altera.com/products/fpga/overview.html.

[10] Cadence, "Tensilica Customizable Processor IP," 2015. http://ip.cadence.com/ipportfolio/tensilica-ip.

[11] Top 500, "TOP 10 Sites for November 2014," 2014. http://www.top500.org/lists/2014/11/.

[12] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Toward dark silicon in servers," *Micro, IEEE*, vol. 31, pp. 6–15, July 2011.

[13] A. Wood, "The internet of things is revolutionising our lives, but standards are a must," *The Guardian*, March 2015.

[14] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, (New York, NY, USA), pp. 33–42, ACM, 2012.

[15] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," *SIGPLAN Not.*, vol. 45, pp. 347–358, Mar. 2010.

[16] K. Kennedy and U. Kremer, "Initial Framework for Automatic Data Layout in Fortran D: A Short Update on a Case Study," Tech. Rep. CRPC-TR93324-

S, Rice University, Houston, Texas, USA, 1993. http://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR93324-S.pdf.

[17] C. Ding and K. Kennedy, "Inter-array data regrouping," in *Languages and Compilers for Parallel Computing* (L. Carter and J. Ferrante, eds.), vol. 1863 of *Lecture Notes in Computer Science*, pp. 149–163, Springer Berlin Heidelberg, 2000.

[18] Y. Zhong, M. Orlovich, X. Shen, and C. Ding, "Array Regrouping and Structure Splitting Using Whole-program Reference Affinity," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, (New York, NY, USA), pp. 255–266, ACM, 2004.

[19] X. Shen, Y. Gao, C. Ding, and R. Archambault, "Lightweight Reference Affinity Analysis," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, (New York, NY, USA), pp. 131–140, ACM, 2005.

[20] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, (New York, NY, USA), pp. 57–68, ACM, 2013.

[21] Intel Corporation, "The Intel Threading Building Blocks," 2006. https://www.threadingbuildingblocks.org//.

[22] Standard C++ Foundation, "Serialization and Unserialization," 2015. https://isocpp.org/wiki/faq/serialization.

[23] K. Kennedy, C. Koelbel, and H. Zima, "The Rise and Fall of High Performance Fortran: An Historical Object Lesson," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, (New York, NY, USA), pp. 7–1–7–22, ACM, 2007.

[24] S. Chatterjee, S. Tasrlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with mpi," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 712–725, May 2013.

[25] U. Consortium, "UPC Language Specifications, v1.2," Tech. Rep. LBNL-59208, Lawrence Berkeley National Lab Tech Report, Berkeley, California, USA, 2005. http://www.gwu.edu/~upc/publications/LBNL-59208.pdf.

[26] Khronos, "OpenCL: The open standard for parallel programming of heterogeneous systems," 2010. http://www.khronos.org/opencl/.

[27] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 225–234, ACM, 2008.

[28] A. Sidelnik, S. Maleki, B. Chamberlain, M. Garzaran, and D. Padua, "Performance portability with the chapel language," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 582–594, May 2012.

[29] M. G. Burke, K. Knobe, R. Newton, and V. Sarkar, "The Concurrent Collections Programming Model," Tech. Rep. TR 10-12, Rice University, Houston, Texas, USA, 2010.

[30] Microsoft, "Task Parallel Library (TPL)," 2011. http://msdn.microsoft.com/en-us/library/dd460717.aspx.

[31] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, (New York, NY, USA), pp. 212–223, ACM, 1998.

[32] Intel Labs, "iHRC," 2011. https://github.com/IntelLabs/iHRC.

[33] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive Heterogeneous Scheduling for Integrated GPUs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), pp. 151–162, ACM, 2014.

[34] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[35] J. Knoop, O. Rüthing, and B. Steffen, "Optimal code motion: theory and practice," *TOPLAS*, vol. 16, pp. 1117–1155, July 1994.

[36] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The Tao of Parallelism in Algorithms," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 12–25, ACM, 2011.

[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Oct 2009.

[38] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[39] B. Werth, "Pet Me," 2011. http://software.intel.com/en-us/articles/multi-core-simulation-of-soft-body-characters-using-cloth/.

[40] G. Bradski, "OpenCV," *Dr. Dobb's Journal of Software Tools*, 2000.

[41] Q. Wayne, "Mandelbrot," 2008. https://sites.google.com/site/quickwayne/home.

[42] R. Reed, "n-bodies: exploring a parallel TBB solution, intro and peek ahead," 2009. http://software.intel.com/en-us/blogs/2009/08/19/n-bodies-exploring-a-parallel-tbb-solution-intro-and-peek-ahead/.

[43] D. Cherkassov, "First-Rays," 2012. https://github.com/4DA/codermind-raytracer.

[44] D. Quinlan, "ROSE: Compiler Support For Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[45] "PolyOpt: A complete source-to-source Polyhedral Compiler." http://www.cs.ucla.edu/~pouchet/software/polyopt/.

[46] "OpenScop:A Specification and a Library for Data Exchange in Polyhedral Compilation Tools." http://icps.u-strasbg.fr/people/bastoul/public_html/development/openscop/index.html.

[47] D. Majeti, R. Barik, J. Zhao, V. Sarkar, and M. Grossman, "Compiler Driven Data Layout Transformation for Heterogeneous Platforms," in *The International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, HeteroPar '13, (Aachen, Germany), LNCS, 2013.

[48] P. Feautrier, "Parametric integer programming," *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[49] D. Chavarría-Miranda and J. Mellor-Crummey, "Effective communication coalescing for data-parallel applications," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, (New York, NY, USA), pp. 14–25, ACM, 2005.

[50] "Halliburton Services." http://www.halliburton.com/en-US/.

[51] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[52] T. McCabe, "A Complexity Measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, pp. 308–320, Dec 1976.

[53] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[54] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement," in *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, (Berlin, Heidelberg), pp. 172–187, Springer-Verlag, 2010.

[55] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: optimizing memory access patterns for heterogeneous systems," in *Proceedings of 11th International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[56] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 513–522, ACM, 2010.

[57] E. Raman, R. Hundt, and S. Mannarswamy, "Structure Layout Optimization for Multithreaded Programs," in *Proc. of CGO*, pp. 271–282, 2007.

[58] NVIDIA, "CUDA Toolkit Documentation v6.5," in *NVIDIA Corportation*, 2014.

[59] G. Mei and H. Tian, "Performance Impact of Data Layout on the GPU-accelerated IDW Interpolation," *CoRR*, vol. abs/1402.4986, 2014.

[60] W. mei W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and

superscalar compilation," *The Jounral of SuperComputing*, vol. 7, pp. 229–248, 1993.

[61] K. Kennedy and U. Kremer, "Automatic data layout for high performance Fortran," in *Proc. of SC*, 1995.

[62] M. R. Garey and D. S. Johnson, *Computers and intractability*, vol. 174. freeman San Francisco, 1979.

[63] K. G. Murty and C. Perin, "A 1-matching blossom-type algorithm for edge covering problems," *Networks*, vol. 12, no. 4, pp. 379–391, 1982.

[64] J. Zhao and V. Sarkar, "Intermediate Language Extensions for Parallelism," in *Proc. of SPLASH Workshops*, pp. 329–340, 2011.

[65] J. Knoop, O. Rüthing, and B. Steffen, "Lazy Code Motion," in *Programming language design and implementation*, vol. 27, pp. 224–234, ACM, 1992.

[66] V. Sarkar, "Automatic Selection of High-order Transformations in the IBM XL FORTRAN Compilers," *IBM J. Res. Dev.*, vol. 41, pp. 233–264, May 1997.

[67] Center for Domain Specific Computing, "CDSC Research Applications," 2009. http://www.cdsc.ucla.edu/research/.

[68] Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *In Proceedings of the IEEE International Symposium on Workload Characterization*, ISWC'09, pp. 44–54, 2009.

[69] "PolyBench/GPU." http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU/.

[70] "Intel OpenCL SDK." http://software.intel.com/en-us/vcsource/tools/opencl-sdk.

[71] "NVIDIA SDK." https://developer.nvidia.com.

[72] "AMD APP SDK v2.8." http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk.

[73] NVIDIA Corporation, "The CUDA Specification," 2015. www.nvidia.com.

[74] A. Corporation, "Grand Central Dispatch," 2009. https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html.

[75] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, (Berlin, Heidelberg), pp. 244–263, Springer-Verlag, 2010.

[76] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), pp. 341–352, ACM, 2012.

[77] Khronos, "The OpenACC: Application Programming Interface," 2011. www.openacc-standard.org/.

[78] "OpenMP." www.openmp.org.

[79] D. Unat, X. Cai, and S. B. Baden, "Mint: realizing CUDA performance in 3D stencil methods with annotated C," in *Proceedings of the international conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 214–224, ACM, 2011.

[80] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, (Berlin, Heidelberg), pp. 286–305, Springer-Verlag, 2011.

[81] " C++ Accelerated Massive Parallelism." http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx.

[82] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers)," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 1–12, ACM, 2012.

[83] M. Grossman, A. Simion Sbîrlea, Z. Budimliíc, and V. Sarkar, "CnC-CUDA: Declarative Programming for GPUs," in *Languages and Compilers for Parallel Computing*, vol. 6548 of *Lecture Notes in Computer Science*, pp. 230–245, Springer Berlin Heidelberg, 2011.

[84] D. Cunningham, R. Bordawekar, and V. Saraswat, "GPU Programming in a High Level Language: Compiling X10 to CUDA," in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, (New York, NY, USA), pp. 8:1–8:10,

ACM, 2011.

[85] U. Kremer, "NP-completeness of Dynamic Remapping," Tech. Rep. CRPC-TR93330-S, Rice University, Houston, Texas, USA, 1993.

[86] R. Bixby, K. Kennedy, and U. Kremer, "Automatic Data Layout Using 0-1 Integer Programming," in *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94*, pp. 111–122, 1994.

[87] J. M. Anderson and M. S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," in *In Proceedings Of The SIGPLAN '93 Conference on Programming Language Design And Implementation*, pp. 112–125, 1993.

[88] C. Ding and K. Kennedy, "Improving effective bandwidth through compiler enhancement of global cache reuse," in *Parallel and Distributed Processing Symposium., Proceedings 15th International*, p. 10, Apr 2001.

[89] V. De La Luz and M. Kandemir, "Array regrouping and its use in compiling data-intensive, embedded applications," *Computers, IEEE Transactions on*, vol. 53, pp. 1–19, Jan 2004.

[90] I.-J. Sung, G. Liu, and W.-M. Hwu, "DL: A data layout transformation system for heterogeneous computing," in *In Proceedings of Innovative Parallel Computing*, InPar'12, pp. 1–11, May.

[91] K. Jeff, J. Terry, and Q. Dan, "TALC: A Simple C Language Extension For Improved Performance and Code Maintainability," in *Proceedings of the the 9th LCI International Conference on High-Performance Clustered Computing*, 2008.

[92] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for CPU-GPU architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO'12, 2012.

[93] 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 *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI'11, 2011.

[94] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, "Static Compilation Analysis for Host-Accelerator Communication Optimization," in *Languages and Compilers for Parallel Computing* (S. Rajopadhye and M. Mills Strout, eds.), vol. 7146 of *Lecture Notes in Computer Science*, pp. 237–251, Springer Berlin Heidelberg, 2013.

[95] C.-W. Tseng, "An Optimizing FORTRAN D Compiler for MIMD Distributed-Memory Machines (Ph.D. thesis)," Tech. Rep. CRPC-TR93291-S, Rice University, Houston, Texas, USA, 1993.

[96] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[97] M. F. P. O'Boyle, Z. Wang, and D. Grewe, "Portable mapping of data parallel programs to OpenCL for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, (Washington, DC, USA), pp. 1–10, IEEE Computer Society,

2013.

[98] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT '13, (Piscataway, NJ, USA), pp. 245–256, IEEE Press, 2013.

[99] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable performance on heterogeneous architectures," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, (New York, NY, USA), pp. 431–444, ACM, 2013.

[100] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a Data-flow Programming Model Onto Heterogeneous Platforms," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, (New York, NY, USA), pp. 61–70, ACM, 2012.

[101] A. Moreton-Fernandez, A. Gonzalez-Escribano, and D. Llanos, "Exploiting distributed and shared memory hierarchies with Hitmap," in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pp. 278–286, July 2014.

[102] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. Llanos, "An Extensible System for Multilevel Automatic Data Partition and Mapping," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, pp. 1145–1154, May 2014.

[103] J. Kim, S. Seo, and J. Lee, "An Efficient Software Shared Virtual Memory for

the Single-chip Cloud Computer," in *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, (New York, NY, USA), pp. 4:1–4:5, ACM, 2011.

[104] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, (New York, NY, USA), pp. 287–296, ACM, 2008.

[105] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, "Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, (New York, NY, USA), pp. 156–166, ACM, 2007.