COMPILER AND RUNTIME TECHNIQUES FOR BULK-SYNCHRONOUS PROGRAMMING MODELS ON CPU ARCHITECTURES

BY

HEE-SEOK KIM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Professor Wen-Mei Hwu, Chair
Professor Stephen Boppart
Professor Deming Chen
Associate Professor Steven Lumetta

ABSTRACT

The rising pressure to simultaneously improve performance and reduce power consumption is driving more heterogeneity into all aspects of computing devices. However, wide adoption of specialized computing devices such as GPUs and Xeon Phis comes with a programming challenge. A carefully optimized program that is well matched to the target hardware can run many times faster and more energy efficiently than one that is not. Ideally, programmers should write their code using a single programming model, and the compiler would transform the program to run optimally on the target architecture. In practice, however, programmers have to expend great effort to translate performance enjoyed on one platform to another. As such, single-source code-based portability has gained substantial momentum and OpenCL, a bulk-synchronous programming language, has become a popular choice, among others, to fulfill the need for portability. The assumed computing model of these languages is inevitably loosely coupled with an underlying architecture, obligating a combined compiler and runtime to find an efficient execution mapping from the input program onto the architecture which best exploits the hardware for performance.

In this dissertation, I argue and demonstrate that obtaining high performance from executing OpenCL programs on CPU is feasible. In order to achieve the goal, I present compiler and runtime techniques to execute OpenCL programs on CPU architectures. First, I propose a compiler technique in which the execution of fine-grained parallel threads, called *work-items*, is collectively analyzed to consider the impact of scheduling them with respect to data locality. By analyzing the memory addresses accessed in a kernel, the technique can make better decisions on how to schedule work-items to construct better memory access patterns, thereby improving performance. The approach achieves geomean speedups of 3.32× over AMD's and 1.71× over Intel's state-of-the-art implementations on Parboil and Rodinia

benchmarks. Second, I propose a runtime that allows a compiler to deposit differently optimized kernels to mitigate the stress on the compiler in deriving the most optimal code. The runtime systematically deploys candidate kernels on a small portion of the actual data to determine which achieves the best performance for the hardware-data combination. It exploits the fact that OpenCL programs typically come with a large number of independent work-groups, a feature that amortizes the cost of profiling execution of a few work-items, while the overhead is further reduced by retaining the profiling execution result to constitute the final execution output. The proposed runtime performs with an average overhead of 3% compared to an ideal/oracular runtime in execution time.

To my family.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Professor Wen-mei Hwu, for his amazing support and guidance throughout my PhD. He has provided me with invaluable advice, and I cannot imagine completing this work without his patience and motivation. He is one of the smartest people I know and a true gentleman I really want to resemble. I would also like to express my appreciation to the rest of my committee: Professor Stephen Boppart, Professor Deming Chen, and Professor Steven Lumetta for their insightful comments and collaboration through my study.

During my study, I have been privileged to work with many brilliant people. Getting to know Hong-Seok Kim was a lifetime event for me and I cannot imagine how my life would have been without him. He is an exceptional mentor and I am sincerely thankful to him for his help through the course of my life. I am deeply indebted to John Stratton who opened the door to the portable performance research in the heterogeneous era, the subject of my work. Izzat El Hajj is an extraordinary researcher, engineer and friend, and collaboration with him resolved many challenging problems I could not have solved alone. Li-Wen Chang is a good classmate and office mate who always stimulated me with a new idea and an instant solution to a research problem.

My sincere thanks go to the IMPACT group members and colleagues, particularly Christopher Rodrigues, I-Jui Sung, and Nasser Ansari (I miss you a lot) for useful discussions, advice and encouragement. I am grateful to Thomas Jablin for his sharp judgment about research ideas and tremendous advice in paper writing. I would also like to thank Xiao-Long Wu, Steven Wu, Simon Garcia de Gonzalo, Carl Pearson, Abdul Dakkak, Jie Lv, Sitao Hwang, Nady Obeid and John Larson. Marie-Pierre Lassiva-Moulin has always been helpful and friendly with my miscellaneous requests and on top of that I cannot be more thankful that I have such a nice lady as the godmother of

my daughter. Jan Progen, who patiently helped me polish the thesis, is one of the friendliest faces I met in the new ECE building. I am also deeply thankful to Jamie Hutchinson in the ECE Editorial Services for professional quality correction.

Friends provided me with relaxation and support, and their untarnished friendship kept me going when I was faced with troubles during my study. Lions Club, Outsider Club and Disciplined Life friends share a strong friendship from the good old days when we were undergrad students, though we all now live in different places around the world. I will be missing the time with Wooil Kim, when we would sneak away from the professors and have various discussions at a coffee shop. Dan and Xiaolin, who always welcomed me and my family just like family with beer and ice cream, are the best neighbors I have had.

Finally, I would like to express my sincere gratitude to my family for their support. Hyo Hoon Jeong, my greatest lover and best friend, is probably the only one who tolerates my quirkiness and other shortcomings. I would not have completed this work without her love, encouragement, understanding and support. My sisters Young-Joo and Hyun-Joo cheered me from Korea as they always did in my life. I would also like to recognize the invaluable contributions from my parents and parents-in-law who have provided unconditional love and support. Lastly, but not the least, I would like to thank Angela and David, my beloved daughter and son, who are the reason for me to open my eyes in the morning.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AST	Abstract Syntax Tree
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
CU	Compute Unit
CG	Conjugate Gradient
DSP	Digital Signal Processor
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
JDS	Jagged Diagonal Storage
LLVM	Low Level Virtual Machine
MKL	Math Kernel Library
PDE	Partial Differential Equation
PMU	Performance Monitoring Unit
SM	Streaming Multiprocessor
TBB	Thread Building Blocks
TLB	Translation Lookaside Buffer
VLIW	Very Long Instruction Word

CHAPTER 1

INTRODUCTION

The demand for computing devices with increased performance at reduced energy budget continues to grow. In the mobile community, such devices enable more functionality and longer battery life. In the high-performance computing community, such devices make exascale computing possible. As performance improvements from the semiconductor fabrication process diminish, architects are compelled to introduce more diversity into all aspects of computing devices: function units, interconnect fabrics, and memory hierarchies. Modern computing systems are thus transitioning to heterogeneous platforms, integrating both CPUs and other types of accelerators such as GPUs, FPGAs, and Xeon Phis.

Programmers of these devices must understand and exploit a wider set of architectural entities to achieve high performance, way beyond the traditional instruction set architecture and uniform memory space. CUDA and OpenCL, for instance, are designed for highly parallel execution based on lightweight threads, but desirable performance often requires carefully crafted work assignment to threads, multi-level tiling, and scheduling of the threads [1, 2]. While obtaining high performance on one device is challenging on its own, providing code that can achieve high performance across a diverse set of devices is a daunting, tedious task for even the most skilled programmers.

It is therefore desirable to support *performance portability* [3] over different device architectures. One fundamental challenge with targeting heterogeneous platforms is maintaining multiple source code versions optimized for different platforms to achieve portable performance. Ideally, programmers would write their code using a single programming model, and the compiler would transform the program to run optimally on the target architecture. Features specific to one particular architecture cannot be found and easily exploited universally. Thus the language for performance portability must have a common programming model which can be compiled to many archi-

tectures in a way that produces fast and low-power executable code. With this approach, a set of architecture-specific compiler and runtime to run a program written in a portable language, called *stack*, must be designed to consider detailed facts about the target architecture.

Several emerging languages such as OpenCL and C++AMP are designed for portability over heterogeneous systems. They adopt fine-grained thread-level parallelism for the common programming model, which is a sensible decision because parallelism has become the main source of performance scaling in many emerging applications. Heavily motivated for GPU programming, such languages are used primarily to program GPUs in which the assumed programming model matches well with the underlying architecture. CPUs, however, employ less a parallelism than GPUs, forcing serialization of the execution of the independent workload. Major concerns with the serialization are finding the right criteria for performance and implementing the serialization as efficiently as possible. The serialization criterion dictates scheduling of instructions from many threads so as to maximize a property which has a high impact on performance such as data locality or instruction throughput. The implementation concern is the engineering effort to realize the serialized execution, and as an example the execution of fine-grained threads can be done using CPU threads or a loop that iterates through their work.

Unfortunately, techniques to achieve high performance on a CPU using portable languages do not seem mature yet. One fundamental reason is that the portable languages commonly lack a guide to implement portable performance on platforms other than a GPU. For instance, AMD's and Intel's OpenCL stacks for CPUs are meant to execute OpenCL programs with the same functionality, but their performance varies significantly, mainly due to their distinct design goals [4, 5]. Implicit assumptions for one architecture are not transferable to others, and the expected optimization effect based on the assumption is not universally observable in other architectures [6, 7]. As such, tuning the same program differently by programmers for individual devices and platforms has become a norm in pursuit of performance [8, 9, 10].

In this dissertation, I argue that it is feasible to obtain high performance from executing programs written in OpenCL, a popular portable language. I first demonstrate that a fixed scheduling of work-items execution ingrained by the conventional OpenCL compilers substantially contributes to low performance by being oblivious to data locality and its impact on performance.

To overcome this problem, I propose an OpenCL compiler that performs data-locality-centric work-item scheduling. By analyzing the memory addresses accessed within a kernel, the technique can make better decisions on how to schedule work-items to construct better memory access patterns, thereby improving performance. This technique is particularly useful when the input program contains loops, which are primary sources of large working sets posing a memory performance challenge. Also, I propose a code generation technique that implements the scheduling method. This technique includes the creation of scheduling boundaries, transformation of a region for preferred scheduling, and vectorization. The proposed method works in the presence of control divergence with both types of conditional and loop. A fully working prototype is implemented to demonstrate and evaluate the idea and performance measurement on real hardware is conducted. The approach achieves geomean speedups of 3.32× over AMD's and 1.71× over Intel's state-of-the-art implementations on Parboil and Rodinia benchmarks.

This work is done with OpenCL because of its popularity, openness, large user bases in both the application and support from hardware vendors with potential impact on the community. OpenCL has features making it an eligible language to demonstrate performance portability, which is explained more in Chapter 2. The technique presented in this dissertation can also be applicable to other languages so long as they share programming and execution models similar to that of OpenCL.

1.1 Motivation

1.1.1 Lack of OpenCL Performance on CPU

Presently, it is common to experience the poor performance of OpenCL programs on CPUs. Figure 1.1 compares the performance of three BLAS kernels of **saxpy**, **sgemv** and **sgemm** in OpenCL. Execution time is measured on AMD and Intel OpenCL stacks, which is normalized to the result of the functionally equivalent implementation from Intel MKL library, a highly tuned mathematical library for CPU. More details on the code, experimental setup and analyses can be found later in Chapter 6.

One observation is that the current state-of-the-art OpenCL compilers

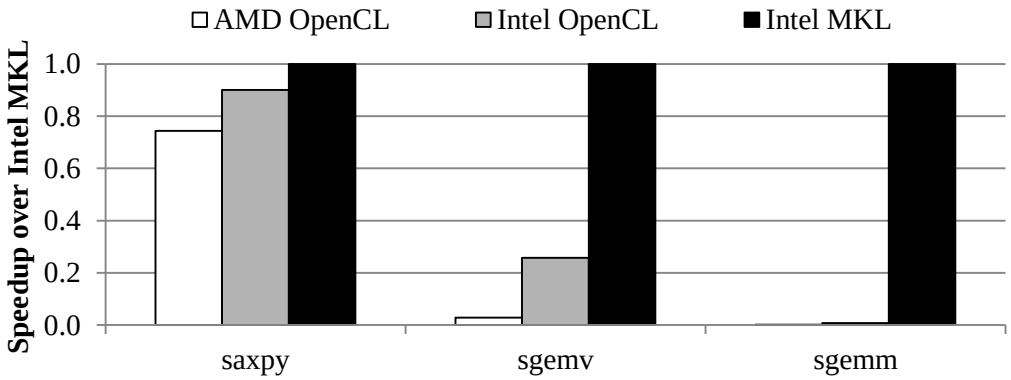


Figure 1.1: Performance of BLAS kernels written in OpenCL using AMD and Intel OpenCL stacks, which is compared to the performance of equivalent implementation in Intel MKL library. More details on the experimental setup are described in Chapter 6.

yield far from desired performance. The performance gap is particularly large with **sgemv** and **sgemm**, in which multidimensional data are used and a sizable working set is thus involved. Another observation is the performance disparity between the two CPU OpenCL stacks due to different criteria in their design, which one must consider when arguing about the performance portability. While Intel's result is much more favorable than AMD's, it is hard to make a precise judgment on performance portability, qualitatively and quantitatively, without a deep understanding of how these compilers are architected.

A potentially worrisome consequence of the observations is the hardening of a myth that portable performance based on a common language is not feasible or reliable. Because CPUs are arguably the forefront target to demonstrate performance portability, inability to show evidence for them could lead to a strongly biased opinion when targeting architectures other than CPUs. Ultimately, the low OpenCL program performance on CPUs fosters expensive solutions by making programmers hold on to the conventional idea that each device requires its own optimized programs with preferred languages.

1.1.2 Underutilized CPU in Heterogeneous Platforms

It is utterly wasteful not to utilize CPU for useful workload processing. However, the lack of performance in OpenCL programs on CPUs diminishes the role of CPUs in heterogeneous systems. Modern CPUs are in fact essential

computing resource for performance computing, and are therefore an eligible target for OpenCL programs as well as other portable languages. Though the absolute performance of CPUs is overshadowed by what specialized computing devices are capable of, the advantages of CPUs such as versatility, ubiquity, and performance should not be simply overlooked. Moreover, the performance of CPUs continues to increase by integrating more cores and embracing wide vector execution units. For instance, Table 1.1 compares hardware specifications of two recent generations of CPUs and GPUs from Intel and NVIDIA, respectively. The table shows that the ratios of computation throughput and memory bandwidth between CPU and GPU are up to 2.35× and 3.67×, respectively, which are further reduced when normalized with power consumption. Researchers have shown that a properly optimized CPU code can effectively be used for throughput computing, closing the performance gap between CPU and GPU [11].

Table 1.1: A brief comparison of current high-end CPUs and GPUs.

	Intel Core i7-5820K	Intel Xeon E5-2687W v3	NVIDIA Tesla C2050	NVIDIA Tesla K40
Price	~$400	~$2,000	~$1,000	~$3,000
GFlops (in double precision)	317	496	515	1,170
Bandwidth (GB/s)	68	68	144	250
Power consumption (W)	140	160	238	225

1.2 Summary of Contributions

The following list summarizes the contributions of this disseration.

- I argue that the widespread belief that performance portability does not exist for OpenCL programs on CPUs is in fact due to immature compiler techniques available today. In particular, I demonstrate that

the lack of concern for data locality in designing OpenCL compilers constitutes a substantial fraction of the low performance, by discovering the relationship between the conventional schedule and its implication for data locality and performance.

- I propose an alternative schedule that is better suited to data locality than the conventional schedule in practice.
- An adaptive method is introduced in order to construct a data-locality-friendly schedule by statically analyzing memory access patterns. A code generation technique implementing the schedule is also proposed.
- A complete OpenCL compiler is implemented and its performance is evaluated on real hardware to verify the proposed ideas.
- A runtime technique to aid a compiler is proposed so as to relieve the burden placed on the compiler by having to pick the optimal code by building a performance model with high accuracy. Instead, the proposed technique allows a compiler to deposit several differently optimized codes, each of which is individually evaluated at runtime with a fraction of real input data to determine the optimal one for the rest of workload processing.

1.3 Organization of This Dissertation

The rest of this dissertation is organized as follows. Chapter 2 summarizes the OpenCL programming model and previous approaches for OpenCL compiler on CPUs along with their drawbacks in terms of data locality. It also details the reason for the performance disparity between GPUs and CPUs, along with the consequence of how people tune their programs for CPU performance. Chapter 3 analyzes the drawbacks of previous approaches and proposes a better schedule approach and a compiler technique to derive a schedule that is data-locality aware. Chapter 4 details a code generation technique for implementation of such an OpenCL compiler. Chapter 5 evaluates the proposed technique against the state-of-the-art implementations. The performance of the proposed method is evaluated in the context of portable performance using well-known algorithms from BLAS, which is discussed in

Chapter 6. Chapter 7 presents runtime support to aid the compiler in deriving the most optimal kernel variant, when the compiler cannot make an optimal decision due to limitations of static performance modeling. Finally, Chapter 8 summarizes this work and offers a conclusion.

CHAPTER 2

BACKGROUND

2.1 OpenCL Overview

OpenCL [12] is general purpose parallel programming language targeting CPUs, GPUs and other discrete computing devices organized into a single platform. Developed by Khronos Group, an industry consortium, it is considered one of the most prominent parallel programming languages today. OpenCL is endorsed by both hardware vendors and application developers. Virtually all GPUs can run OpenCL programs [13, 14, 15, 16, 17]. Also, most CPU architectures have either vendor-provided OpenCL implementations [18, 19] or open source projects that support them [20, 21]. Recently, support for other types of architectures such as DSPs and FPGAs has become available [22, 23]. The rich foundation encourages many applications to adopt OpenCL such as MAGMA [24] and OpenCV [25].

OpenCL provides a unified interface for diverse device architectures. The abstract computing model of OpenCL is intended to be architecture neutral, enabling functional equivalence across architectures. The bulk-synchronous programming model assumes an abstract device architecture composed of multiple *compute units*, each consisting of multiple *computing elements*. The program is organized into multiple *work-items* which are grouped into *work-groups*. Work-items within a work-group execute on a single compute unit and can synchronize and share memory with each other. Work-groups execute independently on different compute units and cannot synchronize with each other. It is largely the vendor's responsibility to map the abstract computing model to the physical execution resources.

The computing model of OpenCL can be easily mapped onto GPU execution resources. Contemporary GPUs are composed of dozens of independent cores, each of which can run thousands of concurrent threads. The core im-

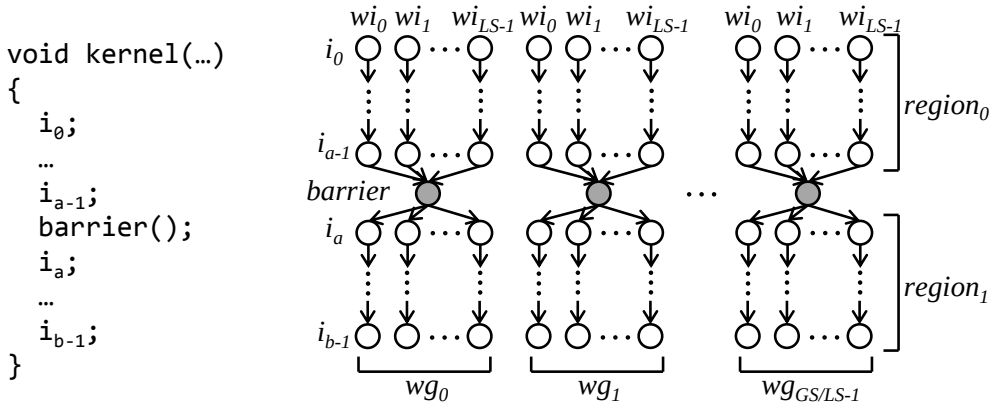
plements control and memory synchronizations for threads it manages. Each core is equipped with its local memory which can be shared among threads running within the core. Thus, it is intuitive to map computing units to cores and computing elements to threads in a core. SM (Simultaneous Multiprocessor) in NVIDIA GPU or CU (Compute Unit) in AMD GPU are corresponding implementations of the core.

The execution of threads in a GPU core is done using several concurrent execution entities, called *warps* in NVIDIA GPU or *wavefronts* in AMD GPU. Warps or wavefronts share the same instruction for all threads mapped on them, making progress in a lock-step manner. They are subject to scheduling by GPU hardware and long latency operations can be tolerated when multiple warps or wavefronts are available. Although the OpenCL specification does not regulate hardware implementations, OpenCL programmers often exploit the fact that neighboring work-items run concurrently because of warps or wavefronts. For example, a hardware support that converts multiple consecutive memory requests issued within a warp or wavefront into one bulk request, called *coalesced memory access*, is a particularly useful feature when memory bandwidth is concerned. The memory bandwidth utilization can be high when a GPU program takes advantage of coalesced memory access, because multiple memory accesses by work-items can be handled with fewer outstanding memory requests, which one must consider for memory performance.

2.2 OpenCL Execution Model

Launching an OpenCL kernel requires kernel code and index space. Kernel code is a user-provided program that runs on each work-item. Index space of N defines a dimension of work-items ranging from zero to $N-1$. Figure 2.1 shows an example kernel code and its execution over an index space of GS with an optional work-group size of LS. The work-items are equally divided into work-groups such that work-items within a work-group can synchronize using *barrier* instructions while work-items in different work-groups cannot.

Figure 2.1(b) depicts a dependence graph in the execution of the kernel where each circle represents a set of closely related instructions such as basic blocks, and arrows indicate the immediate dependencies between the circles.



(a) Kernel code. (b) Dependency graph in the execution of the kernel code.

Figure 2.1: An example OpenCL kernel code and its dependence graph for execution of the kernel. Immediate dependencies occur between dynamic instructions or instruction blocks (i) in OpenCL kernels. Each work-group (wg) contains local size (LS) work-items (wi). Work-items in different work-groups execute independently until completion. Work-items in the same work-group synchronize at barriers. Barriers divide the program into code regions within which work-items execute independently. Work-item independence within regions provides great flexibility for work-item scheduling.

Note that the arrows do not represent memory dependencies. The graph is conservative because the arrows indicate all dependencies that *may* exist between instructions for some program. The absence of an arrow between instructions indicates independence that is guaranteed by the programming model. Crafting an OpenCL compiler therefore boils down to finding a mapping between the dependence graph and a set of computing resources, often comprised of multi-cores and vector execution units in CPUs.

One observation is that work-groups are completely independent because there is no path connecting work-items in different work-groups. This property allows all work-groups to run concurrently, independently, and in any order. All existing implementations as well as the implementation presented in this dissertation handle work-groups by scheduling them in distinct CPU threads. Thus, CPU threads do not need to synchronize until kernel completion, which is convenient because synchronization across threads on a CPU is expensive. At this level, an important criterion in distributing work-groups over CPU cores is load balancing. A runtime library that has a strong support for load balancing, such as Intel TBB [26], thus would be an eligible

platform for the execution of work-groups.

Another observation is that dependencies between work-items within a work-group are only introduced by barrier instructions. Therefore barriers divide the kernel into *regions* such that work-items within a region execute independently. Existing approaches employ a *region formation* [27] algorithm to divide up the kernel into barrier-separated regions. Regions are meant to run one after another within a work-group, thus scheduling freedom does not exist.

The remaining problem to be tackled is the scheduling of work-items within a region. Given its huge degree of scheduling freedom, scheduling of work-items within a region can have a large impact on data locality because it directly impacts the order of memory accesses. On GPUs, scheduling is dictated by the hardware. The GPU notion of warps in NVIDIA GPU (or wavefronts in AMD GPU) enforces that a sub-group of work-items in a work-group executes the same instruction before moving on to the next. Moreover, the current warp scheduler controls the execution of warps. Since the programmer has little control over instruction scheduling, it becomes incumbent on the programmer to adapt their code and data structures to the anticipated hardware scheduling policy for better data locality. Such adaptations are the subject of many GPU optimizations such as data layout transformation, memory coalescing, and dynamic tiling [28, 29, 30].

On the other hand, the CPU hardware is not actively involved with the scheduling of work-item instructions within a region, leaving instruction scheduling up to the compiler and runtime. This allows the compiler to adapt its scheduling to the code to achieve the best memory access pattern, alleviating the programmer's burden to optimize for data locality.

2.3 Previous Approaches

There is a wealth of literature on compiling OpenCL programs for CPUs. To the best of my knowledge, no existing implementations [5, 13, 20, 21] consider the impact of work-item scheduling on data locality. Prior approaches only consider correctness and instruction throughput when scheduling work-item instructions. The following categorizes the alternative approaches by the implementation of work-item scheduling.

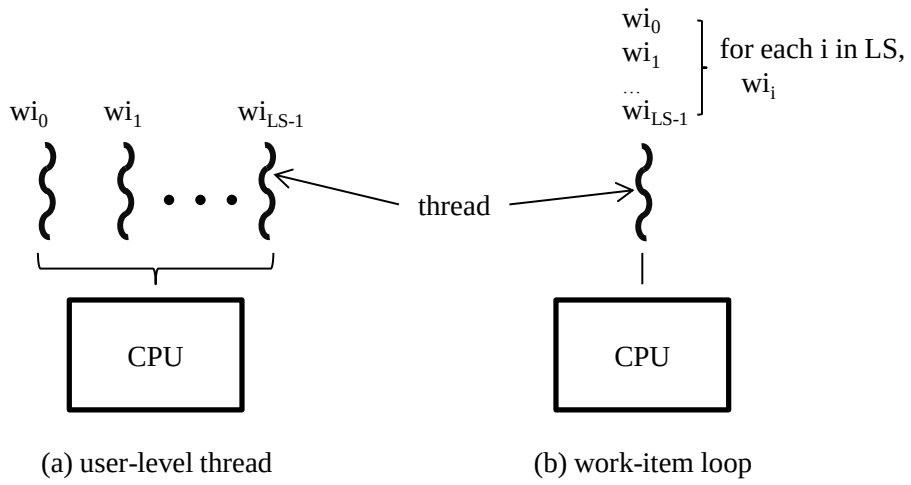


Figure 2.2: Previous approaches in mapping execution of work-items.

User-level thread approach. User-level threads are assigned to each work-item [13], which is shown in Figure 2.2(a). This approach has the advantage of moving work-item scheduling into the runtime instead of relying on compiler techniques. The scheduling unconditionally executes a work-item until the region boundary and moves on to the next, oblivious to potential data locality across work-items. Additionally, this approach hinders important performance optimizations such as vectorization and redundancy elimination among work-items. This approach also suffers from having to maintain many threads for work-items of fine-grained workload, a situation where the threading overhead is not negligible.

Work-item loop approach. The compiler inserts loops around each region that iterate over all work-items in a work-group [18, 20, 31, 32], as shown in Figure 2.2(b). The advantage of using work-item loops as opposed to the user-level thread approach is that it enables compiler optimizations, which are important for performance. One such optimization is selective replication [27]. In the presence of barriers, OpenCL variables need to be replicated for each work-item so that all work-items can run concurrently. However, replication is unnecessary for uniform variables (variables having the same value for all work-items) and variables whose lifetime is confined to a region. With work-item loops, replication is done via scalar expansion of variables into arrays. It can therefore be selectively avoided by keeping candidate variables scalar. With the user-level thread approach, the work-

item context is replicated unconditionally as a whole, and therefore cannot be selectively avoided for individual variables.

Another important optimization is strip-mining of work-item loops to benefit from SIMD vectorization for maximizing instruction throughput. One approach [20] does so by annotating the work-item loops using LLVM parallel loop annotations. Other approaches [18, 32] do so explicitly using vector instructions. Generating a high-quality vectorized code from a region requires sophisticated compiler analyses and transformations, particularly when control divergence is observed. The vectorization of work-items execution is a subject of many studies [32, 33, 34].

The vectorized work-item loop approach approach, however, also results in suboptimal data locality. Many regions contain loops, which the work-item loop approach wraps in an outermost loop across work items. Having the work item loop be outermost in these loop nests, however, does not always produce good memory locality. Often, OpenCL programs are written for GPUs in the first place and their fixed scheduling policy assumes that the work-item loop is placed at the innermost level. Although some approaches [18, 32] do benefit from vectorization for higher performance, their scheduling is largely similar to the user-level thread approach in terms of data locality when dealing with a large working set.

2.4 Locality of Previous Approaches

Scheduling of instructions dictates memory access pattern. When an OpenCL kernel deals with large multidimensional data where working set management has a substantial impact on performance, a memory access pattern generated by a program needs to exploit memory system architecture with efficiency. However, previous OpenCL stack implementations for CPU unconditionally employ a fixed scheduling, regardless of scheduling concern for memory performance.

Figure 2.3 compares memory access patterns of an example OpenCL code for GPU and CPU. The code snippet is from StreamCluster in Rodinia benchmark suite [35], which is discussed as an exemplar to demonstrate that performance is not portable on CPU in a previous work [7].

The code shown in Figure 2.3(a) is reasonably well-tuned for GPU with

```
__kernel void pgain_kernel(…) {  // in OpenCL
  int thread_id = get_global_id(0);
  ...
  float x_cost = 0.0;
  for(int i = 0; i < dim; ++i) {
    x_cost += (coord_d[(i*num)+thread_id] – coord_s[i]) *
              (coord_d[(i*num)+thread_id] – coord_s[i]);
  }
  ...
}
```

(a) StreamCluster code snippet.

```
coord_d[0*num + 0..NumThreadInWarp-1]  coord_s[0]
coord_d[1*num + 0..NumThreadInWarp-1]  coord_s[1]
coord_d[2*num + 0..NumThreadInWarp-1]  coord_s[2]
                  ...                      ...
```

(b) Memory address trace on GPU for coord_d and coord_s.

```
void pgain_kernel_cpu(…) { // in C
  ...
  for thread_id in LS,     // iteratively execute work-items
    float x_cost = 0.0;
    for(int i = 0; i < dim; ++i) {
      x_cost += (coord_d[(i*num)+thread_id] – coord_s[i]) *
                (coord_d[(i*num)+thread_id] – coord_s[i]);
    }
    ...
}
```

(c) Translated code snippet using work-item loop.

```
coord_d[0*num + 0]          coord_s[0]
coord_d[1*num + 0]          coord_s[1]
coord_d[2*num + 0]          coord_s[2]
        ...                     ...
```

(d) Memory address trace on CPU for coord_d and coord_s from the translated code.

Figure 2.3: An example to show memory access pattern via the work-item loop approach.

respect to memory performance. Many GPU-tuned programs are optimized for coalesced memory access in GPU, which is properly exploited in loading `coord_d` in the example code. Coalesced memory access assumes that execution across work-items has higher priority than the instructions order. Therefore, the memory access pattern for coalesced memory access shows consecutive address per warp or wavefront. In this particular example, the memory address per warp or wavefront in loading `coord_d` starts from `i*num` and increases up to the number of threads packed in a warp or wavefront (denoted as `NumThreadInWarp` in the figure), which is repeated for subsequently scheduled warps or wavefronts. Also, `coord_s[i]` is reused for work-items because the value of `i` is the same in a warp or wavefront. In other words, spatial locality and temporal locality exist in accessing `coord_d` and `coord_s`, respectively. The memory access pattern for the first few memory operations is shown in Figure 2.3(b), which assumes only one warp or wavefront is available.

The translated code using the work-item loop approach is shown in Figure 2.3(c), which sequentially executes work-items in a work-group on a CPU thread. With the translation, a loop is created to surround the code for serializing execution of the code over work-items. Figure 2.3(d) lists the first few memory addresses when the translated code runs. The addresses of the consecutive memory accesses are regularly spaced by a variable called `num`, which in fact is a large stride. Since `i` changes faster than `thread_id`, the memory access pattern makes a much larger stride than typical cache line size, resulting in poor cache line utilization. It would also suffer from frequent data TLB misses and a penalty associated with it. For example, when `num` multiplied by the element size of the load operation is equal to or larger than the page size, the memory access pattern shown in Figure 2.3(d) may cause data TLB miss every time loading `coord_d` gets executed. As for `coord_s`, sequential access occurs which exploits spatial locality; however, the GPU execution is more efficient due to temporal locality among work-items.

The comparison shows that the memory access pattern from the translated code does not match with what a typical programmer would expect on GPU. Moreover, the comparison reveals that a substantially suboptimal memory access pattern takes place with the previous approach for CPU execution. The disparity between data locality in well-tuned GPU programs and the resulting schedule from the previous work largely explain the lack of

performance on CPU for programs dealing with a sizable working set.

Researchers have experienced the symptom, and data layout transformation is often suggested as an antidote to the problem in order to adapt to the memory access pattern [4, 7]. However, data layout transformation is an unacceptable solution in practice due to high cost and difficult deployment. Data layout transformation touches the entire memory object at least once, which is non-trivial overhead for both execution time and power consumption. Out-of-place transformation, where a separate output is allocated for the transformation, obligates management of an additional storage. In case of in-place transform, where the same memory object is used but memory elements are shuffled within the object, the number of movements per element is several times higher than the out-of-place transform when a high-performance implementation [29] is concerned. Also, data layout transformation algorithms typically assume that the dimension of the data is known. However, retrieving the dimension of a memory object is not trivial according to OpenCL programming interface, because it uses a C-like pointer to create, deliver and use a memory object, but no dimension information is delivered separately.

CHAPTER 3

LOCALITY-CENTRIC SCHEDULING

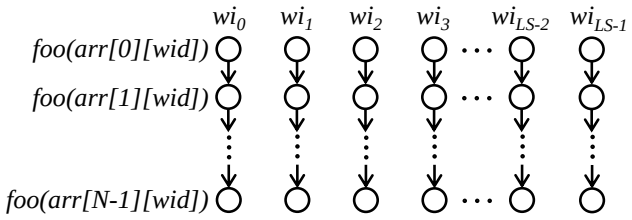
In the previous chapter, I demonstrated that traditional work-item scheduling is not always good for locality. An alternative schedule is suggested that performs better for applications having particular classes of memory accesses. A selection algorithm is introduced that picks the schedule likely to result in better locality based on a static analysis of the memory access patterns.

3.1 Depth-First Order and Breadth-First Order Scheduling

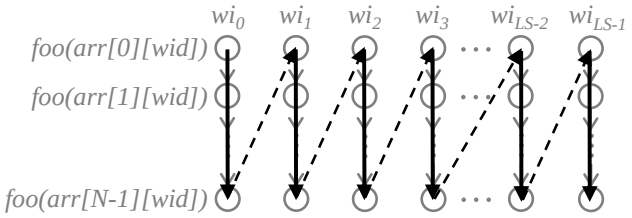
The example in Figure 3.1 demonstrates the effect of work-item scheduling on locality. Figure 3.1(b) depicts the dependence graph of the code in Figure 3.1(a), where each white circle represents a dynamic instruction block from a single loop iteration. If the traditional work-item scheduling is used to execute this region, each work-item executes the region to completion before the next work-item begins, as shown in Figure 3.1(c). Such a traversal is suboptimal because it results in a sequence of memory loads having a large stride. A better traversal of loads can be achieved by scheduling the work-items as shown in Figure 3.1(d). Such a traversal results in the largest number of unit stride accesses. The proposed technique focuses on regions containing loops because loops are the source of the longest running regions having working sets large enough such that locality is a major concern. Among the 30 Parboil and Rodinia benchmarks, 18 of them have loops within kernel regions. These 18 benchmarks will be used to evaluate the approach. The main question is whether to schedule a work-item to execute an entire region before the next work-item begins (the approach taken by existing compilers), or to schedule all work-items to execute the same loop iteration before moving on to the next (the alternative approach shown in

```
wid = get_local_id(0);
for(k=0; k < N; ++k){
  foo(arr[k][wid]);
}
```

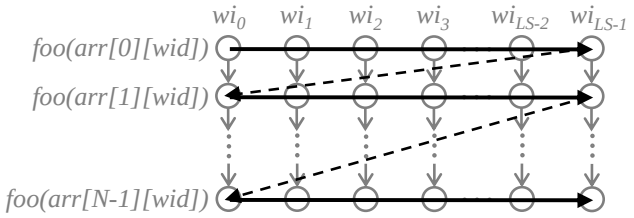
(a) Simple example of OpenCL code region.



(b) Region dependence graph.



(c) Depth-first order (DFO) traversal using traditional work-item loops results in large strided accesses.



(d) Breadth-first order (BFO) traverses array elements with stride 1, in the order stored in memory.

Figure 3.1: A motivating example demonstrating the impact of scheduling on the memory access pattern.

Figure 3.1(d)). These two scheduling techniques are denoted as *depth-first order (DFO)* and *breadth-first order (BFO)* respectively, based on how they traverse the dependence graph.

There is no single approach that fits all. Chapter 5 shows that out of 18 benchmarks, DFO does better for 5 while BFO does better for 13. DFO is well suited for capturing locality among the memory accesses within each work-item, whereas BFO will expose collective memory locality across work-items.

The better schedule choice depends on the memory accesses dominating the loop body.

3.2 Memory Access Classification

Locality-centric (LC) scheduling selects between DFO and BFO based on which technique is predicted to have better locality. The first step is to classify the memory operations inside the loop at compile time. The classification in use is summarized in Table 3.1. This classification is based on two dimensions: loop iteration stride and work-item stride. For each of these dimensions, memory accesses are classified as stride zero, stride one, or other.

Table 3.1: Classification of memory accesses and scheduling decision preferred by each class (if any).

		Work-item Stride		
		0 (W0)	**1** (W1)	**Other** (WX)
Loop Iteration Stride	**0** (L0)	-	DFO	DFO
	1 (L1)	BFO	-	DFO
	Other (LX)	BFO	BFO	-

Stride zero (i.e., invariant) means that the memory access index is the same for all loop iterations or all work-items in a work-group, respectively. Stride one means that the memory access index increases by one for consecutive loop iterations or consecutive work-items respectively. Other means that the memory access index is neither invariant nor stride one. These access types are abbreviated as shown in Table 3.1 where 'W' means work-item, 'L' means loop iteration, '0' means stride zero, '1' means stride one, and 'X' means other.

A class of memory operations favors the schedule resulting in a smaller memory access stride. If a memory access had a smaller stride with respect to the loop index, then it is best traversed when a work-item runs deeply to finish executing a loop before the next work-item begins. Therefore the memory access prefers DFO. If a memory access had a smaller stride with respect to the work-item id, then it is best traversed when a loop iteration executes broadly across work-items before the next iteration begins. Therefore the memory access prefers BFO.

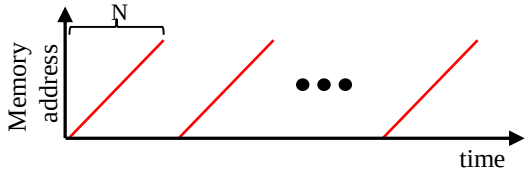
```
for(k=0; k<N; ++k) {
  foo(arr[k]);
}
```

(a) An example region.

		Work-item stride		
		0 (W0)	1 (W1)	Other (WX)
Loop iteration stride	0 (L0)	-	DFO	DFO
	1 (L1)	BFO	-	DFO
	Other (LX)	BFO	BFO	-

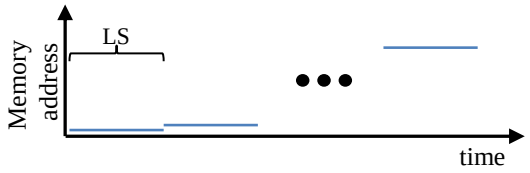
(b) Schedule selection.

```
for each wid in LS,
  for(k=0; k<N; ++k) {
    foo(arr[k]);
  }
```



(c) DFO schedule and memory access pattern.

```
for(k=0; k<N; ++k) {
  for each wid in LS,
    foo(arr[k]);
}
```



(d) BFO schedule and memory access pattern.

Figure 3.2: Memory class example of L1W0.

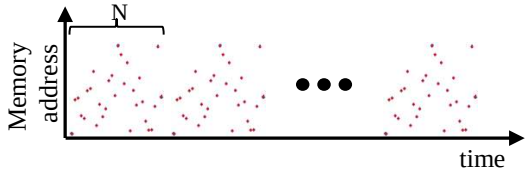
```
for(k=0; k<N; ++k) {
  foo(arr[f(k)]);
}
```

(a) An example region.

		Work-item stride		
		0 (W0)	1 (W1)	Other (WX)
Loop iteration stride	0 (L0)	-	DFO	DFO
	1 (L1)	BFO	-	DFO
	Other (LX)	BFO	BFO	-

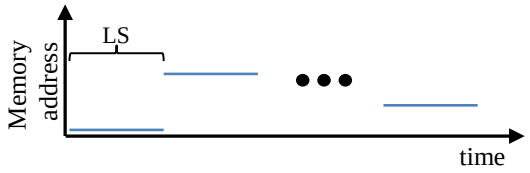
(b) Schedule selection.

```
for each wid in LS,
  for(k=0; k<N; ++k) {
    foo(arr[f(k)]);
  }
```



(c) DFO schedule and memory access pattern.

```
for(k=0; k<N; ++k) {
  for each wid in LS,
    foo(arr[f(k)]);
}
```



(d) BFO schedule and memory access pattern.

Figure 3.3: Memory class example of LXW0.

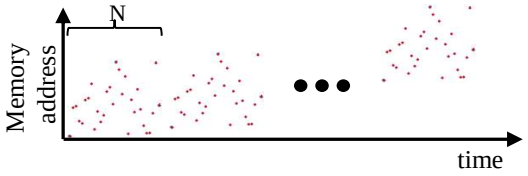
```
for(k=0; k<N; ++k) {
  foo(arr[f(k)+tid]);
}
```

(a) An example region.

		Work-item stride		
		0 (W0)	1 (W1)	Other (WX)
Loop iteration stride	0 (L0)	-	DFO	DFO
	1 (L1)	BFO	-	DFO
	Other (LX)	BFO	BFO	-

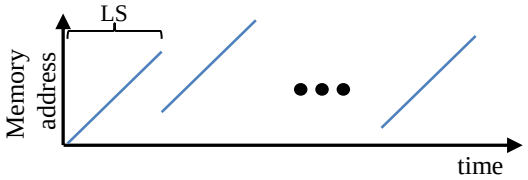
(b) Schedule selection.

```
for each wid in LS,
  for(k=0; k<N; ++k) {
    foo(arr[f(k)+wid]);
  }
```



(c) DFO schedule and memory access pattern.

```
for(k=0; k<N; ++k) {
  for each wid in LS,
    foo(arr[f(k)+wid]);
}
```



(d) BFO schedule and memory access pattern.

Figure 3.4: Memory class example of LXW1.

Figures 3.2, 3.3 and 3.4 show how cache utilization is maximized with the schedule decision. Each figure shows an example code and both schedules of DFO and BFO for the code, accompanied by the memory access pattern for each case. DFO code can be obtained by executing the input code within a canonical loop iterating over the work-item index space. For brevity of explanation, BFO code is presented as if the two loops of the kernel loop and the loop for work-items in the DFO code are interchanged. More detail on the code generation appears later in Chapter 4. For the purpose of the explanation, both LS and N are assumed to be much larger than cache line size so that memory access patterns have an outstanding performance impact. The function `f(k)` is assumed to return a non-linear integer number to `k` with no side effect.

For L1W0, shown in Figure 3.2, memory address for DFO linearly increases up to N and this process is repeated LS times. In conrast, the memory access pattern for BFO remains stationary for LS times and is repeated N times, with incrementally changing unit address. DFO exploits spatial locality while BFO enjoys temporal locality. The overall numbers of cache misses for both cases are $LS \times N/CacheLineSize$ and $N/CacheLineSize$ for DFO and BFO,

respectively. Therefore, BFO must be selected in this case. Due to symmetry, L0W1, which is not shown in the figure, should choose DFO for the same reason but with the two schedules interchanged.

Figure 3.3 illustrates the LXW0 element of our schedule selection table and shows both schedules for the region. Similarly, it shows both schedules for the region. DFO does not expect cache memory reuse while BFO exploits spatial locality. Their cache miss counts are $LS * N$ and N for DFO and BFO, respectively. Similarly, BFO must be chosen in this case and L0WX reverses the situation due to symmetry.

Figure 3.4 compares two schedules for LXW1 in the table. The cache miss counts are $LS * N$ and $N * LS/CacheLineSize$ for DFO and BFO, respectively. The selection logic is similar to previous cases.

Note that prefetching in some cases can reduce the performance difference between the BFO and DFO schedules by hiding memory access latency. For instance, L1W0 shows a sequential memory access pattern when DFO is chosen, as shown in Figure 3.2. A good prefetcher should be able to bring data into the cache in advance so as to minimize the latency for subsequent memory operations. Though it may help to reduce the latency, it still suffers from having to occupy a larger footprint in cache memory and move all the data multiple times, wasting memory bandwidth and energy, and hurting overall system performance.

3.3 Stride Analysis

In order to select a decision from our schedule selection table, one must identify stride values for both loop index and work-item index. In this work, stride analysis factorizes an expression with respect to a variable of interest, loop index variable or work-item. This work focuses on stride-zero (invariant) and stride-one memory accesses because these are most common in practice and sufficient for the proof of concept. However, the same approach can be generalized to any non-unit stride value, which is left for future work. To classify memory accesses, multiple analyses are needed such as loop-invariance analysis, loop index analysis, work-item uniformity analysis, and stride analysis. These analyses are individually solved problems in the literature [36, 37]. The rest of this subsection summarizes the analyses and how

they are used for scheduling selection.

```
int tid = get_local_id(0);
for (i = 0; i < N; ++i) {
  ... = A[i];
  ... = B[n*i + 12];
  ... = C[tid];
  ... = E[A[i] + 4*tid];
}
```

(a) Example kernel loop with memory operations in it.

	factorization for	
index expr	loop index (`i`)	work-item index (`tid`)
`i`	`1*i + 0`	`0*tid + I`
`n*i + 12`	`X*i + 12`	`0*tid + n*i+12`
`tid`	`0*i + tid`	`1*tid + 0`
`A[i] + 4*tid`	`X*i + 4*tid`	`X*tid + A[i]`

(b) Stride analysis result using factorization by 0, 1 and other(X).

Figure 3.5: Stride analysis example.

Loop Induction Variable and Stride Analysis

Loop induction variable detection scans recurring variables in a loop to find induction variables, and the amount of changed value per each iteration is determined as stride for each variable. Figure 3.5 shows an example of the analysis where all base expressions for memory operation such as A and B are assumed invariant to the loop index. The loop has a canonical induction variable of `i`. The variable can be factorized into $1 \cdot$ `i + 0` and thus the stride is determined as 1, or unit stride as shown in the table. This is because the address is incremented by 1 when the `i` value is increased by 1. Next, `n*i` is treated as having unknown stride because `n` is a variable, which is marked as X in the classification table. When the factor of `i` is not involved in an expression, it is regarded as invariant, or as zero stride as shown in the third case. The value of `A[i]` has unknown stride, thus the entire expression gets unknown stride as shown in the fourth case.

Work-item Stride Analysis

Work-item stride analysis can be performed via forward slicing of work-item variables. The work-item variables have a unit stride because index values are defined over consecutive integers (Chapter 2). All others are initialized with zero. When no such expression is used in an expression, as shown in the first and second cases in Figure 3.5, zero stride is given, denoting invariance to work-item values. `tid` in the example is of stride one and `C[tid]` thus shows a consecutive memory access pattern. However, `4*tid` is treated as having unknown stride, though the compiler can identify 4 being a constant integer and compare it against other constant values. As mentioned before, currently 0, 1 and other strides are only used for the stride decision, and any constant values other than 0 and 1 are collectively identified as unknown stride.

Approximate Stride Analysis

A precise stride analysis is not necessary for making scheduling decisions because the stride information is used to inform an optimization decision that does not impact correctness. For this reason, an approximate stride analysis is used when applicable to more aggressively classify the stride. Three operators of modulus, division and select are quite common in OpenCL code when doing index calculations and boundary conditions, but they make it hard to drive a precise stride value. Supporting these common operators with the approximate stride analysis turns out quite useful. The approximate stride analysis differs from the precise stride analysis in the treatment of the three operators:

- Modulus: In the statement $a = b\%N$ where b is stride one and N is arbitrary, a precise stride analysis labels a as unknown. However, a in practice is stride one for the most part. Thus, the result of a modulus on a stride one variable is approximated as stride one.

- Division: In the statement $a = b/N$ where b is stride one and N is arbitrary, the value of a in consecutive threads differs by either 0 or 1. Used as an array index, a will result in a memory pattern that is at least as good as a stride one pattern. For this reason, a can be approximated as stride one.

- Select/Phi functions: In the statement $a = (b > 0)?b : 0$ where b is stride one, the precise stride of a is unknown. However, the footprint created by a as an index is comparable to that of the worse of the two select/phi parameters (in this case, b). The decision is modeled as a semi-lattice (stride-zero $\rightarrow$ stride-one $\rightarrow$ unknown) such that the return value of a select or phi operator is classified according to the meet operation of its operands' classifications.

3.4 Scheduling Policy Selection

Once the classification of each memory access is performed, the number of memory accesses favoring each schedule is tallied, and the schedule with the greater number of tallies is chosen. In the case of a tie, DFO is selected to avoid the overhead of performing BFO, particularly in divergent contexts (discussed in Section 4.3).

Subroutine 1 isBFOLoop(L)

```
DFO = 0, BFO = 0
for every memory access M in L.body do
  switch typeof M:
    case W0L1:  case W0LX:  case W1LX:
      ++BFO
      break
    case W1L0:  case WXL0:  case WXL1:
      ++DFO
      break
if BFO > DFO then
  return true
else if BFO < DFO then
  return false
else        // tie breaker
  return false
```

The decision for the preferred schedule is made on a per-loop basis. Therefore, different loops in the same region could receive different schedules. Moreover, in the case where decisions in loop nests cannot be simultaneously granted, priority is given to the inner loops. Therefore, in the case where a DFO loop contains a BFO loop, the outer DFO loop is scheduled

with BFO to enable the inner BFO loop to be scheduled correctly. This is because the inner loops have more impact on the memory access pattern than the outer loops. The algorithm for performing this scheduling is discussed in the next section. Subroutine 1 shows the algorithm for the decision logic.

3.5 Scheduling Example

Figure 3.6 shows an example kernel code and schedule decision for the code. The figure shows the kernel code in `spmv` (Sparse Matrix-Vector Multiplication) benchmark from Parboil benchmark suite. The code assumes JDS (Jagged Diagonal Storage) format. There are four load operations within the kernel loop and stride values of the memory address expressions are shown in the table. The final decision is to use BFO schedule because there are more BFO-preferred memory operations than DFO-preferred operations.

The table details the stride analysis result and the scheduling decision. The loop index `k` has stride one to the loop but zero stride to the work-item value. The expression of `jds_ptr_int[k]` only depends on loop index of `k`, so the load operation prefers BFO. The value of the expression has unknown stride with respect to loop index. Therefore, `j` has unknown stride to the loop index but unit stride with respect to work-item due to addition of `ix`, which has stride one to work-item. Conveniently, `j` falls into LXW1 in the scheduling decision table. The memory access pattern of `d_index[j]` falls into LXW1 as `j` is used for indexing, thus the expression prefers BFO. Similarly, `d_data[j]` adds another vote to BFO. Lastly, the algorithm cannot decide a preferred schedule for `x_vec[in]`, because `in` is classified as LXWX in the schedule decision table. The value of the variable equals the value of `d_index[j]`, which has unknown stride for both loop index and work-item index.

```
__kernel void spmv_jds_naive(
    __global float *dst_vector, __global float *d_data,
    __global int *d_index, __global int *d_perm,
    __global float *x_vec, const int dim,
    __constant int *jds_ptr_int,
    __constant int *sh_zcnt_int)
{
  int ix = get_global_id(0);
  if (ix < dim) {
    float sum = 0.0f;
    int bound = sh_zcnt_int[ix/32];
    for(int k = 0; k < bound; k++)
    {
      int j = jds_ptr_int[k] + ix;
      int in = d_index[j];
      float d = d_data[j];
      float t = x_vec[in];
      sum += d*t;
    }
    dst_vector[d_perm[ix]] = sum;
  }
}
```

(a) spmv kernel code in OpenCL.

Expression	Loop index stride	Work-item stride	Schedule decision
k	1	0	N/A
jds_ptr_int[k]	X	0	BFO
ix	0	1	N/A
j	X	1	N/A
d_index[j]	X	X	BFO
in	X	X	N/A
d_data[j]	X	X	BFO
x_vec[in]	X	X	Unknown

(b) Summary of the stride analysis for the kernel loop.

Figure 3.6: Schedule decision example using `spmv` kernel. There are three memory operations that prefer BFO and one the compiler cannot tell the scheduling preference. The final decision is therefore BFO.

CHAPTER 4

CODE GENERATION TECHNIQUE

In this chapter, scheduling unit detection and code generation techniques according to the schedule decision are presented. The technique assumes that the input code is structured; in other words, a program is composed of basic blocks, conditionals and loops. The structured program matches with an AST-based representation as it can be easily decomposed in that way. However, graph-based representations such as LLVM IR may require a preprocessing step such as structural analysis in order to understand the program in the structural way. For brevity, the explanation in this chapter uses an AST-based representation and its terminology.

The grammar of the language used to describe the code generation technique is shown in Figure 4.1. The language is a subset of OpenCL language, because dealing with the complete OpenCL language requires substantially more details which are not essential to deliver the idea presented in this chapter. For instance, the language omits `for`-loop, because its distinction from `while`-loop yields only a minor difference in practice when it comes to applying the technique.

4.1 Subregion Formation

A *subregion* is a list of consecutive statements that do not contain barriers. The execution of a subregion for work-items is meant to be serialized. Subregions are used in order to implement BFO schedule for a region containing a loop. By declaring a loop body as a subregion, BFO schedule for the loop can be implemented. On the other hand, a subregion enclosing a loop entirely implements DFO schedule for the loop.

In order to form a subregion inside a loop, a barrier due to scheduling can be introduced, which is called *scheduling barrier* or *boundary*. A scheduling

$$
\begin{array}{rcl}
\textit{Statements} & \rightarrow & \textit{Statement} \mid \textit{Statement}\ \textit{Statements} \\
\textit{Statement} & \rightarrow & \textit{StructureStmt} \mid \textit{ExprStmt} \\
\textit{StructureStmt} & \rightarrow & \textit{CompoundStmt} \mid \textit{ControlStmt} \\
\textit{CompoundStmt} & \rightarrow & \{\ \textit{Statements}\ \} \\
\textit{ControlStmt} & \rightarrow & \textit{IfStmt} \mid \textit{LoopStmt} \\
\textit{IfStmt} & \rightarrow & \texttt{if}\ (\ \textit{Expr}\ )\ \texttt{then}\ \textit{Statement}\ \texttt{else}\ \textit{Statement} \\
\textit{LoopStmt} & \rightarrow & \texttt{while}\ (\ \textit{Expr}\ )\ \textit{Statement} \\
\textit{ExprStmt} & \rightarrow & \textit{Expr}\ ; \\
\textit{Expr} & \rightarrow & \textit{Expr}\ \textit{binary\_op}\ \textit{Expr} \mid \textit{unary\_op}\ \textit{Expr} \mid \\
 & & \textit{Expr}\ (\ \textit{Expr}\ ) \mid \textit{Expr}\ [\ \textit{Expr}\ ] \mid \\
 & & (\ \textit{Expr}\ ) \mid \text{literal} \mid \text{term} \\
\textit{binary\_op} & \rightarrow & \texttt{=|+|-|*|/|<<|>>|<|>|==|!=|>=|<=|,} \\
\textit{unary\_op} & \rightarrow & \texttt{+|-|!|\~{}}
\end{array}
$$

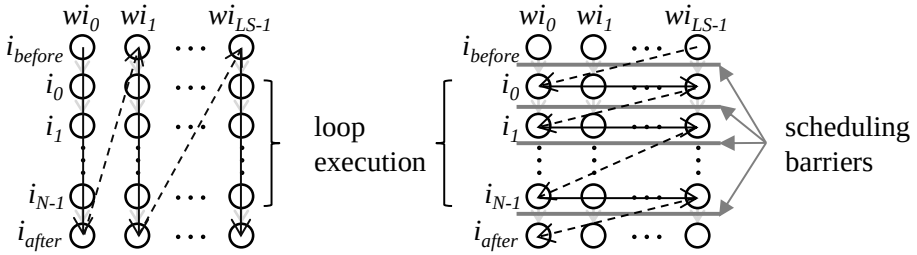
Figure 4.1: The grammar of the language used for code generation.

barrier divides a region into subregions, and similarly a barrier instruction divides a kernel into regions (Chapter 2). Note that unlike barrier instructions, scheduling barriers may be introduced as needed by the compiler for scheduling flexibility, but they do not change the semantic of the region. By adaptively introducing scheduling barriers, the compiler can selectively implement DFO or BFO schedule for a loop. The proposed compiler inserts a scheduling barrier at the beginning of a loop body when the loop prefers BFO. No scheduling barrier is created for DFO.

Figure 4.2 shows how scheduling barriers are used to implement both DFO and BFO schedules. An example region containing a loop is shown in Figure 4.2(a). Assuming that the loop prefers DFO, scheduling barriers are not created and the execution of the entire region is serialized, shown in Figure 4.2(b). With BFO schedule, a scheduling barrier is introduced at the beginning of the loop body of the loop, as shown in Figure 4.2(c). Due to the scheduling barrier, execution of the loop body over work-items gets

```
i_before;
while (k < N) {
  i_k;
  k = k + 1;
}
i_after;
```

(a) An example region.



(b) DFO scheduling.

(c) BFO scheduling with scheduling barriers.

Figure 4.2: An example of scheduling barriers deployed to implement BFO scheduling. Execution of `k = k + 1;` is not shown.

synchronized at the scheduling barrier. In contrast, DFO schedule does not introduce the scheduling barrier at the loop body and thus the execution of the loop for a work-item is not entirely interrupted. As a side effect of the scheduling barrier, the execution for work-items before the loop is synchronized at the loop entry as shown by a scheduling barrier between i_{before} and i_0. Likewise, execution of the statements after the loop can only begin once the entire execution of the loop terminates. As a consequence, the scheduling barrier effectively creates four barriers at these locations: right before the loop, the beginning of the loop body, the end of the loop body, and right after the loop.

A *boundary statement* is a structure statement which includes a subregion. Since the execution of the statement cannot be done as a whole iteratively for work-items, the statement acts as a barrier so that the execution for the enclosed subregion is available. In the example code in Figure 4.2, the `while` statement is the boundary statement.

Figure 4.3 shows an example of code used throughout the code generation. The code has a loop which runs conditionally by a conditional expression. By the language definition, conditional expressions for *IfStmt* and *LoopStmt* do not contain structure statements. For the purpose of explanation, early

```
1  i = get_local_id(0);
2  if (foo()) {
3    bar(i);
4    while(baz()) {
5      qux(i);
6    }
7  }
```

(a) Example region.

```
[S1,S2] (CompoundStmt)
 ├─ S1 (ExprStmt)
 └─ S2 (IfStmt)
     └─ [S3,S4] (CompoundStmt)
         ├─ S3 (ExprStmt)
         └─ S4 (LoopStmt)
             └─ [S5] (CompoundStmt)
                 └─ S5 (ExprStmt)
```

(b) AST structure of the region.

Figure 4.3: A running example for code generation.

exit and return are not allowed in any of `bar()`, `baz()`, `foo()` and `qux()`. The AST representation shows the hierarchy of the program. $\texttt{S}_{num}$ indicates a statement at line *num* in the code. As for *CompoundStmt*, the notation of `[S1,S2,...]` is used to indicate that the compound statement has `S1` and `S2` and others for its children statement. Regions are treated as *CompoundStmt* and thus share the same notation. Each statement has its type, which is shown in parentheses.

Subroutine 2 subRegionFormation(*Region* **as** *CompoundStmt*)

markBoundaries(*Region*)
return createSubRegions(*Region*)

The subregion formation is done in two phases: (1) marking subregion boundaries within a region, and (2) creating subregions between those boundaries. The pseudo-code for the overall process is shown in Subroutine 2. The following subsections detail each phase.

4.1.1 Scheduling Boundary Creation

Subroutine 3 iterates statements in the region and marks scheduling boundaries for each statement. Scheduling boundaries for individual statements are created differently according to the statement type. The *CompoundStmt* marks boundaries for each statement individually in its children statements, and the *CompoundStmt* is marked as containing a scheduling boundary if one of its children statements is marked as a scheduling boundary. A loop is marked as a scheduling boundary if it is selected for BFO scheduling, or if its body contains a scheduling boundary. An *IfStmt* is marked as a scheduling boundary if either its `then`- or `else`-statements contain a scheduling boundary. All other statements are not scheduling boundaries. The existence of scheduling boundaries is recorded in a field called `hasBoundary` for each statement. Schedule preference decision by locality analysis is done for each loop and `isBFOLoop(S)` in Subroutine 3 is available during the execution of the algorithm. That is, a boundary is assumed to exist at the loop body of a loop which prefers BFO scheduling.

Subroutine 3 markBoundaries(*S*)

```
switch typeof S:
  case CompoundStmt:
    S.hasBoundary = false
    for every statement C in S.children:
      markBoundaries(C)
      S.hasBoundary |= C.hasBoundary
  case LoopStmt:
    markBoundaries(S.body)
    S.hasBoundary = hasBFOLoop(S) ∨ S.body.hasBoundary
  case IfStmt:
    markBoundaries(S.then)
    markBoundaries(S.else)
    S.hasBoundary = S.then.hasBoundary ∨S.else.hasBoundary
  default:
    S.hasBoundary = false
```

Figure 4.4(a) shows how subregion boundaries are created for the example region. The procedure begins with calling *markBoundaries* in Subroutine 3 with `[S1,S2]` as the input which is the entire region. A region is treated as *CompoundStmt* as mentioned previously. Thus, it matches with *Compound-*

```
markBoundaries([S1,S2]) {
  markBoundaries(S1) {
  ❶ S1.hasBoundary = False;
  }
❷ [S1,S2].hasBoundary |= S1.hasBoundary;
  markBoundaries(S2) {
    markBoundaries([S3,S4]) {
      markBoundaries(S3) {
      ❸ S3.hasBoundary = False;
      }
     ❹[S3,S4].hasBoundary |= S3.hasBoundary;
      markBoundaries(S4) {
        markBoundaries([S5]) {
          markBoundaries(S5) {
           ❺S5.hasBoundary = False;
          }
         ❻[S5].hasBoundary = False;
        }
      ❼ S4.hasBoundary = [S5].hasBoundary | isBFOLoop(S4);
      }
     ❽[S3,S4].hasBoundary |= S4.hasBoundary;
    }
  ❾ S2.hasBoundary = [S3,S4].hasBoundary;
  }
❿ [S1,S2].hasBoundary |= S2.hasBoundary;
}
```

(a) Execution trace of marking scheduling boundaries for the example region.

```
[S1,S2]
 ├─ S1
 └─ S2
    └─ [S3,S4]
       ├─ S3
       └─ S4
          └─ [S5]
             └─ S5
```

(b) Scheduling boundaries marked on the AST of the region when the loop prefers DFO scheduling(No scheduling boundaries are marked).

```
✓[S1,S2]
  ├─ S1
  └─ ✓S2
     └─✓[S3,S4]
       ├─ S3
       └─ ✓S4 (BFO loop)
          └─ [S5]
             └─ S5
```

(c) Scheduling boundaries marked on the AST of the region when the loop prefers BFO scheduling(Checkmarks (✓) represents scheduling boundaries).

Figure 4.4: Scheduling barrier assignment example.

Stmt and the corresponding case iteratively checks scheduling boundaries with its children statements. Since the type of `S1` does not match with any

of *CompoundStmt*, *IfStmt* and *LoopStmt*, the execution falls into the default case in which the statement is marked as not a boundary(❶). The recursion finishes here and the control returns to where `S1` is tested. The boundary information of `[S1,S2]` is updated but the result remains unchanged(❷). Next, marking boundary for `S2` follows by invoking *markBoundaries* subroutine recursively. The type of `S2` matches with *IfStmt* and its `then`- and `else`-statements are individually tested if boundary exists. Similar to `S1`, `S3` is not declared as a boundary due to its type(❸). Thus, the boundary information of `[S3,S4]` is not changed(❹). The following `S4` is of *LoopStmt* type and the execution deviates to the case with the matching type. The case for *LoopStmt* type first tests if the body of the loop contains scheduling boundaries, which is followed by checking the scheduling preference of the loop by the locality analysis. In this example, the loop body does not contain scheduling boundaries (❺ and ❻). Thus, whether `S4` is a boundary is dependent on the scheduling preference of the loop (❼). When the loop prefers DFO, no boundary is marked and the recursion finishes as there are no scheduling boundaries for the entire region. When BFO is chosen for the loop according to the locality analysis, the loop statement(`S4`) and all of its parent statements(`[S3,S4]`, `S2` and `[S1,S2]`) are marked as boundaries(❽❾❿). Final assignment of subregion boundaries is shown in Figure 4.4(b) and (c) based on the scheduling preference of the loop.

4.1.2 SubRegion Creation Using Boundaries

Subroutine 4 constructs subregions between the boundaries which are marked by Subroutine 3. The procedure checks if the input statement is a boundary. When a boundary is found, the execution deviates based on the type of the statement. Processing for *CompoundStmt* starts with an empty subregion and iterates through every statement, adding it to the subregion until a boundary is reached. Once a boundary is reached, the subregion is added to the subregion list for the input statement. The boundary is handled by processing the child statement recursively. For *IfStmt*, the processing recursively creates subregions for both `then-` and `else-` children. Handling *LoopStmt* is done by creating subregions with its body. The return value is `S` when a boundary exists, which indicates that `S` is a boundary statement

Subroutine 4 createSubRegion(*S*)

```
if S.hasBoundary,
    switch type of S,
        case CompoundStmt:
            SubRegion = []
            for each C in S.children,
                if C.hasBoundary,
                    if SubRegion is not empty,
                        S.SubRegions += <SubRegion>
                        SubRegion = []
                    S.SubRegions += createSubRegion(C)
                else,
                    SubRegion += C
        case IfStmt:
            S. SubRegions += createSubRegion(S.then)
            S. SubRegions += createSubRegion(S.else)
        case LoopStmt:
            S. SubRegions += createSubRegion(S.body)
    return S
else,
    return <S>
```

with a subregion in its children. When the input does not have a boundary, the execution of the input can be done iteratively for work-items. In this case, the return value is `<S>`, where the notation of `<S>` represents that `S` is declared as a subregion.

Figure 4.5 shows how subregions are created using the boundary information for the example region. It shows a trace of running the algorithm for the example region and boundary assignments based on the preferred scheduling of the loop. The trace when the loop prefers DFO scheduling is shown in Figure 4.5(a). In this case, there are no scheduling boundaries in the region. As a result, the entire input region is declared as a subregion.

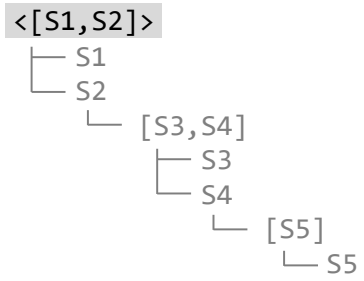
Figure 4.5(b) shows the trace of the subroutine when the loop in the example region prefers BFO scheduling. First, the region has a boundary which leads the execution to the statement-type-specific processing. The region is treated as *CompoundStmt* and it checks if `S1` is the source of the boundary. `SR1` collects `S1` because `S1` is not a boundary. The following `S2` is a boundary, which causes `SR1` or `S1` to be declared as a subregion(❶). The execution con-

```
createSubRegion([S1,S2]) {
  return <[S1,S2]>;
}
```

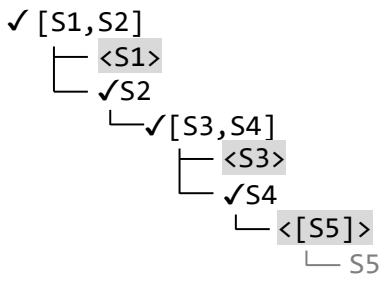
(a) Subregion formation algorithm trace when the loop prefers DFO.

```
createSubRegion([S1,S2]) {    // CompoundStmt
  SR1 += S1;
❶ [S1,S2].SubRegions += <[S1]>;    // SR0
❷ [S1,S2].SubRegions +=
    createSubRegion(S2) {    // IfStmt
    ❸ S2.SubRegions +=
        createSubRegion([S3,S4]) {    // CompoundStmt
          SR2 += S3;
        ❹┌[S3,S4].SubRegions += <[S3]>;    // SR1
         └[S3,S4].SubRegions +=
           ❺createSubRegion(S4) {    // LoopStmt
               S4.SubRegions +=
                 createSubRegion([S5]) {
                  ❻return <[S5]>;
                 }
              ❼return S4;
             }
         ❽return [S3,S4];
        }
    ❾ return S2;
    }
❿ return [S1,S2];
}
```

(b) Subregion formation algorithm trace when the loop prefers BFO.



(c) The result of subregion formation when the loop prefers DFO. Shaded are subregion(s).



(d) The result of subregion formation when the loop prefers BFO(Checkmarks (✓) represent scheduling boundaries). Shaded are subregions.

Figure 4.5: An example demonstrating how subregions are created.

tinues recursively with the processing for `S2`(❷). Since `S2` is a boundary, the execution finds a matching type, which is *IfStmt* in this case. Recursively, subregion creation continues with `then-` part of `S2`, which is of *Compound-Stmt* type(❸). Similarly to the region, `S3` is declared as a subregion because `S3` is not a boundary and but `S4` is(❹). According to the type of `[S3,S4]` which is *CompoundStmt*, the execution recursively progresses with `S4`(❺). `S4` is of *LoopStmt* type and the execution checks the loop body, `[S5]`. Since `[S5]` is not a boundary, a subregion of `<[S5]>` is returned which is added to `SubRegions` list for its parent statement, `S4`(❻). The returned result of processing `S4` is `S4`(❼), which means that `S4` is a boundary statement. The returned result of a statement is added to `SubRegions` list for parent statement of the statement, recursively(❽❾❿). In doing so, `[S3,S4]`, `S2` and `[S1,S2]` are identified as boundary statements. Figure 4.5(c) and (d) illustrate the result of the subregion formation for both DFO and BFO preferred cases, respectively.

4.1.3 Subregion Refinement by Invariance

The idea of subregion refinement is to group statements in a subregion by invariance such that a different work-item loop can be assigned to each group during code generation. The refinement is required because executing an invariant expression multiple times due to work-items may result in an incorrect behavior. Subroutine 5 shows pseudo-code for the refinement for the identified subregion by Subroutine 4. Although not shown in Subroutine 4, *refineSubRegion* can be easily integrated into the pseudo-code.

Subroutine 5 refineSubRegion(*SR of type <S>*)

```
RefinedSubRegion = []
R = []
for each S in SR,
    if R is empty or invariance(SR) equals to invariance(S),
        R += S
    else,
        RefinedSubRegion += <R>
        R = [S]
if R is not empty,
    RefinedSubRegion += <R>
return RefinedSubRegion
```

```
int tx = get_local_id(0);
int base = 0;
for(k = 0; k < K; ++k) {
  ... = A[k*N + base + tx];
  base += delta;
}
```

subregion

(a) Example BFO loop and the subregion of interest.

```
for(k = 0; k < K; ++k) {
    for wid in LS {
        ... = A[k*N + base + wid];
        base += delta;
    }
}
```

(b) Incorrect code generation for the loop.

```
for(k = 0; k < K; ++k) {
    for wid in LS {
        ... = A[k*N + base + wid];
    }
    base += delta;
}
```

(c) Correct code generation for the loop.

Figure 4.6: Refinement of subregion according to invariance. (a) The body of the loop is identified as a subregion, as the loop prefers BFO scheduling. (b) Without refinement of the subregion, the value of a variable `base` changes as the execution of the subregion progresses. (c) The refined subregion excludes the invariant expression and produces the correct result.

Figure 4.6 shows the effect of the refinement. The identified region in the example code has two statements. In the subregion, the first statement is not invariant but the second statement is with respect to work-item values. The first statement requires to execute the subregion for all work-items because the behavior of the statement is unique to each statement. The requirement is fulfilled by putting a work-item loop over the region, as shown in Figure 4.6(b). This approach, however, results in an incorrect behavior, because the second statement should be executed for each iteration of the kernel loop, not for work-item loop iteration. With the refinement, shown in Figure 4.6(c), the subregion is divided into two, one for each statement in the subregion. The execution of the first refined subregion is serialized

over work-items as required. The second refined subregion is invariant to work-items and is left as it is, which produces the correct result.

4.2 Code Generation for Convergent Control Flow

The code generator mainly handles two cases: (1) boundary statement, and (2) subregions. The boundary statement is regarded as a boundary and the code generation assumes that there are two boundaries right before and after the statement. Execution of subregions for work-items is meant to be serialized, thus the resulting code is generated by creating loops iterating over work-items around each subregion.

Subroutine 6 presents pseudo-code for code generation after the subregion formation. The subroutine works with a top-down approach in which the execution deviates based on the type of the input. In this section, the control expression for control statements (*IfStmt* and *LoopStmt*) is assumed to be invariant or uniform to work-items, or *convergent control flow*. When the control expression is not invariant to work-items, or equivalently *divergent control flow* exists, the control expression evaluation and a condition to execute the subregions is required, which will be explained in Section 4.3.

StructureStmt assumes a boundary exists at the statement. The assumption enables execution of a subregion that belongs to the structure statement for all work-items, because the execution of work-items is assumed to be done. The assumption is to guarantee that the execution context for all work-items is present before executing the subregion.

A *CompoundStmt* can have a mix of identified subregions and boundary statements in its `SubRegions` list. All elements in `SubRegions` of the statement are handled iteratively in order, as the structure statement assumes linear control flow among its children. As for *IfStmt*, the control expression is first evaluated. Based on the result, the execution of the resulting code can fall into either the `then`- part or `else`- part, or both when the condition expression is not invariant to work-items. Processing *LoopStmt* is similar to *IfStmt* regarding the boundary behavior. The loop structure is created whose execution is determined by the conditional expression. Similarly, convergent control flow is assumed.

When the input is a subregion, code to execute the subregion is generated,

Subroutine 6 genCode(*S*)

```
switch type of S,
    case CompoundStmt:
        for each SR in S.SubRegions,
            genCode(SR)
    case IfStmt:
        print "if (" + S.expr + ")"
        genCode(S.SubRegions[0])
        print "else"
        genCode(S.SubRegions[1])
    case LoopStmt:
        print "while(" + S.expr + ")"
        genCode(S.SubRegions[0])
    default:
        serialize(S)
```

Subroutine 7 serialize(<*SR*>)

```
if SR is non-uniform w.r.t. work-items,
    print "for wid in LS {"
    print code of SR with
          for each Expr E in SR,
              if E is get_local_id(),
                  replace E with wid
              if E is non-uniform w.r.t. work-items,
                  if E is live-in or live-out,
                      replace E with E[wid]
    print "}"
else,
    print code of SR as it is
```

which is shown in Subroutine 7. First, invariance with respect to work-item values is computed for the subregion to determine to wrap the code with work-item loop or not. When a subregion is invariant to work-items, serialization of execution of the subregion over work-items is not required. In the opposite case, a work-item loop is required to serialize execution of the enclosed code for work-items. As previously shown in Figure 4.6, each subregion *may* have a work-item loop. During the code generation, resulting code for the subregion is modified to correctly supply context for different work-items. First, the work-item value is replaced with *wid*, the index of the work-item loop for the region. Second, handling of live variables takes place,

which is explained in Subsection 4.2.1.

4.2.1 Live Variables and Stride-based Optimization

A *live variable* is defined as a variable whose definition and uses span multiple subregions. When a live variable is used but not defined in a region, it is called *live-in* for the region. When a live variable is defined in a region, it is called *live-out* for the region. Because values for a live variable for all work-items must coexist, live variables require scalar expansion by the dimension of work-item space, *LS*.

Stride-based optimization replaces scalar expansion with linear extrapolation for a live variable with a stationary stride. In particular, when values of a live variable across work-items differ by a fixed amount, called *stride*, one can extrapolate all of the values using the stride and the initial value, called *offset*. In this case, scalar expansion is not required, but instead two scalar variables are needed to deliver the stride and the offset. Definition of the variable initializes the stride and the offset. Users of the variable need to add the varying part determined by the work-item index value and the stride.

Figure 4.7 shows how live variables are handled during code generation. There are two live variables, `VarA` and `VarB` from the identified subregions. After scalar expansion, each becomes an array with given size of LS, the work-item dimension, which is shown in Figure 4.7(b). As previously mentioned, the value of `get_local_id(0)` is replaced with `wid` in both subregions. `VarB` can be the target of the stride-based optimization. Stride analysis can be used here again to extract the stride and the offset. Note that the stride value can be any invariant value to the work-item loop, unlike the values permitted for the schedule decision. In this particular case, `VarB` is replaced with two variables for the stride and the offset to deliver the value associated with the variable, `VarB_stride` and `VarB_offset`, respectively. When the live variable is used, the treatment is to add a varying component from the stride multiplied by the work-item loop index to the offset, which is shown in Figure 4.7(c).

The obvious benefit of the stride-based optimization is that the compiler has more information on the value of the variable and advanced optimiza-

```
SubRegion 1 ┌ int tid = get_local_id(0);
            │ VarA = A[tid];
            └ VarB = tid + offset;

SubRegion 2 ┌ ... = VarA;
            └ ... = Array[VarB];
```

(a) Example subregions with two live variables of `VarA` and `VarB`.

```
for wid in LS {
  int tid = wid;
  VarA[wid] = A[tid];
  VarB[wid] = tid + offset;
}

for wid in LS {
  ... = VarA[wid];
  ... = Array[VarB[wid]];
}
```

(b) Code generation with scalar expansion for the two live variables.

```
for wid in LS {
  int tid = wid;
  VarA[wid] = A[tid];
  VarB_stride = 1;
  VarB_offset = offset;
}

for wid in LS {
  ... = VarA[wid];
  ... = Array[VarB_stride * wid +
              VarB_offset];
}
```

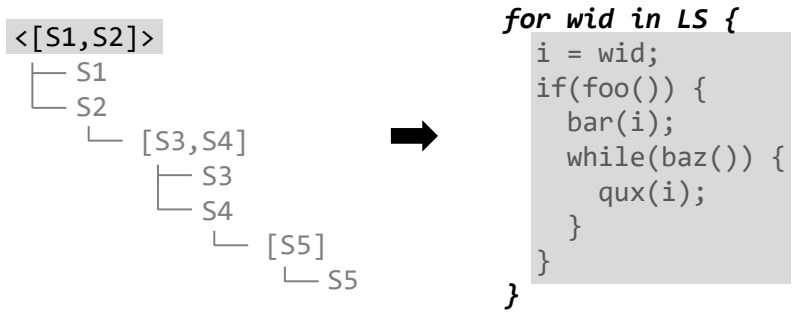
(c) Stride-based optimization applied for `VarB`.

Figure 4.7: An example to demonstrate scalar expansion for live variables and stride-based optimization.

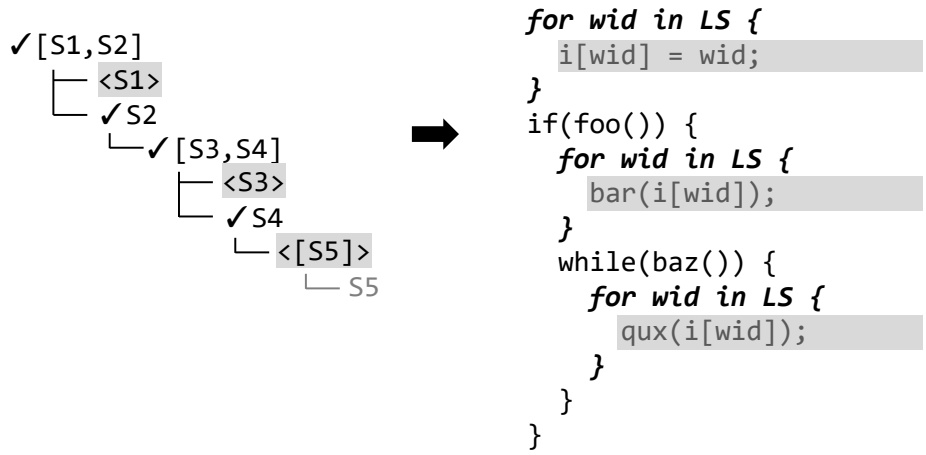
tion can take place based on the information, such as memory operation vectorization when the value of the stride is one. In practice, a moderate fraction of live variables can be factorized in this way, improving memory performance noticeably. Moreover, the stride and offset assignments and associated computation are loop invariant to the work-item loop. Thus, further optimizations can hoist the relevant expressions out of the loop for efficiency.

4.2.2 Code Generation Example

Figure 4.8 illustrates how the code generation is done for the example region. The figure shows the result of subregion formation in two cases depending on whether the loop prefers DFO and BFO. The shaded area in the final output code represents subregions wrapped with work-item loops. Note that stride-based optimization is not applied in this example, with which the definition and uses of `i` can be optimized.



(a) Code generation when the loop prefers DFO scheduling.



(b) Code generation when the loop prefers BFO scheduling.

Figure 4.8: Example of scheduling a non-divergent loop nested in a non-divergent if-statement. Here, `foo()` and `baz()` are assumed work-item independent. Shaded areas represent subregions. Checkmarks (✓) represent boundary statements.

Figure 4.8(a) shows the code generation example when DFO scheduling is preferred for the loop. In this case, execution for work-items of the entire region is serialized. The result of the subregion formation is `<[S1,S2]>`. In

this case, *serialize* shown in Subroutine 7 is called immediately after checking the type of the input upon invoking *genCode* in Subroutine 6. The resulting code thus has a work-item loop over the entire region.

Figure 4.8(b) shows the code generation example when BFO scheduling is chosen for the loop. The subroutine takes `[S1,S2]` as the input, which is a boundary statement. By the type of the statement, *CompoundStmt*, the execution iterates element in `SubRegions` list for the statement. First, `<S1>` is a subregion, thus *serialize* subroutine is called with it. Next, `S2` is identified as a boundary statement. Since the type of the statement is *IfStmt*, the subroutine prints conditional structure with the conditional expression. `Then-` part of the statement is recursively handled with the code generation, which is `[S3,S4]` of *CompoundStmt* type. Upon iterating `SubRegions` for the statement, processing `<S3>` generates code for serialized execution of `S3`. `S4` is a boundary statement of *LoopStmt* type. The loop structure is generated, followed by recursive code generation for its body. The loop body, `S5`, is identified as a subregion. From the resulting code, it is obvious that the way in which work-item loops are formed effectively changes the memory access pattern that may appear in `qux` for DFO and BFO, respectively.

4.2.3 Discussion

One could interpret BFO scheduling as selectively introducing barrier synchronizations inside loops to force work-items to synchronize after every iteration so that they do not get ahead of each other in accessing memory. BFO scheduling is analogous to the dynamic tiling optimization [28] on GPUs where the programmer introduces synchronizations inside loops which are not necessary for correctness but enhance performance by preventing work-items from getting too far ahead of each other, thereby improving temporal and spatial locality.

Another way one could interpret BFO scheduling is taking the traditional DFO-scheduled code and optimizing it with a series of scalar expansions, loop distributions, and loop interchanges. However, there are multiple reasons why it is not always feasible to pass DFO-scheduled code to another compiler for automatic transformation into BFO code. First, a traditional compiler attempting to perform such an optimization would have to first con-

servatively prove that the loops are interchangeable. However, it cannot always be determined that there are no loop-carried dependencies across work-item loop iterations, especially when indirect references obfuscate the loop-dependence analysis. On the other hand, a compiler with direct access to the OpenCL kernel has that guarantee from the programming model, so it can make stronger assumptions without complicated loop-dependence analyses. Second, the presence of control divergence makes a simple loop interchange infeasible and requires much more complex transformations. Dependency between the outer loop (work-item loop) and inner loop (kernel loop) does not exist in the OpenCL program, but the formation of two nested loops inherently brings dependency between the two loops, which hampers the loop interchange feasibility in this case. For these reasons, BFO scheduling can much more effectively be performed when work-item loops are inserted, rather than being outsourced to loop-manipulating optimization passes by an underlying compiler.

4.3 Code Generation for Divergent Control Flow

Control divergence arises when work-items in a work group take different execution paths. In a schedule which only uses DFO, the multiple execution paths for work-items are not an issue. Region boundaries are by definition points of synchronization in the program. Since all work-items must be active at synchronization points, it is safe to assume that work-items are always convergent at the entry and exit points of a region. For this reason, a loop over all work-items can be inserted around the entire region without any concern about some work-items not being active.

On the other hand, not all work-items are guaranteed to be active at the entry and exit points of a subregion because a subregion could be within the body of a divergent conditional or loop. Therefore, wrapping subregions with a work-item loop is not sufficient for BFO scheduling. Instead, control divergence is handled by introducing a predicate array that tracks which work-items are active. Before the subregion is executed for a particular work-item, the predicate array must be checked for whether the work item is active. The combination of the work-item loop with the predicate check is denoted as a *predicated work-item loop*. A predicate array is created for

a divergent control flow and is used for children statements enclosed by a control statement.

In order to support predication, the *genCode* subroutine shown in Subroutine 6 needs to be modified for control statements to generate predicate when the conditional expression is not uniform to work-items. The generated predicate value is propagated down to children statements of the boundary statement. Also, the *serialize* subroutine shown in Subroutine 7 requires changes to use predicate in order to selectively execute a subregion when predicate array is applied.

```
i = get_local_id(0);
 if(foo(i)) {
   bar(i);
   while(baz()) {
     qux(i);
   }
 }
```

(a) An example region.

```
For wid in LS {
  i = wid;
  if(foo(i)) {
    bar(i);
    while(baz()) {
      qux(i);
    }
  }
}
```

(b) DFO scheduling.

```
For wid in LS {
  i[wid] = wid;
  pred[wid] = foo(i[wid]) != 0;
  numActive += pred[wid];
}
if(numActive > 0) {
  for wid in LS {
    if(pred[wid]) {
      bar(i[wid]);
    }
  }
  while(baz()) {
    for wid in LS {
      if(pred[wid]) {
        qux(i[wid]);
      }
    }
  }
}
```

(c) BFO scheduling.

Figure 4.9: Example of scheduling a non-divergent loop in a divergent context using predicated work-item loops.

Control divergence can be introduced whenever there is work-item dependent control flow due to conditionals or loops. Figure 4.9 illustrates the case where a loop is guarded with a divergent conditional. The example region shown in Figure 4.9(a) is the same code used as a running example, shown in Figure 4.3, but the conditional expression, `foo(i)`, is evaluated differently for work-items. Subregions for the example code are the same as shown in Figure 4.5, because control divergence of a boundary statement is not relevant for subregion formation. DFO scheduling is done by simply wrapping

the entire region with a work-item loop as shown in Figure 4.9(b). However, to perform BFO scheduling, the condition evaluation must be stored and used for executing the subregions inside the conditional via predicated work-item loops, as shown in Figure 4.9(c).

```
i = get_local_id(0);
if (foo()) {
  bar(i);
  while(baz(i)) {
    qux(i);
  }
}
```

(a) An example region.

```
For wid in LS {
  i = wid;
  if (foo()) {
    bar(i);
    while(baz(i)) {
      qux(i);
    }
  }
}
```

(b) DFO scheduling.

```
for wid in LS {
  i[wid] = wid;
}
if(foo()) {
  for wid in LS {
    bar(i);
  }
  numActive = 0;
  for wid in LS {
    pred[wid] = baz(i) != 0;
    numActive += pred[wid];
  }
  while(numActive > 0) {
    numActive = 0;
    for wid in LS {
      if (pred[wid]) {
        qux(i);
        pred[wid] = baz(i) != 0;
        numActive += pred[wid];
      }
    }
  }
}
```

(c) BFO scheduling.

Figure 4.10: Example of scheduling a divergent loop.

Figure 4.10(a) illustrates the case where a loop is control flow divergent because the loop condition, `baz(i)`, is dependent on the work-item id. Again, this code is similar to the running example shown in Figure 4.3, sharing the same subregion formation result, but the loop condition expression is evaluated differently for work-items. The DFO code still follows the same strategy as shown in Figure 4.10(b). The BFO code is shown in Figure 4.10(c). In addition to storing a predicate array and using predicated work-item loops to wrap subregions, the total number of active work-items is also maintained at all iterations to know when the loop must terminate.

4.4 Vectorization

```
for wid in LS {
  if(pred[wid]) {
    ...
  }
}
```

(a) Predicated work-item loop.

```
if(numActive == LS) {
  strip-mined work-item loop
} else {
  predicated work-item loop
}
```

(b) Vectorization of predicated work-item loop.

Figure 4.11: Vectorization based on runtime convergence checking.

BFO scheduling enables vectorization opportunity via strip-mining of the work-item loops. Work-item loops provide an ideal condition for vectorization, as they are canonical loops by definition; at the same time they do not have any loop-carried dependence. Canonical loops use a single primary loop induction variable whose value is incremented by one at the end of the loop. Work-item loops are meant to serialize originally parallel execution, hence no loop-carried dependence. However, predicated work-item loops are difficult to vectorize because of the loop-dependent conditional surrounding the body of the loop. For this reason, the prototype tool statically generates two versions of the code and selects between them dynamically based on a runtime divergence check. The first version uses a regular strip-mined work-item loop that is selected when all work-items in the work-group are active. The second version is a serial predicated work-item loop that the execution falls back on when not all work-items are active. The resulting code is shown in Figure 4.11. A similar technique was employed in [34] and [38]. In some cases, vectorization can be improved using control-flow to data-flow conversion techniques such as those employed in [5] and [39].

CHAPTER 5

EVALUATION OF PROPOSED SCHEDULING

The performance of the proposed compiler with locality-centric scheduling is evaluated in this chapter. I demonstrate that locality-centric scheduling is able to consistently select the schedule having fewer data cache misses. Next, comparison of the prototype implementation with other OpenCL implementations from the industry is presented to demonstrate the overall performance of the technique. The result shows that the proposed implementation achieves substantial improvements on both memory hierarchy efficiency and performance. Compared to the two state-of-the-art industry OpenCL stacks from AMD and Intel, the prototype implementation achieves reduced number of L1 data cache misses by 9.81× and 3.35×, and speedup of 3.32× and 1.71×, respectively.

5.1 Experimental Setup

The proposed compilation approach is implemented as an extension of the Clang compiler framework. An AST-level source-to-source translator takes OpenCL code and emits C code. Vectorization is performed by annotating work-item loops without enclosed structured control flow with `#pragma simd` pragmas. Note that the pragma is not a suggestion but a command for vectorization. The pragma therefore must be carefully used, and thus the pragma is applied to work-item loops without control flow inside to avoid unwanted side effect from vectorizing code with control flow. The final machine binary is assembled using the Intel C Compiler (ICC), version 14.0.1. The same compiler is used for building all benchmarks. For work-group distribution, Intel's TBB [26] is used to exploit work stealing for efficient load balancing.

The evaluation platform consists of an Intel i7-3820 processor running at

3.6GHz, having 4 cores with hyperthreading enabled. With the vectorization turned on, AVX instruction which is 256-bit wide can be used and executed on the CPU. The memory hierarchy includes 32KiB L1 private data caches, 10MiB shared last-level cache, and 16GiB of DDR3 DRAM with dual channel configuration. The system is running 64-bit Debian Jessie distribution. A PMU-based performance monitoring library, perfmon2 [40], is used for collecting performance counters throughout.

The industry implementations we compare against are AMD's [13] and Intel's [18] OpenCL compilers. The driver versions used are 1445.5 and 1.2.0.8, respectively.

Throughout the experiments section, data are normalized against the approach scoring highest for the metric under study as opposed to a common baseline. The reason for doing so is that if a single baseline is taken, the values for locality and speedup could span three to four orders of magnitude ($0.01\times$ to $10\times$) which is difficult to plot on a single axis. This normalization methodology makes the graph more readable and makes better use of the space than log plots.

5.2 Benchmarks

Eighteen benchmarks from the Parboil [41] and Rodinia [35] benchmark suites were selected for evaluation. The benchmarks selected are those having loops which are completely contained within a code region such that the proposed technique is applicable. The remaining benchmarks are not relevant because they either do not contain loops within regions, or the loops have short constant trip counts such that they disappear after unrolling.

Table 5.1 lists the benchmarks evaluated and the abbreviations used throughout this chapter for each. Each benchmark is executed ten times for evaluation of the average execution time and associated performance counters. Three benchmarks (`hst`, `lkct`, and `mrig`) have device functions in the dominant loops. These functions are manually inlined to focus the comparison with AMD and Intel on locality, since their compilers seem to inline device functions while our framework does not currently support that.

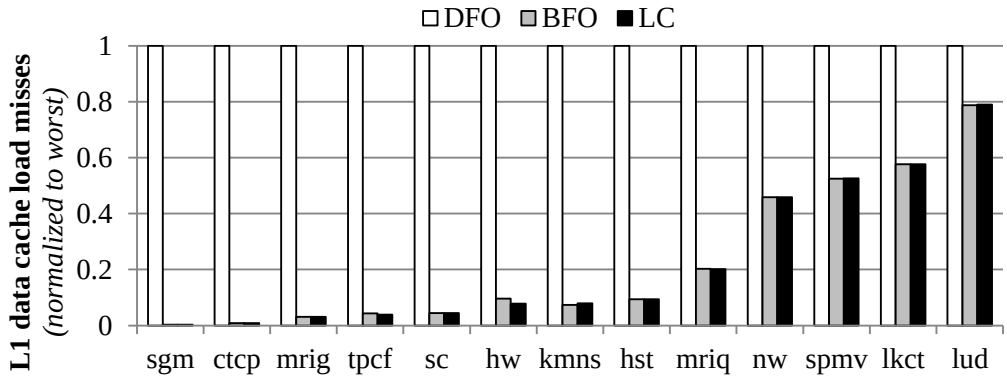
Table 5.1: Evaluated benchmarks from Parboil and Rodinia benchmark suites with abbreviations used.

Benchmark	Abbreviation	Description
cutcp	ctcp	Computing short-range electrostatic potentials
heartwall	hw	Movement tracking of a mouse heart over a sequence ultrasound images
histo	hst	Histogram
kmeans	kmns	Clustering algorithm used extensively in data-mining
lavaMD	lmd	Particle potential and relocation due to mutual forces between particles within a large 3D space
leukocyte	lkct	Rolling leukocytes tracking in vivo video microscopy
lud	lud	LU decomposition
mri-gridding	mrig	A non-uniform input data in 3-D space mapping onto a regular 3-D grid of the same space
mri-q	mriq	3D MRI reconstruction algorithm in non-Cartesian space.
nw	nw	Needleman-Wunsch algorithm
parboil's bfs	pbfs	Queue-based breadth first search
particlefilter	pf	Statistical estimator of the location of a target object given noisy measurements of that target's location and an idea of the object's path
rodinia's bfs	rbfs	Read-based breadth first search
sad	sad	Sum of absolute difference
sgemm	sgm	Generalized matrix-matrix multiplication in single precision
spmv	spmv	Sparse matrix-vector multiplication
streamcluster	sc	Finding predetermined number of medians so that each point is assigned to its nearest center
tpacf	tpcf	Two-point angular correlation function

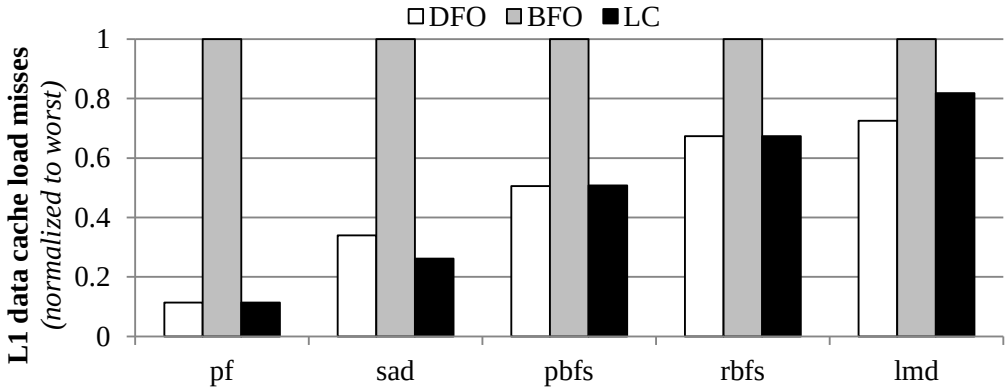
5.3 Impact of Scheduling on Locality

Figure 5.1 compares the number of L1 data cache misses (lower is better) of DFO, BFO, and LC scheduling. The values for each benchmark are normalized to the policy having the highest (worst) number of misses. The benchmarks are categorized according to the schedule (DFO or BFO) having better performance and sorted in decreasing order of LC's relative performance.

The graph shows that 13 benchmarks have better locality with BFO scheduling and 5 are better with DFO. Moreover, it shows that LC scheduling consistently selects the correct schedule, achieving geomean reductions in L1 data cache misses of 5.72× and 1.29× over DFO and BFO respectively. Because different loops can receive different schedules, the results of adaptive scheduling for `sad` and `hw` are better than a fixed scheduling of DFO or BFO, because the benchmarks have multiple loops with different scheduling preference. Nested loops can also benefit from the individual scheduling, as depicted by `tpcf` which has a BFO-preferred loop at the outermost for doubly-nested loops. Thus, neither DFO nor BFO can exploit the best data locality for `sad`, `hw` and `tpcf`. In case of `lmd`, two loops in the kernel are ana-



(a) BFO preferred benchmarks.



(b) DFO preferred benchmarks.

Figure 5.1: Locality comparison of DFO, BFO, and locality-centric (LC) scheduling. Results are normalized to the worst performing schedule. LC has geomean reduction in L1 data cache misses of 5.72× and 1.29× over DFO and BFO respectively.

lyzed as BFO-preferred but the decision is worse than DFO schedule, because the BFO schedule necessitates scalar expansion of associated variables along with predicated work-item loop due to a control divergence, while the loop trip count is not large enough to amortize the cost. Since the two loops are not the most significant in execution time and data locality, the end result is worse than DFO schedule, which is optimal but better than BFO schedule.

Table 5.2 shows the schedule decision result for the benchmarks. Interestingly, most of the benchmarks have skewed statistics toward one or the other schedule. An exception is `lmd`, which is equal for both schedules and DFO is chosen as a tie-breaker (Chapter 4). It also shows that most loops have only a few memory operations, where the median is 4.

Table 5.2: Relevant benchmarks from Parboil and Rodinia benchmark suites with classification of the memory accesses in the most significant loop.

Name	Neutral				DFO				BFO			
	W0L0	W1L1	WXLX	Total	W1L0	WXL0	WXL1	Total	W0L1	W0LX	W1LX	Total
cfd	15		5	**20**							4	**4**
ctcp									4			**4**
fft			2	**2**					6			**6**
hw											2	**2**
hst			4	**4**								
kmns									2		2	**4**
lmd					11			**11**	11			**11**
lkct			2	**2**								
lud									1		1	**2**
mrig									9			**9**
mriq									5			**5**
nw											2	**2**
pbfs			2	**2**			1	**1**				
pf					2		3	**5**		2		**2**
rbfs			3	**3**	4		1	**5**				
sad			1	**1**			1	**1**				
sgm										1	1	**2**
spmv			1	**1**					1		2	**3**
sc									2		2	**4**
tpcf			3	**3**						3		**3**

The conclusions drawn from this experiment are:

- Current state-of-the-art work-item scheduling techniques (i.e., DFO) yields suboptimal data locality behavior (Chapter 2).
- A single scheduling technique (whether DFO or BFO) will not always result in the best locality, thereby necessitating that scheduling be locality-aware.
- Our locality-centric scheduling is successful at choosing the schedule, resulting in better locality in most cases.

5.4 Locality Comparison with Industry Implementations

Figure 5.2 compares the number of L1 data cache misses (lower is better) of AMD, Intel, and LC. The values for each benchmark are normalized to the approach having the highest (worst) number of misses. The benchmarks are sorted in increasing order of LC's relative performance.

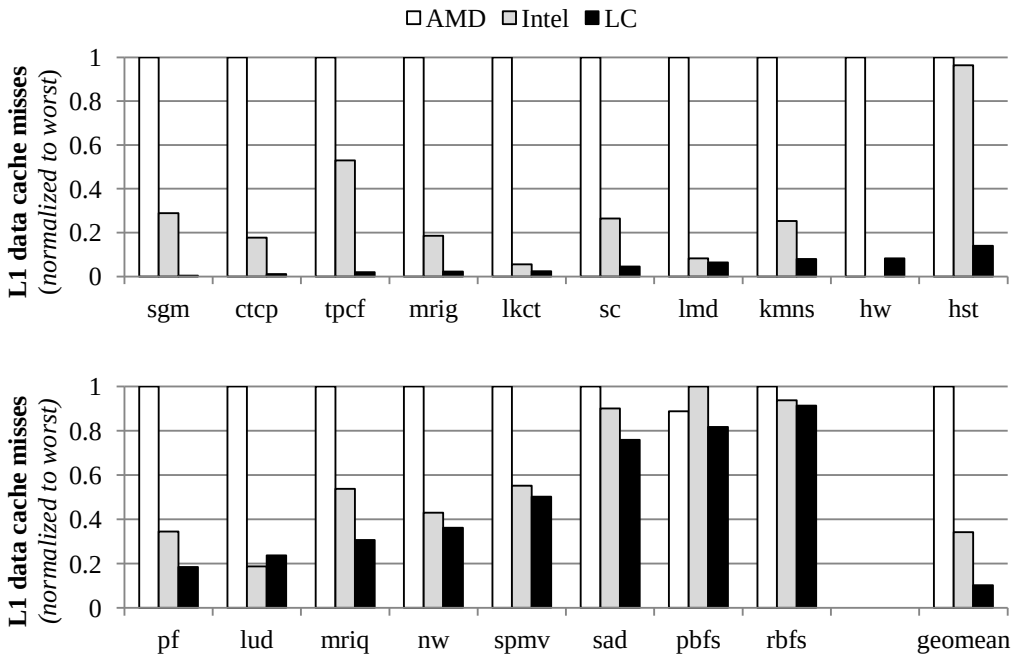


Figure 5.2: Locality comparison of AMD, Intel, and LC compilation approaches. Results are normalized to the worst performing tool. LC has geomean reduction in L1 data cache misses of 9.81× and 3.35× over AMD and Intel respectively.

The graph shows that our locality-centric scheduling achieves locality results which are consistently better than or as good as that of AMD's and Intel's implementations. The missing `hw` datapoint for Intel is because Intel's compiler crashed when compiling this benchmark. LC scheduling was able to achieve geomean reductions in L1 data cache misses of 9.81× over AMD and 3.35× over Intel.

The conclusions drawn from this experiment are:

- Industry implementations of current state-of-the-art work-item scheduling yield suboptimal data locality behavior.
- Our locality-centric scheduling achieves better data locality on average than current industry implementations.

On a side note, we observe that AMD's locality results are significantly worse than Intel's and LC's, even for cases where DFO is better for locality, which is demonstrated with the result of `pf` as a clear example. The poor data locality of AMD is due to the overhead of replicating variables for all work-items regardless, even when variables are uniform. The result of `pf`

reveals a fundamental limitation in AMD's user-level threads technique in working set management and data locality [13].

5.5 Performance Comparison with Industry Implementations

Figure 5.3 compares the relative performance (inverse of time, higher is better) of AMD, Intel, and LC. Since AMD does not seem to vectorize across work-items while Intel does, the result also includes performance results for a vectorized and non-vectorized version of LC to isolate the impact of locality for fair comparison with both. The values for each benchmark are normalized to the best performing tool. The benchmarks are sorted in alphabetical order.

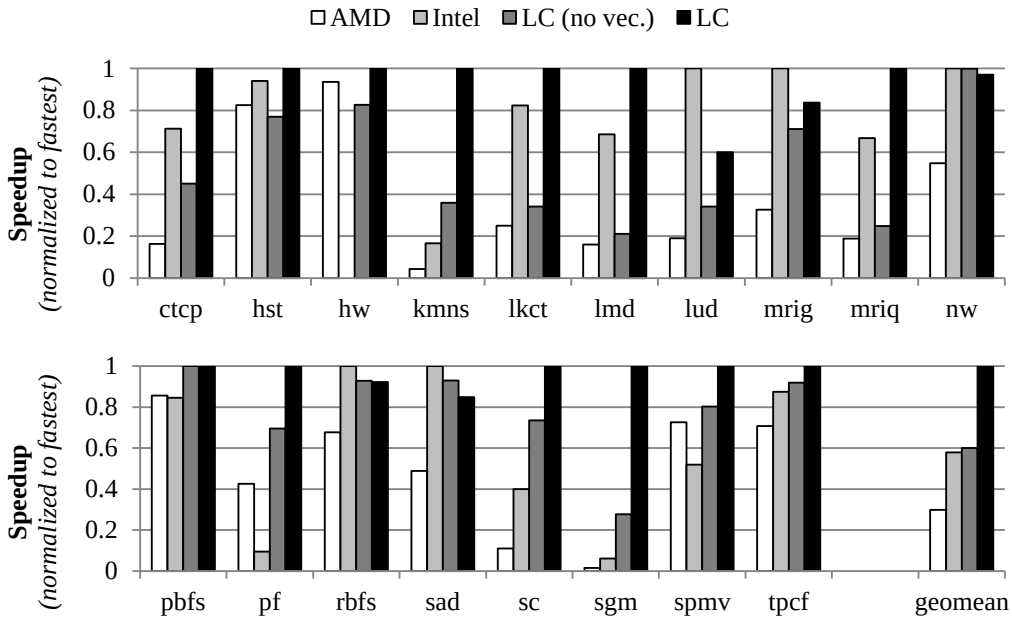


Figure 5.3: Performance comparison with AMD and Intel. Results are normalized to the faster tool. LC achieves a geomean speedup of 3.32× and 1.71× over AMD and Intel, respectively.

The graph shows that the proposed OpenCL implementation for CPUs with locality-centric scheduling achieves significant speedups over AMD and Intel. LC outperforms AMD in most benchmarks and achieves geomean

speedups of 2.01× and 3.32× over AMD for the non-vectorized and vectorized versions respectively. In comparison to Intel, one factor that impacts the performance comparison is that LC and Intel adopt different vectorization strategies [34]. Intel's OpenCL vectorization is left turned on, which is the default behavior. However, LC with vectorization turned off is still able to match Intel's implementation with vectorization turned on, achieving a geomean speedup of 1.04×. This reflects the importance of locality-centric scheduling in enhancing performance. LC achieves a geomean speedup of 1.71× over Intel with vectorization.

The LC performance of `lud`, `sad` and `mriq` is worse than Intel though their locality behavior is superior with LC. In particular, LC cannot determine a better schedule for `lud` as the numbers of BFO and DFO preferred memory operations are the same. Moreover, the loop trip counts for both the kernel loop and work-item loop are small, 16 for both. In this situation, LC chooses BFO but its overhead stands out, without definitive benefit from locality. As for `sad` and `mriq`, the disparity is due to the difference in code generation, particularly dealing with control divergence when BFO-preferred loops exist. Intel's approach would produce machine code for predication [34], while the emitted C code from LC cannot exploit the rich feature of the native instruction set. When the inefficient code is matched with relatively low loop trip count, the cost to implement the BFO schedule is exposed as overhead.

The conclusions drawn from this experiment are:

- Our OpenCL implementation with locality-centric scheduling meets industry performance standards and outperforms state-of-the-art industry implementations in most cases.

Table 5.3 summarizes the comparison between our approach and the other industry implementations for L1 data cache misses, speedup, and other metrics. The comparison shows that the locality-centric schedule brings positive impact throughout the memory system.

Table 5.4 compares the ratio of overall speedup and number of instruction counts for both non-vectorized and vectorized code for work-item execution. AMD uses a user-level thread for each work-item, which serializes the execution work-item. LC without vectorized code emits scalar operations only. Thus, the two mainly differ in the work-item loop arrangement with slight advantage to LC due to less overhead of user-level thread management. A

Table 5.3: Geomeans summarizing the comparison of locality-centric scheduling with industry implementations.

Metric	LC/AMD	LC/Intel
Speedup	3.32x	1.71x
L1 Data Cache Misses	0.10x	0.30x
Data TLB Misses	0.26x	0.33x
LLC Misses	0.92x	0.77x

Table 5.4: Ratio of speedup to instruction counts.

Metric	LC(no vec)/AMD (not vectorized)	LC/Intel (vectorized)
Speedup	2.01	1.71
Instruction counts	0.80	0.98

similar comparison can be made for the vectorized implementation pair between Intel and LC with vectorization turned on. In both cases, LC executes a comparable number of instructions, but the speedups are far more than the sole benefit from the reduced instruction counts. For that, data locality must have played a critical role for the performance.

CHAPTER 6

EVALUATION WITH BLAS KERNELS

This chapter evaluates the performance of the proposed compiler in the context of performance portability. We pick three well-understood kernels from BLAS, one for each level. They are `saxpy`, `sgemv` and `sgemm`. The selected kernels are written in OpenCL and their performance using the prototype OpenCL compiler is compared to the highly optimized counterpart for CPU, Intel MKL [42]. The goal of this experiment is to more rigorously evaluate the approach. A better judgment on portable performance can be made by comparing against well known algorithms with highly hand-optimized implementations, instead of a comparison to the performance of arbitrarily written code for indigenous algorithms.

The proposed approach achieves 98.3%, 85.2% and 76.1% of Intel MKL performance for `saxpy`, `sgemv` and `sgemm`, respectively, when combined with tiling and work-group resizing. On top of locality-centric scheduling, resource management is identified as an important optimization, which can be improved via tiling of loops and work-group size adjustment. I argue and show that the locality-centric scheduling lays a foundation to implement the resource management optimizations, which enables portable performance. The rest of this chapter shows performance evaluation and analyzes how the aforementioned techniques achieve the performance.

6.1 BLAS-1: SAXPY

Figure 6.1 shows `saxpy` code in OpenCL for the experiment. The code for each work-item consumes two elements to produce one output. It has streamlined control flow and the vectorized work-item loop would yield good performance for CPU. The work-group size is set to 512, which is borrowed from the GPU programming which achieves 100% occupancy.

```
__kernel void saxpy(
  __global float *y, const __global float *x, float a) {
    int i = get_global_id(0);
    y[i] = a*x[i] + y[i];
}
```

Figure 6.1: OpenCL implementation for `saxpy`.

The performance result is drawn in Figure 6.2 which compares the performance over exponentially increasing workload sizes. Both LC and Intel's OpenCL implementations show good and stable performance compared to MKL for all inputs. This is ultimately memory bandwidth bounded and there is not much variation in code generation. As such, all three perform similarly with a sizable input. The performance gap between OpenCL and Intel MKL at smaller input sizes is due to instruction overhead in order to launch the kernel, which is amortized with larger inputs.

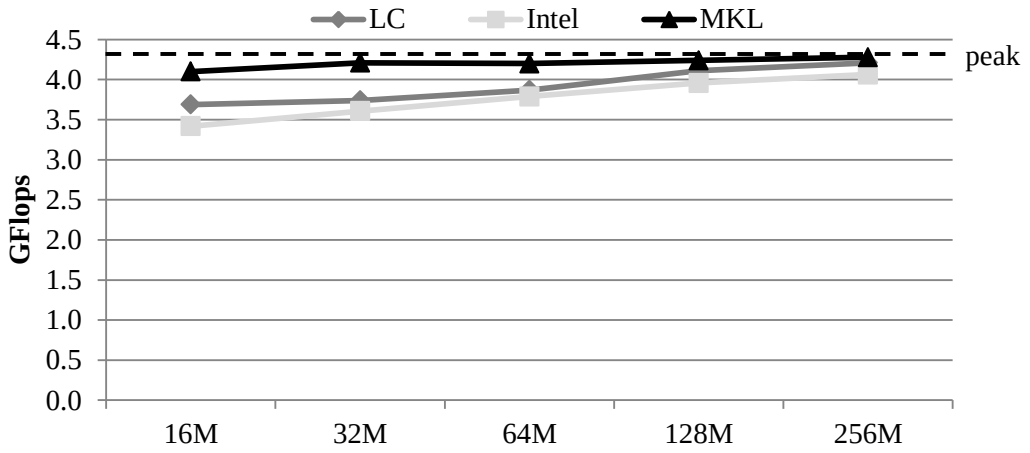


Figure 6.2: Performance of `saxpy` using the proposed method(LC), Intel's OpenCL stack and Intel MKL implementations.

6.2 BLAS-2: SGEMV

Code listing for `sgemv` is shown in Figure 6.3. The implementation has one loop for dot product and the result is stored back to its own location per work-item. The implementation assumes column-major order data layout. The work-group size is set to 512 for this experiment. The proposed compiler selects BFO for the kernel loop.

```
__kernel void sgemv(__global float * y,
                    const __global float * A,
                    const __global float * x,
                    float alpha, float beta,
                    int nRows, int nCols) {
  int r = get_global_id(0);
  if (r < nRows) {
    float result = 0.f;
    for (int c = 0; c < nCols; c++) {
      result += A[nRows*c+r]*x[c];
    }
    y[r] = alpha*result + beta*y[r];
  }
}
```

Figure 6.3: OpenCL implementation for `sgemv`.

The performance result is presented in Figure 6.4. Increasing sizes of square matrices are used which are shown on the x-axis of the figure. The performance trend of Intel MKL demonstrates stable results across the board, while the two OpenCL stacks tend to perform better when the input size is small, due to the fact that a large fraction of input is cached. When input size gets large, the proposed approach of LC shows 85% of Intel MKL performance. The Intel OpenCL stack only obtains 25% of Intel MKL performance when a large input is given. The higher performance of the proposed approach is due to the BFO schedule of the loop, which is preferred scheduling for data locality. The result reaffirms the importance of locality-centric scheduling.

6.2.1 Work-group Size Adjustment

In this subsection, the impact of work-group size adjustment is detailed using `sgemv`. BFO schedule with muldimensional data can benefit from a large work-group size due to reduced number of data TLB miss counts.

Figure 6.5 illustrates the work-group size and its impact on data TLB miss counts in the `sgemv` kernel. The figure shows a mapping of work-groups to process the input matrix for `sgemv`. In the figure, it is assumed that there are four CPU cores and the tall input matrix can be processed with four work-groups, shown in Figure 6.5(a). The dot product loop in the kernel of

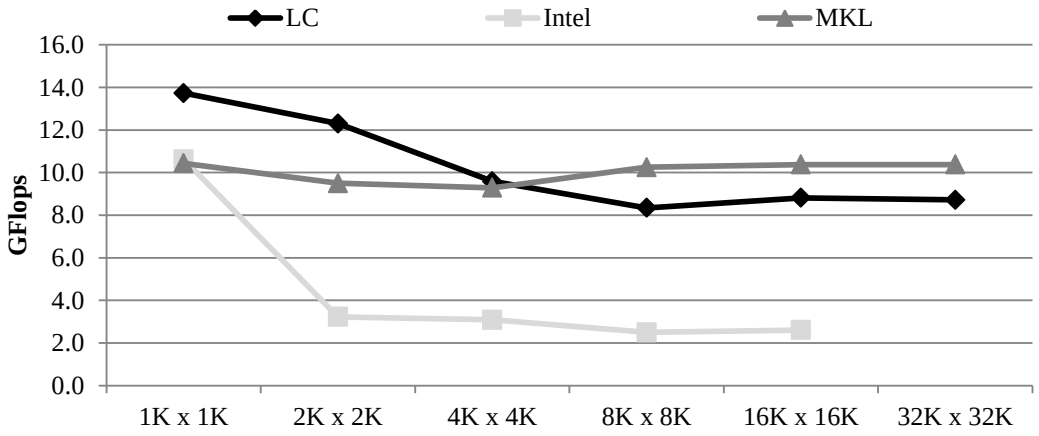
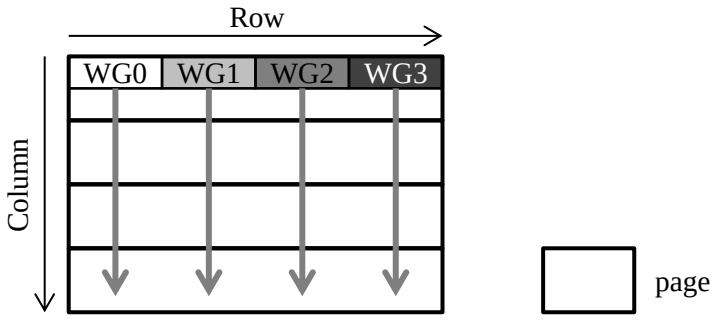


Figure 6.4: Performance of `sgemv` using the proposed method(LC), Intel's OpenCL stack and Intel MKL implementations.

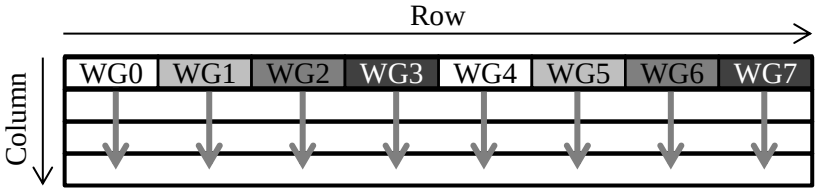
`sgemv` iterates through columns in a row, and due to BFO schedule all work-items make similar traversal. Provided that the progress of each work-group on each core is similar, the overall number of different data pages accessed is four. With a fat matrix as an input, shown in Figure 6.5(b), eight work-groups are required to process the input. Since there are four CPU cores, the processing takes two rounds of four work-groups at a time. With this work-group assignment, the number of different data pages accessed is double that of the case for the tall matrix. As a consequence, the fat and wide input occurs for as many as twice the data TLB miss counts when the input is large, yielding lower performance although the amount of work is comparable for both. The data TLB miss rate changes exponentially from 0.01% for the smallest input to 4.32% for the largest input when the work-group size is 256. Other work-group sizes show similar trends.

Figure 6.6 shows the performance of the `sgemv` kernel for varying shapes of input. The number of elements in the matrix is fixed with 256 Mi elements, but the shape of the input changes from skinny tall matrix of 1 Ki $\times$ 256 Ki to fat short matrix of 256 Ki $\times$ 1 Ki. The work-group sizes tested are 128, 256, 512 and 1024. Intel OpenCL stack results are only shown with 1024 work-items because others behave similarly and are omitted for brevity.

The performance of LC is high when the ratio of the number of work-groups to the number of cores is close to one, which matches or outperforms the performance of Intel MKL. In this particular example, the ratio is determined by `nCols` divided by the work-group size which is further divided by the



(a) A tall matrix being processed with four work-groups mapped on four cores.



(b) A fat matrix being processed with eight work-groups mapped on four cores.

Figure 6.5: The impact of varying shape and work-group size to data TLB miss counts. The figure assumes that there are four physical CPU cores available. The tall matrix in (a) can be processed with four work-groups, while the fat matrix in (b) requires eight work-groups for the same work-group size. The number of data TLB miss counts is double with the latter case.

number of cores. For instance, an input matrix of 1K columns has the ratio of one when the work-group size is 256 with four cores. The high performance achieved when the ratio is close to one can be explained by the low data TLB miss counts as previously discussed. The performance gets saturated when large input is used due to data TLB misses. The larger work-group size entails fewer data TLB misses, yielding higher performance. When the ratio is too small, CPU cores are underutilized, thus performance suffers.

6.3 BLAS-3: SGEMM

In this section, `sgemm` performance is measured and analyzed. Figure 6.7 lists the OpenCL code used in this experiment. The kernel loop prefers BFO

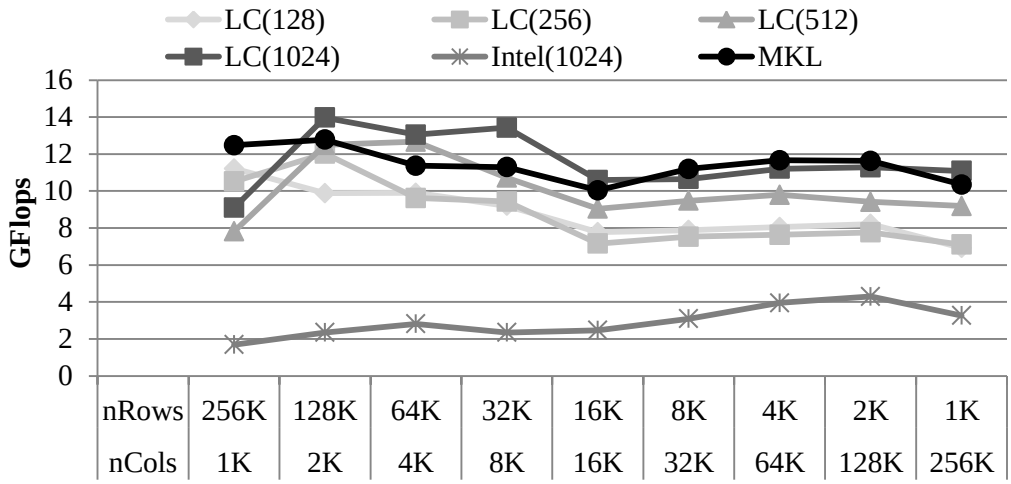


Figure 6.6: Performance of `sgemv` using the proposed method (LC), Intel's OpenCL stack and Intel MKL implementations with varying shapes of input. Different work-group sizes are used for OpenCL stacks.

schedule for both dimensions of x and y, which is explained later in this section.

```
__kernel void sgemmNT(__global const float *A, int lda,
                      __global const float *B, int ldb,
                      __global float* C, int ldc,
                      int k, float alpha, float beta ) {
  float c = 0.0f;
  int m = get_global_id(0);
  int n = get_global_id(1);
  for (int i = 0; i < k; ++i) {
    float a = A[m + i * lda];
    float b = B[n + i * ldb];
    c += a * b;
  }
  C[m+ldc*n] = C[m+ldc*n] * beta + alpha * c;
}
```

Figure 6.7: OpenCL implementation for `sgemm`.

Figure 6.8 illustrates the performance result for `sgemm`. The work-group size is 32x32. The performance trend of MKL reaches up to 70% of the peak throughput. LC steadily achieves about 44 GFlops, or equivalently 30% of MKL performance. Intel's OpenCL stack shows decreasing performance from 7% down to 0.8% of MKL performance as input size grows. Again, its data-locality-oblivious schedule hampers performance, which worses gradually as

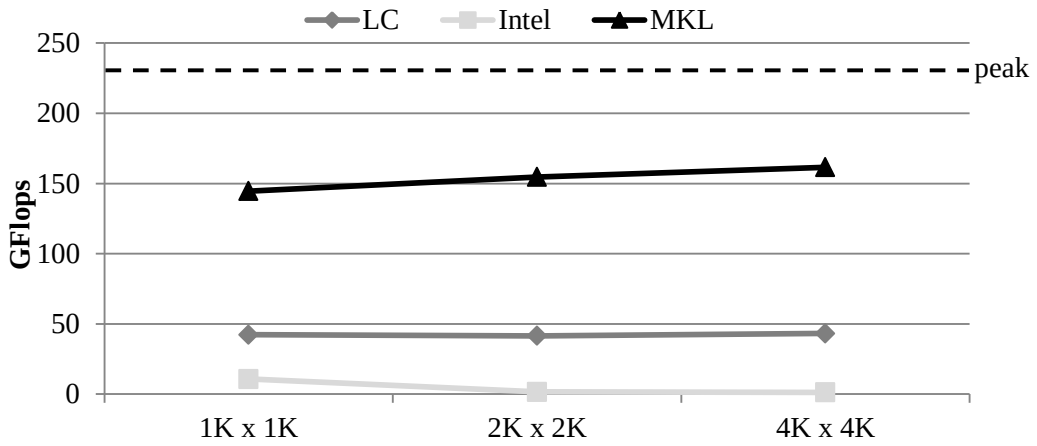


Figure 6.8: Performance of `sgemm` using the proposed method, Intel's OpenCL and Intel MKL implementations.

it deals with a larger working set. Though the proposed compiler yields much higher performance compared to Intel's OpenCL stack up to 37.5× speedup, its relative performance to carefully crafted code is yet to be analyzed and improved.

6.3.1 Tiling and Work-group Size Optimizations

BLAS is core computation for many mathematical libraries and its performance is continuously monitored and upgraded due to its importance. Products like Intel's MKL and open source efforts such as OpenBLAS [43] therefore incorporate several techniques and strategies in pursuit of ultimate performance. While full details of such an implementation are beyond the scope of this dissertation, a simplified program structure can be used to analyze missing pieces toward desirable performance.

Figure 6.9 shows how subregions are formed for the loop in `sgemm`. The kernel loop uses x and y dimensions, which are represented as 0 and 1 in calling `get_global_id` index function. The loop prefers BFO schedule for both of the dimensions, as shown in Figure 6.9(a). Since each expression in the subregion has a distinct schedule, presented in Figure 6.9(b), *refineSubRegion* in the code generation algorithm shown in Chapter 4 will decompose the subregion of the loop body into three subregions. Figure 6.9(c) shows how the subregion of the loop body is divided into multiple refined subregions by invariance of expression statements in the subregion. The three subregions

Expression Statement	Memory access pattern classification	
	x-dimension	y-dimension
`float a = A[m + i*lda];`	W1LX(BFO)	W0LX(BFO)
`float b = B[n + i*ldb];`	W0LX(BFO)	W1LX(BFO)

(a) Memory access pattern classification and preferred schedule.

Expression Statement	invariant to x-dimension?	invariant to y-dimension?	Work-item loop dimension
`float a = A[m + i*lda];`	no	yes	x
`float b = B[n + i*ldb];`	yes	no	y
`c += a * b;`	no	no	xy

(b) Invariance analysis result for the subregion.

```
float a[LS_x];
float b[LS_y];

// subregion 1 - load(A, LS_x)
for wid_x in LS_x,
  a[wid_x] = A[wid_x + offset_wid_x + i*lda];

// subregion 2 - load(B, LS_y)
for wid_y in LS_y,
  b[wid_y] = B[wid_y + offset_wid_y + i*ldb];

// subregion 3 - compute(LS_x, LS_y)
for wid_y in LS_y,
  for wid_x in LS_x,
    c[wid_y][wid_x] += a[wid_x] * b[wid_y];
```

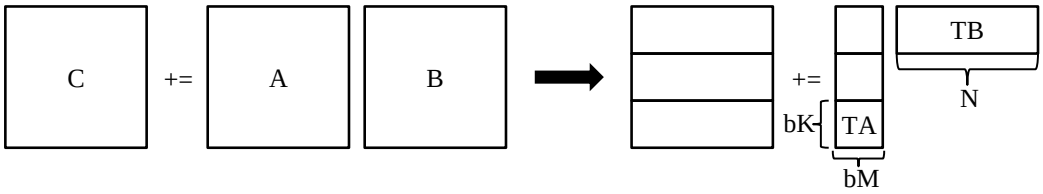
(c) Subregion refinement and code generation for the resulting subregions.

Figure 6.9: Subregion formation for the kernel loop in `sgemm` after refinement of the original subregion. After the refinement, `a` and `b` become live variables and thus scalar expanded with their corresponding dimension size. The code generation also reflects the invariance so that it only subscribes relevant dimensions in generating work-item loops.

can be represented as `load(A)`, `load(B)` and `compute`, respectively, as shown in the comments.

Figure 6.10 illustrates a simplified program structure of high-performance `sgemm` [44] and the generated code by the proposed approach. It highlights the loop structure and tiling strategy, which are core structures for working set control. Subroutine `load` performs loading a tile of a designated size from a matrix and `compute` performs series of multiplication and addition, as similarly annotated in Figure 6.9. The argument specifies dimension of

tile or amount of computation load.



(a) Decomposition of `sgemm` in a high-performance implementation.

```
parallel for each chunk of bM*K in A,
    for (ls = 0; ls < K; ls += bK) {
        load(B, N, bK);                  // loads TB
        for (is = 0; is < M; is += bM) {
            load(A, bM, bK);             // loads TA
            compute(bM, bN, bK);
        }
    }
```

(b) Simplified version of high-performance `sgemm` code.

Figure 6.10: Analysis of high-performance `sgemm`.

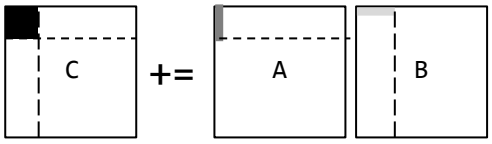
The decomposition shows that the optimized program loads the entire tile TB from matrix B, which is repeatedly multiplied with tiles from matrix A, denoted as TA. The output is accumulated in matrix C. The tile sizes are tuned to utilize cache memory at its best per the actual device the program is running on. On the test machine environment, the tile sizes are 384x768 or 295 KiB and 4096x384 or 1.5 MiB, respectively for TA and TB. As for `compute`, the implementation is hand-optimized to a great degree for instruction throughput, although the detail is not shown. The tile sizes and subroutine for moving data are also aligned with assumptions for `compute` such as alignment requirement in order to guarantee safety of using vector intrinsics.

Comparing the code generated by the prototype compiler to the highly tuned code reveals three suboptimal features. First, the loop arrangement is oblivious to cache memory hierarchy. The size of TB in Figure 6.10 is intentionally chosen to be large so as to exploit L2 and L3 caches in the target machine, which are 256 KiB per core and 10 MiB per chip, respectively. Such consideration is not incorporated with the OpenCL output code. Second, the loop arrangement results in too small working set compared to the high performance version. Input tile sizes for both matrices are LS_x and LS_y,

which is 128 bytes for each, as both variables are set to 16, which are initially borrowed from GPU program. Using the value as it is entails significantly smaller working set than what its counterpart uses. The consequence of the small working set is that the program frequently brings uncached data from memory, and at the same time a large fraction of cache memory is left unused. Third, the code uses a less optimized code sequence. Vectorization in particular is not easily deployed, from having to check legality at compile time and/or runtime such as alignment and recurrence.

```
for (k = 0; k < K; k+=SK) {
    load(A, LS_x, 1);
    load(B, LS_y, 1);
    compute(LS_x, LS_y, 1);
}
```

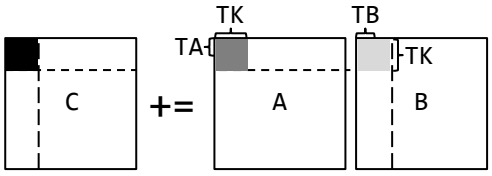
(a) Simplified version of the compiler generated code.



(b) Illustration of the tiling configuration for the compiler generated code.

```
for (k = 0; k < K; k+=TK) {
    for (y = 0; y < LS_y; y+=TY) {
        for (x = 0; x < LS_x; x+=TX) {
            load(A, TX, TK);
            load(B, TY, TK);
            compute(TX, TY, TK);
        }
    }
}
```

(c) Tiled version of the generated code.



(d) Illustration of the tiling configuration of the tiled code.

Figure 6.11: Tiling transformation for the proposed code.

Figure 6.11 compares the LC generated code and a tiled version of the code to reflect the desired change with the analysis. Figure 6.11(a) shows

the simplified result of the generated code, where Figure 6.11(b) illustrates the tiling configuration of the code. In Figure 6.11(c), tiling is applied for both work-item loop and kernel loop to construct efficient instruction sequence and increased data reuse by exploiting larger working set. Similarly, Figure 6.11(d) shows the tiling configuration of the code. Note that variables of a and b in the input code need to be expanded with two dimensions of TK and corresponding work-item dimension. Tiling factor for work-item loops, denoted as TX and TY for x- and y- dimension respectively, can be picked from a few candidates determined by the vector instruction data path width, cache line size and work-group size which is often less than or equal to 512. The difficulty with automating this transformation in the compiler should be moderate, as work-item loop carries good properties for loop transformations (Chapter 4), though selecting a good tiling factor of TX, TY and TK at compile time requires empirical study or a heuristic. Also note that now the loop trip counts for the inner loops are all constant, which provides rich information to the compiler in order to generate an optimized instruction sequence. For the final output generation, the inner loops of the work-item loops are annotated with `#pragma ivdep` to indicate no loop-carried dependency exists.

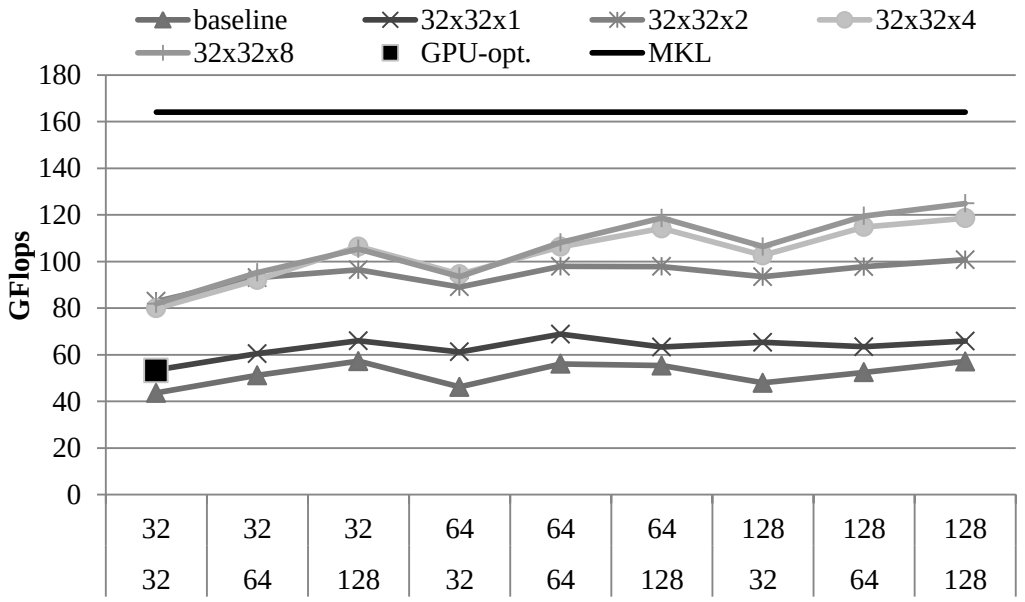


Figure 6.12: Performance of `sgemm` optimized with tiling over varying work-group sizes. Two rows in the x-axis represent y and x dimensions, respectively.

Figure 6.12 shows that enhanced instruction sequence and larger working set can improve the performance. Baseline performance is obtained from the proposed compiler generated output. $TX \times TY \times TK$ in the legend represents blocking factor for work-item loops of x-dimension, y-dimension and kernel loop, respectively. The tiled kernel shows significant speedups across the board, ranging from 1.22× to 2.03× for varying TK values over the baseline. Particularly, the kernel loop tiling not only changes working set footprint, but allows more freedom to schedule closely related instructions for instruction throughput. Tiling factor of 8 or more for the kernel loop demonstrates saturated performance trend, potentially due to register spilling from too many live variables during code generation. Bigger work-group size in general yields greater performance result, which is also likely saturated around 64x128 and higher. Changing work-group size alone does not guarantee the speedup as witnessed from the baseline. The combined compiler optimization and work-group resizing achieve 128 GFlops from the best tiling combination and work-group size, achieving 76% of MKL performance. The compiler solution alone achieves 1.8× speedup, or equivalently 50% of MKL performance.

The GPU-optimized kernel performs 53 GFlops when the work-group size is 32x32. The fixed tiling parameters for the kernel allow little space for optimizations. Though the performance is higher than the baseline, the fixed tiling parameters do not allow room to change work-group sizes in both dimensions, thus only one data point is shown.

6.4 Summary

In summary, the conclusions drawn from this chapter are:

- Processing multidimensional data commonly involves the data locality issue, with respect to which the proposed compiler outperforms the conventional approach.
- Work-group size can have a significant impact on data TLB miss counts and must be incorporated into designing an OpenCL compiler for CPU.
- It is desirable to incorporate tiling optimization into the compiler framework to improve working set management and instruction throughput.

- Combined with work-group size adjustment and tiling optimizations, the proposed approach provides evidence for performance portability using OpenCL programs on CPU, achieving near optimal performance for `saxpy` and up to 85.2% and 76.1% of Intel MKL performance for `sgemv` and `sgemm`, respectively.

CHAPTER 7

RUNTIME-BASED SCHEDULING SELECTION

7.1 Motivation

Compilers for performance-sensitive applications are carefully designed. An optimizing compiler must incorporate a good combination among available analyses and transformations so as to extract the most performance out of the target platform. Typically, transformations often involve a decision making that can have from trivial to substantial consequences for performance. For instance, an incorrectly picked locality-centric schedule for `sgemm` would yield disastrous results of more than an order of magnitude speed difference, as shown in Chapter 5.

When it comes to making a decision for transformations, a compiler must choose a *better* option over others. This relative comparison is based on measurable quantity on a specific property. In case of the locality-centric schedule, the metric is strideness of memory accesses, reflecting how cache memory works. This is an approximation of the real hardware in a very simplified form, or *modeling*. The virtue of using models in the decision making is that it usually deals with a few critical factors and quickly returns a reasonable answer based on them. A typical optimizing compiler chain is composed of dozens of passes, and the number of potential optimized outcomes is exponential if each pass carries its own decision. Thus, it is crucial to provide a precise yet simple model.

However, the heuristics are often not as accurate as desired in practice. Figure 7.1 demonstrates how different choices of optimizations can result in substantially disparate results of Intel OpenCL stack [5], which is the current state-of-the-art vectorizing compiler. The figure compares the performance of heuristically selected optimization [45] for `sgemm` and `spmv` (denoted as `spmv-jds`) in Parboil [41] against that of a scalar version and two alterna-

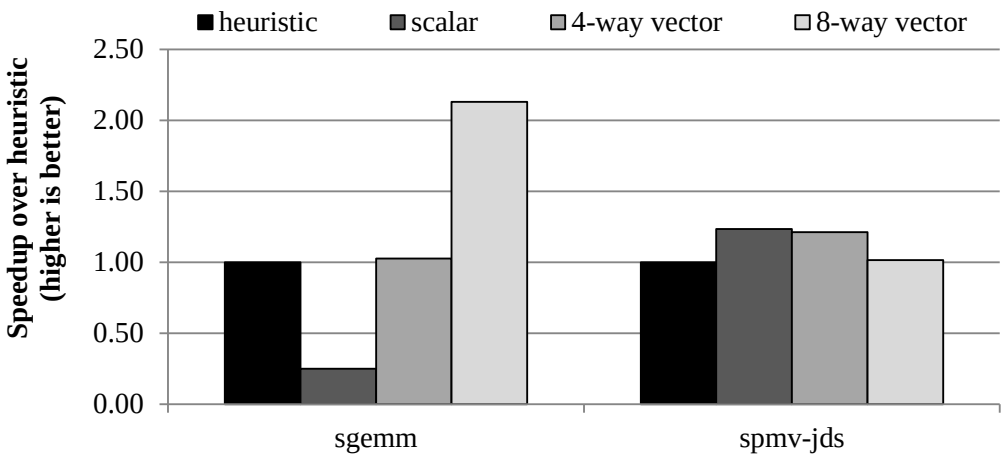


Figure 7.1: Performance of Intel CPU OpenCL stack with different vectorization strategies.

tive vectorized versions. The figure shows that the heuristic has made good but suboptimal decisions for both cases, falling short of the best achievable performance by a factor of 2.13× and 1.24×, respectively. One observation is that the result clearly demonstrates the importance of choosing the most optimal code. Another observation is any single static heuristic for choosing optimizations will likely fall short due to the complexity of interactions between the device, the computation, and the data.

High-precision performance modeling has been a subject of many previous works [5, 13, 20, 27, 30, 46, 47, 48, 49, 50, 51, 52]. Performance models are widely used to prune the design space for autotuning [48, 51], or to guide optimization strategies [30, 47, 49, 50, 51]. The proposed compiler technique presented in Chapter 3 is also an example in this category which employs a heuristic cache model for making a decision on work-item schedule [50]. PORPLE [47] relies on GPU memory or cache models to analyze work-item access patterns of regular applications for data placement. However, these works approach model-specific aspects of the device architecture of interest, while other important factors are not considered or are assumed to be decoupled from the aspects being considered. Such assumptions considerably reduce the accuracy of the model-based approach. Moreover, they are limited by ignoring factors that are only known at runtime, such as the actual data shape. As a result, accurately predicting the effect of optimizations is not likely viable at compile time. Optimizations based on inaccurate predictions

can lead to disappointing performance.

Several runtime-based approaches [47, 53, 54, 55, 56] have proposed to mitigate the problem with the static performance prediction approach. Reactive tiling [55] uses an online trained tiling model and chooses likely optimal tiling parameters for the given working set size and system load. PORPLE [47] leverages runtime micro-simulation on a CPU to refine the GPU memory or cache models, when inputs are irregular and cannot be statically analyzed. Although more information is accessible at runtime, model-driven approaches at runtime can still have limitations and blind spots of unconsidered factors of models like static model-driven approaches, resulting in suboptimal decisions.

To overcome the aforementioned problem, I propose a runtime framework that matches the best code arrangement with the actual device and data combination, thereby improving performance. The proposed approach removes the burden of determining the most optimal code from an optimizing compiler and allows to produce several likely candidate variants from the input code. Then the runtime performs *micro-profiling*, a process of deploying the candidates on a small portion of the actual data on the actual device and determining the best version to be used to process the rest of the workload. The advantage of this approach is that it can work with virtually any combination of compiler and device architecture as dynamic selection could mitigate the cost of having to develop an accurate performance model. When dealing with a large workload, the cost of micro-profiling can be easily amortized, and the overall benefit can be much greater than unconditionally executing a single code selected by the compiler.

The proposed runtime is implemented for work-groups execution at OpenCL driver level. Experimental results demonstrate that the proposed approach correctly chooses the optimal code version with less than 8% overhead in the worst observed case compared to oracle results.

The rest of the chapter begins with design space for this approach, followed by implementation detail and evaluation.

7.2 Design

This section provides a background of profiling for kernel-based data-parallel programming, and introduces the idea of the proposed runtime.

7.2.1 Profiling for Kernel-based Data-parallel Programming

The proposed runtime evaluates different code variants at runtime in order to determine which variant performs best. The runtime measures performance of each variant on a small portion of the actual data and identifies the best performing one, a process which we call *micro-profiling*. The chosen version will be used to process the rest of the workload. The key ingredients of the system are efficient profiling and accurate performance projection.

Popular kernel-based data-parallel programming models, such as OpenCL, CUDA, OpenACC, and C++AMP, allow over-decomposition of workloads for maximized parallelism. Work-groups in OpenCL, for example, are designed to run **independently** of each other enabling efficient parallel execution on a variety of architectures, such as CPUs and GPUs. With these programming models, workload processing is done via repeatedly executing the kernel code over a small subset of the workload, which often takes place in parallel.

The decomposition makes the number of independent kernel executions fairly **large** in practice, which helps to amortize the cost of allocating a few of them for evaluation of code variants. The overhead for evaluating code variants can often be amortized over a large number of executions.

An individual kernel execution is assumed to have **similar performance** throughout subsequent launches. This is due to the nature of data-parallel computing where the same code is used to process large data. Thus, observed performance from a kernel execution is likely to be indicative for others, which helps keep the required sampling frequency, and thus the overhead, low. These properties make work-groups an ideal granularity for micro-profiling.

Figure 7.2 shows accumulated occurrences of kernel launches in different numbers of work-groups from all OpenCL benchmarks in Parboil [41] and Rodinia [35] benchmark suites. The statistics support the low-cost profiling hypothesis based on workload decomposition, as a significant number of kernel launches fall into the range of 128 to 32768 work-groups. Kernel launches

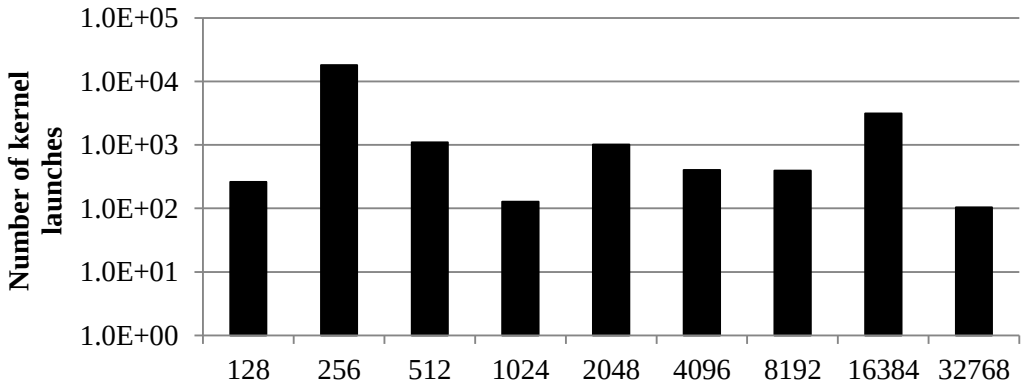


Figure 7.2: Accumulated occurrences of kernel launches categorized by the number of work-groups from Parboil and Rodinia benchmarks.

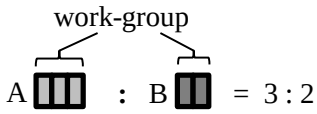
with less than 128 work-groups are rarely observed and so are dropped from the figure. Kernel launches with small number of work-groups can be sensitive to profiling overhead, but a small number of work-groups also indicates relatively small workloads, and performance variation from the level of optimization might not be critical. The proposed technique mainly targets kernels with a large number of work-groups. The profiling-based kernel selection is deactivated for small workload with merely a few work-groups.

7.2.2 Productive Micro-Profiling

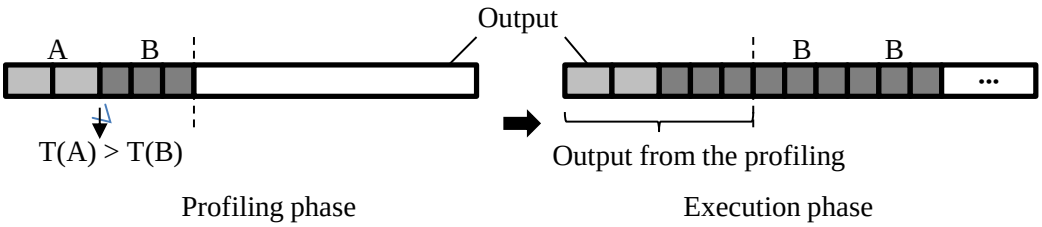
The proposed runtime system employs *productive* micro-profiling, where execution from profiling also participates in the workload processing. Each kernel launch during profiling takes a different part of the workload data. This is a departure from offline profiling, where the performance characteristic is extracted while the result is simply discarded. This strategy reduces the overhead of profiling, since workload processed during profiling does not require reprocessing.

Figure 7.3 shows the three productive profiling techniques used. In this example, we assume that the compiler produces two implementations as follows. The ratio of workload per work-group between version A and B is 3:2 as shown in Figure 7.3(a). According to safe point analysis [55], the runtime launches two and three work-groups for version A and B, respectively, in order to make a fair throughput comparison during profiling.

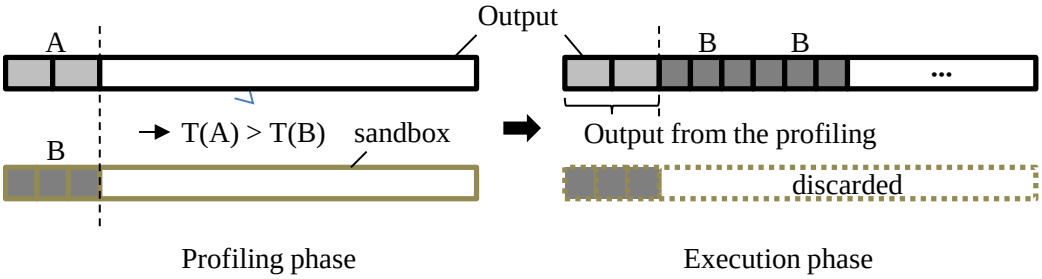
Fully productive profiling, shown in Figure 7.3(b), is the most efficient



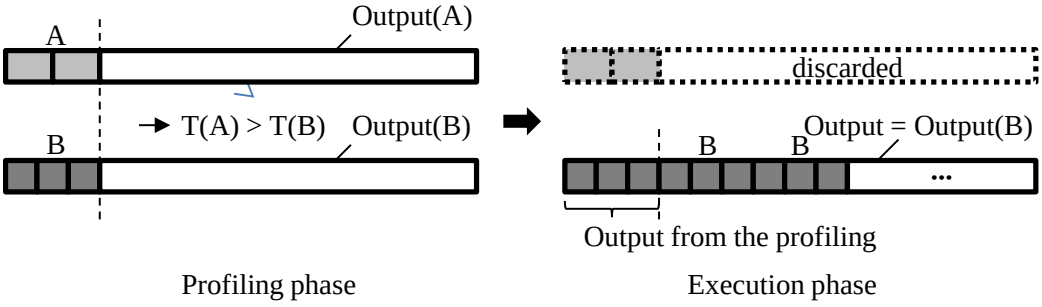
(a) Work assignment for two kernels of A and B per work-group.



(b) Fully productive profiling.



(c) Hybrid-based partial productive profiling.



(d) Swap-based partial productive profiling.

Figure 7.3: Illustration of three profiling modes with example kernels of A and B.

profiling mode. Each kernel launched during profiling takes a different part of the workload data and computes a valid contribution to the final output. In Figure 7.3(b), both kernel versions compute and profile different parts of the workload, and write to the final output. After profiling, version B is chosen to compute the remaining workload, as version B turns out to run faster. Fully productive profiling can take place as long as the individual launches do not have overlap in the final output, which is a dominating pattern in data-parallel programming.

Two other modes of productive profiling, called partially productive profiling, are proposed to overcome the limitations of the applicability of fully productive profiling. Hybrid-based partial productive profiling, shown in Figure 7.3(c), is designed for irregular workload with a non-overlapping final output. By running a set of kernels over the same workload, profiling can be fair among different kernel launches. However, multiple kernels may write to the same memory location, called write conflict. As a solution, both kernel versions compute for the same portion of output, but the other kernel executions are provided with their own private output space or sandbox to avoid the write conflict problem. In this example, version A is assigned with the final output space while version B dumps its output to a private sandbox. After the profiling, version B is chosen to process the remaining workload, by writing its results in the final output, where version A wrote its results during the profiling. The private space allocated for the purpose of profiling is discarded. Since the final output is partially computed by both versions of A and B, it is called hybrid-based.

Swap-based partially productive profiling, shown in Figure 7.3(d), is proposed to allow overlapping outputs by running a set of kernels over the same workload but with their own private output spaces. After profiling, the selected kernel and output (version B and its output in the example shown in Figure 7.3(d)) will remain for the rest of the execution while other kernels as well as their output space are discarded. Since the final output is swapped with output from B, this method is called swap-based. It is worth mentioning that swap-based partial productive profiling can be considered as a speculation approach to version selection.

The runtime can optionally adjust frequency and range of the profiling. When the profiling execution takes place at first, code and data used repeatedly are not fully loaded, resulting in higher execution time due to cache

misses, page faults and TLB misses. Also, irregularity of input during execution may change preferred kernel over time. In order to mitigate these problems, the number of profiling execution and range can be introduced.

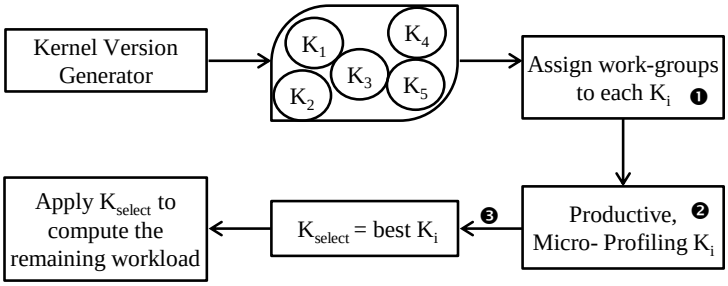
The proposed runtime system relies on compilers or programmers to specify productive profiling mode for a kernel. Interaction between the proposed runtime and compilers and programmers is discussed in more detail later in this chapter.

7.2.3 Applicability

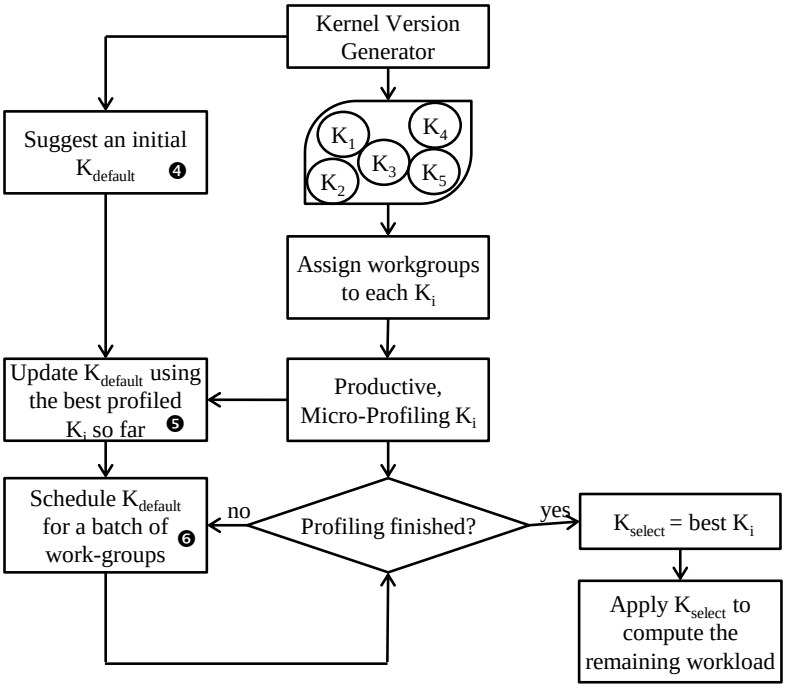
The choice of profiling mode is determined based on programming patterns and optimizations for kernels. First, fully productive profiling is appropriate for kernels with regular or near-regular workloads. Large workload variations can significantly impact the fairness of comparison among kernels. Therefore, fully productive profiling is only suitable for applications with regular workload, such as BLAS, or stencil. Fully productive profiling can select between kernels with different levels of optimizations such as tiling, thread coarsening, data layout transformation (including padding), input binning [2, 57], loop-interchange, locality-centric scheduling [50], vectorization, software prefetching, data placement [47, 49], and input format transformation [58]. Some optimizations require special treatment during profiling. Tiling and thread coarsening require normalization of throughput using safe point analysis [55] to ensure fairness. Data layout transformation, input binning, and input format transformation may require duplication of inputs to meet the assumption of different kernel implementations.

Second, hybrid-based partially productive profiling supports all patterns and optimizations supported by fully productive profiling. Additionally, this profiling method is applicable to irregular workload. By profiling the same portion of workload across different kernels, unfair throughput comparison can be avoided. The applicable kernels typically have in-kernel loops with varying bounds across work-groups, such as sparse BLAS. Uniform workload analysis [20] can be used to detect such in-kernel loops with varying bounds.

Finally, swap-based partial productive profiling further supports output overlapping across kernels, and in theory is applicable to any optimizations, such as privatization, regularization, compaction, output binning, scatter-



(a) Synchronous



(b) Asynchronous

Figure 7.4: Synchronous and asynchronous method flows.

to-gather [2, 57], kernel fusion, kernel fission, optimizations using atomic operations, and even algorithm change. Although swap-based partial productive profiling is the most applicable profiling mode, it has less output contribution efficiency than fully productive profiling.

7.2.4 Orchestration for Profiling and Execution

The way the proposed runtime system orchestrates micro-profiling and execution at runtime can have significant impact on profiling overhead. We

present two orchestration designs in this subsection.

Figure 7.4(a) shows the overall flow of the *synchronous* method. The compiler deposits several code versions to the kernel pool in the executable binary file. Upon execution of the kernel, the runtime dispatches code versions from the pool and executes them (❷) in one of the productive modes described in the previous section, with a few work-groups (❶) assigned using safe point analysis [55]. The runtime waits until all versions finish profiling execution and compares their execution time to pick the best one. Then the rest of the execution runs with the selected kernel. The implementation is simple. However, this method incurs latency penalty if there is large disparity between the best and the worst versions since the latency of the profiling phase is determined by the slowest execution (❸).

Figure 7.4(b) shows the flow for the *asynchronous* method. Unlike the synchronous method, the rest of the execution can begin as soon as the first candidate finishes its micro-profiling execution, even before the profiling is complete. We denote this type of execution as *eager execution*. When the non-profiling execution begins, the runtime launches what is known as the best so far. To support eager execution, the compiler or programmer needs to provide a suggestion on the initial version (❹). We will discuss more on the initial selection and its impact on performance later. As profiling progresses, the selection gets updated once a faster version is found (❺). The asynchronous method must be able to switch to the best kernel found so far; therefore the eager execution is done via launching a series of chunks (❻), instead of a single batch. While the asynchronous method can better tolerate the latency of profiling, the implementation gets more complicated for two reasons. First, the method requires careful workload management so that profiling can be done with a higher priority than the eager execution. Second, the eager execution is divided into many chunks which may impose associated kernel launch overhead.

Figure 7.5 illustrates the execution timing for both synchronous and asynchronous methods. The example assumes that there are four concurrent execution units such as CPU cores and two kernels in the kernel pool. The lighter gray kernel runs faster than the darker one. With the synchronous method, the runtime waits for all kernels to finish profiling execution. This method underutilizes the execution units while waiting for the slow kernel to complete its profiling execution. The asynchronous method overcomes

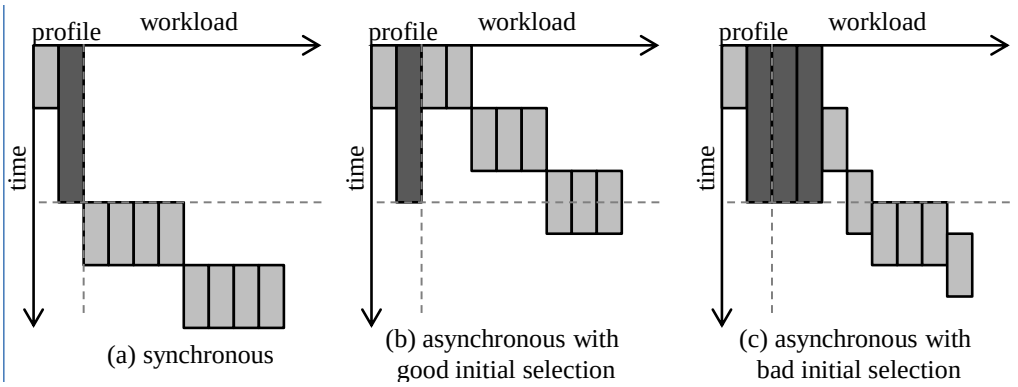


Figure 7.5: Timing illustration for synchronous and asynchronous methods.

this problem by eagerly launching useful work on the vacant execution units with the initially selected version, which is shown Figure 7.5(b) and (c). However, the quality of the initial selection potentially can impact overall performance, as suboptimal code occupies execution units longer, as shown in Figure 7.5(b). In either case, the asynchronous method yields better utilization and throughput compared to the synchronous one.

Table 7.1: Summary of proposed productive profiling, where K is the number of variants in the pool.

Profiling mode	Productive output in profiling	Extra space requirement	Asynchronous support
Fully productive	K	0	Yes
Hybrid-based partial productive	1	$\leq$ K – 1	Yes
Swap-based partial productive	1	$\leq$ K	No

Table 7.1 summarizes throughput, extra space requirement, and support of the asynchronous method for the three proposed productive profiling modes. Given K kernel variants in the kernel pool, all K profiled portions of the workload contribute to the final workload in fully productive profiling, while only 1 profiled portion does so in the two partial productive profiling modes. In terms of extra space requirement, fully productive profiling directly writes results into the original output space and needs no extra space, while the two partial productive profiling methods require at most $K-1$ or K copies of space for either sandboxes or private outputs, respectively. It is worth mentioning that the extra space requirement can be further reduced if the footprint of memory accesses during profiling can be predicted so that a sub-

set of output is allocated for the sandbox. Last, both fully productive profiling and hybrid-based partial productive profiling support the asynchronous method, since profiling results are directly written into the distinct, final output space, while swap-based partial productive profiling cannot support the asynchronous method, because the final output space is not determined until profiling is complete.

7.3 Implementation

7.3.1 Runtime Interface

Unlike traditional runtimes, the proposed runtime allows compilers or programmers to deposit multiple implementations under the same kernel function signature. Figure 7.6(a) shows the kernel implementation registration API. The specific requirement for the runtime is to provide *work assignment factor*, which is the number of workload units packed into each work-group for accurate profiling, as shown in Figure 7.3. Figure 7.6(b) shows the kernel launch API. The API is designed to allow the caller to specify whether profiling is activated or not using a *profiling activation flag* along with *profiling mode*.

Work Assignment Factor. Resource management is an important class of optimizations for OpenCL programs because hardware utilization can be improved. Among them, coarsening [9] and tiling change the amount of work assigned to each thread and thus the work assignment per kernel launches. The runtime needs to know the relative work assignment between variant kernels for fair comparison. Once such workload changing optimization is done, the compiler needs to inform the runtime about the change. When user provided kernels are used, programmers are in charge of providing the correct work assignment ratios.

Profiling Activation Flag. A class of applications, such as stencil operations in partial differential equation (PDE) solvers or sparse matrix-vector multiplication (spmv) in conjugate gradient (CG) iterative solvers, launches a kernel iteratively without changing workload or data shape between iterations. In this scenario, the kernel can just be profiled in the first iteration and the selected variant can be reused for the later iterations. The profiling

```
AddKernel(
  string kernel_sig,              // kernel name
  func_ptr implementation,        // kernel implementation
  dim3 wa_factor,                 // work assignment factor
  vector<int> sandbox_index=[]    // argument offsets for
                                  // private outputs
);
```

(a) Kernel implementation registration API

```
LaunchKernel(
  string kernel_sig,              // kernel name
  bool profiling=true,            // profiling activation flag
  enum mode=fully_async           // profiling mode
);
```

(b) Kernel launch API

Figure 7.6: Runtime interface.

activation flag allows the user to turn on profiling only for the first iteration. When the flag is turned off, the runtime launches the default kernel without profiling, which may have been selected from previous profiling executions.

Profiling Mode. As mentioned previously, applicability, throughput, and cost are profiling mode specific factors. Different classes of optimizations require their own productive profiling mode for efficient and fair profiling. The asynchronous profiling potentially can reduce overheads of profiling.

7.3.2 Implementation for OpenCL Runtime

Work distribution and prioritized execution for profiling are two main requirements for CPU implementation of the proposed approach. Intel's TBB [26] has strong support for both and thus is used for the implementation. TBB's work stealing feature provides load balancing over multiple cores while its concurrent task groups allow assigning higher scheduling priority to profiling execution. In the profiling task group, kernel launches are wrapped by timer calls to measure execution time of a kernel being profiled. Updating the current best implementation is done via atomic operation when the execution time of a variant is found to be smaller than the current minimum. The non-profiling task group invokes the current best implementation upon launch. When profiling is activated, the runtime first launches the profiling task group with higher priority, which is followed by launching the non-profiling task

```
class NDRange : public tbb::task {
  tbb::task* execute() {
    if (is_profiling_on) {
      uint64_t sel = klist.getDefaultKernel();
      if (profiling_mode == SYNC) {
        ProfileTask p(klist, kargs, &sel);
❶       wait_for_all();
        ExecuteTask e(&sel, klist, kargs, true);
        wait_for_all();
      } else if (profiling_mode == ASYNC) {
        ProfileTask p(klist, kargs, &sel);
❷       ExecuteTask e(&sel, klist, kargs, true);
        enqueue(e, tbb::priority_low);
        wait_for_all();
      }
    } else {
❸     ExecuteTask e(&sel, klist, kargs, false);
      wait_for_all();
    }
    return NULL;
  }
};

static NDRange::run(klist, args) {
  NDRange ndrange(klist, kargs);
  wait_for_all();
}
```

Figure 7.7: Top-level task management for profiling and non-profiling work-group executions.

group. The synchronous mode puts a barrier to wait for the profiling task group to finish its execution between the two task group launches, while the asynchronous mode schedules both task groups concurrently. When profiling is not activated, the runtime launches the non-profiling task group only.

Figure 7.7 depicts the simplified pseudo-code of the runtime, which implements the sketch described above. Both profiling and execution are mapped to TBB's task, `ProfileTask` and `ExecuteTask`, respectively. In the synchronous method, the runtime launches `ProfileTask` first and waits the execution to finish, which is followed by launching `ExecuteTask` (❶). The `true` flag in instantiating `ExecuteTask` indicates that profiling is activated. In the asynchronous method, `ExecuteTask` is enqueued with lower priority than others, `ProfileTask` in this particular case. Both tasks will run together(❷)

```
ProfileTask {
  tbb::task* execute() {
    for each k in klist,
      Profile p(k, kargs, p_sel);
  }
}

Profile {
  tbb::task* execute() {
    elapsed = 0;
    for (i = 0; i < Nsample; ++i) {
      begin = timer();
      k(kargs);                       // work-group launch
      elapsed += (timer() - begin);
      atomicUpdateIfMin(              // update kernel selection
        p_sel, (elapsed / (i+1), k.index) );
    }
  }
}
```

Figure 7.8: Profile task implementation.

as the parent task schedules them. When profiling is not activated, the runtime launches `ExecuteTask` only, without the profiling task(❸). Similarly, the last flag in instantiating `ExecuteTask` tells that profiling is not activated.

Figure 7.8 shows simplified code for the profiling task. In `execute` method, the runtime creates a task for each and every candidate kernel, which can run in parallel. Each task is created by instantiating `Profile` class with a unique kernel to measure performance. Upon execution, the task runs the assigned kernel for `Nsample` times to smooth out glitches associated with sampling. This is necessary because the timing measurement from earlier execution may not be accurate as hardware is not fully warmed up, due to the cold cache miss effect. The sampling also copes with sporadic system noise. When the averaged execution time turns out to be smaller than the current minimum, the profiling task atomically updates the current kernel selection as well as the new minimum execution time to a designated location, `p_sel` in this case. `atomicUpdateIfMin` is implemented using the compare-and-swap intrinsic.

The rest of the workload processing is done in parallel, as shown in Figure 7.9. As mentioned before, the task has lower priority so that profiling can be done faster. The task skips the first `Nsample` × `klist.size()` workgroups when profiling is activated, because the profiling will cover the area. Dropping a fraction of workload may result in load imbalance, but TBB's

```
ExecuteTask {
  tbb::task* execute() {
    parallel_for(range(0, numWorkGroups), *this);
  }

  void operator()(range& r) {
    if (is_profiling_on)
      if (r < klist.size() * Nsample)
        return;
    k = klist[*p_sel];            // fetch the current best
    k(kargs);                     // launch work-group
  }
}
```

Figure 7.9: Execution task implementation.

work stealing would resolve the issue over time. The task fetches the current best kernel at the time of launching a kernel which may dynamically change as profiling progresses. Again, `p_sel` is used to communicate with the profiling task about the current best kernel implementation.

7.4 Evaluation

This section evaluates the proposed runtime. The implementation is done as part of the prototype OpenCL stack which includes the locality-centric OpenCL compiler. The experiments are done with selected benchmarks from Parboil, Rodinia and SHOC [59] benchmark suites. Different experiments subscribe to their own sets of benchmarks according to their own purposes, which are individually described for each. The system configuration and software used are the same as detailed in Chapter 5.

7.4.1 Comparison to Static Heuristic

In this subsection, evaluation of the proposed runtime with the locality-centric scheduling of work-item executions for CPUs is presented. There are four benchmarks used - `sgm`, `spmv` (denoted as `spmv-jds`), `stencil` and `ctcp` from Parboil, `kmns` from Rodinia, and another `spmv` using scalar dot products on a CSR format matrix without padding (denoted as `spmv-csr`) from SHOC. These benchmarks are selected because their CPU performance is sensitive to the scheduling policy. The inputs for Parboil and Rodinia

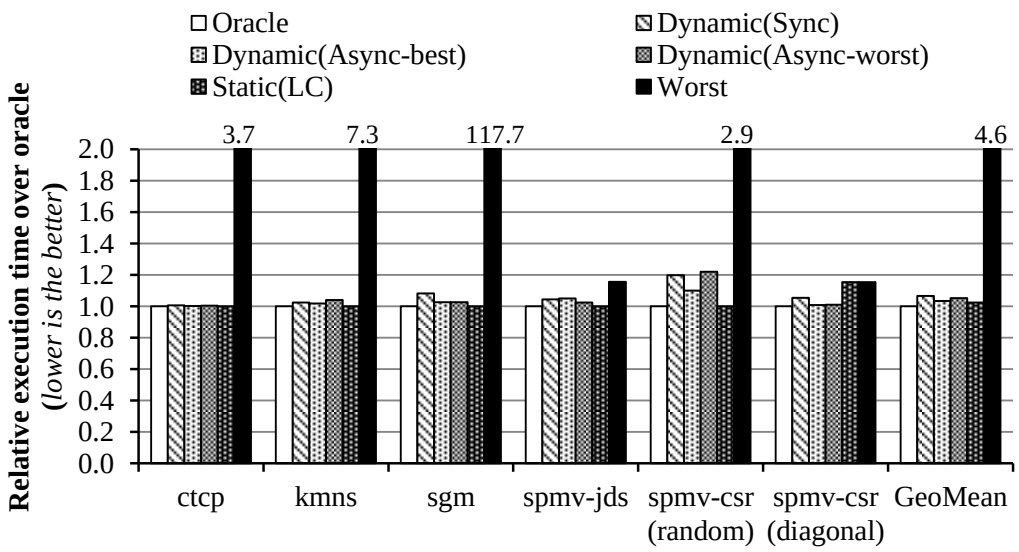


Figure 7.10: Performance of the proposed runtime when both of DFO and BFO kernels are used as candidates.

benchmarks are all default, while the inputs for `spmv-csr` include a $16K$-by-$16K$ random sparse matrix with 1% probability of non-zeros, denoted as `random`, and a $2M$-by-$2M$ diagonal matrix, denoted as `diagonal`. Fully productive profiling is used for the benchmarks, except `spmv-jds` and `spmv-csr`, both of which use hybrid-based partial productive profiling as they handle irregular workload. The three benchmarks of `ctcp`, `kmns` and `sgm` are chosen as they have increasing order of magnitude in the performance gap between the oracle and the worst. The spmv kernels are chosen as they exhibit data-dependent behavior, which may cause static heuristic to fail.

Figure 7.10 compares performance of the proposed runtime and compile-time heuristic over oracle. LC indicates the locality-centric compiler presented in Chapter 3 which represents the compile-time heuristic. For this experiment, the LC compiler generates both versions of DFO and BFO scheduled kernels for the most significant loop and registers them to the runtime. Oracle represents the best selection among DFO and BFO for each benchmark, whereas Worst means the opposite.

The proposed runtime achieves close to optimal results with negligible overhead. The result also reaffirms that different schedule choices can result in substantial performance differences. A heuristic-based static selection could have caused a large performance loss with a suboptimal decision. However,

the proposed approach correctly selects the optimal schedule for all given benchmarks. In the case of `spmv-csr` with the `diagonal` input, the LC static heuristic selected incorrectly but the mistake is avoided with the proposed approach. When no or little performance variation due to input distribution is expected, the runtime chooses the optimal from profiling, witnessed by `ctcp`, `kmns` and `sgm`. In a situation where input distribution has a high impact on data locality, such as `spmv-csr`, the static approach works well with a certain input distribution, but a statically chosen kernel cannot cope with all possible cases with equal efficiency. The proposed runtime approach, on the other hand, adaptively chooses between two schedules, yielding close to optimal performance for both cases of `spmv-csr`.

The adaptability comes at a cost. For the synchronous method, the overall overhead becomes significant when the number of work-groups is relatively small or the ratio of best to worst is large. For instance, `sgm` has the sharpest performance gap between DFO and BFO and the synchronous method has to tolerate executing the worst kernel. When the ratio is small, such as `ctcp`, profiling only adds negligible overhead. The overall overhead for the synchronous method is 7%.

The asynchronous method shows better performance than the synchronous mode. This confirms that the method hides latency that yields better performance, unlike the synchronous method. The initial selection seems relevant and matters to some benchmarks. With correct initial selection, the average overhead is 3% compared to the oracle. With the worst initial selection, the overhead increases to 5%. Although the dynamic runtime selection certainly guarantees closer to the optimal than to the worst performance, the result implies that reasonable static performance modeling can be helpful for better performance.

Another scenario in which the proposed approach is useful is when a compiler cannot foresee the impact of combined optimizations due to the limitation of modeling. OpenCL programs are often optimized with multiple optimizations such as tiling, coarsening and data placement using scratchpad. Figure 7.11 (a) compares the performance of the proposed approach when differently optimized kernels are provided. Each benchmark deposits two kernels with different optimization level from the Parboil benchmark suite, called `naive` and `opt`. LC is used to compile these kernels but no heuristic is provided to compare each pair, and only a random selection could be used.

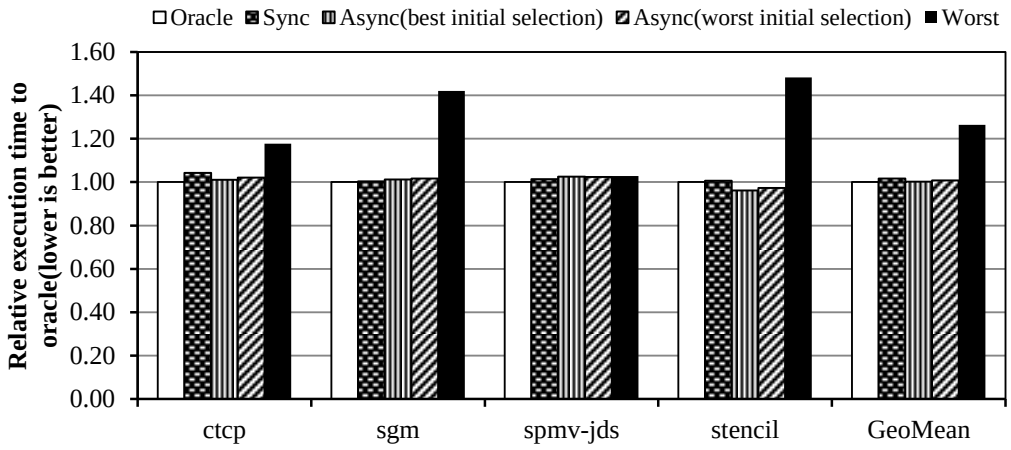


Figure 7.11: Performance comparison when differently optimized kernels are used.

The intention of this experiment is to demonstrate the adaptability of the runtime to the random selection.

The proposed approach achieves near optimal results for all benchmarks, less than 2% overhead on average compared to oracle for all methods. Interestingly, the naive versions are always best for CPU as they allow the greatest flexibility for the compiler in planning how to serialize execution of work-items. GPU-specific optimizations such as data placement and data prefetching using scoreboarding make no difference for CPU. Tiling using scratchpad memory typically leads to negative results on CPUs because there is no latency gain using them after they are lowered to CPU's uniform memory space.

7.4.2 Input Adaptability

In this subsection, an input-dependent version selection scenario using spmv-csr from the SHOC benchmark suite is tested. Two `spmv-csr` versions, one using scalar dot product (denoted as scalar) and the other using vector dot product (denoted as vector), are chosen. The optimal version of `spmv` on a CSR-format matrix is highly dependent on matrix sparsity [58], which is typically unknown at compile time. The evaluation is performed with two matrices, the random sparse matrix and the diagonal matrix, described in the previous experimental setup. The profiling modes are also the same as in the previous experiment.

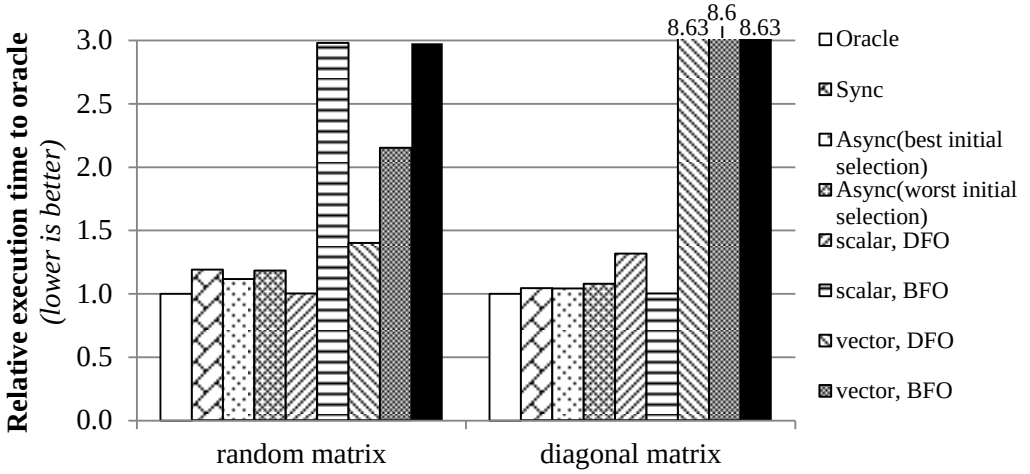


Figure 7.12: Performance results on input-dependent kernels.

The purpose of this experiment is to demonstrate the adaptive selection capability of the proposed approach when the compiler simply cannot predict the performance due to lack of critical information, which is sparsity of the actual matrix in this experiment.

Figure 7.12 shows performance of the proposed method compared to that of a scalar kernel and a vector kernel on all possible combinations of work-item scheduling for them. As for `random` input, the proposed runtime performs second best to oracle with up to 19% and 12% overheads, respectively, for the synchronous and the asynchronous methods. With `diagonal` input, the runtime runs with 4% overhead to oracle for both methods. The selection here is particularly complicated by the dimension of schedule, kernel version and input data distribution. Here, LC chooses DFO to iterate in-kernel loops first for both scalar and vector implementations and uses the code unconditionally. However, the static choice does not cope well with unfavorable input distribution from the diagonal matrix, where BFO schedule is desired. As for version selection among scalar and vector, scalar performs better when DFO is chosen, mainly because of less overhead having to deal with control divergence. In terms of instruction counts, the vector version is inferior because the code uses local memory which causes additional copy of data for final reduction, an overhead in instruction throughput which LC does not consider with its heuristic. This is a reason why BFO schedule for the vector kernel performs similar to DFO even though BFO achieves favorable data locality.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

In this dissertation, I presented a compiler technique for high-performance OpenCL programs on CPUs. The proposed technique selects a schedule for work-items execution such that data locality is best exploited. The state-of-the-art (depth-first) approach to scheduling work-items in existing OpenCL compilers for CPUs can result in suboptimal memory access patterns for certain workload classes. An alternative (breadth-first) work-item schedule is proposed which provides scheduling similar to that expected by GPU-optimized programs. Static analyses and transformation techniques are also proposed in order to correctly select and generate the schedule for better data locality. The proposed locality centric scheduling results in geomean L1 data cache miss reductions of 9.81× over AMD and 3.35× over Intel, and geomean speedups of 3.32× over AMD and 1.71× over Intel, based on real hardware measurements. As the memory system becomes increasingly important for performance and energy efficiency in future computing systems, the appropriate selection of work-item schedules will play an even more important role in the future.

The importance of data locality in data parallel programs is critical and specialized architectures such as GPUs encourage programmers to optimize a program to accommodate good data locality. Previous OpenCL compilers targeting CPUs are oblivious to how data locality is exploited in such programs and resulted in unfavorable performance. This had contributed to the widespread belief that portable performance is infeasible from GPU to CPU. The observation on data locality and adaptive scheduling toward better memory system efficiency recovers significant performance.

To that end, the compiler technique discussed in this dissertation also opens up a new opportunity of scheduling technique in an angle of loop transformations for OpenCL programs and alike. Complete independence between execution of work-items allows GPUs and other parallel architectures to ex-

ploit parallelism. When serialization of work-items execution is desirable, such as targeting CPUs, the property is translated into the work-item loop, which is inherently a canonical loop with no loop carried dependency. This strong assumption enables a high degree of freedom in scheduling work-items, as discussed in this work. Combining with other types of loop transformations would further improve performance from CPUs, as exemplified in BLAS kernels studies shown in Chapter 6.

As compilers incorporate increasing numbers of advanced optimizations, it is unavoidable to rely on static performance modeling to guide through the compiler phases. Data locality scheduling presents one such challenge for which the adaptive heuristic works reasonably well. However, the inherent limitation of modeling makes finding an optimal solution extremely challenging. To address this issue, I proposed a solution to mitigate the burden of having to pick the best implementation by offloading the selection process to runtime. Instead of emitting one output, the proposed runtime allows a compiler to deposit differently optimized programs. The runtime evaluates their performance using a fraction of the workload and chooses the best version for the rest of workload processing. The proposed approach is implemented as a runtime to support the locality-centric OpenCL compiler, which shows close to optimal results. As developing precise static performance modeling gets harder, the importance of adaptive runtime solution will draw more attention.

As for future work, I believe that locality optimization can be further improved with more accurate analyses of access patterns and improved code generation schema. Furthermore, the runtime selection mechanism can be further improved based on the experience in the experiments reported in this dissertation.

APPENDIX A

CODE GENERATION EXAMPLE FOR SPMV

In this chapter, a complete code generation example is presented using `spmv` in Parboil benchmark suite. It has divergent IF- and LOOP- statements with a BFO loop, making it a perfect case to demonstrate many features of the code generation. As discussed in Chapter 3, the kernel loop prefers BFO schedule. Subregion formation is assumed to be done, which is indicated in comments along with the generated code. Figure A.1 shows the kernel code in OpenCL.

```
int ix = get_global_id(0);
if (ix < dim) {
    float sum = 0.0f;
    int bound=sh_zcnt_int[ix/32];
    for(int k=0;k<bound;k++) {
        int j = jds_ptr_int[k] + ix;
        int in = d_index[j];
        float d = d_data[j];
        float t = x_vec[in];
        sum += d*t;
    }
    dst_vector[d_perm[ix]] = sum;
}
```

Figure A.1: Code listing of `spmv`.

The output for the input code is shown below. Note that the code is not simplified but real output code from the prototype compiler, retouched only with formatting and renaming variables. The output code is a legal C code.

```
1 #define workItemLoop for(unsigned int wid = 0; wid < LS; wid++)
2
3 // SUBREGION 1 ================================================
4 // int ix = get_global_id(0); ================================
5 int ix = __get_global_id[0];
6
```

```
7  // DIVERGENT IF ==========================================
8  // if (ix < dim) { ==========================================
9  unsigned int pred0[LS];
10 unsigned int numActive0 = 0;
11 workItemLoop {
12     if (ix+wid < dim) {
13         pred0[wid] = 1;
14         numActive0++;
15     } else {
16         pred0[wid] = 0;
17     }
18 }
19 if (numActive0 > 0) {
20
21     // SUBREGION 2 ==========================================
22     // float sum = 0.0f; ==========================================
23     // int bound=sh_zcnt_int[ix/32]; ==========================================
24     float sum[LS];
25     int bound[LS];
26     if (numActive0 == LS) {
27         #pragma simd
28         workItemLoop {
29             sum[wid] = 0;
30             bound[wid] = sh_zcnt_int[(ix+wid)/32];
31         }
32     } else {
33         workItemLoop {
34             if(pred0[wid]) {
35                 sum[wid] = 0;
36                 bound[wid] = sh_zcnt_int[(ix+wid)/32];
37             }
38         }
39     }
40
41     // DIVERGENT LOOP ==========================================
42     // for(int k=0;k<bound;k++) { ==========================================
43     int k = 0;
44     unsigned int pred1[LS];
45     unsigned int numActive1 = 0;
46     workItemLoop {
47         if (pred0[wid] && k<bound[wid]) {
48             pred1[wid] = 1;
49             numActive1++;
```

```
50          } else {
51              pred1[wid] = 0;
52          }
53      }
54      while (numActive1 > 0) {
55
56          // SUBREGION 3 ==========================================
57          // int j = jds_ptr_int[k] + ix; =========================
58          // int in = d_index[j]; =================================
59          // float d = d_data[j]; =================================
60          // float t = x_vec[in]; =================================
61          // sum += d*t; ==========================================
62          if (numActive1 == LS) {
63              #pragma simd
64              workItemLoop {
65                  int j = jds_ptr_int[k] + ix;
66                  int in = d_index[j+wid];
67                  float d = d_data[j+wid];
68                  float t = x_vec[in];
69                  sum[wid] += d*t;
70              }
71          } else {
72              workItemLoop {
73                  if (pred1[wid]) {
74                      int j = jds_ptr_int[k] + ix;
75                      int in = d_index[j+wid];
76                      float d = d_data[j+wid];
77                      float t = x_vec[in];
78                      sum[wid] += d*t;
79                  }
80              }
81          }
82
83          // DIVERGENT LOOP EPILOGUE ==============================
84          k++;
85          workItemLoop {
86              if (pred1[wid] && !(k<bound[wid])) {
87                  pred1[wid] = 0;
88                  numActive1--;
89              }
90          }
91      }
92
```

```
93      // SUBREGION 4 ==========================================
94      // dst_vector[d_perm[ix]] = sum; ========================
95      if (numActive0 == LS) {
96          #pragma simd
97          workItemLoop {
98              dst_vector[d_perm[ix+wid]] = sum[wid];
99          }
100     } else {
101         workItemLoop {
102             if (pred0[wid]) {
103                 dst_vector[d_perm[ix+wid]] = sum[wid];
104             }
105         }
106     }
107 }
```

REFERENCES

[1] P. Xiang, Y. Yang, M. M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, "Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement," *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing - ICS '13*, p. 433, 2013.

[2] J. Stratton, N. Anssari, C. Rodrigues, I. Sung, N. Obeid, L. Chang, G. Liu, W. Hwu et al., "Optimization and architecture effects on GPU computing workload performance," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–10.

[3] J. A. Stratton, "Performance portability of parallel kernels on shared-memory systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, May 2013. [Online]. Available: http://hdl.handle.net/2142/44383

[4] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting GPU compute applications within GPU Ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 9:1–9:9.

[5] N. Rotem, "Intel OpenCL implicit vectorization module," *LLVM Developer Meeting*, 2011.

[6] J. H. Lee, K. Patel, N. Nigania, H. Kim, and H. Kim, "OpenCL performance evaluation on modern multi core CPUs," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1177–1185.

[7] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance traps in OpenCL for CPUs," in *Parallel, Distributed and Network-Based Processing, 2013 21st Euromicro International Conference on*, Feb. 2013, pp. 38–45.

[8] A. Ali, U. Dastgeer, and C. Kessler, "OpenCL for programming shared memory multicore CPUs," in *Proceedings of the 5th Workshop on MULTIPROG, in conjunction with HiPEAC*, 2012.

[9] A. Magni, C. Dubach, and M. F. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, p. 11.

[10] S. Seo, G. Jo, and J. Lee, "Performance characterization of the NAS parallel benchmarks in OpenCL," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, Nov 2011, pp. 137–148.

[11] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund et al., "Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 451–460.

[12] Khronos OpenCL Group, *The OpenCL Specification*, 2008.

[13] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "Twin Peaks: A software platform for heterogeneous computing on general-purpose and graphics processors," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010, pp. 205–216.

[14] "Beignet," 2015. [Online]. Available: https://01.org/beignet

[15] "NVIDIA OpenCL SDK," 2015. [Online]. Available: https://developer.nvidia.com/opencl

[16] "Mali OpenCL SDK," 2015. [Online]. Available: http://malideveloper.arm.com/develop-for-mali/sdks/mali-opencl-sdk/

[17] "PowerVR SDK," 2015. [Online]. Available: http://community.imgtec.com/developers/powervr/

[18] "Intel SDK for OpenCL Applications," 2015. [Online]. Available: https://software.intel.com/en-us/intel-opencl

[19] "OpenCL Development Kit for Linux on Power," 2015. [Online]. Available: http://www.alphaworks.ibm.com/tech/opencl

[20] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable OpenCL implementation," 2014.

[21] "FreeOCL," 2015. [Online]. Available: https://code.google.com/p/freeocl/

[22] "MCSDK HPC 3.x OpenCL," 2015. [Online]. Available: http://processors.wiki.ti.com/index.php/MCSDK_HPC_3.x_OpenCL

[23] "Altera SDK for OpenCL," 2015. [Online]. Available: http://dl.altera.com/opencl/

[24] "MAGMA," 2015. [Online]. Available: http://icl.cs.utk.edu/magma/software/

[25] "OpenCV," 2015. [Online]. Available: http://opencv.org/

[26] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[27] J. A. Stratton, S. S. Stone, and W. W. Hwu, "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs," in *Languages and Compilers for Parallel Computing*, J. N. Amaral, Ed., 2008, pp. 16–30.

[28] L. Chang, J. Stratton, H. Kim, and W. Hwu, "A scalable, numerically stable, high-performance tridiagonal solver using GPUs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 27:1–27:11.

[29] I. Sung, J. A. Stratton, and W. 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*, 2010, pp. 513–522.

[30] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 86–97.

[31] 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*, 2012, pp. 341–352.

[32] R. Karrenberg and S. Hack, "Improving performance of OpenCL on CPUs," in *Proceedings of the 21st International Conference on Compiler Construction*, 2012, pp. 1–20.

[33] S. Mahlke, R. Hank, J. McCormick, D. August, and W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *In Proceedings of the 22th International Symposium on Computer Architecture*, 1995, pp. 138–150.

[34] S. Timnat, O. Shacham, and A. Zaks, "Predicate vectors if you must," in *WPMVP '14: Workshop on Programming Models for SIMD/Vector Processing*, 2014.

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

[36] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira, "Divergence analysis and optimizations," in *Parallel Architectures and Compilation Techniques, 2011 International Conference on*, Oct. 2011, pp. 320–329.

[37] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Interprocedural parallelization analysis in SUIF," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 4, pp. 662–731, July 2005.

[38] A. Kerr, G. Diamos, and S. Yalamanchili, "Dynamic compilation of data-parallel kernels for vector processors," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 23–32.

[39] R. Karrenberg and S. Hack, "Improving Performance of OpenCL on CPUs," in *Proceedings of the 21st International Conference on Compiler Construction*, 2012, pp. 1–20.

[40] S. Eranian, "Perfmon2: A flexible performance monitoring interface for Linux," in *Proc. of the 2006 Ottawa Linux Symposium.* Citeseer, 2006, pp. 269–288.

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

[42] Intel, "Intel math kernel library," 2007.

[43] Z. Xianyi, W. Qian, and Z. Chothia, "OpenBLAS," 2013. [Online]. Available: http://www.openblas.net/

[44] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

[45] "Vectorizer knobs," 2015. [Online]. Available: https://software.intel.com/en-us/node/540483

[46] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, and W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *ACM Sigplan Notices*, vol. 45, no. 5, 2010, pp. 105–114.

[47] G. Chen, B. Wu, D. Li, and X. Shen, "PORPLE: An extensible optimizer for portable data placement on GPU," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 88–100.

[48] Y. Dotsenko, S. Baghsorkhi, B. Lloyd, and N. Govindaraju, "Auto-tuning of fast Fourier transform on graphics processors," in *ACM SIGPLAN Notices*, vol. 46, no. 8, 2011, pp. 257–266.

[49] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 105–118, 2011.

[50] H. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W. Hwu, "Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015, pp. 257–268.

[51] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S. Ueng, J. Stratton, and W. Hwu, "Program optimization space pruning for a multithreaded GPU," in *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2008, pp. 195–204.

[52] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *ACM SIGPLAN Notices*, vol. 47, no. 8, 2012, pp. 11–22.

[53] J.-F. Dollinger and V. Loechner, "Adaptive runtime selection for GPU," in *Parallel Processing, 2013 42nd International Conference on*, 2013, pp. 70–79.

[54] L. Li, U. Dastgeer, and C. Kessler, "Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems," in *High Performance Computing for Computational Science-VECPAR 2012*, 2013, pp. 329–345.

[55] J. Srinivas, W. Ding, and M. Kandemir, "Reactive tiling," in *Code Generation and Optimization, 2015 IEEE/ACM International Symposium on*, 2015, pp. 91–102.

[56] J. R. Wernsing and G. Stitt, "Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *ACM SIGPLAN Notices*, vol. 45, no. 4, 2010, pp. 115–124.

[57] J. Stratton, C. Rodrigues, I. Sung, L. Chang, N. Anssari, G. Liu, W. Hwu, and N. Obeid, "Algorithm and data optimization techniques for scaling to massively threaded systems," *Computer*, vol. 45, no. 8, pp. 0026–32, 2012.

[58] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 18:1–18:11.

[59] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63–74.