TOWARDS A SYSTEMATIC EXPLORATION OF THE OPTIMIZATION SPACE FOR MANY-CORE PROCESSORS

TOWARDS A SYSTEMATIC EXPLORATION OF THE OPTIMIZATION SPACE FOR MANY-CORE PROCESSORS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K. C. A. M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 21 oktober 2014 om 12:30 uur

door

Jianbin FANG

Master of Engineering in Computer Science and Technology,
National University of Defense Technology, China
geboren te Qingdao, China.

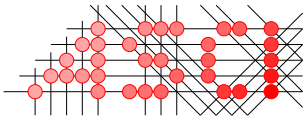
Dit proefschrift is goedgekeurd door de promotor:

Prof.dr.ir. H.J. Sips

Copromotor: Dr.ir. A.L. Varbanescu

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft, promotor
Dr.ir. A.L. Varbanescu,	Universiteit van Amsterdam, copromotor
Prof.dr. K.L.M. Bertels,	Technische Universiteit Delft
Prof.dr. H. Corporaal,	Eindhoven University of Technology, the Netherlands
Prof.dr. P.H.J. Kelly,	Imperial College London, United Kingdom
Prof.dr. C.W. Kessler,	Linköping University, Sweden
Prof.dr. W. Zhang,	National University of Defense Technology, China
Prof.dr. C. Witteveen,	Technische Universiteit Delft, reservelid



The work described in this thesis has been carried out in the ASCI graduate school. ASCI dissertation series number 314.

The work was supported by China Scholarship Council (CSC).

Keywords: Multi-/Many-core Processors, Performance, Portability, Vectorization, Memory Hierarchy, Local Memory, OpenCL

Printed by: Wöhrmann Print Service in the Netherlands

Front & Back: The cover image (designed by Kun Sun) is entitled “Magic Cube”. The connection to the thesis is that improved performance of parallel programs can be achieved by using “patterns” and code “transformations” – much like what happens when we play with a Rubik’s Cube.

Published and distributed by: Jianbin Fang
Email: j.fang.cn@gmail.com

ISBN 978-94-6186-378-2

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

ACKNOWLEDGEMENTS

The past four years will definitely be one of the most memorable times in my life. During this time, I have been helped by many people around me. This thesis could not have been achieved without their help. Here I would like to thank them one by one.

I would like to express my thanks to China Scholarship Council, for making my overseas study in the Netherlands possible. Also, I would like to thank Koen Bertels, Henk Corporaal, Paul Kelly, Christoph Kessler, Weimin Zhang, and Cees Witteveen for accepting to be part of my committee, and for their valuable comments on the thesis draft.

I want to thank my supervisors Henk Sips and Ana Lucia Varbanescu. Henk, you are a supervisor with great wisdom and humour. Thank you for your consistent support and encouragement. I appreciate your broad vision and valuable suggestions during my research. Thank you for calling me *Mr. Phi*. Actually, I really appreciate this "title", and it gives me some confidence. Thank you also for being a pointer to the international workshops/seminars which were beneficial for me. Besides, the chats with you influenced me more than I can describe.

Ana, I am honoured to be your first PhD student. I would like to say "thank you", although words cannot express my gratitude. The emails between you and me tell the stories. You are always kind and tolerant. You have taught and trained me a lot on my way of conducting research. I will never forget the effort you made for each of my papers (from beginning to the last minutes). When I was off the track, you were always the person to "drag" me back. Both the professional knowledge and the code of practice as a computer scientist, I learned from you, will continuously guide me in my future career.

In ancient China, one learned from a master who was living in the mountains or forests. Today, we can learn from experts who are working on the other side of the Internet. In my world, these guys are Pekka Jääskeläinen from TUT, Finland, Evghenii Gaburov from SURFsara, the Netherlands, Georg Hager from FAU, Germany. Pekka, my one-week visit to you got me into the compilation world. Btw, your colleagues were really helpful and I thank all of them. Thank you, Evghenii, for the on-line discussions on Xeon Phi and micro-benchmarking. Your neat and efficient code was impressive. Georg, I was impressed by the way you optimize numeric code (just like an art) at the aiXcelerate workshop, and thank you for our on-line discussions. All these experiences will not only be beneficial for my thesis, but, more importantly, for my whole life.

I want to thank my collaborators: Gorkem Saygili and Laurens van der Maaten on stereo matching, Lilun Zhang and Chuanfu Xu on Tianhe-2. Gorkem and Laurens, I have learnt a lot from our collaboration, and, more importantly, this collaboration indicated a path for my thesis. Gorkem, you are an enthusiastic guy, always pushing the work to its next step. Lilun and Chuanfu, thank you for your invitation which provided me with an opportunity to work on Tianhe-2. I really appreciated the working experience (in particular, the intensive NEMO5 compilation) on the giant.

During these years, my officemates have created a pleasant working atmosphere. Boudewijn, thanks for ordering the white board for me. Arno, thank you for helping me with Latex when writing my first paper. Bogdan, I have enjoyed the music and the tour experience you shared with me. Jie, thank you for including me in your work and publications. I hope that our discussions were helpful to you. Marcin, you are one smart guy and I believe you will make a great fortune from your genius ideas. Just stick to them!

I would also like to thank my colleagues. Alex, thank you for the suggestions you gave for my posters and presentations. Alexander, thank you for your help at my PhD startup. Boxun, I really enjoyed your Beijing-style jokes and our numerous casual chats. Siqi, thank you for sharing your life experience in the Netherlands. Otto and Sietse, thank you for translating my thesis summary and propositions in Dutch. Adele, Siqi, Yong, and Jie, we have enjoyed a lot of Chinese dinners. Thank you, Tamás, Lucia, Niting, Rahim, Niels, Mihai, Dimitra, Riccardo, Otto, Alex (small), Kefeng, Alexey, Paolo, Lipu ... for the fun lunch time. I myself am a silent guy, but I really enjoyed what you *bla bla* during the lunch time. I also enjoyed the sports time for badminton, volleyball, and basketball with you. I wish you all have a bright future.

I started to play basketball when I was in primary school. In Delft, we have a small basketball team: Tao, Ming, Wangwang, Linfeng, Mingxin, Jitang, Yongchang, Song, Dalong, ... In the afternoon, I was delighted when one of us says "Hey, guys, basketball time." During this time, I just enjoyed joking and playing, and left my worries behind.

I am lucky to have many friends (and roommates) in Delft: Meng, Chang, Shuhong, Yihui, Lilan, Yong, Linfeng, Ping and Yan, and Wuyuan. We came to the Netherlands almost at the same time. From those very moments on, we started to live on ourselves, and share the good/bad moments of our life. I will never forget those days in the Professor Street. I would like to thank my roommates: Wangwang, Wenhao, Tiantian, Xi, and Weichen. Wangwang and Wenhao, I still remember the long-trip cycling to Lisse and Castle De Haar. As we have discussed, only the sceneries along the roads can represent the real Netherlands. Tiantian, Xi, and Weichen, we had a lot of discussions on music, movies, food, and material science. I will never forget the memories from our daily life, and I hope that you will have a large number of ACTAs and PRBs. I also want to express my thanks to my friends from NUDT. It is my honour to have all your friendship.

I would like to thank our secretaries and the ICT colleagues: Ilse Oonk, Rina Abbriata, Shemara van der Zwet, Stephen van der Laan, Paulo Anita, and Munire van der Kruyk. You are always helpful when I have a problem. Thank you, Stephen, for the help on the Internet/machine access. Thank you, Paulo and Munire, for the help when I worked as TA for the IN4049 course. All your help saved me a lot of time.

Last but not least, I want to express my sincerest appreciation to my family. Papa and Mama, thank you for bringing me to this world and teaching me to be a man. I am proud of being your son. Confucius said, 'when your parents are still living and ageing, avoid working in a place far away from home.' I am sorry for staying so far away from home that I cared for you very little. I want to thank my sisters, Jianhong and Nini, who took care of you when I was overseas. I want to thank my wife, Haiye Lu. Thank you for always being there for me, and for your understanding and tolerance. I love you!

Jianbin Fang
Delft, September 2014

CONTENTS

1 Introduction **1**
- 1.1 Multi-/Many-Core Processors . . . 2
- 1.2 Processing Cores . . . 2
- 1.3 Memory Hierarchy and Local Memory . . . 3
- 1.4 Programming Models . . . 4
- 1.5 Portability and Performance . . . 4
- 1.6 Research Questions . . . 5
- 1.7 Thesis Contributions . . . 7
- 1.8 Thesis Outline . . . 8

2 OpenCL Against CUDA **11**
- 2.1 Similarities of CUDA and OpenCL . . . 12
- 2.2 Methodology and Experimental Setup . . . 13
  - 2.2.1 Unifying Performance Metrics . . . 13
  - 2.2.2 Selected Benchmarks . . . 14
  - 2.2.3 Experimental Testbeds . . . 14
- 2.3 Performance Comparison and Analysis . . . 14
  - 2.3.1 Comparing Peak Performance . . . 14
  - 2.3.2 Performance Comparison of Real-world Applications . . . 16
  - 2.3.3 A Fair Comparison . . . 21
- 2.4 A Brief Evaluation of OpenCL's Portability . . . 23
- 2.5 Related Work . . . 24
- 2.6 Summary . . . 26

3 Exploring Optimization Space: A Case Study **27**
- 3.1 A First Trial . . . 28
- 3.2 Algorithms and the Representation . . . 29
  - 3.2.1 Aggregation Strategies . . . 29
  - 3.2.2 A Template for Cost Aggregation Kernels . . . 30
- 3.3 Implementations and Optimizations . . . 32
  - 3.3.1 OpenCL Implementations . . . 32
  - 3.3.2 Optimization Steps for CA on GPUs . . . 32
- 3.4 Overall Performance . . . 38
  - 3.4.1 Accuracy . . . 38
  - 3.4.2 Speed on the Quadro5000 . . . 39
  - 3.4.3 Speed on the Low-end GPU . . . 40
  - 3.4.4 Putting it all together . . . 40

3.5 Supplementary Results on a Multi-core CPU . . . 41
3.5.1 Mapping Work-items to Data . . . 42
3.5.2 Using Local Memory . . . 42
3.5.3 Unrolling Loops . . . 43
3.5.4 Increasing Data Parallelism . . . 43
3.6 Related Work . . . 43
3.7 Summary . . . 44

4 Evaluating Vector Data Type Usage 47
4.1 Source-to-Source Translation . . . 48
4.1.1 OpenCL and VDT . . . 48
4.1.2 Using Vector Data Types . . . 49
4.1.3 Code Transformations . . . 50
4.2 Experimental Setup . . . 52
4.2.1 Selected Benchmarks . . . 52
4.2.2 Platforms and Devices . . . 53
4.3 VDT Execution Model . . . 53
4.3.1 Execution Model Analysis . . . 53
4.3.2 Compiler-level Analysis . . . 55
4.3.3 Lessons Learned . . . 55
4.4 Inter-vdt Performance Impact on Macro-Benchmarks . . . 56
4.4.1 Matrix Multiplication . . . 58
4.4.2 Image Convolution . . . 58
4.4.3 Black Scholes . . . 59
4.4.4 SOR . . . 59
4.4.5 Lessons Learned . . . 61
4.5 Intra-vdt Performance Impact on Macro-Benchmarks . . . 62
4.6 Performance Portability Discussion . . . 62
4.7 Related Work . . . 63
4.8 Summary . . . 64

5 Quantifying the Performance Impacts of Using Local Memory 67
5.1 Three Observations as Motivation . . . 69
5.1.1 Data Reuse ≠ Performance Improvement . . . 69
5.1.2 No Data Reuse ≠ Performance Loss . . . 69
5.1.3 Local Memory Use on CPUs ≠ Performance Loss . . . 70
5.2 The Design of Aristotle . . . 70
5.3 MAP Description . . . 71
5.3.1 The Notation . . . 71
5.3.2 eMAP . . . 72
5.3.3 iMAP . . . 73
5.3.4 MAP = eMAP+iMAP . . . 74
5.4 Design Space Exploration and Code Generation . . . 75
5.4.1 Exploring Design Space . . . 75
5.4.2 Code Generator . . . 77

5.5 Performance Database . . . 78
5.5.1 Performance Metric . . . 78
5.5.2 Experimental Setup . . . 78
5.5.3 Performance Optimization Considerations . . . 79
5.5.4 Performance Database . . . 80
5.6 Composing MAP Impacts . . . 84
5.7 Composing Rules Validation . . . 86
5.7.1 A MAP Composer . . . 86
5.7.2 Rule Validation . . . 87
5.7.3 Using Aristotle . . . 87
5.8 Related Work . . . 88
5.9 Summary . . . 88

6 ELMO: An API to Enable Local Memory Usage 91
6.1 ELMO Requirements . . . 92
6.1.1 Challenge I: Geometry Mismatch . . . 93
6.1.2 Challenge II: Work-items Masking and Binding Switches . . . 93
6.1.3 Challenge III: Inefficient Local Memory Organization . . . 93
6.2 ELMO Design . . . 95
6.3 ELMO Implementation . . . 96
6.3.1 BWR . . . 96
6.3.2 COM . . . 98
6.3.3 LMM . . . 98
6.4 Experimental Evaluation . . . 101
6.4.1 Experimental Setup . . . 101
6.4.2 Performance Comparison with Native Kernels . . . 101
6.4.3 Performance Comparison with Hand-tuned Kernels . . . 104
6.5 Discussion . . . 105
6.5.1 Productivity . . . 105
6.5.2 Usability . . . 106
6.5.3 Limitations . . . 106
6.6 Related Work . . . 107
6.7 Summary . . . 108

7 Grover: Reverse-Engineering Local Memory Usage 109
7.1 Motivation . . . 110
7.1.1 Disabling Local Memory Usage . . . 110
7.1.2 Performance Impact . . . 111
7.2 Grover: Systematically Disabling Local Memory Usage . . . 112
7.2.1 Overview . . . 112
7.2.2 The Method behind Grover . . . 113
7.2.3 An Example: Matrix Transpose . . . 115

7.3 Grover Implementation . . . 115
7.3.1 Selecting Candidates . . . 116
7.3.2 Building the Index Expression Trees . . . 116
7.3.3 Determining the Data Index . . . 117
7.3.4 Creating and Solving the Linear System . . . 117
7.3.5 Duplicating the New Load Instructions . . . 118
7.3.6 Updating the New Expression Tree . . . 119
7.4 Experimental Setup . . . 120
7.4.1 Incorporating Grover . . . 120
7.4.2 Selected Benchmarks . . . 120
7.4.3 Platforms and Devices . . . 121
7.5 Performance Evaluation and Discussion . . . 122
7.5.1 Calculating the New Data Index . . . 122
7.5.2 Results Summary . . . 122
7.5.3 Performance Analysis . . . 122
7.5.4 Limitations . . . 124
7.6 Related Work . . . 124
7.7 Summary . . . 126

8 Sesame: Towards a Portable Programming Framework 127
8.1 A Realistic Scenario . . . 127
8.2 The Framework . . . 128
8.3 Sesame Inputs . . . 129
8.3.1 Input Kernels . . . 129
8.3.2 Platform Models . . . 130
8.4 Sesame Implementation . . . 130
8.4.1 Vectorization . . . 130
8.4.2 Local Memory Usage . . . 131
8.5 Related Work . . . 131
8.6 Summary . . . 132

9 Conclusions and Future Work 133
9.1 Conclusions . . . 133
9.2 Future Research Directions . . . 135

A Test-Driving Intel Xeon Phi 137
A.1 Benchmarking Intel Xeon Phi . . . 139
A.1.1 The Architecture . . . 139
A.1.2 Programming . . . 140
A.1.3 MIC-Meter . . . 140
A.2 Empirical Evaluation . . . 141
A.2.1 Vector Processing Cores . . . 141
A.2.2 Memory Latency . . . 143
A.2.3 Memory Bandwidth . . . 145
A.2.4 Ring Interconnect . . . 148
A.2.5 PCIe Data Transfer . . . 150

A.3 SCAT: An Xeon Phi Model . . . 151
A.4 Leukocyte Tracking . . . 153
A.4.1 Performance Analysis . . . 153
A.4.2 Performance Optimization . . . 154
A.4.3 Discussion . . . 156
A.5 Related Work . . . 158
A.6 Summary . . . 158

B Auto-tuning Clustering Data Streams **161**
B.1 Hand-optimizing CDS in OpenCL . . . 162
B.1.1 A Memory-efficient Solution . . . 163
B.1.2 Further Optimizations . . . 164
B.1.3 Experimental Results . . . 165
B.2 Auto-tuning . . . 166
B.2.1 Case: when $a < b$ . . . 168
B.2.2 Case: when $a > b$ . . . 169
B.2.3 Experimental Results . . . 170
B.3 Related Work . . . 170
B.3.1 Clustering Data Streams on GPUs . . . 170
B.3.2 Auto-tuning on GPUs . . . 171
B.4 Summary . . . 172

Bibliography **173**

Summary **187**

Samenvatting **189**

Curriculum Vitæ **191**

1

INTRODUCTION

At the beginning of the 2000s, high performance computing (HPC) was still a niche activity, focused on scientific models for drug discovery or weather prediction, and done almost exclusively in supercomputing labs. Since 2005, when multi-core processors have started to emerge in machines other than the most exclusive supercomputers, the landscape has changed: more and more applications find interesting, new ways to make use of compute power to gain more insight into their respective scientific fields. HPC is also making way in daily life: HPC computer vision algorithms are used to analyze personal photography collections, movies and games become more realistic than ever, and personal genome analysis is reaching the affordability threshold.

These exciting developments are made possible by the fast development of processors and computing in general: limited by the power-, memory-, and parallelism-walls, computing architectures have become parallel, combining multiple cores on the same die. Different solutions have emerged, from homogeneous multi-core CPUs to heterogeneous machines like the Cell/B.E. and massively parallel accelerators, like the GPUs. Despite their different designs, all these processors promise impressive performance and, therefore, significant acceleration of various applications.

With the transition towards parallel hardware, a change in the software was also necessary: sequential, single-threaded applications have suddenly observed low utilization rates and even performance decay. Only parallel applications are able to use these multi-core platforms at their real potential. New parallel applications must be designed and implemented, and existing versions must be updated to this new generation of parallelism. New algorithms, implementations, and optimization strategies are emerging, and together with them arises the issue of productivity: there are not enough expert parallel programmers to address the challenges of this "multi-core revolution". A better-scaling solution is needed to cope with the diversity of the platforms and the large number of applications that require acceleration.

We believe this problem can be tackled by offering more accessible programming tools, featuring a tunable balance between control and transparency, and targeted at non-expert programmers. This thesis shows how such tools can be built and used.

1

1.1. MULTI-/MANY-CORE PROCESSORS

In 2001, IBM released the POWER4, the first general-purpose commercial multi-core processor [156]. Since then, multi-cores have been replacing the traditional single-core processors from personal computers to servers. Manufacturers such as AMD, IBM, Intel, and Sun have developed various multi-core processors, which are built by integrating multiple complex processing cores onto the same die. These cores are interconnected by buses, rings, meshes, or crossbars, and share a memory with multiple levels of cache.

Many-core processors have significantly more cores than multi-core processors, but each core is simpler. In addition, they feature scratch-pad memories, relatively simple caches, and high-speed (graphics) memories. All these features enable many-core processors to excel in data parallel applications and offer impressively high throughputs. A typical example is a GPU (Graphics Processing Unit), which was originally targeted at graphics processing but is now also widely used in general-purpose computing (known as GPGPU) [124].

Multi-cores and many-cores are good at processing different workloads. Multi-cores have been the main workhorse for traditional workloads with moderate parallelism and irregular patterns [89]. Many-cores are particularly good at executing programs with massive (data) parallelism, and regularity in their control flow and memory access patterns. We foresee that multi-cores and many-cores will coexist and be complementary to each other in the future.

1.2. PROCESSING CORES

The number of processing cores on a chip has been increasing over years. In Figure 1.1, we show the number of (single-precision) cores varies over time for AMD Radeon HD GPUs, Intel Xeon processors (and Intel Xeon Phi), and NVIDIA GTX GPUs. Multi-core CPUs and many-core GPUs differ from each other in core structure: multi-cores gain more parallelism by using vector units/SIMD, while many-cores have fine-grain cores that can be further grouped for coarser-grain parallelism. Thus, we count the number of

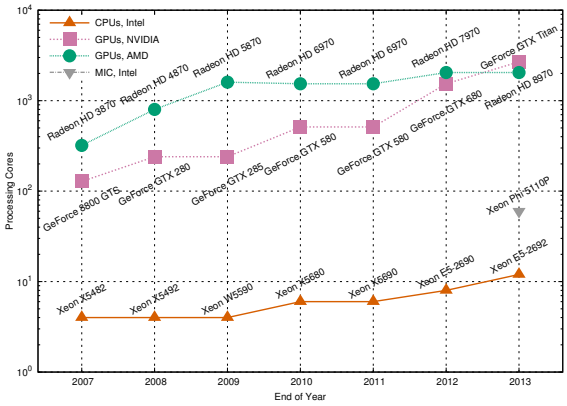


Figure 1.1: Number of processing cores for high-end hardware [78].

cores in different ways: for multi-core CPUs (and Xeon Phi), we regard a vector-core (a SIMD unit) as a single core, while for many-core GPUs, we regard a processing element as a single core. In Figure 1.1, we see that the number of cores is up to a dozen on multi-core processors while it can be hundreds or thousands on many-core processors.

Intel's MIC (Many Integrated Cores), also known as Xeon Phi, integrates around 60 simplified vector cores (512-bit SIMD) on a die, and it is blurring the border between multi-core and many-core processors. An empirical study of this processor is given in Appendix A.

1.3. MEMORY HIERARCHY AND LOCAL MEMORY

As processing cores become faster and more, the performance of many programs is limited by memory accesses. To improve the performance of these memory accesses, a deep(er) memory hierarchy has been introduced, where a small *local memory* has been added, working as a buffer between registers and off-chip memories (Figure 1.2). Due to its on-chip placement, accessing it is much faster than accessing off-chip memories; using this local memory can bring significant performance enhancement [62]. In multi-/many-core processors, there are two types of such memories: caches and scratch-pad memories (SPM).

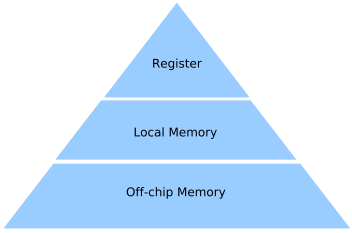


Figure 1.2: Memory Hierarchy.

When using caches, data movement between off-chip memories and caches is managed automatically by hardware protocols (i.e., cache coherency protocols). Caches have been widely implemented in modern multi-core processors. Typically, such processors have up to three levels of cache. The major advantage of adopting caches is that programmers do not have to care about how to move data across memory levels. However, programmers have no direct control of data movements between caches and off-chip memories.

Different from a cache, a SPM has to be managed by software (also known as software-managed cache) [11]. Compared to caches, using SPMs has many advantages. As shown in [11], SPMs consume 40% less energy than caches because of no tag arrays and comparators. With regard to area, a SPM consumes 46% less than a cache of the same size. This is because SPMs use simpler hardware design than caches. Therefore, using SPMs has been very popular in DSPs, game consoles (IBM Cell/B.E.), and graphic processors (GPUs from AMD and NVIDIA). Nevertheless, data movements between off-chip memories and SPMs have to be managed by programmers, which leads to a significant increase

in coding efforts.

The recently released GPUs from both NVIDIA (such as K20) and AMD (such as HD7970) adopt a hybrid solution, by providing both caches and SPMs. In particular, the SPMs from NVIDIA's GPUs are program-configurable, i.e., programmers can specify the ratio of caches to SPMs.

1.4. PROGRAMMING MODELS

Programming models provide developers with an interface to access a machine. The rise of multi-/many-core processors has lead to a plethora of new programming models. According to their abstraction level, we roughly divide programming models into two groups: low-level and high-level.

The low-level programming models such as OpenCL [151], CUDA [115], and DirectCompute [100], require explicit specification of parallelism and provide users with a lot of control over the machine. OpenCL, a unified model managed by the Khronos Group, is a portable framework that supports NVIDIA GPUs, AMD GPUs, multicore CPUs, Intel Xeon Phi, DSPs, and FPGAs. By comparison, CUDA is an NVIDIA-specific programming model for only NVIDIA's GPUs and DirectCompute is only available in Windows. However, just like OpenCL, they require the software developer to explicitly orchestrate data movement, select where variables live in the memory hierarchy, and manually express parallelism in the code. On top of these low-level programming models, C++ Accelerated Massive Parallelism (C++ AMP) [99] and SYCL [81] have been proposed to exploit the flexibility of C++ and to ease programming by using a higher-level abstraction layer.

The high-level programming models such as OpenMP [121], OpenACC [119], and OmpSs [22], use directives (annotations) and library routines to guide compilers to parallelize applications. OpenACC uses a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators. The idea is similar to how OpenMP can be used to parallelize CPU programs. Likewise, OmpSs is an effort to integrate features from the StarSs programming model into a single programming model, based on extending OpenMP with new directives to support asynchronous parallelism and heterogeneity. Therefore, the high-level programming models hide many of the parallelization decisions, and the overall performance heavily relies on dedicated compilers.

1.5. PORTABILITY AND PERFORMANCE

In the Cambridge Dictionary [1], 'portability' is defined as: (1) 'the ability of be easily carried', and (2) 'the ability to be used for a different purpose or on a different system'. In this thesis, we focus on the latter definition, and emphasize 'functional portability'. When an application is functionally portable, it can be compiled, run, and verified in multiple environments without any modification. Therefore, a portable implementation of an application will save time and lower the development/debugging cost. Portability is particularly desirable in the multi-/many-core era when we have a large number of plat-

[1] http://dictionary.cambridge.org/

forms and (vendor-specific) programming models. For example, a code written in CUDA for NVIDIA GPUs can neither run on AMD GPUs, nor run on multi-core CPUs.

Functional portability on multi-/many-core processors can be achieved by using a unified programming model: programmers code the application once, and, with the help of compilers, the code can run on various platforms without modifications.

Before discussing performance, we define the following terms:

- ***Domain experts and Programming experts***
  Domain experts master domain-specific knowledge, but have very basic knowledge about programming. Programming experts master programming models and know architectural details.
- ***Platform-agnostic implementation and Platform-specific implementation***
  We consider any input kernel, as given by users (typically domain experts), to be platform-agnostic: users develop kernels for a virtual platform model (e.g., the OpenCL platform model), and apply certain kernel optimizations (e.g., using vector data types or enabling the use of local memory), without assuming prior knowledge of the target platform. By contrast, a platform-specific kernel would be specialized, by using the right mix of optimizations for the real hardware.

Figure 1.3 shows a typical scenario for achieving high performance with a unified programming model. *Domain experts* write a functionally portable implementation which is assumed to be platform-agnostic. Thereafter, *programming experts* perform a set of optimizations to transform the basic platform-agnostic implementation into a highly optimized, platform-specific version. This development mode heavily relies on programming experts (shown in Figure 1.3 : ❶).

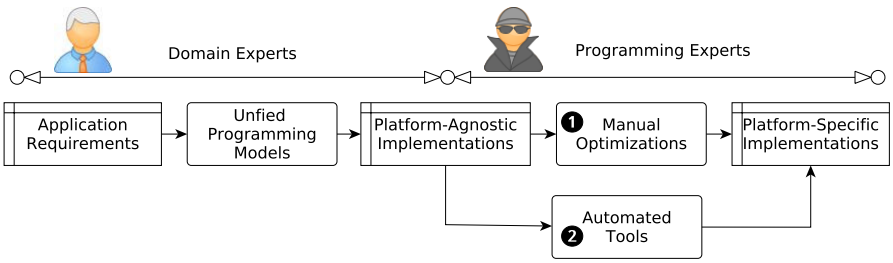


Figure 1.3: A scenario: towards achieving high performance.

1.6. RESEARCH QUESTIONS

Currently, unified programming models cannot systematically deal with platform-specific optimizations: what works for one platform brings performance penalties on another, severely limiting any hope for performance portability. The goal of this thesis is to address this problem at the compiler/API level, by offering domain experts (semi-)automated tools to include/exclude platform-specific optimizations when needed (shown in Figure 1.3 : ❷). Ultimately, we aim to enable unified programming models to cope with

any optimizations/choices the domain experts make without performance penalties; effectively, this means we aim to provide tools to transform any platform-specific optimization into a platform-agnostic one. Therefore, we focus on the following research questions.

RQ1: Is OpenCL a suitable unified programming model for multi-/many-core processors regarding performance and portability?

We need to select a unified programming model as our research vehicle. Among the aforementioned programming models, OpenCL is one proposed to program across platforms. To verify whether OpenCL can achieve matching performance (to native programming models) and ensure functional portability, we present an empirical evaluation of OpenCL versus CUDA by using various benchmarks. Further, we make an extensive analysis of the performance gaps between them (if any). We also give a brief validation of OpenCL's portability on different devices.

RQ2: Is there a platform-specific optimization space for a given kernel?

Given an OpenCL kernel, we explore the interactions between the kernel and the underlying hardware, aiming to achieve optimal performance. Using a case study in computer vision, we explore its optimizations and further analyze the interactions between optimizations and architecture features for different platforms. Based on this analysis, we define an optimization space for the given kernel and quantify how platform-specific they are.

We further focus on two concrete platform-specific optimizations: *vectorization* and *local memory usage*.

RQ3: Can vectorization be a platform-agnostic optimization?

Vector-core processors and scalar-core processors diverge in core organization, and thus vectorization is a platform-specific optimization. To verify whether vectorization can be platform-agnostic, we propose two approaches to vectorize OpenCL kernels, and we measure the performance impact of vectorization on both vector-core processors and scalar-core processors. Given the performance impact vectorization has, we further present suitable options for integrating it in an unified programming model.

RQ4: Can using local memory be a platform-agnostic optimization?

Using local memory is yet another platform-dependent optimization. Because the usage of local memory is a more complex, application-dependent optimization, we split this question into three sub-questions.

RQ4a: When is using local memory beneficial?

Properly using local memory plays a key role in improving performance. However, due to the mixed design of caches and SPMs, local memory is implemented in different manners on different platforms, using local memory leads to unpredictable performance. To investigate when using local memory gives a positive impact and when using it gives a negative impact, we propose a micro-benchmark-based approach which generates a performance database. By querying the database, one can retrieve the benefits of using local memory.

RQ4b: How can we enable local memory usage with high productivity?

Once getting a performance indicator of using local memory, we need to either enable local memory usage or disable local memory usage (i.e., code specialization). Enabling local memory is time-consuming and error-prone. To show how we can facilitate this process, we propose an API-based approach to help programmers enable local memory usage. We investigate how to design such an API to improve productivity while preserving performance.

RQ4c: How can we disable local memory automatically?

Similarly, removing local memory usage is time-consuming and error-prone. This is particularly true in a complex program context and/or when the code is written by a third party. By automatically removing the negative effects of using local memory when it would be detrimental to performance, using local memory becomes a platform-agnostic feature in OpenCL.

RQ5: Can the optimization space for an application be explored systematically?

Beyond the usage of vector types and local memory, many other architectural features must be considered. Therefore, we attempt a generalization of the work we did for local memory and vectorization, and propose the Sesame framework, which compiles and executes the best platform-specific form of a given kernel, aiming to achieve the best performance for the given implementation.

1.7. THESIS CONTRIBUTIONS

In this thesis, we aim to tackle platform-specific optimizations by investigating (semi-) automated methods and techniques. During this process, we make the following contributions.

Contribution 1: We show that OpenCL, as a unified programming model, is a promising alternative to native programming models regarding performance and portability.

In Chapter 2, we see that OpenCL can achieve matching performance to CUDA on NVIDIA GPUs with both synthetic benchmarks and real-world benchmarks. This is further confirmed by our case study in Appendix B. We also see that functional portability is largely achieved by OpenCL on various devices.

Contribution 2: We bring empirical evidence that non-algorithmic kernel optimizations are functionally portable, but have platform-specific performance impacts.

In Chapter 3, we implement, parallelize, and optimize stereo matching on GPUs and CPUs. Our experience and analysis show that there is a platform-specific optimization space (such as using local memory and coalescing memory accesses) for this application and we need to investigate the interactions between platforms and the application for improved performance. In Chapter 4, we observe that using vector types can lead to a better or a worse performance.

Contribution 3: We show the impact of vectorization on different architectures and demonstrate general approaches to achieve it.

In Chapter 4, we use micro-benchmarks and macro-benchmarks to evaluate the performance impact of explicitly using vector data types. Our results show that explicit vectorization plays a key role in achieving high performance on vector-core processors

while it might degrade performance on scalar-core processors. We propose a solution to preserve performance across platforms.

Contribution 4: We show that the benefits of using local memory vary over devices and introducing caches leads to unpredictable performance.

In Chapter 5, we evaluate micro-benchmarks (with and without local memory) on a large range of devices. We find that the benefits of using local memory vary over devices, because the used devices have different memory hierarchies. We also find that the overall performance of using local memory is often unpredictable in the presence of caches.

Contribution 5: We propose a query-based approach to indicate performance gain/loss of using local memory.

In Chapter 5, we develop a suite of micro-benchmarks to evaluate the performance impact of using local memory. Our micro-benchmarks are based on memory access patterns, which makes our approach application-agnostic. Evaluating these micro-benchmarks gives us a performance database. A query in the database indicates whether it is beneficial to use local memory.

Contribution 6: We propose an efficient approach to enable local memory usage.

In Chapter 6, we propose an API to use local memory. When designing the API, we consider performance, productivity, and usability. This API summarizes three typical patterns of using local memory. We implement them in a back-end specifically optimized for GPUs. Our results and analysis show that we achieve improved productivity while preserving high performance.

Contribution 7: We propose an automated compiler-based approach to disable local memory usage.

In Chapter 7, we propose an approach to remove local memory usage. Starting from kernels with local memory, our approach can remove the usage of local memory automatically. This approach is based on building the correspondence between local memory accesses and global memory accesses. We have implemented our approach as a compiling pass, which aims to fully free programmers from removing local memory usage by hand.

Contribution 8: We have designed a framework to tackle platform-specific optimizations systematically.

Beyond vectorization and local memory usage, there are other platform-specific optimizations. Therefore, we propose a portable programming framework, aiming to address all the platform-specific optimizations in Chapter 8.

1.8. THESIS OUTLINE

The organization of this thesis is presented in Figure 1.4, and described in the following.

Chapter 2 presents a comprehensive performance comparison between CUDA and OpenCL. We make an extensive analysis of the performance gaps taking into account programming models, optimization strategies, architectural details, and underlying compilers. Our results show that, for most applications, OpenCL can achieve matching per-

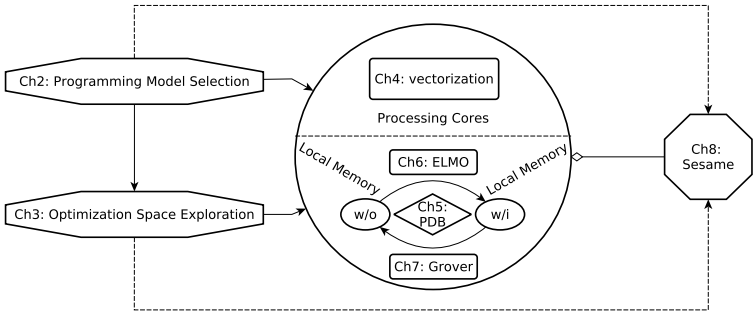


Figure 1.4: Thesis organization.

formance to CUDA. We also investigate OpenCL's functional portability. This chapter is based on our work previously published in ICPP'11 [49].

Chapter 3 takes real-time stereo matching as an example, and presents a generic representation and suitable implementations for three commonly used cost aggregators. We show how to parallelize and optimize these kernels, which leads to a significant performance improvement. Further, we evaluate the optimizations on a multi-core CPU, and relate each optimization to architecture features. This chapter is based on our work previously published in ICPADS'12 [45].

Chapter 4 investigates the usage of vector data types in a systematic way. First, we propose two different approaches to enable vector data types in OpenCL kernels. After obtaining vectorized code, we further evaluate the performance effects with benchmarks. With microbenchmarks, we study the execution model of vector data types and the role of the compiler-aided vectorizer, on a range of processors. With macro-benchmarks, we explore the performance impact from application characteristics. Further, we discuss how to deal with performance portability in the presence of vector data types. This chapter is based on our work published in Concurrency and Computation: Practice and Experience [47].

Chapter 5 quantifies the performance impact of using local memory in multi/many-core processors. To do so, we systematically describe memory access patterns (MAPs) in an application-agnostic manner. Next, for each identified MAP, we generate two microbenchmarks: one without local memory and the other one with local memory. We further evaluate both of them on typically used platforms, and we log their performance. What we eventually obtain is a local memory performance database, indexed by various MAPs and platforms. Given an application, its MAPs, and a platform, a query in the database can indicate the performance impact of using local memory. This chapter is based on our work previously published in MuCoCoS'13 [40] and in Scientific Programming [42].

Chapter 6 introduces an easy-to-use API (ELMO) that improves productivity while preserving the high performance of local memory operations. Specifically, ELMO is a generic API that covers different local memory use-cases. We present prototype implementations for these APIs and perform multiple GPU-inspired optimizations. Experimental results show that using ELMO we can achieve performance comparable with that

of hand-tuned applications, while the code is shorter, clearer, and safer. This chapter is based on our work previously published in PDP'13 [44].

Chapter 7 presents Grover, a method to automatically remove local memory usage from OpenCL kernels. In particular, we create a correspondence between the global and local memory spaces, which is used to replace local memory accesses by global memory accesses. We have validated Grover and found that it can successfully disable local memory usage. We have observed performance improvements for more than a third of the test cases after Grover disabled local memory usage. This chapter is based on our work previously published in ICPP'14 [41].

Chapter 8 presents a portable programming framework for parallel applications running on many-core processors (Sesame). Taking a platform-agnostic code provided by a domain expert as input, Sesame chooses and includes/excludes the most suitable architecture-specific optimizations, aiming to improve the overall application performance in a user-transparent way. This chapter is based on our work previously published in CCGrid'13 [46].

In Chapter 9, we summarize our key findings and present future directions.

Appendix A introduces our experience on Intel Xeon Phi at two different levels: the micro-benchmark level, and the application level. At the micro-benchmarking level, we show the high performance of five components of the architecture, focusing on their maximum achieved performance and the prerequisites to achieve it. Next, we choose a medical imaging application as a case study. We observed that it is rather easy to get functional code and start benchmarking, but the first performance numbers are not satisfying. This appendix is based on our work previously published in ICPE'14 [43].

Appendix B provides an OpenCL implementation for clustering data streams, and then presents several optimizations for it, to make it more efficient in terms of memory usage. To maximize performance for different problem sizes and architectures, we also propose an auto-tuning solution. Experimental results show that our fully optimized implementation can perform significantly faster than the native OpenCL implementation; it can also achieve better performance than the original solution. This appendix is based on our work previously published in CSE'11 [48].

2

OpenCL Against CUDA

In this chapter, we investigate whether the performance of OpenCL is compromised due to its cross-platform promise. We compare the performance of OpenCL against CUDA on NVIDIA GPUs with diverse applications. Further, we briefly discuss OpenCL's portability on a range of devices.

Today's GPUs (Graphic Processing Units), greatly outperforming CPUs in arithmetic throughput and memory bandwidth, can use hundreds of parallel processor cores to execute tens of thousands of parallel threads [53, 107]. Researchers and developers are becoming increasingly interested in harnessing this power for general-purpose computing, an effort known collectively as GPGPU (for "General-Purpose computing on the GPU") [123], to rapidly solve large problems with substantial inherent parallelism.

Due to this large performance potential, GPU programming models have evolved from shading languages such as Cg [116], HLSL [101], and GLSL [80] to modern programming languages, alleviating programmers' burden and thus enabling GPUs to gain more popularity. Particularly, the release of CUDA (Compute Unified Device Architecture) by NVIDIA in 2006 has eliminated the need of using the graphics APIs for computing applications, pushing GPU computing to more extensive use [115]. Likewise, APP (Advanced Parallel Processing) is a programming framework which enables ATI/AMD GPUs, working together with the CPUs, to accelerate many applications beyond just graphics [4]. All these programming frameworks allow programmers to develop a GPU application without mastering graphic terms, and enables them to build large applications easier [124].

However, every programming framework has its unique method for application development. This can be inconvenient, because software development and related services must be rebuilt from scratch every time a new platform hits the market [153]. The software developers were forced to learn new APIs and languages which quickly became out-of-date. Naturally, this caused a rise in demand for a single language capable of handling any architecture. Finally, an open standard was established, now known as "OpenCL" (Open Computing Language). OpenCL can give software developers portable

This chapter is based on our work published in the Proceedings of ICPP 2011 [49].

and efficient access to the power of diverse processing platforms. Nevertheless, this also brings up the question of whether the performance is compromised, as it is often the case for this type of common languages and middle-wares [153]. If the performance suffers significantly when using OpenCL, its usability becomes debatable (users may not want to sacrifice the performance for portability).

To investigate the performance-vs-portability trade-offs of OpenCL, we make extensive investigations and experiments with diverse applications ranging from synthetic ones to real-world ones, and we observe the performance differences between CUDA and OpenCL. In particular, we give a detailed analysis of the performance differences and then conclude that under a fair comparison, the two programming models are equivalent, i.e., there is no fundamental reason for OpenCL to perform worse than CUDA.

We focus on exploring the performance comparison of CUDA and OpenCL on NVIDIA's GPUs because, in our view, this is the most relevant comparison. First, for alternative hardware platforms it is difficult to find comparable models: on ATI/AMD GPUs, OpenCL has become the "native" programming model, so there is nothing to compare against; on the Cell/B.E., OpenCL is still immature and a comparison against the 5-year old IBM SDK would be unfair "by design"; on the general purpose multi-core processors, we did not find a similar model (i.e., a model with similar low level granularity) to compare against. Second, CUDA and OpenCL, which are both gaining more and more attention from both researchers and practitioners, are similar to each other in many aspects.

The rest of this chapter is organized as follows: Section 2.1 compares CUDA and OpenCL at the conceptual level. Section 2.2 illustrates our methodology, the selected benchmarks and the testbeds. Section 2.3 gives an overall performance comparison and identifies the main reasons for the performance differences. Then we define a fair comparison for potential performance comparisons and analyses of CUDA and OpenCL. Section 2.4 shows OpenCL's ability in code-portability. Section 2.5 presents some related work on performance comparison of parallel programming models on multi-/many-cores. Section 2.6 summarizes this chapter.

2.1. Similarities of CUDA and OpenCL

CUDA is a parallel computing framework designed only for NVIDIA's GPUs, while OpenCL is a standard designed for diverse platforms including CUDA-enabled GPUs, some ATI/AMD GPUs, multi-core CPUs from Intel and AMD, and other processors such as the Cell/B.E.

OpenCL shares a range of core ideas with CUDA: they have similar platform models, memory models, execution models, and programming models [115, 151]. To a CUDA (or an OpenCL) programmer, the computing system consists of a host (typically a traditional CPU), and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units [82]. There also exists a mapping between CUDA and OpenCL in memory and execution terms, as is presented in Table 2.1. Additionally, their syntax for various keywords and built-in functions are fairly similar to each other. Therefore, it is relatively straightforward to translate CUDA programs to OpenCL programs.

Table 2.1: A comparison of general terms [5]

CUDA terminology	OpenCL terminology
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory
Thread	Work-item
Thread-block	Work-group

Table 2.2: Selected benchmarks

App.	Suite	Dwarf/Class*	Performance Metric	Description
BFS	Rodinia	Graph Traversal	sec	Graph breadth first search
Sobel	SELF	Dense Linear Algebra	sec	Sobel operator on a gray image in X direction
TranP	SELF	Dense Linear Algebra	GB/sec	Matrix transposition with shared memory
Reduce	SHOC	Reduce*	GB/sec	Calculate a reduction of an array
FFT	SHOC	Spectral Methods	GFlops/sec	Fast Fourier Transform
MD	SHOC	N-Body Methods	GFlops/sec	Molecular dynamics
SPMV	SHOC	Sparse Linear Algebra	GFlops/sec	Multiplication of sparse matrix and vector (CSR)
St2D	SHOC	Structured Grids	sec	A two-dimensional nine point stencil calculation
DXTC	NSDK	Dense Linear Algebra	MPixels/sec	High quality DXT compression
RdxS	NSDK	Sort*	MElements/sec	Radix sort
Scan	NSDK	Scan*	MElements/sec	Get prefix sum of an array
STNW	NSDK	Sort*	MElements/sec	Use comparator networks to sort an array
MxM	NSDK	Dense Linear Algebra	GFlops/sec	Matrix multiplication
FDTD	NSDK	Structured Grids	MPoints/sec	Finite-difference time-domain method

2.2. METHODOLOGY AND EXPERIMENTAL SETUP

In this section, we explain the methodologies we adopt in this chapter. The used benchmarks and experimental testbeds are also explained.

2.2.1. UNIFYING PERFORMANCE METRICS

In order to compare the performance of CUDA and OpenCL, we define a normalized performance metric, called *Performance Ratio(PR)*, as follows:

$$PR = \frac{Performance_{OpenCL}}{Performance_{CUDA}} \tag{2.1}$$

For $PR < 1$, the performance of OpenCL is worse than its counter-part; otherwise, OpenCL will give a better or the same performance. In an intuitive way, if $|1 - PR| < 0.1$, we assume CUDA and OpenCL have similar performance.

When it comes to different domains, performance metrics have different meanings. In memory systems, the bandwidth of memories can be seen as an important performance metric. The higher the bandwidth is, the better the performance is. For sorting algorithms, performance may refer to the number of elements a processor finishes sorting in unit time. Floating-point operations per second (Flops/sec) is a typical performance metric in scientific computing. Exceptionally, performance is inversely proportional to

the time a benchmark that takes from start to end. Therefore, we have selected specific performance metrics for different benchmarks, as illustrated in Table 2.2.

2.2.2. Selected Benchmarks

Benchmarks are selected from the SHOC benchmark suite, NVIDIA's SDK, and the Rodinia benchmark suite [1]. We also use some self-designed applications. These benchmarks fall into two categories: synthetic applications and real-world applications.

Synthetic Applications

Synthetic applications are those which provide ideal instructions to make full use of the underlying hardware. We select two synthetic applications from the SHOC benchmark suite: `MaxFlops` and `DeviceMemory`, which are used to measure peak performance (floating-point operations and device-memory bandwidth) of GPUs in GFlops/sec and GB/sec. In this chapter, peak performance includes theoretical peak performance and achieved peak performance. Theoretical performance can be calculated using hardware specifications, while achieved performance is measured by running synthetic applications on real hardware.

Real-world Applications

Such applications include algorithms frequently used in real-world domains. The real-world applications we select are listed in Table 2.2. Among them, `Sobel`, `TranP` in both CUDA and OpenCL, and `BFS` in OpenCL are developed by ourselves (denoted by "SELF"); others are selected from the SHOC benchmarks suite ("SHOC"), NVIDIA's CUDA SDK ("NSDK") and the Rodinia benchmark suite (only `BFS` in CUDA, denoted by "Rodinia"). Following the guidelines of the 7+ Dwarfs [8], different applications fall into different categories. Their performance metrics and descriptions are also listed in the table.

2.2.3. Experimental Testbeds

We obtain all our measurement results on real hardware using three platforms, called Dutijc, Saturn, and Jupiter. Each platform consists of two parts: the host machine (one CPU) and its device part (one or more GPUs). Table 2.3 shows the detailed configurations of these three platforms. A short comparison of the three GPUs we have used (NVIDIA GTX280, NVIDIA GTX480, and ATI Radeon HD5870) is presented in Table 2.4 (*MIW* there stands for Memory Interface Width). Intel(R) Core(TM) i7 CPU 920@2.67GHz (or Intel920) and Cell Broadband Engine (or Cell/BE) are also used as OpenCL devices. For the Cell/B.E., we use the OpenCL implementation from IBM. For the Intel920, we use the implementation from AMD (APP v2.2), because Intel's implementation on Linux was still unavailable at the moment of writing (March 2011).

2.3. Performance Comparison and Analysis

2.3.1. Comparing Peak Performance

Bandwidth of Device Memory

TP_{BW} (Theoretical Peak Bandwidth) is given as follows:

$$TP_{BW} = MC \times (MIW/8) \times 2 \times 10^{-9} \tag{2.2}$$

Table 2.3: Details of underlying platforms

	Saturn	Dutijc	Jupiter
Host CPU	Intel(R) Core(TM) i7 CPU 920@2.67GHz		
Attached GPUs	GTX480	GTX280	Radeon HD5870
gcc version	4.4.1	4.4.3	4.4.1
CUDA version	3.2	3.2	—
APP version	—	—	2.2

Table 2.4: Specifications of GPUs

	GTX480	GTX280	HD5870
Architecture	Fermi	GTX200s	Cypress
#Compute Unit	60	30	20
#Cores	480	240	320
#Processing Elements	—	—	1600
Core Clock(MHz)	1401	1296	850
Memory Clock(MHz)	1848	1107	1200
MIW(bits)	384	512	256
Memory Capacity(GB)	GDDR5 1.5	GDDR3 1	GDDR5 1

where MC is the abbreviation for Memory Clock, and MIW is short for Memory Interface Width. Using Equation 2.2 we calculate TP_{BW} of GTX280 and GTX480 to be 141.7 GB/sec and 177.4 GB/sec, respectively.

AP_{BW} (Achieved Peak Bandwidth) is measured by reading global-memory in a coalesced manner. Moreover, our experimental results show that AP_{BW} depends on work-group size (or block size), which we set to 256. The results of the experiments with `DeviceMemory` on Saturn (GTX480) and Dutijc (GTX280) are shown in Figure 2.1a. We see that OpenCL outperforms CUDA in AP_{BW} by 8.5% on GTX280 and 2.4% on GTX480. Further, the OpenCL implementation achieves 68.6% and 87.7% of TP_{BW} on GTX280 and GTX480, respectively.

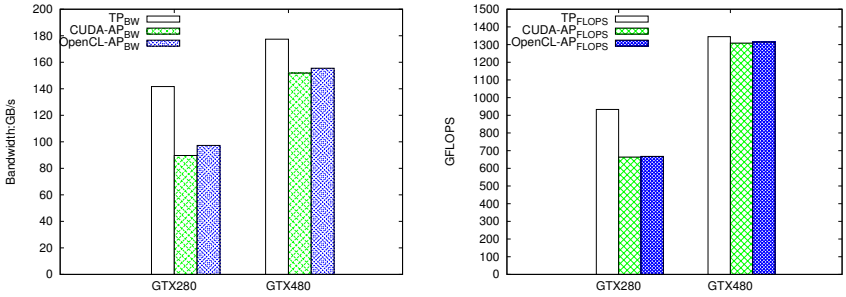
FLOATING-POINT PERFORMANCE

TP_{FLOPS} (Theoretical Peak Floating-Point Operations per Second) is calculated as follows:

$$TP_{FLOPS} = CC \times \#Cores \times R \times 10^{-9} \tag{2.3}$$

where CC is short for Core Clock and R stands for maximum operations finished by a scalar core in one cycle. R differs depending on the platforms: it is 3 for GTX280 and 2 for GTX480, due to the dual-issue design of the GT200 architecture. As a result, TP_{FLOPS} is equal to 933.12 GFlops/sec and 1344.96 GFlops/sec for these two GPUs, respectively.

AP_{FLOPS} (Achieved Peak FLOPS) in `MaxFlops` is measured in different ways on GTX280 and GTX480. For GTX280, a `mul` instruction and a `mad` instruction appear in an interleaved way (in theory they can run on one scalar core simultaneously), while only `mad` instructions are issued for GTX480. The experimental results are compared in Figure 2.1b. We see that OpenCL obtains almost the same AP_{FLOPS} as CUDA for GTX280 and GTX480, accounting for approximately 71.5% and 97.7% of the corresponding TP_{FLOPS}.



(a) Bandwidth (b) Flops

Figure 2.1: A comparison of the peak bandwidth and FLOPS for GTX280 and GTX480.

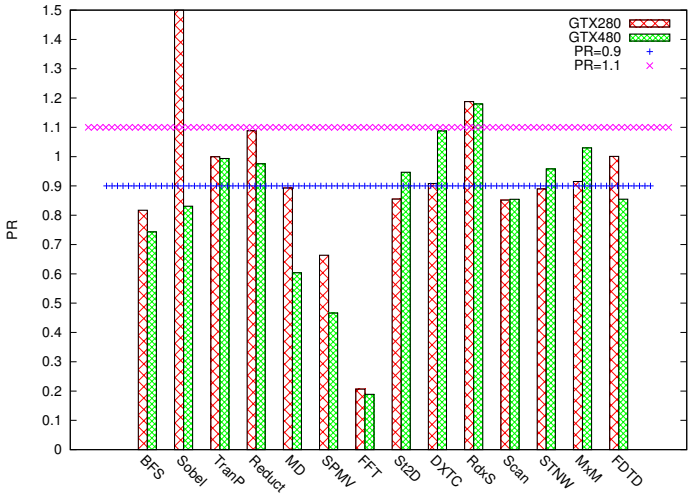


Figure 2.2: A performance comparison of selected benchmarks. When the top border of a rectangle lies in the area between Line $\{PR = 0.9\}$ and Line $\{PR = 1.1\}$, we assume CUDA and OpenCL have similar performance. (Note that on GTX280, the PR for Sobel is 3.2)

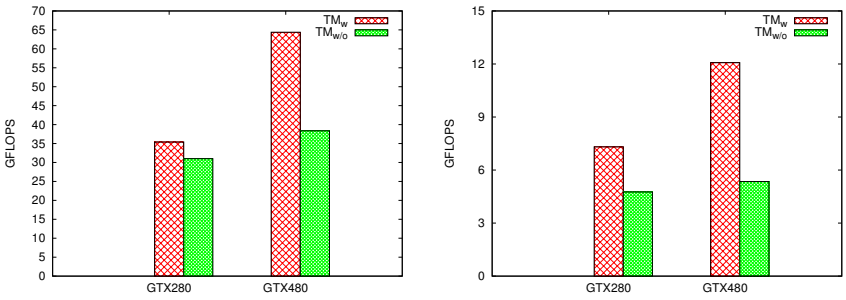
Thus, CUDA and OpenCL are able to achieve similar peak performance (to be precise, OpenCL even performs slightly better), which shows that OpenCL has the same potential to use the underlying hardware as CUDA.

2.3.2. Performance Comparison of Real-world Applications

The real-world applications mentioned in Section 2.2.2 are selected to compare the performance of CUDA and OpenCL. The PR of all the real-world applications without any modifications is shown in Figure 2.2. As can be seen from the figure, PR varies a lot when using different benchmarks and underlying GPUs. We analyze these performance differences in the following.

PROGRAMMING MODEL DIFFERENCES

As is shown in Section 2.1, CUDA and OpenCL have many conceptual similarities. However, there are also several differences in programming models between CUDA and OpenCL. For example, *NDRange* in OpenCL represents the number of work-items in the whole problem domain, while *GridDim* in CUDA is the number of blocks.



(a) MD Benchmark (b) SPMV Benchmark

Figure 2.3: Performance impact of texture memory

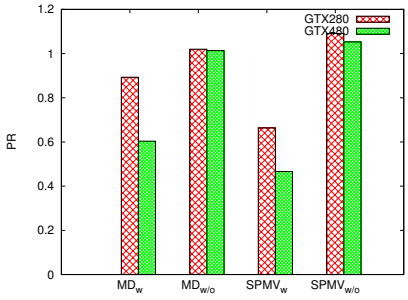


Figure 2.4: Performance ratio before and after removing texture memory

Additionally, they have different abstractions of device memory hierarchy, where CUDA explicitly supports specific hardware features which OpenCL avoids for portability reasons. Through analyzing kernel codes, we find that texture memory is used in the CUDA implementations of MD and SPMV. Both benchmarks have intensive and irregular access to a read-only global vector, which is stored in the texture memory space. Figure 2.3 shows the performance of the two applications when running with and without the usage of texture memory. As can be seen from the figure, after the removal of the texture memory, the performance drops to about 87.6%, 65.1% on GTX280 and 59.6%, 44.3% on GTX480 of the performance with texture memory for MD and SPMV, respectively. We compare the performance of OpenCL and CUDA after removing the usage of texture memory. The results of this comparison are presented in Figure 2.4, showing

similar performance between CUDA and OpenCL [1]. It is the special support of texture cache that makes the irregular access look more regular. Consequently, texture memory plays an important role in performance improvement of kernel programs.

Different Optimizations on Native Kernels

In [113], many optimization strategies are listed: (i) ensure global memory accesses are coalesced whenever possible; (ii) prefer shared memory access wherever possible; (iii) use shift operations to avoid expensive division and modulo calculations; (iv) make it easy for the compiler to use branch prediction instead of loops, etc.

One of the important optimizations to be performed in kernel codes is to reduce the number of dynamic instructions in the run-time execution. Loop unrolling is one of the techniques that reduces loop overhead and increases the computation per loop iteration [14]. NVIDIA's CUDA provides an interface to unroll a loop fully or partially using the pragma `unroll`. When analyzing the native kernel codes of FDTD (as is illustrated in the following list), we find these two codes are the same except that the CUDA code uses the pragma `unroll` at both unroll points a and b, while the OpenCL one unrolls the loop only at point b.

```
1  // Code segment of FDTD kernel
2  // Step through the xy-planes
3  #pragma unroll 9   //unroll point: a
4  for (int iz=0; iz<dimz; iz++){
5    // some work here
6  #pragma unroll RADIUS   //unroll point: b
7    for (int i=1; i<=RADIUS; i++){
8      // some work here
9    }
10   // some work here
11 }
```

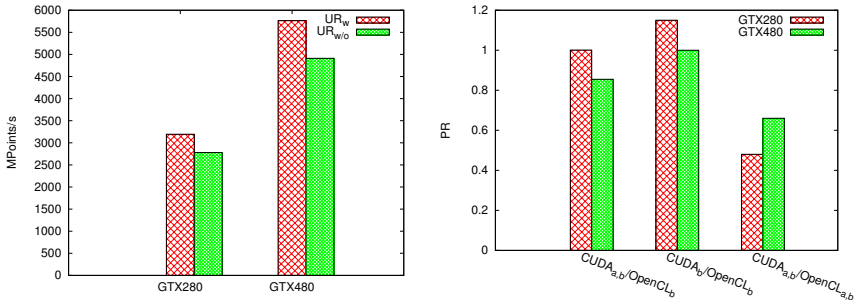
Figure 2.5: FDTD code.

The performance of the application (in CUDA only) with and without the pragma `unroll` at point a is shown in Figure 2.6a. We can see that the performance without the pragma `unroll` drops to 85.1% and 82.6% of the performance with it for GTX280 and GTX480. We then remove the pragma at point a from the CUDA version and present a performance comparison between CUDA and OpenCL in Figure 2.6b. It can be seen that they achieve similar performance on GTX480, while OpenCL outperforms CUDA by 15.1% on GTX280. Moreover, we observe that when adding the pragma `unroll` at unroll point a of the OpenCL implementation, the performance degrades sharply to 48.3% and 66.1% of that of the CUDA implementation for GTX280 and GTX480, also shown in Figure 2.6b.

Architecture-related Differences

Since the birth of the original G80, the Fermi architecture can be seen as the most remarkable leap forward for GPGPU computing. It differs from the previous generations

[1] Alternatively, we can use *Image Objects* to exploit texture memory in OpenCL.



(a) (b)

Figure 2.6: (a) Performance impact of loop-unrolling (CUDA only), (b) A performance comparison of FDTD with/without loop-unrolling at different points ($CUDA_x$ represents we execute loop-unrolling at point x, and it is the same for OpenCL. For example, the third group $CUDA_{a,b}/OpenCL_{a,b}$ represents we unroll the loop at both points for CUDA and OpenCL).

by, e.g, (i) improved double precision performance; (ii) ECC support; (iii) true cache hierarchy; (iv) faster context switching [112].

The introduction of the cache hierarchy has a significant impact on Fermi's performance. When looking at Figure 2.2, we see that the values diverge remarkably for Sobel on GTX280 and GTX480. On GTX280, the OpenCL version runs three times faster than the CUDA one, but it only obtains 83% of CUDA's performance when the benchmark runs on GTX480. These differences are caused by the constant memory and the cache. In the implementation with OpenCL, constant memory is employed to store the "filter" in Sobel, while it is not in the CUDA version.

After removing the usage of constant memory, we do the same experiments on these two GPUs. The execution time is presented in Figure 2.7. On the one hand, we see the kernel execution time drops to one quarter of that without using constant memory on GTX280. On the other hand, there are few changes on GTX480 due to the availability of the global-memory cache in the Fermi architecture. Overall, CUDA and OpenCL achieve similar performance with/without constant memory on GTX480.

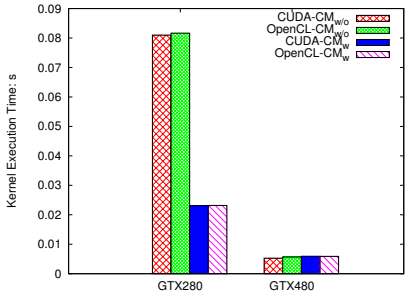


Figure 2.7: A performance comparison for Sobel with and without constant memory on GTX280 and GTX480

Table 2.5: Statistic for PTX instructions

		Instruction Count				Instruction Count	
Class	**Instructions**	**CUDA**	**OpenCL**	**Class**	**Instructions**	**CUDA**	**OpenCL**
Arithmetic	add	93	191	Data Movement	cvt	16	16
	sub	83	95		mov	687	88
	mul	33	138		ld.param	1	1
	div	0	2		ld.local	97	64
	fma	0	37		ld.shared	32	32
	mad	2	22		ld.const	0	24
	neg	9	36		ld.global	8	8
	and	1	291		st.local	250	78
Sub-total		220	521		st.shared	32	32
Logic Shift	or	2	33		st.global	8	8
	not	0	4	Sub-total		1131	351
	xor	0	4	Flow Control	setp	2	80
	shl	0	50		selp	0	40
	shr	1	43		bra	2	68
Sub-total		4	163	Sub-total		4	188
Synchronization	bar	7	7	Total		1366	1230

Compiler and Run-time Differences

Among all the benchmarks, the performance gap between OpenCL and CUDA is the biggest for the FFT. Their native kernel codes are exactly the same. However, when looking into their PTX codes, we find notable differences between them. A quantitative comparison of these two PTX kernels is presented in Table 2.5. The statistics are gathered for the "forward" kernel of the FFT implementation.

From Table 2.5, the differences between these two PTX codes become visible. The OpenCL front-end compiler generates two times more arithmetic instructions than its CUDA counter-part. There are rarely any logic-shift instructions in CUDA, while there are 163 such instructions in the OpenCL kernel. A similar situation happens with the flow-control instructions: there are many more for OpenCL than for CUDA. Although there are many more data-movement instructions for CUDA, most of them are `mov`, simply moving data to or from registers or local memories. Finally, we note that all time-consuming instructions such as `ld.global` and `st.global` are exactly the same.

We can explain this situation by assuming that the front-end compiler for CUDA has been used and optimized more heavily, thus is more mature, than that of OpenCL. As a result, when it comes to some kernels like "forward" in FFT, OpenCL performs worse than CUDA.

BFS is also an interesting example here. It has to invoke the kernel functions several times to solve the whole problem. Thus, the kernel launch time (the time that a kernel takes from entering the command-queue until starting its execution) plays a significant role in the overall performance. Our experimental results show that the kernel launch time of OpenCL is longer than that of CUDA (the gap size depends on the problem size), due to differences in the run-time environment. The longer kernel launch time may also

explain why OpenCL performs worse than CUDA for applications like BFS.

In the previous analysis, we only identify the most influential factor for each application that shows an observable performance difference. It is important to note that several factors may often affect the program performance together, leading to larger performance discrepancies. An analysis of such combinations, as well as the investigation of lower level factors (such as compiler optimizations), is left for future work.

2.3.3. A FAIR COMPARISON

So far, we have shown that the performance gaps between OpenCL and CUDA are due to programming model differences, different optimizations on native kernels, architecture-related differences, and compiler differences. It has been shown that performance can be equalized by systematic code changes. Therefore, we present an eight-step fair comparison approach for CUDA and OpenCL applications from the original problem to its final solution, which provides guidelines for investigating the performance gap between CUDA and OpenCL (if any). A schematic view of this approach is shown in Figure 2.8.

(1) PROBLEM DESCRIPTION

This step describes what the problem is and what form the solutions could be.

(2) ALGORITHM TRANSLATION

How to address the problem is given using certain algorithms. The algorithms can be described in pseudo-code which is environment-independent and easier for humans to understand.

(3) IMPLEMENTATION

In this step, the algorithms mentioned above are implemented with different programming models or languages. As for GPU programs, there are two parts: one is the host program and the other is the kernel code running on GPUs. On NVIDIA GPUs, CUDA+C and OpenCL+C are usually adopted to implement GPU programs. If two implementations use similar APIs to access the same type of hardware resources, we consider these two implementations to be the same. Note that two implementations also have to use the same type of timers to measure performance.

(4) NATIVE KERNEL OPTIMIZATIONS

After implementation, architecture-dependent optimizations on kernel programs are executed. For example, whether to use the shared memory (or local memory in OpenCL), whether to employ vectorization, whether to unroll loops, whether to reduce bank-conflicts, whether to use texture memory in CUDA, and whether to access global memory in a coalesced way, are decisions that should be taken into account. On the one hand, optimizations on native kernels is a time-consuming and error-prone job; on the other hand, it can contribute to performance improvement significantly.

(5) FIRST-STAGE COMPILATION AND OPTIMIZATION

The first-stage compiler adopted in CUDA is called NVOPENCC. There is a similar front-end compiler for OpenCL in this stage. This stage compiles kernel codes into PTX codes,

a low-level parallel thread execution virtual machine and instruction set architecture (ISA) developed by NVIDIA [114]. Some advanced optimizations are also executed in this stage.

(6) Second-Stage Compilation and Optimization

PTXAS (the back-end compiler) translates PTX codes into binary format in this step and it may execute some additional optimizations.

(7) Program Configuration and Start-up

Before executing the program prepared so far, we need to configure two kinds of parameters: (1) problem parameters (the parameters of the problem to be solved such as the size of the matrix), and (2) algorithmic parameters (for example, block-size or work-group size). Although these parameters do not change the correctness of final results, they can have a significant impact on the performance of the application.

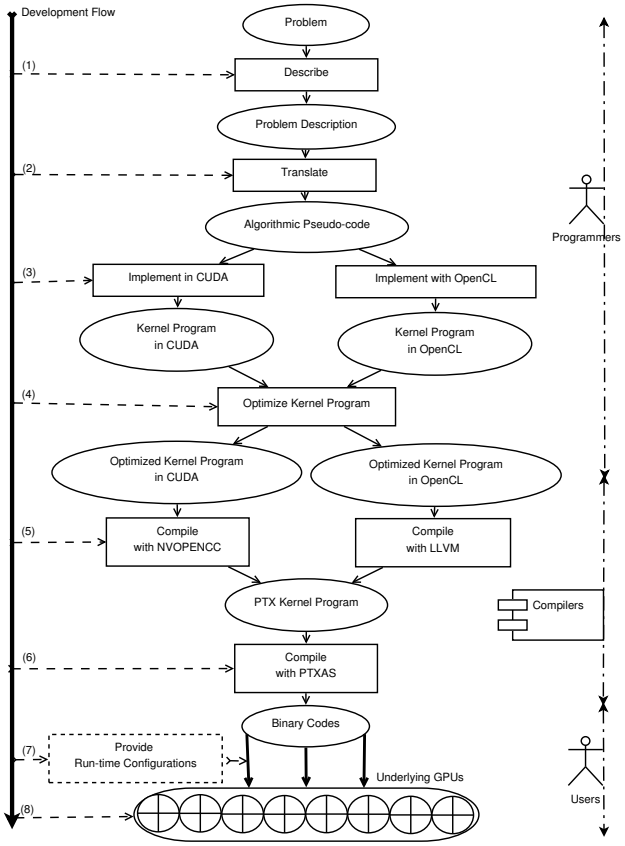


Figure 2.8: Development flow of GPU kernel programs (The ellipses represent entities such as a program or a description and the rectangles represent actions on the entities. We categorize three types of roles participating the whole process: programmers, compilers, and users.)

(8) RUNNING ON GPUS

With the help of drivers, the binary codes are finally scheduled to run on the GPUs.

These eight steps make up the application development flow from an original problem to its final solution. Based on this, we define that a comparison for CUDA and OpenCL is "fair" when configurations in all the eight steps of the comparison are the same. According to the analysis in previous subsection, OpenCL can obtain similar performance to CUDA in the case of "a fair comparison". In the real world, programmers are responsible for steps (1) - (4) and compilers take charge of steps (5), (6). Finally, users will employ the application through steps (7) and (8), as is illustrated in Figure 2.8. Each of the eight steps is probably executed by different programmers (they have different programming habits, abilities and choices) or different compilers (they may execute different optimizations) or different users (they have different requirements and investments). All those lead to the difficulty of making sure that a performance comparison is fair for CUDA and OpenCL.

2.4. A BRIEF EVALUATION OF OPENCL'S PORTABILITY

We have seen so far that for a set of 16 benchmarks, OpenCL implementations differ from the CUDA ones on performance. Given that OpenCL's portability is typically invoked as a good reason for performance drops, we investigate if the portability claim holds by porting all the real-world benchmarks from NVIDIA's GPUs to HD5870, Intel920 and Cell/B.E. All performance data is listed in Table 2.6 (the performance units are the same as those shown in Table 2.2).

Table 2.6: Performance data on prevailing Platforms

	BFS	Sobel	TranP	Reduct	MD	SPMV	FFT	St2D	DXTC	RdxS	Scan	STNW	MxM	FDTD
HD5870	0.0246	0.0048	5.951	114.4	28.60	4.665	36.10	1.666	14.50	FL	177.6	42.43	205.3	3352
Intel920	0.1455	0.1553	2.411	0.9936	2.597	3.805	1.424	238.4	14.48	FL	1.071	0.7605	0.8857	3787
Cell/BE	1.159	5.425	0.1993	0.0528	0.1264	0.0809	ABT	0.1178	ABT	ABT	1.620	ABT	1.473	19.15

When comparing performance on NVIDIA's GPUs, we find that most benchmarks, without additional optimizations on HD5870, achieve comparable performance with that on GTX280. An exceptional example is TranP on HD5870 which performs much worse than it does on GTX280.

When benchmarks run on Intel920 and Cell/BE, we have to make some minor modifications of changing CL_DEVICE_TYPE_GPU to CL_DEVICE_TYPE_CPU or to CL_DEVICE_TYPE_ACCELERATOR for benchmarks selected from the CUDA SDK. Moreover, there are more programming constraints on Cell/BE (e.g. `get_local_id` or `cosine` functions are not allowed within inline definition of another function). When it comes to performance on Intel920, we observe that the bandwidth of TranP drops from 2.411 GB/sec to 0.2150 GB/sec because of using local memory: all OpenCL memory objects for CPU are cached implicitly by hardware and thus explicitly using local memory just introduces unnecessary overhead [40–42]. Another interesting observation is that SPMV sees a performance degradation from 3.805 GFlops/sec to 0.1247 GFlops/sec when employing warp-oriented optimization (using a warp of threads to work together on one matrix row). We believe this happens because there are orders of magnitude fewer processing

cores in CPUs than in GPUs.

In Table 2.6, "ABT" means the programs (FFT, DXTC, RdxS, and SNTW) exit, showing "aborted". It is mainly because there are not enough resources on the Cell/B.E. For example, DXTC shows "CL_OUT_OF_RESOURCES" when invoking the kernel because of insufficient registers or local memories. The possible solution we can imagine is to make the input problem size smaller.

We also find that RdxS can end normally, but get wrong results (denoted by "FL" in the Table 2.6) on HD5870 and Intel920. The benchmark uses the four-step radix sort in each pass proposed in papers [135, 176]. However, the implementation of RdxS depends on warp-size in CUDA, i.e., wavefront-size in APP. The warp-size is 32 in CUDA, while it is 64 in APP. Therefore, only one half wavefront of threads are able to map keys into buckets and the other half are not. This disparity leads to incorrectly sorted sequences. This is a typical example of hiding platform specific details into programs, and can be considered as a programmer's mistake.

To sum up, all the benchmarks compile correctly and most of them run properly on the other platforms, illustrating OpenCL's cross-platforms portability. In order to make OpenCL programs run on more platforms, programmers are encouraged to use vendor-independent terms (for example, CL_DEVICE_TYPE_ALL) and provide users with optional choices. After all, even minor modifications and additional debugging can be time-consuming. When it comes to a specific architecture, an auto-tuner could be used to boost performance [172]. Finally, we note that OpenCL is very useful as a prototyping tool, enabling portability while still achieving good performance.

2.5. Related Work

There has been a fair amount of work on performance comparison of programming models for multi-core/many-core processors. Rick Weber et al. [166] presented a collection of Quantum Monte Carlo algorithms implemented in CUDA, OpenCL, Brook+, C++, and VHDL. They gave a systematic comparison of several application accelerators on performance, design methodology, platform, and architectures. Their results show that OpenCL provides application portability between multi-core processors and GPUs, but may incur a loss in performance. Rob van Nieuwpoort et al. [157] explained how to implement and optimize signal-processing applications on multi-core CPUs and many-core architectures. They used correlation (a streaming, possibly real-time, and I/O intensive application) as a running example, investigating the aspects of performance, power efficiency, and programmability. This study includes an interesting analysis of OpenCL: the problem of performance portability is not fully solved by OpenCL and thus programmers have to take more architectural details into consideration.

In [161], the authors compared programming features, platform, device portability, and performance of GPU APIs for cloth modeling. Implementations in GLSL, CUDA and OpenCL are given. They conclude that OpenCL and CUDA have more flexible programming options for general computations than GLSL. However, GLSL remains better for interoperability with a graphics API. In [7], a comparison between two GPGPU programming approaches (CUDA and OpenGL) is given using a weighted Jacobi iterative solver for the bidomain equations. The CUDA approach using texture memory is shown to be faster than the OpenGL version. Kamran Karimi et al. [77] compared the performance of

CUDA and OpenCL using complex, near-identical kernels. They showed that there are minimal modifications involved when converting a CUDA kernel to an OpenCL kernel. Their performance experiments measure and compare data transfer time to and from the GPU, kernel execution time, and end-to-end application execution time for both CUDA and OpenCL. Only one application or algorithm is used in all the work mentioned above.

Ping Du et al. [36] evaluated many aspects of adopting OpenCL as a performance-portable method for GPGPU application development. The triangular solver (TRSM) and matrix multiplication (GEMM) have been selected for implementation in OpenCL. Their experimental results show that nearly 50% of peak performance could be obtained in GEMM on both NVIDIA Tesla C2050 and ATI Radeon 5870 in OpenCL. Their results also show that good performance can be achieved when architectural specifics are taken into account in the algorithm design. In [83], the authors quantitatively evaluated the performance of CUDA and OpenCL programs developed with almost the same computations. The main reasons leading to these performance differences are investigated for applications including matrix multiplication from the CUDA SDK and CP, MRI-Q, MRI-HD from the Parboil benchmark suite. Their results show that if the kernels are properly optimized, the performance of OpenCL programs is comparable with their CUDA counterparts. They also showed that the compiler options of the OpenCL C compiler and the execution configuration parameters have to be tuned for each GPU to obtain its best performance. These two papers inspired us to analyze the performance differences by looking into intermediate codes.

Anthony Danalis et al. [30] presented a Scalable HeterOgeneous Computing (SHOC) benchmark suite. Its initial focus was on systems containing GPUs and multi-core processors, and on the new OpenCL programming standard. SHOC is a spectrum of programs that test the performance and stability of these scalable heterogeneous computing systems. At the lowest level, SHOC uses micro-benchmarks to assess architectural features of the system. At higher levels, SHOC uses application kernels to determine system-wide performance including many systems features. SHOC includes benchmark implementations in both OpenCL and CUDA in order to provide a comparison of these programming models. Some of the benchmarks used in this work are selected from SHOC.

The majority of previous work has used very few applications to compare existing programming models. In our work, we tackle the problem by observing a large set of diverse applications to show the performance differences of CUDA and OpenCL. We also give a detailed analysis of the performance gap (if any) from all possible aspects. Finally, we discuss an eight-step fair comparison strategy to judge the performance of any applications implemented in both programming models.

In addition to the comparison work of OpenCL versus CUDA on GPUs, we have compared OpenCL versus OpenMP on multi-core CPUs [140–142]. We have selected multiple Rodinia benchmarks, and compared their OpenCL performance with the OpenMP performance. We have shown that, although there is a mismatch between the OpenCL platform model and multi-cores, our systematic tuning tips enable OpenCL developers to achieve a comparable performance. Hence, we claim that this work is a necessary step for enabling inter-platform performance portability in OpenCL.

2.6. SUMMARY

In this chapter, we have compared the performance of OpenCL versus CUDA on NVIDIA GPUs. We have used both synthetic benchmarks and real-world benchmarks. OpenCL performs as well as CUDA on synthetic benchmarks. A first comparison on the real-world benchmarks has shown performance gaps between OpenCL and CUDA. We have given a through investigation to the performance gaps, and analyzed that the performance gaps are due to differences in user inputs and compilers. We have also shown that how the performance gaps can be reduced by systematically changing code. Therefore, OpenCL can be a good alternative to CUDA on NVIDIA GPUs in terms of performance.

Our recent measurements have shown that the updated compiler (CUDA v4.2) and driver can further reduce the performance gaps between CUDA and OpenCL. We believe that, with the maturity of compilers, the performance gaps (from compilers) will be eliminated ultimately. Besides, other techniques to reduce performance gaps (e.g., from programmers' misuse) are equally required.

In addition, we evaluate portability of OpenCL on Intel CPUs, Cell/B.E., and AMD GPUs. We observe that the same OpenCL can run on these devices with no/minor code changes, i.e., achieving code portability. In this thesis, what we are looking for is a unified programming model that can provide functional portability. Given OpenCL's ability in functional portability and its potential in achieving matching performance with native programming models, we will use OpenCL as our research vehicle to investigate cross-platform performance in the following chapters.

3

EXPLORING OPTIMIZATION SPACE: A CASE STUDY

In this chapter, we present our parallelization for an application (with different algorithms) in computer vision. We implement it in OpenCL and explore its optimization space on a mid-range GPU (Quadro 5000) and a low-end embedded GPU (Quadro NV140M). Further, we try the same optimizations on a multi-core CPU (X5650) to investigate whether the optimization steps are equally effective.

Stereo-matching algorithms aim to find an estimate of the depth inside a scene based on rectified stereo pairs of images [136]. The input to the stereo matching consists of two images of the same scene, a *reference* image and a *target* image, with each image displaying the scene from a different perspective along the x-axis. The goal is to accurately retrieve the *disparity*, i.e., the relative depth information for each pixel along the x-axis. The results are often used to estimate distance to an object in the scene, as objects corresponding to pixels of greater disparity are closer to the camera(s) than objects corresponding to pixels of lesser disparity.

Stereo algorithms can be divided into two main categories: (1) local algorithms and (2) global algorithms. Local algorithms [102, 133, 174, 177] use aggregation of similarity measures around a local support region, i.e., the energy minimization is performed over local patches. By contrast, global algorithms [20, 21, 51] incorporate smoothness constraints on the depth estimates, leading to an energy minimization problem that involves all pixels in the images. In general, global algorithms outperform local algorithms in terms of accuracy due to the smoothness constraints, especially in non-textured image regions. However, global algorithms need to solve minimization problems that are generally NP-hard [12], prompting the use of approximations based on variational methods, graph cuts, or Monte Carlo simulation. Unfortunately, these approximations are still too slow for use in real-time algorithms, so local algorithms are preferred in this case.

This chapter is based on our work published in the Proceedings of ICPADS 2012 [45].

Most consumer cameras operate at rates of 25 or 30 fps (frames per second), which may be sufficient for computer graphics due to human perception. However, many applications have even higher requirements, e.g., stereo algorithms used in automatically driving cars have to operate at a minimum of 100 fps [98]. Otherwise, in a scene where objects move at several dozen meters per second, those cameras will produce severe motion blur or temporal aliasing effects [98]. The high fps rates pose high demands for both computational capability and memory bandwidth. For this, an alternative to algorithm innovation is to use accelerators, e.g., GPUs, to speedup stereo matching without losing accuracy.

In this chapter, we study the performance of local stereo-matching algorithms when implemented in OpenCL and executed on GPUs. Our main contributions are as follows:

- We analyze three commonly used state-of-the-art cost aggregators (a key step in stereo matching), and propose a unified representation that facilitates optimizations, performance prediction, and derivation of new algorithms (see Section 3.2).
- We provide implementations for GPUs in OpenCL, perform multiple incremental optimizations for the three aggregators, and show that they are up to hundreds of times faster than the single-core CPU code (see Section 3.3 and Section 3.4).
- We run our software on a multi-core CPU to check whether the optimization techniques are equally applicable on the multi-core (see Section 3.5).

The rest of the chapter is organized as follows: Section 3.1 presents the execution-time breakdown of the local stereo-matching algorithm. Section 3.2 introduces three typically used algorithms of cost aggregation, and gives a unified representation. Section 3.3 shows our stereo-matching framework implemented in OpenCL and five incremental optimization steps. Section 3.4 demonstrates our performance results and analysis in accuracy and speed on GPUs. Section 3.5 investigates the optimization steps on a multi-core CPU. Section 3.6 gives the related work in stereo matching, and Section 3.7 concludes this work.

3.1. A FIRST TRIAL

Local stereo-matching algorithms comprise four main steps: (1) matching cost computation (CC), (2) cost aggregation (CA), (3) disparity estimation (DE), and (4) disparity refinement (DR) [136]. The matching cost computations (CC) involve calculating pixel-wise differences– based on the image intensities or gradients– between the two images in the stereo pair. These costs are aggregated over a support region of interest in the cost aggregation step (CA). In the disparity estimation step (DE), the disparity that minimizes the aggregated cost at a location is selected as the disparity estimate for that location (generally, using a winner-takes-all/WTA procedure). The disparity refinement step (DR) fine-tunes the estimated disparities using a simple type of smoothing. Figure 3.1 shows the amount of computation time spent in each of the aforementioned four components for three different CA solvers. The CA step accounts for the most of the computation time in stereo matching (83.88%, 93.96%, and 50.28%, respectively). Therefore, we focus on accelerating CA in this chapter.

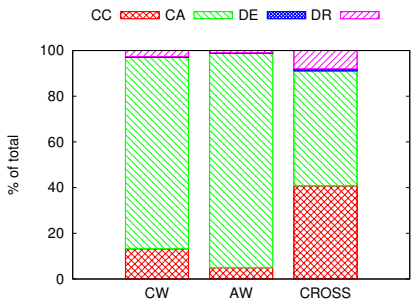


Figure 3.1: The percentage of each component for three CA solvers (CW, AW, CROSS). The window radius for CW and AW is 10, and L, Tao for CROSS are 17, 20, respectively. We use (AD+Census) cost similarity measures to compute match cost (CC)[97], WTA (DE), and several optimizers in the refinement step (DR).

3.2. ALGORITHMS AND THE REPRESENTATION

3.2.1. AGGREGATION STRATEGIES

An aggregation strategy combines the image-matching costs over a (local) support region in order to obtain a reliable estimate of the costs of assigning a disparity d to image location (x, y). We investigate three commonly used local aggregation strategies: (1) constant window aggregation, (2) adaptive weight aggregation, and (3) cross-based aggregation.

Constant Window Aggregation (CW) is a straightforward aggregation of any similarity measure cost C over a constant-size neighborhood:

$$C_{CW}(x,y,d) = \sum_{\forall (x',y') \in \mathcal{N}(x,y)} C(x',y',d), \tag{3.1}$$

where $\mathcal{N}(x,y)$ represents the set of pixels that are neighbors of pixel (x,y). CW assumes that every pixel in the window should share the same disparity. This assumption is violated in image regions in which the disparity is discontinuous (i.e., at object boundaries).

Adaptive Weight Aggregation (AW) [174] uses color similarity and proximity based weights as aggregation coefficients for the similarity measures pixels around pixels of interest. The dissimilarity between pixels $p = (x,y)$ in the reference image and $p_d = (x-d,y)$ in the target image is:

$$C_{AW}(p,p_d) = \frac{\sum_{q \in \mathcal{N}(p), q_d \in \mathcal{N}(p_d)} w(p,q)\, w'(p_d,q_d) C(p,d)}{\sum_{q \in \mathcal{N}(p), q_d \in \mathcal{N}(p_d)} w(p,q)\, w'(p_d,q_d)}, \tag{3.2}$$

where p_d and q_d are the corresponding pixels in the target image of p and q with disparity shift of d. The weights $w(p,q)$ depend on the color similarity and Euclidean distance between pixels p and q:

$$w(p,q) = \exp\left(-\left(\frac{\|I(p)-I(q)\|^2}{\gamma_c} + \frac{\|p-q\|^2}{\gamma_s}\right)\right), \tag{3.3}$$

where γ_c and γ_s are constant parameters that are set empirically. Calculation of target image weights $w'(p_d,q_d)$ is the same for target image pixels p_d and q_d.

Cross-Based Aggregation (CROSS) overcomes the problem of CW by constructing variable support for aggregation that depends on color similarity [177]. The first step of the algorithm is to construct a cross for each pixel inside the image. Four parameters are stored for each pixel, h_p^-, h_p^+, v_p^-, v_p^+, which identify the lengths of the four arms of the cross as shown for pixel p in Figure 3.2a (the light-shaded area). The decision on whether or not a pixel p' is included is made on its color similarity with pixel p as given in Equation 3.4.

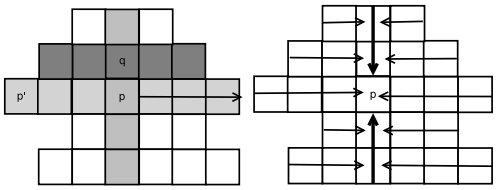


Figure 3.2: Cross-based aggregation: (a) Cross Construction, (b) Horizontal Aggregation, and Vertical Aggregation.

$$\delta((x,y),(x',y')) = \begin{cases} 1, & \text{iff } |I(x,y) - I(x',y')| \leq \tau \\ 0, & \text{otherwise.} \end{cases} \tag{3.4}$$

After calculating the arm lengths in four directions for each pixel, the final aggregation support region is constructed by integrating the horizontal arms of each neighboring pixel in vertical direction (e.g., the horizontal arms of pixel q in dark gray shown in Figure 3.2a). The same steps are also performed for the pixels of the target image. At the aggregation step, the intersection of the reference and the target support regions is aggregated. Orthogonal Integral Images (OIIs) can be used to speed up aggregation [177]. The technique separates the 2D aggregation procedure into two 1D aggregation steps: horizontal and vertical aggregation (see Figure 3.2b).

3.2.2. A TEMPLATE FOR COST AGGREGATION KERNELS

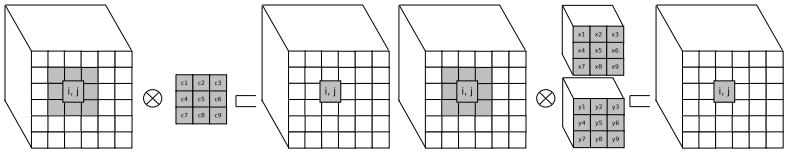
In order to give an overview of the three cost aggregation strategies, Figure 3.3 presents a kernel template for all of them. We see that the basic operation is a convolution between the input cost volume and the filter in the area determined by offsets(*offset_t, offset_b, offset_l, and offset_r*). We calculate the cost value (*ref_cost[]*, and *tar_cost[]*) at each *Disparity(D)* level for each pixel(x,y). When the *filter* and *offsets* are specified, the kernel template evolves into three different cost aggregation kernels (shown in Figure 3.4).

- **Constant Window Aggregation (CW)**: as shown in Figure 3.4a, the filter is filled with constant values, and the offset is constant for all the pixels (i.e., the window size is the same for all the pixels in the image).
- **Adaptive Weight Aggregation (AW)**: as illustrated in Section 3.2.1, each pixel has its own filter, which is calculated according to the input images and is different be-

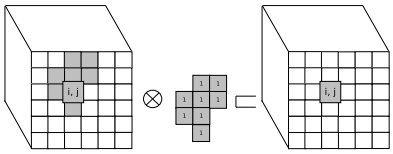
```
__kernel void aggregate(cost[], filter[], \
                        out_ref[], out_tar[]){

  for each y in H
    for each x in W
      for each d in D{
        res = 0;
        for each y_ in <offset_t ... offset_b>{
          for each x_ in <offset_l ... offset_r>{
            res += cost[] OP filter[];
          }
        }
        out_ref[] = res;
        out_tar[] = res;
      }
}
```

Figure 3.3: A unified kernel template for cost aggregation.



(a) CW Aggregation (b) AW Aggregation



(c) CROSS Aggregation

Figure 3.4: Three cost aggregators for pixel(i, j) and disparity 0. As for AW, each pixel has a unique filter, thus forming a filter cube shown in (b).

tween pixels. However, the filter size is fixed for all the pixels (Figure 3.4b). Further, two filters from the reference and target images are required for each pixel.

- **Cross-Based Aggregation (CROSS)**: as shown in Figure 3.4c, there is no fixed filter for CROSS, but, for the sake of this common representation, we can define it as an irregular region filled with all ones. The offsets are calculated according to each pixel.

Using this template, we can develop new algorithms for cost aggregation. For example, the combination of AW and CROSS leads to a new cost aggregation solver with adaptive sized and weighted filters. Further, our template enables common optimizations at both architecture-level (e.g., exploiting data sharing for CW and AW) and algorithm-level (e.g., using the *OII* technique when filters are all filled with ones).

3.3. IMPLEMENTATIONS AND OPTIMIZATIONS

We discuss here implementations and optimizations of the three cost aggregation solvers (CW, AW, and CROSS) on GPUs, and evaluate their performance impact [1].

3.3.1. OPENCL IMPLEMENTATIONS

Figure 3.5 shows our framework for implementing the local stereo matching algorithm in OpenCL [2]. Before porting kernels to the computing device, OpenCL has to set up a context and allocate the required device memory. Thereafter, we upload images (or video data) to the device (H2D), and compute disparity in four steps (CC, CA, DE and DR), as mentioned in Section 3.1. We then transfer the disparity image back to the host after these four steps (D2H). When finalizing the program, these contexts and device memory space are released. As stated in Section 3.1, we focus on the cost aggregations (CA, shaded in Figure 3.5).

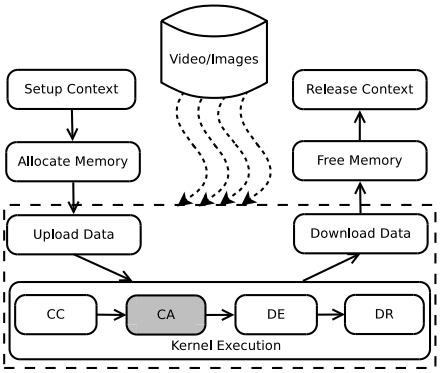


Figure 3.5: Stereo matching framework implemented in OpenCL.

According to the kernel template for cost aggregations in Figure 3.3, we let each work-item process one pixel(x, y). Thus, a grid of $W \times H$ work-items is created (with an exception to be illustrated in Section 3.3.2).

3.3.2. OPTIMIZATION STEPS FOR CA ON GPUS

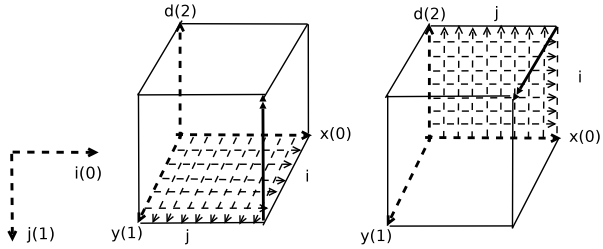
We focus on five incremental optimization steps for cost aggregations: (1) mapping work-items to data, (2) exploiting data sharing, (3) caching data for coalescing, (4) unrolling loops, and (5) increasing parallelism.

MAPPING WORK-ITEMS TO DATA

The input into stereo cost aggregation is a 3-dimensional data matrix (the input cost volume), and a 2-dimensional thread grid is commonly used for GPU implementations. Then the question is how to efficiently map the work-items to the data matrix (i.e., the iteration space)? To maximize memory bandwidth on GPUs, neighboring work-items

[1] The experiment setup is given in Section 3.4.
[2] The source code is on-line available: https://github.com/haibo031031/rtstereo.

prefer accessing the spatially close data elements in groups, i.e., coalesced memory access [118]. For cost aggregation, we have $A_3^2 = 3 \times 2 = 6$ ways to map the work-item space to the data space. Figures 3.6b and 3.6c show two ways for the work-items to iterate through the data space, i.e., to force the work-items in the first dimension of a work-group to access the data elements in the first dimension of the input cost volume.



(a) (b) $(i, j) \rightarrow (x, y)$ (c) $(i, j) \rightarrow (x, d)$

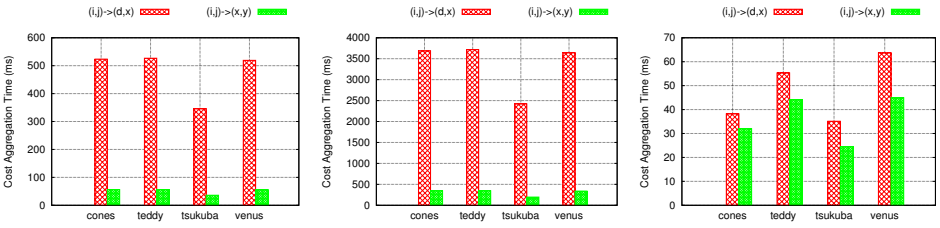
Figure 3.6: Iteration space. (i, j) are the 2 dimensions of the work-items space, and (x, y, d) are the 3 dimensions of the input cost volume.

We design micro-benchmarks for the six ways (of mapping 2D work-items to 3D data), and show how they perform on three data sets: 512×512, 1024×1024, 2048×2048 (when $D = 16$) in Table 3.1. We see that the $(i, j) \rightarrow (x, y)$ and $(i, j) \rightarrow (x, d)$ mappings (shown in Figure 3.6b and 3.6c) perform much better than the other four (about 20x speedup over the case $(i, j) \rightarrow (d, y)$). Further, $(i, j) \rightarrow (x, y)$, which is our final preference (except the case mentioned in 3.3.2), performs slightly better than $(i, j) \rightarrow (x, d)$, due to more work-groups to hide latency.

Table 3.1: Performance comparison of six ways to map 2D work-items to 3D data (ms).

	$(i, j) \rightarrow (x, y)$	$(i, j) \rightarrow (x, d)$	$(i, j) \rightarrow (y, x)$	$(i, j) \rightarrow (y, d)$	$(i, j) \rightarrow (d, x)$	$(i, j) \rightarrow (d, y)$
512x512	0.42	0.50	1.06	6.33	2.58	7.08
1024x1024	1.50	1.72	4.19	24.48	12.15	28.70
2048x2048	5.80	7.01	16.76	93.92	64.00	117.31

We implement two mappings: $(i, j) \rightarrow (x, y)$ (our preference) and $(i, j) \rightarrow (d, x)$ (one of the 'bad' mappings) in the three cost aggregation kernels, and compare their performance (shown in Figure 3.7). We see that $(i, j) \rightarrow (x, y)$ shows a significant performance gain over $(i, j) \rightarrow (d, x)$ (the average speedups for CW and AW are 9.4x and 11.1x, respectively). Further, the CROSS kernel can be only accelerated by around 24%. This is because the filter for each pixel is so adaptive that neighboring work-items may access data elements far away from each other (depending on the input images). In other words, coalesced access is not fully ensured for the CROSS kernel, which will be further discussed in Section 3.3.2.



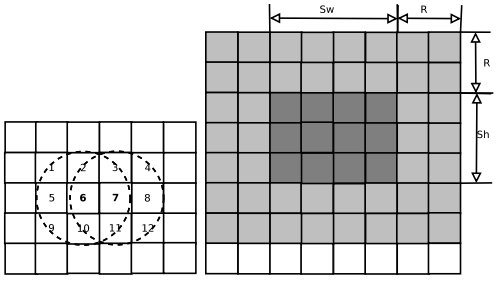
(a) CW (b) AW (c) CROSS

Figure 3.7: Aggregation time of three aggregation strategies (CW, AW, and CROSS) on four datasets (cones, teddy, tsukuba, and venus) in Middlebury using two mappings ($(i,j) \rightarrow (x,y)$ and $(i,j) \rightarrow (d,x)$). The window radius for CW and AW is 10, and L, Tao for CROSS are 17, 20, respectively.

EXPLOITING DATA SHARING

For the window-based aggregation (CW and AW), each work-item requires a block of data elements, and neighboring work-items share the overlapped data to aggregate cost. For example, when the window radius is 1 (shown in Figure 3.8), work-item 6 requires to load data elements at $\{1,2,3,5,6,7,9,10,11\}$, while work-item 7 needs data elements at $\{2,3,4,6,7,8,10,11,12\}$. Thus, we can use the on-chip $local\ memory$ to share the data elements at $\{2,3,6,7,10,11\}$. In general, suppose the window radius is R (shown in Figure 3.8b), and the work-group size is $S_w \times S_h$, we calculate the $Shared\ Data\ Ratio$ (SR) for a work-group using Equation 3.5. For the example shown in Figure 3.8a, $SR = 2/3$. In theory, we expect to achieve $1/SR$ speedup for memory-bound applications.

$$SR = \frac{(S_w + 2 \times R) \times (S_h + 2 \times R)}{S_w \times S_h \times (2 \times R + 1)^2} \tag{3.5}$$



(a) An example. (b) A generic model.

Figure 3.8: Data sharing demonstration.

The difficulty of using local memory lies in dealing with *halo data* [15], the data elements shaded in light-gray in Figure 3.8b. As stated in the CUDA SDK, we use an extended work-group: the boundary work-items only participate in loading data from global memory to local memory, and remain idle for the other time. Figure 3.9 shows how the optimized CW performs when varying R. When the window is very small, we

see a performance gain. However, when the R is larger than 5, the performance gain disappears because more and more work-items stay idle after loading data. Further, this approach is not scalable with the window size, i.e., as the radius of the window increases, the percentage of idle work-items increases and the size of a work-group may exceed its maximum limit [151]. Given the expected poor performance for AW, we choose not to implement AW with this approach.

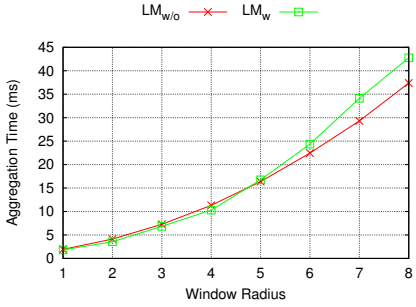
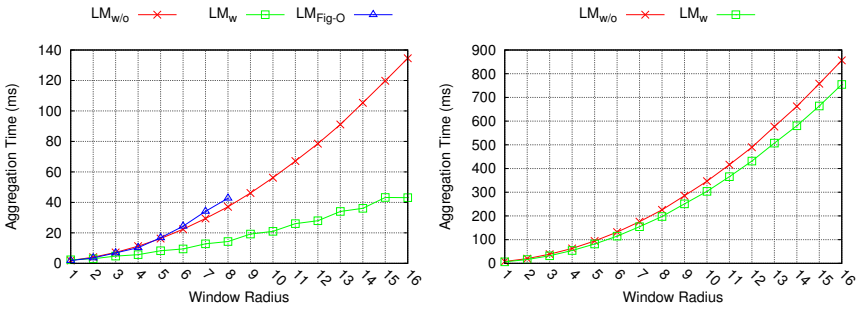


Figure 3.9: Performance comparison of CW when using local memory for different window-radius. $LM_{w/o}$ represents the aggregation time without using local memory, and LM_w represents the case with that.

To tackle the lack of scalability, an alternative approach is to keep the geometry and the size of a work-group constant and load multiple data elements per work-item (the exact number of data elements processed by one work-item depends on the filter radius and the work-group size). From Figure 3.10, we see the performance improvement using this revised approach. The average speedup is 2.3 for CW and 1.2 for AW, respectively. Using local memory can accelerate AW less significantly, because AW needs to load two other elements (i.e., filter elements) from the global memory, which becomes the performance bottleneck.



(a) CW (b) AW

Figure 3.10: Performance comparison when using local memory for two aggregation methods (CW, AW) and different window-radius. $LM_{w/o}$ represents the aggregation time without using local memory, and LM_w represents the case with that. LM_{Fig-O} means that we use local memory in the way shown in Figure 3.9.

CACHING DATA FOR COALESCED ACCESS

We see from Figure 3.4c that the CROSS cost aggregation requires an adaptive area of data elements for each pixel. Thus, neighboring work-items will access the global memory possibly in an un-coalesced way. Local memory can be used to explicitly convert a scattered access pattern to a regular (coalesced) pattern for global memory access [118]. To be specific, when using local memory, the coalesced access from global memory to local memory is ensured and operating the local memory itself has no coalescing constraints. Thus, for CROSS, we first load data elements within the area of radius L (the predefined maximum limit on radius) from global memory in a coalesced way, and then use the required data elements from local memory.

In Figure 3.11, we see that using local memory for caching can improve the CROSS performance by 1.5x on average. Although we load more data elements than what we need in this situation, we have achieved better performance, due to the guarantee of coalesced data access from global memory.

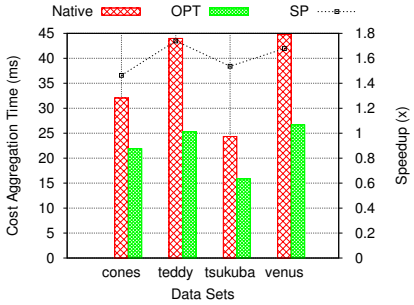


Figure 3.11: CROSS performance on four data sets when caching data wiht local memory ($L = 17$, $Tao = 20$). OPT represents the kernel with caching.

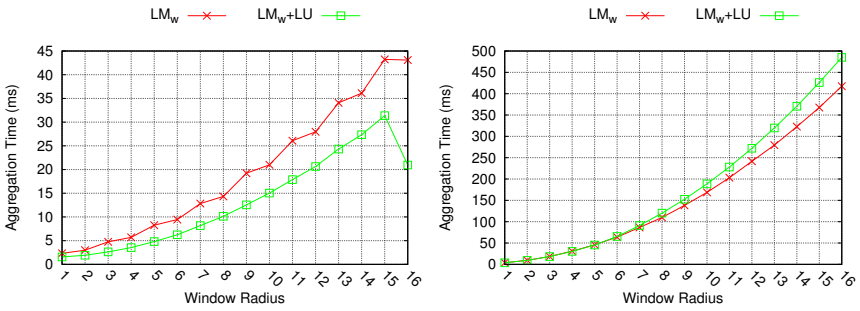
UNROLLING LOOPS

Loop unrolling is an optimization technique in which the body of a suitable loop is replaced with multiple copies of itself, and the control logic of the loop is updated at well [106]. In this way, we can reduce the number of branch instructions, thus improving performance. For-loops exist in cost aggregation kernels due to the heavy usage of filters (shown in Figure 3.12). Thus, we generate the kernels in the fully loop-unrolled fashion and then evaluate their performance.

Figure 3.13a shows that we can further improve the CW performance using local memory plus loop unrolling (the average speedup 1.5x). Additionally, we see that using loop unrolling on AW decreases the performance (in Figure 3.13b). This slow-down is due to register pressure (i.e., each work-item of AW needs to load two more filter elements from global memory for each iteration). Thus, we choose not to perform loop unrolling on AW. As for CROSS, the irregularity of filters hinders the loop-unrolling during compiling time.

```
1  // ... head code ...
2  // cache[] represents the usage of local memory
3  // l_c and  l_r is the local column and row index
4  if(l_c<16 && l_r<16){
5      float cost_aggr=0;
6      for(int y_ = 0; y_ < wnd_side; y_++){
7          for(int x_ = 0; x_ < wnd_side; x_++){
8              int x_start = l_c + x_;
9              int y_start = l_r + y_;
10             cost_aggr += \
11               cache[y_start*length+x_start]*filter[];
12         }
13     }
14 }
15 // ... tail code ...
```

Figure 3.12: Cost aggregation kernels with loops.



(a) CW (b) AW

Figure 3.13: Performance comparison between with and without loop unrolling for CW, AW and different window-radius. LM_w represents the aggregation time only using local memory (no loop unrolling), and $LM_w + LU$ represents the case with both local memory and loop unrolling.

INCREASING PARALLELISM

When we parallelize aggregation kernels, letting each work-item process one pixel is a natural choice. However, it becomes difficult to do so when using *OII* to pre-compute prefix sum for the CROSS cost aggregation (see Section 3.2.1 and [177] on *OII*), because of the data dependency in the x or y direction (e.g., data dependency in the x direction shown in Figure 3.14).

In this case, we maximize parallelism by using a third dimension, namely D (representing *Disparity*). Specifically, we map the 2D work-items to the 2D elements in the (x,d) or (y,d) plane, and iterate over the third dimension (y or x, respectively). Once the third dimension is used, more work-groups can fully populate the compute unit to hide latency, leading to high occupancy [118]. We show that the performance differs significantly between using 1D and 2D parallelism in Table 3.2. The speedup ranges from 1.4x to 4.1x, depending on the size of the images. For the larger images, the gain is less significant. In fact, for very large images, we expect that the performance gain using 2D parallelism will disappear, as there will be enough work-groups to hide latency in one dimension. However, in real-time stereo matching, the commonly used images, e.g., the Middlebury dataset [28], are smaller than or close to 512×512, and they will benefit a lot

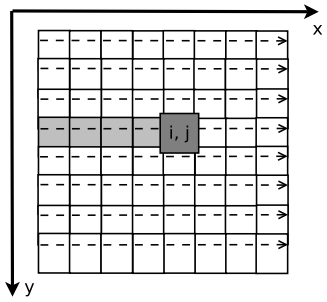


Figure 3.14: Data dependency when calculating prefix sum in x direction: to calculate the cost value at (i, j), we use Equation $cost(i, j) = \sum_{i' \leq i, j' = j} cost(i', j')$. So the integrated cost value at (i, j) depends on all its previous elements, namely the elements shaded in gray.

from this 2D parallelism model.

Table 3.2: Prefix sum execution time using 1D and 2D parallelism (D=16).

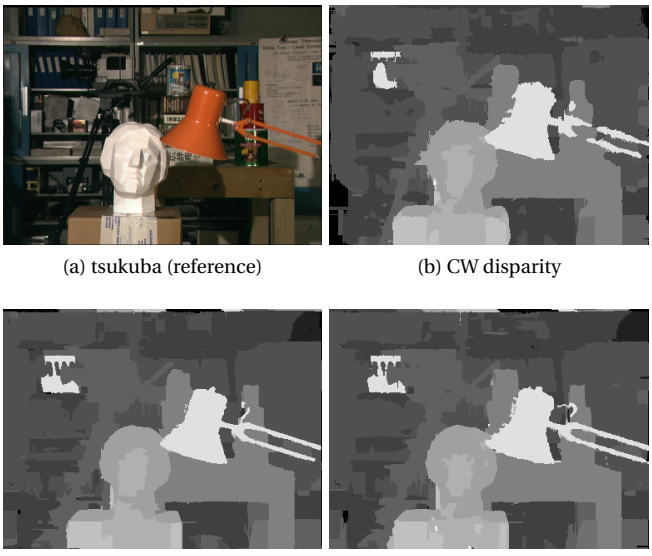
	512x512	1024x1024	2048x2048	4096x4096
1D(ms)	24.03	49.28	122.73	421.58
2D(ms)	5.81	22.18	76.83	299.11
speedup(x)	4.14	2.22	1.60	1.41

3.4. OVERALL PERFORMANCE

In this section, we show and discuss the overall performance results. All the experiments (except those in Section 3.4.3) are performed on a NVIDIA Quadro5000 Fermi GPU, connected to the host (Intel Xeon X5650). The card has Compute Capability 2.0 and consists of 352 cores divided among 11 multiprocessors. The amount of local memory available per multiprocessor is 48KB. We compile all the programs with the OpenCL implementation from CUDA version 4.2 and GCC version 4.4.3. We use four image pairs from the Middlebury dataset: cones, teddy, tsukuba, and venus. We perform initial experimental analysis on accuracy (in Section 3.4.1), and focus on speed (in Section 3.4.2 and 3.4.3). For the speed part, we use the average results of these four data sets.

3.4.1. ACCURACY

Figure 3.15 shows the disparity results for tsukuba. Compared with the ground truth image, the error rates are 8.29%, 4.28% and 5.41% for CW, AW and CROSS, respectively, i.e., $AW > CROSS > CW$ in terms of accuracy. As accuracy analysis or improvement is not the focus of this paper (we focus on improving the speed of the three aggregation solvers without a loss in accuracy), we will not dive into more details.



(a) tsukuba (reference) (b) CW disparity

(c) AW disparity (d) CROSS disparity

Figure 3.15: Accuracy comparison: (a) the tsukuba data set; (b)-(c) are the output disparity images for CW, AW, and CROSS using AD+Census cost similarity measures[97], WTA, and several optimizers in the refinement step.

3.4.2. SPEED ON THE QUADRO5000

We show the aggregation time (Sequential versus Optimized) for CW, AW, and CROSS in Figure 3.16. Note that the sequential version is the single-core CPU code (with the default optimization level). From Figure 3.16a, we see that AW is the most time-consuming aggregation solver (although it can achieve the most accurate disparity map shown in Figure 3.15), and its aggregation time increases in a polynomial manner with the window-radius. When using very small windows, CW runs faster than CROSS. However, as the window becomes bigger, the time for CW increases, while it remains stable for CROSS (that is because CROSS is window-independent).

As shown in Figures 3.16b, 3.16c and 3.16d, the optimization steps improve the aggregation performance significantly. The speedups over the sequential code stay around 90 for AW, while we can achieve more performance improvement for CW (the average speedup is 523) with the increasing of window sizes. This is due to more data sharing within a work-group. Further, we can achieve relatively small speedup for the CROSS solver (53x on average), due to loading extra data elements to local memory.

As can be seen from Figure 3.16d, when the window radius is 16, the performance improves dramatically (compared with the case when the window radius is 15), because all work-items in a work-group are active when loading data from global memory to local memory. Further, we can verify that the achieved GFLOPs is below the theoretical GFLOPs: the size of tsukuba is 384×288, the filter radius is 16, $D = 16$, and it takes 13.37 ms to perform the CW aggregation. The achieved computational throughput is

144 *GFLOPs* (see Equation 3.6), which is lower than the theoretical peak of Quadro5000 - 359 *GFLOPs* for addition only operations.

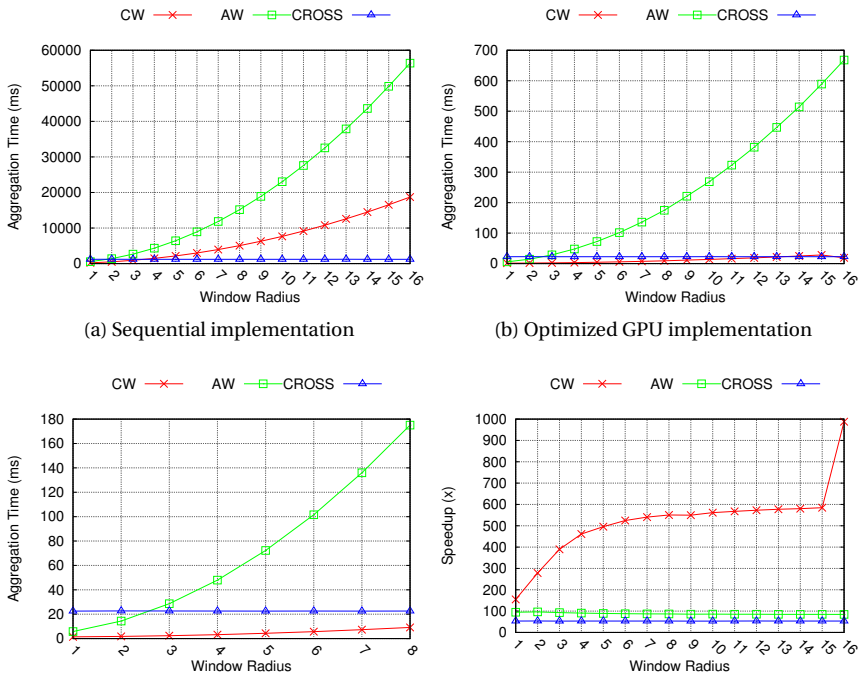
$$\frac{384 \times 288 \times 16 \times 33 \times 33}{13.37} \times 10^{-3} = 144\ GFLOPs \tag{3.6}$$

3.4.3. SPEED ON THE LOW-END GPU

We evaluate our parallel implementation on a low-end GPU: NVIDIA Quadro NV 140M (2 multiprocessors, 16 cores each). Figure 3.17 shows the speedup of CW compared with its sequential implementation. We see that the optimized CW can obtain impressive speed-up factors, ranging from 5x to around 60x. One the other hand, the AW fails to run, because it consumes a lot of device memory, which exceeds the maximum limits. As for CROSS, it suffers a 25% performance loss, because the older generation NV 140M imposes more severe constraints on effective memory accesses.

3.4.4. PUTTING IT ALL TOGETHER

As mentioned in Section 3.3.1, the local stereo-matching algorithm consists of six steps when implementing it in OpenCL (apart from the context management). After performing all these optimization steps, the percentages of each component are shown in Figure



(a) Sequential implementation

(b) Optimized GPU implementation

(c) Zoomed show of (b)

(d) Speedup

Figure 3.16: Performance comparison of sequential implementation and the optimized GPU implementation.

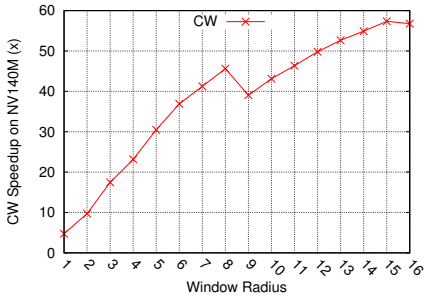


Figure 3.17: Speedup of the optimized CW over its 1-thread implementation on the low-end NV140M.

3.18. We see that CC becomes the performance bottleneck, rather than CA (although we perform the optimization on the other steps besides CA). Thus, we will study the local stereo-matching algorithm as a whole to get better performance for further work.

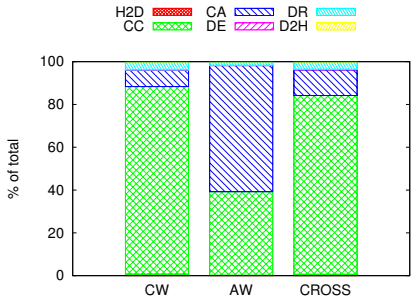


Figure 3.18: The percentage for each component using OpenCL. The window radius for CW and AW is 10, and L, Tao for CROSS are 17, 20, respectively. We use (AD+Census) cost similarity measures to compute match cost (CC)[97], WTA (DE), and several optimizers in the refinement step (DR).

To summarize, we have identified and performed five optimization steps on the aggregation solvers (CW, AW, and CROSS). Compared with the sequential implementations, the experimental results show significant speedup on NVIDIA Quadro5000: CW is around 500 times faster (50-700 fps), while AW and CROSS are tens of times faster (AW reaches 1.5-170 fps and CROSS 45 fps). Thus, meeting real-time requirements is feasible. Further, based on the standard template plus these key optimizations steps, a straightforward extension of this work is to build an auto-tuner for stereo matching.

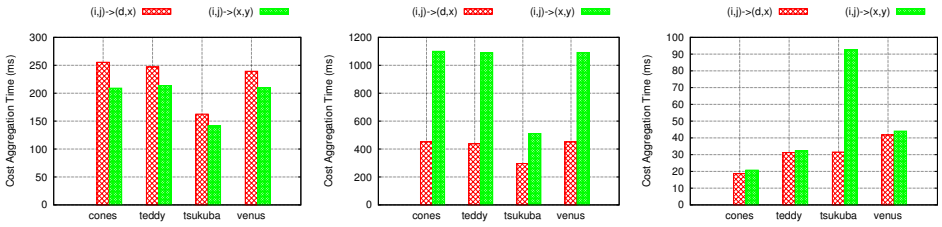
3.5. SUPPLEMENTARY RESULTS ON A MULTI-CORE CPU

In this section [3], we investigate whether the optimization steps specialized for many-core processors are equally effective on multi-core processors. To this end, we try the same optimization steps on a dual hexa-core CPU (Intel Xeon X5650) with Intel OpenCL SDK 2014.

[3] This section has been extended based on our work published in ICPADS 2012 [45]

3.5.1. MAPPING WORK-ITEMS TO DATA

We compare the two mappings: $(i, j) \rightarrow (x, y)$ (our preference on GPUs) and $(i, j) \rightarrow (d, x)$ (one of the 'bad' mappings for GPUs) in the three cost aggregation kernels in Figure 3.19. We note that, different from GPUs, the preferred mapping $(i, j) \rightarrow (x, y)$ performs worse than the bad mapping $(i, j) \rightarrow (d, x)$ for AW (Figure 3.19b) and CROSS (Figure 3.19c). Using $(i, j) \rightarrow (x, y)$ runs faster for CW (Figure 3.19a) whereas the performance benefits are much smaller (than that on GPUs). Therefore, applying coalesced memory access may not apply to multi-core processors.

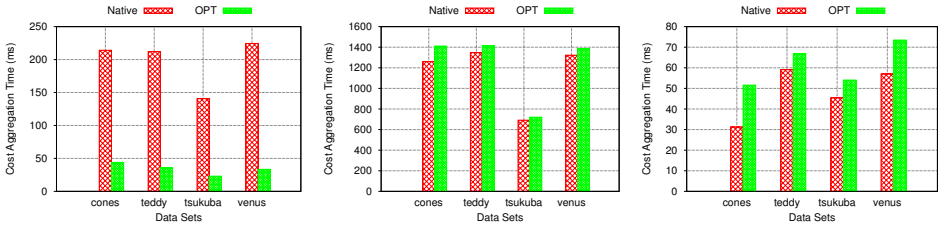


(a) CW (b) AW (c) CROSS

Figure 3.19: Aggregation time of three aggregation strategies (CW, AW, and CROSS) on four datasets (cones, teddy, tsukuba, and venus) in Middlebury using two mappings ($(i, j) \rightarrow (x, y)$ and $(i, j) \rightarrow (d, x)$). The window radius for CW and AW is 10, and L, Tao for CROSS are 17, 20, respectively.

3.5.2. USING LOCAL MEMORY

In Section 3.3.2, we use local memory to *exploit data sharing* and *cache data for coalesced access*. We use the same optimization step on X5650 and show results in Figure 3.20. We see that using local memory on X5650 can reduce the aggregating time significantly for CW (Figure 3.20a). Meanwhile, using local memory for AW (Figure 3.20b) and CROSS (Figure 3.20c) runs slower than the naive version. We conclude that the benefits of using local memory depend on both aggregators and platforms.

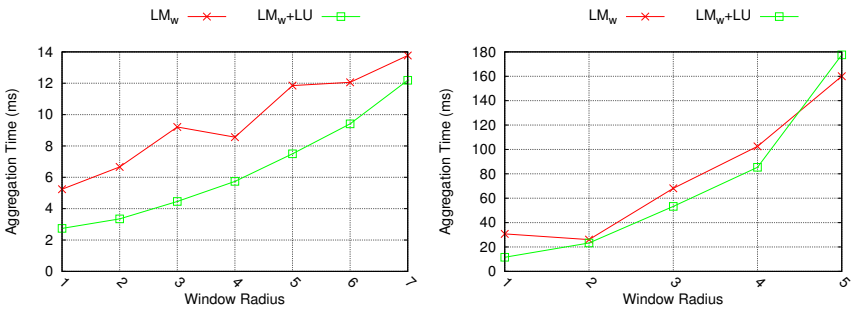


(a) CW (b) AW (c) CROSS

Figure 3.20: Aggregation time between without ($Naive$) and with (OPT) local memory.

3.5.3. UNROLLING LOOPS

Figure 3.21 shows that how unrolling loops impacts the overall execution time of CW and AW. We see that unrolling loops decreases aggregation time when the window radius is small. When using a large window (e.g., 8 for CW and 6 for AW), the run-time reports 'CL_OUT_OF_RESOURCES'. Compared with many-core GPUs, multi-core CPUs have an order of magnitude fewer registers. Therefore, the unrolled kernels consume too many registers to launch the kernel execution.



(a) CW (b) AW

Figure 3.21: Performance comparison between with and without loop unrolling for CW, AW and different window-radius. LM_w represents the aggregation time only using local memory (no loop unrolling), and $LM_w + LU$ represents the case with both local memory and loop unrolling.

3.5.4. INCREASING DATA PARALLELISM

Table 3.3 shows the execution time when increasing parallelism. We see that using this technique decreases performance significantly (by 2×–3×). This is because multi-core processors have fewer cores, and a medium level of parallelism can meet the requirement.

Table 3.3: Prefix sum execution time on X5650 using 1D and 2D parallelism (D=16).

	512x512	1024x1024	2048x2048	4096x4096
1D(ms)	2.44	5.00	21.18	76.85
2D(ms)	4.87	14.25	62.06	227.75
speedup	0.50	0.35	0.34	0.34

To summarize, an optimization step on NVIDIA Quadro 5000 often leads to a performance drop on X5650 for the cost aggregators. Therefore, optimizations steps are platform-dependent.

3.6. RELATED WORK

In this section, we present previous work on GPU-assisted stereo matching (we have mentioned related work on stereo cost aggregation solvers in Section 3.2).

In [165], Wang et al. introduce an adaptive aggregation step in a dynamic-programming (DP) stereo framework, and utilize the vector processing capability and parallelism in commodity graphics hardware, increasing the performance by over two orders of magnitude. Their performance improvements are mainly based on the usage of texture memory. Gong [54] gives an implementation of six aggregation approaches, two of which (CW and AW) are also implemented in our work, and evaluates them on a real-time stereo platform in terms of both accuracy and speed. The used platform includes programmable graphics hardware. Compared with our work, both these solutions use older GPU programming methods and focus on different optimizations.

More recently (since 2006), GPUs have evolved into computing devices solving general-purpose problems. At the same time, CUDA [115] and OpenCL [151] make programming GPUs easier. As a result, more research effort is put in using GPUs for stereo matching. In [175], the authors propose a GPU-based stereo system, consisting of hardware-aware algorithms and code optimization techniques, which improves both the accuracy and the speed of stereo-matching.

Scott Grauer-Gray et al. [56] explore methods to optimize a CUDA-implementation of belief propagation (i.e., a global stereo matching method). They focus their investigation on the optimization space of using local, shared, and register memory options for data storage on the GPU. Mei et al. [97] present a near real-time stereo system with accurate disparity results using multiple techniques, viz., AD-Census costs, cross-based support regions (which is also evaluated in our work), scanline optimization, and a systematic refinement process. The results are achieved on NVIDIA GTX480 within 0.1 seconds. By contrast, our work focuses larger sets of aggregators, and investigates the performance benefits of a unified representation and optimizations on them.

There exists also some work in the literature on finding appropriate agglomeration and tiling factors for such stencil-like computations with a given stencil radius, taking local memory size into account. For instance, Werkhoven et al. present an adaptive tiling to implement a highly efficient, yet flexible, library-based convolution operation for modern GPUs [158]. In [32], Dastgeer et al. describe their work on providing a generic yet optimized GPU implementation for the 2D MapOverlap skeleton, where they explain their implementation with the help of a 2D convolution application, and the memory (constant and shared memory) and adaptive tiling optimizations are applied. In our work, we are not focusing on adaptive tiling, which will be explored as a next step.

In this still sparse space of GPGPU-enabled stereo-matching, our work proposes a unified approach for multiple cost-aggregation kernels, in terms of both implementations and optimizations; furthermore, because of this unified approach, we are able to propose the first (to the best of our knowledge) simple and effective performance model to predict an upper bound for the GPU-based stereo-matching speed on a given platform [4].

3.7. SUMMARY

In order to meet real-time requirements for cost aggregations, we have studied three typical aggregation solvers in OpenCL on GPUs: CW, AW, and CROSS. For the three ag-

[4] Our performance model is not shown in the thesis, and can be seen in [45].

Table 3.4: A summary of optimizations and the relevant architecture features (↗: increase, ↘: decrease, −: not applicable, ⇝: varied).

	Optimizations	Architecture Features	Quadro5000			X5650		
			CW	AW	CROSS	CW	AW	CROSS
1	Mapping work-items to data	coalesced memory access	↗	↗	↗	↗	↘	↘
2	Exploiting data sharing	local memory	↗	↗	−	↗	↘	−
3	Caching data for coalesced access	local memory	−	−	↗	−	−	↘
4	Unrolling loops	(less dynamic instructions)	↗	↘	−	↗	⇝	−
5	Increasing parallelism	many cores	−	−	↗	−	−	↘

gregators, we devised a unified representation and implementation, and we performed five incremental optimization steps: (1) mapping work-items to data, (2) exploiting data sharing, (3) caching data for coalescing, (4) unrolling loops, and (5) increasing parallelism (Table 3.4). Experimental results show that we can significantly boost the performance of cost aggregations without a loss in accuracy on NVIDIA GPUs.

To verify whether the optimization steps proposed for GPUs are equally applicable in a different context, we ran our software on a multi-core CPU. We have observed that using the optimizations leads to a performance decrease in most cases (Table 3.4). Therefore, these optimization steps are not portable to multi-core CPUs, and each platform has a specific optimization space.

Further, we relate the optimizations to the underlying architectural features. Optimizations (1)–(3) relate closely to memory hierarchy and the use of on-chip memory. Optimization (4) is generic in that it aims to reduce the number of dynamic instructions. And optimization (5) exploits the many-core feature and uses more parallelism to hide latency and achieve high throughput. Therefore, the architectural differences between multi-cores and many-cores lead to differences in performance impact when enabling the same optimization. This is further confirmed by our case study in Appendix B.

To achieve high performance across platforms, we need to customize this optimization space, i.e., transforming a platform-agnostic implementation to a platform-specific version. In this thesis, we investigate the platform-specific optimization space and its enabling/disabling technologies for multi-cores and many-cores. In particular, we explore the optimization space from both processing cores (Chapter 5) and memory hierarchies (Chapter 6– 8).

4

EVALUATING VECTOR DATA TYPE USAGE

In this chapter, we investigate vectorization (i.e., explicitly using vector data types) for OpenCL kernels on both scalar-core processors and vector-core processors. In particular, we present two approaches to use vector data types (VDT), and further evaluate the performance impact of using VDT with micro-benchmarks and macro-benchmarks. In addition, we discuss the balance between using vector data types and the performance-portability requirement.

A vector of type `vec`*n* has *n* elements of type `vec`, i.e., a vector is a fixed-length collection of scalar data elements. Using VDTs is regarded as *explicit* and *high-level vectorization* (different from using the low-level intrinsics [43]). This technique aims to enable the mapping of vector data directly to the hardware vector registers. When the underlying hardware has vector units, using VDTs may facilitate better utilization, thus bringing new opportunities for improved performance.

Although vectorization itself is a well-studied topic, there are very few studies conducted on the usage of vector data types [125]. Thus, many questions are still unanswered: (Q1) *how should we use VDTs given a scalar kernel code*, (Q2) *what is the execution model of VDTs for different processors*, and (Q3) *what is the performance impact of using VDTs*? Further, the *vector length (VL)* differs for different processors and applications (see Section 4.4), making it challenging to achieve high performance without tuning for each machine. So, another question is (Q4) *how do we balance the VDT usage and the performance-portability requirement*?

To address these four questions, we first present a source-to-source approach (*inter-vdt* and *intra-vdt*) to translate scalar kernels to vectorized ones. Next, to investigate the performance effects of using VDTs, we apply the approach on two types of benchmarks[1]: micro-benchmarks and macro-benchmarks. With micro-benchmarks, we investigate

This chapter is based on our work published in Concurrency and Computation: Practice and Experience [47].
[1] The benchmarks are on-line available: `https://github.com/haibo031031/vdt`

how vector data types are mapped on five different devices. With macro-benchmarks, we thoroughly evaluate the performance effects of using VDTs. In particular, we analyze the changes in memory access patterns and bandwidth due to the usage of VDTs. We have found that (1) VDTs are mapped in different manners onto different processors; (2) using VDTs not only brings changes to computation, but also to memory access patterns. Based on these observations, we discuss how to achieve more portable performance with VDTs.

In summary, our main contributions are as follows:

- We propose two orthogonal source-to-source translation approaches to use VDTs, and demonstrate their use on micro-benchmarks and macro-benchmarks.
- With the vectorized micro-benchmarks, we investigate the execution model(s) of VDTs on different architectures, and obtain a decision tree on when to use VDTs.
- With the vectorized macro-benchmarks, we study the performance impact of using VDTs.
- We analyze the role of auto-vectorization modules, and discuss how to deal with performance portability in the presence of VDTs.

The main goal of this chapter is to understand the usage of vector data types in a systematic way. Most previous work has emphasized thread-coarsening [92, 172] and OpenCL vectorization on CPUs [88]. To the best of our knowledge, ours is the first study dedicated to vector data types in the context of performance and portability. We believe that our work increases the understanding of VDT mappings on diverse processors and its performance effects, and helps programmers write performance-portable code in the presence of VDTs.

The remainder of the chapter is organized as follows. In Section 4.1, we briefly describe OpenCL and VDTs, and then present our source-to-source translation approaches. Section 4.2 describes the experimental setups: benchmarks and devices. Section 4.3 presents the microbenchmark results and our analysis at both the architecture level and compiler level. Section 4.4 and Section 4.5 discuss the performance results of using vector data types with macro-benchmarks. Section 4.7 introduces the vectorization background and related work. We discuss performance portability in the presence of VDTs in Section 4.6 and conclude the chapter in Section 4.8.

4.1. SOURCE-TO-SOURCE TRANSLATION

In this section, we introduce vector data types, and show how to explicitly use VDTs in OpenCL kernels. We propose two different approaches of using VDT in OpenCL kernels, and present the transformations that have to be applied to enable them.

4.1.1. OPENCL AND VDT

The secret of OpenCL's portability is that it uses a unified platform model. The model consists of a host with one or multiple OpenCL devices. An OpenCL device is divided into one or more compute units (CUs) which are further divided into one or multiple

processing elements (PEs). Each component has its own memory space. When implementing OpenCL, vendors map this model to physical machines.

OpenCL uses a special language (based on C99) for writing kernels and APIs that are used to define and then control the devices. In OpenCL terms, applications are composed of "host code" - i.e., the code that manages execution contexts and runs on the hosts, and "kernel code" - i.e., the massively parallel code that runs on the devices. When a kernel is submitted for execution by the host, an index space, named *NDRange*, is defined. An instance of the kernel is known as a *work-item*. Work-items are organized into *work-groups*, providing a more coarse-grained decomposition. Each work-item has its own private memory space, and can share data via local memory with the other work-items in the same work-group. All work-items can read/write global device memory.

To maximize hardware utilization, OpenCL provides vector data types and related built-in functions. The vector data type is defined with the type name, i.e., `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `float`[2], `long`, `ulong`, followed by a literal value *n* that defines the number of elements in the vector. Manipulating vectors is performed in an element-wise manner. Built-in vector data types are supported by the OpenCL implementation even if the underlying compute device has no support for them in hardware. These are to be converted by the device compiler to appropriate instructions that use underlying built-in types supported natively by the compute device.

Starting with a scalar kernel, how can we transform the code into a vectorized format, where each operand is represented in a vector? Taking OpenCL as both the input and the output language, we provide, in the following sections, transformation recipes to explicitly use VDTs.

4.1.2. USING VECTOR DATA TYPES

Most current vectorization optimization techniques are loop-based: once a loop is determined to be *vectorizable*, the loop is strip-mined by vector length and each scalar instruction is replaced by the corresponding vector instruction (i.e., the operands are vectors, and the operation is element-wise) [3, 26, 170]. Additionally, a straight-line-code vectorization technique called *SLP (Superword-Level Parallelism)*, proposed in [86, 143, 145], targets basic blocks rather than loop nests. Thus, traditional vectorization is an *aggregation of consecutive loop iterations or instructions within a basic block*. Because explicitly using vector data types is a high-level vectorization, we can use a similar approach when enabling VDTs.

We explore two different ways of using VDT: *inter-vdt*, and *intra-vdt*. For *inter-vdt*, we assemble multiple work-items (the basic unit of computation for OpenCL kernels) to fill up a vector. In other words, *inter-vdt* is based on work-items merging, which is similar to *SLP*. This technique merges multiple neighbouring work-items as a new work-item with a (*VL*×) coarser granularity. This *thread coarsening* technique is used by Yi Yang et. al. to enhance data sharing [92, 172]. In our work, we merge work-items, and store operands using vector data. Thus, all the data elements are manipulated in the form of vectors.

As for *intra-vdt*, we focus on vectorizing the work performed a work-item. *Intra-vdt* is based on loop unrolling (much like the traditional vectorization). Specifically, we unroll the loop of an OpenCL kernel for *VL* times, and rewrite the scalar computation

[2] We will use `float` and its vector data types across the chapter.

Listing 4.1: scalar kernel

```
1  #define N 32
2
3  __kernel void native(const __global float * in, \
4    __global float * out){
5
6    int idx = get_global_id(0);
7    float val = 0.0;
8    for(int i=0; i<N; i++){
9      val += in[idx+i];
10   }
11   out[idx] = val;
12 }
```


into its vector form. Note that, with *intra-vdt*, we need to deal with the loop remainder when the loop count is not a multiple of *VL*; it is the same for *inter-vdt* when the number of work-items in the first dimension[3] is not a multiple of *VL*.

4.1.3. CODE TRANSFORMATIONS

In this section, we describe the transformations to be made when using vector data types. To make clear how both *inter-vdt* and *intra-vdt* work, we use a running example, shown in Listing 4.1. The example calculates the sum of *N* input data elements and stores the result.

APPLYING *Inter-vdt*

We use the following steps to translate an OpenCL kernel to its vectorized format by merging work-items.

- Preserve the kernel function definition (function name, arguments and data types), and control flow statements (e.g., `if` or `for` statements).
- Re-calculate the work-item index in the first dimension (i.e., times *VL* in the x direction).
- Duplicate declarations and statements *VL* times.

  We re-name variables with unique identifiers, still derived from the original name, e.g., use a 'name_counter' scheme. The replication procedure starts with output data. By analyzing the data dependence, we recursively build a statement tree, in which each node is duplicated. Furthermore, we need to type-cast constant numbers to a vector format as well.
- Duplicate the writing-back statements. Make sure that we write the results to the right buffer.
- Replace the duplicated scalar data with vector data.

  When the kernel calls a subroutine, we also need to provide a vectorized version of the subroutine. When data accesses are contiguous, we use the built-in functions–

[3] Work-items are arranged in at most three dimensions and we merge work-items in the first dimension.

Listing 4.2: *Inter-vdt*: duplication

```
#define N 32
#define VL 2
__kernel void inter_1(\
  const __global float * in, \
  __global float * out){

  int idx = get_global_id(0) * VL;
  float val_0 = 0.0;
  float val_1 = 0.0;
  for(int i=0; i<N; i++){
    val_0 += in[idx+i+0];
    val_1 += in[idx+i+1];
  }
  out[idx+0] = val_0;
  out[idx+1] = val_1;
}
```

Listing 4.3: *Inter-vdt*: with VDT

```
#define N 32
#define VL 2
__kernel void inter_2(\
  const __global float * in, \
  __global float * out){

  int idx = get_global_id(0)*VL;
  float2 val = (float2)0.0f;
  for(int i=0; i<N; i++){
    val += vload2(0, &(in[idx+i]));
  }

  vstore2(val, 0, &(out[idx]));
}

.
```

4

vstoren and *vloadn*. When the memory accesses are scattered, we need to assemble/disassemble the individual data elements into/from a vector.

The vectorized form of the original kernel (Listing 4.1) is shown in Listing 4.3. To this end, two neighbouring work-items are merged into one and the instance is duplicated (Listing 4.2). Thereafter, the duplicated statements are replaced with vector data types and the data access operations are replaced with *vloadn* and *vstoren* (shown in Listing 4.3). Finally, we need to change the size of *NDRange* in the host program (not shown here).

APPLYING *Intra-vdt*

Here we show the rules of translating an OpenCL kernel to its vectorized format using loop unrolling.

- Preserve the kernel function definition (function name, arguments and data types). Different from *inter-vdt*, the work-item index calculation should also be preserved.
- Unroll the *for* loop *VL* times (e.g., *VL=2*).
- Apply reduction for the calculation of the final result (e.g., a sum).
- Preserve the data writing part.
- Replace the scalar operations with vector operations.

When using *intra-vdt*, we do not need to make any changes on the host program. The transformations of the scalar kernel (Listing 4.1) based on these rules are shown in Listing 4.4 and Listing 4.5. By unrolling the loop twice, we get two copies of the loop. Instead of changing the task granularity of a work-item, we explicitly unroll the inner loop. In this way, we get two copies of partial results and we need to perform reduction on them.

To summarize, *inter-vdt* can be applied to any OpenCL kernel code while *intra-vdt* is only applicable to kernels with loops. Furthermore, *inter-vdt* requires a change in the host program, i.e., to reduce the number of work-items by *VL*. The two approaches

Listing 4.4: *Intra-vdt*: loop unrolling

```
#define N 32
#define VL 2
__kernel void intra_1(\
  const __global float * in, \
  __global float * out){

  int idx = get_global_id(0);
  float val_0 = 0.0;
  float val_1 = 0.0;
  for(int i=0; i<N; i=i+VL){
    val_0 += in[idx+i+0];
    val_1 += in[idx+i+1];
  }

  out[idx] = val_0 + val_1;
}
```

Listing 4.5: *Intra-vdt*: with VDT

```
#define N 32
#define VL 2
__kernel void intra_2(\
  const __global float * in, \
  __global float * out){

  int idx = get_global_id(0);
  float2 val = (float2)(0.0);

  for(int i=0; i<N; i=i+VL){
    val+=vload2(0,&(in[idx+i+0]));
  }

  out[idx] = val.x + val.y;
}
.
```


– *inter-vdt* and *intra-vdt* – are orthogonal, but not exclusive. In other words, they can both be applied individually or together on the same kernel, with different performance effects.

4.2. EXPERIMENTAL SETUP

In this section, we introduce the benchmarks and the devices used in the experiments.

4.2.1. SELECTED BENCHMARKS

To evaluate the performance effects of using VDTs, we use two types of benchmarks: micro-benchmarks and macro-benchmarks. The micro-benchmarks includes `flops` and `bandwidth`, which measure the computational capability and memory access capability of the devices, respectively. The macro-benchmarks include four typically used kernels. The kernels and their datasets are listed in Table 4.1. Note that the datasets have been selected to be much larger than the last-level cache of the devices (shown in Table 4.2). For each benchmark, we start with a scalar version, and transform it into a format using vector data types $vecn$ ($n \in \{2,4,8,16\}$) with the approaches proposed in Section 4.1. We run each benchmark 20 times and obtain the mean kernel execution time (the reciprocal of performance). The normalized performance is the ratio between the performance of the scalar version to that vector versions. Hence, when the number is larger than 1, using VDTs is beneficial; otherwise, it leads to a performance loss.

Table 4.1: The selected benchmarks

Macro-benchmark	Acronym	Dataset
Matrix Multiplication	MM	2048x2048
Image Convolution	IC	8192x8192x16
Black Scholes	BS	8192x8192
Successive Over-Relaxation	SOR	8192x8192

4.2.2. PLATFORMS AND DEVICES

We have tested platforms and devices from different vendors as shown in Table 4.2. We have selected Tesla K20 and Tesla C1060 from NVIDIA. The first one is based on the Kepler architecture and has 2496 CUDA cores in total. For comparison, we use NVIDIA C1060 that has no on-chip caches. From AMD we have selected Tahiti HD7970. The Graphics Core Next (GCN) architecture in Tahiti is a significant change from the traditional VLIW design on old GPUs. Each of the 32 compute units contains a scalar unit and 4 vector units. From Intel we have selected two processors: a multicore CPU (Sandybridge) with dual hexa-cores and a co-processor Xeon Phi 5110P with 60 (512-bit) SIMD cores.

Table 4.2: Platforms and Devices.

	C1060	K20	Phi-5110P	E5-2620	HD7970
Host	Intel Core i7 920	Intel Xeon E5620	Intel Xeon E5-2620	Intel Xeon E5-2620	Intel Xeon E5620
Host OS	UBUNTU v11.10	CentOS v6.2	CentOS v6.2	CentOS v6.2	CentOS v6.2
Device	NVIDIA Tesla C1060	NVIDIA Tesla K20m	Intel Xeon Phi 5110P	Intel Xeon E5-2620	AMD HD7970
GCC	v4.6.1	v4.4.6	v4.4.6	v4.4.6	v4.4.6
OpenCL	CUDA v5.5	CUDA v5.5	Intel OCL SDK v3.0	Intel OCL SDK v3.0	AMD APP v2.9

4.3. VDT EXECUTION MODEL

To investigate VDT execution model(s) on a range of processors from different vendors, we have designed two OpenCL microbenchmarks–`bandwidth` and `flops`. In `bandwidth`, each work-item loads a unique data element from global memory, performs a `mad` (multiply-and-add) operation, and stores the new value back to the global memory (shown in Listing 4.6). By using *inter-vdt*, we obtain a vectorized version of the code shown in Listing 4.7, where n is the vector length ($n \in \{2,4,8,16\}$). Different from `bandwidth`, `flops` repeats the arithmetic instruction (Line 9) for 32000 times so that the computation dominates the kernel execution time. We show the normalized performance in Figure 4.1.

Listing 4.6: The scalar version

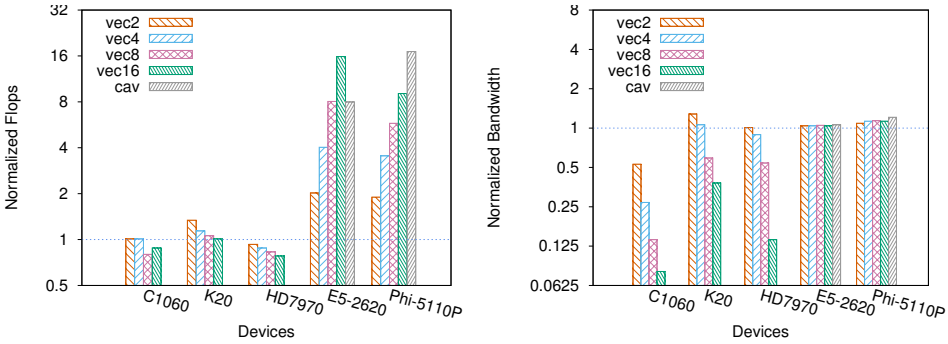
```
1  __kernel void bandwidth(\
2    const __global float * in, \
3    __global float * out){
4
5    int idx = get_global_id(0);
6    float alpha = 3.0;
7    float beta = 0.98;
8    float v = in[idx];
9    v = alpha - beta * v;
10   out[idx] = v;
11 }
```

Listing 4.7: The vector version

```
1  __kernel void bandwidth(\
2    const __global float * in, \
3    __global float * out){
4
5    int idx = get_global_id(0)*n;
6    floatn alpha = 3.0;
7    floatn beta = 0.98;
8    floatn v = vloadn(0, &(in[idx]));
9    v = alpha - beta * v;
10   vstoren(v, 0, &(out[idx]);
11 }
```

4.3.1. EXECUTION MODEL ANALYSIS

According to the core organization, we divide processors into two groups: scalar-core and vector-core. The former features with scalar cores, which can run independently



(a) Flops (b) Bandwidth

Figure 4.1: Normalized performance with `flops` and `bandwidth`.

(but not necessarily simultaneously). C1060, K20 and HD7970 fall into this category. Such cores on NVIDIA GPUs are also coined as *SIMT* cores [118]. In contrast, each core of vector-core processors is a *SIMD* core, where the SIMD lanes run in a strict lock-step manner. E5-2620 and Phi-5110P are examples of such processors.

For scalar-core processors, it is natural to run a work-item (or a thread) on a scalar core (Figure 4.2a), and run a group of work-items on a group of scalar cores (an SMX on NVIDIA GPUs or a vector unit on AMD GCN GPUs). When using VDTs in kernels, the vector components are accessed one by one (in a loop style). Figure 4.2b shows each vector has 4 components ($VL = 4$), and each work-item needs 4 iterations to access all the components of the vector data. This serialization execution model introduces extra overheads, as shown in Figure 4.1a. Using VDTs, the flops number is no better than that of the scalar code on the scalar-core processors (C1060, K20, and HD7970). An exception can be observed on K20: using $vec2$ or $vec4$ leads to slightly better performance.

Elements in a vector are guaranteed to be stored in contiguous storage locations. Thus, $D00$ and $D01$ are stored contiguously in memory space, while $D00$ and $D04$ are not (Figure 4.2a and 4.2b). When using VDTs on scalar-cores, each core in a CU will gather a data component from its corresponding vector, leading to un-coalesced memory accesses. Figure 4.1b shows the bandwidth decreases over VL on C1060, K20, and HD7970. To summarize, using VDTs on scalar-core processors often leads to a performance decrease due to the serialization overheads and unfriendly memory access patterns. Therefore, VDTs should not be used on the scalar-cores.

On vector-core processors, each core has a vector of lanes running in a lock-step manner (4 lanes in Figure 4.2c and 4.2d). A work-item with scalar data can only utilize a part of the lanes (shaded in Figure 4.2c) and leaves others unused. In this situation, we need to use vector data types to fill up the processing lanes. As shown in Figure 4.2d, we use $vec4$ so that all the lanes are occupied. Figure 4.1a shows that using vector data types, the performance can be increased significantly on the vector-cores (E5-2620 and Phi-5110P). In particular, the flops number increases over the vector length. Intuitively, the maximum flops can be achieved with $float8$ (256 bits) on E5-2620. However, we ob-

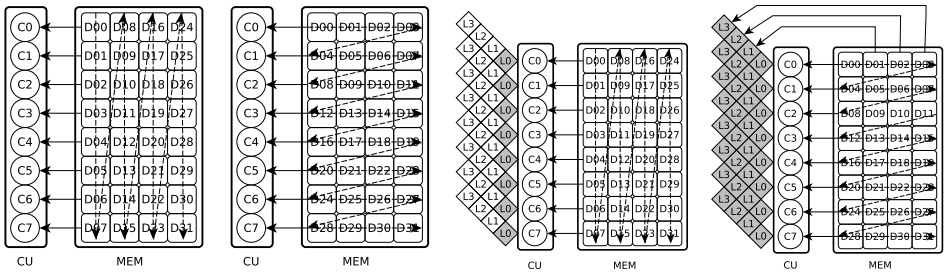


Figure 4.2: Execution model: (a) Scalar data on scalar cores, (b) Vector data on scalar cores, (c) Scalar data on vector cores, and (d) Vector data on vector cores (C: Processing Cores, CU: Compute Unit, MEM: Memory, L: Lane, D: Data element). Note that the dashed lines indicate how data elements are stored in MEM.

serve a further improvement when using $float16$. This is because, when using $float16$, we have more independent instructions to fill the bubbles between two dependent instructions.

Our `bandwidth` tests on vector-cores show that there is no performance change by using vector data types (Figure 4.1b). This is because both ways can utilize spatial data locality. Therefore, using VDTs benefits the computation and the overall performance, and it is mandatory to explicitly use VDTs for vector-core processors.

4.3.2. COMPILER-LEVEL ANALYSIS

Intel provides a compiler-aided vectorization (*cav*) module in its OpenCL implementation. This module packs several work-items and executes them with SIMD instructions. The basic idea is to transform scalar data type operations on adjacent work-items into an equivalent vector operation. This enables us to benefit from the vector units in vector-core processors without writing explicit vector code. In this section, we compare the performance of *cav* with that of explicitly using VDTs (and we further show the performance effects of using *cav* on macro-benchmarks in Section 4.4).

From Figure 4.1a, we see that *cav* can achieve similar performance to that of explicit vectorization (with $float8$) on E5-2620. On Phi-5110P, however, we observe that the implicit vectorization module does a better job than the explicit vectorization. In particular, the cav module obtains almost 100% vectorization efficiency, while it is only around 60% by using VDTs. We believe this is due to the inefficient mapping of vector data to the underlying hardware. Therefore, using compiler-aided vectorization is a good candidate replacing the VDTs usage on the vector-core processors. At the time of writing (May 2014), AMD has not released a *cav* module. To fully exploit AMD processors of vector-core style (e.g., AMD CPUs and AMD VLIW GPUs), explicit use of VDTs is still required.

4.3.3. LESSONS LEARNED

Based on the core features, programmers should decide to use vector data types at the right time. NVIDIA GPUs (e.g., C1060 and K20) and the recently released AMD GPUs

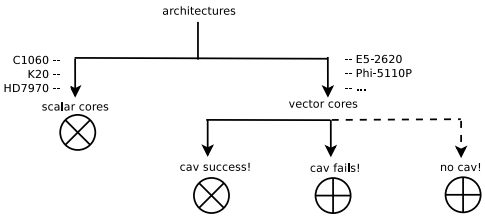


Figure 4.3: Architectures, compiler and VDTs (⊗ represents we do not need VDTs, while ⊕ means we need explicitly use vector data types).

(e.g., HD7970) use scalar cores. It is natural to map work-items to the same number of scalar cores. Therefore, using VDTs on OpenCL kernels is not mandatory on such processors. On architectures with SIMD cores (e.g., E5-2620 and Phi-5110P), vectorization is needed to fully utilize the SIMD lanes. The job can be done either automatically by compilers (Section 4.3.2) or manually by programmers (e.g., using VDTs). When the automated compiler-aided vectorizer fails (which can seen from compiling messages), programmers need to manually use VDTs. This conclusion is summarized in Figure 4.3.

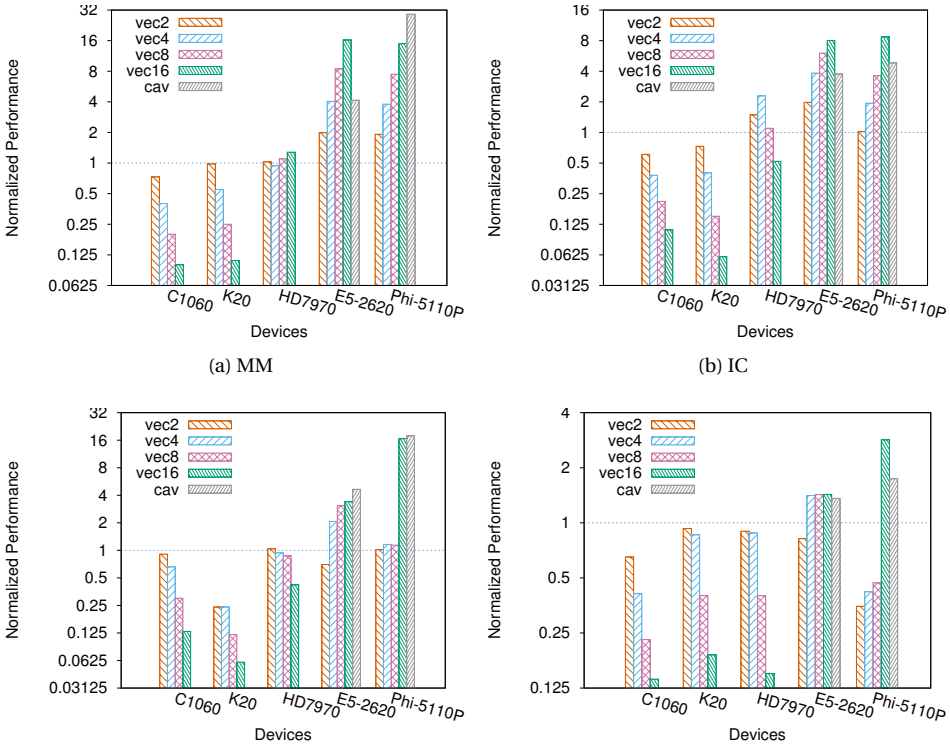
4.4. INTER-VDT PERFORMANCE IMPACT ON MACRO-BENCHMARKS

In this section, we discuss the performance effects of using *inter-vdt* on our target kernels (Table 4.1). Based on the execution models and lessons learned in Section 4.3, we further analyze the performance changes when using different vector data types.

We show the normalized performance of these kernels in Figure 4.4. The performance effects of using VDTs vary over devices, applications and vector lengths. As indicated in the performance models [167], the overall performance is determined by two factors: computation and memory access. Thus, we will analyze the performance effects of using VDTs by looking into these two factors. Based on the execution models described in Section 4.3, using VDTs will accelerate the computation part on the vector-core processors, and should bring no changes on the scalar-core processors.

As for memory access, using VDTs changes memory access patterns (MAPs) and thus might impact the memory bandwidth. Therefore, we give a detailed description of memory accesses before and after using VDTs for each kernel. Typically, we describe MAPs at two levels: work-item level and work-group level [40, 42]. In Figure 4.5, each work-item works on one element of the output data, and needs to read the corresponding elements from the input data ((I) one row, (II) one column, (III) one block of elements, or multiple elements in random formats, shown in Figure 4.5a). At the work-group level, neighbouring work-items access the same or separate elements simultaneously, also showing access patterns. When introducing *inter-vdt*, MAPs change, i.e., each work-item works on *VL* elements of output data, and its input data is expanded, as shown in Figure 4.5b ($VL = 2$).

In the following sections, we first analyze the changes in memory access patterns with VDTs. To quantify the bandwidth changes before and after using VDTs, we develop a benchmark for each memory access pattern. By comparing the trends of overall performance with that of bandwidth numbers, we investigate how using VDTs affects per-

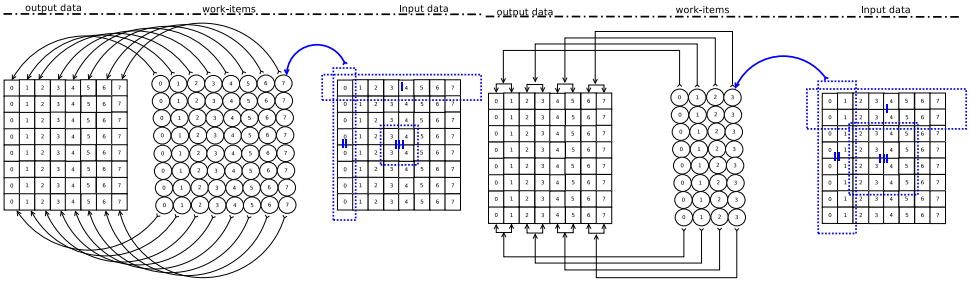


(a) MM (b) IC

(c) BS (d) SOR

Figure 4.4: Normalized performance for the macro-benchmarks: MM, IC, BS, and SOR.

formance.



(a) w/o VDT (b) w/ VDT

Figure 4.5: Mapping work-items to data. Circles represent a grid of work-items, squares represent input/output data arrays. There are 64 (8x8) work-items altogether, and each work-item computes one element from the output data. When using *inter-vdt*, each work-item works on two output elements.

4.4.1. MATRIX MULTIPLICATION

MM ($C = A \times B$) has two input matrices and thus two memory access patterns: *MAP MMA* and *MAP MMB*. For *MAP MMA*, each work-item accesses a whole row of elements indexed by its row index[4] (see Figure 4.6a). At the work-group level, neighbouring work-items will access the same data element at a time. When using *inter-vdt*, the number of work-items is reduced *VL* times (Figure 4.6b), and each work-item will access the same element for *VL* times, leading to register data sharing. Therefore, we can achieve linear bandwidth improvements on all the selected devices, as shown in Figure 4.7a.

For *MAP MMB*, each work-item loads a whole column of data elements indexed by its column index (see Figure 4.6c). At the work-group level, neighbouring work-items will access spatially close data elements at a time. When using *inter-vdt*, the number of work-items is also reduced *VL* times (Figure 4.6d). Each work-item will access *VL* columns of data elements, and the distance with its neighbouring work-items is *VL*, rather than 1. This leads to un-coalesced memory accesses on C1060 and K20 and the observed bandwidth decreases (shown in Figure 4.7b). However, the bandwidth shows no changes with VDTs on HD7970. On E5-2620 and Phi-5110P, the bandwidth increases with *VL*. Figure 4.6c shows each work-item accesses data elements in the column-major manner while data is stored in the row-major manner. When requesting data from memory space, a cache-line of data elements are serviced. Typically, the cache-line size is 64 bytes, i.e., 16 single-precision data elements. Thus, using a longer vector can use more data elements in a cache-line and brings us a larger memory bandwidth. To conclude on Figure 4.7b, GPUs prefer $VL = 1$ while the vector-core processors perform better with longer vectors.

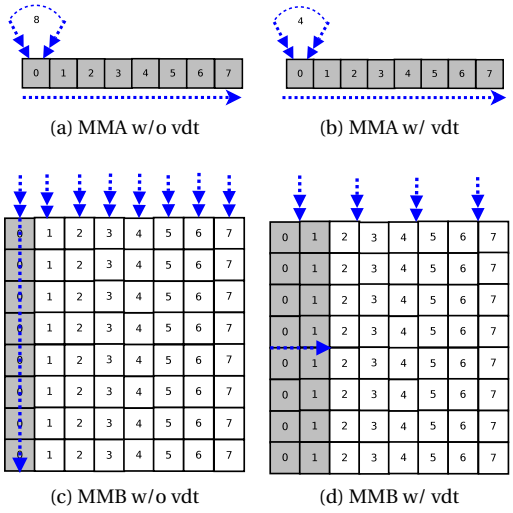
As we can see from Figures 4.4a, 4.7a, and 4.7b, the overall performance shows similar trends with the bandwidth of `Matrix B`. Thus, the overall performance of `MM` is limited by accessing this matrix.

4.4.2. IMAGE CONVOLUTION

The memory access pattern of `Image Convolution` is shown in Figure 4.8a. Each work-item reads a block (2×2) of data elements from the input data. At the work-group level, neighbouring work-items access spatially contiguous data elements. When using *inter-vdt*, the data block used by a work-item is expanded (e.g., a 2×3 data block when $VL = 2$). In this way, using *inter-vdt* enhances data sharing [172]. Further, the distance between neighbouring work-items is *VL*, not 1.

In Figure 4.7c, we can see that the bandwidth decreases over *VL* due to the un-coalesced memory access on C1060, K20, and HD7970. E5-2620 and Phi-5110P show similar trends: the bandwidth is larger when using a wider vector. Thus, these two processors can benefit from the data reuse enhancement from using *inter-vdt*. In particular, using $float16$ can achieve the largest bandwidth. Therefore, the optimal *VL* is not necessarily the machine vector length, because the performance gain is due to enhanced data sharing. We can observe that the application presents similar trends between the overall execution time and the bandwidth in Figure 4.4b and Figure 4.7c. Therefore, the performance of `Image Convolution` is limited by the memory access.

[4] The row index of a work-item is its index in the second dimension and the column index of a work-item is its index in the first dimension.



(a) MMA w/o vdt (b) MMA w/ vdt

(c) MMB w/o vdt (d) MMB w/ vdt

Figure 4.6: MAP MMA and MMB (*VL=2*)

4.4.3. BLACK SCHOLES

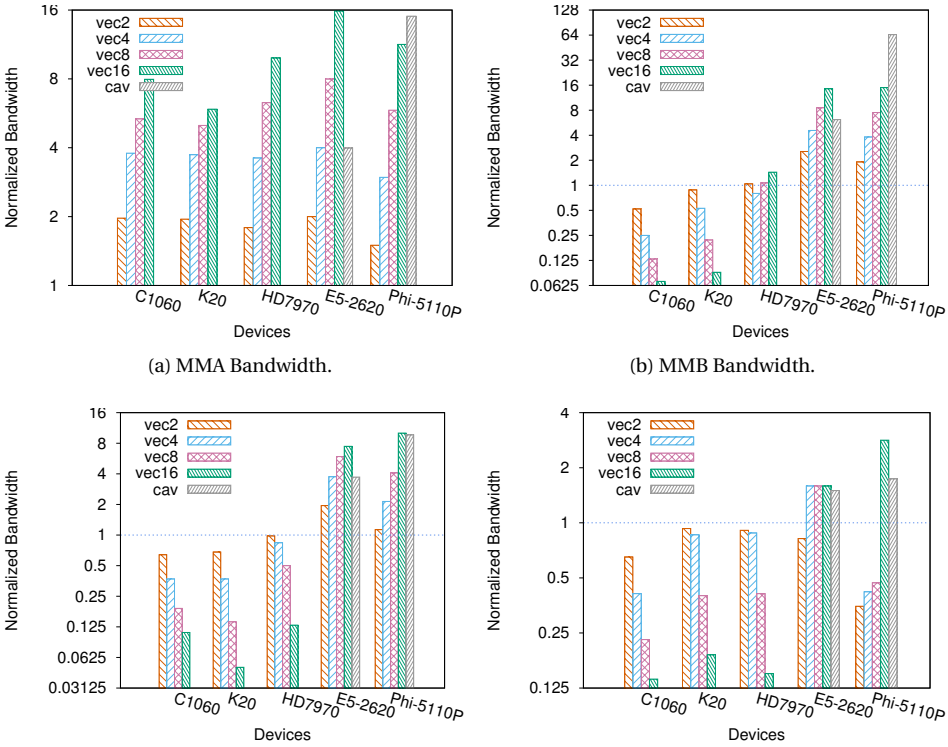
Figure 4.9a shows the memory access pattern presented in `Black Scholes`. We see that each work-item reads one element from the input data, and writes the results back when finishing computation. At the work-group level, neighbouring work-items access physically close elements ($VL = 1$) at a time. When using *inter-vdt*, each work-item will access *VL* physically contiguous data elements ($VL = 2$ in Figure 4.9b); the distance between neighbouring work-items becomes *VL*, i.e., non-coalesced memory accesses. This pattern is the same as the one in `bandwidth`, and the results have already been seen in Figure 4.1b.

The overall performance (Figure 4.4c) presents a totally different trend with that of memory bandwidth (Figure 4.1b). This is because the overall performance of `Black Scholes` is limited by computation, rather than memory accesses. Due to better utilization of the SIMD units, we expect improved performance on vector-core processors which can be confirmed in Figure 4.4c. However, due to the overhead of using VDTs, the NVIDIA GPUs and the AMD GPU see a performance drop.

4.4.4. SOR

The memory access pattern of `SOR` is shown in Figure 4.10a. Each work-item will access its four neighbours (left, right, top, bottom). Neighbouring work-items access spatially close data elements at a time. When using VDTs, the data elements accessed by one work-item become *VL* times as many as before (the 8 data elements shaded in Figure 4.10b for $VL = 2$). Again, the access distance between neighbouring work-items is *VL*.

In Figure 4.7d, the performance shows a similar trend with the other kernels on C1060, K20, and HD7970, i.e., the bandwidth decreases over *VL*. On the vector-core pro-



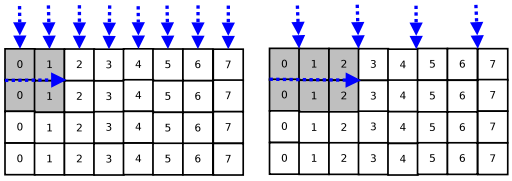
(a) MMA Bandwidth.

(b) MMB Bandwidth.

(c) IC Bandwidth.

(d) SOR Bandwidth.

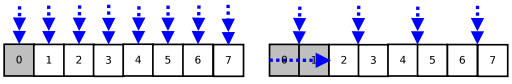
Figure 4.7: Bandwidth comparison with VDTs.



(a) w/o VDT

(b) w/ VDT

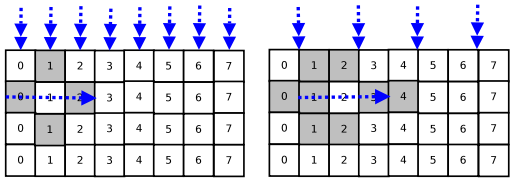
Figure 4.8: IC MAP changes (*VL=2*).



(a) w/o VDT

(b) w/ VDT

Figure 4.9: BS MAP changes (*VL=2*)



(a) w/o VDT (b) w/ VDT

Figure 4.10: SOR MAP changes (*VL=2*)

cessors (E5-2620 and Phi-5110P), the bandwidth gain is smaller than that of IC, because there is little data reuse with *inter-vdt*. The overall performance (Figure 4.4d) shows similar performance trends with that of the bandwidth (Figure 4.7d). Therefore, the performance of SOR is limited by memory accesses.

4.4.5. LESSONS LEARNED

To summarize, using *inter-vdt* leads to significant changes in the memory access patterns, and thus impacts the bandwidth. These effects are represented in Table 4.3. Overall, we see that using VDTs indeed changes bandwidth which varies with processors and MAPs. Specifically, all the selected devices can benefit from using *inter-vdt* on MAP MMA, due to register data reuse. For the other MAPs, using *inter-vdt* brings a bandwidth decrease on GPUs (C1060, K20, and HD7970), because of the un-coalesced memory accesses introduced by this vectorization. For MAP MMB and MAP IC, using larger vector length on vector-core processors gives a larger bandwidth due to better cache utilization. Furthermore, we have found that the optimal *VL* is not necessarily equal to the machine vector length, but it depends on the memory access patterns. In other words, using the machine vector length might produce a sub-optimal performance.

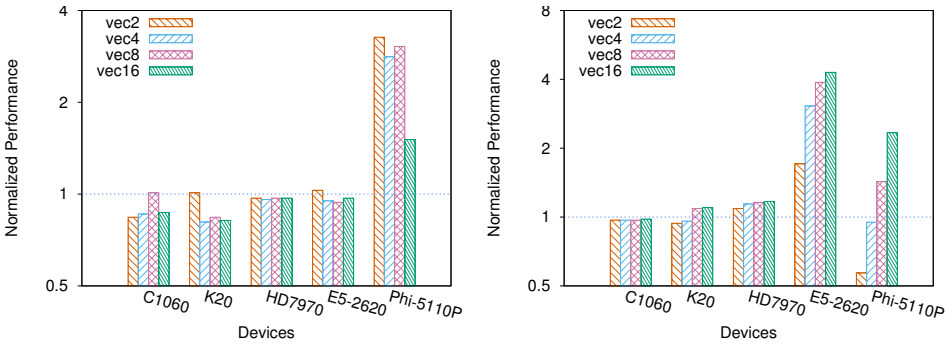
Table 4.3: Bandwidth changes from using VDTs
(↗: increase, ↘: decrease, ⇋: similar, ⇝: the bandwidth change depends on the vector length).

		MMA	MMB	IC	BS	SOR
scalar-core	C1060	↗	↘	↘	↘	↘
	K20	↗	↘	↘	⇝	↘
	HD7970	↗	⇋	↘	↘	↘
vector-core	E5-2620	↗	↗	↗	⇋	⇝
	Phi-5110P	↗	↗	↗	⇋	⇝

In Figure 4.4, we see that *cav* can achieve similar performance to that of explicitly using VDTs. This observation shows the compiler-aided vectorization is a good candidate for explicit vectorization in terms of performance. Among the four kernels, only BS is compute-bound. Its overall performance trend (Figure 4.4c) roughly matches the data derived from the rules in Figure 4.3. Therefore, given an OpenCL kernel, we can analyze its performance effects of using VDTs from memory access (Table 4.3) and computation (Figure 4.3).

4.5. INTRA-VDT PERFORMANCE IMPACT ON MACRO-BENCHMARKS

For kernels (`MM` and `IC`) with loops, we can use *intra-vdt* by unrolling the loops and then using vector data types. According to the vector length, we need to unroll loops for VL times ($VL \in \{2,4,8,16\}$). Figure 4.11 shows the normalized performance with VDTs on `MM` and `IC`. Since the compiler-aided vectorization (*cav*) is different from *intra-vdt*, we do not compare the results with *cav*. Instead, we only show the performance of different vectorized versions.



(a) MM (b) IC

Figure 4.11: Normalized performance for the macro-benchmarks: MM and IC.

Scalar-core processors see a slight change in performance from using vector data types (Figure 4.11). Specifically, using vector types on scalar-cores leads to an upto 20% performance drop for `MM`, while the performance changes for `IC` vary over devices. Moreover, we see that the vectorized `MM` runs faster on Phi-5110P and the vectorized `IC` runs faster on both E5-2620 and Phi-5110P. Different from *inter-vdt*, *intra-vdt* does not change the memory access patterns. Thus, the bandwidth should be similar before and after using *intra-vdt*. The computation part is able to use the SIMD cores with *intra-vdt*. Thus, the computation-intensive kernels will be accelerated.

4.6. PERFORMANCE PORTABILITY DISCUSSION

For the optimal performance of a kernel, we need a specialized VDT version and decide (1) whether or not to use VDTs, and (2) to select a right VL. This will lead to a large number of code variants, and inevitably challenges performance portability. To attempt to achieve portable performance, we propose the following two solutions.

The first is to use parametric code when developing applications. This parametric code can be obtained by using a vector template, and it is specialized into a code variant suitable for the target platform during compiling time. We do not have to specify the template content until we know the details of the target platform. This solution has been demonstrated in Listing 4.6 and Listing 4.7. We have used a vector template (`floatn`) in the kernel code. Once we know the target architecture is Phi-5110P, for example, we specify `floatn` as `float16` (note that the vector register on Phi-5110P is of 512-bits). In this

way, we can customize the kernel code based on the underlying architecture. However, this solution needs manual manipulation of vectors.

Second, we recommend programmers to start by using scalar data types in OpenCL. Then, for scalar-core processors, there should be no performance loss; for vector-core processors, the compiler will attempt to pack the work-items so as to fully utilize the SIMD cores. As programmers have already indicated parallelism by writing an explicitly parallel program (in OpenCL), it is easier for the compiler to perform vectorization (i.e., mapping parallel tasks to SIMD hardware) than doing the traditional vectorization. In case auto-vectorization fails, or when using AMD's or ARM's compilers, which currently have no implicit vectorization module, programmers should manually use vector data types. As soon as all OpenCL implementations will support auto-vectorization (if ever), our evaluation needs to be performed for these versions as well. Once these auto-vectorization modules become efficient enough, the vector data types will become 'redundant' and could be deprecated from the OpenCL specification. In a nutshell, this solution relies on the *cav* module.

4.7. RELATED WORK

Using vector data types is a high-level vectorization approach that relies on compilers to map the high-level vectors to the underlying hardware. In this section, we give a brief overview of vectorization, and we discuss related work on auto-vectorization for OpenCL, thread-coarsening, and memory access patterns.

Vectorization is a well-studied topic. With the advent of vector computers, there has been an increased interest in making vector operations available. In [3], Allen and Kennedy present a translator to transform programs from FORTRAN to FORTRAN 8x. In [26], the authors describe a research project that automatically optimizes multimedia programs by exploring loop level parallelism for the SUN VIS instruction set. They employ a two-phase source-to-source optimization strategy. Since the year of 2000, multimedia extensions have been adopted by most major computer vendors such as MMX/SSE/AVX for Intel processors, 3DNow! for AMD processors, VIS for SUN SPARC, AltiVec/VSX for POWER, NEON for ARM. These processing units can be characterized as SIMD processors operating on packed fixed-length vectors. Programmers can use in-line assembly code, intrinsic functions, or library routines to exploit these extensions [131]. The individual vectorization issues such as addresses alignment, mixed data types, and multiple scopes for extracting parallelism have been addressed in [38, 87, 170, 171]. Generating code for vector hardware heavily relies on target-specific manual optimizations, which lack portability. This is a painful deficiency in view of the diversity and constantly evolving nature of vector architectures. To this end, Dorit Nuzman et al. have presented a cross-platform vectorizer [110, 111]. The goal is to auto-vectorize once and run everywhere. In [63], the authors have proposed a retargetable SIMD code optimization framework that is integrated into an industrial retargetable C compiler.

Despite all these efforts, only a few loops from real applications can be successfully vectorized [94]. Auto-SIMD compilers rarely work when applied to real codes, mainly because compilers fail to recognize a parallel loop out of a sequential program [127]. As a solution, using a new language such as CUDA or OpenCL is promising due to its explicit specification of parallelism. Using such a language saves the effort of looking for

vectorization candidates. Several studies use compiler-aided vectorization for OpenCL kernels. For example, the Intel's OpenCL compiler has an implicit vectorization module based on LLVM IR [69]. This module has a sequence of LLVM transformations and analysis passes, and generates code for multiple Intel devices. In particular, it widens a single element into a vector of elements using a 'packetizer'. In [79], Karrenberg and Hack present a language- and platform-independent code transformation that performs whole-function vectorization for data parallel programs. The output code can utilize the intra-core parallelism provided by the SIMD instruction set. However, developing a vectorizing module is not the main focus of this work. Instead, we aim to compare the performance of implicit vectorization with that of explicit vectorization (i.e., using VDTs). Therefore, we must evaluate the contribution of the automatic vectorization module such as these discussed in [69, 79] for performance portability.

Thread-coarsening is an optimization technique that increases the per-thread task granularity. In [92], Alberto Magni et al. evaluate the thread-coarsening effects across a range of devices using a source-to-source OpenCL compiler based on LLVM. They also use statistical regression to analyse and explain program performance in terms of hardware-based performance counters. In [172], Yi Yang et al. use such a technique to exploit data sharing between threads. This optimization technique is used in our work, but it serves a different purpose, i.e., merging threads to use vector data types (*inter-vdt*). In our case, the coarsening factors are the vector lengths provided by the OpenCL specification.

Previous work on memory access patterns (MAPs) is also relevant to this work. MAPs have been explored to maximize data locality for the traditional single-core processors. The classical approaches are either based on array restructuring [90], or based on loop transformation [168]. Recently, these approaches have been extended to support many-core architectures [72], [25]. In our work, we analyze the MAP changes brought by using vector data types to give an in-depth analysis of the performance effects with VDTs.

To summarize, OpenCL introduces vector data types so that programmers can manipulate vectors like intrinsics, but at a higher level. In this chapter, we investigate how OpenCL and its VDTs deal with vectorization on different processors. To the best of our knowledge, this is the first work that systematically studies the usage of vector data types and vectorization in OpenCL.

4.8. SUMMARY

OpenCL has existed since late 2008. However, the usage of vector data types has not been investigated in the context of performance and portability. In this work, we propose two orthogonal approaches to use vector data types (*inter-vdt* and *intra-vdt*) and we apply them on scalar kernels (Q1). After getting the vectorized code, we first investigate the execution model of vector data types by using two micro-benchmarks (Q2). Further, with macro-benchmarks, we analyze the performance changes that occur when using vector data types, from the perspective of memory access patterns (Q3). We found that (1) scalar-core processors run VDTs in a different and unexpected way than vector-core processors, (2) using VDTs changes the memory access patterns and memory bandwidth, (3) the performance effects depend on the compilers-aided vectorization (e.g., Intel compiler performs at least as good as explicitly using VDTs on Intel processors), and

(4) *inter-vdt* brings a larger performance change than *intra-vdt*. Based on the lessons learned, we discuss how to improve performance portability with two potential solutions (Q4).

In the future, we would like to investigate *cav* modules systematically. In the long run, we plan to implement a generic *cav* module on vector-cores.

5

QUANTIFYING THE PERFORMANCE IMPACTS OF USING LOCAL MEMORY

Modern multi/many-core processors like GPUs use programmer-managed *scratch-pad memories (SPM)*. We have introduced the advantages of using SPMs in Chapter 1. In Chapter 3, we have seen that properly using memory hierarchy and in particular, local memory can lead to a performance boost. In this chapter, we investigate the performance impact of local memory usage systematically.

OpenCL [151], recognizes SPMs under the name of *local memory* in its conceptual device architecture [1]. Eager to be part of the development and deployment of the common programming model for many-cores, many vendors have implemented OpenCL and local memory on top of their hardware and software stacks. For example, NVIDIA maps local memory onto the on-chip SPM, while the cache-only processors [2] such as the multicore CPUs map it to the off-chip memory [61].

Due to architectural disparities and, in particular, the differences in implementing local memory, programmers often use the *trial-and-error* approach to enable local memory and evaluate its efficiency: taking a naive kernel, they translate the code into an "optimized" version that uses local memory and then measure its impact. This is a time-consuming process, as programmers have to address, in their OpenCL code, challenges like (1) geometry mismatches, (2) work-items masking and binding switches, and (3) inefficient local memory organization [44]. Similarly, for architectures where using local memory is not recommended, programming effort is often spent on removing the

This chapter is based on our work published in the Proceedings of MuCoCoS 2013 [40] and in Scientific Programming [42].

[1] NVIDIA uses the term 'shared memory', while AMD calls it 'local data store'. In this chapter, we use the OpenCL name 'local memory' [151].

[2] Cache-only processors have on-chip caches but no SPMs.

code related to local memory for improved performance. We argue that solving these problems requires a lot of effort to be spent on non-computational and non-functional details of kernels, which hinders productivity. Therefore, we propose a solution to quantify the performance impact of using local memory *before* implementing it. Our analyzer will help sparing a lot of useless programming effort.

Despite common belief [4, 70, 118], the impact of local memory usage on performance is not easy to determine. For example, *data reuse* is a commonly recognized source of performance gains of using local memory [58, 71, 95, 134]. However, data reuse and local memory are not always correlated: data reuse does not automatically lead to a higher local memory efficiency (Section 5.1.1), nor does the lack of data reuse mean lack of performance improvement (Section 5.1.2). Furthermore, in the case of CPUs, the off-chip placement of the local memory makes programmers choose not to use it [70], but properly using it can lead to performance improvement (Section 5.1.3).

In this chapter, *we address the issue of performance unpredictability when using local memory in a two-stage approach: quantification and composition.* For quantification, we develop a benchmark-based approach to quantify the performance impacts of using local memory for 33 memory access patterns (MAPs) in isolation. For each MAP, we generate two types of benchmarks: with and without using local memory. We empirically evaluate these benchmarks on typically used platforms, and record the achieved performance in a *performance database*. In practice, we can obtain the performance benefits of using local memory for a single MAP by querying the database. For composition, we present a set of rules (empirically validated) to determine whether to use local memory or not in the presence of multiple MAPs. The database plus the composing rules will produce an indicator of whether to use local memory for a given application. We name the approach, including the code generator and validator, *Aristotle*[3].

To summarize, we make the following contributions:

- We formalize the benchmark design space and develop a code generator which helps applying our approach on any OpenCL-compliant platform.
- We evaluate the performance impacts of using local memory on a broad category of processors and generate a comprehensive and representative database.
- We design and validate a set of composing rules to determine whether to use local memory in the presence of multiple MAPs.

The chapter is organized as follows: We list three counter-intuitive observations of using local memory in Section 5.1. Our approach is presented in Section 5.2. We extend a mathematical model to describe memory access patterns and derive a set of MAPs in Section 5.3. We explore the design space of using local memory and produce benchmarks using a code generator in Section 5.4. We generate a performance database by running the microbenchmarks on seven typically used platforms in Section 5.5. In Section 5.6 and Section 5.7, we propose and validate a set of composing rules in the presence of multiple MAPs. We present related work in Section 5.8 and we summarize our findings in Section 5.9.

[3] *Aristotle* was named after the muppet *Aristotle* (a blind monster who can make sandwiches) in the American TV series *Sesame Street*. Likewise, we have *ELMO* in Chapter 6 and *Grover* in Chapter 7.

5.1. THREE OBSERVATIONS AS MOTIVATION

Our work is based on the observation that local memory, although perceived as a guarantee of performance gain, does not always behave as such. In this section, we give a more detailed analysis of three types of such behaviors/observations.

5.1.1. DATA REUSE ≠ PERFORMANCE IMPROVEMENT

The occurrence of data reuse is a widely used criterion of moving data from global memory to local memory. However, this statement does not always hold. Table 5.1 shows the memory bandwidth when running NBody [115] on NVIDIA GTX580. We see that although the input data elements are shared by all the threads for NBody, using local memory performs worse than not using it (by around 20%). The performance loss is due to the fact that GTX580 has caches (L1 and L2) that make better use of data sharing than the local memory. Specifically, local memory enables data sharing among work-items within one work-group, while the L1 cache can identify the data sharing within one work-group, and the L2 cache will enable global data sharing on the input data (i.e., among work-groups as well). Additionally, using local memory introduces extra overheads for data movement operations in and out of local memory. Therefore, the caches may "cancel" the performance gains of using local memory.

Table 5.1: Memory bandwidth (GB/s) of NBody where the local memory is allocated dynamically ($LM_{w/o}$ represents the naive kernel, and $LM_{w/i}$ represents the kernel using local memory. We use five datasets each with a different matrix/input size).

	64x64	128x128	256x256	512x512	1024x1024
$LM_{w/o}$	613.50	636.43	646.06	616.42	589.95
$LM_{w/i}$	512.44	495.28	516.04	518.61	520.64
Loss(%)	16.47	22.18	20.13	15.87	11.75

5.1.2. NO DATA REUSE ≠ PERFORMANCE LOSS

Let us consider data movements between local memory and global memory. Suppose we have N compute units and the bandwidth of local memory access is W_l. An application requires D data elements to be moved when using global memory only (with a bandwidth of W_g), and $D^{'}$ data elements to be moved from global memory to local memory (with a bandwidth of $W_g^{'}$). We compute the time of data movement without ($T_{w/o}$) in Equation 5.1 and with local memory ($T_{w/i}$) in Equation 5.2. We see that performance improvement of using local memory comes from two factors: either the decrease of data amount ($D^{'} < D$), and/or the increase of global memory bandwidth ($W_g^{'} > W_g$). Thus, considering data reuse as a must for local memory performance gain is incorrect and will lead to missed opportunities for local memory usage. Taking a straightforward approach for a Matrix Transpose on GPUs for example, the implementation will violate the coalescing constraints on the global memory access. Using local memory, we can ensure coalescing for both input and output memory access, and thus improve the bandwidth.

$$T_{w/o} = \frac{D}{W_g} \tag{5.1}$$

$$T_{w/i} = \begin{cases} \frac{D}{W_l} + \frac{D'}{W'_g} & N = 1 \\ max\left(\frac{D}{W_l}, \frac{D'}{W'_g}\right) = \frac{D'}{W'_g} & N > 1 \end{cases} \tag{5.2}$$

5.1.3. LOCAL MEMORY USE ON CPUS ≠ PERFORMANCE LOSS

At the moment of writing, local memory is allocated within the main memory space of the CPUs (global memory in OpenCL). Thus, it is not recommended to use local memory on CPUs [70]. However, we have found that this does not always hold. Table 5.2 shows the memory bandwidth of a convolution kernel on Intel Xeon E5620 (a dual-socket 4-core processor). We see that using local memory delivers better performance than not using it (around 2× faster). Using local memory on CPUs introduces extra overheads, but it also changes the usage of caches and allows compilers to do specific optimizations for data placed by the users in local memory.

Table 5.2: Memory bandwidth (GB/s) of convolution with and without local memory for six datasets.

	64x64	128x128	256x256	512x512	1024x1024	2048x2048
$LM_{w/o}$	6.81	7.77	7.81	8.06	8.13	8.15
$LM_{w/i}$	12.23	13.81	14.08	14.56	14.70	14.56
$Speedup$	1.80	1.78	1.80	1.80	1.81	1.79

To summarize, these three counter-intuitive observations show that using local memory makes it difficult to predict the performance gain, thus leading to performance unpredictability. Further, our analysis indicates that the unpredictability results from the diversity of architectures/processors and applications.

5.2. THE DESIGN OF ARISTOTLE

Based on all these observations (Section 5.1), we believe that a better understanding of the cases when local memory is useful, and better quantifying its usefulness are equally required. Thus, we propose a hybrid approach to tackle this issue: use MAP modelling to generate microbenchmarks, and use traditional performance measurement to quantify local memory usefulness.

Figure 5.1 shows the Aristotle framework. For all memory access patterns, we generate 2 benchmark kernels: one without local memory, and the other one with local memory. Then we evaluate the benchmarks on typically used many-core processors and generate a performance database. Given a kernel, we identify the MAPs embedded in it and use the composing rules to generate a performing list of whether or not to use local memory on each MAP.

Note that our benchmarks start with memory access patterns (MAPs), which we consider to be models of the input kernels. Our goal is to evaluate the benchmarks empirically, giving accurate information on the benefit of using local memory. Thus, given an

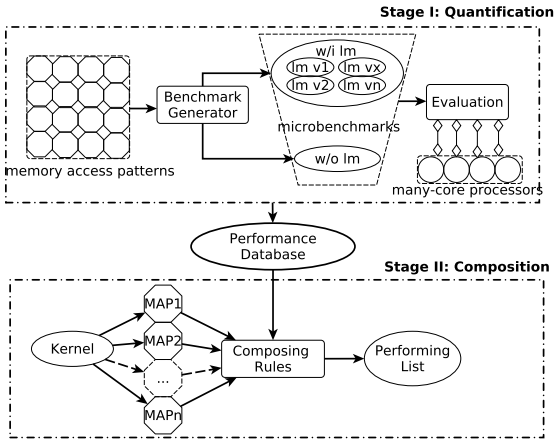


Figure 5.1: Aristotle overview.

5

application MAP and a platform, a simple query in our database can show how using local memory impacts the application performance. Furthermore, the memory access patterns can be manually identified from input kernels [72], or automatically abstracted during run-time [122], and are, for now, outside the scope of this work.

5.3. MAP DESCRIPTION

We express a MAP as a *memory access sequence*, which allows us to represent discrete memory references and loops. Our approach is based on the notation in [72, 90]. We make use of a similar notation, which enables us to study memory access patterns systematically. To keep the number of analyzed MAPs under control, we rewrite the formulation such that we clearly separate the inter-thread and intra-thread parallelism. Specifically, we assume a 2D thread configuration (t_x, t_y), for which we investigate the resulting inter-thread access patterns, and five different intra-thread access patterns, to match the most important MAPs found in real-life applications. Using these limitations, we are able to fully analyze a set of MAPs that are intuitive and cover a large set of real-life applications.

5.3.1. THE NOTATION

According to [72], a memory access sequence $\vec{s}$ can be expressed as a combination of a memory access matrix, $\mathbf{M}$, an iteration vector, $\vec{i}$, and an offset vector, $\vec{o}$. The dependency is presented in Equation 5.3. Note that this notation is applicable to loop nests of arbitrary depth, and depending on the mapping of these iterations onto the thread space, the memory access matrix will cover both the inter- and intra-thread memory access patterns.

$$\vec{s} = \mathbf{M}\vec{i} + \vec{o}, \tag{5.3}$$

$$\begin{bmatrix}0&0\\0&0\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}0&0\\1&0\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}0&1\\0&0\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}$$
(01) (02) (03) (04)

$$\begin{bmatrix}1&0\\0&0\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}0&0\\1&1\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}0&1\\0&1\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}$$
(05) (06) (07) (08)

$$\begin{bmatrix}1&0\\1&0\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}0&1\\1&0\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}1&1\\0&0\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}1&1\\1&0\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}$$
(09) (10) (11) (12)

$$\begin{bmatrix}1&1\\0&1\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}1&0\\1&1\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}0&1\\1&1\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}\ \begin{bmatrix}1&1\\1&1\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix}$$
(13) (14) (15) (16)

Figure 5.2: eMAP cases (numbered 01 to 16).

$$\vec{s} = \overrightarrow{eMAP} + \overrightarrow{iMAP} = \mathbf{M}\overrightarrow{tid} + \overrightarrow{iMAP}, \tag{5.4}$$

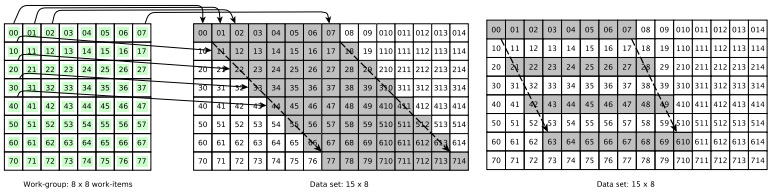
We have adapted this notation to express our specific range of MAPs - see Equation 5.4. In this new notation, we have clearly separated the inter-thread ($e\vec{M}AP$) and intra-thread ($i\vec{M}AP$) components. Intuitively, *eMAP* generates a base access index for each thread, while *iMAP* provides an offset which represents the distance from the base address. We focus on 2D thread organization: **M** becomes a 2×2 mapping matrix of the threads ($\vec{tid}$) to the data. The $i\vec{M}AP$ component is a vector representation of the intra-thread access pattern. We further rewrite Equation 5.4 to Equation 5.5, and we use this form to exhaustively generate our benchmarks.

$$\vec{s} = \begin{bmatrix}M_{00}&M_{01}\\M_{10}&M_{11}\end{bmatrix}\begin{bmatrix}t_y\\t_x\end{bmatrix} + \begin{bmatrix}iMAP_0\\iMAP_1\end{bmatrix} \tag{5.5}$$

5.3.2. eMAP

When $M_{00}, M_{01}, M_{10}, M_{11} \in \{0,1\}$, we generate 16 cases of eMAP (shown in Figure 5.2). As we have mentioned, eMAP encodes the base index of the memory references for each thread. For example, Figure 5.3 shows the base index of eMAP-14 for each thread. We assume a 8×8 workgroup, and a dataset of (at least) 15×8; for simplicity, in this example, we consider $i\vec{M}AP = \begin{bmatrix}0\\0\end{bmatrix}$. In this case, consecutive work-items in the x dimension will access contiguous data elements in the horizontal direction; consecutive work-items in the y dimension will access the elements on the diagonal line. Thus, the base index of each thread is located within the shaded area (Figure 5.3a).

When $M_{00}, M_{01}, M_{10}, M_{11} \notin \{0,1\}$, the eMAPs become more complex. When $M_{00} = 2$, we see (Figure 5.3b) 'gaps' between rows due to the larger stride, compared with eMAP-14. We can imagine that any non-unit stride will introduce such 'gaps'. For now, our work only considers [0,1] cases (Figure 5.2). We believe the extension to larger strides



(a) (b)

Figure 5.3: Base index example: (a) eMAP-14: the shaded elements are the ones accessed by the whole 8×8 workgroup; the arrows indicate (some of) the one-to-one relations between threads and data items; (b) the base index in the data structure when $M_{00} = 2$ (only show the first four rows).

will not bring changes to our methodology. However, it will lead to cases very rarely seen in real-life applications and a large increase in the experimentation time.

5.3.3. iMAP

iMAP captures the memory access patterns of a single thread, i.e., the way one thread accesses data elements. We have identified five typical iMAPs from real-life applications [44]) - namely, Single, Row, Column, Block, and Neighbor - and briefly describe them (Figure 5.4):

- **Single (1)**: each thread accesses one data element indexed by its base index.
- **Row (2)**: each thread references a row of data elements within the row indexed by its base.
- **Column (3)**: each thread accesses a column of data elements within the column indexed by its base index.
- **Block (4)**: each thread accesses a block of data elements within the block centered at the base index and sized $((2R_x + 1) \times (2R_y + 1))$.
- **Neighbor (5)**: each thread accesses the data elements lying at the base index and its four (or more) neighbors.

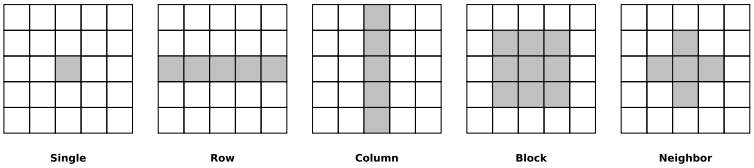


Figure 5.4: iMAPs.

The iMAP representations are listed as follows, where W and H represent the width and height of the input matrix, respectively; (R_x, R_y) is the radius of a block.

$$Single\colon iMAP = \{\begin{bmatrix}0\\0\end{bmatrix}\}$$

$$Row\colon iMAP = \{\begin{bmatrix}0\\i\end{bmatrix} \mid 0 \le i < W, i \in N\}$$

$$Column\colon iMAP = \{\begin{bmatrix}j\\0\end{bmatrix} \mid 0 \le j < H, j \in N\}$$

$$Block\colon iMAP = \{\begin{bmatrix}j\\i\end{bmatrix} \mid -R_x \le i \le R_x, i \in N; -R_y \le j \le R_y, j \in N\}$$

$$Neighbor\colon iMAP = \{\begin{bmatrix}-1\\0\end{bmatrix}, \begin{bmatrix}0\\-1\end{bmatrix}, \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}\}$$

5.3.4. MAP = EMAP+IMAP

Once eMAP and iMAP are specified, we get 80 (16 × 5) memory access patterns (MAPs), and hence need to generate and evaluate 80 microbenchmarks. The name of each MAP is a concatenation of the iMAP and eMAP numbers. For example, MAP-407 is a combination of iMAP-4 (Block) and eMAP-07. In the remainder of this chapter, we also group MAPs by their iMAP name, having Single MAPs (the MAPs that have the "Single" iMAP), and similarly Row MAPs, Column MAPs, Block MAPs and Neighbor MAPs.

When analyzing our 80 MAPs, we find that some combinations of eMAP and iMAP are either underspecified (resulting in non-interesting cases) or overspecified (resulting in contradictory definitions). Take for example MAP-101, in which each thread should access one element (according to the iMAP), but due to the eMAP (01), all threads end up accessing the same element (i.e., the (0,0) element from the dataset), an uninteresting case - i.e., an *underspecified MAP*. MAP-206 is also *underspecified*: all threads end up accessing the same row (i.e., row 0 from the dataset). On the other hand, MAP-216 is an *overspecified MAP*, as the eMAP and iMAP specify contradictory rules for accessing the same elements.

We generalize the classes of compatible eMAPs for each iMAP as follows:

- **Single: M** should have at least one '1' per row.
- **Row: M** should have no '1' on the bottom row, and at least one '1' on the top row.
- **Column: M** should have no '1' on the top row, and at least one '1' on the bottom row.
- **Block:** similar to (1).
- **Neighbor:** similar to (1).

After removing the under/overspecified MAPs, only 33 MAPs remain valid, and are listed in Table 5.3. Note that we do not take the random memory access into account since we assume that local memory is mainly suitable for applications with specific memory access patterns.

We note that this approach is, so far, application-agnostic. In other words, we attempt to generate all possible MAPs for our representation and evaluate their local memory impacts. Thus, our database is generic and fully reusable by any application.

Table 5.3: The memory access patterns ('–' represents an impossible MAP - either under or overspecified).

	Single (1)	Row (2)	Column (3)	Block (4)	Neighbor (5)
01	–	–	–	–	–
02	–	–	302	–	–
03	–	–	303	–	–
04	–	204	–	–	–
05	–	205	–	–	–
06	–	–	306	–	–
07	107	–	–	407	507
08	108	–	–	408	508
09	109	–	–	409	509
10	110	–	–	410	510
11	–	211	–	–	–
12	112	–	–	412	512
13	113	–	–	413	513
14	114	–	–	414	514
15	115	–	–	415	515
16	116	–	–	416	516

5.4. DESIGN SPACE EXPLORATION AND CODE GENERATION

5.4.1. EXPLORING DESIGN SPACE

When generating benchmarks (for a MAP) with local memory, we need to consider the issues of *local space allocation*, *local data staging*, and *local memory access*.

LOCAL SPACE ALLOCATION

Regarding the size of local space, we propose two approaches: the *min-approach* and the *max-approach* [40]. The min-approach allocates a right-sized space of local memory to hold the necessary data elements with none or very few wasted cells, while the max-approach allocates a large enough space according to the shape of a work-group. We demonstrate how these two approaches work for MAP-407 in Figure 5.5, where $R_x = R_y = 1$ and thus each thread needs a 3×3 data block. Using the min-approach consumes less local memory (Figure 5.5b), and may enable more work-groups to be active. Nevertheless, when using the min-approach, programmers need to perform work-item binding and data element shuffling according to specific memory access patterns. By comparison, the max-approach is easier for implementation (e.g., from a script). In this work, we implement both the max-approach and the min-approach, and we compare their performance in Section 5.5.3. Because we know the size of local space in advance, we use the static allocation approach (i.e., allocating local memory in the kernel).

When using the max-approach, we calculate the size of local space as follows. Each MAP has two parts– eMAP and iMAP, and the size of local space `Range` is determined by these two factors. The eMAP part specifies the `Base` (the area outlined by the dashed-line rectangle shown in Figure 5.5c) and the iMAP part specifies the `Border`. Suppose the `Base` is of size $w \times h$, and w, h is calculated as follows ($WG_x \times WG_y$ represents work-group size):

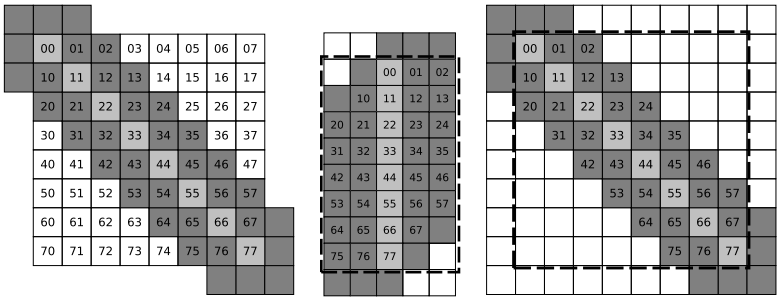


Figure 5.5: Two approaches to hold data elements using local memory for MAP-407: (a) data elements in global memory space for a work-group of 8 × 8 (only the shaded cells need to be transferred into local memory), (b) data elements in local memory space using the min-approach (occupying 50 local memory cells), (c) data elements in local memory space using the max-approach (occupying 100 local memory cells).

$$w = \begin{cases} WG_x & (M_{00} \oplus M_{01} = 1) \\ 2 \times WG_x & (M_{00} \wedge M_{01} = 1) \end{cases}$$

$$h = \begin{cases} WG_y & (M_{10} \oplus M_{11} = 1) \\ 2 \times WG_y & (M_{10} \wedge M_{11} = 1) \end{cases}$$

We can then calculate the `Range` covered by a work-group as $(w+2\times R_x)\times(h+2\times R_y)$. Furthermore, the use of the min-approach depends on the MAPs and thus it is a MAP-dependent optimization.

LOCAL DATA STAGING

After allocating the required local memory, we need to stage data in the local space specified by the `Range`. Note that this process is independent of how data is used (to be mentioned in Section 5.4.1), which provides a large degree of freedom to optimize the data staging process. In [44], we have proposed the `FCTH` (i.e., loading the base data first, and then the border data) and `TBT` (i.e., reading data in a tile-by-tile fashion) to stage the local data and shown that `FCTH` gives us a better performance. Hence, in this work, we use the `FCTH` approach.

LOCAL DATA ACCESS

Compared with accessing the data in global space, the key issue of accessing local data is the `index space conversion`. Specifically, we need to use the `local` thread index instead of `global` thread index while keeping the logic of using global memory. In addition, to ensure that the work-items within a work-group efficiently reference the data elements in local space, we need to avoid bank conflicts, i.e., to force the access requirements from multiple work-items of a work-group fall into different banks. By using data padding, we remove bank-conflicts from the generated microbenchmarks.

5.4.2. CODE GENERATOR

Our code generator consists of two templates: host code and kernel code. The engine of the host code creates a driver that allocates/deallocates global space, initializes the data space, transfers data between the host and the device and launches kernels. It also has a module of time keeping and results validation. With regard to the kernel code, each microbenchmark of using local memory includes three major steps: statically allocate local memory space, load data elements into local memory, and use them. We found that different iMAPs differ in their code generators. Thus, we have developed a different code generating engine based on the iMAPs. Figure 5.6 shows the kernel template for the Block iMAP.

Taking MAP-407 (Figure 5.5) for example, we show the three steps in detail.

Step I: Allocating local memory (@lmAlc)

We use the approaches mentioned in Section 5.4.1 to calculate the local size and allocate the local space. For MAP-407 (shown in Figure 5.5), the min-approach and max-approach need different amounts of local space. The local space size is calculated as $(WG+2\times R)\times(4\times R+1)$ for the min-approach and $(WG+2\times R)^2$ for the max-approach, where WG is the work-group size ($WG_x = WG_y = WG$), R is the radius of the block ($R_x = R_y = R$). In Figure 5.5, $WG = 8$ and $R = 1$. Thus, the min-approach needs 50 cells, while the max-approach needs 100 cells. We have implemented the *max-approach* in the code generator and we use the *min-approach* as a post-optimization step.

Step II: Loading data into local memory (@lmLoad)

When moving data elements from global memory to local memory, multiple passes

```
1  __kernel void CG(@type in[], @type out[], \
2          int cdim, int rdim){
3    int tgx = get_global_id(0);
4    int tgy = get_global_id(1);
5    int tlx = get_local_id(0);
6    int tly = get_local_id(1);
7    int wgx = get_group_id(0);
8    int wgy = get_group_id(1);
9    @bxy  // Get the base index of a group
10   @lmAlc  // Step 1: allocte local memory space
11   @varDec
12   // Step 2: load data from GM to LM
13   @lmLoad
14   barrier();
15   // Step 3: use the data elements in LM
16   @lmUse
17   barrier();
18   // output
19   out[tgy*cdim+tgx] = @lmOut;
20   return ;
21 }
```

Figure 5.6: A code template to generate kernels in OpenCL (*@type* represents the used data type, *@bxy* represents the base data index for each group which is MAP-dependent, *@lmAlc* is the name-holder for local space allocation, *@varDec* declares temporal variables, *@lmLoad* is the name-holder of loading data into local memory and *@lmUse* is that of how to use it, and *@lmOut* is the final returned results).

are needed with `FCTH` [44]. When using the max-approach, we bind one thread to one data element in the central shaded area (outlined by dashed square in Figure 5.5c). Thereafter, we load the border data into the local space. Nevertheless, loading data with the min-approach is more complicated. Apart from thread masking, we have to deal with *data shuffling* to put the data elements in the right places and thus it is MAP-dependent.

Step III: Accessing local memory (@lmUse)

Using data elements in local memory is straightforward. The key is to find the correspondence between the global data index and its local data index. For MAP-407, each thread needs a block of data elements around its thread index (the light-shaded elements in Figure 5.5b and 5.5c). Besides, we need to *re-shuffle* the access index when using the min-approach.

By using our code generator, we obtain 66 microbenchmarks (33 using local memory and 33 using global memory only) [4]. Our experience shows that the technical difficulty of designing the code generator is dealing with the aforementioned three steps. Given that the use of the min-approach and bank-conflict removal varies from MAP to MAP, we implement them as manual transformations applied after the automatic code generation.

5.5. PERFORMANCE DATABASE

5.5.1. PERFORMANCE METRIC

We use memory bandwidth as our performance metric. Suppose we have $W \times H$ threads, and each needs N data elements of `type` data type (N is determined by iMAP). We run each kernel and measure the kernel execution time T. Then we calculate the bandwidth as $W \times H \times N \times size(type)/T$. We measure the memory bandwidth for cases without (b) and with local memory (B), use b as the reference, and calculate the memory bandwidth ratio ($mbr = B/b$). If $mbr > 1$, using local memory is beneficial in terms of memory bandwidth; otherwise, using local memory leads to a performance loss.

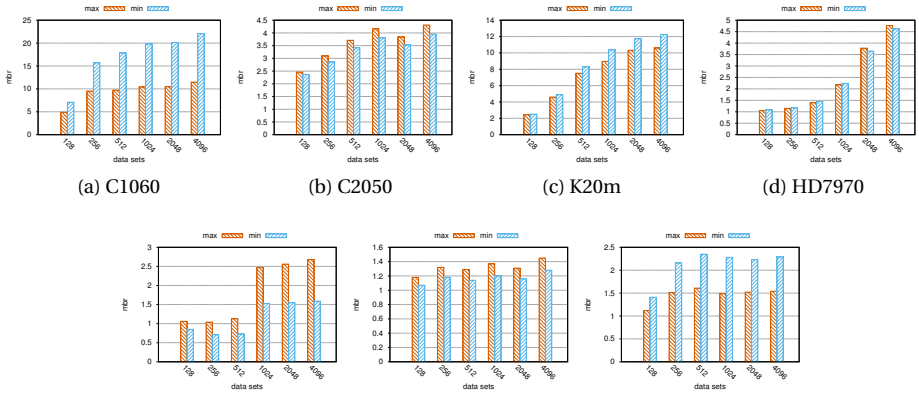
5.5.2. EXPERIMENTAL SETUP

We have run and compared the benchmarks on seven platforms, whose configurations are shown in Table 7.2. When measuring memory bandwidth, we used six data sets ($W \times H$ in Section 5.3.3): 128×128, 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096. For the Block MAPs, we set the radius to be 3 ($R_x = R_y = 3$) and we expect memory bandwidth to increase with a larger radius. For each measurement, we run 21 iterations (the first iteration as a warm-up run). To avoid data reuse between iterations and cache interference, we flush caches between iterations. Furthermore, we believe that the choice of work-group sizes has an impact on the memory bandwidth. In this work, we set it to be 16×16.

[4] The code generator is available: https://github.com/haibo031031/aristotle

Table 5.4: Details of the used platforms.

	Platform I	Platform II	Platform III	platform IV
Host	Intel Core i7 920	Intel Xeon E5620	Intel Xeon E5620	Intel Xeon E5620
Host OS	UBUNTU v11.10	CentOS v6.2	CentOS v6.2	CentOS v6.2
Device	NVIDIA Tesla C1060	NVIDIA Tesla C2050	NVIDIA Tesla K20m	AMD HD7970
GCC	v4.6.1	v4.4.6	v4.4.6	v4.4.6
OpenCL	CUDA v5.5	CUDA v5.5	CUDA v5.5	AMD APP v2.8
	Platform V	**Platform VI**	**Platform VII**	
Host	Intel Xeon E5-2620	Intel Xeon E5-2620	Intel Xeon X5650	
Host OS	CentOS v6.2	CentOS v6.2	CentOS v6.2	
Device	Intel Xeon Phi 5110P	Intel Xeon E5-2620	Intel Xeon X5650	
GCC	v4.4.6	v4.4.6	v4.4.6	
OpenCL	Intel OCL SDK v3.0	Intel OCL SDK v3.0	Intel OCL SDK v3.0	



(a) C1060 (b) C2050 (c) K20m (d) HD7970

(e) Phi-5110P (f) E5-2620 (g) X5650

Figure 5.7: Performance comparison of the max/min approaches.

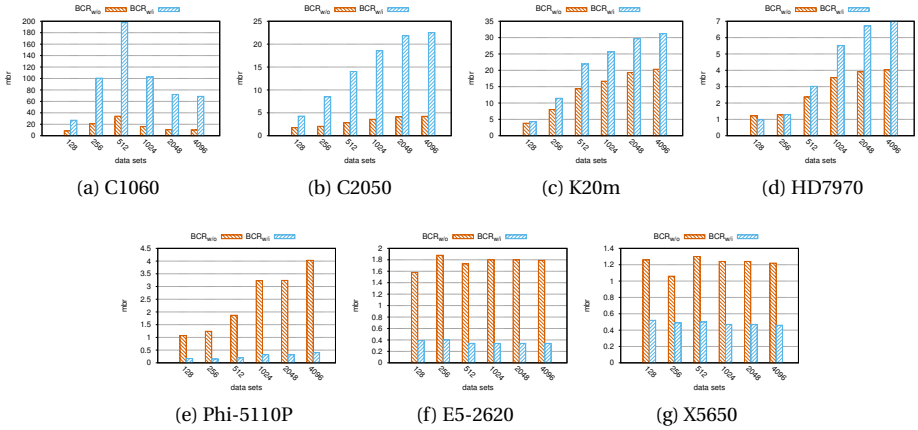
5.5.3. PERFORMANCE OPTIMIZATION CONSIDERATIONS

THE MAX-/MIN- APPROACHES

For MAP-407, we compare the *mbr* of the max and min approaches on the seven platforms in Figure 5.7. We see that the min-approach is not always performing better than the max-approach. In fact, the min-approach can achieve much better performance on C1060 and X5650, and it performs slightly better on K20m (up to 15%). On C2050, HD7970, and E5-2620, the performance of the min-approach is slightly worse. We also note that the bandwidth suffers around a 40% loss with the min-approach on Xeon Phi. Thus, the overall performance can be significantly influenced by the way of using local memory. When predicting performance, we need to take the design choice into account.

REMOVING BANK-CONFLICTS

We manually use the *padding* approach to remove bank-conflicts. Taking MAP-204 as an example, we show the performance impacts of removing bank-conflicts in Figure 5.8. We see that removing bank-conflicts on the GPUs (i.e., C1060, C2050, K20m, HD7970)



(a) C1060 (b) C2050 (c) K20m (d) HD7970

(e) Phi-5110P (f) E5-2620 (g) X5650

Figure 5.8: Performance comparison before and after removing bank-conflicts.

significantly increases the memory bandwidth (up to 7×), while the 'optimization' leads to a performance decrease on the cache-only processors (i.e., Xeon Phi, E5-2620, and X5650). Thus, we conclude that this optimization is specific for processors with a SPM.

5.5.4. PERFORMANCE DATABASE

DATABASE RECORD

After running the microbenchmarks, we obtain a performance database indexed by three items (platform, map, dataset) shown in Figure 5.9. Once the index is specified, a query in the database will return a database *record*. Each record consists of the memory bandwidth without local memory (b), the memory bandwidth with local memory (B), and their ratio (mbr).

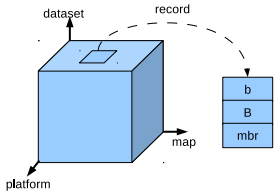


Figure 5.9: The database dimensions and its record.

OBSERVATIONS

We run each experiment 21 times, and calculate the average value and the standard derivation value. For demonstration simplicity, we show the b and B (the average and the standard derivation number) of the performance database (available online[5]): the hori-

[5] PDB: https://github.com/haibo031031/aristotle/tree/master/pdb.

zontal axis represents the six data sets and the vertical axis represents bandwidth. Overall, we found that the performance benefits of using local memory are heavily dependent on the size of the data sets. In most cases, the bandwidth increases over datasets. Only in a few cases (e.g., the Column MAPs on E5-2620 and X5650), the bandwidth without local memory decreases over datasets. Besides, we make the following observations for each platform:

C1060 The GPU differs from other processors in that it has a scratch-pad memory, but no caches. We see that using local memory on C1060 can achieve a memory bandwidth increase for most memory access patterns (29 out of 33 MAPs). When looking into the microbenchmarks, we found that there are two factors leading to the bandwidth increase - data reuse and changes in global memory access orders. As we have mentioned in Section 5.1, data reuse is a common indicator of using local memory. Taking the Single MAPs for example (see Table 5.3), we can reuse data for MAPs-(107, 116). Using local memory can also change the memory access order and reduce the number of memory transactions (thus increase the off-chip memory bandwidth). For the Single MAPs, memory access orders of MAPs-(107, 110, 112, 113, 115) are changed when using local memory. For MAP-108, performance is lost ($mbr < 1$) because no data reuse or changes in memory access order appear. The performance loss results from the overhead of using local memory. Although data reuse exists in MAP-109, the data request from the off-chip memory can be serviced in a broadcast manner, and thus using local memory brings no bandwidth improvement. Furthermore, all the Row MAPs, Column MAPs, and Block MAPs can benefit from data reuse, and some of them can even achieve a bandwidth increase due to the common effort of both data reuse and memory access changes.

C2050 and K20m They have not only scratch-pad memories but also caches. Similar to C1060, using local memory on C2050 and K20m is highly beneficial in most cases. On C2050 and K20m, the bandwidths follow the same trends with that on C1060, but the changes are less significant: the older cache-less C1060 benefits much more from local memory than the newer C2050 and K20m. This happens because the caches will alter the performance benefits of the explicit usage of local memory. In some cases, hardware caches are able to make use of the inherent data locality in the MAP without using scratch-pad memory (see MAPs-116, 508, and 514). In other cases, with more complicated locality patterns, explicit usage of local memory remains beneficial on C2050 and K20m (see MAPs-204, 303, 410, etc).

HD7970 The processor also has both scratch-pad memories and caches. For most MAPs, the performance benefits are less significant and the bandwidth varies a lot over the data sets, which significantly differs from that on NVIDIA GPUs. We believe this is because HD7970 has a different cache architecture and implementation compared to C2050 and K20m.

Phi-5110P, E5-2620 and X5650 These processors only have caches on-chip, and implement OpenCL local memory on global memory, in an emulation mode. Thus, using local memory is equivalent to using the cached off-chip global memory and introduces

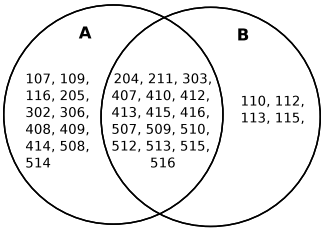


Figure 5.10: Performance factors and MAPs distribution.

extra overheads (compared with using global memory directly), which might slow down the execution. However, we get better performance for some MAPs (e.g., Column MAPs) by using local memory (see Table 7.4). This is due to better caching when using a smaller memory space. We also note that the bandwidth varies more significantly between runs on E5-2620 and X5650 than on Phi-5110P and GPUs. Another interesting observation is that using local memory preserves bandwidth on MAPs like MAP-302 while the bandwidth drops over datasets on E5-2620 and X5650. We conclude that data reuse is a must to obtain a bandwidth increase by using local memory on cache-only processors.

5

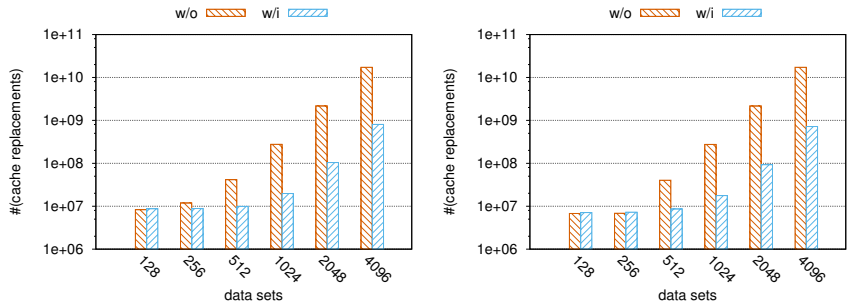
PERFORMANCE FACTORS ANALYSIS

As we have shown, two factors contribute to the memory bandwidth improvement: data reuse (Factor A) and access order changes (Factor B). We analyze the MAPs and identify the factors for each MAP in Figure 5.10. We see that 11 MAPs present the potentials of reusing data, while 4 MAPs can benefit from the changes in memory access orders due to the usage of local memory. Typically, data access orders can be changed when loading data from global space to local space. Besides, there are 16 out of the 33 MAPs that can use both of them. Data reuse can be a benefit source for both caches and scratch-pad memories, whereas access order changes does not necessarily lead to a bandwidth increase.

ARCHITECTURE-DEPENDENT ANALYSIS

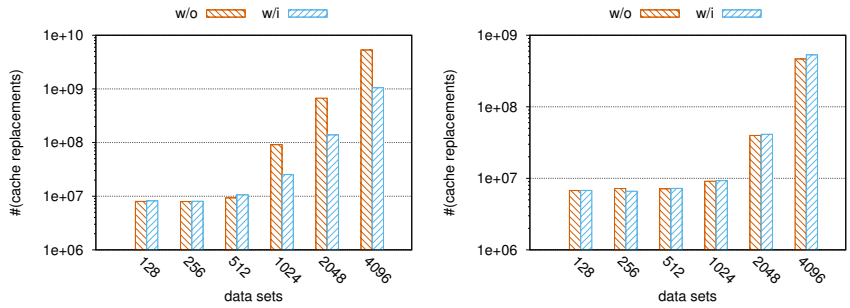
We roughly divide the selected processors into three groups: the SPM-only processors, the SPM-Cache processors, and the Cache-only processors. The SPM-only processors (e.g., C1060) have a scratch-pad memory, but have no on-chip caches. Using local memory on such processors can benefit from either data reuse (i.e., less off-chip data movements) or higher effective off-chip bandwidth (shown in Section 5.1.2). For C2050, K20m, and HD7970, they have both scratch-pad memories and caches. Using local memory can give a higher bandwidth (than without it) when the MAPs are cache-unfriendly. Otherwise, adopting local memory leads to a performance decease due to the overheads.

The cache-only processors (Phi-5110P, E5-2620 and X5650) do not have a on-chip scratch-pad memory and local memory is allocated on the global space. However, using local memory can change data layouts (i.e., access orders) and thus can be seen as a 'software' optimization technique. By using local memory, the data elements are first loaded into the local space and then accessed within the local space. In this way, we may avoid 'unnecessary' cache-line replacements and have a better utilization of caches.



(a) L1 cache replacement. (b) L2 cache replacement.

Figure 5.11: The number of cache replacements for MAP-302.



(a) L1 cache replacement. (b) L2 cache replacement.

Figure 5.12: The number of cache replacements for MAP-204.

We measure the number of cache-line (L1 and L2) replacements for MAP-302 and MAP-204 on E5-2620 (Figure 5.11 and Figure 5.12). For MAP-302, we see that the number of cache replacements is larger without using local memory for both L1 cache and L2 cache. Thus, using local memory on MAP-302 is beneficial on E5-2650 especially for large datasets. However, for MAP-204, cache-lines are replaced less frequently when using local memory on the L1 cache while it occurs more often on the L2 cache. We see that using local memory gives a smaller bandwidth for this MAP.

PERFORMANCE GAIN/LOSS DISTRIBUTION

We define that using local memory has a *similar* performance to that using global memory when $|1.0-\delta| \leq mbr \leq |1.0+\delta|$. Therefore, using local memory has a *gain* performance when $mbr > |1.0+\delta|$, and using local memory has a *loss* performance when $mbr < |1.0-\delta|$. We show the overall performance gain/loss distribution in Table 7.4, where $\delta = 0.05$ (5%). We note that, in most cases on NVIDIA GPUs, using local memory gives us a bandwidth increase. Specifically, the number is around 90% on C1060, and 80% on C2050 and K20m. Thus, using caches partially 'cancels' the benefits of using local memory. On AMD HD7970, over half of the MAPs have a similar performance

Table 5.5: Performance gain/loss distribution ($\delta = 0.05$).

		C1060	C2050	K20m	HD7970	Phi-5110P	E5-2620	X5650
128	Gain	27(81%)	21(63%)	20(60%)	6(18%)	7(21%)	12(36%)	9(27%)
	Loss	1(3%)	4(12%)	0(%)	5(15%)	16(48%)	17(51%)	17(51%)
	Similar	5(15%)	8(24%)	13(39%)	22(66%)	10(30%)	4(12%)	7(21%)
256	Gain	29(87%)	25(75%)	26(78%)	13(39%)	7(21%)	9(27%)	9(27%)
	Loss	2(6%)	5(15%)	4(12%)	3(9%)	20(60%)	22(66%)	20(60%)
	Similar	2(6%)	3(9%)	3(9%)	17(51%)	6(18%)	2(6%)	4(12%)
512	Gain	29(87%)	26(78%)	26(78%)	13(39%)	8(24%)	7(21%)	11(33%)
	Loss	3(9%)	6(18%)	6(18%)	3(9%)	18(54%)	20(60%)	21(63%)
	Similar	1(3%)	1(3%)	1(3%)	17(51%)	7(21%)	6(18%)	1(3%)
1024	Gain	29(87%)	26(78%)	26(78%)	14(42%)	16(48%)	8(24%)	11(33%)
	Loss	4(12%)	6(18%)	6(18%)	1(3%)	14(42%)	24(72%)	19(57%)
	Similar	0(%)	1(3%)	1(3%)	18(54%)	3(9%)	1(3%)	3(9%)
2048	Gain	29(87%)	26(78%)	26(78%)	15(45%)	17(51%)	8(24%)	8(24%)
	Loss	4(12%)	6(18%)	6(18%)	5(15%)	15(45%)	24(72%)	21(63%)
	Similar	0(%)	1(3%)	1(3%)	13(39%)	1(3%)	1(3%)	4(12%)
4096	Gain	29(87%)	26(78%)	26(78%)	16(48%)	16(48%)	9(27%)	12(36%)
	Loss	4(12%)	6(18%)	6(18%)	5(15%)	15(45%)	22(66%)	20(60%)
	Similar	0(%)	1(3%)	1(3%)	12(36%)	2(6%)	2(6%)	1(3%)

between with and without adopting local memory for the small datasets, while the number becomes smaller for the larger datasets and more MAPs can benefit from using local memory. On the cache-only processors (Phi-5110P, E5-2620, X5650), using local memory leads to a performance decrease for around half of the MAPs for small datasets. For large datasets, we note less MAPs on Phi-5110P but more MAPs on E5-2620 and X5650 that has a performance decrease by using local memory. This is mainly due to the fact that they have a difference cache architectures (i.e., Xeon Phi has a distributed last-level cache while the other processors have a unified one). We recommend using local memory for the *better* MAPs based on the guidelines mentioned in Section 5.4; for the cases with little or no bandwidth increase (or even bandwidth decrease), using local memory is not recommended due to the low ratio between performance gain and programming effort.

5.6. COMPOSING MAP IMPACTS

We have quantified the performance impacts of using local memory for isolated MAPs and a simple query in the database can tell us the performance benefits. However, a real-world kernel often has multiple data structures (and memory access patterns). In this section, we propose composition rules in the presence of multiple MAPs to give the performing order of using local memory.

For a given *MAP*, let $\oplus$ represent that using local memory brings a 'positive' performance impact (positive MAP) and let $\ominus$ represent that using local memory gives a 'negative' performance impact (negative MAP). Assume we have two MAPs (two data structures in a kernel): *MAP1* and *MAP2*, which can be the same pattern or two different pat-

terns. For each MAP, we have two choices: *l*– choose to use local memory, and *g*– choose not to use local memory (thus using global memory). Then we can obtain four versions of code for this kernel: (*g*,*g*), (*g*,*l*), (*l*,*g*), and (*l*,*l*). We need to pick the most efficient choice among the four.

We use two metrics to evaluate the efficiency: *performance* and *programming efforts*, of which *performance* is taken as our first priority. In this work, we consider an effort of enabling local memory usage for a data structure as the unit of programming effort. Ideally, we prefer an efficient solution with less programming effort. To compose MAP impacts, we propose and analyze the following rules.

Rule 5.1: $\ominus + \ominus \rightarrow (g, g)$.

Analysis. For either *MAP1* or *MAP2*, using local memory leads to a performance loss. When composing them, we cannot find any sources of a performance gain. Thus, we choose not to use local memory for both of them.

Rule 5.2: $\oplus + \ominus \rightarrow (l, g)$, $\ominus + \oplus \rightarrow (g, l)$.

Analysis. Suppose *MAP1* can benefit from using local memory ($\oplus$), while *MAP2* suffers a performance loss ($\ominus$). Since using local memory on *MAP2* brings us no performance gain, we choose not to use local memory on it. Next, let us consider *MAP1*. Further suppose that we need D_1 data elements for *MAP1* and D_2 data elements for *MAP2*, and their bandwidths are W_1 and W_2, respectively. Let W be the overall bandwidth and T represent the data transfer time. Thus, we can obtain

$$T = \frac{D_1 + D_2}{W(W_1, W_2)}$$

As we have analyzed, the performance gain of using local memory comes from two factors: either the decrease of data amount (D), and/or the increase of global memory bandwidth (W). Thus, we need to consider two cases:

Case 1: Using local memory on *MAP1* decreases D_1. In such a case, we need to move less data from global memory (D), thus posing less contention for the shared resources (e.g., channels and ports) and leaving more chances for *MAP2* to transfer data. Therefore, using local memory on *MAP1* will improve the overall performance, i.e., $\phi(l, g) > \phi(g, g)$[6].

Case 2: Using local memory on *MAP1* increases W_1. In case of a under-utilized bandwidth, we can better utilize shared resources (ports and channels) and thus have a better performance. Once we reach the maximum achievable bandwidth, using local memory further brings us no performance gain. In other words, using local memory on *MAP1* can guarantee that $\phi(l, g) \geqslant \phi(g, g)$.

Therefore, we choose (l, g) in this case. Likewise, we choose (g, l) for the $\ominus + \oplus$ combination.

Rule 5.3: $\oplus + \oplus \rightarrow (?, l)$.

Analysis. As we infer from Rule 5.2, we can guarantee that $\phi(l, g) \geqslant \phi(g, g)$. Thereafter, we need to take a decision of enabling local memory on *MAP2*. We can see similar performance benefits as shown in Rule 5.2. However, using local memory on *MAP2* also increases the amount of used local memory and thus may reach the maximum limit on

[6] $\phi(l, g)$ denotes the performance when using local memory on one MAP, while $\phi(g, g)$ denotes the performance when we do not use local memory.

the device. Therefore, we need to check whether there remains enough local space beforehand. If there is enough local space for MAP2, we will choose to perform allocation for it.

Rule 5.4: $\underbrace{\oplus+\oplus+...+\oplus}_{m}+\underbrace{\ominus+\ominus+...+\ominus}_{n}\rightarrow(\underbrace{?,?,...,l}_{m},\underbrace{g,g,...,g}_{n})$.

Analysis. Assume that, when using local memory, we divide the *MAPs* into two groups based on the performance benefits: *m MAPs* can benefit from using local memory while *n MAPs* suffers in performance. By iteratively using Rule 5.1, we choose not to use local memory on these *n* negative *MAPs*. According to Rule 5.2, we use local memory on the right-most *MAP*. Thereafter, it is unclear whether to use local memory or not based on Rule 5.3. Thus, this rule is a derivation of Rule 5.1, 5.2, and 5.3.

Up to now, there remains one question: *which positive MAP do we select first?* Different MAPs may differ in performance benefits due to the MAP feature and its run-time dataset. Suppose when using local memory on MAP1, the performance benefit is mbr_1 and the dataset size is D_1; when using local memory on MAP2, the performance benefits is mbr_2 and the dataset size is D_2. Then we can calculate the *performing order weight* ω. When using local memory, we will select the MAP with the largest ω until we do not have enough local space.

$$\omega_1 = \frac{D_1}{D_1+D_2}\times mbr1$$

$$\omega_2 = \frac{D_2}{D_1+D_2}\times mbr2$$

5.7. COMPOSING RULES VALIDATION

5.7.1. A MAP COMPOSER

To validate our composition rules, we compose MAPs based on the code generator mentioned in Section 5.4. When multiple MAPs are used in a kernel, we consider the use of local memory in an incremental manner. In other words, the kernel template takes multiple MAPs as input and we use local memory on them one by one. For 2 MAPs, we have built a composer and show its structure in Figure 5.13 [7]. The composer generates three code versions: (v0) without using local memory, (v1) a code version of using local memory on MAP1, and (v2) a code version of using local memory on both MAP1 and MAP2. Thus, when we have *N* MAPs (they can be the same or different), the composer will generate $N+1$ versions of code.

We validate our proposed rules for each use of local memory. We take the test case as a successful prediction when the results meet with those from the composing rules. We calculate the prediction accuracy as the number of successful tests divided by the total number of tests. We assume that a test fails when we observe one of the follows:

1. A positive MAP gives a performance degradation from using local memory.

2. A negative MAP gives a performance increase from using local memory.

[7] The composer is available: https://github.com/haibo031031/aristotle

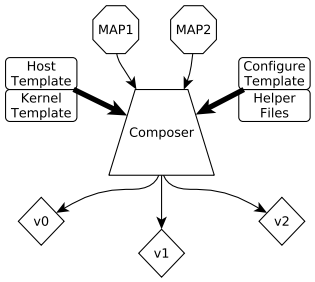


Figure 5.13: The Composer architecture.

5.7.2. RULE VALIDATION

We use 7 platforms and 6 datasets (see Section 5.5.2), and use 2 data structures which are of the same MAP or 2 different MAPs [8]. Note that we can guarantee there is sufficient local space when considering 2 data structures. With 33 MAPs, we need to evaluate 2178 ($33 \times 33 \times 2$) test cases for each platform and dataset. Thus, we can evaluate the performing order ω in an exhaustive manner.

Our validation results are shown in Table 5.6. We see that the rules holds with an accuracy of around 90% on NVIDIA GPUs, while the number ranges from 55% to 75% on the AMD GPU. On the cache-only processors (Xeon Phi, E5-2620), the prediction accuracy is up to 80%, while we note that the rules see a relatively low accuracy on X5650. We believe this is because of the cache interferences between the data structures.

Table 5.6: Rule validation results.

	128	**256**	**512**	**1024**	**2048**	**4096**
C1060	90%	95%	94%	93%	92%	93%
C2050	93%	94%	94%	93%	92%	91%
K20m	90%	91%	91%	90%	89%	88%
HD7970	55%	63%	69%	69%	75%	66%
Phi-5110P	74%	80%	81%	77%	79%	83%
E5-2620	61%	73%	78%	80%	81%	80%
X5650	65%	68%	72%	70%	73%	65%

5.7.3. USING ARISTOTLE

Given a kernel, users first need to abstract the MAP for each data structure. Depending on the given platform and the MAP, we query the performance database. If it is beneficial, we will perform code transformation to enable local memory on the input kernel. Otherwise, we keep the original kernel code. When the given kernel has multiple MAPs,

[8] Running the validation experiments for 2 MAPs takes 2~6 days per platform and we cannot afford to validate more MAP composition.

we need to calculate the performing order weight ω based on the isolated mbr. The composing rules (in Section 5.6) provide users with a reference on how to compose multiple MAPs. For NVIDIA GPUs, the prediction accuracy is high, but it is relatively low on the cache-only processors such as X5650. We recommend the use of local memory (as a software optimization technique) for the MAPs that have a large bandwidth benefit (e.g., $mbr > 1.5$). For the MAPs that show smaller bandwidth benefits, using local memory is not recommended.

5.8. RELATED WORK

In this section, we discuss prior work on benefits prediction and code transformation of using local memory. In [58], the author presents a method of computing precisely which memory cells are reused due to temporal locality. In [71], Issenin presents an automated approach for analyzing the data reuse opportunities in a program, that allows modification of the program to use scratch-memory. The approach can reduce energy consumption and improve performance. However, they focus on enabling local memory only when data reuse is available. Our study tackles a more generic problem.

Research on code transformer of enabling local memory is yet another interesting topic. Baskaran et al. [13] develop an approach to effective automatic data management for on-chip memories, including creation of buffers in on-chip memories for holding the needed data elements, determination of array access functions, and generation of code that moves data between slow off-chip memory and fast local memories. In [172], Yang et al. propose a GPGPU compiler, which converts the un-coalesced accesses to coalesced ones, and enhances the data reuse. Our work is orthogonal to this work, as our solution is intended to estimate the performance gain of their approaches and dedicates the necessity of using the code transformers.

A more generic related topic is auto-tuning. Generally, there are two types of auto-tuning: empirical optimization [57, 91, 105, 109] and model-driven optimization [10, 64, 85, 178]. Although empirical optimization techniques give the optimal performance, it generates a large number of parameterized code variants and the time cost of searching for the best code variant makes it less attractive. In contrast, model-driven optimizations self-tune implementation-related parameters to obtain optimal performance. Using model-driven auto-tuning typically has an $O(1)$ cost, since the parameters can be derived from the analytical model. However, it may not give optimal performance, because analytical models are only simplified abstractions of architectures and/or applications. Our approach relates both: we use modelling to build our database, and use the database to potentially prune the search space of empirical auto-tuners.

5.9. SUMMARY

Architecture diversity and application implementation differences make the performance benefits of using local memory much less predictable than expected. In this work, we presented a benchmark-based approach (*Aristotle*) to tackle this issue starting with the memory access patterns (MAPs). For each such MAP, we generated benchmarks for a naive version (without local memory) and an optimized one (using local memory). We evaluated the microbenchmarks on the NVIDIA GPUs (C1060, C2050, K20m), AMD GPUs

(HD7970), Intel Xeon CPUs (E5-2620 and X5650), and Intel Xeon Phi 5110P, and obtained a performance database.

By analyzing the memory access patterns and the performance impacts of using local memory, we have found that both *data reuse* and *changes in access order* may contribute to the effective bandwidth increase. On the processors with both scratch-pad memories and caches, the performance benefits of using local memory in OpenCL kernels are less significant. Furthermore, using local memory on the cache-only processors (e.g., the traditional multicore CPUs) can be seen as a software optimization and might be efficient by better utilizing caches.

This is the first extensive, systematic study of local memory impacts based on generalized MAPs. Not only can this work provide essential information for performance prediction with database queries, but it can also give a performance indicator of local memory usage. Further, we presented four rules to generate the performing order of using local memory when we have multiple data structures. Our results validated the usefulness of our composing rules on GPU architectures in particular. Meanwhile the prediction accuracy is relatively low on the cache-only processors largely due to the cache interference between multiple data structures. We believe that this issue is impossible (or difficult at least) to fix because of the dynamic nature of cache interferences.

We note that for new emerging platforms (with OpenCL support), the database can be easily extended: one can simply use the microbenchmarking and logging tools to expand it. The performance database, together with the composing rules will give us an indicator of whether or not to use local memory. Once given this information, we will enable or disable local memory usage accordingly, which will be illustrated in Chapter 6 and Chapter 7.

6

ELMO: AN API TO ENABLE LOCAL MEMORY USAGE

In this chapter, we propose an API to enable local memory usage. The API facilitates code specialization when using local memory is profitable. We focus on the API design requirements, its front-end and back-end. We also discuss its productivity and usability.

Because local memory is situated on-chip, it is much faster than the global memory. Thus, a proper use of local memory often leads to higher memory bandwidth and thus performance improvement. Nevertheless, using local memory is an error-prone and time-consuming process. Programmers often have to manually address, in their code, challenges like (1) geometry mismatch, (2) work-items masking and binding switches, and (3) inefficient local memory organization (see Section 6.1). We argue that when solving these problems manually, programmers waste too much time on non-computational and non-functional coding details, which hinders productivity and bloats the code.

Multiple approaches have been proposed to improve productivity while achieving high performance for parallel architectures, which can be loosely classified into (i) new languages (e.g., OptiML [24]), (ii) auto-parallelizing compilers (e.g., OpenACC [120], Mint [155]), and (iii) libraries/APIs (e.g., Thrust [2]). In all these cases, programmers are isolated, in one way or another, from the difficult implementation details related to the platform architecture: they can focus on the functional parts of the application and leave these non-functional elements to be solved by run-times, compilers, or libraries.

In this chapter, taking the third approach, we focus on the design and implementation of a high-level API targeting the efficient usage of local memory on modern many-core processors. As the main difficulty of these operations is the complex and dynamic nature of the binding between the threads (work-items) and the data elements in global or local memory, we propose *ELMO*, a collection of easy-to-use APIs that (1) present a friendly front-end to make these bindings/mappings transparent to users (see Section

This chapter is based on our work published in the Proceedings of PDP 2013 [44].

6.2), and (2) provide implementations and perform several optimizations to ensure the efficiency of the local memory operations (see Section 6.3).

Summarizing, our contributions are as follows:

- We present three challenges of using local memory and thus summarize the ELMO requirements.
- We illustrate the design of three APIs, and further provide GPU-based back-end implementations for them (The source code is on-line available[1]).
- We evaluate ELMO's performance against native kernels and hand-tuned kernels.
- We discuss the programmability and usability of ELMO, and its limitations.

Our results show that with ELMO the kernels can run by up to 3.7× faster over native kernels and deliver matching performance with hand-tuned kernels on NVIDIA Quadro5000 (see Section 6.4). Using ELMO, programmers can focus solely on the functional side of the application, which improves their productivity by enabling faster and less error-prone coding (see Section 6.5). Furthermore, the back-end optimizations can be adapted to novel architectures, providing better opportunities to improve the performance portability of ELMO code.

ELMO is targeting OpenCL-compliant platforms and kernels. Due to the cross-platform capability of OpenCL, the APIs are applicable for any OpenCL-compliant devices with local memory. However, since multi-core CPUs allocate local memory on global memory space (not on-chip memory), we cannot ensure the performance benefits of using local memory on them, and thus the back-end of ELMO is, for now, targeting GPUs. Therefore, all the experiments and results included in this chapter relate to GPUs.

The remainder of this chapter is organized as follows: Section 6.1 illustrates the challenges of using local memory. Section 6.2 introduces the design of ELMO. Then the implementations of the APIs are presented in Section 6.3. We evaluate the performance of ELMO in Section 6.4, and discuss the usage of ELMO in Section 6.5. Finally, we give the related work in Section 6.6, and conclude the work in Section 6.7.

6.1. ELMO REQUIREMENTS

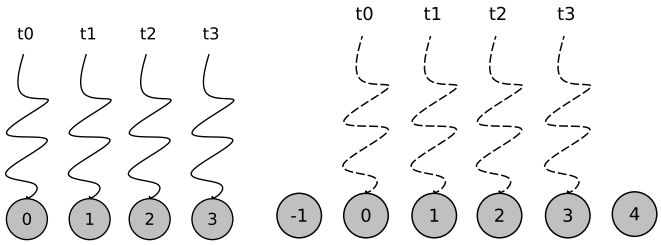
In this section, we give a brief description of the basic operations that programmers need to do when using local memory and explain the challenges behind them. Based on this, we define our API's requirements.

Local memory operations consist of two types: (1) data transfers between global memory and local memory, and (2) data transfers between registers (or private memory) and local memory. In theory, any work-item of a work-group can access any data element in local memory, and there is no data coherence guarantee. Thus, it becomes difficult to manage this process especially when the number of work-items increases from 'multi'-scale to 'many'-scale. The following are the challenging scenarios we have identified when using these two types of basic operations.

[1] https://github.com/haibo031031/elmo

6.1.1. CHALLENGE I: GEOMETRY MISMATCH

When accessing data in the local memory and/or when bringing data to local memory, the simple cases of $1:1$ or $1:n$ work-items per data elements are easily solved: each work-item will access exactly 1 or n data elements. However, many applications (e.g., image convolution) also need *halo data* - i.e., the data elements neighboring the central part (see Figure 6.1). This will often lead to a geometry mismatch between the work-items used to bring the data and the data elements themselves. Thus, binding work-items to data elements in an orderly fashion becomes difficult. And for multi-dimensional data (2D or 3D), the binding between work-items and data elements will make the situation worse.



(a) 1:1 mode (b) with halo data

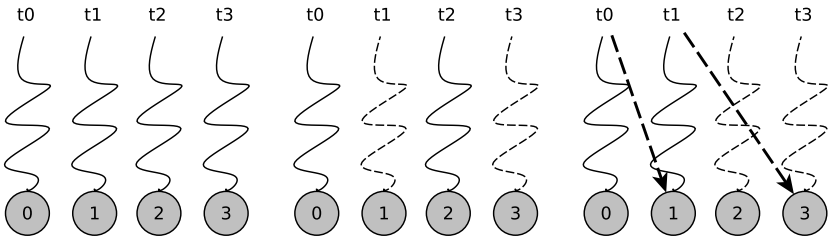
Figure 6.1: Geometry mismatch for 4 work-items and: 4 data elements in the 1:1 mode, or 6 data elements in the mismatch mode.

6.1.2. CHALLENGE II: WORK-ITEMS MASKING AND BINDING SWITCHES

For applications like reduction, multiple rounds are required to execute a single kernel, and not every work-item has to remain active in each round. Thus, programmers need to deactivate work-items that are not used in the next round (see Figure 6.2a and 6.2b) to avoid unnecessary data updates. Furthermore, the bindings between work-items and data elements change even in a single kernel. In Figure 6.2c, work-items $t0$ and $t1$ process data elements $d0$ and $d1$ in one round, but switch to update $d1$ and $d3$ in the following round, respectively. This binding switch makes code even more complex.

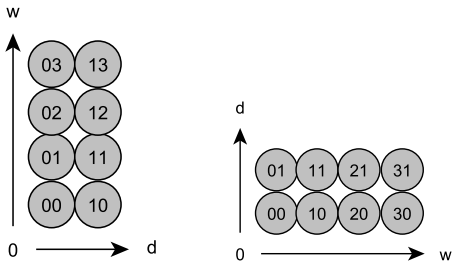
6.1.3. CHALLENGE III: INEFFICIENT LOCAL MEMORY ORGANIZATION

Local memory often works as temporal storage or plays the role of registers. In such cases, each work-item requires multiple data elements. The way that these elements are stored in local memory can have a significant impact on performance. Figure 6.3 shows two typical ways to organize the local memory space: Block and Cyclic. To avoid performance penalties (e.g., due to bank conflicts), programmers need to choose a proper way according to the access patterns of applications. Being aware of how data is organized and how accesses are mapped to banks, programmers can avoid these penalties. However, understanding the banks scheme and mapping strategy of the local memory needs a detailed code analysis, which is an error-prone and time-consuming process.



(a) Initial State (b) Mask t1 and t3 (c) Binding Switch

Figure 6.2: Work-items masking and binding switch for 4 work-items and 4 data elements. In the following rounds, some work-items are deactivated in 6.2b. Figure 6.2c shows multiple bindings.



(a) $Block\ \langle d, w\rangle$ (b) $Cyclic\ \langle w, d\rangle$

Figure 6.3: Two ways to organize the local memory space for 4 work-items that each requires 2 data elements. 'w' and 'd' represent the work-item dimension and data dimension, respectively.

Summarizing the basic operations and challenges we presented above, we specify four high-level components of our API and their functionalities (CP and CH are short for 'Component' and 'Challenge', respectively):

1. GM → LM Operations: this component has to cover the operations needed to transparently load data from global memory to local memory ($CP_1 \sim CH_I$).
2. LM → Registers Operations: this component has to provide programmers with simplified ways to address the move from local memory to registers ($CP_2 \sim CH_I$).
3. Communication Operations: this component has to enable the users to make use of local memory as a synchronization and communication mechanism ($CP_3 \sim CH_{II}$).
4. Local Memory Management Operations: this component allows programmers to efficiently organize and operate local memory when it works as temporal storage or plays the role of the scarce registers ($CP_4 \sim CH_{III}$).

6.2. ELMO DESIGN

To satisfy the requirements (the four CPs) mentioned in Section 6.1, we design ELMO as a middle layer between kernels and the basic operations for the local memory (see Figure 6.4). The basic idea is to keep the bindings between work-items and data elements transparent to users via a high-level API. The ELMO APIs consist of: (1) Block-Write Random-Read APIs (BWR), (2) APIs for Communications (COM), and (3) APIs for Local Memory Management (LMM).

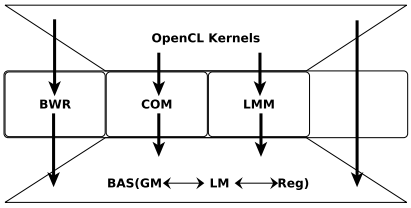


Figure 6.4: ELMO Stack Overview (BAS represents the basic operations of using local memory).

BWR, proposed to address CP_1 and CP_2, allows the data to be loaded from global memory (into the local memory) in a block-wise way and used in arbitrary (or random) patterns presented in kernels. This API includes two operations: writing data from global memory into local memory (*G2L*), and reading data from local memory into registers (*L2R*). For *G2L*, users only need to give simple information such as the global/local memory addresses and the radius (see the model in Section 6.3.1). After that, we enable the index conversion from global space to local space in *L2R*. Thus, BWR makes the process of moving data between local memory and global memory or registers transparent to the users.

This API can be widely used in real-world applications. First, BWR can be used for data sharing in applications where the data elements needed by one work-item overlap with the ones needed by its neighbors. Image convolution is a typical example of such an application. Second, the local memory can be used to explicitly convert a scattered access pattern to a regular (coalesced) pattern for read/write from/to global memory [4]. Thus, BWR is built to achieve high memory bandwidth even when the original memory access patterns are architecture-'unfriendly' (reversed or random memory access). Matrix transpose and cross-based aggregation in stereo matching [177] are representative examples of such applications.

COM, proposed to address CP_3, encapsulates the complex communication and synchronization procedures that include multiple rounds to update data elements, and thus need work-item masking and binding switches. The encapsulation hides these confusing coding details from users.

Currently, the API mainly includes aggregation operations, such as reduction, prefix sum, and scan, but other communication operations like binomial reduction (often used in BinomialOption [6]) are to be added. The aggregation involves multiple passes to read/write data elements from/into memory, possibly in the form of irregu-

lar accesses. Thus, it is expected that local memory performs better on aggregations than global memory. Further, the aggregations usually require data communication (via work-item masking and binding switches) between work-items, which means that the registers cannot be used to replace local memory.

LMM, proposed to address CP_4, aims to manage local memory space efficiently. The LMM API differs from the COM API in that the work-items within the same work-group process on their own space and do not have to communicate with each other via local memory. Thus, efficiently organizing and operating the local memory space becomes the main concern of LMM.

An example of a LMM operation is *initialization*, i.e., setting each element of a local memory region to a certain value (e.g., when computing histograms, memory has to be initialized to be zero). The operation should be straightforward, but programmers are likely to ignore the avoidance of bank-conflicts during initialization. Thus, we delegate this task to the LMM API to avoid the performance pitfall.

A different example of a LMM operation is *relocation*, needed to avoid the performance hits of register spilling. Register spilling occurs because OpenCL compilers give priority to allocating private memory for each work-item on register files (Figure 1), but these are limited in size. When an instance of a kernel consumes too many registers, register spilling occurs. The spilled data is usually transferred to global memory, which increases memory traffic and instruction count. Thus, local memory plays back up of registers via relocation.

To summarize, the front-end of ELMO comprises, by design, of three APIs that aim to enable the easy use of local memory. The design is user-oriented, decoupling the programmability issues from the performance ones (left for the back-end implementation).

6.3. ELMO IMPLEMENTATION

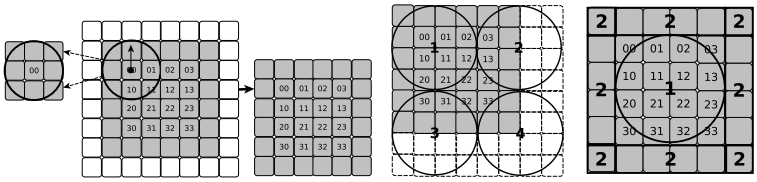
In this section, we provide implementations for each API. Specifically, we compare different implementations via micro-benchmarking, perform GPU-oriented optimizations, and give our preference on different application constraints.

6.3.1. BWR

By design, BWR consists of two separate steps: (1) G2L, loading the required data from global memory to local memory, (2) L2R, reading the data from the local memory when performing computation.

G2L When performing computation, each work-item needs to load data elements in the area of radius r, centered on its *thread index* (shown in Figure 6.5a). Note that r can be different between the top, bottom, left, and right halo data. When $r = 0$, i.e., no halo data, we use the 1-to-1 mode (between work-items and data elements) to load data, saturating the global memory bandwidth. However, when $r > 0$, the bandwidth of global memory may not be saturated, if the geometry of the input data block is not corresponding with that of a work-group. There are two ways we propose for this process (also shown in Figure 6.5b, 6.5c): (1) reading data in a **t**ile-**b**y-**t**ile fashion (*TBT*), or (2) loading the **c**entral data **f**irst, and **t**hen the **h**alo data (*FCTH*).

From Figure 6.6, we see that FCTH performs better than TBT when $r \leq 16$. This is



(a) G2L (b) TBT (c) FCTH

Figure 6.5: G2L Model: the work-group is 4 × 4, and the radius is 1. Thus, each work-item (e.g., work-item *00*) needs a 3 × 3 data block, and each work-group needs to load a 6 × 6 data block. When loading data into local memory, TBT needs 4 passes, numbered with 1, 2, 3, and 4 (the area outlined by dashed-line squares is given for illustrative purposes). FCTH first reads the central data, and then use 8 extra passes to load the halo data.

because TBT introduces extra branch overheads for a generic implementation. Further, when the dimension (width/height) of the local data block is not a multiple of that of a work-group (e.g., $r = 9$), TBT will lead to many more 'small tiles', wasting memory bandwidth. Thus, we prefer selecting the FCTH implementation when the radius is smaller than the dimension of a work-group, but we need to use TBT when this is not the case.

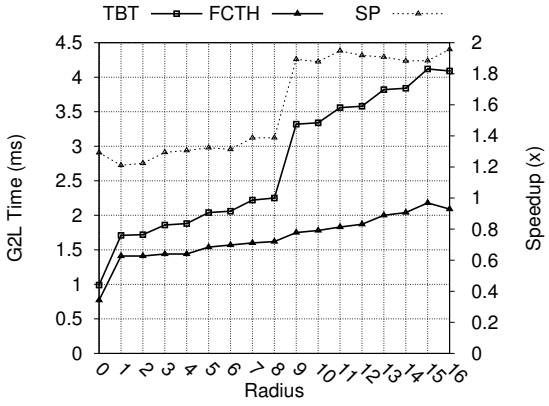


Figure 6.6: The data loading time from global memory to local memory and speedup (FCTH versus TBT). The data is obtained when the work-group is 16 × 16, and the input data is 2048 × 2048.

L2R When performing computations using data elements in local memory, the key issue is to determine the correspondence F between the index (D_{gx}, D_{gy}) of the data elements in the global data space and the index (D_{lx}, D_{ly}) of the data elements in the local data space within one work-group (shown in Equation 6.1).

$$F : (D_{gx}, D_{gy}) \rightarrow (D_{lx}, D_{ly}) \tag{6.1}$$

As we know,

$$\begin{gathered} D_{gx} = T_{gx} + \delta_x,\ D_{gy} = T_{gy} + \delta_y, \\ D_{lx} = T_{lx} + \sigma_x,\ D_{ly} = T_{ly} + \sigma_y, \\ \sigma_x = \delta_x + r,\ \sigma_y = \delta_y + r, \\ T_{gx} \sim T_{lx},\ T_{gy} \sim T_{ly}. \end{gathered} \tag{6.2}$$

From Equation 6.2, we can establish the correspondence F (δ and σ in the equations are implementation and iteration dependent parameters; (T_{lx}, T_{ly}), (T_{gx}, T_{gy}) are the local and global work-item index). When implementing the L2R API, we first make a conversion of data index from the global space to the local space, and then use the data in the local memory space.

6.3.2. COM

The implementation of the COM API is based on a generalization of [138], in which the authors present a segmented scan algorithm and its CUDA implementation. It consists of three steps: (1) intra-warp scan, (2) intra-block scan, and (3) global scan. We generalize this segmented algorithm to all the aggregation algorithms in the COM API's implementations. Since the schedule units (*warp* from NVIDIA [117] and *wavefront* from AMD [6]) differ in size/width across vendors, we start with intra-block operations, and organize the algorithms as follows:

1. Loading the input array into local memory (some applications generate on-the-fly data as input).
2. Performing aggregations at the work-group-level (within one work-group).
3. Performing global aggregation on the results from all the work-groups.

For the moment, the supported aggregation operations in ELMO are reduction, prefix sum, and scan. As a proof of concept, we illustrate the *reduction* API and its implementation as follows:

Reduction needs to aggregate (sum, avg, max, min) the data to one final value. Each invocation of the kernel reduces the input array block to a single value within one work-group; it then writes this value to the output and reduces the partial results to a final result, which is sent to the host [6]. The reduction of each work-group is done in multiple passes. In the first pass, half of the work-items are active, and they update their values in local memory by aggregating the other half. This continues as shown in Figure 6.7.

The above-mentioned reduction maps one work-item to one data element. In practice, each work-item can perform reductions on multiple data elements (i.e., *granularity coarsening*). In Figure 6.8, we see that the granularity-aware implementation performs poorly for very small granularity due to more overheads from work-item creation, branches, and synchronization. The performance also decreases slightly when the granularity is too large because of less work-groups to hide latency. Therefore, we provide a parameter *granularity* in the API, for performance tuning.

6.3.3. LMM

The back-end of LMM has to optimize the organization and management of the local memory. The two operations - the memory initialization and relocation - are used to

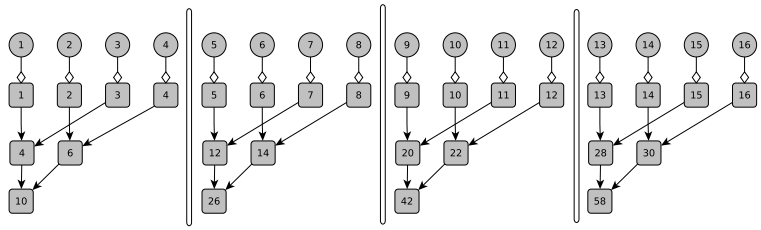


Figure 6.7: Reduction (sum): the circles represent work-items, the squares represent data elements in local memory, and the long bars separate work-items into different work-groups (4 work-items in each work-group).

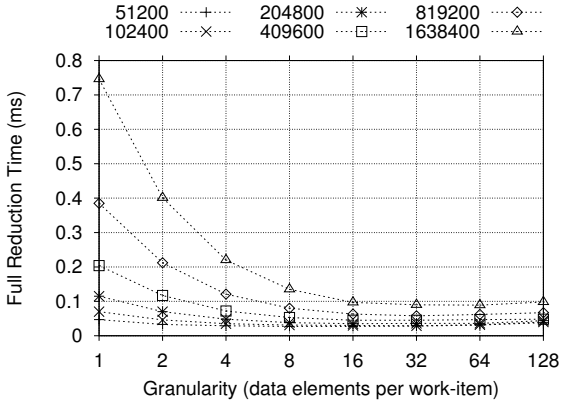


Figure 6.8: Full reduction time for different data size (51200-1638400) and granularity.

show the types of optimizations needed for LMM, their complexity and performance impacts.

Initialization Initializing the local memory is required when we perform statistics. For example, calculating a histogram needs to reset all $bins$ prior to the computation itself. Initialization is independent of local memory organization (block or cyclic), i.e., there is no inherent binding between work-items and the local memory space. We compare two typical approaches to initialize the local memory: row-major (*RMI*) and column-major (*CMI*). For RMI, neighboring work-items perform the initialization on data elements in the same row, while CMI initializes the data elements in the column-major order (by neighboring work-items).

Table 6.1 shows the performance comparison of these two approaches. We see that RMI performs much better than CMI, with speedup ranging from 2.4× to 13.5×. This happens because CMI forces all the accesses into one memory bank, leading to serializing the writing operations, while RMI can successfully avoid it. Thus, we choose to use RMI in ELMO.

P2L Using this API, we can store the likely to-be-spilled variables to local memory, rather than in private memory. The native kernels start with the cases that use *private*

Table 6.1: Performance comparison of CMI and RMI. The input data is one-dimensional, with 327680 work-items, and the work-group size is 32. N is the number of elements required by one work-item.

N	8	16	32	64	128	256
$CMI(ms)$	0.35	1.15	4.44	11.54	45.32	269.89
$RMI(ms)$	0.15	0.22	0.45	0.90	3.41	19.99
$Speedup$	2.40	5.12	9.85	12.81	13.27	13.50

memory in the form of an *array* or a *variable*. Where register spilling could occur, we transform the usage of *private* memory to the usage of *local* memory (i.e., we relocate the variables from the private to the local memory). A comparison of the two implementations is shown in Figure 6.9.

```
1  #define N 32
2  __kernel void kernel_pv(...){
3      __private uint i, j;
4      __private type a;
5      __private type b;
6      ...
7      __private type c;
8      ...
9      __private type array[N];
10
11     // write data into array
12     array[i] = a;
13     // ...
14     // read data from array
15     b = array[j];
16 }
```

```
1  #define N 32
2  __kernel void kernel_lc(...){
3      __private uint i, j;
4      __private type a;
5      __private type b;
6      // S: work-group size
7      __local type c[S];
8      ...
9      __local type array[N*S];
10
11     // write data into array
12     array[_clm_idx_p2l(i)] = a;
13     // ...
14     // read data from array
15     b = array[_clm_idx_p2l(j)];
16 }
```

Figure 6.9: Replace private memory with local memory. The left kernel shows the native implementation with *private* variables, which are replaced with *local* variables in the right kernel.

The key issue to be solved here is *index conversion* from private space to local space. Once the way of organizing the local memory is determined (in Figure 6.3), we convert the index as follows:

$$D_l = \begin{cases} T_l \cdot N + D_p & \text{if } Block \\ D_p \cdot S + T_l & \text{if } Cyclic \end{cases} \tag{6.3}$$

where D_l, D_p, T_l represent local data index, private data index, and local index of a work-item, respectively; N is the number of variables required by one work-item, and S is the work-group size. When the spilled data is a variable, rather than an array ($N = 1$ and $D_p = 0$), these two approaches will be one and the same.

Currently, we only provide users with the index conversion APIs. Users remain responsible for detecting the potential cases of register spilling (e.g., from the verbose information returned by compilers), but with the help of this API, users can be in control of the spilling and prevent expensive spills to global memory. For the future, an automated tool would be needed to detect the occurrence of register spilling, and relocate the to-be-spilled variables to local memory.

6.4. EXPERIMENTAL EVALUATION

In this section, we present our experiments with ELMO focusing on the performance improvements it brings from two angles: (1) comparing the performance with native kernels (i.e., kernels without local memory or using local memory improperly) to evaluate our implementations and optimizations in ELMO, (2) comparing the performance with hand-tuned kernels further to see how ELMO performs and how far we can go in terms of performance.

6.4.1. EXPERIMENTAL SETUP

All the experiments are performed on a NVIDIA Quadro5000 Fermi GPU. The card has Compute Capability 2.0 and consists of 352 cores divided among 11 multiprocessors. The number of 32-bit registers allocated to each multiprocessor is 32K, while the amount of local memory available per multiprocessor is 48K. We compile all the program with the OpenCL implementation from CUDA version 4.1 and GCC version 4.4.3.

Besides Reduction (RD) mentioned in Section 6.3.2, we have implemented five more representative kernels in our experiments: Image Convolution (IC), Matrix Transpose (MT), Cost Aggregation (CA), Histograms (HT), and Marching Cubes (MC) using ELMO. Furthermore, we have applied ELMO to five applications from the AMD/NVIDIA SDKs, and compare ELMO's performance to that of hand-tuned code.

6.4.2. PERFORMANCE COMPARISON WITH NATIVE KERNELS

Image Convolution We implement the kernel in three different ways: (1) Native kernel without local memory (*Native*); (2) Optimized kernel using the BWR API in *TBT* mode (OPT_{TBT}); (3) Optimized kernel using the BWR API in *FCTH* mode (OPT_{FCTH}). In Figure 6.10, we see the performance for these three implementations. When $r = 1$, the optimized kernels perform worse than the native kernel (by around 32% and 16% for OPT_{TBT} and OPT_{FCTH}, respectively). This slowdown is due to the extra overhead of accessing local memory and the branches introduced in ELMO. The gain from data sharing in this application is offset by the overhead. Thereafter, the optimized kernels can yield significant performance improvement compared to the native implementation, with speedups of $1.3\times - 2.8\times$ for OPT_{TBT} and $1.5\times - 3.1\times$ for OPT_{FCTH}.

Matrix Transpose Since $r = 0$ (i.e., no halo data), we use the FCTH approach from ELMO for the optimized implementation. The results for different implementations and multiple data sizes are shown in Figure 6.11. We see that using ELMO will improve performance significantly, especially when the input data size is large. The native kernel (without local memory) violates the coalescing access constraints either on reading or on writing. While using ELMO, the coalesced access from global memory to local memory is ensured and operating the local memory itself has no coalescing constraints. When the input matrix size is small, the performance gap is minor, because the input data fits the L1/L2 caches on Quadro5000.

For the Matrix Transpose, we use a 16×16 work-group, with a block of local memory space of the same size. When reading data elements from the local memory, bank-conflicts will occur. We remove the bank-conflicts by padding data, i.e., changing the row-size of the local memory (with an additional parameter in the BWR G2L API). Figure 6.11 shows the further performance improvement of the optimized kernel OPT_{BCR}

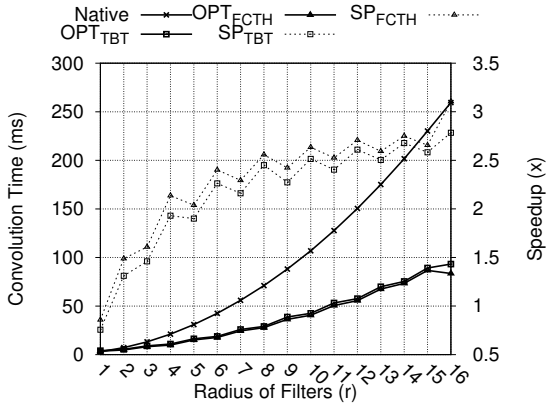


Figure 6.10: Execution time and speedup of IC for different radius and implementations. The input data size is 2048 × 2048, and the r varies from 1 to 16.

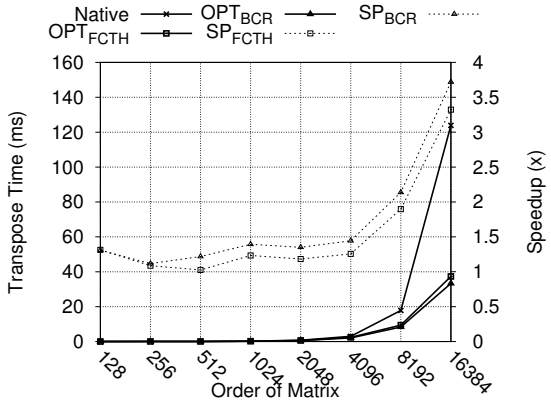


Figure 6.11: Execution time and speedup of MT for different input data.

without bank-conflicts.

Cost Aggregation is a necessary step in local-based stereo matching of computer vision [136]. Cross-based cost aggregation [178] is a typical approach, in which performing computation on each pixel depends on an adaptive area of data elements around it, and the maximum radius is limited by a pre-defined L value. This leads to un-coalesced global memory access with extremely low memory bandwidth. Thus, we use the BWR API to load all the data elements within the area of radius L (more data elements than needed).

We use the four data sets (cones, teddy, tsukuba, and venus) from Middlebury [136] to evaluate the performance. The maximum limit L is 17, which is larger than the dimension of a work-group, meaning that we can only use the *TBT* approach. Figure 6.12 shows the execution time for the two implementations ($Native$ versus OPT). We see that the optimized solver can achieve decent performance improvement, with speedup

around 1.5×. Although we load more data elements than what we need in this situation, we have achieved better performance using ELMO, due to the coalesced access from global memory to local memory.

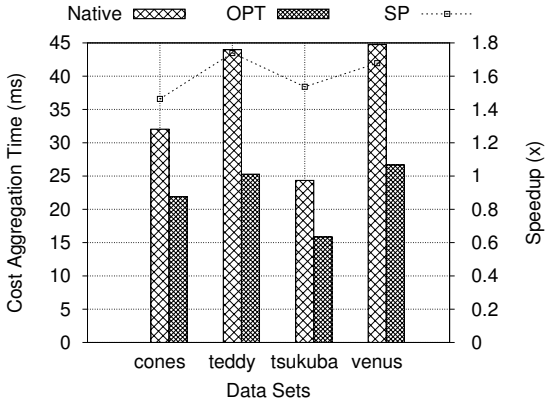


Figure 6.12: The execution time and speedup on four datasets.

Histograms Calculating a histogram requires both local memory initialization and partial results summary. Figure 6.13 shows the results for the histogram implemented using ELMO-LMM, comparing the execution time of all four combinations of RMI/CMI initialization and block/cyclic organization. The optimization on initialization accounts for more than that on partial results summary, with an average speedup 1.46× versus 1.31×. The combined optimizations can achieve a speedup of 1.27× to 2.96× compared with the native implementation.

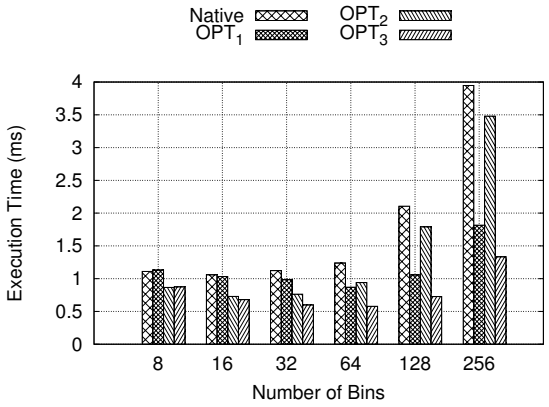


Figure 6.13: Execution time of Histograms with different optimizations. The optimizations are the in the {initialization + partial results summary} format: Native={CMI + Cyclic}, OPT1={RMI + Cyclic}, OPT2={CMI + Block}, OPT3={RMI + Block}.

Marching Cubes is a computer graphics algorithm for extracting a polygonal mesh of an isosurface from a three-dimensional scalar field. One of the five kernels implemented

in the NVIDIA SDK MC is called *generateTriangles*, which is used to calculate the flat surface normal for each triangle [118]. In the kernel, we allocate 16 $float4$ data elements for each work-item to find the vertices where the surface intersects the cube, i.e., each work-item requires more than 64 registers, exceeding the capability of Quadro5000.

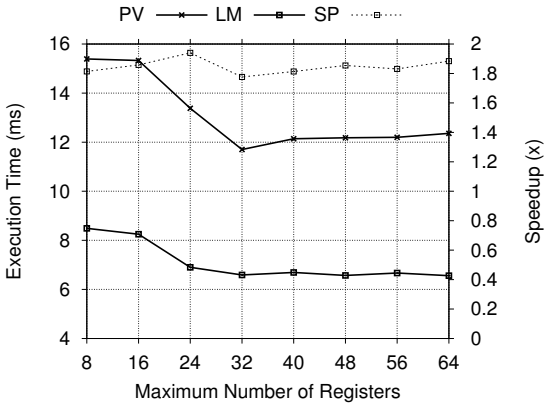


Figure 6.14: Execution time and speedup of MC with maximum register limits.

We use ELMO's P2L API and compare two implementations: PV - allocate all the variables directly on private space, and LM - allocate the array variable on local memory. The execution time for *generateTriangles* when limiting the maximum number of registers per work-item is shown in Figure 6.14, for an input grid of $32 \times 32 \times 32$. The increase in number of registers limits results in performance improvement, especially from 16 through 24 to 32. Comparing to the register-only implementation, using local memory can significantly improve the performance (up to 2× faster). This is because the compiler spills data to the global memory space for the register-only implementation, which can be avoided by using ELMO.

6.4.3. PERFORMANCE COMPARISON WITH HAND-TUNED KERNELS

To compare the performance with that of hand-tuned kernels, we select five representative equivalent kernels (i.e., kernels with the same algorithms) from AMD/NVIDIA SDKs, and apply ELMO to them. The results are shown in Table 6.2 (For CA, we cannot find a comparison reference from SDKs).

Table 6.2: Performance comparison with hand-tuned kernels.

Applications	EQ. Kernels	SDKs	DataSize	Speedup
BoxFilter	IC	NVIDIA	1024x1024	1.24
Transpose	MT	NVIDIA	2048x2048	0.97
–	CA	–	–	–
MatVecMul	RD	NVIDIA	1100x100000	0.77
Histograms	HT	AMD	2048x2048	2.63
MarchingCubes	MC	NVIDIA	32x32x32	1.04

From Table 6.2, we observe that the performance varies for different kernels. As for BoxFilter and Histograms, using ELMO performs 1.24× and 2.63× faster than the hand-crafted kernels. When looking into the hand-tuned code of BoxFilter, we notice that it uses an extended work-group (while we use the TBT approach of the ELMO BWR): the boundary work-items only participate in loading data from global memory to local memory, and remain idle for the other time-slices. The performance improvement of Histograms comes from the efficient initialization, i.e., the usage of RMI rather than CMI, enabled by ELMO.

For MatVecMul, however, ELMO performs 23% worse than the hand-tuned kernel. This is because the hand-tuned kernel uses warp-specific optimizations when performing reduction, which does not require any synchronization between two consecutive reduction passes. For the moment, we ignore these vendor-specific optimizations for portability on multiple and future platforms. Finally, we see that ELMO can achieve comparable performance with the hand-tuned versions of MarchingCubes and Transpose.

6.5. DISCUSSION

In this section, we discuss the productivity and usability of ELMO. We also discuss the situations when ELMO is not suitable.

6.5.1. PRODUCTIVITY

To estimate the productivity to be gained by using ELMO, we assume that a typical user will use one API call instead of a number of lines of code (LOC) with the same functionality. Given that our back-end is an optimized, yet generic implementation of the API, it is fair to assume that an average programmer will use 75% ~ 100% of the lines of code from ELMO's back-end, for a custom-made implementation of the same functionality. Therefore, for each ELMO API call, the code is shorter, on average, by 22 ~ 30 lines (see APIs and LOC in Table 6.3). This means that using ELMO can not only simplify code writing, but avoid code bloating from non-computation related elements.

Table 6.3: Overview of ELMO and its LOC.

Category	APIs	LOC	Implementation
BWR	*_clm_write_tbt*	38	G2L in TBT
	_clm_write_fcth	81	G2L in FCTH
	_clm_read	7	L2R
COM	*_clm_reduction*	21	Reduction
	_clm_prefixsum	40	PrefixSum
	_clm_scan	45	Scan
LMM	*_clm_idx_p2l*	5	P2L
	_clm_init	13	Initialization

6.5.2. USABILITY

To get an idea of the usability potential of ELMO, we have investigated the AMD/NVIDIA SDK applications. In total, we have found 30 applications that use local memory. Out of these, 20 can be covered by ELMO (see Table 6.4). For the remaining applications, the patterns of data transformations and communications are application-specific. Thus, future work is to investigate how to abstract these operations to more generic local memory access patterns and include them in ELMO.

Table 6.4: Applications covered (already and not-yet) by ELMO.

APIs	Applications
BWR	BoxFilter, ConvolutionSeparable, FDTD3d, MatrixMul, MedianFilter, NBody, Particles, RecursiveGaussian, SobelFilter, MatrixTranspose, LUDecomposition, QuasiRandomSequence
COM	HiddenMarkovModel, MatrixVecMul, Reduction, ScanLargeArrays, PrefixSum
LMM	Histogram, MarchingCubes, URNG
Not-yet	DCT8x8, DXTCompression, RadixSort, SortingNetworks, Tridiagonal, AESEncryptDecrypt, BinomialOption, DwtHaar1D, FFT, GaussianNoise

From a user-perspective, ELMO requires programmers to identify the APIs to be used based on their application. We aim to improve both the detection of the patterns and the choice for the right APIs in the near future, hoping these advances will allow ELMO to be a starting point for (semi-) automated tools for enabling the smart use of local memory.

6.5.3. LIMITATIONS

In practice, not all ⟨application, architecture⟩-pairs can benefit from using ELMO. For example, when performing the NBody simulation on NVIDIA Quadro5000 (with L1/L2 caches besides local memory), each body will read the states of all the others, leading to full data sharing [4]. Table 6.5 shows the execution time when simulating different number of bodies. We see the slight performance degradation when using ELMO (by around 10%). This slowdown is due to the fact that caches (L1 and L2) make better use of data sharing than the local memory. Specifically, local memory enables data sharing among work-items within one work-group, while the L1 cache can identify the data sharing within one work-group, and the L2 cache will enable data sharing globally on the input data. Additionally, using ELMO introduces extra overheads for data movement operations in and out of local memory. In this situation, one should disable the usage of ELMO for not sacrificing any performance.

Finally, we note that (at the time of writing), OpenCL for Multicore CPUs will map local memory onto global memory. All memory objects are cached by the hardware and explicitly using local memory for caching will most likely add unnecessary overheads [70]. Thus, more research is required in implementing a CPU-friendly back-end. Until then, ELMO is recommended for architectures equipped with a separate, fast local memory.

Table 6.5: The execution time (in ms) of NBody with different bodies.

#Bodies	10240	20480	40960	81920	163840	327680
Native	1.74	6.75	25.32	98.42	389.76	1585.24
ELMO	1.93	7.23	28.67	108.88	437.57	1762.64
Slowdown	1.11	1.07	1.13	1.11	1.12	1.11

6.6. RELATED WORK

In this section, we discuss prior work on tools, compilers, and optimization techniques of using local memory. Baskaran et al. [13] develop an approach to effective automatic data management for on-chip memories, including creation of buffers in on-chip memories for holding the needed data elements, determination of array access functions, and generation of code that moves data between slow off-chip memory and fast local memories. In [172], Yang et al. propose a GPGPU compiler, which converts the un-coalesced accesses to coalesced ones, and enhances the data reuse. Because of our analysis of the challenges of using local memory, we present broader and more generic uses of local memory, besides their proposed data reuse and data layout changing.

In [15], Bauer et al. present CudaDMA, an extensible API for efficiently managing data transfers between the on-chip and off-chip memory of NVIDIA GPUs. The driving force of CudaDMA is to emulate the use of asynchronous hardware DMA engines for GPUs at a software level. However, there is no DMA hardware on GPUs, which makes the asynchronous approach less fascinating. Further, the CudaDMA uses two classes of warps: DMA warps and Compute warps, taking charge of data movement and computation, respectively. This warp-specialization implementation introduces code bloat, as we can see in the paper. In contrast, we build our ELMO based on multiple models and keep the native way of using local memory, allowing our APIs to be easy-to-use and user-friendly.

ELMO also relates to skeleton programming libraries such as Thrust [73], SkePU [39], and SkeCL [146]. Thrust is a parallel algorithms library with high-level interface greatly that enhances programmer productivity while enabling performance portability between GPUs and multicore CPUs. SkePU provides a simple and unified interface for specifying data-parallel computations with the help of skeletons on GPUs using CUDA and OpenCL. The interface is also general enough to support other architectures, and SkePU implements both a sequential CPU and a parallel OpenMP backend, and supports multi-GPU systems. Similar ideas have been addressed by SkeCL. In contrast, ELMO is a programming API oriented to local memory usage.

Research on local memory optimization techniques is yet another interesting related topic. For example, in [103], Moazeni et al. present a memory reuse technique to minimize the use of local memory space. This is to address the concern that an incremental increase in the usage of local memory per thread can result in a substantial decrease in performance. In [132], Ren et al. propose a framework for automatically tuning applications to machines with software-managed memory hierarchies. Such techniques could be used to further optimize ELMO's back-ends.

Overall, our related work survey shows that (1) ELMO's design is novel and could

provide users an alternative way of using local memory in OpenCL, and (2) the back-end can be further extended with more complex, yet fairly portable optimizations.

6.7. SUMMARY

On multiprocessors with explicitly managed memory hierarchies, programmers have the responsibility of moving data in and out of the local memory for high performance. This task can be complex and error-prone even for programming experts. Thus, we propose the ELMO API to improve productivity while preserving high performance. Addressing the challenges of (1) geometry mismatch, (2) work-items masking and binding switches, and (3) inefficient local memory organization, the API presents a user-friendly interface and covers diverse using scenarios.

Our experimental results show that ELMO can improve performance by up to 3.7× on NVIDIA Quadro5000. Even when compared with hand-tuned code for using local memory, ELMO can deliver matching performance and does not hinder other types of hand-tuning. Besides, ELMO enables us to write 22 ~ 30 less lines of code per kernel, which will significantly improve productivity.

7

GROVER: REVERSE-ENGINEERING LOCAL MEMORY USAGE

For a kernel with local memory, it is necessary to remove local memory usage when we detect that using it leads to a performance degradation. In this chapter, we present a compiling pass to remove the usage of local memory automatically. We show our approach, its implementation, and experimental results.

When implementing applications, OpenCL programmers typically make explicit use of local memory in the attempt to gain performance, especially when programming GPUs. However, the architectural diversity of the underlying platforms makes the performance impacts of using local memory unpredictable [40, 44]: enabling local memory usage for the same application can lead to performance improvements for (some) GPUs and performance losses for other GPUs and/or (some) CPUs (Section 7.1). In this work, we propose an empirical solution to address this unpredictability: by disabling the use of local memory for OpenCL kernels, programmers can make a direct performance comparison between the two versions of a kernel (with and without local memory), and choose the best performing version for a given platform.

However, disabling the local memory usage requires in-depth code analysis for systematic address translation between the global and local memory spaces. Such a "reverse engineering" process is always error-prone and time-consuming, and it is especially difficult when kernels are very complex and/or designed by a third party. Instead, an automated tool is desirable to facilitate these changes. Moreover, by embedding such a tool in a compiler, the choice between kernels with and without local memory can easily become a performance auto-tuning step, and can enable code specialization for performance portability [141].

To this end, we propose Grover, a method to automatically disable local memory in OpenCL kernels, and implement it as a compiler pass. The key challenge for Grover

This chapter is based on our work published in the Proceedings of ICPP 2014 [41].

is to create a correspondence between the accesses from the local and global memory spaces. We show how we built this correspondence (Section 7.2) and how it can be used in the LLVM compiling framework as an optimization pass (Section 7.3). We evaluate our approach on 11 applications (Section 7.4) and show that Grover is able to transform all of them. Moreover, we obtain interesting performance numbers on three different platforms: by disabling local memory, 36% of the test cases show performance improvement, while 27% of them suffer a performance loss (Section 7.5). Overall, we conclude that Grover is a first auto-tuning prototype that can transparently alleviate performance losses due to ineffective, platform-unfriendly usage of local memory.

Most previous work [13, 58, 76, 84, 128, 154, 160, 162] focuses on enabling SPMs or local memory. However, we believe that in the context of performance portability over diverse processors and their different local memory implementations, "reversing" local memory usage becomes as important as enabling it. To the best of our knowledge, our work is the first study on disabling the use of local memory in OpenCL kernels. Although we focus on OpenCL in this chapter, our approach is similarly applicable to CUDA and shared memory, which can show performance losses for different generations of GPUs [40]. To summarize, the contributions of our work are as follows:

- We propose a formal approach to determine the correspondence between global and local memory spaces for a given OpenCL kernel.
- We present and demonstrate an empirical approach to detect the usage of local memory in an OpenCL kernel.
- We describe Grover, a method for disabling the use of local memory in OpenCL kernels, and its implementation as a compiler pass based on LLVM.
- We empirically prove the usability of Grover as a performance auto-tuning tool on a set of 33 test-cases (11 applications on 3 different platforms).

The rest of the chapter is organized as follows: We motivate our work in Section 7.1. At the core of the chapter, we present our method in Section 7.2 and implement it in Section 7.3. We give the experimental setup in Section 7.4 and our performance results in Section 7.5. In Section 7.6, we list the related work. Finally, we summarize this chapter in Section 7.7.

7.1. MOTIVATION

In this section, we present the challenges of disabling local memory. We show that disabling local memory usage in OpenCL kernels can lead to unexpected improved performance, a conclusion that further motivates our work.

7.1.1. DISABLING LOCAL MEMORY USAGE

Applications written in OpenCL are functionally portable: the same application runs correctly on different hardware platforms. However, the optimization strategies differ over platforms. For example, GPU programming guides recommend enabling the use of local memory as a performance booster [115], while CPU programming guides argue

```
1  __kernel void
2  MatTrans(const __global  float * in, \
3    __global  float * out, int W, int H){
4
5    __local float lm[S][S];
6    lm[ly][lx]=in[(wx*S+ly)*W+(wy*S+lx)];
7
8    barrier(CLK_LOCAL_MEM_FENCE);
9
10   float val = lm[lx][ly];
11   out[gy * H + gx] = val;
12 }
```

(a) Original code.

```
1  __kernel void
2  MatTrans(const __global  float * in, \
3    __global  float * out, int W, int H){
4
5    //__local float lm[S][S];
6    //lm[ly][lx]=in[(wx*S+ly)*W+(wy*S+lx)];
7
8    //barrier(CLK_LOCAL_MEM_FENCE);
9
10   float val = in[(wx*S+lx)*W+(wy*S+ly)];
11   out[gy * H + gx] = val;
12 }
```

(b) Remove local memory.

Figure 7.1: Removing local memory usage on Matrix Transpose. (lx, ly) is the local work-item index, (wx, wy) is the work-group index, (gx, gy) is the global work-item index, (W, H) is the global data size, (S, S) is the local data size.

for avoiding it [70]. Thus, migrating GPU-optimized code to CPUs might require some degree of code specialization [141], especially when the performance penalties are significant.

Disabling local memory usage is such a code specialization, which requires programmers to analyze kernel code, locate the candidate data structures that are placed and accessed in local memory, perform address translation between the local and global memory spaces, and remove redundant instructions. Doing the reversing work manually can be error-prone and time-consuming, which is particularly true in a complicated program context.

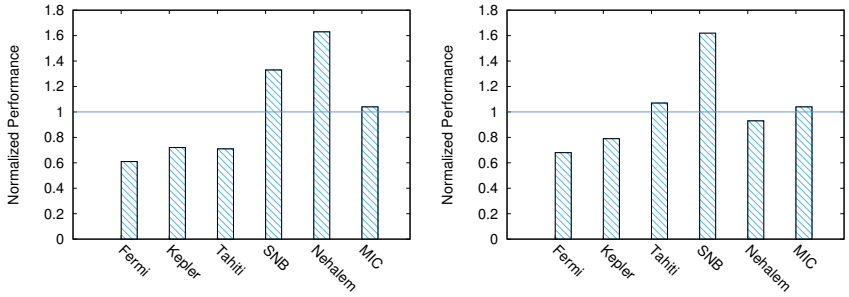
7.1.2. PERFORMANCE IMPACT

To illustrate how significant the performance impact of disabling local memory is on different platforms, we use two benchmarks from the NVIDIA SDK: MT–`Matrix Transpose` and MM–`Matrix Multiplication`. We compare their performance on 6 platforms (including GPUs, CPUs, and an Intel Xeon Phi described in Section 7.4.3).

In the original MT code (Figure 7.1a, line 6), local memory is used to stage data (i.e., cache it `in`). To disable local memory usage, we identify the candidate data structure (Figure 7.1a, line 5), manually substitute the local memory access with its corresponding global memory access (Figure 7.1b, line 10) and remove the redundant instructions (Figure 7.1b, lines 5-8), including data structure declarations and barriers. For MM, which calculates $C(i, j) = A(i, k) \times B(k, j)$, we manually remove the local memory usage for matrix A, while keeping it enabled for matrix B.

The performance impacts of these transformations, for both MT and MM, are presented in Figure 7.2. For MT, removing the local memory usage leads to performance losses on GPUs (Fermi, Kepler, and Tahiti), but improves performance for the cache-only processors (SNB, Nehalem, and MIC). The performance increase is up to 1.3× on SNB and 1.6× on Nehalem. For MM, by disabling local memory, we achieve a better performance on Tahiti, SNB (1.6×), and MIC, but we lose performance on the other three processors.

These results show that removing local memory usage can lead to (significant) performance improvement, but the cases when improvements appear are not as predictable as expected (i.e., the rule “local memory for GPUs, no local memory for CPUs” does not



(a) Matrix Transpose (b) Matrix Multiplication

Figure 7.2: The performance impacts of removing local memory on two applications (the normalized performance is ratio of the performance without local memory to that with local memory).

always hold). Therefore, we argue that, performance-wise, disabling local memory usage becomes a significant optimization for OpenCL kernels, especially in the context of inter-platform portability. In the remainder of this chapter, we show how this optimization can be applied automatically.

7.2. GROVER: SYSTEMATICALLY DISABLING LOCAL MEMORY USAGE

In this section, we present our method of disabling local memory usage, and give a brief proof for it. We finally demonstrate our method on a practical example.

7.2.1. OVERVIEW

By understanding how local memory is used - i.e., the typical patterns that appear in OpenCL kernels, we can systematically reverse the process, step-by-step. In this chapter, we focus on the most common use-case, when local memory is used as a software-controlled cache. Specifically, this pattern has two stages of interest (seen in Figure 7.3): *data storage* and *data usage*. In the *storage* stage, data is loaded from global memory (global load operation, *GL*) and stored into a data structure in local memory (local store operation, *LS*), as shown in Figure 7.1a, line 6. The *usage* stage refers to computation performed on/with the data stored in local memory (local load operation, *LL*), as shown Figure 7.1a, line 10. A synchronization (by local barrier) is required between these stages (Figure 7.1a, line 8).

When disabling local memory, we need to determine which data element in the global space (x', y', z') corresponds to a given data element in the local space (x, y, z) (Figure 7.3). Hence, the challenge is to determine a correspondence function between the global space and the local space.

TERMINOLOGY

For readability purposes, we define a brief list of important terms commonly used in this chapter.

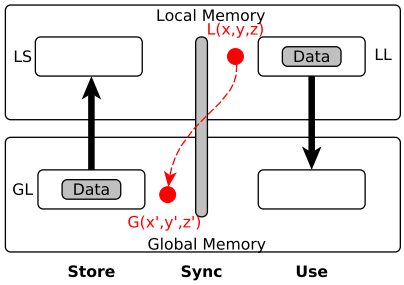


Figure 7.3: A common pattern for using local memory. The figure is simplified to only present the two stages of interest for Grover and the main operations that affect the translation.

Thread Index OpenCL defines an N-dimensional thread (or work-item) index space ($N \leq 3$). Each work-item in this space has a unique local thread index (lx, ly, lz), a global thread index (gx, gy, gz), and a work-group index (wx, wy, wz). Note that a global thread index can be calculated from the local thread index and the work-group index.

Data Index In OpenCL, data buffers are collection of data elements, each identified by a *data index*. Data elements stored in the *local memory space* are qualified by $\_\_local$, and will be further called *local data elements*, indexed by a local index (x, y, z). Similarly, data elements stored in *global memory space* are qualified by $\_\_global$, and will be called global data elements (indexed by a global data index, (x', y', z')). Our goal is to find a systematic correspondence between the global (memory) space and the local (memory) space, such that we can determine the global data index corresponding to a given local memory access.

Typically, each thread works on data elements in a region determined by its thread index. Hence, a *data index* is a function of a *thread index*: a local data index is a function of a local thread index (lx, ly, lz), while the global data index is a function of both the local thread index (lx, ly, lz) and the work-group index (wx, wy, wz).

7.2.2. THE METHOD BEHIND GROVER

Before disabling local memory, we need to select all the candidate data structures by analyzing the kernel code. We assume for now these candidates are known (the details on determining them are mentioned in Section 7.3.1), and we focus on determining the global data index with the following steps.

S1. Analyze the local memory accesses (*LS* and *LL*) and determine the local data index for both *store* and *load* operations. Here we represent the *LS* data index as (x, y, z) and the *LL* data index as (x_{LL}, y_{LL}, z_{LL}). Given that local data index is a function of local thread index, we obtain Equation 7.1.

$$\begin{cases} x = f(lx, ly, lz) \\ y = g(lx, ly, lz) \\ z = h(lx, ly, lz) \end{cases} \tag{7.1}$$

where we assume f, g, and h are linear functions of lx, ly, and lz. Thus, we further substitute these functions in Equation 7.1 to obtain Equation 7.2.

$$\begin{cases} x = a_0 \cdot lx + b_0 \cdot ly + c_0 \cdot lz + d_0 \\ y = a_1 \cdot lx + b_1 \cdot ly + c_1 \cdot lz + d_1 \\ z = a_2 \cdot lx + b_2 \cdot ly + c_2 \cdot lz + d_2 \end{cases} \tag{7.2}$$

where a_i, b_i, c_i, and d_i ($i \in \{0,1,2\}$) are constants for a local memory usage. Similarly, we can represent the *LL* data index x_{LL}, y_{LL}, and z_{LL} as a function of local thread index. However, we consider them to be constants here, and seek to determine the global data index for the local data index (x_{LL}, y_{LL}, z_{LL}).

S2. Create a linear system and solve it for (lx, ly, lz). From the previous step, we can establish a system of linear equations in Equation 7.3, where lx, ly, and lz are the *unknowns* and a_i, b_i, c_i, and d_i ($i \in \{0,1,2\}$) are the coefficients, and x_{LL}, y_{LL}, z_{LL} are the constant terms of the system. By solving the system, we obtain the solution (lx, ly, lz).

$$\begin{cases} a_0 \cdot lx + b_0 \cdot ly + c_0 \cdot lz + d_0 = x_{LL} \\ a_1 \cdot lx + b_1 \cdot ly + c_1 \cdot lz + d_1 = y_{LL} \\ a_2 \cdot lx + b_2 \cdot ly + c_2 \cdot lz + d_2 = z_{LL} \end{cases} \tag{7.3}$$

We note that the global data index is reversible if the system has a single unique solution. Consequently, when the system does not have a unique solution, Grover will not be able to cancel the use of the local memory for that particular case.

S3. Analyze the *GL* operation to determine G, a function of the the work-group index *and* the local thread index: $(x', y', z') = G((wx, wy, wz), (lx, ly, lz))$.

S4. Substitute the solution of the system in G to find the new global index. $G((wx, wy, wz), (\mathsf{lx}, \mathsf{ly}, \mathsf{l}$ is then the global data index corresponding to the local data index (x_{LL}, y_{LL}, z_{LL}).

Proof. By analyzing the process of loading data elements from the global space to the local space, we establish a correspondence G (G is determined for a memory access).

$$(x, y, z) \rightarrow \mathsf{G}(x, y, z). \tag{7.4}$$

With Equation 7.1 and the global data index expression (in *S3*), we derive Equation 7.5 from Equation 7.4.

$$\begin{aligned} &(f(lx, ly, lz), g(lx, ly, lz), h(lx, ly, lz)) \\ &\quad \rightarrow \mathsf{G}((wx, wy, wz), (lx, ly, lz)). \end{aligned} \tag{7.5}$$

Given a local data index (x_{LL}, y_{LL}, z_{LL}), we obtain

$$(x_{LL}, y_{LL}, z_{LL}) \rightarrow \mathsf{G}((wx, wy, wz), (\mathsf{lx}, \mathsf{ly}, \mathsf{lz})),$$

where the work-group part (wx, wy, wz) stays the same for the threads within a work-group and (lx, ly, lz) is the solution to Equation 7.3. Therefore, $\mathsf{G}((wx, wy, wz), (\mathsf{lx}, \mathsf{ly}, \mathsf{lz}))$ is the global data index of the local data index (x_{LL}, y_{LL}, z_{LL}). □

7.2.3. AN EXAMPLE: MATRIX TRANSPOSE

To illustrate how Grover's method works, we discuss the Matrix Transpose example (see Figure 7.1a). Furthermore, to bridge to the implementation phase (Section 7.3), we introduce *index expression trees* (Figure 7.4 and Figure 7.5) as a notation for data indexes. In an expression tree, the leaves are operands, such as constants or variable names, and the internal nodes contain operators, such as additions (+) and multiplications (∗). By applying the operator to the operands, we evaluate an index expression tree and obtain the data index.

S1. By analyzing the code in Figure 7.1a, we abstract the *LS* data index as (lx, ly) and the *LL* data index as (ly, lx). Accordingly, their expression trees are shown in Figure 7.4. Note that *S* is the width of the allocated local data space.

S2. We know that,

$$LS: \begin{cases} x = lx \\ y = ly \end{cases}, \quad LL: \begin{cases} x_{LL} = \underline{ly} \\ y_{LL} = \underline{lx} \end{cases},$$

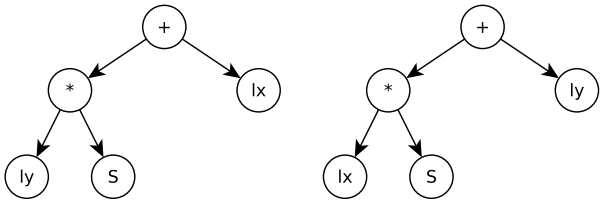
where we underline the *LL* data index (constant terms of Equation 7.3) to differentiate it from the *LS* data index (unknowns of Equation 7.3). We then create a system of linear equations as,

$$\begin{cases} lx = \underline{ly} \\ ly = \underline{lx} \end{cases}$$

It is straightforward to get the solution of this system, i.e., (lx, ly) = $(\underline{ly}, \underline{lx})$.

S3. By analyzing the code in Figure 7.1a, we obtain that the *GL* data index is $((wy, wx), (lx, ly))$ and its index expression tree is shown in Figure 7.5a. We note that, due to a more complicated index composition (work-group index and local thread index), the index tree has more levels than the local index tree (see Figure 7.4).

S4. We update the global data index with the solution $(\underline{ly}, \underline{lx})$ and obtain $((wy, wx), (\underline{ly}, \underline{lx}))$, which is the new global load index shown in Figure 7.5b.

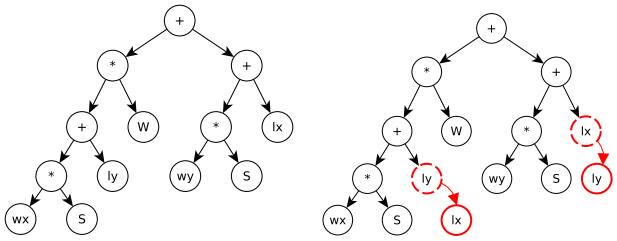


(a) Local store. (b) Local load.

Figure 7.4: Local access data index in expression tree.

7.3. GROVER IMPLEMENTATION

Ideally, *Grover* transforms any OpenCL kernel that uses local memory into a version without local memory usage. To achieve this goal, we need to create the new global



(a) Global load. (b) New global load.

Figure 7.5: Global load data index in expression tree.

load instruction *nGL*, and its index-related instructions. Therefore, we implement the reversing algorithm (in Section 7.2.2) in the following six steps:

1. Selecting the reversing candidates (Section 7.3.1).
2. Building the index expression trees (Section 7.3.2).
3. Determining the data index (Section 7.3.3).
4. Creating and solving the linear system (Section 7.3.4).
5. Duplicating the new load instruction (Section 7.3.5).
6. Updating the new expression tree (Section 7.3.6).

7.3.1. SELECTING CANDIDATES

Before removing local memory usage, we need to select the candidate data structures and detect the three operations: *GL*, *LS*, and *LL*. To do so, we first investigate all the *GL* operations in the kernel and check whether their paired store operations are *LS* operations. Next, we locate all *LL* operations that use the same data structures as the identified *LS* operations.

We note that there are applications - such as `image convolution`, for example - where multiple passes are required to load data from global memory to local memory. This means that the detection phase will identify more (*GL*, *LS*) pairs. However, using any of the pairs leads to the same correspondence between the global and the local space. Hence, we can choose any one of these pairs.

7.3.2. BUILDING THE INDEX EXPRESSION TREES

To enable the transformations mentioned in Section 7.2.2, we use the *index expression tree* to represent a data index, as introduced in Section 7.2.3. Figure 7.6 shows the tree node data structure, which has four fields: (1) the *value* field, (2) the *state* field, (3) the pointers to its children nodes, and (4) the pointer to its parent node. The *value* field can be an instruction, a built-in function, a constant number, or an argument. The *state* field is designed for a special requirement: to mark whether the current node needs to

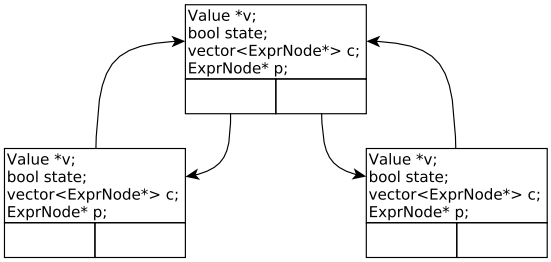


Figure 7.6: Tree node structure (*ExprNode*).

update the data index. To facilitate tree traversing, the structure also contains pointers to its children nodes and its parent node.

The index expression tree for a memory access is built recursively. The internal nodes of an index tree can have one child like a *type cast instruction*, or two children like an *addition instruction*. Thus, an index expression tree can have one (or two) sub-tree(s). The recursive algorithm stops when the *value* is one of the follows: (1) a *call instruction*, (2) a *constant number*, (3) a *function argument*, or (4) a *phi node*. In this way, we can build the index expression trees *GLTree*, *LSTree*, and *LLTree* for *GL*, *LS*, and *LL*, respectively.

7.3.3. DETERMINING THE DATA INDEX

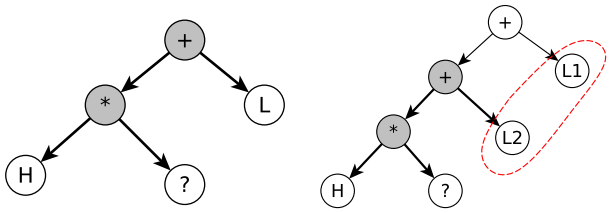
To create the linear equations (in *S2* of Section 7.2.2), we need to specify the data index of *LS* and *LL* from their index expression trees *LSTree* and *LLTree*. In this chapter, we use a pattern (Figure 7.7a, a 2D example) to identify the data index: when traversing an index tree top-down, we first find the '+' node (a node with an addition instruction), which splits the high dimension (H) and the low dimension (L). The high dimension is further identified by the '$*$' node (a node with a multiply instruction). Note that it can also be a *shift-left* operation instead of a '$*$' operation. When it comes to a 3D example, we use a similar way to determine the *LS* data index (x, y, z) and the *LL* data index (x_{LL}, y_{LL}, z_{LL}).

In real-world cases, the pattern can be more complicated as shown in Figure 7.7b, where `L1` is a loop-dependent term while the others are independent of the loop. Therefore, the `L1` term lies at the second-level of the tree, to avoid computing the loop-independent terms repeatedly. However, the '$+ \rightarrow *$' pattern (in Figure 7.7a) remains the same. We consider the '$+ \rightarrow + \rightarrow *$' pattern as a derived one and use a special case to handle it. In this way, we get the low dimension of the data index as $L1 + L2$ (see Figure 7.7b).

7.3.4. CREATING AND SOLVING THE LINEAR SYSTEM

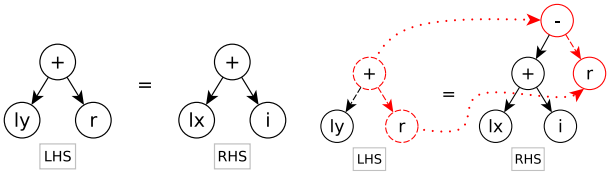
Based on the data index of *LS* and *LL* obtained in the previous step, we can create a system of linear equations, each of which has a left-hand-side (LHS) term and a right-hand-side (RHS) term. To obtain (lx, ly, lz) from the system, we need to simplify the LHS term. At the same time, complementary operations are required on the RHS term.

Figure 7.8 shows an example of how we simplify the LHS terms. The LHS term is $(ly + r)$ and the RHS term is $(lx + i)$. By subtracting r from both sides, we can obtain ly on the LHS and get the RHS expression tree $((lx + i) - r)$ (Figure 7.8b). Similarly, other



(a) Data index pattern. (b) A derived pattern.

Figure 7.7: Data index patterns.



(a) Initial form. (b) Final form.

Figure 7.8: Simplifying the left-hand-size terms.

complementary operations are equally required for a more complicated linear system.

7.3.5. DUPLICATING THE NEW LOAD INSTRUCTIONS

Creating a new global load instruction (*nGL*) includes creating the load instruction itself and the instructions for the calculation of its index-related instructions. We need to prepare these two parts when replacing the *LL* instruction. Since the original global load operations (*GL*) might be used somewhere else in the code (other than the one used by the local store), it is not safe to re-use and update *GLTree*. Hence, we need to duplicate *GL* (as *nGL*) and its index-related instructions.

To do so, two sub-steps are required: (1) update the state of each tree node, and (2) create the index-related instructions. First, we mark the nodes that need instruction duplication in *GLTree*. To this end, we locate the lx, ly, and lz which are the nodes to be replaced. From there, all the nodes preceding these nodes are marked as *to be updated*. We backtrack the tree until we reach the root node. We reuse the sub-expressions that are shared by the *GL* instruction and the *nGL* instruction when it is not required to update the node.

With the updated expression tree, we duplicate the instructions and insert them before the *LL* instruction. When inserting instructions into the kernel, special care on the expression construction order is needed. Here we use the post-order DFS approach to traverse the tree and create the required instructions (Algorithm 1). Note that we need to set the *use-and-value* relationship between instructions. In this way, we can create the *nGL* and its index calculation instructions, while keeping the *GL* instruction.

Algorithm 1: Duplicating instructions algorithm.

```
input  : A pointer to the current node node
input  : A position to insert the instruction pos
output : A pointer to the newly created value v
duplicateInst(node, pos)
begin
    v ← getValue(node);
    if isCallInst(v) or isConst(v) or isArgs(v) or isPHI(v) then
        if node.state then
            oldInst ← getInst(v);
            newInst ← cloneInst(oldInst);
            insertInst(newInst, pos);
            return newInst;
        return v;
    else
        nChildren ← getNumOfChildren(node);
        if nChildren == 1 then
            child ← getChild(node, 0);
            vL ← duplicateInst(child, pos);
            if node.state then
                oldInst ← getInst(v);
                newInst ← cloneInst(oldInst);
                insertInst(newInst, pos);
                setOperand(newInst, vL);
                return newInst;
            return v;
        if nChildren == 2 then
            lChild ← getChild(node, 0);
            rChild ← getChild(node, 1);
            vL ← duplicateInst(lChild, pos);
            vR ← duplicateInst(rChild, pos);
            if node.state then
                oldInst ← getInst(v);
                newInst ← cloneInst(oldInst);
                insertInst(newInst, pos);
                setOperand(newInst, vL);
                setOperand(newInst, vR);
                return newInst;
            return v;
```

7.3.6. UPDATING THE NEW EXPRESSION TREE

Once we have introduced the *nGL* instruction, a new index expression tree *nGLTree* is built with the method mentioned in Section 7.3.2. Thereafter, starting with the root node of *nGLTree*, we locate the lx, ly, and lz nodes and substitute them with the solution

(lx, ly, lz) obtained in Section 7.3.4. Note that type casting instructions might be required when performing the substitution. Finally, all the uses of the *LL* instruction are replaced with the *nGL* instruction.

7.4. EXPERIMENTAL SETUP

This section discusses how we incorporate our compiler pass with vendors' run-time, the benchmarks, and the devices used in the experiments.

7.4.1. INCORPORATING GROVER

To allow Grover to be fully automated, we have implemented it in the LLVM/Clang (v3.2) compiling framework. The pipeline is shown in Figure 8.2. Taking OpenCL C kernels, the LLVM front-end (Clang) transforms them into the SPIR [59] format (LLVM IR format). Thereafter, *Grover* analyses the SPIR code and removes local memory usage (when detecting any). Our framework then exports the SPIR kernels into vendor-specific run-time such as Intel SDK for OpenCL Applications 2013.

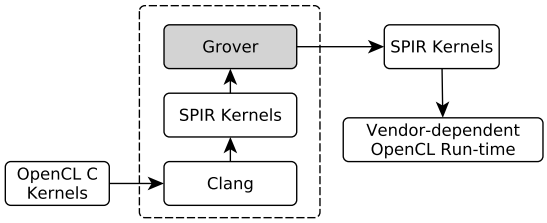


Figure 7.9: Incorporating our grover.

7.4.2. SELECTED BENCHMARKS

For evaluating Grover, we selected 11 applications from the AMD SDK, the NVIDIA SDK, the Rodinia benchmarking suite and the Parboil suite. The applications and the datasets we used in the experiments are listed in Table 7.1. All these applications are making use of local memory in their original versions that fit the pattern mentioned in Section 7.2. We use Grover to disable local memory usage and compare the performance of the kernels (the average of the kernel execution time over 20 runs) before and after Grover. The `oclMatriMul` application is a special case: the SDK uses local memory on two data structures (`Matrix A` and `Matrix B`). We remove them one by one, obtaining three versions of the code: `NVD-MM-A` (removing local memory for `Matrix A`), `NVD-MM-B` (removing local memory for `Matrix B`), and `NVD-MM-AB` (removing both).

Finally, we note that although the choice of the work-group size has a significant impact on the benchmark performance [139], selecting the optimal work-group size is beyond the scope of this work (i.e., we focus on the local memory disabling, not on optimizing the overall performance of either of the kernel versions). Therefore, all the experiments use the default work-group size settings, as specified in the original benchmarks.

Table 7.1: OpenCL benchmarks.

Benchmark	Source	ID	DataSet
StringSearch	AMD SDK	AMD-SS	StringSearch_Input.txt
MatrixTranspose	AMD SDK	AMD-MT	10240x10240
RecursiveGaussian	AMD SDK	AMD-RG	RecursiveGaussian_Input.bmp
MatrixMultiplication	AMD SDK	AMD-MM	2048x2048x2048
oclTranspose	NVIDIA SDK	NVD-MT	10240x10240
oclMatrixMul	NVIDIA SDK	NVD-MM-A	A(800 x 1600), B(800 x 800)
oclMatrixMul	NVIDIA SDK	NVD-MM-B	A(800 x 1600), B(800 x 800)
oclMatrixMul	NVIDIA SDK	NVD-MM-AB	A(800 x 1600), B(800 x 800)
oclNbody	NVIDIA SDK	NVD-NBody	163840
stencil	Parboil	PAB-ST	small
streamcluster	Rodinia	ROD-SC	(10 20 256 65536 65536 1000)

Table 7.2: The platforms used for our experiments. The GPU devices do not support SPIR, and were only used when manual kernel transformations were applied (Section 7.1)

.

Name	Fermi	Kepler	Tahiti
Host	Intel Xeon E5620	Intel Xeon E5620	Intel Xeon E5620
Host OS	CentOS v6.2	CentOS v6.2	CentOS v6.2
Device	NVIDIA Tesla C2050	NVIDIA Tesla K20m	AMD HD7970
GCC	v4.4.6	v4.4.6	v4.4.6
OpenCL	CUDA v5.5	CUDA v5.5	AMD APP v2.8
SPIR	No support	No support	No support
Name	**SNB**	**Nehalem**	**MIC**
Host	Intel Xeon E5-2620	Intel Xeon X5650	Intel Xeon E5-2620
Host OS	CentOS v6.2	CentOS v6.2	CentOS v6.2
Device	Intel Xeon E5-2620	Intel Xeon X5650	Intel Xeon Phi 5110P
GCC	v4.4.6	v4.4.6	v4.4.6
OpenCL	Intel OCL SDK v3.2.1	Intel OCL SDK v3.2.1	Intel OCL SDK v3.2.1
SPIR	ver 1.2	ver 1.2	ver 1.2

7.4.3. PLATFORMS AND DEVICES

At the moment of writing [1], only Intel has released an OpenCL implementation supporting SPIR [59]. NVIDIA and AMD have no SPIR support yet. Thus, we have run and compared the benchmarks on three devices from Intel (Nehalem, SNB, and MIC), whose configurations are shown in Table 7.2. While this selection of devices might lead to a certain bias in the performance gain/loss ratio after using Grover (as CPUs are likely to benefit more from disabling local memory [70] than GPUs), it does not challenge the correctness of this proof-of-concept: the kernels Grover builds will be portable and thus execute correctly on any devices that support SPIR.

[1] February, 2014.

7.5. PERFORMANCE EVALUATION AND DISCUSSION

In this section, we first use Grover to test whether we can disable local memory usage for each benchmark. We also evaluate the performance impact of this transformation and discuss the method limitations.

7.5.1. CALCULATING THE NEW DATA INDEX

Table 7.3 shows the data index of *nGL* for each benchmark. We first abstract the index of *GL*, *LS*, and *LL*. With the approach proposed in Section 7.2.2, we calculate the data index of *nGL*. For `AMD-SS`, `NVD-NBody`, and `ROD-SC`, the work-group index is zero. This is because all the work-items share the same data block. For example, the pattern string in `AMD-SS` is shared by all the work-items. Note that *wx, wy, wz, lx, ly, lz* represent the work-group and work-item indexes; all the other symbols are application specific. After the transformation, each benchmark still runs correctly, proving the correctness of the changes.

Table 7.3: Determining the data index of *nGL*.

ID	GL	LS	LL	nGL
AMD-SS	((0, 0, 0), (lx, 0, 0))	(lx, 0, 0)	(i, 0, 0)	((0, 0, 0), (i, 0, 0))
AMD-MT	((wx, wy, 0), (lx, ly, 0))	(lx, ly, 0)	(lx, ly, 0)	((wy, wx, 0), (lx, ly, 0))
NVD-MT	((wx, wy, 0), (lx, ly, 0))	(lx, ly, 0)	(ly, lx, 0)	((wx, wy, 0), (ly, lx, 0))
AMD-RG	((wx, wy, 0), (lx, ly, 0))	(lx, ly, 0)	(lx, ly, 0)	((wx, wy, 0), (lx, ly, 0))
AMD-MM	((wx, wy, 0), (lx+i*S, ly, 0))	(lx, ly, 0)	(j, ly, 0)	((wx, wy, 0), (j+i*S, ly, 0))
NVD-MM-A	(lx+a, ly, 0)	(lx, ly, 0)	(k, ly, 0)	(k+a, ly, 0)
NVD-MM-B	(lx+b, ly, 0)	(lx, ly, 0)	(lx, k, 0)	(lx+b, k, 0)
NVD-MM-AB	–	–	–	–
NVD-NBody	((0, 0, 0), (i+lx, 0, 0))	(lx, 0)	(_i, 0)	((0, 0, 0), (i+_i, 0, 0))
PAB-ST	((wx, wy, 0), (lx, ly, k))	(lx, ly, 0)	(lx, ly, 0)	((wx, wy, 0), (lx, ly, k))
ROD-SC	((0, 0, 0), (x, lx, 0))	(lx, 0, 0)	(i, 0, 0)	((0, 0, 0), (x, i, 0))

7.5.2. RESULTS SUMMARY

The performance results on SNB, Nehalem, and MIC are shown in Figure 7.10. The baseline performance is obtained when using local memory. The normalized performance (*np*) is the ratio of the performance without local memory to that with local memory. When this ratio is close to 1 (within 5%), the two versions of the kernel have *similar* performance. For ratios below 1, Grover's pass leads to *performance losses*, while for ratios larger than 1 we speak about *performance improvement*.

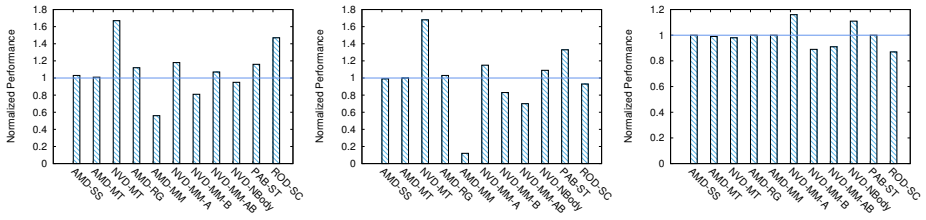
We show the overall performance gain/loss distribution for a similarity threshold of 5% in Table 7.4. In total, for our 33 test-cases, 36% benefit from disabling local memory, while 27% show performance loss.

7.5.3. PERFORMANCE ANALYSIS

We further analyze the performance results on SNB, Nehalem, and MIC, as shown in Figure 7.10. We observe that disabling local memory leads to varied performance results. On SNB, we observe speedups of 1.67×, 1.12×, 1.18×, 1.07×, 1.16× for `NVD-MT`, `AMD-RG`,

Table 7.4: Performance gain/loss distribution

	SNB	Nehalem	MIC	Total (%)
Gain	6	4	2	12 (36%)
Loss	2	4	3	9 (27%)
Similar	3	3	6	12 (36%)



(a) On SNB (b) On Nehalem (c) On MIC

Figure 7.10: Normalized performance on three devices.

`NVD-MM-A`, `NVD-MM-AB`, `PAB-ST`, respectively. The kernel performance drops by 44% for `AMD-MM`, 19% for `NVD-MM-B` and 5% for `NVD-NBody`. For `AMD-SS`, `AMD-MT`, and `AMD-RG`, the performance is only marginally affected.

We noticed a significant performance increase for `NVD-MT` on SNB. To ensure all global reads and writes are coalesced, `NVD-MT` uses local memory to stage data on GPUs. On CPUs, however, the coalescing rules are not necessary. Further, the work-items within a work-group are usually mapped onto a hardware thread [61], which is a kind of *tiling* and implicitly enables data locality. The reason for the performance increase is the same for `AMD-RG`. For `AMD-MT`, removing local memory brings little gain due to the explicit usage of vector data types (i.e., each work-item works on 4×4 matrix elements).

For `NVD-MM-A`, SNB shows performance improvement. When analyzing the memory access of `Matrix A`, we noticed that each work-item needs a row of data elements from it. On GPUs, this generates a large amount of data reuse among the work-items of the same work-group. When it comes to CPUs, this data reuse is implicitly exploited by the on-chip caches. That is, the data elements (in the form of cache-lines) loaded by a work-item can be reused by its neighboring work-items. In this way, using local memory itself becomes an extra overhead, compared with only using the caches. Therefore, removing local memory gives a performance increase. The explanations stay the same for `PAB-ST`.

We noticed a performance drop on `AMD-MM` and `NVD-MM-B` by removing local memory usage. We found that the application uses local memory on the column-wise accessed matrix. For such a case, using local memory changes the data layout, and ensures that data elements are reused before they are kicked out of caches. Meanwhile, `AMD-MM` uses data in a row-major manner, but it exploits vector data types, which changes the memory access pattern to be column-major. Thus, removing local memory is also detrimental to `AMD-MM`'s performance.

By removing local memory, we see a performance drop on `NVD-NBody` on SNB, but a

slight speedup on Nehalem and MIC. This happens because in `NVD-NBody`, each work-item needs to access all the input data elements (bodies). Thus, the work-items within a work-group will access the same element simultaneously. The pattern should be identified by caches and therefore we expect that removing local memory should give a performance increase (the reasons behind the performance loss on SNB are under investigation). A similar observation can be made about `ROD-SC`: performance increases on SNB, but decreases on Nehalem and MIC. We expect removing local memory to lead to a performance boost, due to all work-items sharing a small array of 16 data elements, stored far from each other (not in a cacheline). When using local memory, these elements are gathered and stored contiguously in the local space. Thus, this case resembles `NVD-MM-B`, which gives a better cache utilization by using local memory.

In general, Nehalem and SNB show similar performance trends when we disable local memory (Figure 7.10b), with the exception of the number for `NVD-MM-AB`. MIC behaves significantly different: we observe that most applications have similar performance with and without using local memory; only minor differences can be observed for `NVD-MM-A/B/AB`. This is mainly because MIC has a different cache hierarchy: a distributed last-level cache, compared with a unified one on Nehalem and SNB. This architectural difference minimizes the performance gaps between with and without local memory.

Overall, disabling local memory still leads to unpredictable performance outcomes on cache-only processors from different generations. Thus, the empirical exploration of Grover remains the ideal approach for choosing the best performing version of a kernel for a given platform.

7.5.4. LIMITATIONS

Grover can successfully remove local memory usage, but it has its limitations. Being built based on a common use-case (i.e., a local memory usage pattern), the tool is applicable for all kernels fitting this use-case. For other use-cases - e.g., when local memory is used as temporal storage for repeated read/write operations - the analysis must be adjusted. However, in our experience, such applications (e.g., `reductions`), typically benefit from using local memory [44] on any platform.

Furthermore, using local memory often leads to implementing tiling, and the resulting code may have a different algorithm (code skeleton) than its original (sequential) form. Grover transforms the code into a version without local memory, but not to its original form. Such a case can be seen with our tests with `NVD-NBody`, which achieves a better performance with the original form.

7.6. RELATED WORK

We list here several directions of past research that are related to this work: address translation, enabling local memory, special compiling of GPU code for CPUs, and special compiler passes for performance improvement.

Address translation between a large (global) space and a small (local) space is an old research topic in operating systems and computer architecture. For example, address translation between virtual and physical memory requires a mapping stored in a *page*

table, where each entry includes flags and a frame number [149]. Similarly, to find a line in a cache, both the slot bits and tag bits are used to correlate the global address and local cache address [62]. In our work, this correspondence needs is neither fixed, nor static: it needs to be investigated and built for any kernel.

Enabling local memory has been studied extensively for the SPMs on GPUs. Most studies focus on identifying data reuse (e.g., using a polyhedral model) [13, 58, 84, 160, 162] and exploit it by enabling local memory. Alternatively, in [128], the authors present a fully automated C-to-FPGA framework, including an end-to-end solution for on-chip buffer optimization that automatically detects and implements the available date reuse in a loop nest. However, all such studies focus on data access patterns and its exploitation, while we focus on code analysis for disabling local memory.

Several API-based approaches have also been proposed to enable local memory. In [15], the authors present CudaDMA, an extensible API for efficiently managing data transfers between the on-chip and off-chip memories of GPUs. In [44], the authors present a user-friendly API, ELMO, based on identifying patterns of local memory usage. We got inspired by the idea of local memory usage patterns in these papers, but our approach is fully automated for a given local memory usage pattern.

Compiling GPU kernels for CPU architectures is another old challenge. For example, MCUDA [148] is a source-to-source translator from CUDA for GPU architectures to multi-threaded C for multi-core CPU architectures. It serializes the work of a thread block within a single CPU thread and parallelizes the work of the kernel at thread block granularity. In [61], the authors present *Twine Peaks*, a software platform for heterogeneous computing that executes code originally targeted for GPUs efficiently on CPUs as well. In particular, the system maximizes the utilization of functional units in the CPUs by exploiting the data locality and data parallelism exposed by the GPGPU model through computation kernels. Neither of these systems addresses explicitly the local/shared memory issue, opting instead for generic transformations of the code.

Compiling passes for improving OpenCL's performance are being increasingly popular in search of performance portability. In [173], Yang et al. propose a source-to-source translator (based on the Cetus compiler framework) to address two major challenges: effective utilization of GPU memory hierarchies and judicious management of parallelism. In [79], Ralf Karrenberg and Sebastian Hack present a language- and platform-independent code transformation that vectorizes a function given by an arbitrary control flow graph in SSA form. They present a data-flow analysis that determines which code regions have constraints for vectorization concerning alignment and consecutiveness. In [92], Alberto Magni et al. consider *thread-coarsening* of OpenCL kernels and evaluate its effects across a range of devices based on LLVM. In this context, our approach is unique in finding a pass that has not been yet explored: the disabling of local memory.

To summarize, ours is the first study focused on automatically disabling the usage of local memory in OpenCL kernels through a compiler pass. While we were inspired by previous studies on compiler passes for OpenCL code optimization and/or specialization, address translation research, and local memory enabling, our simple and functional approach is novel in goal, method, and implementation.

7.7. SUMMARY

While functional portability is ensured by the OpenCL platform model and the efforts of the hardware vendors to properly support it, performance portability remains a challenging task. Thus, several platform-specific optimizations have to be enabled or disabled when porting kernels from one device to another. One example of such an optimization is the use of local memory, rendered unpredictable by the diversity of hardware platforms and OpenCL mappings.

In this chapter, we have shown evidence that using local memory can lead to significant performance penalties for many devices - both GPU and CPU platforms. Thus, we proposed an approach for reverse-engineering OpenCL kernels with local memory. Specifically, we designed and implemented Grover, a method for automatically removing local memory usage from OpenCL kernels in search for these performance improvements. Grover is implemented as a compiler pass, which enables programmers to autotune the use of local memory (i.e., on/off) to obtain the best performing version of a kernel for a given platform.

We have validated Grover on a set of 11 applications, showing that the local memory usage has been correctly canceled. For more than a third of the 33 test-cases (11 applications on 3 different platforms), we observed performance improvements. Thus, Grover can be used for exploiting potential performance improvements due to removing local memory usage in OpenCL kernels. We believe Grover proves that the performance portability of OpenCL codes can be improved by automated code specialization.

8

SESAME: TOWARDS A PORTABLE PROGRAMMING FRAMEWORK

To achieve high performance across platforms, we have proposed in previous chapters code specialization techniques for vectorization and local memory usage. We have shown that performance can be largely preserved by enabling platform-specific optimizations with a unified programming model. In this chapter, we propose a generic framework to improve the performance portability of parallel applications beyond beyond vectorization and local memory usage, and sketch the design of such a generic framework.

8.1. A REALISTIC SCENARIO

It is often the case that poor cross-platform performance of parallel applications is due to non-portable optimizations. Typically, a domain expert writes code with a unified programming model, performs some optimizations as recommended in the "Best Practice Book", and obtains decent performance on a specific platform. However, when running in another context, this application might lead to disappointing performance.

In previous chapters, we have shown how OpenCL is suffering from non-portable optimizations. However, this also happens in other programming models. For example, working with OpenMP, a traditional shared memory programming model [29], we observed similar results on Intel MIC: using loop tiling leads a much better utilization on a traditional multi-core CPU than on MIC. Using additional pragmas, OpenMP has been extended to support programming on many-core processors (like Intel MIC), and legacy code parallelized in OpenMP can now run on the many-core processors with minor modifications. For example, we ported BT-MZ (the multi-zone version of the Block Tri-diagonal solver in the NAS Parallel Benchmark [74]) written in OpenMP onto Intel MIC. The obtained performance, illustrated in Figure 8.1, peaks at around 30 GFLOPS. Besides BT-MZ's irregular memory accesses, the low performance (3% of the peak) is

This chapter is based on our work published in the Proceedings of CCGrid 2013 [46].

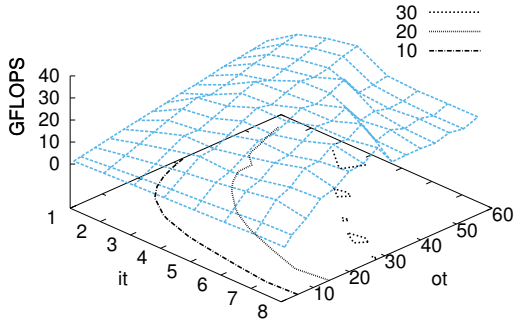


Figure 8.1: FLOPS obtained from BT-MZ when varying the number of outer threads (ot) and inner threads (it).

due to the architecture disparities with the traditional CPUs on cache hierarchies and processing core inter-connection.

To achieve high performance on multiple platforms, programmers have to customize their code (to be platform-specific), and, in the worse case, generate one variant per target platform. This process becomes unmanageable by hand when the kernels are complicated or/and multiple optimizations are required simultaneously. The previous chapters have proven that such code specialization can be solved in a systematic way on vectorization and local memory usage, by adapting standard and user-customized optimizations to platforms (semi-)automatically. Going a step further, we extend our approach into a generic programming framework called *Sesame*.

8.2. THE FRAMEWORK

Figure 8.2 shows the Sesame framework. Taking platform-agnostic code as input, Sesame performs code transformations and generates specialized kernels according the underlying platform(s). The Sesame framework consists of four components: (1) a feature identifier, (2) an impact predictor, (3) a transformer, and (4) an auto-tuner.

The *feature identifier* finds the architectural features that are sensitive to application performance, and generates corresponding optimization techniques into the *optimiza-*

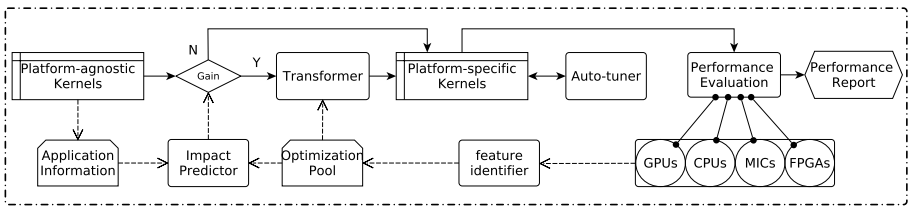


Figure 8.2: Sesame: a framework for many-core processors.

tion pool. In previous chapters, we regard local memory and vector cores as two key features. Accordingly, whether or not to use local memory/vectorization is regarded as an optimization in this optimization pool. The *impact predictor* analyzes the input kernel and checks which optimization (from the optimization pool) is efficient. The predictor can provide essential information on whether we need to enable or disable an optimization. Once we know that it is beneficial to enable/disable an optimization, a *transformer* applies the optimization by generating or adapting existing code to a parameterized version. The number and types of these parameters depends on the optimization and the existing code – for example, local memory usage can be parameterized with an ON/OFF switch (that is, enable or disable its usage), while vectorization might require a vector length. The parameterized code makes up the Sesame optimization space for a given application. Using an *auto-tuner*, we can find (near-)optimal solutions within this optimization space for a given target many-core processor.

When implementing Sesame, we continue to use OpenCL. Up to now, we have implemented three modules in Sesame: S2S Vectorizer, ELMO, and Grover. Our Sesame framework is scalable and it can integrate more modules when new architectural features and optimizations are identified from emerging platforms.

Different from prior work (see Section 8.5), Sesame brings four new ideas:

- It allows domain experts to use an existing, standardized programming model (namely, OpenCL), also allowing existing (legacy) code to be easily processed.
- It estimates the performance impact of key architectural features on given kernels and platforms.
- It transforms platform-agnostic kernels into platform-specific kernels.
- It obtains the right mix of optimizations for a given kernel by auto-tuning.

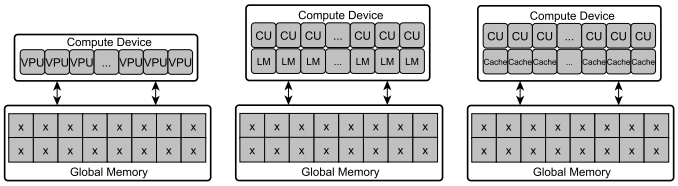
8.3. SESAME INPUTS

In this section, we describe the Sesame inputs: input kernels and platform models. Implicitly, these are the factors that define the applicability of Sesame.

8.3.1. INPUT KERNELS

From the users' perspective, a unified programming model ensures functional portability. With this unified model, users (domain experts) implement their applications and then try some "optimizations" on their kernels. Our Sesame framework will transform the kernels into platform-specific kernels.

The goal of Sesame is to add as many optimizations as possible, from a predefined, as complete as possible set (an optimization pool) – for example, vectorization, usage of local memory, memory access patterns, and granularity increase/decrease. The *impact predictor* checks whether an optimization (new one or existing one) would pay off. If it can make a decision, it will say YES or NO: for new optimizations, it will or will not implement it, and for existing optimizations, it will keep or remove them.



(a) MM-II: w/ VPU (b) MM-II: w/ LM (c) MM-II: w/ Cache

Figure 8.3: Sesame models.

8.3.2. PLATFORM MODELS

There are two platform models relating to Sesame: a front-end model for users, and multiple back-end models with specific architectural features for Sesame. For example, when users select OpenCL as their programming model, the OpenCL model will be the front-end model. On the other hand, a Sesame model relates to an architecture feature and an optimization from the optimization pool. Due to the diversity of many-core architectures, we give a set of Sesame models with the architectural properties that are significant to the overall performance. Examples of such key features are VPUs (vector processing units), on-chip programmer-managed local memory, and user-oblivious caches, and are shown in Figure 8.3. These back-end models are exposed to Sesame developers, but hidden to users.

We identify these features from studying the state-of-the-art many-core processors. For example, Intel MIC uses wider vectors, and thus VPU plays a key role in the overall performance. These key features can be either read from specifications, or extracted from *(micro-)benchmarking* (see Appendix A). Each target platform may have multiple performance-relevant features, and Sesame should be extended to perform the corresponding code transformations based on these key features.

8

8.4. SESAME IMPLEMENTATION

Up to now, we have implemented multiple modules in Sesame. The current status is summarized as follows.

8.4.1. VECTORIZATION

Currently, there are multiple many-core architectures that use SIMD cores. For these architectures, vectorization is a mandatory optimization: disabling it leads to poor resource utilization. In Chapter 4, we evaluate the performance impacts of explicitly using vector data types. We found that using vector types is still required on vector-core processors in the absence of an implicit vectorizer. Therefore, we propose a source-to-source translator that starts from a generic (scalar) kernel and applies step-by-step transformations to obtain a vectorized one. Currently, the translation is systematic, but performed by hand. We plan to implement it in a compiling pass and make it an automatic vectorization tool.

8.4.2. LOCAL MEMORY USAGE

The usage of local memory is another optimization already implemented in the Sesame framework. Specifically, we provide an impact predictor, an API to enable local memory usage, and a compiling pass to disable local memory usage.

AN IMPACT PREDICTOR

The ultimate solution for local memory usage is to use automated code transformation, typically in the form of a compiler pass. This code translation consists of two steps: (1) predict the performance benefits of using local memory, and then (2) perform code translation. In Chapter 5, we have focused on addressing the issue of unpredictability of performance benefits from using local memory. Thus, we developed a microbenchmark-based approach to quantify the performance impacts of using local memory. We designed and evaluated the benchmarks on typically used platforms, and store the results into a performance database. Based on this database, we have developed a query-based impact predictor (*Aristotle*) to indicate the performance gain/loss of using local memory.

ELMO

For the second step, i.e., code transformation, we have designed and implemented a high-level API targeting the efficient usage of local memory on modern many-core processors (Chapter 6). Specifically, we propose ELMO, a collection of easy-to-use APIs that (1) present a friendly front-end to make the bindings/mappings transparent to users, and (2) provide implementations and perform several optimizations to ensure the efficiency of the local memory operations. By using ELMO, users provide the impact predictor and the auto-tuner with already parameterized code, much easier to analyze and further optimize or remove the usage of local memory in a given input kernel.

GROVER

When using local memory is detected to have a negative performance impact, we need to disable local memory usage. In Chapter 7, we provide a compiling pass to automatically reverse-engineer the use of local memory. From the cases that the impact predictor (*Aristotle*) is able to analyze, Grover is able to remove 36%; all these cases are the ones when the performance of the new code (without local memory usage) on the new target platform is improved by Sesame.

8.5. RELATED WORK

The clash between productivity/portability and performance is not new in the multi-/many-core world. In fact, multiple approaches have been proposed to improve productivity while achieving high performance for many-core processors, which can be loosely classified into (i) new languages (e.g., OptiML [24]), (ii) auto-parallelizing compilers (e.g., OpenACC [120]), and (iii) libraries/APIs (e.g., Thrust [2]). In all these cases, programmers are isolated, in one way or another, from the difficult implementation details related to the platform architecture: they can focus on the functional parts of the application and leave these non-functional elements to be solved by run-times, compilers, or libraries.

Several FP7 projects have been focusing on this programming challenge. PEPPHER [1] is a 3-year European FP7 project (2010-2012) that aims to provide a unified framework for programming and optimizing applications for heterogeneous many-core processors to enable performance portability. In particular, PEPPHER provides a composition tool that adapts applications written in PEPPHER component model to the runtime system [31]. At the low level, PEPPHER uses StarPU which is a task programming library for hybrid architectures [9]. With StarPU, programmers can concentrate on algorithmic concerns, rather than handling low-level issues. The ENCORE project [2] is another FP7 project that addresses such a challenge. In particular, ENCORE uses a programming model, called *OmpSs* [37], for multi-cores and many-cores for increased portability and scalability, while preserving high performance for real-world applications. With ENCORE, we can significantly reduce the number of lines of code required to adapt an application for multi-cores, and thus need less development time.

8.6. SUMMARY

In this chapter, we introduced our vision for a generic framework to extend our work, aiming to perform architecture-specific optimizations (beyond vectorization and using local memory) and ultimately improve performance portability. Our observation is that performance portability is hindered by platform-specific optimizations, which can be either implicit or explicit, and difficult for end-users to generalize. Therefore, our Sesame framework aims to support any kernels and apply or remove a set of parameterized optimizations, when suitable. By auto-tuning these parameterized kernels, we automatically obtain platform-specific kernels that perform well across platforms (see Appendix B).

We have implemented three modules in the framework: S2S Vectorizer, ELMO, and Grover, which have proven the feasibility of our approach. Once a new architectural feature and its optimization is identified (by *feature identifier*), a new module (including an *impact predictor*, a *transformer*, and an *auto-tuner*) is considered to be extended into the framework.

[1] http://www.peppher.eu/
[2] http://www.encore-project.eu/

9

CONCLUSIONS AND FUTURE WORK

Multi-cores and many-cores have become pervasive in computing machines. At the same time, their architectural diversity poses a challenge for users to efficiently exploit their hardware potential. As more and more applications demand acceleration, an increasing number of programming experts are needed to sustain this development. Their task is to implement and optimize applications for given platforms; implicitly, they are expected to find the best match of the architecture with the software, which in turn requires significant manual labor in application design, implementation, and tuning. Leveraging a unified programming model enables functional portability, but, because of the diversity, it cannot guarantee high performance across platforms. Therefore, programming tools and models that also deal with optimizations and tuning as uniformly as possible are required.

This thesis has given evidence that this problem can be addressed successfully: we have investigated the enabling/disabling techniques for platform-specific optimizations with a unified programming model. We have selected OpenCL as our research vehicle, and identified that each platform has a specific optimization space for a given kernel. Taking two concrete examples, we have proposed solutions on how to hide the architectural disparities with a unified programming model.

In this chapter, we present the main conclusions of our work, discuss answers to our main research questions. Further, we sketch future directions of research that can continue and improve our work.

9.1. CONCLUSIONS

Our research has led to the following major conclusions:

1. **Unified programming models - OpenCL being a good example - can achieve similar performance with native ones (RQ1).** The diversity of multi-/many-core processors and programming models requires a unified programming model to save development cost. As an instance of unified programming standards, OpenCL stands out for its cross-platform ability. Our experimental results have shown

that OpenCL can largely guarantee code portability on various computing devices. With regard to performance, we have shown that OpenCL can achieve comparable performance with CUDA (the native programming model on NVIDIA GPUs). For synthetic benchmarks, OpenCL and CUDA have similar performance. However, we have observed some performance gaps between OpenCL and CUDA when using real-world applications. We have found that such performance gaps can be reduced by systematically changing code from programming models, kernel optimizations, and compilers.

2. **The optimization space for each application can be explored systematically, but the results are platform- and application-specific (RQ2).** To achieve high performance, we need to apply optimization techniques on a given kernel. We have shown that such optimization techniques can significantly improve the overall performance on a computer vision case study (i.e., stereo matching). In addition, we have shown that the recommended optimization techniques are not transferable: they can be implemented in the unified programming model, but their impact on performance can vary widely. This variation is a consequence of *both* the platforms having different architectural features, *and* the kernels having different patterns of computation and memory accesses.

3. **Providing vector types in a unified programming model such as OpenCL can lead to unexpected performance penalties (RQ3).** By using a low-level programming model such OpenCL, programmers can already specify parallelism explicitly. We have shown that further vectorizing OpenCL kernels (i.e., using vector data types) often leads to a performance increase on vector-core processors (e.g., CPUs), but shows a performance drop on scalar-core processors (e.g., GPUs). This is a case where performance portability is compromised by construction. Assuming compilers can autovectorize scalar, fine-grained parallel code for vector-core architectures, the presence of vector types in a unified model such as OpenCL is concluded to be redundant.

4. **Despite common belief, the benefits of enabling the use of local memory are not as predictable as expected (RQ4a).** We have found that the performance impact of local memory usage is very unpredictable. This is because (1) applications differ in memory access patterns, and (2) platforms differ in memory hierarchies. Therefore, we have proposed a query-based approach to determine whether this impact is positive or negative.

5. **A specialized high-level API can be added to a unified programming model to enable the usage of specific features in a portable manner. Specifically, we show how the usage of local memory benefits from such a specialized API (RQ4b).** Enabling the usage of local memory requires non-trivial transformations of the code. Specifically, users have to deal with challenges related to loading and storing local memory data. We have found that these challenges can be easily avoided by providing a high-level API that keeps the bindings between work-items and data elements transparent to users. With the help of the API, productivity can be improved. In addition, such an API improves the performance portability of the ap-

plication, due to the platform-tuned back-end. The main challenge in providing such an API is to insure its completeness.

6. **To enable users to explore the full optimization space of an application, techniques that are not suitable for given platforms should be automatically reversed. Specifically, the usage of local memory can be disabled at compile time (RQ4c).** Using local memory can lead to significant performance penalties for several devices. Automatically removing the usage of local memory is therefore desirable in such cases. We have found that for typical cases of local memory usage, a compiler pass can reverse this optimization. Such a tool makes the process of disabling local memory usage transparent to users, and completely free them from such non-functional operations.

9.2. FUTURE RESEARCH DIRECTIONS

Our work so far has proved that several application optimizations can be automatically enabled or reversed with little effort from the user. However, in order to generalize this approach to a user-friendly exploration of the optimization space of an application, there are still many gaps to fill. We list several promising research directions towards this goal.

1. **Auto-tuning architecture-specific parameters.** This thesis proves that we can (semi-)automatically enable/disable optimizations. Further research can be done to auto-tune these optimizations (i.e., selecting a right tuning value). For example, what is the proper local memory size for a given application, a platform, and an input dataset. This tuning approach can either be empirical or model-based or both.

2. **Auto-vectorizer for explicitly parallelized kernels.** Low-level programming models can already specify parallelism in kernels. To hide the architectural differences (scalar cores vs. vector cores) and achieve portable performance, an implicit vectorizer is desirable. We have shown that the existing implicit vectorizer such as the one in Intel compiler is performing well on vector-core architectures. Likewise, an auto-SIMD module is desired for other vector-core processors. In this way, users do not have to manually vectorize applications any more.

3. **Automated abstraction of memory access patterns.** Our performance database starts with memory access patterns. However, the process of abstracting memory access patterns relies on users, i.e., they have to manually extract patterns from the given application. This becomes difficult especially when the target kernels are complicated. Therefore, an automated (on-line) tool is desirable. The approaches based on polyhedral model might be taken as a starting point [108].

4. **Extending ELMO.** ELMO's front-end still needs extensions to cover more access patterns from real-world applications. Its back-end should be improved to support even more (classes of) platforms, which requires additional research into the performance tuning of these platforms. As a further step, it is interesting to put

each API of ELMO into a compiling pass that can automatically enable local memory usage.

5. **Extending Grover.** Further investigation can be done on Grover's impact on other types of devices (e.g., GPUs). Using Grover, it is also interesting to model the performance benefits/losses due to local memory usage on CPUs. Furthermore, incorporating Grover into a high-level auto-tuning framework for OpenCL kernels could allow code specialization to be autotuned for different classes of platforms.

6. **Designing new architectures with configurable memories.** Our performance database indicates that SPMs perform better on some applications, while caches perform better on others. Future architecture designs should keep both memories, and allow (re-)configuration of the hierarchy according to the application needs. This requires more research into application patterns and classes, which should be used to tune architectural design towards more flexible memory configurations.

7. **Identifying more performance-changing architectural features.** We have considered vectorization and local memory usage as concrete case studies. Investigating the simplification, auto-tuning, and cancellation of other high-impact performance-changing optimization techniques offers more research opportunities. As an example, we could investigate how the non-uniform caches impact the performance on Intel's MIC. Note that the architectural features are not limited to a single feature like local memory, but can be a combination of several features.

8. **Defining the optimal ordering of the optimizations.** A challenging research direction is the ordering of a given set of optimizations for a given application. Ideally, this should be based on a platform-specific model, but the complex interconnections between some optimization techniques (e.g., enabling the usage of local memory can constrain automated vectorization) might prevent such an elegant solution. Instead, empirical and learning-based approaches can be used.

9. **A complete implementation of the Sesame framework.** Once all the issues have been addressed, further efforts are required to incorporate them into the Sesame framework, which then would allow to systematically address platform-specific optimizations and ultimately achieve portable performance.

A

TEST-DRIVING INTEL XEON PHI

In this appendix, we perform an empirical study on Intel Xeon Phi at two levels: the micro-benchmark level, and the real-world application level. At the microbenchmarking level, we show the high performance of five components of the architecture, focusing on their maximum achieved performance and the prerequisites to achieve it. At the application level, we show our porting experience on a medical imaging application. This appendix shows how to obtain key architectural features, required by our *Sesame* framework (Chapter 8), for a given processor.

Intel Xeon Phi (Phi) is the newest high-throughput architecture targeted at high performance computing (HPC). Without a doubt, Phi will be part of the very next generation of supercomputers that will challenge TOP500[1]. To achieve its theoretical high performance (around 1 TFlop), Intel Xeon Phi [68] uses around 60 cores and 30 MB of on-chip caches, and relies on traditional many-core features like vector units or SIMD/SIMT, high throughput, and high bandwidth [89]. It adds to that some "unconventional" features, such as the overall L2 cache coherency and the ring interconnect, all for the sake of performance and usability.

By taking Phi as a black-box with over 200 hardware threads, we ran Leukocyte Tracking (a medical imaging application [130]) on it. We found that (1) the sequential application (a single thread) on Phi runs about 5× slower than the same sequential execution on a "traditional" multi-core processor, and (2) that the Phi version scales only up to 40 threads (Figure A.13, more details in Section A.4). To explain this (observed) performance behavior, as well as to eventually improve it, we require a deeper understanding of the architecture and its parameters.

Moreover, previous experiences with massively parallel high performance platforms such as NVIDIA GPUs or the Cell/B.E. showed that a trade-off between performance and

This appendix is based on our work published in the Proceedings of ICPE 2014 [43].

[1] In June 2013, two Xeon Phi supercomputers - TIANHE-2 from NUDT and STAMPEDE from TACC - were ranked first and sixth in TOP500: http://www.top500.org.

ease-of-use is necessary: "simple" programming often leads to disappointing performance [137, 159]. Therefore, given Phi's promise of breaking this pattern, this work focuses on a test drive of the platform: we have conducted a two-stage empirical study of the Xeon Phi, stressing its high-performance features both in isolation (aiming to quantify their maximum achievable performance), and in the real-life case-study (aiming to understand its regular performance).

To this end, we have implemented and used dedicated microbenchmarks - gathered in a suite called *MIC-Meter* [2] - to measure the performance of four key architectural features of Xeon Phi: the processing cores, the memory hierarchies, the ring interconnect, and the PCIe connection. Following these experiments "in isolation", we propose a conceptual model of the processor that facilitates the performance analysis and optimization of the real-life case-study.

Such a thorough evaluation can benefit two different classes of Phi users: the *experts*, who are interested in in-depth architectural knowledge, and the *production users*, interested in simple and effective ways to use processors. For expert users - like most high performance computing (HPC) programmers and compiler developers are - knowing the requirements for density and placement of threads per cores, the optimal utilization of the core interconnections, or the difference in latency between the different types of memories on chip are non-trivial details that, when properly exploited, can lead to significant performance gains. For production users, a simplified view of the Xeon Phi machine is mandatory to help exploring different parallelism strategies. Such a model is simplified view of the machine, including the most important functionality and performance constraints.

The main contributions of our work are as follows:

- We present our hands-on experience achieved while microbenchmarking the Xeon Phi (Section A.2). This experience also leads to interesting numerical results for the capabilities of Phi's cores, memories, interconnects (i.e., the ring and the PCIe).
- We synthesize four essential platform-centric performance guidelines, aimed at easing the development and tuning of applications for the Xeon Phi (Section A.3).
- We propose a conceptual model of Phi (*SCAT*), which strips off the performance irrelevant architectural details, presenting the programmers with a simple, functionality-based view of the machine (Section A.3).
- Using a case study (leukocyte tracking), we analyze the application and optimize it, discussing the lessons to be learned from this experience (Section A.4).

The remainder of this appendix is organized as follows: Section A.1 presents the background and our approach to benchmark the Xeon Phi. In Section A.2, we benchmark the Xeon Phi from different perspectives. In Section A.3, we summarize our observations and show a machine model for the Xeon Phi. In Section A.4, we show our hands-on experience on a case study in medical imaging. We present the related work in Section A.5 and summarize this appendix in Section A.6.

[2] https://github.com/haibo031031/mic-meter

A.1. BENCHMARKING INTEL XEON PHI

In this section, we introduce Intel Xeon Phi - with its novel features and typical programming models, and we present our benchmarking methodology.

A.1.1. THE ARCHITECTURE

Intel Xeon Phi has over 50 cores (the version used in this work belongs to the 5100 series and has 60 cores) connected by a high-performance on-die bidirectional interconnect (shown in Figure A.1). In addition to these cores, there are 16 memory channels (supported by memory controllers) delivering up to 5.0 GT/s [66]. When working as an accelerator, Phi can be connected to a host (i.e., a device that manages it) through a PCI Express (PCIe) system interface - similar to GPU-like accelerators. Different from GPUs, a dedicated embedded Linux μOS (version: `2.6.38.8`) runs on the platform.

Each core contains a 512-bit wide vector unit (VPU) with vector register files (32 registers per thread context). Each core has a 32KB L1 data cache, a 32KB L1 instruction cache, and a core-private 512KB unified L2 cache. In total, a 60-core machine has a total of 30MB of L2 cache on the die. The L2 caches are kept fully coherent by the hardware, using DTDs (distributed tag directories), which are referenced after an L2 cache miss. Note that the tag directory is not centralized, but split up into 64 DTDs, each getting an equal portion of the address space and being responsible for maintaining it globally coherent. Another special feature of Xeon Phi is the fast bidirectional ring interconnect. All connected entities use the ring for communication purposes, using special controllers called *ring stops* to insert requests and receive responses on the ring.

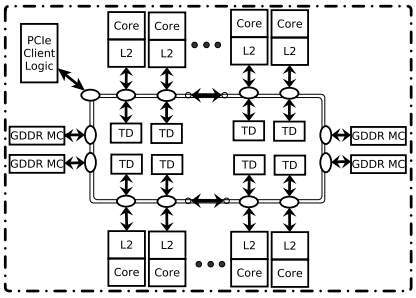


Figure A.1: The Intel Xeon Phi Architecture.

The novelties of the Xeon Phi architecture relate to five components : (1) the vector processing cores, (2) the on-chip memory, (3) the off-chip memory, (4) the ring interconnect, and (5) the PCIe connection. As these are the features that differ, in one way or another, from a typical CPU - vectors are wider, there are many more cores, cache coherency and shared memory are provided with low penalty for 60 or more cores, and a ring interconnect holds tens of agents that can interchange messages/packets concurrently -, we focus our benchmarking efforts on these features.

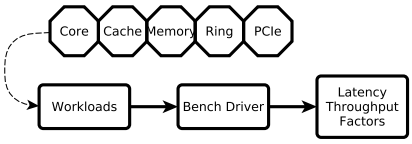


Figure A.2: The MIC-Meter Overview.

A.1.2. PROGRAMMING

In terms of usability, there are two ways an application can use Intel Xeon Phi: (1) in *offload mode* - the main application is running on the host, and it only offloads selected (highly parallel, computationally intensive) work to the coprocessor, or (2) in *native mode* - the application runs independently, on the Xeon Phi only, and can communicate with the main processor or other coprocessors [67] through the system bus. In this work, we benchmark Xeon Phi in both modes.

Finally, to program applications on Xeon Phi, users need to capture both functionality and parallelism. Being an `x86` SMP-on-a-chip architecture, Xeon Phi offers the full capability to use the same tools, programming languages, and programming models as a regular Intel Xeon processor. Specifically, tools like Pthreads [34], OpenMP [18], Intel Cilk Plus [17], and OpenCL [147] are readily available. Given the large number of cores on the platform, a dedicated MPI version is also available. In this work, all the experiments we present are programmed using C/intrinsics/assembly with OpenMP/Pthreads; we also use Intel's `icc` compiler (`V13.1.1.163`).

A.1.3. MIC-METER

We show our MIC-Meter in Figure A.2. The goal of our benchmarking is two-fold: to show how the special capabilities of Xeon Phi can and should be measured, to quantify the performance of this novel many-core architecture, and eventually to identify the impacting factors. To this end, we choose a microbenchmarking approach: we measure each capability in isolation, under variable loads, and we quantify its performance in terms of both latency-oriented and throughput-oriented metrics.

Simply put, *latency* is the time required to perform an operation and produce a result. As latency measurement focuses on a single action from its beginning to its end, one needs to isolate the operation to be measured and use a highly accurate, non-intrusive timing method. Alternatively, we can measure a long enough sequence of operations with an accurate timer, and estimate latency per operation by dividing the measured time by the number of operations. In this work, latency measurements are done with a single thread (for individual operations) or two threads (for transfer operations) with Pthreads. All latency benchmarks are written in C (with inline assembly).

Throughput is the number of (a type of) operations executed in a given unit of time. As higher throughput means better performance, microbenchmarking focuses on measuring the *maximum achievable throughput* for different operations, under different loads; typically, the benchmarked throughput values are slightly lower than the theoretical ones. Thus, to measure maximum throughput, the main challenge is to build the workload such that the resource that is being evaluated is fully utilized. For example,

when measuring computational throughput, enough threads should be used to fully utilize the cores, while when measuring memory bandwidth, the workload needs to have sufficient threads to generate enough memory requests. For all the throughput measurements in this paper, our multi-threaded workloads are written in C and OpenMP.

We note that the similarities between Phi and a regular multi-core CPU allow us to adapt existing CPU benchmarks to the requirements of Xeon Phi. In most cases, we use such "refurbished" solutions, that prove to serve our purposes.

A.2. EMPIRICAL EVALUATION

In the following sections, we present in detail the MIC-Meter and the results for each of the components: (1) the vector processing cores, (2) the on-chip and off-chip memory, (3) the ring interconnect, and (4) the PCIe connection.

A.2.1. VECTOR PROCESSING CORES

We evaluate the vector processing cores in terms of both instruction latency and throughput. For latency, we use a method similar to those proposed by Agner Fog [52] and Torbjorn Granlund [55]: we measure instruction latency by running a (long enough) sequence of dependent instructions (i.e., a list of instructions that, being dependent on each other, are forced to be executed sequentially - *an instruction stream*).

The same papers propose a similar approach to measure throughput in terms of *instructions per cycle (IPC)*. However, we argue that a measurement that uses all processing cores together, and not in isolation, is more realistic for programmers. Thus, we develop a `flops` microbenchmark to explore the factors for reaching the theoretical maximum throughput on Xeon Phi (Section A.2.1).

VECTOR INSTRUCTION LATENCY

Xeon Phi introduces 177 vector instructions [65]. We roughly divide these instructions into five classes [3]: mask instructions, arithmetic (logic) instructions, conversion instructions, permutation instructions, and extended mathematical instructions.

The benchmark for measuring the latency of vector instructions is measuring the execution time of a sequence of 100 vector operations using the same format: $zmm1 = op(zmm1, zmm2)$, where $zmm1$ and $zmm2$ represent two vectors and op is the instruction being measured. By making $zmm1$ be both a source operand and the destination operand, we ensure the instruction dependency - i.e., the current operation will depend on the result of the previous one.

For special classes of instructions - such as the `conversion` instructions `vcvtps2pd` and `vcvtpd2ps` - we have to measure the latency of the conversion pair ($zmm2 = op12(zmm1)$; $zmm1 = op21(zmm2)$) in order to guarantee the dependency between contiguous instructions (i.e., it is not possible to write the result of the conversion in the same source operand, due to type incompatibility). Similarly, we measure the latency of extended mathematical instructions such as `vexp223ps` and `vlog2ps` in pairs, to avoid overflow (e.g., when using 100 successive `exp()`'s).

[3] Note that we choose not to measure the latency of memory access instructions because the latency results are highly dependent on the data location(s).

The interesting results for vector instruction latency are presented in Table A.1. With these latency numbers, we know how many threads or instruction streams we need to hide the latency on one processing core.

Table A.1: The vector instruction latency (in cycles).

Instruction	Category	Latency
kand, kor, knot, kxor	mask instructions	2
vaddpd, vfmadd213pd, vmulpd, vsubpd	arithmetic instructions	4
vcvtdq2pd, vcvtfxpntdq2ps, vcvtfxpntps2dq, vcvtps2pd	convert instructions	5
vpermd, vpermf32x4	permutation instructions	6
vexp223ps, vlog2ps, vrcp23ps, vrsqrt23ps	extended mathematical instructions	6

VECTOR INSTRUCTION THROUGHPUT

The Xeon Phi 5100 has 60 cores working at 1.05 GHz, and each core can process 8 double-precision data elements at a time, with maximum 2 operations (`multiply-add` or `mad`) per cycle in each lane (i.e., a vector element). Therefore, the theoretical instruction throughput is 1008 GFlops (approximately 1 TFlop). But **is this 1 TFlop performance actually achievable?** To measure the instruction throughput, we run 1, 2, 4 threads on a core (60, 120, and 240 threads in total). During measurement, each thread performs one or two instruction streams for a fixed number of iterations: $b_{i+1} = b_i \; op \; a$, where i represents the iteration, a is a constant, and b serves as an operand and the destination. The loop was fully unrolled to avoid branch overheads. The microbenchmark is vectorized using explicit intrinsics, to ensure a 100% vector usage.

The results are shown in Figure A.3. We note that the peak instruction throughput - i.e., one vector instruction per cycle (1TFlops in total) - can be achieved when using 240 threads and the multiply-add instruction. As expected, the `mad` throughput is twice larger than the `mul` throughput. Further, two more observations can be added. First, when using 60 threads (one thread per core), the instruction throughput is low compared with the cases when using 120 or 240 threads. This is due to the fact that it is not possible to issue instructions from the same thread context in back-to-back cycles [66]. Thus, programmers need to run at least two threads on each core to be able to fully utilize the hardware resources. Second, when a thread is using only one instruction stream, we have to use 4 threads per core (240 threads in total) to achieve the peak instruction throughput. This is because the latency of an arithmetic instruction is 4 cycles (Table A.1), and we need no less than four threads to totally hide this latency (i.e., fill the pipeline bubbles [62]). To comply, programmers need to either use 4 threads per core or have more independent instruction streams.

To summarize, for a given instruction mix (`mul` or `mad`), the achievable instruction throughput depends not only on the number of cores and threads, but also on the issue width (i.e., the number of independent instruction streams). We also benchmarked the

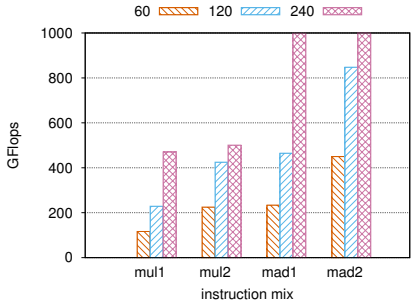


Figure A.3: Arithmetic throughput using different numbers of threads (60, 120, 240), different instruction mixes (`mul` versus `mad`), and issue widths (using one and two independent instruction streams).

EMU (extended math unit) and see [50] for more details.

A.2.2. MEMORY LATENCY

Available benchmarks, such as `BenchIT` [150] and `lmbench` [96] use *pointer-chasing* to measure the on-chip and off-chip memory access latency. This approach has the advantage of not only determining the latency itself, but also exposing the differences between consecutive layers of a memory hierarchy (i.e., different layers of caches and main memory will have significantly different latencies). Thus, we use a similar approach to measure the latency for an Xeon Phi core (i.e., the latency for accessing local caches and main memory - see Section A.2.2).

When more than two cores communicate, measuring latency is complicated. For this, Daniel Molka et al. proposed an approach to quantify cache-coherency effects [104]. In our work, we adapt this approach to Xeon Phi using the correct memory fences and cache flushing instructions [4].

ACCESS LATENCY ON A SINGLE CORE

To reveal the local access latency, we use a *pointer-chasing* benchmark similar to those used by `BenchIT` and `lmbench`. Essentially, the application traverses an array A of size S by running $k = A[k]$ in a fully unrolled loop. The array is initialized with a $stride$, i.e., $A[k] = (k + stride)\%S$. By measuring the execution time of the traversal, we can easily obtain an estimate of the average execution time for one iteration. This time is dominated by the latency of the memory access. The traversal is done in one thread and utilizes only one core. Therefore, the memory properties obtained here are local and belong to one core.

The results are shown in Figure A.4. We see that the Xeon Phi has two levels of data caches (L1 and L2). The L1 data cache is 32KB, while the L2 data caches should be smaller than 512KB. Furthermore, the accessing latency of L1 and L2 data caches is around 2.87 ns (3 cycles) and 22.98 ns (24 cycles), respectively. With a stride of 64 bytes, Xeon Phi takes 287.51 ~ 291.18 ns (302 ~ 306 cycles) to finish a data access in the main memory

[4] Since Xeon Phi has no `mfence` or `clflush`, we need to change the benchmark by searching and replacing them with equivalent instructions.

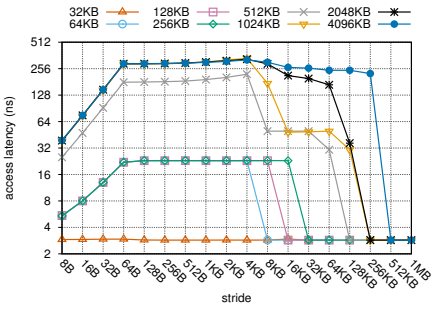


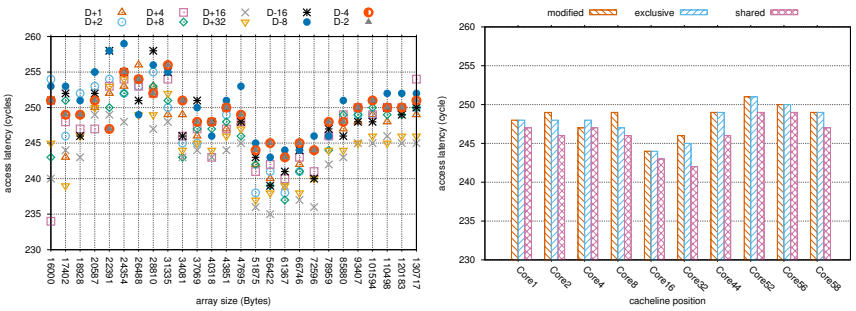
Figure A.4: Average memory latency when changing strides and datasets. The x-axis is logarithmic and it represents the pointer chasing stride.

(when the dataset is larger than 512KB). We note that when traversing the array in a larger stride (e.g., 4KB), the latency of accessing data in off-chip memory is slightly larger. This is because the contiguous memory accesses fall into different pages. Furthermore, we can observe (from the upper trend) that threads operate the data in a batch manner, i.e., a 64-byte cache-line. Information about cache associativity can also be seen in Figure A.4 (see [144] for the calculation approach).

REMOTE CACHE LATENCY

We have illustrated our measurements and results for memory latency on a single core in Section A.2.2. In this section, we focus on measuring remote cache latency. For these measurements, we use an approach based on that proposed for a traditional multi-core processor by Daniel Molka [104]. Our setup is built as follows: prior to the measurement, the to-be-transferred cache-lines are placed in different locations (cores) and in a certain coherency state (`modified`, `exclusive`, or `shared`). In each measurement, we use two threads (T0, T1), with T0 pinned to Core 0 and T1 pinned on another core (Core *X*). The latency measurement always runs on Core 0, transferring a predefined number of cache lines from Core *X* to Core 0.

Figure A.5 shows our results for remote cache accesses latency on Xeon Phi. In Figure A.5a, we see that when the cache line is in `modified` state, the overall latency of remote access averages around 250 cycles, which is much larger than the local cache access latency (by an order of magnitude) but still smaller than the off-chip memory access latency (by 17%). By getting the *median* value of all the input data sets (up to 128 KB), we get the overall remote latency shown in Figure A.5b. We note no relationship between the remote access latency and the cache-line states, except that accessing remote `shared` cachelines takes a few less cycles. This is because in whichever state a cacheline is, when a core accesses it, a transfer is needed from a remote core (different from a traditional multi-core CPU with cores sharing the last-level cache). Furthermore, Xeon Phi adopts the `MOESI` cache coherence protocol [66] to share a cacheline before writing it back, and thus Figure A.5b shows no penalty of writing data back. In [50], our experiments have shown that there is a relation between the latency and the core distances on an older version of the Xeon Phi (namely, 31S1P), but this effect seems to have disappeared on the newer Xeon Phi 5110.



(a) Modified (b) Overall

Figure A.5: Read latencies of Core 0 accessing the cache lines on Core 1 (D+1), Core 2 (D+2), Core 4 (D+4), Core 8 (D+8), Core 16 (D+16), Core 32 (D+32), Core 44 (D-16), Core 52 (D-8), Core 56 (D-4),and Core 58 (D-2).

A.2.3. MEMORY BANDWIDTH

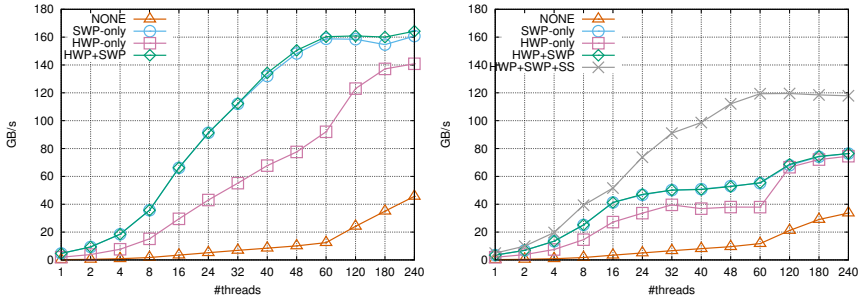
McCalpin's stream benchmark [75] includes a memory bandwidth benchmark and presents results for a large number of high-end systems. However, his solution is based on a combination of both read and write operations. In this work, we want to separate *reads* and *writes* so as to quantify the impacting factors. In `BenchIT`, Daniel Molka et al. presents a solution to measure bandwidth in a similar way with that of latency measurement (see Section A.2.2). His microbenchmark requires compiler optimizations to be disabled (i.e., the code should be compiled with the `-O0` option), thus disabling the software prefetching on Xeon Phi. As a result, this measurement will underestimate bandwidth. In this section, we present our own `OpenMP` implementation of a memory bandwidth microbenchmark, considering hardware/software prefetching, streaming stores, ECC effects and off-chip/on-chip differences.

OFF-CHIP MEMORY BANDWIDTH

The Xeon Phi used in this work has 16 memory channels, each 32-bits wide. At up to 5.0 GT/s transfer speed [5], it provides a theoretical bandwidth of 320 GB/s. But **is this theoretical bandwidth really achievable in real cases?** To answer this question, we use separate benchmarks to measure the memory bandwidth for both `read` and `write` operations. The `read` benchmark reads data from an array A ($b = b + A[k]$). The `write` benchmark writes a constant value into an array A ($A[k] = C$). Note that A needs to be large enough (e.g., 1 GB) such that it cannot fit in the on-chip memory. To avoid the impact of "cold" TLBs, we start with two "warm-up" iterations of the benchmarks, before we measure a third one. We use different numbers of running threads - from 1 to 240.

Our results are shown in Figure A.6 (HWP+SWP) (we plot the median value of ten runs of the benchmarks). Overall, we see that the maximum bandwidth for both `read` and `write` is far below the theoretical peak of 320 GB/s. Moreover, both the `read` and `write` memory bandwidth increases over the number of threads - which happens because when using more threads, we can generate more requests to memory controllers,

[5] GT/s stands for Giga Transfers per second.



(a) read (b) write

Figure A.6: Read and write memory bandwidth.

thus making the interconnect and memory channels busier. Thus, if aiming to achieve high memory bandwidth, programmers need to launch enough threads to saturate the interconnect and the memory channels. Figure A.6a shows that the `read` bandwidth peaks at 164 GB/s, achievable with using 60 threads or more (pinning at least one thread to a core). However, we can obtain the maximum `write` bandwidth (76 GB/s, as seen in Figure A.6b) only when using 240 threads. In general, the `write` bandwidth is around half of the `read` bandwidth. This happens because Xeon Phi implements a write-allocate cache policy and the original content has to be loaded into caches before we overwrite it completely. To avoid the memory bandwidth waste, programmer can use *streaming stores* [6] on Xeon Phi [66]. We see that using *streaming store* instructions speeds-up write operations up to 1.7 times (Figure A.6b:HWP+SWP+SS), with memory write bandwidth now peaking at 120 GB/s. Thus, programmers must consider using *streaming stores* to optimize the memory bandwidth.

Prefetch Effects: Xeon Phi supports both hardware prefetching (HWP) and software prefetching (SWP). The L2 cache has a streaming hardware prefetcher that can selectively prefetch code, read, and RFO (Read-For-Ownership) cachelines into the L2 cache [66]. Figure A.6 shows the memory bandwidth of four different configurations: no prefetching, HWP or SWP only, or both. When disabling both HWP and SWP, the memory bandwidth is low (45 GB/s for reading and 33 GB/s for writing). With only SWP, we already achieve similar memory bandwidth to that achieved when enabling both of them. This similarity indicates that the hardware prefetcher will not kick in when software prefetching performs well. Furthermore, enabling only HWP delivers about half of the bandwidth achieved when enabling only SWP (the bandwidth is roughly 1.9× smaller, on average).

To further evaluate the efficiency of prefetching on Xeon Phi, we use the Stanza Triad (`STriad`) [33] benchmark with a single thread. `STriad` works by performing a DAXPY (Triad) inner loop for a length L stanza, then jumps over k elements, and continues with the next L elements, until reaching the end of the array. We set the total array size to 128 MB, and set k to 2048 double-precision words. For each stanza, we ran the experiment 10 times, with the L2 cache flushed each time, and we calculate median value of the 10 runs

[6] Streaming stores do not require a prior cache line read for ownership (RFO) but write to memory "directly".

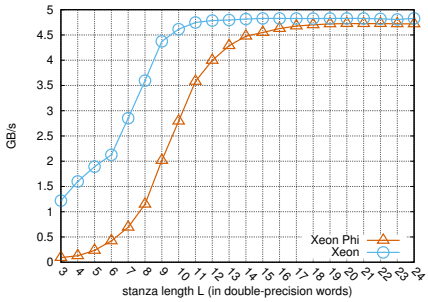


Figure A.7: Performance of STriad on the Xeon Phi (the x-axis is in log scale and the results on Xeon are normalized to those on Xeon Phi).

to get the memory bandwidth for each stanza length. Figure A.7 shows the results of the STriad experiments on both Xeon Phi and a regular Xeon processor (Intel Xeon E5-2620). We see an increase in memory bandwidth over stanza length L, and we note it eventually approaches a peak of 4.7 GB/s (note that this is achieved per core). Further, we see the transition point (from the bandwidth-increasing state to the bandwidth-stable state) appears earlier on Xeon than on Xeon Phi. Therefore, we conclude that non-contiguous access to memory is detrimental to memory bandwidth efficiency, with Xeon Phi showing more restrictions on the stanza length when prefetching data than the regular Xeons. To comply, programmers have to create the longest possible stanzas of contiguous memory accesses, improving prefetching and memory bandwidth.

ECC Effects: The Xeon Phi coprocessor supports ECC (Error Correction Code) to avoid software errors caused by naturally occurring radiation. Enabling ECC adds reliability, but it also introduces extra overhead to check for errors. We examined the bandwidth differences with and without disabling ECC. With ECC disabled, we noticed a 20% to 27% bandwidth increase [50]. Note that all the experiments in this work are performed with ECC enabled. Furthermore, the new μOS kernel on Phi adds support of the transparent huge pages (THP) functionality, which is enabled by default and often improves application performance without any code or environmental changes.

AGGREGATED ON-CHIP MEMORY BANDWIDTH

The available on-chip memory bandwidth is always essential in performance tuning and analysis. So, **how large is the on-chip memory bandwidth that can be achieved?** To answer this question, we measure the cache bandwidth on a single core [7] and calculate the *aggregated* cache bandwidth by multiplying it with the number of cores. We first use a set of `vmovapd` instructions to measure the native `read` or `write` bandwidth. Our results show that the L1 access (read or write) throughput is 64 bytes per cycle. Thus, the aggregated L1 bandwidth is 4032 GB/s for `read` or `write`. Then we measure the maximum achieved bandwidth from programmers' point of view for `scale1` ($O[i] = a \times A[i]$), `scale2` ($O[i] = a \times O[i]$), `saxpy1` ($O[i] = a \times A[i] + B[i]$), and `saxpy2` ($O[i] = a \times$

[7] Note that we choose not to measure the inter-core communication bandwidth because we assume that cache-line transfers occur rather scattered, and not in a large volume. Thus, the measurement of inter-core (remote) access latency is of greater use.

$A[i]+O[i]$) operations. To avoid overheads from the high-level code, we use intrinsics in the kernel code. We also disable the software prefetching due to the fact that the data is located in caches after warming up.

The results are shown in Figure A.8. We see that the maximum achieved bandwidth on a core is 73 GB/s, 96 GB/s, 52 GB/s, 69 GB/s for `scale1`, `scale2`, `saxpy1`, `saxpy2`, respectively. The bandwidth of `scale2` and `saxpy2` is 1.3× larger (than `scale1` and `saxpy1`, respectively) because the data cache allows a read/write cache-line replacement to happen in a single cycle [8]. The L1 bandwidth on a single core could be larger when further unrolling the loops or better scheduling instructions for each dataset. The aforementioned numbers are achieved by unrolling the loops 16 times without changing the assembly code.

Furthermore, it is difficult to exactly measure the L2 bandwidth due to the presence of the L1 cache. The bandwidth depends on the memory access patterns. Specifically, when we use a L2-friendly memory access pattern, the compiler will identify the stream pattern and prefetch data to the L1 cache in time. By this, we will get a much larger bandwidth due to the common efforts of L1 and L2. On the other hand, an unfriendly memory access will experience many L1 misses and result in cache thrashing. Our benchmarking results are obtained when disabling the software prefetching. When using 4 threads on a core, we notice a bandwidth of 11 GB/s, 20 GB/s, 10 GB/s, 16 GB/s for `scale1`, `scale2`, `saxpy1`, `saxpy2`, respectively (Figure A.8).

A.2.4. RING INTERCONNECT

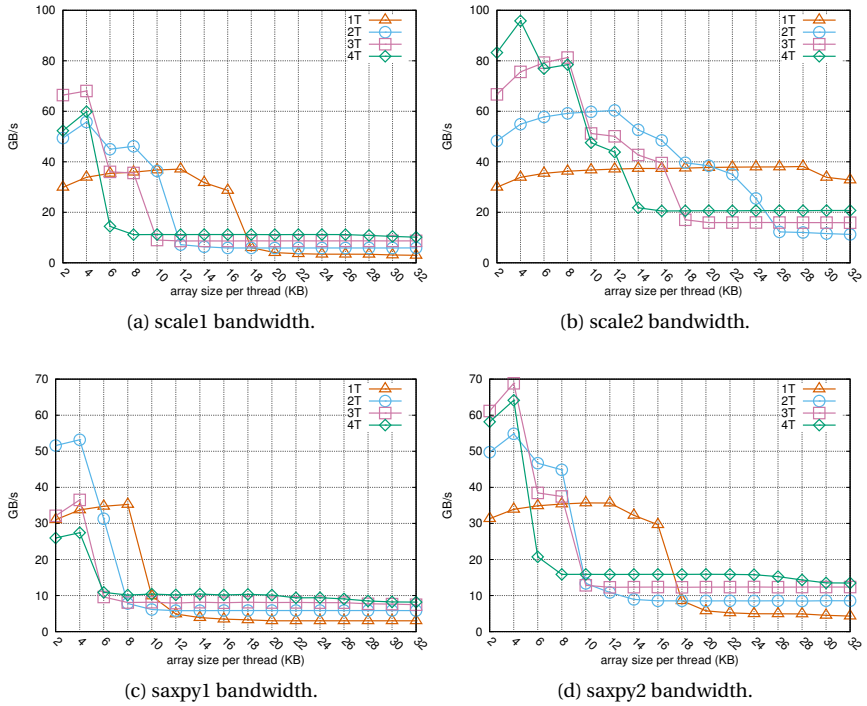
On Xeon Phi, the cores and memory controllers are interconnected in a bi-directional ring. When multiple threads are requesting data simultaneously, shared components like the ring stop or DTDs can become performance bottlenecks. In order to check this hypothesis, and its eventual performance impact, we use *thread affinity* to fix threads on cores, and we run the bandwidth microbenchmarks to quantify potential bandwidth changes (in GB/s) for different thread-to-core mapping scenarios.

CORE/THREAD DISTRIBUTION

First, we measure the `read` memory bandwidth by distributing threads onto separate cores in three different patterns: (1) *compact* - the cores are located close to each other, (2) *scattered* - the cores are evenly distributed around the ring, and (3) *random* - the core `IDs` are selected randomly with no repeats. The bandwidths are measured using 2, 4, 8, and 16 cores and the results are presented in Figure A.9a. We see that the three approaches achieve very similar memory bandwidths. Thus, the cores around the ring are *symmetric* on Xeon Phi, and the distance between has practically no impact on the achieved bandwidth.

Second, as each Xeon Phi core supports up to four hardware threads, we investigate whether there is any impact on bandwidth if the threads are all gathered on the same core (thus, less interconnect traffic) or distributed among different cores. Figure A.9b shows that when the threads run on the same core, the bandwidth stabilizes at 4.7 GB/s. We also note that running threads on separate cores results in a linear bandwidth increase with the number of threads. We conclude that when multiple threads on the same

[8] http://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture



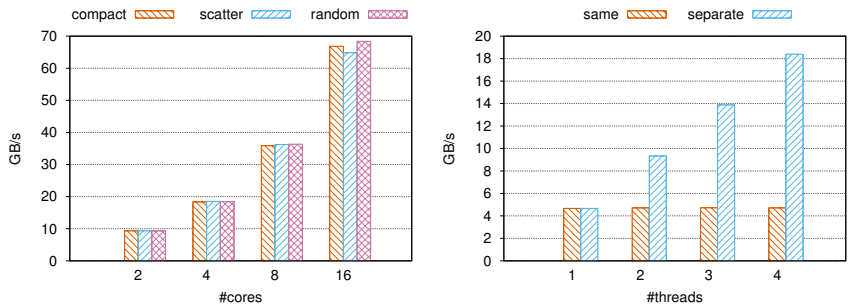
(a) scale1 bandwidth.

(b) scale2 bandwidth.

(c) saxpy1 bandwidth.

(d) saxpy2 bandwidth.

Figure A.8: Cache bandwidth on a single core.



(a) Core Distribution

(b) Thread Distribution

Figure A.9: Core and thread distribution effects (we use the `read` kernel and the array size is 1 GB).

core request data simultaneously, they will compete for the shared hardware resources (e.g., the ring stops), thus serializing the requests. On the bright side, the threads located on the same core share cache data and have faster data accesses (see Section A.2.2).

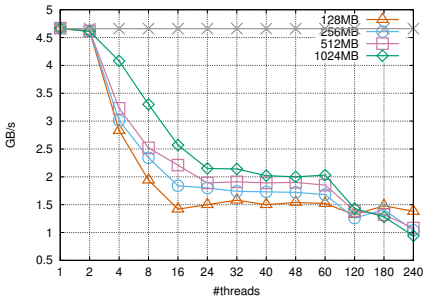
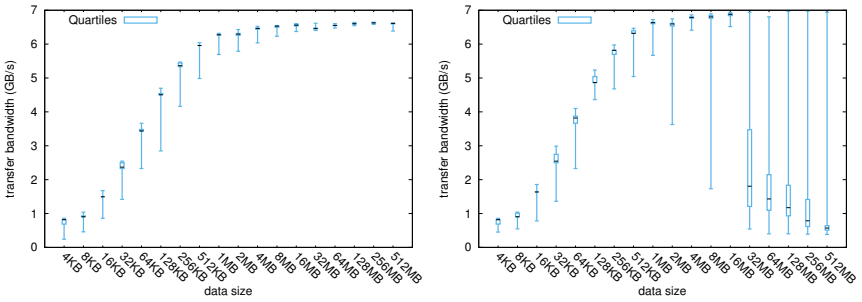


Figure A.10: The memory bandwidth when the threads read the same memory space.



(a) Host ⟹ Phi (b) Phi ⟹ Host

Figure A.11: Achieved data transfer bandwidth (over PCIe) between a host and an Xeon Phi.

ACCESSING SHARED-DATA

Section A.2.4 focuses on the achieved bandwidth when threads access separate memory spaces. In this section we investigate **what is the bandwidth when different threads access the same memory space simultaneously?** We expect that the bandwidth would resemble that obtained by a single thread, assuming the memory requests are served by *broadcasting*. Figure A.10 presents the measured bandwidth, showing that the `read` bandwidth decreases over the number of threads until 24 (or 16). Thereafter, the bandwidth is constant around 1.5-2.0 GB/s (i.e., one third of the single thread bandwidth). When using more threads than cores, the bandwidth drops even further. This behavior is different from the linear increase trend (shown in Figure A.9a) seen when accessing separate memory spaces. We assume the bottleneck lies in the simultaneous access to the DTDs. Therefore, for bandwidth gain, applications should strive to keep threads accessing different parts/cachelines of the shared memory space (for as much as possible), to avoid the effects of contention at the interconnect level.

A.2.5. PCIE DATA TRANSFER

When used as a coprocessor, Xeon Phi is connected via PCIe to a host (e.g., a traditional CPU). When offloading computation to the Xeon Phi, the tasks and the related data need

to be transferred back and forth between the two processors. As seen for GPUs [164], these transfers can be expensive in terms of overall application performance. Thus, we have designed a benchmark to measure the data transfer bandwidth. To do so, we use the `offload` pragma (specifying `in` and `out` for the transfer direction) to transfer datasets of different sizes from host to Xeon Phi and back. The transferred data is allocated with a 4KB alignment, for optimal DMA performance [66].

The achieved bandwidth between host and Xeon Phi is presented in Figure A.11 (we report the results over 1000 times). We note that the bandwidth increases with data size, and it is relatively stable for different runs, for both directions. However, for data transfers larger than 32 MB, the Phi to host bandwidth shows a large variation, with the `median` bandwidth value decreasing sharply (up to 6 times!). The reasons for this large variance are still under investigation.

A.3. SCAT: AN XEON PHI MODEL

We compare our results with the information provided by the Intel Software Development Guide (SDG) in Table A.2. We note that we did improve on the content of the official data: instruction latency data, local and non-local memory access bandwidth and latency data, an interconnect study, and a PCIe offload evaluation. We also have the following key observations, which lead to optimization guidelines.

High Throughput: Xeon Phi is indeed a high-throughput platform. The peak instruction throughput is achievable, but it depends on the following factors: (1) the number of threads and threads/core occupancy, (2) the utilization of the 512-bit vectors, (3) the issuing width (i.e., the number of independent instruction streams), (4) the instruction mix. Furthermore, using single-precision data leads to better performance for math-

Table A.2: A comparison with the data in SDG ('N/A' stands for "not available" in SDG).

Metric	SDG	Measured
VPU		
Latency	general statement	cycles/instruction
Throughput	1008 GFlops	1008 GFlops
EMU evaluation	general statement	quantified
L1 Cache (32KB)		
Latency (local)	1 cycle	3 cycles
bandwidth (local)	N/A	R=64B/c;W=64B/c
L2 Cache (<512KB)		
Latency (local)	11 cycles	24 cycles
Bandwidth (local)	N/A	quantified
Latency (remote)	N/A	250 cycles
Off-chip memory		
Latency	N/A	302 cycles
Bandwidth	320 GB/s	R=164GB/s;W=76GB/s
Prefetching	general statement	quantified
ECC factor	general statement	quantified
Interconnections		
Ring Traffic Contention	N/A	ring stops, DTDs
PCI Express Bandwidth	N/A	up to 7 GB/s

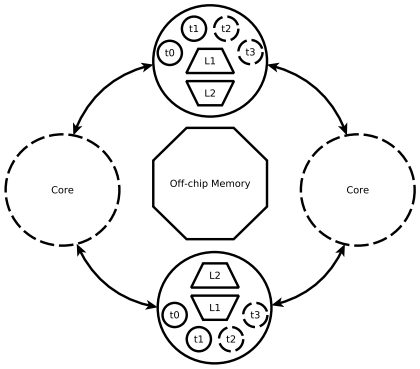


Figure A.12: The SCAT model of Intel Xeon Phi.

intensive kernels (than using a double-precision version).

Memory Selection: Accessing the local L1 cache is 8 times faster than accessing the local L2 cache, which is again an order of magnitude faster than accessing the remote caches or the off-chip memory. However, the difference between a remote cache access and an off-chip memory access is relatively small (17%). Furthermore, the remote access latency does not depend on the cache-line state.

Efficient Memory Access: Data is read and written from/to the off-chip memory in cache lines (64 bytes). The maximum achievable bandwidth is 164 GB/s for `read` operations and 76 GB/s for `write` operations - a lot lower than the theoretical peak of 320 GB/s. With `streaming store` instructions, the `write` bandwidth can increase up to 1.7 times. Further, programmers need many threads (at least 60 - one per core) to issue enough memory requests to saturate the ring interconnect and the memory channels. The hardware and software prefetching can improve bandwidth; their efficiency increases with the length of the `stanzas` of contiguous memory accesses. Finally, disabling ECC leads to an average of 20% increase in bandwidth.

Ring Interconnect: All cores can be seen as *symmetrical* peers, and the distance between cores has little impact on performance. However, memory requests from threads running on the same core are serialized, provided that the bandwidth reaches 4.7 GB/s. Furthermore, when threads (on different cores) are accessing the same data, the simultaneous access to the DTD leads to a bandwidth loss.

Overall, we believe our results are complementary to the SDG, and, being backed up by more practical guidelines, be of added value for programmers using this platform.

SCAT Model: Based on the numbers and the observations, we attempt to build a simple view of the Xeon Phi, providing production users with a platform model for reasoning about parallel algorithm design and performance optimization. Figure A.12 shows the machine model for Xeon Phi. The machine has 60 symmetrical cores, each of which contains 1/2(/3/4) vector threads working on 8 double-precision or 16 single-precision data elements in a lock-step manner. `Family threads` (threads suited in the same core) differ from `remote threads` (threads suited in another core) in that they share and compete local resources. Furthermore, compared with accessing local caches, remote caches

and off-chip memory are slow (see the numbers in Table A.2). We summarize the model as *SCAT* (symmetric cores and asymmetric threads).

This machine model limits itself to those architectural details that are important for performance. For example, programmers do not have to keep the ring interconnect in mind because the cores perform symmetrically. On the other hand, the threads on the same core share and compete the shared resources, putting up `asymmetry` and impelling us to take care of `thread affinity`. Therefore, this platform model captures the key performance features of the processor, ensuring good performance with relatively low programming effort (i.e., using high-level programming tools).

A.4. LEUKOCYTE TRACKING

In this section, we focus on our case-study application, Leukocyte Tracking. Specifically, we aim to evaluate the gap(s) between the achieved performance of the application and the performance indicated by the microbenchmarks.

Leukocyte Tracking is a medical imaging application which detects and tracks rolling leukocytes (white blood cells) in vivo video microscopy of blood vessels. The velocity of rolling leukocytes provides important information about the inflammation process, which aids biomedical researchers in the development of anti-inflammatory medications [130].

In the application, cells are detected in the first video frame and then tracked through subsequent frames [130]. Tracking accounts for around 90% of the total execution time and thus we focus on this procedure. Tracking is accomplished by first computing, in the area surrounding each cell, a Motion Gradient Vector Flow (MGVF) matrix. The MGVF is a gradient field biased in the direction of blood flow, and it is computed using an iterative Jacobian solution procedure. After computing the MGVF, an active contour is used once again to refine the shape and determine the new location of each cell. Unfortunately, leukocyte tracking is computationally expensive, requiring more than four and a half hours to process one minute of video. Boyer et al. have translated the tracking algorithm from Matlab to C and OpenMP [19].

A.4.1. PERFORMANCE ANALYSIS

Without any code changes, we compile and run the kernel on both Phi and SNB (Intel Xeon E5-2620, a dual 6-core processor with hyper-threading disabled), and show their performance in Figure A.13. We see that, on SNB, the execution time decreases when increasing the number of threads. On Phi, the execution time decreases when the number of threads is less than 40. Using more than 40 threads brings no further performance gain. Overall, we note that the performance on Phi (with 40 threads) is 2× worse than that on SNB (with 12 threads), while the sequential execution of the same application (i.e., running on a single thread) on Xeon Phi is 5× slower than on SNB.

To further understand these results, we analyze the overall performance by taking both parallelism and per-thread performance into account, and focus on two aspects: (1) the single-thread performance and (2) scalability. The analysis includes the interactions between kernel characteristics and processor features.

Single thread: When tracking a leukocyte, we use 18 data structures/matrix (1 input

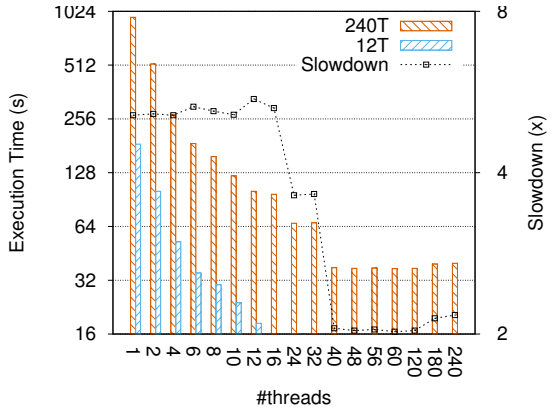


Figure A.13: The initial performance results of Leukocyte Tracking on an Xeon Phi processor (240 threads) and an Xeon processor (12 threads).

sub-image, 1 motion gradient vector field, 8 neighbours to store intensity differences, and 8 neighbours to store the heaviside value). For the given input dataset, each matrix has 41×81 elements (in double-precision). In total, tracking a leukocyte needs 467 KB ($18 \times 41 \times 81 \times 8$), which is smaller than the size of a local L2 cache (Figure A.12 and Table A.2). Thus, the iterative Jacobian solver will work intensively on tracking a leukocyte with data located on-chip, and the tracking speed is not limited by the memory access.

As for computation without vectorization, a thread on Phi (working at 1.05 GHz and issuing an instruction every two cycles, see Section A.2.1) runs 4× slower than one on SNB (working at 2.0 GHz and issuing instructions every cycle). With vectorization, the difference is lowered to roughly 2×. In our practical experience, the single thread performance on Phi is around 5× worse than that on SNB, an indication that vectorization is not applied on both platforms. Indeed, the compiler reports an auto-vectorization failure (consequently, only 12.5% of the SIMD lanes are used).

Scalability: Figure A.13 shows that the performance on Phi varies little when using over 40 threads. Through code analysis, we observed that parallelization is performed over the number of leukocytes. As the number of leukocytes from the input datasets is 36, increasing the number of threads to more than 36 brings no performance gain. In other words, the kernel parallelism does not match Phi's massive hardware parallelism ($36 << 240$, see Figure A.12). On the other hand, SNB has only has 12 threads, showing much better scalability. To fully utilize the hardware resources on Phi, we *must* increase the paralellism of the application.

A.4.2. PERFORMANCE OPTIMIZATION

VECTORIZING THE KERNEL

When tracking a leukocyte, the kernel loops over a fixed-sized portion of a frame (a sub-image with 41×81 pixels). A typical loop is shown in Figure A.14 ($m = 41$, $n = 81$). As we have mentioned, the compiler fails to vectorize this code due to the assumption of data dependency (the original code uses `pointers to pointers` and dereferencing

```
1  input: MAT* z, double v, double e
2  output: MAT* H
3  double one_over_pi = 1.0 / PI;
4  double one_over_e = 1.0 / e;
5  for (i = 0; i < m; i++) {
6      for (j = 0; j < n; j++) {
7      double z_val=m_get_val(z, i, j)*v;
8      double H_val=one_over_pi * \
9             atan(z_val*one_over_e)+0.5;
10     m_set_val(H, i, j, H_val);
11     }
12 }
```

Figure A.14: The Heaviside step function.

the data structure is too complex for the compiler to automate).

We note that enabling vectorization for these cases requires an intervention from the programmer. The typical approach for manual vectorization is to add low-level intrinsics in the high-level C code, thus specifically instructing the compiler to use the vector units.

We identify three main factors that make code vectorization for leukocyte tracking cumbersome. First, `data alignment`: when vectorizing the code, data accesses should start with an address aligned at 64 bytes (512 bits). This must be ensured with specific memory allocation (i.e., dedicated APIs). Second, the `non-unit-strided memory access`: when the 8 elements in a vector are non-contiguous, the offset for each element must be specified. This occurs when calculating the gradient in the tracking kernel. Third, and final, vectorizing a loop requires special care when the number of iterations is not a multiple of the vector length. Thus, we also need to deal with the `remainder` of the inner-loop (i.e., because $n\%8 \neq 0$). Therefore, we use two loops in the tracking kernel: a vector loop (for the bulk of the computation), and a scalar loop (used to deal with the loop remainder).

Fixing all these problems (and thus manually vectorizing this code using intrinsics) takes an expert programmer two days. Moreover, the kernel code doubles in size (from ~200 lines to ~400 lines). Correspondingly, the tracking time per frame decreases to 8.5s from 31s (~4× faster) on Phi. The remaining optimization space is roughly 2×. The limiting factor is that the kernel uses trigonometric operations, which can be further optimized by using EMU (Section A.2.1 and [50]).

CHANGING PARALLELISM

As we have mentioned, the parallelism of leukocyte tracking is limited by the number of leukocytes (36 in the given data set). For the traditional multi-core processors, this number is still larger than that of the hardware threads. But on a Phi with 240 hardware threads, running the tracking kernel with 36 parallel threads can never fully utilize the platform.

We attempt to improve on this situation by increasing parallelism. Thus, we spawn a second-level parallelism over the outer loop of the sub-image in Figure A.14. Next, we need to tune the dimensions of the two parallel levels by specifying the number of first-level threads (FLT) and the number of second-level threads (SLT). We select these

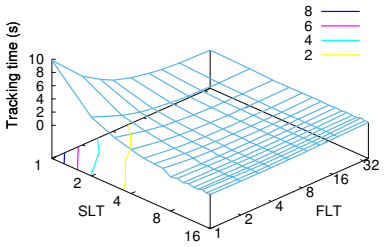


Figure A.15: The selection of FLT and SLT.

two numbers from those that satisfy the following constraints: (1) $FLT \times SLT \leq 240$, (2) $FLT \leq 36$, (3) $SLT \leq 41$. We autotune the kernel using $FLS \in \{1,2,3,4,6,9,12,18,36\}$ and $SLT \in \{1,2,3,4,5,6,7,8,9,10,\ 11,12,13,14,15,16\}$ [9]. Figure A.15 shows the tracking time per frame for different combinations. We see that the best performance achieved by Phi is around 0.1s per frame when $FLT = 4$ and $SLT = 8$, indicating that using more threads does not mean a faster tracking ($4 \times 8 < 240$). According to the SCAT model (Figure A.12), it is of no use binding multiple threads to the same core due to little data reuse.

OVERALL PERFORMANCE

We compare the execution time of tracking leukocytes per frame on Phi against the ones achieved by SNB (using a higher clock and better performing cores, but a lot less parallelism), and on an NVIDIA Kepler GPU (K20m, a GPU with a similar peak performance and more massive parallelism, programmed in CUDA implementation [10]). The comparison is illustrated in Table A.3. We notice that Xeon Phi is 6× faster than SNB, while it is around 40% slower than K20. Admittedly, optimizing the tracking kernel on SNB (by hand-tuning for enabling vectorization) can lead to a performance increase (a maximum 4×, most likely, with SNB-specific intrinsics). K20 performs better than Phi due to the more efficient reduction implemented in the GPU shared memory [19]. Specifically, at the second level of parallelism, we use multiple threads that are bound to separate cores on Phi, while the CUDA implementation runs the same amount of work on a block (and a multi-processor). Thus, when performing reduction, the shared (reduction-)variable on Phi has to be transferred back and forth at the second-level cache. As we have measured, the remote cache access is as slow as accessing the off-chip memory (Figure A.12 and Table A.2). With CUDA on K20, this reduction happens in shared memory, with much higher performance. The final code of leukocyte tracking for Phi is publicly available [11].

A.4.3. DISCUSSION

One of the important selling points of Phi is the continuity of programming models from the traditional multi-core processors - the OpenMP and MPI models and codes are func-

[9] SLT can be as large as 41, but our results show a large SLT is not necessary due to the limited per-thread work.
[10] We change the original Rodinia to a double-precision version for a fair comparison.
[11] https://github.com/haibo031031/mic-apps

Table A.3: Tracking time per frame (in seconds). 'VEC' represents 'vectorization'; 'FMT/SMT' is 'to use the first-level/second-level multi-threading', respectively. The optimizations are incrementally added.

	1T	+VEC	+FMT	+SMT	Overall
Phi	31	8.5	0.7	0.1	0.1
SNB	6	–	0.6	–	0.6
K20	–	–	–	–	0.06

tionally compatible. Ideally, programmers should obtain high performance without a lot of investment in programming model learning (e.g., OpenCL), tuning and hacking low-level code (e.g., assembly code with pthreads). Effectively, the expectation is that re-compiling the code with the `-mmic` option will do. Our experience leads to a different conclusion: porting legacy code or developing new code still needs a lot of developer interventions.

Note 1. Using intrinsics indeed brings us a significant performance gain, but it exposes low-level implementation details to users, conflicting with the principles of encapsulation and high-level programming. It also requires code specialized for Phi, which will further fail to run the on traditional multi-core processors. This deviates from the original design goal of Phi, i.e., to keep using traditional programming models. A possible solution is to provide a high-level vector template/library/model (e.g., ispc [12]). The template can present users with the required operations (e.g., multiply and reduction). When implementing the template, we translate the operations into their equivalent intrinsics specialized to a platform. Thus, we can keep code portable while not hindering performance.

Note 2. Xeon Phi truly needs massive parallelism to fully use the hardware threads. This observation makes a significant difference between SNB and Phi. SNB has a dozen of hardware threads, while Phi has over a two hundred. Only those applications with abundant parallelism can fully utilize the machine. When lacking parallelism, applications can either look for finer grain parallelism (atypical for OpenMP, but useful when available), or find a way to load multiple (independent) tasks on the platform. However, note that the number of required threads depends on applications and their run-time contexts.

Note 3. On Xeon Phi, using OpenMP can perform global reduction on the globally shared caches, but this proves to be less efficient than expected (apparently due to frequent memory transfers). When using CUDA/OpenCL on GPUs, an efficient reduction can be performed in shared memory (or local memory in OpenCL) at the block (or work-group in OpenCL) level. Further, our experience shows that the leukocyte tracking maps more naturally to the GPU architecture: mapping a leukocyte to a multi-processor. While on Phi, we need to map a leukocyte to multiple processing cores. Thus, we believe that a multiprocessor on GPUs is equivalent to multiple processing cores on Phi, at least in the context of leukocyte tracking.

To summarize, we conclude that (1) although it often destroys portability, manual vectorization is mandatory for exploiting Phi's performance; a high-level library can be

[12] http://ispc.github.io/

used to hide the platform-dependent details, but vectorization *must* be enabled as much as possible, and (2) massive parallelism is needed on Phi to fully use the hardware. In a nutshell, merely relying on compilers with traditional programming models to achieve high performance on Phi is still far from reality.

A.5. RELATED WORK

In this section, we survey and briefly discuss the work related to our (micro)benchmarking approach. We focus mainly on existing CPU and GPU benchmarking methods, as there are no other comprehensive studies of Xeon Phi - yet.

In [144], the authors develop a high-level program to evaluate the cache and TLB for any machine. Part of our work is based on their approaches (targeting uni-core processors, though). Multiple studies are also performed on multi-core CPUs. In [126], the authors report performance numbers from three multi-core processors , including not only execution time and throughput, but also a detailed analysis on the memory hierarchy performance and on the performance scalability between single and dual cores. Daniel Molka et al. [104] revealed many fundamental details of the Intel Nehalem using benchmarks for latency and bandwidth between different locations in the memory subsystem. We use similar approaches for the access latency of remote caches.

For GPUs, Volkov et al. [164] presented detailed benchmarking of the GPU memory system that reveals sizes and latencies of caches and TLB. Later, Wong et al. [169] presented an analysis of the NVIDIA GT200 GPU and their measurement techniques. They used a set of micro-benchmarks to reveal architectural details of the processing cores and the memory hierarchies. Their results revealed the presence of some undocumented hardware structures. While these microbenchmarks are in CUDA and targeted NVIDIA GPUs, Thoman et al. [152] develop a set of OpenCL benchmarks targeting a large variety of platforms. They include code designed to determine parameters unique to OpenCL, like the dynamic branching penalties prevalent on GPUs. They also demonstrate how their results can be used to guide algorithm design and optimization

Garea et al. [129] developed an intuitive performance model for cache-coherent architectures and demonstrated its use on Intel Xeon Phi. Their model is based on latency measurements, which match well with our latency results. In addition to the cache access latency, we have shown how we benchmark the instruction throughput, the memory bandwidth at different levels, and the interconnect performance.

A.6. SUMMARY

Given its performance promises, Intel Xeon Phi is very likely to become popular for both low-end high performance computing applications (smaller scale scientific applications like Leukocyte Tracking), and the next generation of supercomputers. In this work, we presented our hands-on experience with this processor - in both the "lab" and using a real application - and discussed several key insights into the performance of this new many-core processor. By using a set of self-designed microbenchmarks, we characterized the major components of this architecture - cores, memory, and interconnections - summarizing them into four machine-centric observations (potential optimization guidelines). We also made a first attempt to provide a simple machine view (*SCAT*)

to facilitate application design and performance tuning on the Xeon Phi.

In general, our benchmarking results are consistent with Xeon Phi's published data. However, the data we have added through this benchmarking effort allowed us to expose more accurately the expected key performance factors for the Xeon Phi. We have shown that the platform is able to deliver its performance promises in terms of computation, but programmers will need to find the right parallelization strategy to fill 240 hardware threads with compute-intensive tasks, while finding the right balance between data partitioning and coherent memory requests to achieve sufficient memory bandwidth. Thus, we believe the number of applications that can easily use Xeon Phi's potential in their existing, naive form is, for now, very limited. And for high performance, our and others' experience shows that programmers need to take a lot of efforts on parallelization, analysis, and optimization.

B

AUTO-TUNING CLUSTERING DATA STREAMS

In this appendix, we show our hands-on experience of porting a Clustering Data Streams (in Rodinia benchmark suite) from CUDA to OpenCL, and present our memory-centric optimization strategies for it on NVIDIA GPUs and AMD GPUs. Furthermore, we present a model to auto-tune this application. This appendix shows an instance of the auto-tuning module in the *Sesame* framework (Chapter 8). This appendix serves as yet another case study on how the optimization space depends on platforms (e.g., using pinned memory has different performance impacts on two GPUs).

The objective of Clustering Data Streams (CDS) is, given a sequence of points, to construct a good clustering of the stream, using a small amount of memory and time. A data stream is a massive, continuous and rapid sequence of data elements [93]. Typically, these data elements can be read only once or a small number of times. Each reading of the sequence is called a linear scan or a pass. This data stream model is widely applied when modelling customer click streams, telephone records, multimedia data, financial transactions, and so on.

As a useful and ubiquitous tool in data analysis, clustering is the problem of finding a partition of a data set so that, under some definition of "similarity", similar items are in the same group of the partition, while different items are in different groups. Clustering Data Streams studies clustering in the data stream context. In [60], the authors provide a clear overview of CDS and its algorithm.

Data Streams are usually far too large to fit in the main memory. Therefore, memory usage becomes an important limiting factor for CDS algorithms. When it comes to GPU platforms, the data transfer between the host and the device should be reduced as much as possible; otherwise, it will become an additional bottleneck.

Several solutions have been proposed for this application. In this work, we start from the SC algorithm in Rodinia [1], and focus on optimizing its memory usage. Specifically,

This appendix is based on our work published in the Proceedings of CSE 2011 [48].

we propose a rake-based memory-efficient solution to CDS. Our rake-based optimization was inspired by 'Loop Raking' (see [176] for a detailed description), which is extensively used nowadays in GPU applications and labeled as 'multi-output' strategy [163]. The basic idea is to let each work-item work on multiple data elements. Then the question becomes: how many such elements should be assigned to one work-item, namely the rake-size? As the rake-size has a significant effect on performance, it is difficult to maximize performance for all problem sizes by setting one fixed value. To address this issue, we also present an auto-tuning solution to select the optimal rake-size per platform and problem-size.

To allow for a portable solution and easy auto-tuning, we propose an OpenCL implementation of CDS. Although OpenCL cannot mask significant differences in the architectures, it does guarantee portability and correctness. Therefore, it is much easier for developers to start with a correctly functioning OpenCL program tuned for one architecture and produce a correctly functioning program optimized for another architecture [147].

To summarize, we make the following contributions: (1) We provide an OpenCL implementation of the SC benchmark from Rodinia, which enables the program to run on various OpenCL-compliant platforms. (2) We apply several optimizations to get better performance, including a rake-based optimization to use memory more efficiently. (3) We propose a model-driven auto-tuning method to maximize the CDS performance.

The rest of the work is organized as follows: Our optimizations and their performance impacts are explained in Section B.1. We propose a simple model to auto-tune performance of CDS in Section B.2, where experimental results are also shown. Section B.3 presents related work in CDS and auto-tuning on GPUs. Finally, we summarize the work in Section B.4.

B.1. HAND-OPTIMIZING CDS IN OPENCL

In [93], the authors discussed five different algorithms for CDS: divide and conquer, doubling, statistical grid-based, STREAM and CluStream. The algorithm used in our work is based on STREAM, a single-pass, constant factor approximation algorithm that was developed based on the k-median problem. The algorithm divides the entire data stream into pieces of m data points. For each piece, STREAM clusters the piece's points into k clusters/groups by using $LOCALSEARCH$ algorithm: it summarizes information of each piece by maintaining only the information regarding the piece centers (as intermediate centers) and their weights, and then discards the other points. After seeing the entire data stream, it will cluster all the intermediate centers into k final centers. The call graph of the implementation is shown in Figure B.1.

In order to speedup $LOCALSREACH$, the authors relaxed the number of centers (larger than k) in the intermediate steps, and achieved exactly k centers in the final step [60]. During this process, they use a $gain$ function to judge whether it is worth opening a facility: given a preliminary solution, the function computes how much cost can be saved by opening a new center; for every new point, it weighs the cost of making it a new center and reassigning some of the existing points to it against the savings caused by minimizing the distance between two points x and y for all points.

The original CUDA version parallelized the $gain$ function [1]. In the following sub-

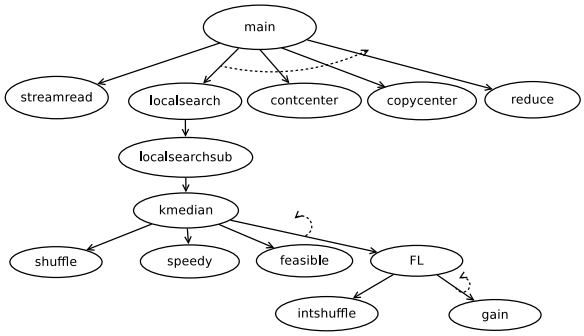


Figure B.1: The call graph of the STREAM-based implementation. The dashed arrows represent that the callee functions will be invoked for multiple times in the order of left-to-right.

sections, we propose several optimizations based on the CUDA implementation. For the majority of the paper, we will use the OpenCL terms (for example, work-item, and work-group, etc.) for illustration, so please refer to [151] for more details about OpenCL.

B.1.1. A MEMORY-EFFICIENT SOLUTION

The key requirement of CDS is to make real-time data processing using limited memory. Therefore, a memory-efficient solution plays an important role in this application. In order to save memory, we present a rake-based solution, inspired by [176] and [163]. As is illustrated in Figure B.2, the idea of our rake-based solution is to let each work-item process several nonconsecutive elements. Neighboring work-items (i.e. work-items with consecutive numbering) work on consecutive elements in a similar manner to how tines work in a rake. The original implementation can be seen as using only one rake.

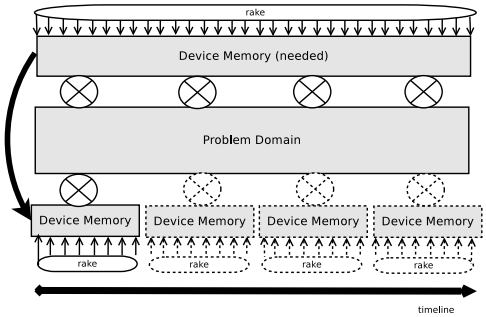


Figure B.2: Our rake-based solution when there are 4 rakes.The upper part shows that the original implementation has only one rake, while the lower part shows there are several (e.g. 4 rakes here) rakes to take charge of the whole task. The short arrows represent work-items on GPU; When using four rakes, we only have to allocate 1/4 of the memory usage of the original implementation, thus reducing both the memory allocation and data transfer.

In order to store 'cost' states, the existing many-core solution has to allocate $Kmax$ (the maximum number of centers) elements for each work-item. By contrast, our rake-

based solution processes the whole problem domain piece by piece and enables the re-usage of device memory (shown in Figure B.2), with at least two advantages: (1) it saves a lot of memory space, and (2) it can finish the whole task by transferring less data between the host and the device (i.e. it is also faster).

SAVED MEMORY SPACE

Let R represent the number of rakes, and S represent the memory amount allocated to store the 'cost' state in the original solution. When using R rakes ($R = 4$ in Figure B.2), our optimized implementation only needs to allocate $1/R$ memory used by the original one to store the 'cost' state. Therefore, the saved memory amount ΔS can be calculated using Equation B.1.

$$\Delta S(R) = S \times (1 - \frac{1}{R}) \tag{B.1}$$

SAVED TIME

The data transferring time can be calculated according to Equation B.2.

$$T_{data}(R) = L + \frac{S_{data}(R)}{BW} \tag{B.2}$$

where L represents latency, including overhead to startup kernels, to prepare data, etc, S_{data} represents the amount of data to be transferred (here $S_{data} = S$ in Equation B.1), and BW stands for the bandwidth of transferring data between devices and the host.

Then we can derive the saved time ΔT_{data} as follows:

$$\Delta T_{data}(R_1, R_2) = \frac{S_{data}(R_1)}{BW(R_1)} - \frac{S_{data}(R_2)}{BW(R_2)} \tag{B.3}$$

where R_1, R_2 stand for two different rake-sizes. In this Equation, BW changes with rake-size R, and they are different when $R = R_1$ and $R = R_2$. Additionally, L is independent of rake-size R. Thus there is no L component in Equation B.3.

To summarize, when increasing R, we can save both memory space and data transferring time. However, when R becomes too large, there will be less work-groups in each rake, which means there will not be enough work-groups to hide latency, thus leading to poorer performance. Therefore, we have to take the variation of R into account for maximizing performance.

B.1.2. FURTHER OPTIMIZATIONS

USING LOOP-UNROLLING

To maximize performance, we should reduce the number of dynamic instructions. One of the effective ways to do so is loop unrolling [106]. In CDS, the kernel program will calculate the distance to the candidate point x. When the dimension of points is high, loop-unrolling becomes necessary.

USING PINNED MEMORY

CDS will frequently copy data between the host and the device. Therefore, it is important to ensure a high bandwidth of data copying; otherwise, it will become a bottleneck.

Pinned memory is an important way to improve the bandwidth between devices and the host, which is illustrated in Table B.1. As can be seen from the Table, the bandwidth can be boosted by up to 25% when using pinned memory on GTX480; there are no significant changes on HD5870 (H2D: the peak data transferring bandwidth from the host to the device; D2H: the peak data transferring bandwidth from the device to the host).

Table B.1: Peak Bandwidth with/without Pinned Memory: MB/s

	H2D		D2H	
	w	w/o	w	w/o
HD5870	1385.6	1381.3	1455.3	1464.6
GTX480	5646.2	5176.2	6107.1	4901.4

B.1.3. EXPERIMENTAL RESULTS

We measure the performance of the CDS implementations, and we use the execution time (in seconds) as the comparison metric. The execution time refers to the sum of the kernel execution time (KE), and data transferring time from the device to the host (D2H), because the other components of the run-time are the same before and after our optimizations.

TESTBED SPECIFICATIONS

We have selected GTX480 from NVIDIA and HD5870 from AMD as our testbeds. Their specifications are shown in Table B.2 (MIW stands for Memory Interface Width).

Table B.2: Specifications of the Selected GPUs Testbeds

	GTX480	HD5870
Architecture	Fermi	Cypress
#Compute Unit	60	20
#Cores	480	320
#Processing Elements	480	1600
Core Clock(MHz)	1401	850
Memory Clock(MHz)	1848	1200
MIW(bits)	384	256
Memory Capacity(GB)	1.5	1

PERFORMANCE COMPARISON OF DIFFERENT OPTIMIZATIONS

We make a performance comparison of different optimizations on GTX480 and on HD5870. The results are shown in Figure B.3 (the execution time is normalized as $speedup$ relative to the native version). We can see that the version with all the optimizations enabled can achieve 2.1x speedup on GTX480, while the speedup is only around 1.4x on HD5870. Moreover, the loop-unrolling and pinned-memory techniques have little effect on the whole performance, especially on HD5870.

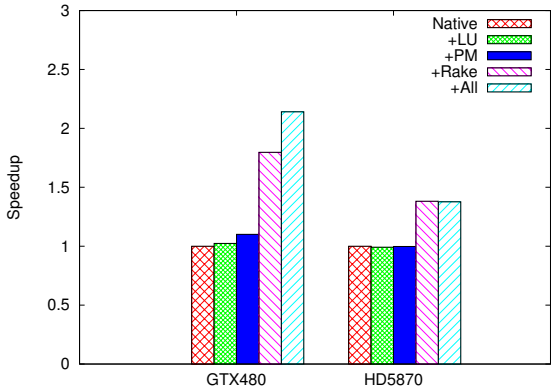


Figure B.3: Performance comparison of different optimizations. Native: the original version in OpenCL; +LU: the native implementation with loop-unrolling optimization; +PM: the native implementation with pinned-memory optimization; +Rake: the native implementation with rake-based optimization; +All: the native implementation with all the optimizations mentioned above.

PERFORMANCE COMPARISON WITH THE ORIGINAL IMPLEMENTATION

We compare the original implementation (in CUDA) with our fully optimized implementation (in OpenCL) on GTX480. Figure B.4 shows that our fully optimized version can perform 1.4x to 3.3x faster than the original one; the performance gain is higher when the problem size is larger.

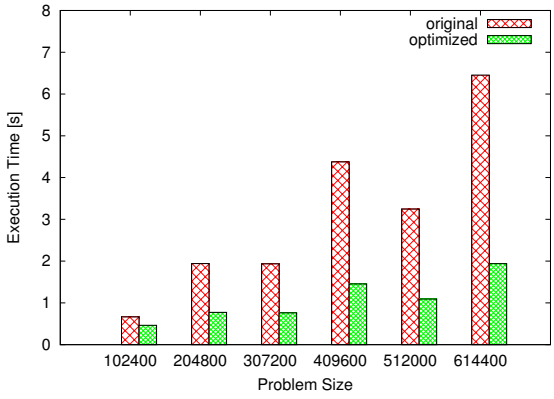


Figure B.4: Performance comparison between the original implementation and our optimized implementation (the problem size in CDS represents the number of points processed in each pass.)

B.2. AUTO-TUNING

In the previous section we proposed a rake-based solution to boost performance in both memory consumption and execution time. In this section, we detail the trade-off in

Table B.3: Hardware-related Parameters: descriptions and values

Parameter	Descriptions	GTX480
WI_{warp}	Number of work-items per warp	32
WI_{mp}	Maximum number of work-items per multiprocessor	1536
MW_{mp}	Maximum number of warps per multiprocessor	48
WG_{mp}	Maximum number of work-groups per multiprocessor	8
Reg_{mp}	Number of 32-bit registers per multiprocessor	32K
LM_{mp}	Maximum amount of local memory per multiprocessor	48K
MP	the number of multiprocessors	15

Table B.4: Application-related Parameters: descriptions and values

Parameter	Descriptions	CDS
WI_{app}	the total number of work-items in the application	409600
DIM	the dimension of points	16
BS	the size of work groups used in the application	512
Reg_{wi}	the number of registers per work-item	23
LM_{wg}	the amount of local memory used per work-group	64B

choosing a proper R when it comes to different platforms and problem sizes, by presenting a simple model.

Let T_{total} represent the total execution time, T_{data} represent the time taken to transfer data (illustrated in Equation B.2), and T_{kernel} represent the time taken to execute kernel functions. Therefore, we get the target function:

$$T_{total}(R) = T_{data}(R) + T_{kernel}(R) \tag{B.4}$$

Our goal is to minimize T_{total}, thus getting optimal performance. First of all, we draw qualitative curves to describe how T_{data} and T_{kernel} change with R. These are illustrated in Figure B.5.

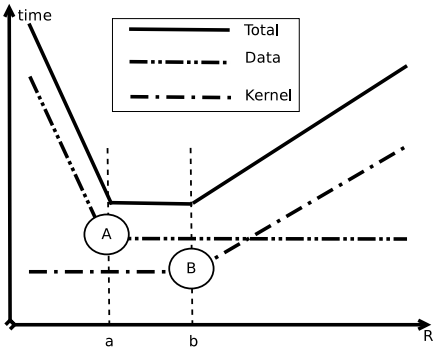


Figure B.5: Qualitative curves to show how T_{data} and T_{kernel} change with R: when $a < b$.

When R increases, T_{data} will decrease due to transferring less data from the device to the host. After reaching point A ($R = a$), T_{data} is mainly determined by L according to

Equation B.2. Therefore, T_{data} will remain almost stable after point A. At the same time, when R increases, T_{kernel} will remain the same because of enough work-groups on each compute unit. After point B, there are very few work-groups to hide latency, thus leading to decreasing performance. We can get the total time T_{total} curve by adding together the two curves shown in Figure B.5. Since we do not know whether $a > b$ or $a < b$, these two cases are discussed separately. Note that, for the simplicity of explanations, we use some CUDA terms (for example, warps, multiprocessor, etc.) as supplements to OpenCL terms in the following discussion.

B.2.1. CASE: WHEN $a < b$

When $a < b$, the optimal $R \in [a, b]$. Moreover, when R increases from a to b, T_{data} will decline slightly due to transferring less data. Therefore, we choose $R = b$ for the optimal performance, and the problem becomes how to determine b.

$R = b$ is the transition point from where there are enough active warps to where there are not. Similar to the method of calculating $occupancy$ in CUDA [118], we can compute the maximum number of active warps (MW) per multiprocessor using the information of the CDS application and the platform. The hardware-related parameters and application-related parameters are described in Table B.3 and Table B.4, respectively.

MW can be calculated in the following four steps:

WORK-ITEM LIMIT

MW limited by work-item specifications can be calculated as follows:

$$MW_{wi} = \frac{\min(WI_{mp}, WG_{mp} \times BS)}{WI_{warp}} \tag{B.5}$$

REGISTER LIMIT

MW limited by register resources can be calculated as follows:

$$MW_{reg} = \left\lfloor \frac{Reg_{mp}}{Reg_{wi} \times WI_{warp}} \right\rfloor \tag{B.6}$$

LOCAL MEMORY LIMIT

MW limited by local memory can be calculated as follows:

$$MW_{lm} = \left\lfloor \frac{LM_{mp}}{LM_{wg}} \times \frac{BS}{WI_{warp}} \right\rfloor \tag{B.7}$$

PUT THEM TOGETHER

$$MW = \min(MW_{mp}, MW_{wi}, MW_{reg}, MW_{lm}) \tag{B.8}$$

Then we can calculate b using Equation B.9:

$$b = \left\lceil \frac{WI_{app}}{MP \times MW \times MI_{warp}} \right\rceil \tag{B.9}$$

Given the device (GTX480) and the application case (when the problem size is 409600), we can calculate $MW = 32$ using Equation B.8, and $b = 27$ using Equation B.9. Finally, the experimental results show that the $R = 28$ ($R \approx b$).

B.2.2. CASE: WHEN $a > b$

When $a > b$, it is unclear how the total time T_{total} will change from b to a (denoted by a question mark illustrated in Figure B.6), but it is still use that the optimal $R \in [b, a]$. We first estimate a, calculate b using the same method as the one mentioned in the previous subsection, and then use an empirical search to get the optimal R.

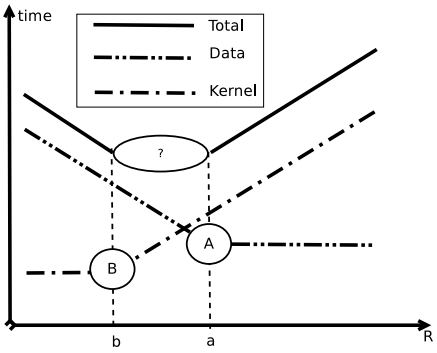


Figure B.6: Qualitative curves to show how T_{data} and T_{kernel} change with R: when $a > b$

Table B.5: The R Comparison between Predicted and Exhaustive

	51200	102400	204800	307200	409600	512000	614400	716800	921600	1024000
exhaustive	8	8	16	20	28	34	40	48	60	68
predicted	8	7	14	20	27	34	40	47	60	67

ESTIMATING A

We make an estimation to a as follows:

$$\frac{S_{data}(R)/BW}{L} \times 100\% \leq \delta \tag{B.10}$$

where the variables are the same with those in Equation B.2, and δ is an empirical threshold. S_{data} can be calculated as follows:

$$S_{data}(R) = \frac{4 \times (K+1) \times WI_{app}}{R} \tag{B.11}$$

where K represents the number of centers, WI_{app} represents the number of work-items, and each element consumes 4 Bytes. In order to estimate a, L is also needed. L is mainly determined by startup time (l) of buffer-related commands, which can be measured by a synthetic benchmark developed by ourselves. Then we can calculate L as follows:

$$L = l \times n \tag{B.12}$$

where n is the total number of invocations of buffer-related commands (i.e. the device-to-host buffer commands) in a certain experiment.

Now we can calculate R using Equations B.10, B.11, and B.12:

$$R \geq \frac{4 \times (K+1) \times WI_{app}}{\delta \times l \times n \times BW} \tag{B.13}$$

and from $a = min(R)$ we can derive a.

EMPIRICAL SEARCH

When $R \in [b, a]$, we can use a simple search to obtain the optimal R. Essentially, this means that we use an exhaustive method to find the optimal R along the interval $[b, a]$.

Let us take the problem size of 51200 as an example. The parameters are measured/listed: $K = 20$, $WI_{app} = 51200$, $\delta = 1\%$, $l = 4.5 \times 10^{-5} s$, $n = 895$, and $BW = BW_{PCIe}/6 = 1GB/s$ (note that when processing the case when $a > b$, we let $BW = BW_{PCIe}/6$ as a good approximation. However, in practice BW is dependent on the amount of data to be transferred). Using Equation B.13, we can derive $R \geq 10.7$ ($a = 11$). We can also calculate $b = 4$. Obviously, $a > b$, and we use the empirical search in the interval $[4, 11]$ to find the optimal $R = 8$.

B.2.3. EXPERIMENTAL RESULTS

We first compare the results of the exhaustive search and our analytical model using ten different problem sizes between 51200 and 1024000. The results are shown in Table B.5. As can be seen from the Table, the predicted R is always very close to that obtained by exhaustive search.

Furthermore, we observe that when R is small the D2H time decreases sharply with the increase of R; after the transition point, it decreases slightly when increasing R, as shown in Figure B.7a. This means that the $a < b$ case is more likely to occur, especially for large problem sizes. In turn, this means that the empirical search will only be used in a few cases; for the rest of the cases, our model-based tuning is able to determine the correct R directly.

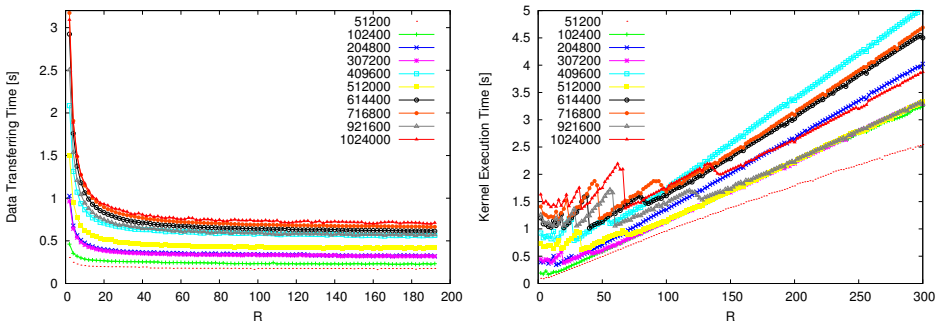
In practice, T_{kernel} tends to present a sawtooth behavior, rather than being stable before point B. This is illustrated in Figure B.7b. These sawteeth appear because of load imbalance (i.e. tasks or work-groups cannot be evenly distributed among multiprocessors). However, this does not conflict with our model. That is because we can always obtain similar kernel execution time at the valley points when there are enough work-groups on each multiprocessor. Before point B, we will have enough work-groups, and thus different selections of valley points will not affect the performance.

B.3. RELATED WORK

In this section, we first present related work in CDS, and then make a short overview of auto-tuning approaches on GPUs.

B.3.1. CLUSTERING DATA STREAMS ON GPUS

Feng Cao et. al proposed a set of algorithms for scalable clustering using graphics processors based on k-means [23]. They introduce two strategies to retrieve data from GPUs, taking into account low bus bandwidth, which has similar motivations to ours. They



(a) (b)

Figure B.7: (a) Experimental curves of showing how the data transferring time changing with R; in the experiments we use 10 different problem sizes. (b) Experimental curves of showing how the kernel execution time changing with R; in the experiments we use 10 different problem sizes.

also extend their GPU-based approach to CDS, but we found too few details in the paper. Moreover, all their implementations use the old graphic terms (i.e. they use shader-programming), rather than the modern programming models.

Shuai Che et. al ported the CDS benchmark in the Parsec Benchmark Suite developed by Princeton University to CUDA and OpenMP [1, 16]. Our work is based on their CUDA implementation found in Rodinia Benchmark Suite [1]. However, we have further adapted the solution to OpenCL, optimized it, and enabled auto-tuning (see Section III and Section IV for more details).

B.3.2. AUTO-TUNING ON GPUS

Automatic performance tuning, or auto-tuning in short, is a technique that has been used intensively to automatically obtain optimal parameters. There are generally two types of approaches for doing auto-tuning: model-driven optimization and empirical optimization [91].

Model-driven optimizations self-tune implementation-related parameters to obtain optimal performance. Parameters such as the block size and the amount of unrolling are determined by analytical models [27, 35]. The model-driven approaches usually work with the help of performance prediction by modelling underlying architectures [10, 64, 85, 178]. In contrast to model-driven optimization, empirical optimization techniques generate a large number of parameterized code variants for a given algorithm and run these variants on a given platform to discover the one that gives the best performance [57, 91, 105, 109].

Model-driven optimizations typically have an $O(1)$ cost, since the parameters can be derived from the analytical model. However, model-driven optimization may not give optimal performance, because analytical models are only simplified abstractions of the underlying processor architectures. In comparison, the time cost of searching for the best code variant, which is usually proportional to the number of variants generated and evaluated, makes empirical optimization less attractive.

As a result, it is interesting to combine these two approaches, and gives a hybrid approach that uses the model-driven approach in the first stage to limit the search space for the second stage of empirical search. Our work applies this hybrid approach: we first build a simple model to determine the optimal parameter (rake-size). When the model can not calibrate the procedure precisely, we also use empirical search. To the best of our knowledge, this work is the first auto-tuned CDS solution for multiple many-core platforms.

B.4. SUMMARY

In this appendix, we have optimized CDS on GPUs. To address the key requirement of CDS- namely, efficient memory usage, we have proposed a rake-based optimization, where R (the number of rakes) is used to boost performance in both memory consumption and execution time. We have found that some optimizations (e.g. pinned memory) are platform/architecture dependent.

In addition, we have developed a simple model to determine the optimal R, addressing the issue that the optimal R is not fixed when problem sizes or architectures are different. Our experimental results show that we correctly identify the optimal R for multiple problem sizes.

BIBLIOGRAPHY

[1] *Rodinia: A benchmark suite for heterogeneous computing*. IEEE, October 2009.

[2] *GPU Computing Gems Emerald Edition (Applications of GPU Computing Series)*. Morgan Kaufmann, 1 edition, February 2011.

[3] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, October 1987.

[4] AMD Inc. AMD Accelerated Parallel Processing (APP) SDK. http://developer.amd.com/gpu/amdappsdk/pages/default.aspx, February 2011.

[5] AMD Inc. Porting CUDA Applications to OpenCL. http://developer.amd.com/zones/OpenCLZone/programming/pages/portingcudatoopencl.aspx, February 2011.

[6] AMD Inc. *AMD Accelerated Parallel Processing - OpenCL*, May 2012.

[7] Ronan Amorim, Gundolf Haase, Manfred Liebmann, and Rodrigo Weber dos Santos. Comparing CUDA and OpenGL implementations for a jacobi iteration. pages 22–32, June 2009.

[8] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: a view from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[10] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen mei. An adaptive performance modeling tool for GPU architectures. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 105–114, New York, NY, USA, 2010. ACM.

[11] Rajeshwari Banakar, Stefan Steinke, Bo-sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *In Tenth International Symposium on Hardware/Software Codesign (CODES), Estes Park*, pages 73–78, 2002.

[12] F. Barahona. On the computational complexity of ising spin glass models. *Journal of Physics A: Mathematical and General*, 15(10):3241+, January 1999.

[13] Muthu M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 1–10, New York, NY, USA, 2008. ACM.

[14] Muthu M. Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[15] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. ACM.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder P. Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[18] OpenMP Architecture Review Board. OpenMP application program interface (version 4.0). Technical report, July 2013.

[19] Michael Boyer, David Tarjan, Scott T. Acton, and Kevin Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, May 2009. IEEE Computer Society.

[20] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(9):1124–1137, September 2004.

[21] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 23(11):1222–1239, November 2001.

[22] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568. IEEE, May 2012.

[23] Feng Cao, Anthony Tung, and Aoying Zhou. Scalable clustering using graphics processors. In Jeffrey Yu, Masaru Kitsuregawa, and Hong Leong, editors, *Advances in Web-Age Information Management*, volume 4016 of *Lecture Notes in Computer Science*, chapter 32, pages 372–384. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.

[24] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM.

[25] Shuai Che, Jeremy Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis(SC'11)*, November 2011.

[26] Gerald Cheong and Monica S. Lam. An optimizer for multimedia instruction sets. In *The Second SUIF Compiler Workshop*. Stanford University, August 1997.

[27] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming(PPoPP '10)*, volume 45 of *PPoPP '10*, pages 115–126, New York, NY, USA, January 2010. ACM.

[28] D. Scharstein and R. Szeliski. Middlebury Stereo Datasets. http://vision.middlebury.edu/stereo/, February 2012.

[29] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, January 1998.

[30] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA, 2010. ACM.

[31] U. Dastgeer, Lu Li, and C. Kessler. The PEPPHER composition tool: Performance-Aware dynamic composition of applications for GPU-based systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 711–720. IEEE, November 2012.

[32] Usman Dastgeer and Christoph Kessler. A performance-portable generic component for 2d convolution computations on gpu-based systems. *Proc. MULTIPROG-2012 Workshop at HiPEAC-2012, Paris*, pages 1–12, 2012.

[33] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, 51(1):129–159, February 2009.

[34] David. *Programming with POSIX Threads*. Addison-Wesley Professional, May 1997.

[35] A. Davidson, Yao Zhang, and J. D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 956–965. IEEE, May 2011.

[36] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Technical report, Department of Computer Science, UTK, Knoxville Tennessee, September 2010.

[37] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES. *Parallel Process. Lett.*, 21(02):173–193, June 2011.

[38] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, volume 39 of *PLDI '04*, pages 82–93, New York, NY, USA, May 2004. ACM.

[39] Johan Enmyren and Christoph W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[40] Jianbin Fang, H. Sips, and A. L. Varbanescu. Quantifying the performance impacts of using local memory for many-core processors. In *Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on*, pages 1–10. IEEE, 2013.

[41] Jianbin Fang, Henk Sips, Pekka Jaaskelainen, and Ana L. Varbanescu. Grover: Looking for performance improvement by disabling local memory usage in OpenCL kernels. In *Proceedings of the 43rd International Conference on Parallel Processing (ICPP'14)*, September 2014.

[42] Jianbin Fang, Henk Sips, and Ana L. Varbanescu. Aristotle: A performance impact indicator for the OpenCL kernels using local memory. *Scientific Programming*, 22(Number 3 / 2014):239–257, June 2014.

[43] Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che, and Ana L. Varbanescu. Test-driving intel xeon phi. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 137–148, New York, NY, USA, 2014. ACM.

[44] Jianbin Fang, A. L. Varbanescu, Jie Shen, and H. Sips. ELMO: A User-Friendly API to enable local memory in OpenCL kernels. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 375–383. IEEE, February 2013.

[45] Jianbin Fang, A. L. Varbanescu, Jie Shen, H. Sips, G. Saygili, and L. van der Maaten. Accelerating cost aggregation for Real-Time stereo matching. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 472–481. IEEE, December 2012.

[46] Jianbin Fang, A. L. Varbanescu, and H. Sips. Sesame: A User-Transparent optimizing framework for Many-Core processors. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 70–73. IEEE, May 2013.

[47] Jianbin Fang, Ana L. Varbanescu, Xiangke Liao, and Henk Sips. Evaluating vector data type usage in OpenCL kernels. *Concurrency and Computation: Practice and Experience*, (accepted).

[48] Jianbin Fang, Ana L. Varbanescu, and Henk Sips. An auto-tuning solution to data streams clustering in OpenCL. In *2011 14th IEEE International Conference on Computational Science and Engineering*, volume 0, pages 587–594, Los Alamitos, CA, USA, August 2011. IEEE.

[49] Jianbin Fang, Ana L. Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing (ICPP'11)*, pages 216–225. IEEE, September 2011.

[50] Jianbin Fang, Ana L. Varbanescu, Henk Sips, LiLun Zhang, Yonggang Che, and Chuanfu Xu. Benchmarking intel xeon phi to guide kernel design. Technical Report PDS-2013-005, Delft University of Technology, April 2013.

[51] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient belief propagation for early vision. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1, pages I–261–I–268 Vol.1. IEEE, June 2004.

[52] Agner Fog. Lists of instruction latencies, throughputs and micro-operation reakdowns for intel, AMD and VIA CPUs. Technical report, Copenhagen University, February 2012.

[53] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28:13–27, July 2008.

[54] Minglun Gong, Ruigang Yang, Liang Wang, and Mingwei Gong. A performance study on different cost aggregation approaches used in Real-Time stereo matching. 75(2):283–296, 2007.

[55] Torbjorn Granlund. Instruction latencies and throughput for AMD and intel x86 processors. Technical report, KTH, February 2012.

[56] Scott G. Gray and John Cavazos. Optimizing and auto-tuning belief propagation on the GPU. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 121–135, Berlin, Heidelberg, 2011. Springer-Verlag.

[57] Dominik Grewe and Anton Lokhmotov. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units(GPGPU2011)*, GPGPU-4, New York, NY, USA, March 2011. ACM.

[58] Armin Größlinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In Oege Moor and MichaelI Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, chapter 17, pages 236–250. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[59] "The Khronos Group". "spir: The standard portable intermediate representation for device programs". http://www.khronos.org/spir, January 2013.

[60] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):515–528, May 2003.

[61] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[62] John Hennessy, John L. Hennessy, David Goldberg, and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1st edition.

[63] Manuel Hohenauer, Felix Engel, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A SIMD optimization framework for retargetable compilers. *ACM Trans. Archit. Code Optim.*, 6(1):1–27, April 2009.

[64] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 152–163, New York, NY, USA, 2009. ACM.

[65] Intel. *Intel Xeon Phi Coprocessor InstructionSet Architecture Reference Manual*, September 2012.

[66] Intel. *Intel Xeon Phi Coprocessor System Software Development Guide*, November 2012.

[67] Intel. *An Overview of Programming for IntelXeon processorsand Intel Xeon Phi coprocessors*, October 2012.

[68] Intel. Intel Xeon Phi Coprocessor. http://software.intel.com/mic-developer, April 2013.

[69] Intel Inc. Intel OpenCL Implicit Vectorization Module.

[70] Intel Inc. *Intel OpenCL Optimization Guide*, April 2012.

[71] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Trans. Des. Autom. Electron. Syst.*, 12(2), April 2007.

[72] Byunghyun Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in Data-Parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, January 2011.

[73] Jared Hoberock and Nathan Bell. Thrust. http://thrust.github.io/, May 2009.

[74] H. Jin and R. F. Van der Wijngaart. Performance characteristics of the multi-zone NAS parallel benchmarks. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 6+. IEEE, April 2004.

[75] John D. McCalpin. STREAM: Sustainable Memory Bandwidth With High Performance Computers, April 2013.

[76] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Proceedings of the 39th Annual Design Automation Conference*, DAC '02, pages 628–633, New York, NY, USA, 2002. ACM.

[77] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. May 2010.

[78] Karl Rupp. CPU, GPU and MIC Hardware Characteristics over Time. http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/, May 2014.

[79] Ralf Karrenberg and Sebastian Hack. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 141–150, Washington, DC, USA, 2011. IEEE Computer Society.

[80] Khronos Group. OpenGL Shading Language. http://www.opengl.org/documentation/glsl/, February 2011.

[81] Khronos Group. SYCL. https://www.khronos.org/opencl/sycl, May 2014.

[82] David Kirk and Wen-Mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, February 2010.

[83] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of opencl programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[84] Athanasios Konstantinidis, Paul H. J. Kelly, J. Ramanujam, and P. Sadayappan. Parametric GPU code generation for affine loop programs. In *The 26th International Workshop on Languages and Compilers for Parallel Computing*, September 2013.

[85] Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan, and Kannan Srinathan. A performance prediction model for the CUDA GPGPU platform. In *Proceedings of 2009 International Conference on High Performance Computing (HiPC)*, pages 463–472, December 2009.

[86] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI'00)*, volume 35 of *PLDI '00*, pages 145–156, New York, NY, USA, May 2000. ACM.

[87] Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 18–29, Washington, DC, USA, 2002. IEEE Computer Society.

[88] Joo H. Lee, K. Patel, N. Nigania, Hyojong Kim, and Hyesoon Kim. OpenCL performance evaluation on modern multi core CPUs. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1177–1185. IEEE, May 2013.

[89] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[90] Shun-tak Leung and John Zahorjan. Optimizing data locality by array restructuring. Technical Report TR 95-09-01, University of Washington, 1995.

[91] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, volume 5544 of *ICCS '09*, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[92] Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. ACM.

[93] Alireza R. Mahdiraji. Clustering data stream: A survey of algorithms. *International Journal of Knowledge-Based and Intelligent Engineering Systems*, 13(2):39–44, January 2009.

[94] S. Maleki, Yaoqing Gao, M. J. Garzaran, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382. IEEE, October 2011.

[95] Andrea Marongiu and Luca Benini. An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs. *IEEE Trans. Comput.*, 61(2):222–236, February 2012.

[96] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, page 23, Berkeley, CA, USA, 1996. USENIX Association.

[97] Xing Mei, Xun Sun, Mingcai Zhou, Shaohui Jiao, Haitao Wang, and Xiaopeng Zhang. On building an accurate stereo matching system on graphics hardware. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 467–474. IEEE, November 2011.

[98] S. Meister, B. Jähne, and D. Kondermann. An outdoor stereo camera system for the generation of Real-World benchmark datasets. *Optical Engineering*, 2011.

[99] Microsoft. AMP C++. http://msdn.microsoft.com/en-us/library/hh265136.aspx, May 2014.

[100] Microsoft. DirectCompute. https://developer.nvidia.com/directcompute/, May 2014.

[101] Microsoft Inc. Reference for HLSL. http://msdn.microsoft.com/en-us/library/bb509635(v=VS.85).aspx, February 2011.

[102] Dongbo Min, Jiangbo Lu, and M. N. Do. A revisit to cost aggregation in stereo matching: How far can we reduce its computational redundancy? In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1567–1574. IEEE, November 2011.

[103] Maryam Moazeni, Alex Bui, and Majid Sarrafzadeh. A memory optimization technique for software-managed scratchpad memory in GPUs. In *2009 IEEE 7th Symposium on Application Specific Processors*, pages 43–49. IEEE, July 2009.

[104] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *18th International Conference on Parallel Architectures and Compilation Techniques, 2009. PACT '09.*, pages 261–270. IEEE, September 2009.

[105] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse Matrix-Vector multiplication for GPU architectures high performance embedded architectures and compilers. In Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *Proceedings of the 5th International Conferences on High Performance Embedded Architectures and Compilers(HiPEAC 2010)*, volume 5952 of *Lecture Notes in Computer Science*, pages 111–125, Berlin, Heidelberg, 2010. Springer Berlin / Heidelberg.

[106] Giridhar S. Murthy, Mahesh Ravishankar, Muthu M. Baskaran, and P. Sadayappan. Optimal loop unrolling for GPGPU programs. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11. IEEE, April 2010.

[107] John Nickolls and William J. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56–69, March 2010.

[108] Cedric Nugteren, Pieter Custers, and Henk Corporaal. Algorithmic species: A classification of affine loop nests for parallel programming. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[109] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, November 2009. ACM.

[110] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *International Symposium on Code Generation and Optimization (CGO 2006)*, CGO '06, pages 11 pp.–294, Washington, DC, USA, March 2006. IEEE.

[111] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 151–160, Washington, DC, USA, 2011. IEEE Computer Society.

[112] NVIDIA Inc. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009.

[113] NVIDIA Inc. *OpenCL Best Practices Guide*, May 2010.

[114] NVIDIA Inc. *PTX: Parallel Thread Execution ISA Version 2.2*, October 2010.

[115] NVIDIA Inc. CUDA Toolkit 3.2. http://developer.nvidia.com/object/cuda_3_2_downloads.html, February 2011.

[116] NVIDIA Inc. NVIDIA Cg Toolkit. http://developer.nvidia.com/page/cg_main.html, February 2011.

[117] NVIDIA Inc. *NVIDIA CUDA C Programming Guide Version 4.1*, 2011.

[118] NVIDIA Inc. *NVIDIA OpenCL C Programming Guide*, June 2011.

[119] OpenACC Group. OpenACC. http://www.openacc-standard.org/, May 2014.

[120] OpenACC Members. *The OpenACC Application Programming Interface V1.0*, November 2011.

[121] OpenMP Group. OpenMP. http://openmp.org/wp/, May 2014.

[122] S. Arash Ostadzadeh, Roel J. Meeuws, Carlo Galuzzi, and Koen Bertels. QUAD: a memory access pattern analyser. In *Proceedings of the 6th international conference on Reconfigurable Computing: architectures, Tools and Applications*, ARC'10, pages 269–281, Berlin, Heidelberg, 2010. Springer-Verlag.

[123] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of General-Purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[124] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[125] Deepak M. Panickal. Exploring the optimization space of Multi-Core architectures with OpenCL benchmarks. Master's thesis, University of Edinburgh, 2011.

[126] Lu Peng, Jih-Kwon Peir, Tribuvan K. Prakash, Carl Staelin, Yen-Kuang Chen, and David Koppelman. Memory hierarchy performance measurement of commercial dual-core desktop processors. *Journal of Systems Architecture*, 54(8):816–828, August 2008.

[127] Peng Wu. The myth of auto-SIMD. http://pengwu.wordpress.com/2014/01/02/the-myth-of-auto-simd, 2014.

[128] Louis N. Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[129] Sabela Ramos and Torsten Hoefler. Modeling communication in cache-coherent SMP systems: A case-study with xeon phi. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 97–108, New York, NY, USA, 2013. ACM.

[130] N. Ray and S. T. Acton. Motion gradient vector flow: an external force for tracking rolling leukocytes with shape and size constrained active contours. *Medical Imaging, IEEE Transactions on*, 23(12):1466–1478, December 2004.

[131] G. Ren, Peng Wu, and D. Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, page 89b. IEEE, April 2005.

[132] Manman Ren, Ji Y. Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed memory hierarchies. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 280–291, New York, NY, USA, 2008. ACM.

[133] Christian Richardt, Douglas Orr, Ian Davies, Antonio Criminisi, and NeilA Dodgson. Real-Time spatiotemporal stereo matching using the Dual-Cross-bilateral grid. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, volume 6313 of *Lecture Notes in Computer Science*, pages 510–523. Springer Berlin Heidelberg, 2010.

[134] Selma Saidi, Pranav Tendulkar, Thierry Lepley, and Oded Maler. Optimizing explicit data transfers for data parallel applications on the cell architecture. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012.

[135] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[136] D. Scharstein, R. Szeliski, and R. Zabih. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Stereo and Multi-Baseline Vision, 2001. (SMBV 2001). Proceedings. IEEE Workshop on*, pages 131–140. IEEE, 2001.

[137] Alessio Sclocco, Ana L. Varbanescu, Jan D. Mol, and Rob van Nieuwpoort. Radio astronomy beam forming on Many-Core architectures. In *IPDPS*, pages 1105–1116. IEEE Computer Society, 2012.

[138] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Graphics Hardware 2007*. ACM, August 2007.

[139] Sangmin Seo, Jun Lee, Gangwon Jo, and Jaejin Lee. Automatic OpenCL work-group size selection for multicore CPUs. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 387–398, Piscataway, NJ, USA, 2013. IEEE Press.

[140] Jie Shen, Jianbin Fang, H. Sips, and A. L. Varbanescu. Performance gaps between OpenMP and OpenCL for multi-core CPUs. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 116–125. IEEE, 2012.

[141] Jie Shen, Jianbin Fang, H. Sips, and A. L. Varbanescu. Performance traps in OpenCL for CPUs. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 38–45. IEEE, February 2013.

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

[143] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-Level parallelism in the presence of control flow. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 165–175, Washington, DC, USA, 2005. IEEE Computer Society.

[144] Alan J. Smith and Rafael H. Saavedra. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Trans. Comput.*, 44(10):1223–1235, October 1995.

[145] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28(4):363–400, August 2000.

[146] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - a portable skeleton library for High-Level GPU programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182. IEEE, May 2011.

[147] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, May 2010.

[148] John A. Stratton, Sam S. Stone, and Wen Mei. Languages and compilers for parallel computing. In José N. Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, chapter MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pages 16–30. Springer-Verlag, Berlin, Heidelberg, 2008.

[149] Andrew S. Tanenbaum. *Modern Operating Systems (3rd Edition)*. Prentice Hall, 3 edition, December 2007.

[150] Technical University of Dresden. BenchIT: Performance Measurement for Scientific Applications, August 2013.

[151] The Khronos OpenCL Working Group. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, February 2011.

[152] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic OpenCL device characterization: Guiding optimized kernel design. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 438–452. Springer Berlin Heidelberg, 2011.

[153] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, and Satoshi Miki. *The OpenCL Programming Book*. Fixstars Corporation, March 2010.

[154] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, May 2006.

[155] Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 214–224, New York, NY, USA, 2011. ACM.

[156] András Vajda. *Programming Many-Core Chips*. Springer, 2011 edition, June 2011.

[157] Rob van Nieuwpoort and John Romein. Correlating radio astronomy signals with Many-Core hardware. *International Journal of Parallel Programming*, 39(1):88–114, February 2011.

[158] Ben Van Werkhoven, Jason Maassen, Henri E. Bal, and Frank J. Seinstra. Optimizing convolution operations on GPUs using adaptive tiling. *Future Gener. Comput. Syst.*, 30:14–26, January 2014.

[159] Ana L. Varbanescu, Alexander S. van Amesfoort, Tim Cornwell, Ger van Diepen, Rob van Nieuwpoort, Bruce G. Elmegreen, and Henk J. Sips. Building high-resolution sky images using the Cell/B.e. *Scientific Programming*, 17(1-2):113–134, 2009.

[160] Nicolas Vasilache, Muthu Baskaran, Benoit Meister, and Richard Lethin. Memory reuse optimizations in the R-Stream compiler. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 42–53, New York, NY, USA, 2013. ACM.

[161] Tzvetomir I. Vassilev. Comparison of several parallel API for cloth modelling on modern GPUs. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, CompSysTech '10, pages 131–136, New York, NY, USA, 2010. ACM.

[162] Sven Verdoolaege, Juan C. Juega, Albert Cohen, José I. Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[163] Vasily Volkov. Use registers and multiple outputs per thread on GPU. International Workshop on Parallel Matrix Algorithms and Applications, July 2010.

[164] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[165] Liang Wang, Miao Liao, Minglun Gong, Ruigang Yang, and David Nister. High-Quality Real-Time stereo using adaptive cost aggregation and dynamic programming. In *Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, 3DPVT '06, pages 798–805, Washington, DC, USA, 2006. IEEE Computer Society.

[166] Rick Weber, Akila Gothandaraman, Robert J. Hinde, and Gregory D. Peterson. Comparing hardware accelerators in scientific applications: A case study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, January 2011.

[167] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[168] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26(6):30–44, May 1991.

[169] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. IEEE, March 2010.

[170] Peng Wu, A. E. Eichenberger, and A. Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 153–164. IEEE, March 2005.

[171] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An integrated simdization framework using virtual vectors. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 169–178, New York, NY, USA, 2005. ACM.

[172] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM.

[173] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. *SIGPLAN Not.*, 45(6):86–97, June 2010.

[174] Kuk-Jin Yoon and In-So Kweon. Locally adaptive support-weight approach for visual correspondence search. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 924–931 vol. 2. IEEE, June 2005.

[175] Wei Yu, Tsuhan Chen, F. Franchetti, and J. C. Hoe. High performance stereo vision designed for massively data parallel platforms. *IEEE Transactions on Circuits and Systems for Video Technology*, 20(11):1509–1519, November 2010.

[176] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 712–721, New York, NY, USA, 1991. ACM.

[177] Ke Zhang, Jiangbo Lu, and G. Lafruit. Cross-Based local stereo matching using orthogonal integral images. *Circuits and Systems for Video Technology, IEEE Transactions on*, 19(7):1073–1079, July 2009.

[178] Yao Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. IEEE, February 2011.

SUMMARY

The architecture diversity of many-core processors - with their different types of cores, and memory hierarchies - makes the old model of reprogramming every application for every platform infeasible. Therefore, inter-platform portability has become a desirable feature of programming models. While functional portability is ensured by standards and compilers (e.g., OpenCL), to achieve high performance across platforms remains a much more challenging task.

In this thesis, we have investigated the enabling/disabling techniques for platform-specific optimizations with a unified programming model. We have selected OpenCL as our research vehicle, and identified that each platform has a specific optimization space for a given kernel. Taking two concrete examples, we have proposed solutions on how to (semi-) automatically tackle platform-specific optimizations with a unified programming model.

We use a case study (in computer vision) to illustrate optimization's dependency on platform. To deal with the difference in processing cores, we propose two approaches to vectorize scalar kernels (i.e., explicitly using vector data types), and reveal the vectorization needs with explicitly parallel programs. To deal with the difference in memory hierarchy, we first present a method to quantify the performance impact of using local memory starting from the memory access patterns. This work produces a performance database, which serves as an indicator of whether using local memory is beneficial. Once this indication is given, we propose a portable solution to simplify programming with local memory. Specifically, we present an easy-to-use API (ELMO) to enable local memory usage and a compiling pass (Grover) to automatically disable the local memory usage for applications where local memory is natively used.

Much like vectorization and local memory usage, other architectural features require performance portable approaches. Therefore, we present our vision for a portable programming framework, called SESAME, which expands to architectural features beyond SIMD units and local memory.

This thesis has given evidence that this problem can be addressed successfully. We conclude that tools such as SESAME help improving the state-of-the-art of existing programming models (like OpenCL, in our case) and ease the task of programmers when dealing with different many-core architectures. This work serves an essential step towards portable performance by systematically exploring the optimization space.

SAMENVATTING

De diversiteit aan architecturen van many-core processoren, zijnde verschillende typen cores en geheugen hiërarchieën, maakt het oude ontwikkelingsmodel van parallelle applicaties waarbij elke applicatie voor elk platform wordt geherprogrammeerd ondoenlijk. Inter-platform portabiliteit, zowel van functionaliteit als performance, is daarom een zeer gewenste eigenschap voor programmeermodellen. Echter, ondanks dat standaarden en compilers (zoals OpenCL) zorgen voor portabiliteit van de functionaliteit, blijft het behalen van goede performance voor alle platformen een zeer uitdagende taak.

In dit proefschrif is onderzoek gedaan naar technieken voor het aan- en uitzetten van platform specifieke optimalisatietechnieken, gebruik makend van een uniform programmeermodel (in dit proefschrift is dat OpenCL). Geconstateerd wordt dat elk platform een specifieke optimalisatie ruimte biedt voor een gegeven applicatie kernel. Met behulp van twee concrete voorbeelden worden oplossingen voorgesteld voor het (semi-)automatisch realiseren van platform specifieke optimalisaties.

Een case study (uit het gebied van de computer vision) wordt gebruikt om de platform afhankelijkheid van optimalisaties te illustreren. Om te kunnen omgaan met de verschillen tussen processing cores, worden twee methodes voorgesteld om scalar kernels te vectoriseren (oftewel expliciet gebruik maken van vector data typen) en wordt de noodzaak aangetoond van het vectoriseren voor expliciet parallelle programma's. Om te kunnen omgaan met de verschillen in geheugen hiërarchie wordt eerst een methode gepresenteerd om de invloed van het gebruik van lokaal geheugen op de prestaties te kwantificeren, beginnend bij geheugen toegangspatronen. Hieruit wordt een prestatie database gegenereerd, die als indicator gebruikt kan worden om te beslissen of het gebruik van lokaal geheugen al dan niet zorgt voor verbetering van de performance. Met behulp van deze indicator stellen we een overdraagbare oplossing voor om het programmeren met lokaal geheugen te vereenvoudigen. Meer specifiek is een gemakkelijk te gebruiken API (ELMO) ontworpen, die het gebruik van lokaal geheugen mogelijk maakt en een compiler toevoeging (Grover) die automatisch het gebruik van lokaal geheugen uitzet voor applicaties waar lokaal geheugen al van nature inzit.

Net als voor vectorisatie en lokaal geheugen gebruik, zijn er andere architecturele eigenschappen die een dergelijke prestatie overdraagbare aanpak vereisen. Daartoe wordt een overdraagbaar programmeer framework, geheten SESAME, gepresenteerd, dat verder gaat dan architecturele eigenschappen als SIMD eenheden en lokaal geheugen.

Dit proefschrift laat zien dat het probleem van performance portabiliteit succesvol aangepakt kan worden. Tools zoals SESAME helpen met het verbeteren van de state-of-the-art van bestaande programmeermodellen (zoals OpenCL in ons geval) en maken de taak van programmeurs makkelijker als zij met verschillende many-core architecturen moeten werken. Dit is een essentiële stap naar het realiseren van overdraagbare prestaties middels een systematische verkenning van de optimalisatie ruimte.

CURRICULUM VITÆ

Jianbin Fang was born in Qingdao, China, October 11th, 1984. He grew up and received his nine-year compulsory education in a small but quiet village in Qingdao. Now he still loves this village and spends a couple of weeks there every year. Upon completing the secondary education, he attended the National College Entrance Examination (a.k.a. Gaokao) and enrolled at the Central South University (CSU) in Changsha, 2003. For the first time, he lives around 2,000 kilometres far away from home. There, he received his BSc in computer science in 2007. Due to his excellent academic performance in CSU (%2), he managed to start his master program in National University of Defense Technology (NUDT) without entrance examination in 2007. There, he focused on simulating computer architectures using parallel approaches, and developed different algorithms to speed-up the simulation process. At the end of 2009, he earned his MSc, and his master thesis, entitled "On optimizing the trace-driven parallel simulations", was selected as Outstanding Thesis of Hunan province. After he was granted a four-year funding by China Scholarship Council (CSC) in 2010, he joined Parallel and Distributed Systems group at Delft University of Technology to pursue a PhD in computer science. His PhD track was focused on parallel programming on multi-/many-cores. During his PhD, he gained experience on the Tianhe-2 supercomputer through two internships: one was in NUDT, Changsha, China, and the other was with Guangzhou Supercompuer Center, Guangzhou, China. He also worked as TA for IN4049 (Introduction to High Performance Computing) at TU Delft.

List of Publications

- **[J]** J. Fang, A. L. Varbanescu, X. Liao, and H. Sips, "Evaluating vector data type usage in OpenCL kernels," Concurrency and Computation: Practice and Experience (accepted).
- **[J]** J. Fang, H. Sips, and A. L. Varbanescu, "Aristotle: A Performance Impact Indicator for the OpenCL Kernels Using Local Memory". Scientific Programming 22:239-257.
- **[C]** J. Fang, H. Sips, P. Jaaskelainen, and A. L. Varbanescu, "Grover: Looking for performance improvement by disabling local memory usage in OpenCL kernels," in Proceedings of the 43rd International Conference on Parallel Processing (ICPP'14), Minneapolis, USA.
- **[C]** J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving intel xeon phi," in Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ser. ICPE '14. New York, NY, USA: ACM, 2014, pp. 137-148. (**Best Paper Award in Industry/Experience Track**)
- **[C]** J. Fang, A. L. Varbanescu, B. Imbernon, J. M. Cecilia, and H. Perez-Sanchez, "Parallel computation of Non-Bonded Interactions in drug discovery: Nvidia GPUs vs. intel xeon phi," in Proceedings of the 2nd International Work-Conference on Bioinformatics and Biomedical Engineering (IWBBIO'2014), Apr. 2014, pp. 579-588.

- **[C]** J. Fang, H. Sips, and A. L. Varbanescu, "Quantifying the performance impacts of using local memory for many-core processors," in Multi-/Many-core Computing Systems (MuCoCoS), 2013 IEEE 6th International Workshop on. IEEE, 2013, pp. 1-10.
- **[C]** J. Fang, A. L. Varbanescu, and H. Sips, "Sesame: A User-Transparent optimizing framework for Many-Core processors," in Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on. IEEE, May 2013, pp. 70-73.
- **[C]** J. Fang, A. L. Varbanescu, J. Shen, and H. Sips, "ELMO: A User-Friendly API to enable local memory in OpenCL kernels," in Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. IEEE, Feb. 2013, pp. 375-383.
- **[C]** J. Fang, A. L. Varbanescu, J. Shen, H. Sips, G. Saygili, and L. van der Maaten, "Accelerating cost aggregation for Real-Time stereo matching," in Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on. IEEE, Dec. 2012, pp. 472-481.
- **[C]** J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in 2011 International Conference on Parallel Processing (ICPP'11). IEEE, Sep. 2011, pp. 216-225.
- **[C]** J. Fang, A. L. Varbanescu, and H. Sips, "An auto-tuning solution to data streams clustering in OpenCL," in 2011 14th IEEE International Conference on Computational Science and Engineering, vol. 0. Los Alamitos, CA, USA: IEEE, Aug. 2011, pp. 587-594.

Co-Authored Publications

- **[J]** C. Xu, X. Deng, L. Zhang, J. Fang, G. Wang, Y. Jiang, W. Cao, Y. Che, Y. Wang, Z. Wang, W. Liu, and X. Cheng, "Collaborating CPU and GPU for large-scale high-order CFD simulations with complex grids on the TianHe-1A supercomputer," Journal of Computational Physics, Aug. 2014.
- **[C]** C. Xu, L. Zhang, X. Deng, J. Fang, G. Wang, W. Cao, Y. Che, Y. Wang, W. Liu, "Balancing CPU-GPU Collaborative High-order CFD Simulations on the TianHe-1A Supercomputer", in Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS'14), PHOENIX (Arizona), USA.
- **[C]** C. Xu, X. Deng, L. Zhang, Y. Jiang, W. Cao, J. Fang, Y. Che, Y. Wang, and W. Liu, "Parallelizing a High-Order CFD software for 3D, multi-block, structural grids on the TianHe-1A supercomputer," in Supercomputing, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer Berlin Heidelberg, 2013, vol. 7905, pp. 26-39.
- **[J]** J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "An application-centric evaluation of OpenCL on multi-core CPUs," Parallel Computing, vol. 39, no. 12, pp. 834-850, Dec. 2013.
- **[C]** J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance traps in OpenCL for CPUs," in Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. IEEE, Feb. 2013, pp. 38-45.
- **[C]** J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance gaps between OpenMP and OpenCL for multi-core CPUs," in Parallel Processing Workshops (ICPPW), 2012 41st International Conference on. IEEE, 2012, pp. 116-125.