Exploring Performance Portability for Accelerators via High-level Parallel Patterns

Kaixi Hou

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

Wu-chun Feng, Chair
Calvin J. Ribbens
Hao Wang
Yong Cao
Gagan Agrawal

May 2, 2018
Blacksburg, Virginia

Keywords: Parallel Patterns, Vector Processing, SIMD, Parallel Accelerators, Multi-core, Many-core, DSL, Algorithmic Skeleton

Exploring Performance Portability for Accelerators via High-level Parallel Patterns

Kaixi Hou

(ABSTRACT)

Parallel computing dates back to the 1950s-1960s. It is only in recent years, as various parallel accelerators have become prominent and ubiquitous, that the pace of parallel computing has sped up and has had a strong impact on the design of software/hardware. The essence of the trend can be attributed to the physical limits of further increasing operating frequency of processors and the shifted focus on integrating more computing units on one chip. Driven by the trend, many commercial parallel accelerators are available and become commonplace in computing systems, including multi-core CPUs, many-core GPUs (Graphics Processing Units) and Intel Xeon Phi.

Compared to a traditional single-core CPU, the performance gains of parallel accelerators can be as high as many orders of magnitude, attracting extensive interest from many scientific domains. Nevertheless, it presents two main difficulties for domain scientists and even expert developers: (1) To fully utilize the underlying computing resources, users have to redesign their applications by using low-level languages (e.g., CUDA, OpenCL) or idiosyncratic intrinsics (e.g., AVX vector instructions). Only after a careful redesign is carried out can efficient kernels be written. (2) These kernels are oftentimes specialized and in turn imply that as far as the performance is concerned, the codes might be not portable to a new given architecture, leading to more tedious and error-prone redesign.

Parallel patterns, therefore, can be a promising solution in lifting the burdens from programmers. The idea of parallel patterns attempts to build a bridge between target algorithms and parallel architectures. There has already been some research on using high-level languages as parallel patterns for programming on modern accelerators, such as ISPC, Lift, etc. However, the existing approaches fall short from efficiently taking advantage of both parallelism of algorithms and diversified features of accelerators. For example, by using such high-level languages, it is still challenging to describe the patterns of the convoluted data-reordering operations in sorting net-

works, which are oftentimes treated as building blocks of parallel sort kernels. On the other hand, compilers or frameworks lack the capability and flexibility to quickly and efficiently re-target the parallel patterns to another different architectures (e.g., what types of vector instructions to use, how to efficiently organize the instructions, etc.).

To overcome the aforementioned limitations, we propose a general approach that can create an effective and abstracted layer to ease the generating efficient parallel codes for given algorithms and architectures. *From algorithms to parallel patterns*, we exploit the domain expertise to analyze the computational and communication patterns in the core computations and represent them in DSL (Domain Specific Language) or algorithmic skeletons. This preserves the essential information, such as data dependencies, types, etc., for subsequent parallelization and optimization. *From parallel patterns to actual codes*, we use a series of automation frameworks and transformations to determine which levels of parallelism can be used, what optimal instruction sequences are, how the implementation changes to match different architectures, etc. In this dissertation, we present our approaches by investigating a couple of important computational kernels, including sort (and segmented sort), sequence alignment, stencils, etc., across parallel platforms (CPUs, GPUs, Intel Xeon Phi). Through these studies, we show: (1) Our automation frameworks use DSL or algorithmic skeletons to express parallelism. (2) Generated parallel programs take into consideration of both parallelism of the target problems and characteristics of underlying hardware. (3) Evaluations are discussed on the portable performance and speedups over the state-of-the-art optimized codes. Besides, for some problems, we also propose novel adaptive algorithms to accommodate the parallel patterns to varying input scenarios.

Exploring Performance Portability for Accelerators via High-level Parallel Patterns

Kaixi Hou

(GENERAL AUDIENCE ABSTRACT)

Nowadays, parallel accelerators have become prominent and ubiquitous, e.g., multi-core CPUs, many-core GPUs (Graphics Processing Units) and Intel Xeon Phi. The performance gains from them can be as high as many orders of magnitude, attracting extensive interest from many scientific domains. However, the gains are closely followed by two main problems: (1) A complete redesign of existing codes might be required if a new parallel platform is used, leading to a nightmare for developers. (2) Parallel codes that execute efficiently on one platform might be either inefficient or even non-executable for another platform, causing portability issues.

To handle these problems, in this dissertation, we propose a general approach using **parallel patterns**, an effective and abstracted layer to ease the generating efficient parallel codes for given algorithms and across architectures. *From algorithms to parallel patterns*, we exploit the domain expertise to analyze the computational and communication patterns in the core computations and represent them in DSL (Domain Specific Language) or algorithmic skeletons. This preserves the essential information, such as data dependencies, types, etc., for subsequent parallelization and optimization. *From parallel patterns to actual codes*, we use a series of automation frameworks and transformations to determine which levels of parallelism can be used, what optimal instruction sequences are, how the implementation change to match different architectures, etc. Experiments show that our approaches by investigating a couple of important computational kernels, including sort (and segmented sort), sequence alignment, stencils, etc., across various parallel platforms (CPUs, GPUs, Intel Xeon Phi).

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor Prof. Wu-chun Feng, who provides me invaluable knowledge and insights on research, communication skills, and introduces me into the world of academia. I have learned so much from Dr. Feng: how to read/write research papers, how to express my idea briefly and deliver a good presentation, how to establish productive collaboration with others, etc. Without Wu's guidance and support, I would have never reach where I am now.

Next, I would like to thank my other committee members: Prof. Calvin J. Ribbens, Dr. Hao Wang, Dr. Yong Cao, and Prof. Gagan Agrawal. I appreciate all their insightful comments and feedback on my dissertation, as well as their precious time they put into serving at various stages of my Ph.D. journey. Special thanks go to Dr. Hao Wang for working closely with me on many research projects and providing me with great suggestion and helpful advise on strengthening my research capability.

I have been fortunate to work with many researchers from other research institutes. I am grateful to Shuai Che (AMD research) for his guidance on the GPU project to accelerate graph algorithms. My thanks also go to Jonathan Gallmeier (AMD research) for the technical discussions on how to efficiently utilize different GPU platforms. I am indebted to Seyong Lee, Jeffrey Vetter, and Mehmet Belviranli (Oak Ridge National Lab), for the extended brainstorming discussions on GPU-based wavefront problems. I would also like to thank Weifeng Liu (University of Copenhagen) for sharing his knowledge and experience on GPU computing.

I also want to share my gratitude to the wonderful people around me throughout these years. I am so fortunate to have Jing Zhang, Xiaodong Yu, Xuewen Cui, Da Zhang as my lab colleagues during my Ph.D. I cannot forget the many "pizza" nights that we spent on the submission deadlines. Special thanks go to my roommates, Fang Liu, Yao Zhang, and Jinxing Wang/Libei Huang couple, at different stages of my Ph.D. I would like to thank the following people who make my life colorful and entertaining in this "isolated" Blacksburg: Bo Li/Yao Wang couple, Hao Zhang, Yue Cheng, Sichao Wu, Liangzhe Chen, Qianzhou Du, Xiaokui Shu, Ji Wang, Run Yu, Mengsu Chen. My best wishes to all of you for the future endeavors.

Finally, I am immensely grateful to my family back in China: my parents, Jianjun Hou and

Lu Zhao. Without their support, understanding, and unconditional love, I cannot go through my ups and downs and finish my Ph.D. A last big thank you to my wife, Lingjie Zhang, for her love, support, for bringing enthusiasm in my life, and for being patient and understanding.

Funding Acknowledgment My research was supported by National Science Foundation (NSF-BIGDATA IIS-1247693, NSF-XPS CCF1337131), Air Force Office of Scientific Research (AFOSR) Basic Research Initiative (Grant No. FA9550-12-1-0442), the European Unions Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No. 752321, Advanced Research Computing (ARC) and Computer Science (CS) Department at Virginia Tech, Advanced Micro Devices (AMD), and Oak Ridge National Lab (ORNL). Many thanks to all for making this dissertation possible.

Declaration of Collaboration In addition to my advisor Wu-chun Feng, this dissertation has benefited from many collaborators, especially:

- Hao Wang contributed to the work included in Chapter 3, Chapter 4, Chapter 5, Chapter 6, and Chapter 7 of the dissertation.
- Weifeng Liu (Norwegian University of Science and Technology) contributed to both design and evaluation parts of the work in Chapter 4.
- Seyong Lee and Jeffrey Vetter (ORNL) contributed to the work in Chapter 6.
- Jonathan Gallmeier and Shuai Che (AMD) contributed to the overall idea of the work in Chapter 7.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

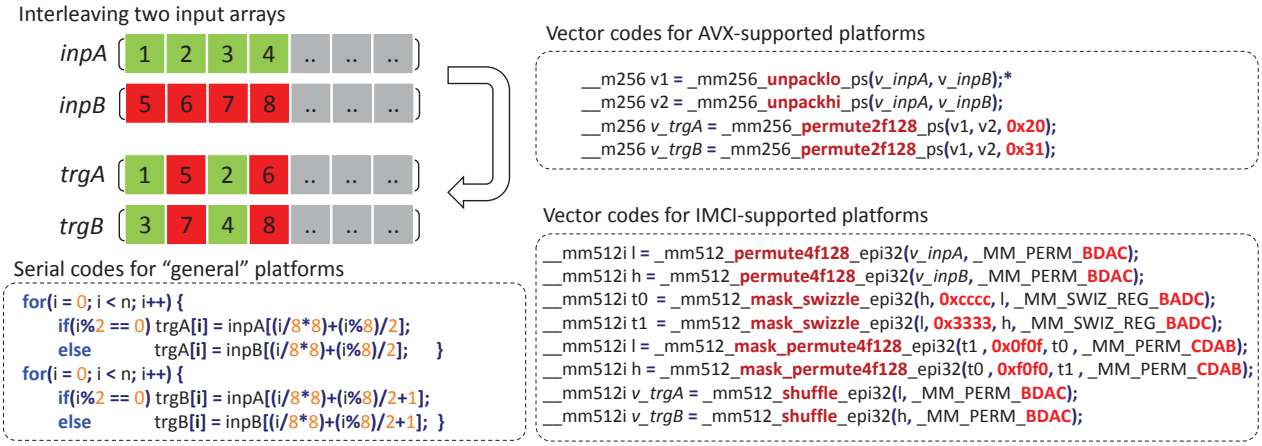
In the chapter, we motivate our work on exploring parallel patterns for the performance portability, outline the important challenges encountered by current studies, and present the contributions of this dissertation.

1.1 Motivation

Over the past decade, the Moore's 2-year performance doubling has been due to the replacement of increasing clock speed of single processor with integrating many efficient cores on one chip. Since we have hit the power limit, the ever-increasing performance is no longer limited to strenuously utilizing a faster single core, but becomes a style known as *parallel computing* [18]. Perhaps nowhere is more evident than with the ubiquitous commercial parallel processors all around us, from mobile processors in our smart phones, to multi-cores in laptops, to gaming GPUs in desktops, to various accelerators (e.g., GPUs, Intel Xeon Phi, etc.) in computing servers. The evolution of parallel computing has sped up and been prominent in high performance computing community and many scientific domains.

The parallel computing power of many orders of magnitude higher than the single core CPU attracts more scientists to port their old software to parallel one. This trend initiates the shift from traditional sequential programming model towards a variety of parallel models. Unfortunately,

the exploitation of the parallel models can be accompanied by many challenges, which oftentimes presents extreme difficulties, even for utilizing the most common multi-core CPUs, and even for the expert programmers. These obstacles include two main aspects: (1) Designing efficient kernels demands great attention and knowledge from programmers in low-level languages, including CUDA or OpenCL for parallelization in GPU platforms, and SSE/AVX intrinsics for vectorization on x86 platforms. It therefore behooves programmers to understand the details of underlying hardware. (2) The carefully designed codes might be dedicated to one platform, causing performance portability issues: the same kernels would fail to achieve high performance when platforms change. If this happens, a tedious re-design will be inevitable. Therefore, in fact, the enthusiasm of using accelerators will be plagued by the costly trade-offs between programming effort and the resultant performance.



(a) Serial codes. (b) Platform-specific vector codes.

Figure 1.1: Serial and vector codes to interleave two given input arrays. *: In vector codes, the vector load and store instructions are ignored for brevity.

The most straightforward solution for these challenges might be to simply rely on the guidance directives (e.g., OpenMP) and compiler options (e.g., -O3) to automatically parallelize the loops or functions that could be executed in parallel [74]. However, the compilers are oftentimes too conservative to parallelize target codes, due to the lack of accurate compiler analysis and effective compiler transformations [108]. In Figure 1.1a, we present the serial codes for instance to interleave two given input arrays; certainly, the codes are parallelizable. Nevertheless, the modern

compilers (e.g., ICC, GCC, and PGCC) cannot effectively parallelize the codes, even if different directives and compiler options are used. That way, programmers have to tweak, transform, or even rewrite the codes to fulfill the architectural properties [135, 19, 74, 176]. As shown in Figure 1.1b, we show the platform-specific vector codes for the same purpose with AVX and IMCI instruction sets respectively. Apparently, both the instructions and associative parameters are idiosyncratic, causing not only the programming burden but also portability issues.

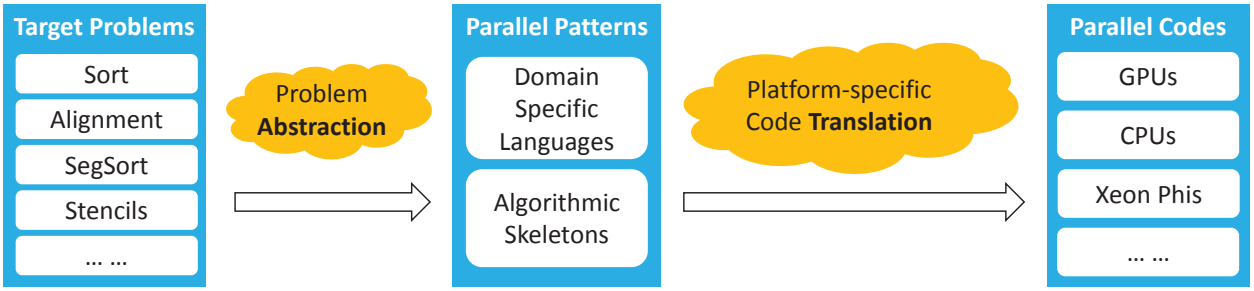


Figure 1.2: Exploring the parallelism and performance-portability via parallel patterns.

In this dissertation, we focus on an alternative solution by taking advantage of **parallel patterns**, i.e., *domain specific languages* (DSLs) and *algorithmic skeletons* [50]. In particular, as shown in Figure 1.2, our approaches start from the *abstraction* of target problems to parallel patterns and end with the *translation* from parallel patterns to real codes. On *abstraction*, we exploit the domain expertise to analyze the computational and communication patterns in the problems and express them in DSLs or algorithmic skeletons. That way, we can preserve the important domain-specific information and create an intermediate layer that can facilitate the subsequent generation of parallel codes. On *translation*, we transform the patterns to efficient parallel codes by considering the characteristics of underlying architectures. We propose, design, and implement a series of automation frameworks and optimizations to determine how different levels of parallelism can be applied, what the optimal instruction sequences are, how to tune generated kernels for the best performance, etc. ***The overarching goal of our approaches is to achieve performance portability across different accelerators without the hassle of programming low level for parallel computing.***

This dissertation selects five important scientific kernels (i.e., sort, segmented sort, alignment,

wavefront, and stencil) as the target research problems and uses three widely-used parallel platforms (i.e., multicore CPUs, manycore Intel Xeon Phi, and GPUs). In the next sections, we briefly describe these research problems and proposed research methodology by using DSLs and algorithmic skeletons respectively.

1.2 Performance Portability via DSLs

1.2.1 Data-reordering in Vectorized Sort

Due to the difficulty that modern compilers have in vectorizing applications on vector-extension architectures, programmers resort to manually programming vector registers with intrinsics in order to achieve better performance. However, the continued growth in the width of registers and the evolving library of intrinsics make such manual optimizations tedious and error-prone. That is, the different architectures, varying vector functionalities, and idiosyncratic instructions and parameters, present a great challenge, from programmers' perspective, for achieving high performance across platforms.

In the first part of this dissertation, we propose a framework for the Automatic SIMDization of Parallel Sorting (ASPaS) on x86-based multicore and manycore processors. ASPaS takes any sorting network and a given instruction set architecture (ISA) as inputs and automatically generates vectorized code for that sorting network. By formalizing the *sort* function as a sequence of comparators and the *transpose* and *merge* functions as sequences of vector-matrix multiplications, ASPaS can map these functions to operations from a selected "pattern pool" that is based on the characteristics of parallel sorting, and then generate platform-specific vector codes. The performance evaluation of our ASPaS framework on the Intel Xeon Phi coprocessor illustrates that automatically generated sorting codes from ASPaS can outperform the sorting implementations from STL, Boost, and Intel TBB, on both CPUs and Xeon Phis.

1.2.2 Data-thread Binding in Parallel Segmented Sort

Segmented sort, as a generalization of classical sort, orders a batch of independent segments in a whole array. Along with the wider adoption of manycore processors for HPC and big data applications, segmented sort plays an increasingly important role than sort. To parallelize the segmented sort on GPUs, directly sorting each segment in parallel could cause severe load imbalance. Even with "dynamic parallelism" techniques, the implementation may cause degraded performance due to high overhead for context switch.

In the second part of this dissertation, we present an adaptive segmented sort mechanism on GPUs. Our mechanisms include two core techniques: (1) a differentiated method for different segment lengths to eliminate the irregularity caused by various workloads and thread divergence; and (2) a register-based sort method to support N-to-M data-thread binding and in-register data communication. We also implement a shared memory-based merge method to support non-uniform length chunk merge via multiple warps. Our segmented sort mechanism shows great improvements over the methods from CUB, CUSP and ModernGPU on NVIDIA K80-Kepler and TitanX-Pascal GPUs. Furthermore, we apply our mechanism on two applications, i.e., suffix array construction and sparse matrix-matrix multiplication, and obtain prominent advantage over state-of-the-art implementations.

1.3 Performance Portability via Algorithmic Skeletons

1.3.1 SIMD Operations in Sequence Alignment

The pairwise sequence alignment algorithms, e.g., Smith-Waterman and Needleman-Wunsch, with adjustable gap penalty systems are widely used in bioinformatics. The strong data dependencies in these algorithms prevent them from exploiting the auto-vectorization techniques in compilers. When programmers manually vectorize them on multi- and manycore processors, two vectorizing strategies are usually considered, both of which initially ignore data dependencies and then appropriately correct in a subsequent stage: (1) *iterate*, which vectorizes and then compensates

the scoring results with multiple rounds of corrections and (2) *scan*, which vectorizes and then corrects the scoring results primarily via one round of parallel scan. However, manually writing such vectorizing code efficiently is non-trivial, even for experts, and the code may not be portable across ISAs. In addition, even highly vectorized and optimized codes may not achieve optimal performance because selecting the best vectorizing strategy depends on the algorithms, configurations (gap systems), and input sequences.

In the third part of this dissertation, we propose a framework AAlign to automatically vectorize pairwise sequence alignment algorithms across ISAs. AAlign ingests a sequential code (which follows our generalized paradigm for pairwise sequence alignment) and automatically generates efficient vector code for iterate and scan. To reap the benefits of both vectorization strategies, we propose a hybrid mechanism where AAlign automatically selects the best vectorizing strategy at runtime no matter which algorithms, configurations, and input sequences are specified. On Intel Haswell and MIC, the generated codes for Smith-Waterman and Needleman-Wunsch achieve up to a 26-fold speedup over their sequential counterparts. Compared to the highly optimized and multi-threaded sequence alignment tools, e.g., SWPS3 and SWAPHI, our codes can deliver up to 2.5-fold and 1.6-fold speedups, respectively.

1.3.2 Data Dependencies in Wavefront Loops

Wavefront loops are widely used in many scientific applications, e.g., partial differential equation (PDE) solvers and sequence alignment tools. However, due to the data dependencies in wavefront loops, it is challenging to fully utilize the abundant compute units of GPUs and to reuse data through their memory hierarchy. Existing solutions can only optimize for these factors to a limited extent. For example, tiling-based methods optimize memory access but may result in load imbalance; while compensation-based methods, which change the original order of computation to expose more parallelism and then compensate for it, suffer from both global synchronization overhead and limited generality.

In the fourth part of this dissertation, we first prove under which circumstances that breaking data dependencies and properly changing the sequence of computation operators in our compensation-

based method does not affect the correctness of results. Based on this analysis, we design a highly efficient compensation-based parallelism on GPUs. Our method provides weighted scan-based GPU kernels to optimize the computation and combines with the tiling method to optimize memory access and synchronization. The performance results on the NVIDIA K80 and P100 GPU platforms demonstrate that our method can achieve significant improvements for four types of real-world application kernels over the state-of-the-art research.

1.3.3 Data Reuse in Stencil Computations

Spatial blocking is a critical memory-access optimization to efficiently exploit the computing resources of parallel processors, such as many-core GPUs. By reusing cache-loaded data over multiple spatial iterations, spatial blocking can significantly lessen the pressure of accessing slow global memory. Stencil computations, for example, can exploit such data reuse via spatial blocking through the memory hierarchy of the GPU to improve performance. However, approaches to take advantage of such blocking require complex and tedious changes to the GPU kernels for different stencils, GPU architectures, and multi-level cached systems.

In the fifth part of this dissertation, we explore the challenges of different spatial blocking strategies over three cache levels of the GPU (i.e., L1 cache, scratchpad memory, and registers) and propose a framework *GPU-UniCache* to automatically generate codes to access buffered data in the cached systems of GPUs. Based on the characteristics of spatial blocking over various stencil kernels, we generalize the patterns of data communication, index conversion, and synchronization (with abstracted ISA-friendly interfaces) and map them to different architectures with highly optimized code variants. Our approach greatly simplifies the design of efficient and portable stencil computations across GPUs. Compared to stencil kernels based on hardware-managed memory (L1 cache) and other state-of-the-art GPU benchmarks, the GPU-UniCache can achieve significant improvements.

1.4 Research Contribution

From the above two aspects, *we demonstrate in this dissertation that we can achieve performance portability for various scientific computational kernels across parallel platforms by using our proposed parallel pattern based approaches*.

We study how to exploit parallel patterns to express the core computation and communication patterns in many scientific kernels. We explore the effectiveness of translating these parallel patterns to real parallel codes. We also investigate how to optimize and tune the codes by considering different inputs, workloads and underlying parallel architectures. In the following, we highlight the specific research contributions that this dissertation makes.

A Framework for Automatic Vectorization of Sorting on CPUs and MICs In this work, we make the following contributions. First, for *portability*, we propose the *ASPaS* framework to automatically generate the cross-platform parallel sorting codes using architecture-specific SIMD instructions, including AVX, AVX2, and IMCI. Second, for *functionality*, using ASPaS, we can generate various parallel sorting codes for the combinations of five sorting networks, two merging networks, and three datatypes (integer, float, double) on Intel Ivy Bridge, Haswell CPUs, and Intel Knights Corner MIC. In addition, ASPaS generates the vectorization codes not only for the sorting of array, but also for the sorting of key,data pairs, which is a requisite functionality to sort the real-world workloads. Third, for *performance*, we conduct a series of rigorous evaluations to demonstrate how the ASPaS-generated codes can yield performance benefits by efficiently using the vector units and computing cores on different hardware architectures.

Fast Segmented Sort on GPUs In this work, the contributions are listed as follows: First, we identify the importance of segmented sort on various applications by exploring segment length distribution in real-world datasets and uncovering performance issues of existing tools. Second, we propose an adaptive segmented sort mechanism for GPUs, whose key techniques contain: (1) a differentiated method for different segment lengths to eliminate load imbalance, thread divergence, and irregular memory access; and (2) an algorithm that

extends sorting networks to support N-to-M data-thread binding and thread communication at GPU register level. Third, we carry out a comprehensive evaluation on both kernel level and application level to demonstrate the efficiency and generality of our mechanism on two NVIDIA GPU platforms.

A SIMD Framework for Pairwise Sequence Alignment on CPUs and MICs In this work, the major contributions include the following. First, we propose the *AAlign* framework that can automatically generate parallel codes for pairwise sequence alignment with combinations of algorithms, vectorizing strategies, and configurations. Second, we identify the existing vectorizing strategies cannot always provide the optimal performance even the codes are highly vectorized and optimized. As a result, we design a hybrid mechanism to take advantages of two vectorizing strategies. Third, using AAlign, we generate various parallel codes for the combinations of algorithms (SW and NW), vectorizing strategies (striped-iterate, striped-scan, and hybrid), and configurations (linear and affine gap penalty systems) on two x86-based platforms, i.e., the Advanced Vector eXtension (AVX2) supported multicore and the Initial Many Core Instructions (IMCI) supported manycore.

Compensation-based Parallelism for Wavefront Loops on GPUs In this work, the key contributions are summarized below. First, we prove that in wavefront loops, if the accumulation operator is associative and commutative and the distribution operator is either distributive over or same with the accumulation operator, breaking through the data dependency and changing the sequence of computation operators properly does not affect the correctness of results. This provides the guidance for developers under which circumstances, the compensation-based method can be used. Second, we design a highly efficient compensation-based method on GPUs. Our method provides the weighted scan-based GPU kernels to optimize the computation, and combines with the tiling method to optimize the memory access and synchronization. Third, we carry out a comprehensive evaluation on both kernel level and application level to demonstrate the efficiency of our method over the state-of-the-art research for wavefront loops.

Automatic Code Generation of Spatial Blocking for Stencils on GPUs In this work, the contributions include the following: (1) *GPU-UniCache*, a framework to automatically generate spatial blocking codes for stencil kernels on GPUs, and (2) a comprehensive evaluation of the GPU-UniCache framework on AMD and NVIDIA GPUs. GPU-UniCache not only improves programming productivity by unifying the interfaces of spatial blocking for different stencils, GPU architectures, and cache levels; but it also provides high performance by optimizing data distribution, indexing conversion, thread communication, and synchronization to facilitate data access in GPU kernels. Compared to hardware-managed memory (L1 cache), with single-precision arithmetic, our automatically-generated codes deliver up to 1.7-fold and 1.8-fold speedups at the scratchpad memory level and register level, respectively, when running on an AMD GCN3 GPU and up to 1.6-fold and 1.8-fold, respectively, when running on a NVIDIA Maxwell GPU. For double precision, it delivers up to a 1.3-fold speedup on both GPU platforms. Compared to the state-of-the-art benchmarks (incl. Parboil, PolyBench, SHOC), it can also provide up to 1.5-fold improvement.

1.5 Dissertation Organization

The remaining dissertation chapters address the following topics. Chapter 2 presents the background information and state-of-the-art related work on the research problems posed in this dissertation. First, Chapter 3 to 4 present DSL-based approaches to achieve performance portability. Chapter 3 presents our proposed framework to generate high-performance sort kernels for different x86-based systems. Chapter 4 introduces an adaptive segmented sort for modern GPUs. Then, Chapter 5 to 7 focus on the approaches using algorithmic skeletons. Chapter 5 presents the vectorized pairwise sequence alignment kernels for high efficiency on CPUs and Intel Xeon Phis. Chapter 6 provides a new compensation-based parallel solution for general wavefront problems on GPUs. Chapter 7 introduces a framework for stencil computations to access reusable data in memory hierarchy on GPUs from different vendors. Finally, Chapter 8 concludes and discusses the future work.

Chapter 2

Background and Related Work

In this chapter, we provide brief background information and state-of-the-art work regarding the research problems of this dissertation. We focus on applying parallel patterns to bridge the gap between complex computational codes and parallelism available in various accelerators. This chapter first introduces the major hardware characteristics that is closely related to this dissertation. Then, we summarize the state-of-the-art research on each research problem. In addition, we compare them with our proposed appoarches and frameworks based on parallel patterns from the perspectives of novelty, efficacy, etc.

2.1 Vector Processing in Modern Parallel Platforms

2.1.1 Vector Processing in x86-based Systems

The vector processing units (VPUs) equipped in the modern x86-based processors are designed to perform one single operation over multiple data items simultaneously. The width of the vectors and richness of the instruction sets are continuously expanding and evolving, forming different generations of vector ISAs, e.g., AVX, AVX2, and IMCI.

AVX/AVX2 on CPUs: The AVX is initially supported by Intels "Sandy Bridge" processors and each of their 16 256-bit registers contains two 128-bit lanes (named as lane B and A in Fig-

ure 2.1), together holding 8 floats or 4 doubles. The three-operands in AVX use a non-destructive form to preserve the two source operands. The AVX2 is available since the "Haswell" processors and it expands the integer instructions in SSE and AVX and supports variable-length integers. Moreover, AVX2 increases the instructional functionalities by adding, for example, gather support to load non-contiguous memory locations and per-element shift instructions. In both AVX and AVX2, the data-reordering operations contain permutation within each 128-bit lane and cross the two lanes. The latter is considered more expensive. The top-left sub-figure in Figure 2.1 shows an example of rearranging data in the same vector. We first exchange the two lanes B and A (using `permute`) and then conduct the in-lane permutation by swapping the middle two elements (using `permute` or `shuffle`). The top-right sub-figure illustrates an another example of using the `unpacklo` instruction to interleave the numbers at even positions from two input vectors. The specific instructions used in AVX and AVX2 might be different from one another. Note, the AVX2 also supports variable-length integers, e.g., `char`, `short`, and `int`.

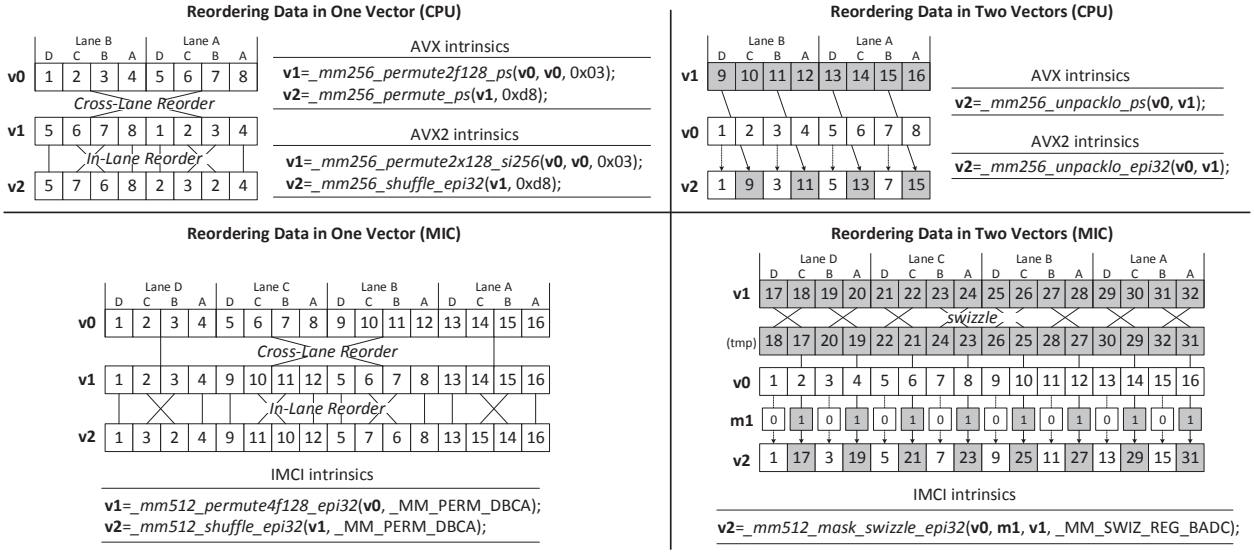


Figure 2.1: Reordering data on Intel CPUs and MICs from the same vector register and the two vector registers, respectively.

IMCI on MICs: The MIC coprocessor consists of up to 61 in-order cores, each of which is outfitted with a VPU. The VPU state for each thread contains 32 512-bit general registers, eight 16-bit mask registers, and a status register. The IMCI is introduced in accordance with the new

VPU. Previous SIMD ISAs, e.g. SSE and AVX, are not supported by the vector architecture of MIC, due to the issues from the wider vector, transcendental instructions, etc. [127]. On MIC, each 512-bit vector is subdivided into four lanes and each lane contains four 32-bit elements. Both of the lanes and elements are ordered as DCBA. The bottom-left sub-figure in Figure 2.1 illustrates the data rearrangement in the same vector register with the `shuffle` and `permute` intrinsics. The permute intrinsic using `_MM_PERM_DBCA` is for cross-lane rearrangement, which exchanges data in lanes C and B. The shuffle intrinsic conducts the same operation but on the element-level within each lane. Because the permute and shuffle intrinsics are executed by different components of the hardware, it is possible to overlap the permute and shuffle intrinsics with a pipeline mode [84]. The bottom-right sub-figure shows another example of picking data from two vector registers with the masked `swizzle` intrinsics. To that end, we use the mask *m1* to select elements from either the swizzled vector of *v1* or the vector *v0*, and then store the result to a blended vector *v2*. The behavior of the mask in Intel MIC is non-destructive, in that no element in source *v0* has been changed if the corresponding mask bit is 0.

2.1.2 Vector Processing in GPUs

In this dissertation, we use GPU programming models from different vendors. The first one is HCC (Heterogeneous Compute Compiler) for AMD GPUs, specifically the GCN3 (Graphics Core Next) architecture. In AMD GPUs, the basic execution unit is called a wavefront (or wave, for brevity) and has 64 lanes. Thus, each thread assigned to a lane ranges from 0 to 63. A wave is assigned to a 16-wide SIMD unit, where each operation takes 4 cycles to finish. The second one is CUDA (Compute Unified Device Architecture) for NVIDIA GPUs, specifically the Kepler, Maxwell, and Pascal architectures, where the basic scheduling unit is a 32-thread warp.

To hide the high latency of off-chip DRAM memory access, modern GPUs possess a cached memory hierarchy. For AMD GCN3, each compute unit (CU) has a 16-kB vector L1 cache and a 64-kB LDS (Local Data Share) as scratchpad memory. In contrast, each multiprocessor (MP) in NVIDIA GPUs has a 48 or 96-kB scratchpad memory (i.e., shared memory). It contains L1 cache as well; however, to enable it, developers may need to turn on `-Xptxas -dlcm=ca` at compile

time, for example, for the Maxwell architecture.

In considering registers as cache, both AMD's CU and NVIDIA's MP possess 256-kB register files that support cross-lane data sharing. For AMD GPUs, this is realized with so-called "permute" instructions, e.g., the backward permute instruction `ds_bpermute_b32`. In contrast, the NVIDIA GPUs support "shuffle" instructions, e.g., the general shuffle instruction `__shlf`. Both `ds_bpermute_b32` and `__shfl` exhibit "pull" semantics, where each thread must read a register value from a target lane. Currently, the GPUs support built-in 32-bit data sharing, while for 64-bit data, one needs to split the data into two values, perform two rounds of data sharing, and then concatenate the results.

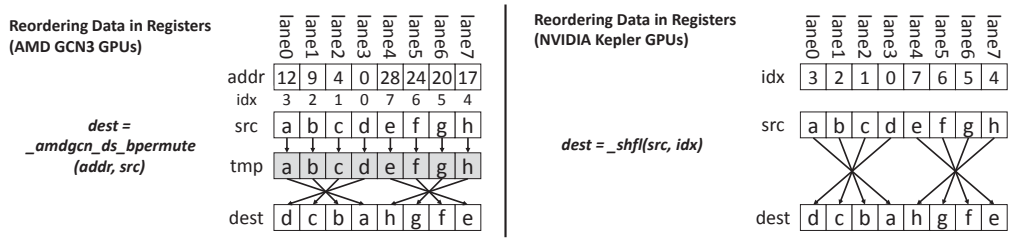


Figure 2.2: Data permute/shuffle in AMD and NVIDIA GPUs. Values in white are stored in registers and the temporary buffer (tmp) is in gray. 8 threads in a wave is for illustration only.

The hardware implementation of data sharing and addressing are different for the two platforms. In AMD GCN3 GPUs, LDS is used to route data between the 64 lanes of a wave, but no actual LDS space is consumed [9]. The left sub-figure in Figure 2.2 shows the target values of *src* that will first be put into a *tmp* buffer. Then, the indices are deduced by ignoring the least two significant bits in *addr*, which are used later to select data from *tmp*. The reordered results are stored in *dest*. In NVIDIA GPUs, threads can directly "read" data from another thread that is actively participating in the `__shfl`. The right sub-figure presents this mechanism of each thread directly accessing data based on the given index.

2.2 Parallel Sort and Segmented Sort

In the following, we provide the related work on the parallel sort on x86-based systems (with the emphasis on vectorization) and segmented sort on GPUs. We also compare the previous work with

our research.

Parallel sort on x86-based systems Efficient sort kernels usually require a careful parallel design to match the underlying hardware, e.g., the vector units. Furtak et al. [65] use SIMD optimizations to handle the base cases of recursive sorting algorithms. AA-sort [82] is a two-phase sorting algorithm with vectorized combsort and odd-even merge on CPUs. Chhugani et al.[48] devise another SIMD-friendly merge sort using the odd-even and bitonic sorting networks. Their solution provides an architecture-specific and hard-coded solution of SSE and Larrabee ISAs but not revealing many details on how the parameters are selected and how to deal with data across vector lanes (e.g., on Larrabee). Inoue et al. [83] propose a stable sorting SIMD solution to rearrange the actual database records. On MICs, Bramas [29] proposes an efficient partition solution for quicksort by using "store some" instructions in AVX-512. Xiaochen et al. [161] studies the bitonic merge sort using both mask and permute IMCI instructions. These existing studies have to explicitly use SIMD intrinsics to handle the tricky data-reordering operations required by different sorting and merging algorithms; while our work (§ 3), in contrast, formalizes the patterns of sorting networks and vector ISAs to facilitate the automatic code generation of efficient and "cross-platform" vector codes.

Segmented sort on GPUs To sort many independent segments in parallel, previous methods mainly rely on the global sort with necessary modification. These solutions may fail to take advantage of both data distribution and architecture, because most of them [53, 63, 122] adopt a "one-size-fits-all" philosophy that treats different segments equally. The mechanisms in [53, 63] sort the whole array after extending input with segment IDs as primary keys, which will consume extra memory space and result in an increased computational complexity. Another mechanism [122] uses one thread block to handle a segment, no matter how different the segments are. It will give rise to some deficiencies when processing a numerable batch of short segments, due to the resource under-utilization. Many GPU applications [154, 102, 186, 185] reformulate the segmented sort problem in terms of global sort and call APIs supported by libraries, sacrificing the benefits of segmentation. In contrast, our method (§ 4) presents a differentiated method for different segment lengths. Especially, we formalize the N-to-M data-tread binding in the register-level sort to

facilitate the searching for the optimal performance of our kernels.

2.3 Parallel Sequence Alignment and Wavefront Computation

Here we provide a brief background on parallelizing the sequence alignment algorithm and its more general form–wavefront computation.

Parallelizing the sequence alignment To parallelize the sequence alignment algorithms (e.g., the Smith-Waterman algorithm [140], Needleman-Wunsch [116] algorithm, and sequence search/alignment tools [184, 183, 182]), the approaches of computation refactoring are usually adopted. One way is to partially ignore the dependencies in one direction and then compensate the intermediate results via multiple times of corrections [105, 62]. An alternative way is to use the prefix sum operations to perform the compensation over the intermediate results [87]. However, the choice of the best parallel strategies depend on the characteristics of input sequences, hardware, etc. Moreover, the realization requires manually working with idiosyncratic vector instructions. Compared to the existing work, the distinctive aspects of our work (§ 5) are to formalize the sequence alignment problems in the view of vector primitives. This additional abstraction layer enables the subsequent vector code generation for different x86 platforms. Furthermore, we provide a new strategy to switch among these strategies no matter the selected algorithms, configurations, and inputs in the runtime.

Parallelizing the wavefront computation The sequence alignment algorithms actually belong to a more general wavefront computation. To parallelize the computation, one direction is to exploit the polyhedral model, which synthesizes affine transformations to adjust the iteration space, to expose the potential parallelism hidden in target loop nests. With regards to GPUs, these efforts include [59, 20, 26]. Another direction concerns how to map the exposed parallelism efficiently on parallel machines. Xiao et al. [160] propose an atomic-based local synchronization to handle dependencies among tiles. PeerWave [24] is a GPU solution for efficient local synchronization between tiles. For the tiling, it utilizes square tiles and hypertiles respectively. Manjikian and Abdelrahman [110] explore the intratile and intertile locality for the large-scale shared-memory

multiprocessors. All the approaches above perform the computation strictly following the original dependency order, which might cause issues on access, locality, and load balance. Differed from them, our work (§ 6) focuses on using a different computational order for more parallelism (from the problems) and more efficiency (from underlying GPUs). This approach is built on a "weighted" prefix sum primitive and we outline its validity and limitation on different combinations of operators using a mathematical proof.

2.4 Parallel Stencil Computation

In stencil computation, each cell is visited multiple times by its neighbors with the computation sweeping over a spatial grid. It is frequently selected in benchmarks for the measure of performance and scalability [90]. The data reuse and memory access is a key problem in stencil computation. Nguyen et al. [117] focus on a 3.5D blocking optimization (i.e., temporal reuse plus spatial 2.5D blocking). Rawat et al. [128] propose an effective tiling strategy to utilize both scratchpad memory and registers for 2D/3D stencils. Vizitiu et al. [149] locate reusable data to constant cache, shared memory, etc., to explore performance variance between different NVIDIA GPUs. Falch and Elster [61] optimize 1D/2D stencils using registers as buffer with manually written shuffle operations. The focus of our work (§ 7) is different, as we are providing a framework for automatically and efficiently accessing neighboring data on different cache levels. It utilizes the knowledge of access patterns to automatically conduct data distribution, synchronization, and communication for different stencils. Note, to take advantage of the shuffle/permute operations, our work is built on AMD and NVIDIA's own programming languages (i.e., HCC and CUDA), rather than the OpenCL and OpenACC.

2.5 Previous Approaches for Performance Portability

Now we introduce the previous approaches related to performance portability: library-based solutions (mainly based on algorithmic skeletons) and framework-based solutions (mainly based on

DSLs).

Library-based solutions Many libraries are designed to take advantage of data-level parallelism from hardware. For x86 platforms, the vector functions (e.g., from the ISPC [123] and Clang 6 [2]) are for the ease of programming vector units and hide the platform-specific codes. For GPUs, many widely-used algorithms and data structures are supported by libraries, e.g., ModernGPU [22], Thrust [71]. However, because not revealing the details of the actual instructions being used or not considering the characteristics of actual inputs, the performance of these codes is not well-understood. Furthermore, programmers still need to figure out how to exploit these functions to meet their demand, e.g., finding appropriate functions and parameters.

Framework-based solutions Since the computation patterns in many applications can be formalized, automation frameworks are proposed to generate codes for parallel machines. Mint [147], Physis [112], Zhang and Mueller [188] present automation mechanisms to translate sequential stencil codes to efficient GPU codes. McFarlin et al. [113] propose a super-optimizer to conduct a guided search of the shortest sequence of vector instructions for desired vector operations. Ren et al. [131] provide an approach to optimize the SIMD code generation for generic permutations. Its code searching part is based on an assumption that any data movements can be directly translated to the SSE's `shufps` instruction. However, the modern vector units of AVX/AVX2/IMCI contain more lanes and more reordering instructions, and thus the search and selection is no long straightforward. This dissertation, by contrast, considers the new features and differences of modern hardwares (e.g., AVX2/IMCI vector operations for CPUs, data exchange with different memory hierarchies for GPUs) to achieve performance portability by studying the research problems, including the data-ordering, data-thread binding, data reuse, etc.

2.6 Broader Performance Portability Issues

The techniques based on parallel patterns can be used to improve not only performance portability but also scalability of many applications in a wide variety of areas.

- Data mining: Many patterns can be found in social media modeling and analysis [35, 36,

38, 164], critical infrastructure analysis [37, 39, 88], network segmentation and summarization [11, 34, 12, 10].

- Data Security: Anomaly detection and web security [178, 179, 180] also follow predictable patterns, which could be accelerated with parallel devices. The parallel patterns can be also applied into security domains to reduce the performance overhead, e.g., application security [146, 145, 144] and web security [143, 162, 41].

Chapter 3

Data-reordering in Vectorized Sort

3.1 Introduction

Increasing processor frequency to improve performance is no longer a viable approach due to its exponential power consumption and heat generation. Therefore, modern processors integrate multiple cores onto a single die to increase inter-core parallelism. Furthermore, the vector processing unit (VPU) associated with each core can enable more fine-grained intra-core parallelism. Execution on a VPU follows the "single instruction, multiple data" (SIMD) paradigm by performing a "lock-step" operation over packed data. Though many regular codes can be auto-vectorized by the modern compilers, some complex loop patterns prevent performant auto-vectorization, due to the lack of accurate compiler analysis and effective compiler transformations [108]. Thus, the burden falls on programmers to implement the manual vectorization using intrinsics or even assembly code.

Writing efficient vectorized (SIMD) code by hand is a time-consuming and error-prone activity. First, vectorizing existing (complex) codes requires expert knowledge in restructuring algorithms to exploit SIMDization potentials. Second, a comprehensive understanding of the actual vector intrinsics is needed. The intrinsics for data management and movement are equally important as those for computation because programmers often need to rearrange data in the vector units before sending them to the ALU. Unfortunately, the flexibility of the data-reordering intrinsics is

restricted, as directly supporting an arbitrary permutation is impractical [84]. As a consequence, programmers must resort to a combination of data-reordering intrinsics to attain a desired computational pattern. Third, the vector instruction set architectures (ISA) continue to evolve and expand, which in turn, lead to potential portability issues. For example, to port codes from the Advanced Vector Extensions (AVX) on the CPU to the codes of the Initial Many Core Instructions (IMCI) on the Many Integrated Core (MIC), we either need to identify the instructions with equivalent functionalities or rewrite and tune the codes using alternative instructions. While library-based optimizations [81] can hide the details of vectorization from the end user, these challenges are still encountered during the design and implementation of the libraries themselves.

One alternate solution to relieve application programmers from writing low-level code is to let them focus only on domain-specific applications at a high level, help them to abstract the computational and communication patterns with potential parallelization opportunities, and leverage modern compiler techniques to automatically generate the vectorization codes that fit in the parallel architectures of given accelerators. For example, McFarlin et al. [113] abstract the data-reordering patterns used in the Fast Fourier Transform (FFT) and generate the corresponding SIMD codes for CPUs. Mint and Physis [147, 112] capture stencil computation on GPUs, i.e., computational and communication patterns across a structured grid.

In this chapter, we focus on the sorting primitive and propose a framework – Automatic SIMDization of Parallel Sorting (a.k.a ASPaS) – to automatically generate efficient SIMD codes for parallel sorting on x86-based multicore and manycore processors, including CPUs and MIC, respectively. ASPaS takes any sorting network and a given ISA as inputs and automatically produces vectorized sorting code as the output. The code adopts a bottom-up approach to sort and merge segmented data. Since the vectorized sort function puts partially sorted data across different segments, ASPaS gathers the sorted data into contiguous regions through a transpose function before the merge stage. Considering the variety of sorting and merging networks[1] [8] that correspond to different sorting algorithms (such as Odd-Even [21], Bitonic [21], Hibbard [70], and Bose-Nelson [27]) and the continuing evolution of instruction sets (such as SSE, AVX, AVX2, and

[1] In this chapter, we distinguish the sorting network and the merging network.

IMCI), it is imperative to provide such a framework to hide the instruction-level details of sorting and allow programmers to focus on the use of the sorting algorithms instead.

ASPaS consists of four major modules: (1) Sorter, (2) Transposer, (3) Merger, and (4) Code Generator. The *SIMD Sorter* takes a sorting network as input and generates a sequence of comparators for the sort function. The *SIMD Transposer* and *SIMD Merger* formalize the data-reordering operations in the transpose and merge functions as sequences of vector-matrix multiplications. The *SIMD Code Generator* creates an ISA-friendly pattern pool containing the requisite data-comparing and reordering primitives, builds those sequences with primitives, and then translates them to the real ISA intrinsics according to the platforms.

We make the following contributions in this chapter. First, for **portability**, we propose the ASPaS framework to automatically generate the cross-platform parallel sorting codes using architecture-specific SIMD instructions, including AVX, AVX2, and IMCI. Second, for **functionality**, using ASPaS, we can generate various parallel sorting codes for the combinations of five sorting networks, two merging networks, and three datatypes (integer, float, double) on Intel Ivy Bridge, Haswell CPUs, and Intel Knights Corner MIC. In addition, ASPaS generates the vectorization codes not only for the sorting of array, but also for the sorting of {key,data} pairs, which is a requisite functionality to sort the real-world workloads. Third, for **performance**, we conduct a series of rigorous evaluations to demonstrate how the ASPaS-generated codes can yield performance benefits by efficiently using the vector units and computing cores on different hardware architectures.

For the one-word type[2], our SIMD codes on CPUs can deliver speedups of up to 4.1x and 6.5x (10.5x and 7.6x on MIC) over the serial sort and merge kernels, respectively. For the two-word type, the corresponding speedups are 1.2x and 2x on CPUs (6.0x and 3.2x on MIC), respectively. Compared with other single-threaded sort implementations, including `qsort` and `sort` from STL [124] and `sort` from Boost [136], our SIMD codes on CPUs deliver a range of speedups from 2.1x to 5.2x (2.5x to 5.1x on MIC) for the one-word type and 1.7x to 3.8x (1.3x to 3.1x on MIC) for the two-word type. Our ASPaS framework also improves the memory access pattern and thread-

[2]We use the 32-bit Integer datatype as the representative of the one-word type, and the 64-bit Double datatype for the two-word type.

level parallelism. That is, we leverage the ASPaS-generated SIMD kernels as building blocks to create a multi-threaded sort (via multi-way merging). Compared with the `parallel_sort` from Intel TBB [130], ASPaS delivers speedups of up to 2.5x and 1.7x on CPUs (6.7x and 5.0x on MIC) for the one-word type and the two-word type, respectively.

3.2 Terminology

This section presents (1) a domain-specific language (DSL) to formalize the data-reordering patterns in our framework, and (2) a sorting and merging network.

3.2.1 DSL for Data-Reordering Operations

To better describe the data-reordering operations, we adopt the representation of a domain-specific language (DSL) from [64, 192] but with some modification. In the DSL, the first-order operators are adopted to define operations of basic data-reordering patterns, while the high-order operators connect such basic operations into complex ones. Those operators are described as below.
First-order operators (x is an input vector):

- S_2 $(x_0, x_1) \mapsto (min(x_0, x_1), max(x_0, x_1))$. The comparing operator resembles the comparator which accepts two arbitrary values and outputs the sorted data. It can also accept two indexes explicitly written in following parentheses.
- A_n $x_i \mapsto x_j, 0 \leqslant i, j < n$, iff $A_{ij} = 1$. A_n represents an arbitrary permutation operators denoted as a permutation matrix which has exactly one "1" in each row and column.
- I_n $x_i \mapsto x_i, 0 \leqslant i < n$. I_n is the identity operator and outputs the data unchanged as its inputs. Essentially, I_n is a diagonal matrix denoted as $I_n = diag(1, 1, \cdots, 1)$.
- L_m^{km} $x_{ik+j} \mapsto x_{jm+i}, 0 \leqslant i < m, 0 \leqslant j < k$. L_m^{km} is a special permutation operator, performing a stride-by-m permutation on the input vector of size km.

High-order operators (A, B are two permutation operators):

($\circ$) The composition operator is used to describe a data flow. $A_n \circ B_n$ means a n-element input vector is first processed by A_n and then the result vector is processed by B_n. The product symbol $\prod$ represents the iterative composition.

($\oplus$) The direct sum operator is served to merge two operators. $A_n \oplus B_m$ indicates that the first n elements of the input vector is processed by A_n, while the rest m elements follow B_m.

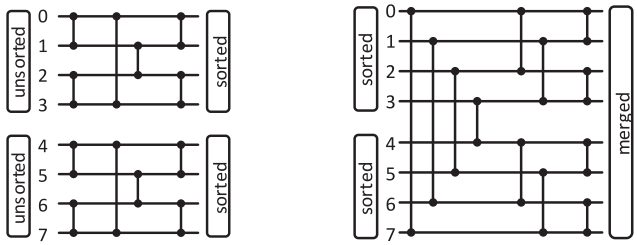
($\otimes$) The tensor product we used in the chapter will appear like $I_m \otimes A_n$, which equals to $A_n \oplus \cdots \oplus A_n$. This means the input vector is divided into m segments, each of which is mapped to A_n.

With the DSL, a sequence of data comparing and reordering patterns can be formalized and implemented by a sequence of vector-matrix multiplications. Note that we only use the DSL to describe the data-comparing and data-reordering patterns instead of creating a new DSL.

3.2.2 Sorting and Merging Network

The sorting network is designed to sort the input data by using a sequence of comparisons, which are planned out in advance regardless of the value of the input data. The sorting network may depend on the merging network to merge pairs of sorted subarrays. Figure 3.1a exhibits the Knuth diagram [8] of two identical bitonic sorting networks. Each 4-key sort network accepts 4 input elements. The paired dots represent the comparators that put the two inputs into the ascending order. After threaded through the wires of the network, these 4 elements are sorted. Figure 3.1b is a merging network to merge two sorted 4-key vectors to an entirely sorted 8-key vector. Although sorting and merging networks are usually adopted in the circuit designs, it is also suitable for SIMD implementation thanks to the absence of unpredictable branches.

In this chapter, the sorting and merging networks are represented by a list of comparators, each of which is denoted as CMP(x, y) that indicates a comparison operation between x-th and y-th elements of the input data.



(a) sorting networks (b) merging network

Figure 3.1: Bitonic networks (a) Two 4-key sorting networks (b) One 8-key merging network.

3.3 Framework and Generalized Patterns

The ASPaS parallel sorting uses an iterative bottom-up scheme to sort and merge segmented data. Algorithm 1 illustrates the scheme: First, the input data are divided into contiguous segments, each of whose size equals to the built-in SIMD width to the power of 2. Second, these segments are loaded into vector registers for sorting with the functions of `aspas_sort` and `aspas_transpose` (the *sort* stage in loop of line 3). Third, the algorithm will merge neighboring sorted segments to generate the output by iteratively calling the function of `aspas_merge` (the *merge* stage in loop of line 9). The functions of `load`, `store`, `aspas_sort`, `aspas_transpose`, and `aspas_merge` will be generated by ASPaS using the platform-specific intrinsics. Since the `load` and `store` can be directly translated to the intrinsics once the ISA is given, we focus on other three kernel functions with the prefix `aspas_` in the remaining sections.

Figure 3.2 depicts the structure of the ASPaS framework to generate the `sort` function. Three modules —*SIMD Sorter*, *SIMD Transposer*, and *SIMD Merger* — are responsible for building the sequences of comparing and data-reordering operations for the aforementioned kernel functions. Then, these sequences are mapped to the real SIMD intrinsics through the module of *SIMD Code Generator*, and the codes will be further optimized from the perspectives of memory access pattern and thread-level parallelism (in § 3.4).

Algorithm 1: ASPaS Parallel Sorting Structure

```
   // w is the SIMD width
1  Function aspas::sort(Array a)
2      Vector v_1, ..., v_w;
3      foreach Segment seg in a do
4          // load seg to v_1, ..., v_w
5          aspas_sort(v_1, ..., v_w);
6          aspas_transpose(v_1, ..., v_w);
7          // store v_1, ..., v_w to seg
8      Array b ← new Array[a.size];
9      for s ←w; s < a.size; s*=2 do
10         for i ←0; i < a.size; i+=2*s do
11             // merge subarrays a+i and a+i+s
12             // to b+i by calling Function aspas::merge()
13         // copy b to a
14     return;
15 Function aspas::merge(Array a, Array b, Array out)
16     Vector v, u;
17     // i_0, i_1, i_2 are offset pointers on a, b, out
18     // load w numbers from a to v
19     // load w numbers from b to u
20     aspas_merge(v, u);
21     // store v to out and update i_0, i_1, i_2
22     while i_0 ⩽ a.size and i_1 ⩽ b.size do
23         if a[i_0] ⩽ b[i_1] then
24             // load w numbers from a + i_0 to v
25         else
26             // load w numbers from b + i_1 to v
27         aspas_merge(v, u);
28         // store v to out + i_2 and update i_0, i_1, i_2
29     // process the remaining elements in a or b
30     return;
```

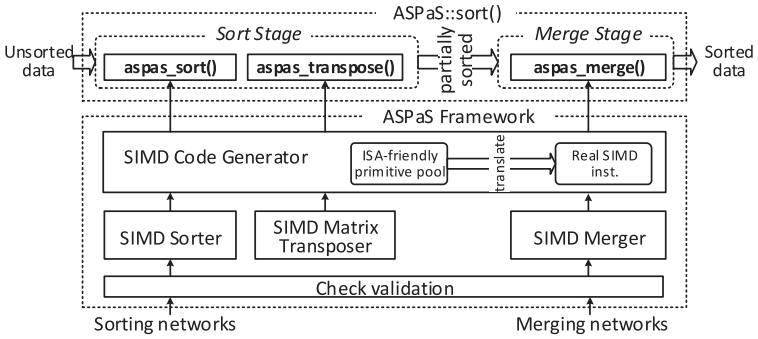


Figure 3.2: The structure of ASPaS and the generated `sort`.

3.3.1 SIMD Sorter

The operations of `aspas_sort` are taken care by the *SIMD Sorter*. As shown in Figure 3.3, `aspas_sort` loads n-by-n elements into n n-wide vectors and threads them through the given sorting network, leading to the data sorted for the aligned positions across vectors. Figure 3.4 presents an example of a 4-by-4 data matrix stored in vectors and a 4-key sorting network (includ-

ing its original input macros). Here, each dot represents one vector and each vertical line indicates a vector comparison. The six comparisons rearrange the original data in ascending order in each column. Figure 3.4 also shows the data dependency between these comparators. For example, CMP(0,1) and CMP(2,3) can be issued simultaneously, while CMP(0,3) can occur only after these two. It is natural to form three groups of comparators for this sorting network. We also have an optimized grouping mechanism to minimize the number of groups for other more complicated sorting networks. For more details, please refer to [75].

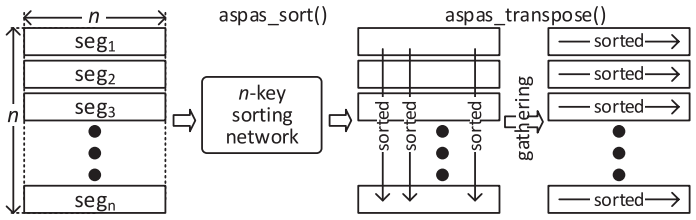


Figure 3.3: Mechanism of the sort stage: operations generated by *SIMD Sorter* and *SIMD Transposer*.

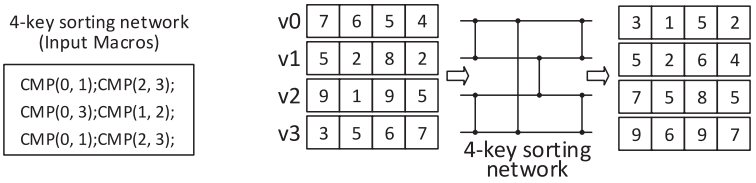


Figure 3.4: Four 4-element vectors go through the 4-key sorting network. Afterwards data is sorted in each column of the matrix.

Since we have the groups of comparators, we can generate the vector codes for the `aspas_sort` by keep two sets of vector variables `a` and `b`. All the initial data are stored in the vectors of set `a`. Then, we jump to the first group of the sorting network. For each comparator in the current group, we generate the vector operations to compare relevant vector variables, and store the results to the vectors in set `b`. The unused vectors are directly copied to set `b`. For the next group, we flip the identities of `a` and `b`. Therefore, the set `b` becomes the input, and the results will be stored back to `a`. This process continues until all groups of the sorting network have been handled. All the vector operations in the `aspas_sort` will be mapped to the ISA-specific intrinsics (e.g., `_mm256_max` and `_mm256_min` on CPUs) later by the *SIMD Code Generator*. At this point, the data is partially sorted but stored in column-major order.

3.3.2 SIMD Transposer

As illustrated in Figure 3.3, the `aspas_sort` function has scattered the sorted data across different vectors. The next task is to gather them into the same vectors (i.e., rows) for further operations. There are two alternative ways to achieve the gathering: one directly uses the gather/scatter SIMD intrinsics; and the other uses the in-register matrix transpose over loaded vectors. The first scheme provides a convenient means to handle the non-contiguous data in memory, but with the penalty of high latency of accessing scattered locations. The second one avoids latency penalty at the expense of using complicated data-reordering operations. Considering the high latency of the gather/scatter intrinsics and the incompatibility with architectures that do not support gather/scatter intrinsics, we adopt the second scheme in the *SIMD Transposer*. To decouple the binding between the operations of matrix transpose and the dedicated intrinsics with various SIMD widths, we formalize the data-reordering operations using the sequence of permutation operators. Subsequently, the sequence will be handed over to the *SIMD Code Generator* to generate the platform-specific SIMD codes for the `aspas_transpose` function.

$$\prod_{j=1}^{t-1}(L_2^{2^t} \circ (I_{2^{t-j-1}} \otimes L_{2^j}^{2^{j+1}}) \circ (I_{2^{t-j}} \otimes L_2^{2^j}) \circ L_{2^{t-1}}^{2^t}[v_{id}, v_{id+2^{j-1}}]) \tag{3.1}$$

Eq. 3.1 gives the operations performed on the preloaded vectors for the matrix transpose, where w is the SIMD width, $t = log(2w)$, and for each j, $id \in \{i \cdot 2^j + n | 0 \leqslant i < \frac{w}{2^j}, 0 \leqslant n < 2^{j-1}\}$, which will form $\frac{w}{2^j} \cdot 2^{j-1} = \frac{w}{2}$ pairs of operand vectors. The sequence of permutation operators preceded each operand pair will be applied on them. The square brackets wrap these pairs of vectors.

Figure 3.5 illustrates an example of in-register transpose with $w = 4$. The elements are preloaded into vectors $v0$, $v1$, $v2$, and $v3$ and have been already sorted vertically. $t-1 = 2$ indicates that there are 2 steps denoted as ① and ② in the figure. For the step $j = 1$, the permutation operators are applied on the pairs $[v0, v1]$ and $[v2, v3]$; and for $j = 2$, the operations are on the pairs $[v0, v2]$ and $[v1, v3]$. After the the vectors go through the two steps accordingly, the matrix is transposed, and the elements are gathered in the same vectors.

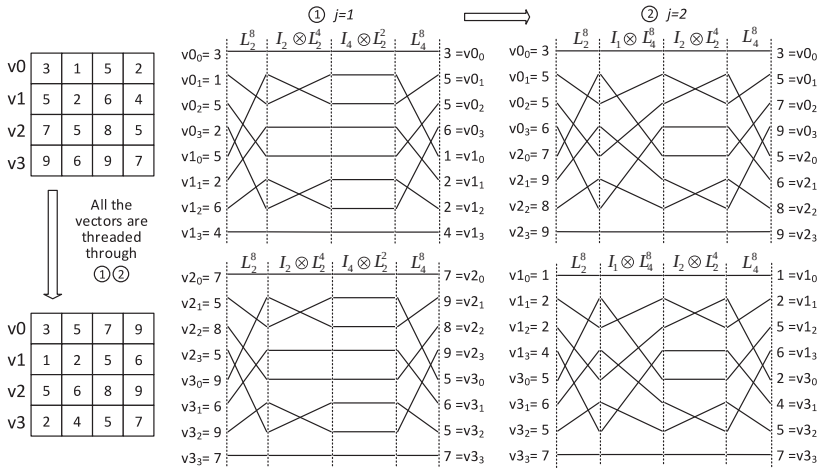


Figure 3.5: Four 4-element vectors transpose with the formalized permutation operators of DSL.

3.3.3 SIMD Merger

For now, the data have been sorted in each segment thanks to the `aspas_sort` and `aspas_transpose`. Then, we use the `aspas_merge` to combine pairs of sorted data into a larger sequence iteratively. The *SIMD Merger* is responsible for its comparison and data-reordering operations according to given merging networks, e.g., odd-even and bitonic networks. In ASPaS, we select the bitonic merging network for three reasons: (1) the bitonic merging network can be easily scaled to any 2^n-sized keys; (2) there is no idle element in the input vectors for each comparison step; and (3) symmetric operations can facilitate the vector instruction selection (discussed in § 3.4.1). As a result, it is much easier to vectorize the bitonic merging network than others. In terms of implementation, we have provided two variants of the bitonic merging networks [192] to achieve the same functionality. Their data-reordering operations can be formalized, as shown below.

$$\prod_{j=1}^{t}(I_{2^{j-1}} \otimes L_2^{2^{t-j+1}}) \circ (I_{2^{t-1}} \otimes S_2) \circ (I_{2^{j-1}} \otimes L_{2^{t-j}}^{2^{t-j+1}})[v,u] \tag{3.2}$$

$$\prod_{j=1}^{t} L_2^{2^t} \circ (I_{2^{t-1}} \otimes S_2)[v,u] \tag{3.3}$$

Similar with § 3.3.2, $t = log(2w)$ and w is the SIMD width. The operand vectors v and u represent two sorted sequences (the elements of vector u are inversely stored in advance). In Eq. 3.2, the data-reordering operations are controlled by the variable j and varies in each step,

while in Eq. 3.3, the permutation operators are independent with j, thereby leading to the uniform permutation patterns in each step. Hence, we label the pattern in Eq. 3.2 as the inconsistent and that in Eq. 3.3 as the consistent. These patterns will be transmitted to and processed by the *SIMD Code Generator* to generate the `aspas_merge` function. We will present the performance comparison of these two patterns in § 3.5.

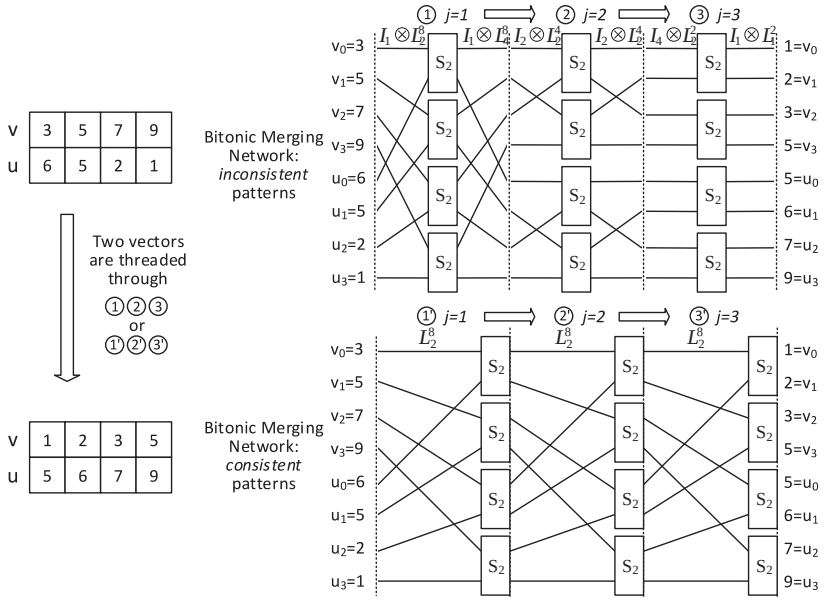


Figure 3.6: Two formalized variants of bitonic merging networks: the inconsistent pattern and the consistent pattern. All elements in vector v and u are sorted, but inversed in vector u.

Figure 3.6 presents an example of the two variants of bitonic merging networks under the condition of $w = 4$. The data-reordering operations from the inconsistent pattern keep changing for each step, while those from the consistent one stay identical. Though the data-reordering operations of the two variants are quite different, both are able to successfully achieve the same merging functionality within the same number of steps, which is actually determined by the SIMD width w.

3.4 Code Generation and Optimization

In the section, we will first show the searching mechanism of ASPaS framework to find out the most efficient SIMD instructions. Then, the generated codes will be optimized to take advantage of memory hierarchy and multi/manycore resources of x86-based systems.

3.4.1 SIMD Code Generator

This module in ASPaS accepts the comparison operations from *SIMD Sorter* and the data-reordering operations from *SIMD Transposer* and *SIMD Merger* in order to generate the real ISA-specific vector codes. We will put emphasis on finding the most efficient intrinsics for the data-reordering operations, since mapping comparison operations to SIMD intrinsics is straightforward. In the module, we first define a SIMD-friendly primitive pool based on the characteristics of the data-ordering operations, then dynamically build the primitive sequences according to the *matching score* between what we have achieved on the way and the target pattern, and finally translate the selected primitives into the real intrinsics for different platforms.

Primitive Pool Building

Some previous research, e.g., the automatic Fast Fourier transform (FFT) vectorization [113], uses the exhaustive and heuristic search on all possible intrinsics combinations, which is time-consuming, especially for the richer instruction sets, such as IMCI. To circumvent the limitation, we first build a primitive pool to prune the search space and the primitives are supposed to be SIMD-friendly. The most notable feature of the data-reordering operations for the transpose and merge is the symmetry: all the operations applied on the first half of input are equivalent with those on the second half in a mirror style. We assume that all the components of the sequences to achieve these operations are also symmetric. We categorize these components as (1) the primitives for the symmetric permute operations on the same vector and (2) the primitives for the blend operations across two vectors.

Permute Primitives: Considering 4 elements per lane (e.g., Integer or Float) or 4 lanes per register (e.g., IMCI register), there are $4^4 = 256$ possibilities for either intra-lane or inter-lane permute operations. However, only those permutations without duplicated values are useful in our case, reducing the possibilities to $4! = 24$. Among them, merely 8 symmetric data-reordering patterns will be selected, i.e. `DCBA` (original order), `DBCA`, `CDAB`, `BDAC`, `BADC`, `CADB`, `ACBD`, and `ABCD`, in which each letter denotes an element or a lane. If we are working on 2 elements per lane (e.g., Double) or 2 lanes per register (e.g., AVX register), there are two symmetric patterns without duplicated values, i.e. `BA` (original order) and `AB`.

Blend Primitives: While blending two vectors into one, the elements are supposed to be equally and symmetrically distributed from the two input vectors. Hence, we can boil down the numerous mask modifiers to only a few. We define a pair of blend patterns (0_2^i, $2^i\_2^i$), where $0 \leqslant i < log(w)$ and w is the vector width. Each blend pattern in the pair represents a 2^{i+1}-bit stream. The first number 0 or 2^i denotes the offset of the first set bit, and the second number 2^i is the number of consecutive set bits. All the other bits are filled with clear bits. The bit streams need to be extended to the vector width by duplicating themselves $\frac{w}{2^{i+1}}$ times. For example, if the w equal to 16, there are 4 possible pairs of patterns: (0_1, 1_1), (0_2, 2_2), (0_4, 4_4), and (0_8, 8_8). Among them, the pair (0_2, 2_2) corresponds to $i = 1$, representing the bit streams $(1100)_4$ and $(0011)_4$ (The subscript 4 means the repetition times).

Now, we further categorize the primitives into 4 types based on permute or blend and intra-lane or inter-lane. Table 3.1 illustrates the categories and associative exemplar operations, where the vector width w is set to 8 (2 lanes) for clarity.

Table 3.1: Primitive Types

Type #	Type	Example (vector_width=8)
0	intra-lane-permute	ABCDEFGH→BADCFEHG (cdab)
1	inter-lane-permute	ABCDEFGH→EFGHABCD (−−ab)
2	intra-lane-blend	ABCDEFGH | IJKLMNOP→ABKLEFOP (2_2)
3	inter-lane-blend	ABCDEFGH | IJKLMNOP→IJKLEFGH (0_4)

The primitives are materialized into permutation matrices in ASPaS. Since the blend primitives always operate on two vectors (concatenated as one $2w$ vector), the dimensions of the blend

permutation matrices are expanded to $2w$ by $2w$ as well. Accordingly, for the permute primitives, we pair an empty vector to the single input vector and specify the primitive works on the first vector v or the second vector u. Therefore, for example, if $w = 16$, there are 32=8(permute primitives)$*$2(intra-lane or inter-lane)$*$2(operating on v or u) and 8(4 pairs of the blend primitives) permutation matrices. Figure 3.7 illustrates examples of the permutation matrices. The matrix "shuffle_cdab_v" and "shuffle_cdab_u" correspond to the same permute primitive on the halves of the concatenated vectors. The matrix "blend_0_1_v" and "blend_1_1_u" correspond to one pair of blend primitives (0_1, 1_1). So far, 4 sub-pools of permutation matrices are created according to the 4 primitive types.

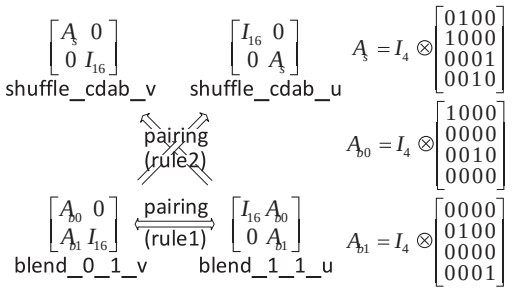


Figure 3.7: Permute matrix representations and the pairing rules.

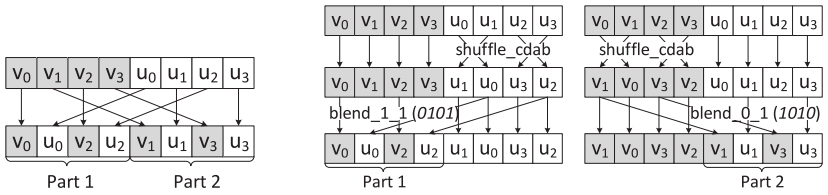
Sequence Building

Two rules are used in the module to facilitate the searching process. They are based on two observations from the formalized data-reordering operations illustrated in Eq. 3.1, Eq. 3.2, and Eq. 3.3. Obs.1 The same data-reordering operations are always conducted on two input vectors. Obs.2 The permute operations always accompany the blend operations to keep the symmetric pattern. Figure 3.8 exhibits the symmetric patterns, which are essentially the first step in Figure 3.5. The default blend is limited to pick elements from aligned positions of two input vectors, while the symmetric blend can achieve an interleaving mode by coupling permute primitives with blend primitives, as the figure shown. Hence, the usage of the two rules in the sequence building algorithm are described as below.

Rule 1: when a primitive is selected for one vector v, pair the corresponding primitive for the other

vector u. For a permute primitive, the corresponding permute has the totally same pattern; while for a blend primitive, the corresponding blend has the complementary blend pattern (i.e. the bit stream, which has already been paired).

Rule 2: when a blend primitive is selected, pair it with the corresponding permute primitive: pair the intra-lane-permute of swapping adjacent elements (`CDAB`) for (0_1, 1_1) blend, the intra-lane-permute of swapping adjacent two elements (`BADC`) for (0_2, 2_2), the inter-lane-permute of swapping adjacent lanes (`CDAB`) for (0_4, 4_4), and the inter-lane-permute of swapping adjacent two lanes (`BADC`) for (0_8, 8_8).



(a) `sym-blend` (b) `sym-blend (details)`

Figure 3.8: Symmetric blend operation and its pairing details.

The sequence building algorithm targets at generating sequences of primitives to achieve given data-reordering patterns for Eq. 3.1, Eq. 3.2, and Eq. 3.3. Two w-sized input vectors of v and u are used and concatenated into the vec_{inp}. Its initial elements are set to the default indices (from 1 to $2w$). The vec_{trg} is the target derived by applying the given data-reordering operators on the vec_{inp}. Then, the building algorithm will select the permutation matrices from the primitive pool, do the vector-matrix multiplications over the vec_{inp}, and check whether the intermediate result vec_{im} approximates the vec_{trg} by using our defined two *matching scores*:

l-score lane-level matching score, accumulate by one when the corresponding lanes have exactly same elements (no matter orders).

e-score element-level matching score, increase by one when the element matches its counterpart in the vec_{trg}.

Suppose we have a vector of w (vector width) and e (number of elements per lane), the maximum l-score equals to $2w/e$ when all the aligned lanes from two vectors match, while the max-

imum e-score is $2w$ when all the aligned elements match. With the matching scores, the process of sequence building is transformed to finding the maximum scores. For example, if we have the input "AB|CD|EF|GH" and the output "HG|DC|FE|BA" (assuming four lanes and two elements per lane), we first search primitives for the inter-lane reordering, e.g, from "AB|CD|EF|GH" to "GH|CD|EF|AB", and then search primitives for the intra-lane reordering and reach to, e.g., from "GH|CD|EF|AB" to "HG|DC|FE|BA". By checking the primitives hierarchically, we add those primitives increasing l-score or e-score and thus approximate to the desired output pattern.

Algorithm 2: Sequence Building Algorithm

```
   Input: vec_inp, vec_trg
   Output: seqs_ret
1  Sequences seqs_cand ← new Sequences(∅); // put an null sequence
2  Int l_score_init ←LaneCmp(vec_inp, vec_trg);
3  if l_score_init=2w/e then
4      seqs_ret ← InstSelector(seqs_cand,Type[0]);
5  else
6      i ←1;
7      while not Threshold() do
8          ty ←Type[i];
9          foreach Sequence seq in seqs_cand do
10             vec_im ←Apply(vec_inp, seq);
11             l_score_old ←LaneCmp(vec_im, vec_trg);
12             foreach Primitive prim in ty do
13                 if n=1 then
14                     prim_prd ←Pair(prim, RULE1);
15                     vec_upd ←Apply(vec_im, prim + prim_prd);
16                     seq_ext ←prim + prim_prd;
17                 else
18                     prim_prd ←Pair(prim, RULE1);
19                     perm_0 ←Pair(prim, RULE2);
20                     perm_1 ←Pair(prim_prd, RULE2);
21                     vec_upd0 ←Apply(vec_im, perm_0 + prim);
22                     vec_upd1 ←Apply(vec_im, perm_1 + prim_prd);
23                     vec_upd ←Combine(vec_upd0, vec_upd1);
24                     seq_ext ←perm_0 + prim + perm_1 + prim_prd;
25                 l_score_new ←LaneCmp(vec_upd, vec_trg);
26                 if l_score_new > l_score_old then
27                     seqs_buf.add(seq + seq_ext);
28         seqs_cand.append(seqs_buf);
29         seqs_buf.clear();
30         i ←((++i)-1)%3+1;
31     seqs_sel ←PickLaneMatchedSeqs(seqs_cand);
32     seqs_ret ← InstSelector(seqs_sel,Type[0]);
33 Function InstSelector(Sequences seqs_cand,Type ty)
34     foreach Sequence seq in seqs_cand do
35         vec_im ←Apply(vec_inp, seq);
36         foreach Primitive prim in ty do
37             prim_prd ←Pair(prim, RULE1);
38             vec_upd ←Apply(vec_im, prim + prim_prd);
39             e_score ←ElemCmp(vec_upd, vec_trg);
40             if e_score=2w then
41                 seqs_ret.add(seq + prim + prim_prd);
42     return seqs_ret;
```

Algorithm 2 shows the pseudocode of the sequence building algorithm. The input contains the aforementioned vec_{inp} and vec_{trg}. The output $seqs_{\mathsf{ret}}$ is a container to hold the built sequences of primitives, which will be translated to the real ISA intrinsics soon. The $seqs_{\mathsf{cand}}$ is to store candidate sequences and initialized to contain a $\emptyset$ sequence. First, the algorithm checks the initial vec_{inp} with the vec_{trg} and get the l-score. If it equals to $2w/e$, meaning aligned lanes have already matched, we only need to select "intra-lane-permute" primitives (line 4) to improve the e-score. Otherwise, we will work on the sub-pools of type 1, 2, or 3 in a round-robin manner. In the while loop, for each sequence in $seqs_{\mathsf{cand}}$, we first calculate the l_score_{old}, and then we will calculate the l_score_{new} by tentatively adding primitives one by one from the current sub-pool. If the primitive $prim$ comes from the "inter-lane-permute", we produce the paired permute primitive $prim_{\mathsf{prd}}$ based on the Rule 1 (line 14). If $prim$ is from the blend types, we produce the paired blend primitive $prim_{\mathsf{prd}}$ based on the Rule 1 and then find their paired permute primitives $perm_0$ and $perm_1$ based on the Rule 2 (line 18-20). The two rules help to form the symmetric operations.

After the selected primitives have been applied, which corresponds to several vector-matrix multiplications, we can get a vec_{upd}, leading to a new l-score l_score_{new} compared to vec_{trg} (line 25). If the l-score is increased, we add the sequence of the selected primitives to $seqs_{\mathsf{cand}}$ for further improvement. The threshold (line 7) is a configuration parameter to control the upper bound of how many iterations the algorithm can tolerate, e.g., we set it to 9 in the evaluation in order to find the sequences as many as possible. Finally, we use `PickLaneMatched` to select those sequences that can make l-score equal to $2w/e$, and go to the "intra-lane-permute" selection (line 32), which can ensure us the complete sequences of primitives.

Primitives Translation

Now, we can map the sequences from the $seqs_{\mathsf{ret}}$ to the real ISA intrinsics. Although the vector ISAs from CPU or MIC platforms are distinct from one another, we can still find desired intrinsics thanks to the SIMD-friendly primitives. If there are multiple selections to achieve same primitive, we always prefer the selection having least intrinsics.

On MIC: if there are multiple shortest solutions exist, we use the interleaved style of inter-

lane and intra-lane primitives, which could be executed with a pipeline mode on MIC as discussed in § 2.1.1. For the primitives from "intra-lane-permute" and "inter-lane-permute", we directly map them into vector intrinsics of `_mm512_shuffle` and `_mm512_permute4f128` with appropriate permute parameters. The primitives from "intra-lane-blend" and "inter-lane-blend" are mapped to the masked permute intrinsics `_mm512_mask_shuffle` and `_mm512_mask_permute4f128`. The masks are derived from their blend patterns. Furthermore, when a primitive is from "intra-lane" and its parameter is supported by the swizzle intrinsics, we will use the light-weighted swizzle intrinsics to optimize the performance.

On CPU: for the primitives from "intra-lane-permute" and "inter-lane-permute", we map them into vector intrinsics of AVX's `_mm256_permute` (or AVX2's `_mm256_shuffle`[3]) and `_mm256_permute2f128` with appropriate permute parameters. For the primitives of blend primitives, we need to find specific combinations of intrinsics, since there are no similar mask mechanisms in AVX or AVX2 as IMCI. For "intra-lane-blend" primitives, if the blend pattern is picking interleaved numbers from two vectors, e.g., 0101 or 1010, we use the `_mm256_unpacklo` and `_mm256_unpackhi` to unpack and interleave the neighboring elements. In contrast, for the patterns that select neighboring two elements, e.g., 0011 or 1100, we use AVX's `_mm256_shuffle`, which can take two vectors as input and pick every two elements from each input. For the "inter-lane-blend" primitives, we use `_mm256_permute2f128`. Note that, since many intrinsics in AVX only support operations on floating point elements, we have to cast the datatypes if we are working on integers; while on AVX2, we can directly use intrinsics handling integers without casting. As a result, for parallel sorting on integers, the generated codes on AVX2 may use much less intrinsics than those on AVX.

3.4.2 Organization of the ASPaS Kernels

So far, we have generated three building kernels in ASPaS: `aspas_sort()`, `aspas_transpose()`, and `aspas_merge()`. As shown in Figure 3.9, we carefully organize these kernels to form the `aspas::sort` as illustrated in Algorithm 1. Note that this figure shows the sort, transpose, and

[3]AVX's `_mm256_shuffle` is different from AVX2's and it reorders data from two input vectors.

merge stages on each thread and the multithreaded implementation will be discussed in the next subsection. First, the `aspas_sort()` and `aspas_transpose()` are performed on every segment of the input to create a partially sorted array. Second, we enter the *merge* stage. Rather than directly merging the sorted segments level by level in our previous research [75], we adopt the multiway merge [83, 48]: merge the sorted segments for multiple levels in each block, the cache-sized trunk, to fully utilize the data in the cache until we move to the next block. This strategy is cache-friendly, since it avoids frequently swapping data in and out the cache. When the merged segments are small enough to fit into the LLC, which is usual in first several levels, we take this multiway merge strategy. For the large segments in the later levels, we fall back to the two-way merge. Similar to existing libraries, e.g., STL, Boost, and TBB, we also provide the merge functionality for programmers as a separate interface. The interface `aspas::merge` is similarly organized as `aspas::sort` shown in the figure but only uses `aspas_merge()`.

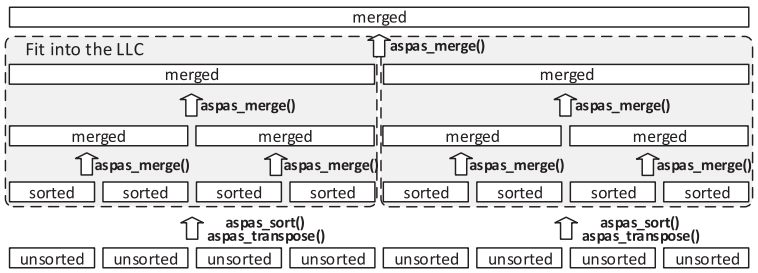


Figure 3.9: The organization of ASPaS kernels for the **single-threaded** `aspas::sort`.

3.4.3 Thread-level Parallelism

In order to maximize the utilization of multiple cores of modern x86-based systems, we integrate the `aspas::sort` and `aspas::merge` with the thread-level parallelism using Pthreads. Initially, we split the input data into separate parts, each of which is assigned to one thread. All the threads can sort their own parts using the `aspas::sort` independently. Then, we merge each thread's sorted part together. The simplest way might be assigning half of the threads to merge two neighboring sorted parts into one by iteratively calling the `aspas::merge` until there is only one thread left. However, this method significantly under-utilizes the computing resources. For

example, in the last level of merging, there is only one thread merging two trunks but all the other threads are idle. Therefore, for the last several levels of merging, we adopt MergePath [68] to let multiple threads merge two segments. Assume for each two sorted segments with the lengths of m and n, we have k threads working on them. First, each thread calculates the i/k-th value in the imagined merged array without actually merging the inputs, where the i is the thread index. This step can be done in O(log($m + n$)). Second, we split the workloads into k exclusive and balanced portions according to the k splitting values. Finally, each thread can merge their assigned portions independently. Note, this strategy is capable of minimizing the data access overhead on remote memory bank of NUMA architecture, since the array is equally split and stored in each memory bank and a thread will first merge data in the local memory region, and then on demand access remote data in a serial mode [83]. In the evaluation, our multithreaded version adopts this optimized design.

3.4.4 Sorting of {key,data} Pairs

In many real-world applications, sorting is widely used to reorder some specific data structures based on their keys. To that end, we extend ASPaS with this functionality: generate the vectorization codes to sort {*key*, *data*} pairs, where the *key* represents the target for sorting and the *data* is the address to the data structures containing that *key*. The research work [48] proposes two strategies to sort {*key*, *data*} pairs. The first strategy to sort {*key*, *data*} pairs is to pack the relative *key* and *data* into a single entry. Then, sorting the entries is equivalent to sorting the keys, since the keys are placed in the high bits. However, if the sum of lengths of *key* and *data* exceeds the maximum length of the built-in data types, it is non-trivial to carry this strategy out. The second strategy is to put the keys and data into two separate arrays. While sorting the keys, the comparison results are stored as masks that will be used to control the data-reordering of associative data. In this chapter, we use the second method. Differed from [48], which focuses on the 32-bit key and data, ASPaS is able to handle different combinations of 32/64-bit keys and 32/64-bit data and their varied data-reordering patterns accordingly.

For implementation, ASPaS uses *compare* intrinsics rather than *max/min* intrinsics to get ap-

propriate masks. The masks may need be stretched or split depending on the differences between the lengths of keys and data. With the masks, we use *blend* intrinsics on both key vectors and data vectors to reorder elements. Table 3.2 shows how the building modules are used to find the desired intrinsics for key and data vectors, respectively.

Table 3.2: The building modules to handle the data-reordering for {key,data} pairs in ASPaS

{key,data}	Input (key)	Building Modules (key)	Input (data)	Building Modules (data)
32-bit, 32-bit	v_0,v_1,...,v_{w-1}	Transpose[w](v_0,v_1,...,v_{w-1})	v'_0,v'_1,...,v'_{w-1}	Transpose[w](v'_0,v'_1,...,v'_{w-1})
64-bit, 64-bit	v,u	Merge_Reorder[w](v,u)	v',u'	Merge_Reorder[w](v',u')
32-bit, 64-bit	v_0,v_1,...,v_{w-1}	Transpose[w](v_0,v_1,...,v_{w-1})	v'_0,v'_1,v'_2,v'_3, ...,v'_{2w-2},v'_{2w-1}	[Transpose[w/2](v'_0,v'_2,...,v'_{w-2}), Transpose[w/2](v'_{w+0},v'_{w+2},...,v'_{2w-2}) Transpose[w/2](v'_1,v'_3,...,v'_{w-1}), Transpose[w/2](v'_{w+1},v'_{w+3},...,v'_{2w-1})]
	v,u	Merge_Reorder[w](v,u)	v'_0,v'_1,u'_0,u'_1	Merge_Reorder[w/2](v'_0,u'_0); Merge_Reorder[w/2](v'_1,u'_1)
64-bit, 32-bit	v_0,v_1,...,v_{w-1}	Transpose[w](v_0,v_1,...,v_{w-1})	v'_0,v'_1,...,v'_{w-1}†	Transpose[w](v'_0,v'_1,...,v'_{w-1})
	v,u	Merge_Reorder[w](v,u)	v',u'†	Merge_Reorder[w](v',u')

†: On MIC, only the first halves of each vector are effective; On CPU, SSE vectors are adopted.

In the table, w represents the number of keys the built-in vector can hold. The modules are in the format of *modName*[*count*](*vlist*), which means generating the *modName* data-reordering intrinsics for vectors in *vlist* and each vector contains *count* elements. There are three possible combinations for different keys and data: (1) When the key and data has the same length, we use the totally same data-reordering intrinsics on the key and data vectors. (2) When the data length doubles the key length, we correspondingly double the number of vectors to hold the enlarged data values. Then, the building modules are performed on halves of the input data vectors as shown in the table: for transpose, we need to use four times intrinsics on data vectors than key vectors to transpose four blocks of data vectors, and change the layout of data vectors from [00, 01, 10, 11] to [00, 10, 01, 11]; for merge, we need to double the intrinsics on data vectors than key vectors since the input vectors are doubled. (3) When the key length exceeds the data length, we take distinct strategies according to the platforms. On CPU, we simply use the SSE vector ISA, because of the backward compatibility of AVX. On MIC, since the platform doesn't support previous vector ISA, we keep the effective values always in the first halves of each 512-bit vectors.

One may wonder why we need to reorder the *data* along with the *key* in each step rather than do it only in the final step. The reason is that this alternative requires an additional "index" vector to keep track of *key* movement, which occurs during each step of reordering of the *keys*. Thus, it is same to our strategy because the *data* in our method is the address to the real data structure.

Moreover, the reordering of *data* in our method has adopted ISA intrinsics for vectorization, which can avoid the irregular memory access. In the perspective of performance, the execution time of sorting {key,data} pairs grows asymptotically compared to sorting the pure key array. Henceforth, we will focus on the performance analysis of sorting pure key array in the evaluation section.

3.5 Performance Analysis

ASPaS supports major built-in data types, i.e., integers, single and double precision floating point numbers. In our evaluation, we use the Integer for the one-word type (32-bit) and the Double for the two-word type (64-bit). Our codes use different ISA intrinsics according to the different platforms. Table 3.3 shows the configurations of the three platforms with Intel Ivy Bridge (IVB), Haswell (HSW), and Knights Corner (KNC), respectively. The ASPaS programs are implemented in C++11 and compiled using Intel compiler *icpc* 15.3 for HSW and KNC and *icpc* 13.1 for IVB. On CPUs, we use the compiler options of *-xavx* and *-xCORE-AVX2* to enable AVX and AVX2, respectively. On MIC, we run the experiments using the *native* mode and compile the codes with *-mmic*. All codes in our evaluations are optimized in the level of *-O3*. All the input data are generated randomly ranging from 0 to the data size, except in § 3.5.5. This chapter focuses on the efficiency of vectorization; and we show detailed performance analysis on a single thread in most sections, while § 3.5.4 evaluates the best vectorized codes in a multi-core design.

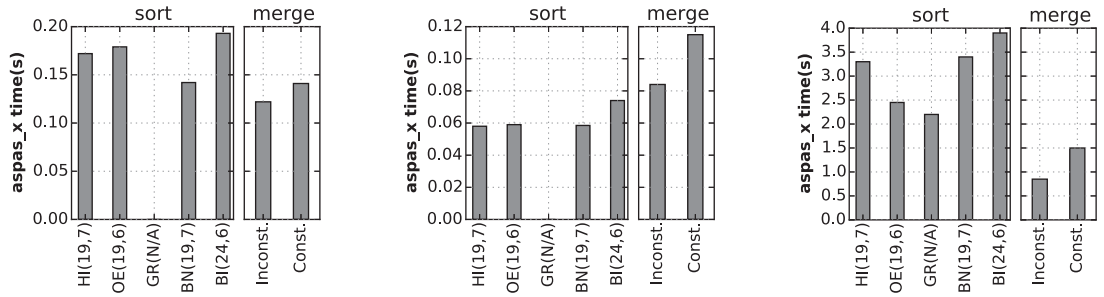
Table 3.3: Testbeds for ASPaS

Model	**Intel Xeon CPU (E5-2697 v2)**	**Intel Xeon CPU (E5-2680 v3)**	**Intel Xeon Phi (5110P)**
Codename	Ivy Bridge	Haswell	Knights Corner
Frequency	2.70GHz	2.50GHz	1.05GHz
Cores	24	24	60
Threads/Core	2	1	4
Sockets	2	2	-
L1/L2/L3	32kb/256kb/30mb	32kb/256kb/30mb	32kb/512kb/-
Vector ISA	AVX	AVX2	IMCI
Memory	64GB	128GB	8GB
Mem Type	DDR3	DDR3	GDDR5

3.5.1 Performance of Different Sorting Networks

We first test the performance of the `aspas_sort` and `aspas_merge` kernels, whose implementation depends on the input sorting and merging networks. For brevity, we only show the graphical results of Integer datatype. We repeat the execution of the kernels for 10 million times and report the total seconds in Figure 3.10.

In the *sort* stage, ASPaS can accept any type of sorting networks and generate the `aspas_sort` function. We use five sorting networks, including Hibbard (HI) [70], Odd-Even (OE) [21], Green (GR) [67], Bose-Nelson (BN) [27], and Bitonic (BI) [21]. In Figure 3.10, since GR cannot take 8 elements as input, the performance for it on CPUs is not available. The labels of x-axis also indicate how many comparators and groups of comparators in each sorting network are. On CPUs, the sorting networks have same number of comparators except the BI sort, thereby yielding negligible time difference with a slight advantage to BN sort on IVB. On MIC, GR sort has the best performance that stems from the less comparators and groups, i.e., (60, 10). Although BI sort follows a balanced way to compare all elements in each step and is usually considered as a candidate for better performance, it uses more comparators, leading to the relatively weak performance for the *sort* stage. Base on the results, in the remaining experiments, we choose the BN, OE, and GR sorts for the Integer datatype on IVB, HSW, and KNC, respectively. For the Double datatype, we also choose the best one, i.e., OE sort, for the rest of the experiments.



(a) IVB (b) HSW (c) KNC

Figure 3.10: Performance comparison of `aspas_sort` and `aspas_merge` with different sorting and merging networks. The kernels are repeated by 10 million times over a built-in vector-length array and total times are reported. The numbers of comparators and groups are given in parenthesis for sorting networks.

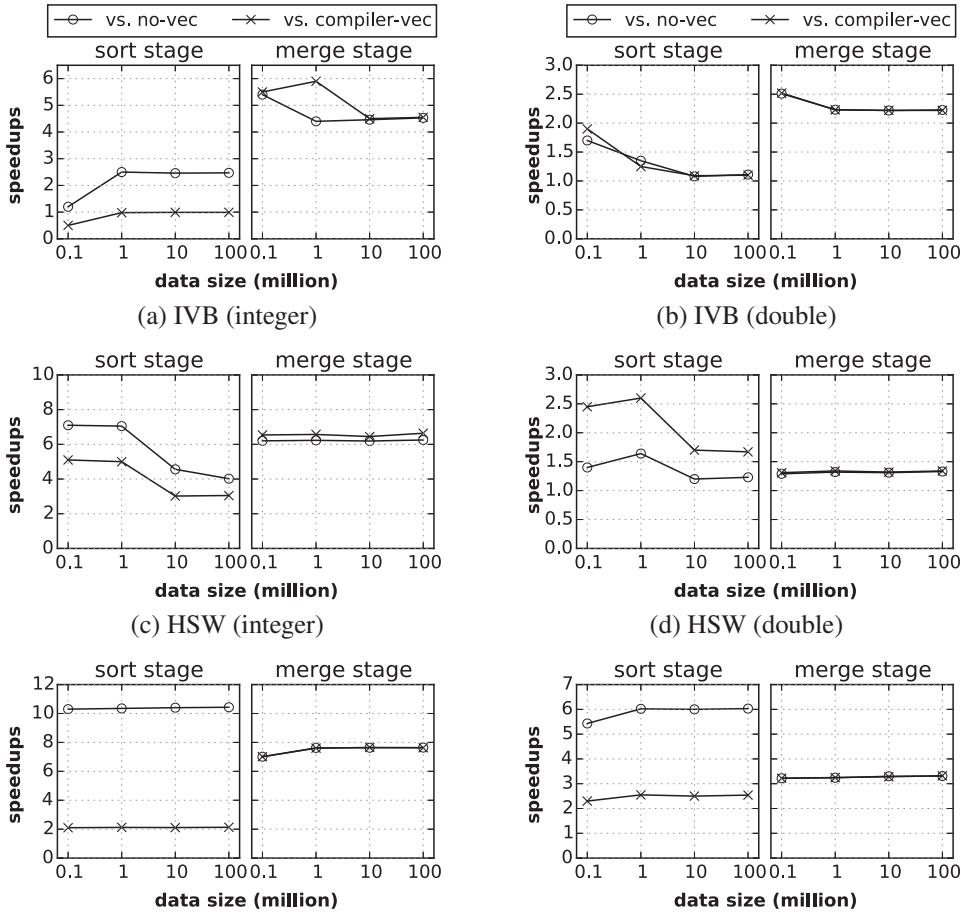
In the *merge* stage, we set two variants of bitonic merging networks (Eq. 3.2 and Eq. 3.3 in § 3.3.3) as the input of ASPaS. Figure 3.10 also presents the performance comparisons for these two variants. The inconsistent merging can outperform the consistent one by 12.3%, 20.5%, and 43.3% on IVB, HSW, and KNC, respectively. Although the consistent merging has uniform data-reordering operations in each step as shown in Figure 3.6, the operations are not ISA-friendly and thus requires a longer sequence of intrinsics. For example, based on Eq. 3.3, the consistent merging uses 5 times of the L_2^{32} data reordering operations on MIC, each of which needs 8 permute/shuffle IMCI intrinsics. In contrast, the inconsistent merging only uses L_2^{32} once and compensate it with much lighter operations (e.g., $I_1 \otimes L_{16}^{32} \circ I_2 \otimes L_2^{16}$ and $I_2 \otimes L_8^{16} \circ I_4 \otimes L_2^8$, each of which can be implemented by an average of 2 IMCI intrinsics). On CPUs, the L_2^{16} operation in the consistent variant only needs 4 AVX intrinsics, leading to the smaller disparity. But, in all cases, the inconsistent bitonic merge provides the best performance. The Double datatype exhibits similar behaviors. Thus we will adopt the inconsistent merging in the remaining experiments.

3.5.2 Speedups from the ASPaS Framework

In this section, we compare the ASPaS *sort* and *merge* stages with their serial counterparts. The counterparts of `aspas_sort` and `aspas_merge` are serial sorting and merging networks (one comparison and exchange at a time) respectively. Note, in the *sort* stage, the `aspas_transpose` is not required in the serial version, since the partially sorted data can be stored directly in a consecutive manner. Ideally, the speedups from the ASPaS should approximate the built-in vector width; though this is impractical because of the extra and required data reordering instructions. By default, the compiler will auto-vectorize the serial codes[4], which is denoted as "compiler-vec". Besides, we explicitly turn off the auto-vectorization, which is shown as "no-vec".

For the *sort* stages with Integer datatype on CPUs in Figure 3.11 (a,c), the ASPaS codes can deliver more performance improvements on HSW over IVB, since the AVX on IVB does not support native integer operations as in AVX2. Thus, we have to split the AVX vector to two SSE vectors before resorting to the SSE ISA for comparisons. For the *sort* stages with Double

[4]We also use the SIMD pragma *pragma vector always* on the target loops.



(a) IVB (integer)

(b) IVB (double)

(c) HSW (integer)

(d) HSW (double)

(e) KNC (integer)

(f) KNC (double)

Figure 3.11: ASPaS vs. *icpc* optimized ("compiler-vec") and serial ("no-vec") codes. For the *merge* stages, the lines of "compiler-vec" and "no-vec" usually overlap.

in Figure 3.11 (b,d), the ASPaS codes exhibit similar performance gains over "no-vec", achieving slight 1.1˜1.2x speedups. The vectorization benefits of Double drop down because less elements in each vector than Integer, leading to relatively higher data reordering overhead. On KNC, ASPaS Integer and Double *sort* codes in Figure 3.11 (e,f) outperform the "no-vec" counterparts up to 10.5x and 6.0x. In addition, the ASPaS codes can also achieve better performance than the "compiler-vec" versions in most cases. By analyzing the generated assembly codes in "compiler-vec", we find: on IVB, the compiler uses multiple `insert` instructions to construct vectors slot by slot from non-contiguous memory locations; instead, the `gather` instructions are used on HSW and KNC. However, neither can mitigate the high latency of non-contiguous memory access. The

ASPaS codes, in contrast, can outperform the "compiler-vec" by using the `load/store` on the contiguous data and the `shuffle/permute` for the transpose in registers. We also observe that in Figure 3.11 (d) the "compiler-vec" of *sort* stage slowdowns the execution compared to the "no-vec". This may stem from the fact that the HSW supports vector `gather` but no equivalent vector `scatter` operations. The asymmetric load-and-store fashion on non-contiguous data with larger memory footprint (Double) causes negative impacts on the performance [108].

The *merge* stages in Figure 3.11 on the three platforms show that the "compiler-vec" versions have the similar performance with the "no-vec". This demonstrates that even with the most aggressive vectorization pragma, the compiler fails to vectorize the merge codes due to the complex data dependency within the loops.

3.5.3 Comparison to Previous SIMD Kernels

In this section, we compare our generated kernels with those manually optimized kernels proposed in previous research. These existing vector codes also focus on using vector instructions and sorting networks to sort small arrays with sizes of multiple of SIMD-vector's length. The reasons for comparing kernels with smaller data sizes rather than any large data size are following: (1) the kernels for sorting small arrays are usually adopted to construct efficient parallel sort algorithms in a divide-and-conquer manner (e.g., quick-sort [49, 29], merge-sort [134, 48]), where input data is split into small chunks each of which fits into registers, the sort kernel is applied on each chunk, and the merge kernel is called iteratively to merge chunks until there is only one chunk left. Under this circumstance, the overall performance significantly depends on the vectorization kernels [49]; (2) Our major motivation of this chapter is to efficiently generate combinations of permutation instructions instead of proposing a new divide-and-conquer strategy for any large data size. As a result, we compare vector codes from Chhugani et al. (CH) [48] and Inoue et al. (IN) [83] on CPUs; while on MICs, we compare vector codes from Xiaochen et al. (XI) [161] and Bramas (BR) [29]. The datatype in this experiment is the 32-bit integer[5]. We use one core (vector unit) to

[5]The BR paper [29] only provides AVX-512 codes for Knights Landing (KNL). Therefore, we have

process randomly-generated data in the segment of 8x8=64 integers for CPUs and of 16x16=256 integers for MICs, respectively. The experiments are repeated for 1 million times and we report the total execution time.

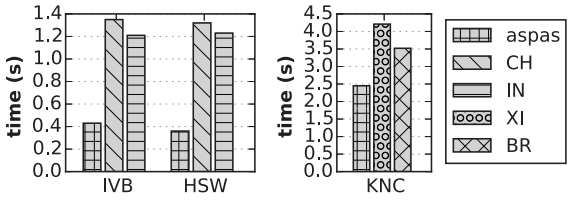


Figure 3.12: ASPaS kernels vs. Previous manual approaches. We repeatedly (1 million times) sort 8x8=64 integers for CPUs and 16x16=256 integers for MICs, respectively. The time on data load from memory to registers and store from registers to memory are included with the sort and merge in registers.

Figure 3.12 shows the performance comparison. On CPUs, both CH and IN methods use SSE instructions to handle intra-lane data-reordering, leading to extra instructions used to process inter-lane communications. Compared to our generated codes using AVX/AVX2 instructions, these solutions are relatively easier to implement, because they only need to process vector lanes one by one and there are always one unused lane for every operation, thus delivering suboptimal performance. To use the AVX/AVX2 instructions, one has to redesign their method and consider the different register length and corresponding instructions. In contrast, our solution automatically looks for the architecture-specific instructions to handle both intra- and inter-lane communications and deliver up to 3.4x speedups over these manual approaches. On MICs, the XI method adopts *mask* instructions to disable some elements for each min/max operation. These unused slots inevitably under-utilize the vector resources. The BR method, on the other hand, directly uses the expensive `permutexvar` instructions to conduct the global data-reordering. As a contrast, our code generation framework can satisfy the underlying architectures, e.g., preferring lightweight intra-lane and swizzle instructions when making the code generation. Therefore, on the KNC platform, our codes can provide up to 1.7x performance improvements over the manually optimized methods.

to port the codes using corresponding IMCI instructions on KNC, e.g., replacing `permutexvar_pd` with `permutevar_epi32` and correct parameters.

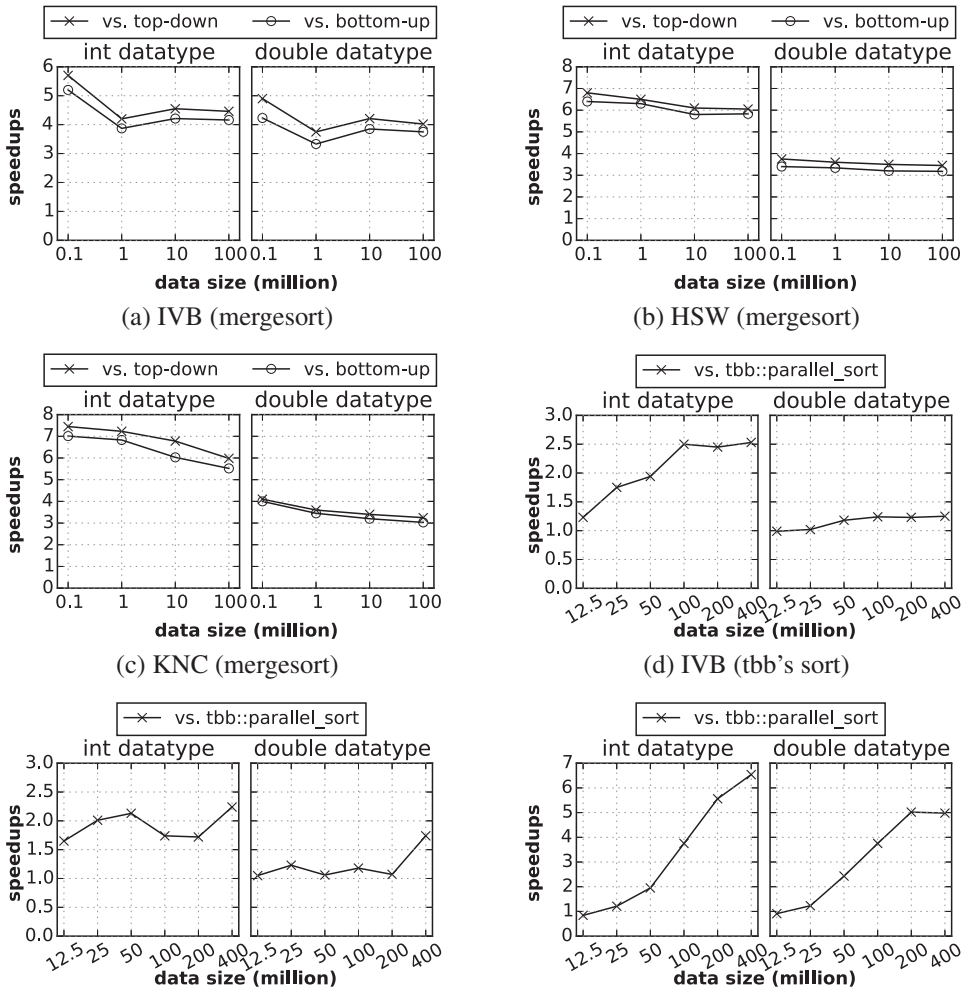
3.5.4 Comparison to Sorting from Libraries

In the section, we will evaluate the single-threaded `aspas::sort` and the multi-threaded `aspas::parallel_sort` by comparing them with their related mergesorts and various sorting tools from existing libraries.

Single-threaded ASPaS: ASPaS is essentially based on the bottom-up mergesort as the partition strategy. We first compare the single-threaded `aspas::sort` with two mergesort variants: top-down and bottom-up. The top-down mergesort recursively splits the input array until the split segments only have one element. Subsequently, the segments are merged together. As a contrast, the bottom-up mergesort, which directly works on the elements in the input array and iteratively merge them into sorted segments. For their implementation, we use the `std::inplace_merge` as the kernel to conduct the actual merging operations. Figure 3.13 (a,b,c) illustrate the corresponding performance comparison on IVB, HSW, and KNC. The bottom-up mergesort can outperform the top-down slightly due to the recursion overhead in the top-down method. The ASPaS of Integer datatype outperforms the bottom-up mergesorts by 4.3x to 5.6x, while the Double datatype provides 3.1x to 3.8x speedups.

ASPaS can efficiently vectorize the *merge* stage, even though the complexity of ASPaS merging is higher than the `std::inplace_merge` used in the bottom-up mergesort. In ASPaS, when merging each pair of two sorted segments, we fetch w elements into a buffer from each segment and then merge these $2w$ elements using the $2w$-way bitonic merging. After that, we store the first half of merged $2w$ elements back to the result, and load w elements from the segment with the smaller first element into the buffer; and then, the next round of bitonic merge will occur (ln. 18-28 in Algorithm 1). Since the $2w$-way bitonic merging network contains $2log(2w)2^{log(2w)-2}$ comparators [21], for every w elements, the total number of comparisons is $(N/w) * 2log(2w)2^{log(2w)-2} = log(2w)N$. As a contrast, the `std::inplace_merge` conducts exactly N-1 comparisons if enough additional memory is available. Therefore, the comparisons in the bottom-up mergesort are considerably less than what we use in Algorithm 1. However, because our code carries out better memory access pattern: fetching multiple contiguous data from the memory and then conducting the comparisons in registers with a cache-friendly manner, we observe better performance

of `aspas::sort` over any of the bottom-up mergesort on all three platforms in Figure 3.13 (a,b,c).

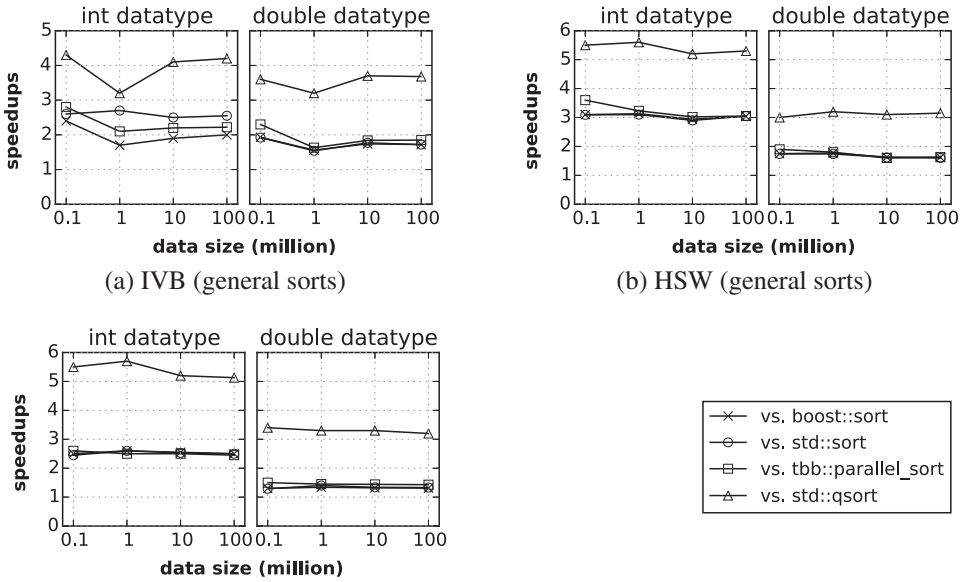


(a) IVB (mergesort)
(b) HSW (mergesort)
(c) KNC (mergesort)
(d) IVB (tbb's sort)
(e) HSW (tbb's sort)
(f) KNC (tbb's sort)

Figure 3.13: (a,b,c): `aspas::sort` vs. the top-down and bottom-up mergesorts; (d,e,f): `aspas::parallel_sort` vs. the Intel TBB parallel sort.

Then, we compare the `aspas::sort` with other existing sorting tools from widely-used libraries, including the `qsort` and `sort` from STL (libstdc++.so.6.0.19), `sort` from Boost (v.1.55), and `parallel_sort` from Intel TBB (v.4.1) (using a single thread). Figure 3.14 presents that the ASPaS codes can provide the highest performance over the other four sorts. The `aspas::sort` on the Integer array can achieve 4.2x, 5.2x, and 5.1x speedups over the `qsort` on

IVB, HSW, and KNC, respectively (`qsort` is also notorious about its function callback for every comparison.). Over the other sorting tools, it can still provide up to 2.1x, 3.0x, and 2.5x speedups. For Double datatype, the performance benefits of `aspas::sort` become 3.8x, 2.9x, and 3.1x speedups over the `qsort`, and 1.8x, 1.7x, 1.3x speedups over others on the three platforms correspondingly.



(a) IVB (general sorts)

(b) HSW (general sorts)

(c) KNC (general sorts)

(d) Legends

Figure 3.14: `aspas::sort` vs. library sorting tools.

Multi-threaded ASPaS: In Figure 3.13 (d,e,f), we compare the multi-threaded ASPaS to the Intel TBB's `parallel_sort` for a larger dataset from 12.5 to 400 million Integer and Double elements. We configure the thread numbers to the integral multiples of cores and select the one that can provide the best performance. On the three platforms, our `aspas::parallel_sort` can outperform the `tbb::parallel_sort` by up to 2.5x, 2.3x, and 6.7x speedups for the Integer datatype and 1.2x, 1.7x, and 5.0x speedups for the Double datatype.

3.5.5 Sorting Different Input Patterns

Finally, we evaluate the `aspas::sort` using different input patterns. As shown in Figure 3.15 (d), we use five input patterns defined in the previous research [49], including random, even/odd, pipe organ, sorted, and push front input. With these input patterns, we can further evaluate the performance of our generated vector codes with existing methods from widely used libraries.

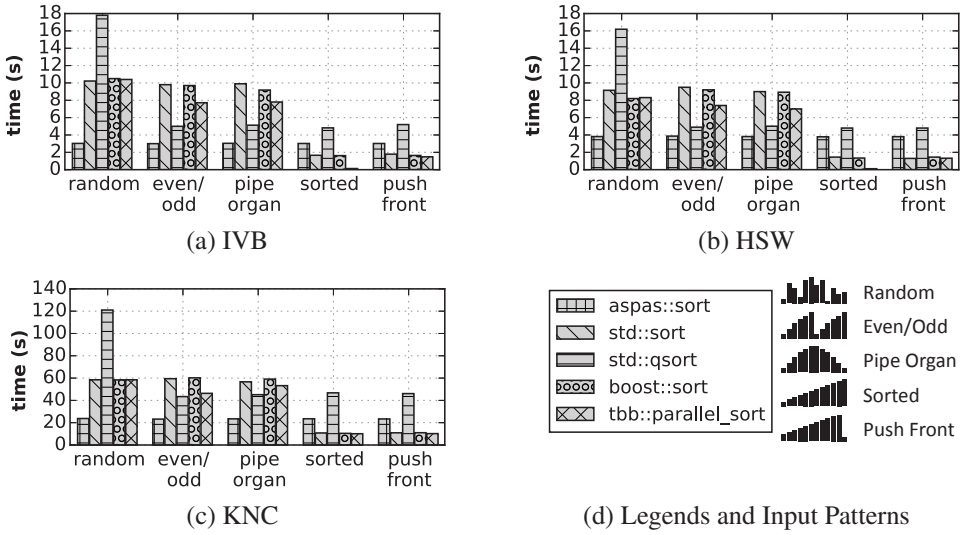


Figure 3.15: Performance of ASPaS sorting different input patterns.

In Figure 3.15 (d), we can find that the sorting tools from modern libraries can provide better performance than our generated codes for the almost sorted inputs, i.e., "sorted" and "push front". That is because these libraries can be adaptive to different patterns by using multiple sorting algorithms. For example, `std::sort` uses a combination of quick sort and insertion sort. For an almost sorted input array, `std::sort` switches from the partition of the quick sort to the insertion sort, which is good at handling the sorted input within O(n). As a contrast, our work focuses on automatically generating efficient sorting kernels for more general cases, e.g., random, even/odd, and pipe organ. At these cases, our sorting codes can yield superior performance.

3.6 Chapter Summary

In this chapter, we propose the ASPaS framework to automatically generate vectorized sorting code for x86-based multicore and manycore processors. ASPaS can formalize the sorting and merging networks to the sequences of comparing and reordering operators of DSL. Based on the characteristics of such operators, ASPaS first creates an ISA-friendly pool to contain the requisite data comparing and reordering primitives, then builds those sequences with primitives, and finally maps them to the real ISA intrinsics. Besides, the ASPaS codes can exhibit a efficient memory access pattern and thread-level parallelism. The ASPaS-generated codes can outperform the compiler-optimized ones and meanwhile yield highest performance over multiple library sorting tools on Ivy Bridge, Haswell, and Knights Corner architectures.

With the emerge of Skylake and Knights Landing architecture, our work can be easily ported to AVX-512, since the ISA subset AVX-512F contains all the permute/shuffle instructions we need for sorting. For GPUs, we will also extend ASPaS to search shuffle instructions to support fast data permutation at register level.

Chapter 4

Data-thread Binding in Parallel Segmented Sort

4.1 Introduction

Sort is one of the most fundamental operations in computer science. A sorting algorithm orders entries of an array by their ranks. Even though sorting algorithms have been extensively studied on various parallel platforms [133, 114, 89, 139], two recent trends necessitate revisiting them on throughput-oriented processors. The first trend is that manycore processors such as GPUs are more and more used both for traditional HPC applications and for big data processing. In these cases, a large amount of independent arrays often need to be sorted as a whole, either because of algorithm characteristics (e.g., suffix array construction in prefix doubling algorithms from bioinformatics [63, 154]), or dataset properties (e.g., sparse matrices in linear algebra [23, 150, 102, 103, 104, 101]), or real-time requests from web users (e.g., queries in data warehouse [175, 157, 187]). The second trend is that with the rapidly increased computational power of new processors, sorting a single array at a time usually cannot fully utilize the devices, thus grouping multiple independent arrays and sorting them simultaneously are crucial for high utilization.

As a result, the *segmented sort* that involves sorting a batch of segments of non-uniform length

concatenated in a single array becomes an important computational kernel. Although directly sorting each segment in parallel could work well on multicore CPUs with dynamic scheduling [138], applying similar methods such as "dynamic parallelism" on manycore GPUs may cause degraded performance due to high overhead for context switch [153, 165, 60, 158]. On the other hand, the distribution of segment lengths often exhibits the skewed characteristics, where a dominant number of segments are relatively short but the rest of them can be much longer. In this context, the existing approaches, such as the "one-size-fits-all" philosophy [122] (i.e., treating different segments equally) and some variants of global sort [53, 22] (i.e., traditional sort methods plus segment boundary check at runtime), may not give best performance due to load imbalance and low on-chip resource utilization.

We in this work propose a fast segmented sort mechanism on GPUs. To improve load balance and increase resource utilization, our method first constructs basic work units composed of adaptively defined elements from multiple short segments of various sizes or part of long segments, and then uses appropriate parallel strategies for different work units. We further propose a register-based sort method to support N-to-M data-thread binding and in-register data communication. We also design a shared memory-based merge method to support variable-length chunks merge via multiple warps. For the grouped short and medium work units, our mechanism does the segmented sort in the registers and shared memory; and for those long segments, our mechanism can also exploit on-chip memories as much as possible.

Using segments of uniform and synthetic power-law length on NVIDIA K80-Kepler and TitanX-Pascal GPUs, our segmented sort can exceed state-of-the-art methods in three vendor libraries CUB [122], CUSP [53] and ModernGPU [22] by up to 86.1x, 16.5x, and 3.8x, respectively. Furthermore, we integrate our mechanism with two real-world applications to confirm their efficiency. For the suffix array construction (SAC) in bioinformatics, our mechanism results in a factor of 2.3–2.6 speedup over the latest skew-SA method [154] with CUDPP [69]. For the sparse matrix-matrix multiplication (SpGEMM) in linear algebra, our method delivers a factor of 1.4–86.5, 1.5–2.3, and 1.4–2.3 speedups over approaches from cuSPARSE [121], CUSP [53], and bhSPARSE [102], respectively. The contributions of this chapter are listed as follows:

- We identify the importance of segmented sort on various applications by exploring segment length distribution in real-world datasets and uncovering performance issues of existing tools.
- We propose an adaptive segmented sort mechanism for GPUs, whose key techniques contain: (1) a differentiated method for different segment lengths to eliminate load imbalance, thread divergence, and irregular memory access; and (2) an algorithm that extends sorting networks to support N-to-M data-thread binding and thread communication at GPU register level.
- We carry out a comprehensive evaluation on both kernel level and application level to demonstrate the efficacy and generality of our mechanism on two NVIDIA GPU platforms.

4.2 Motivation

4.2.1 Segmented Sort

Segmented sort (SegSort) performs a segment-by-segment sort on a given array composed of multiple segments. If there is only one segment, the operation converts into the classical sort problem that gains much attention in the past decades. Thus sort can be seen as a special case of segmented sort. The complexity of segmented sort can be $\sum_{i=1}^{p} n_i \log n_i$[1], where p is the number of segments in the problem and n_i is length of each segment. Figure 4.1 shows an example of segmented sort, where an array stores a list of keys (integer in this case) plus an additional array *seg_ptr* used for storing head pointers of each segment.

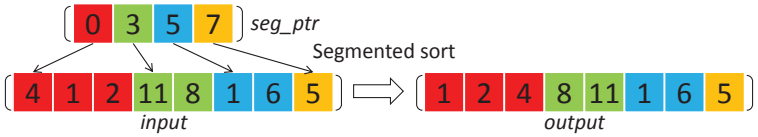
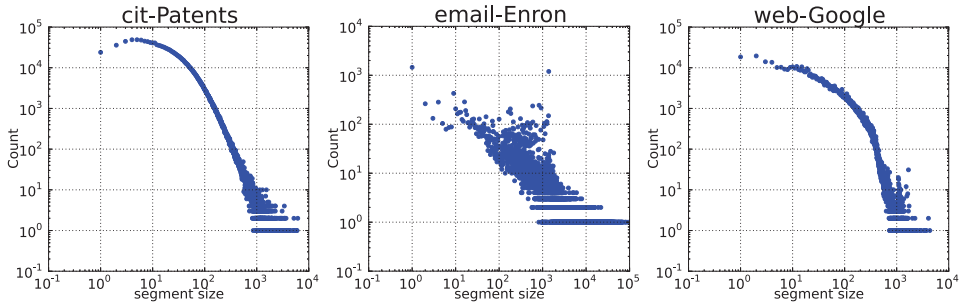


Figure 4.1: An example of segmented sort with four integer segments of various lengths pointed by *seg_ptr*.

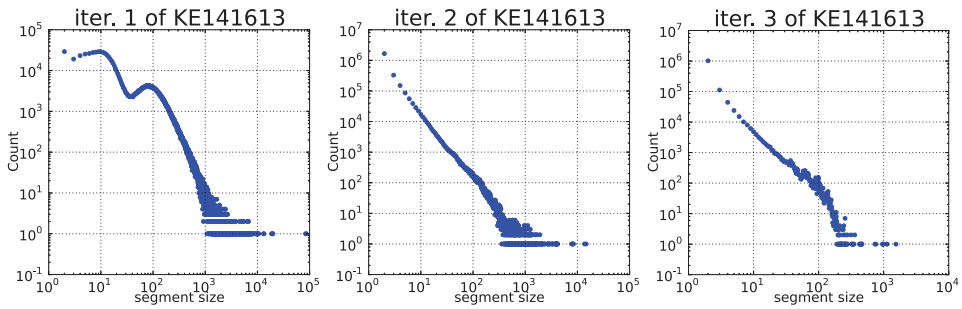
[1]For generality, we only focus on comparison-based sort in this work.

4.2.2 Skewed Segment Length Distribution

We use real-world datasets from two applications to analyze the characteristics of data distribution in segmented sort. The first application is the suffix array construction (SAC) from Bioinformatics, where the prefix doubling algorithm [63, 154] is used. This algorithm calls SegSort to sort each segment in one iteration step, and the duplicated elements will form another sets of segments for the next iteration. This procedure continues until no duplicated element exists. The second one is the sparse matrix-matrix multiplication (SpGEMM). In this algorithm, SegSort is used for reordering entries in each row by their column indices.



(a) Segments from squaring three different matrices in SpGEMM



(b) Segments from the first three iterations in SAC

Figure 4.2: Histogram of segment length changes in SpGEMM and SAC.

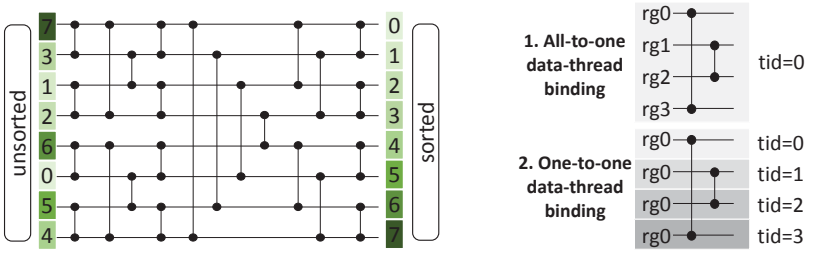
As shown in Figure 4.2, the statistics of segments derived from these two algorithms shares one feature that the small/medium segments dominate the distribution, where around 96% segments in SpGEMM and 99% segments in SAC have less than 2000 elements, and the rest can be much longer but contributes less to the number of entries in the whole problem. Such highly skewed data stems from either the input data or the intermediate data generated by the algorithm at runtime. As a result, the segments of various lengths require differentiate processing methods for

high efficiency. Later on, we will present an adaptive mechanism that constructs basic work units composed of multiple short segments of various sizes or part of long segments, and processes them in a very fine grained way for achieving load balance on manycore GPUs.

4.2.3 Sorting Networks and Their Limitations

A sorting network consisting of a sequence of independent comparisons is usually used as a basic primitive to build the corresponding sorting algorithm. Figure 4.3a provides an example of bitonic sorting network that accepts 8 random integers as input. Each vertical line in the sorting network represents a comparison of input elements. Through theses comparisons, the 8 integers are sorted.

Although a sorting network provides a view of how to compare input data, it does not give a parallel solution of how the data are distributed into multiple threads. A straightforward solution is directly mapping one element of input data to one GPU thread. However, this method will waste the computational power of GPU, because every comparison represented by a vertical line is conducted by two threads. This method also leads to poor instruction-level parallelism (ILP), since the insufficient operations per thread cannot fully take advantage of instruction pipelining. Therefore, it is important to investigate how many elements in a sorting network processed by a GPU thread can lead to best performance on GPU memory hierarchy. In this chapter, we call it the *data-thread binding* on GPUs.



(a) A 8-way bitonic sorting network (b) Data-thread bindings

Figure 4.3: One sorting network and existing strategies using registers on GPUs.

Figure 4.3b presents two examples of using data-thread binding to realize a part of sorting network shown in Figure 4.3a. Figure 4.3b-1 shows the most straightforward method of simply

conducting all the computation within a single thread [22]. This option can exhibit better ILP but at the expense of requesting too many register resources. On the contrary, Figure 4.3b-2 shows the example to bind one element to one thread [57]. Unfortunately, this method wastes computing resources, i.e., the first two threads perform the comparison on the same operands as the last two threads do. Therefore, we will investigate a more sophisticated solution that allows N-to-M data-thread binding (N elements binding to M threads) and evaluate the performance after applying this solution to different lengths of segments on different GPU architectures.

Another related but distinct issue is that even if we know the best N-to-M data-thread binding for a given segment on a GPU architecture, how to efficiently exchange data within/between threads is still challenging, especially at GPU register level. Different with the communications via the global and shared memory of GPU that have been studied extensively, data sharing through registers may require more research.

4.3 Methodology

4.3.1 Adaptive GPU SegSort Mechanism

The key idea of our SegSort mechanism is to construct relatively balanced work units to be consumed by a large amount of warps (i.e., a group of 32 threads in NVIDIA CUDA semantics) running on GPUs. Such work units can be a combination of multiple segments of small sizes, or part of a long segment split by a certain interval. To prepare the construction, we first group different lengths of segments into different bins, then combine or split segments for making the balanced work units, finally apply differentiated sort approaches in appropriate memory levels to those units.

Specifically, we categorize segments into four types of bins as shown in Figure 4.4: (1) A unit bin, which segments only contain 1 or 0 element. For these segments, we simply copy them into the output in the global memory. (2) Several warp bins, which segments are short enough. In some warp bins, a segment will be processed by a single thread, while in others, a segment will be handled by several threads, but a warp of threads at most. That way, we only use GPU registers to

sort these segments. This register-based sort is called **reg-sort** (§ 4.3.2), which allows the N-to-M data-thread binding and data communication between threads. Once these segments are sorted in registers, we write data from the registers to the global memory, and bypass the shared memory. To achieve the coalesced memory access, the sorted results may be written to the global memory in a striped manner after an in-register transpose stage (§ 4.3.4). (3) Several block bins, which consist of medium size segments. In these bins, multiple warps in a thread block cooperate to sort a segment. Besides of using the reg-sort method in GPU registers, a shared memory based merge method, called **smem-merge**, is designed to merge multiple sorted chunks from *reg-sort* in a segment (§ 4.3.3). After that, the merged results will be written into the output array from the shared memory to the global memory. As shown in the figure, the number of warps in the thread block is configurable. (4) A grid bin, designed for the sufficient long segments. For these segments, multiple blocks work together to sort and merge data. Different with the block bins, multiple rounds of **smem-merge** have to move data back and forth between the shared memory and the global memory to process these extremely long segments. In each round of calling **smem-merge**, the synchronization between multiple thread blocks is necessary: after the execution of *reg-sort* and *smem-merge* in each block, intermediate results need to be synchronized across all cooperative blocks via the global memory [163]. After that, the partially sorted data will be repartitioned and assigned to each block by using a similar partitioning method in *smem-merge*, and utilizing inter-block locality will in general further improve overall performance [91].

As shown in Figure 4.4, the binning is the first step of our mechanism and a specific requirement of segmented sort compared to the standard sort. It is crucial to design an efficient binning approach on GPUs. Such an approach usually needs carefully designed histogram and scan kernels, such as those proposed in previous research [86, 163]. We adopt a simple and efficient "histogram-scan-bin" strategy generating the bins across GPU memory hierarchy. We launch a GPU kernel which number of threads is equal to the number of segments. Each thread will process one element in the *segs_ptr* array. *Histogram*: each thread has a boolean array as the predicates and the array length is equal to the number of total bins. Then, each thread calculates its segment length from *segs_ptr* and sets the corresponding predicate to *true* if the segment length is in the

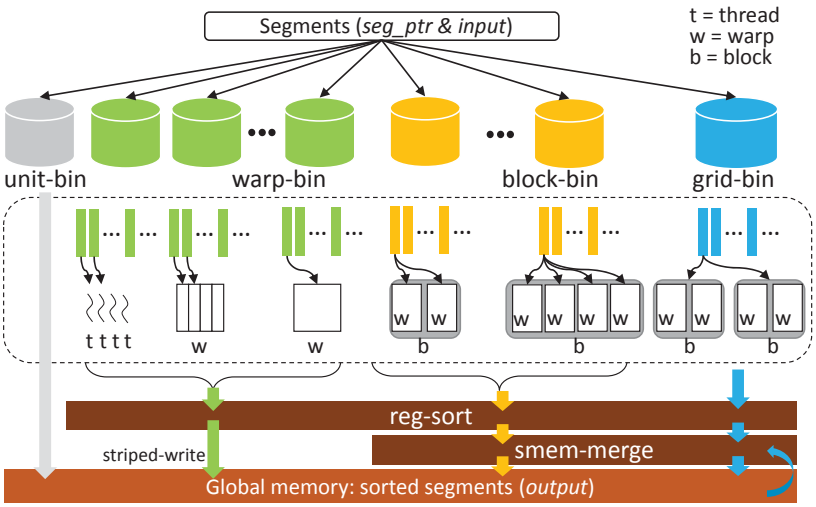


Figure 4.4: Overview of our GPU Segsort design.

current bin. After that, we use the warp vote function `__ballot()` and the bit counting function `__popc()` to accumulate the predicates for all threads in a warp for the warp-level histogram. Finally, the first thread of each warp atomically accumulates the bin sizes in the shared memory to produce the block-level histogram, and after that the first thread in each block does the same accumulation in the global memory to generate the final histogram. *Scan*: We use an exclusive scan on the histogram bin sizes to get starting positions for each bin. *Binning*: all threads put their corresponding segment IDs to positions atomically obtained from the scanned histogram. With such highly efficient designs, the overhead of grouping is limited and will be evaluated in § 4.4.2.

4.3.2 Reg-sort: Register-based Sort

Our *reg-sort* algorithm is designed to sort data in GPU registers for all bins. As a result, our method needs to support N-to-M data-thread binding, meaning M threads cooperate on sorting N elements. In order to leverage GPU registers to implement fast data exchange for sorting networks, M is set up to 32, which is the warp size. For the medium and long segments where multiple warps are involved, we still use *reg-sort* to sort each chunk of a segment and use *smem-merge* to merge these sorted chunks. On the other hand, although our method theoretically supports any value of N, N is bound to the number of registers: if N is too large, the occupancy is degraded significantly

because too many registers are used.

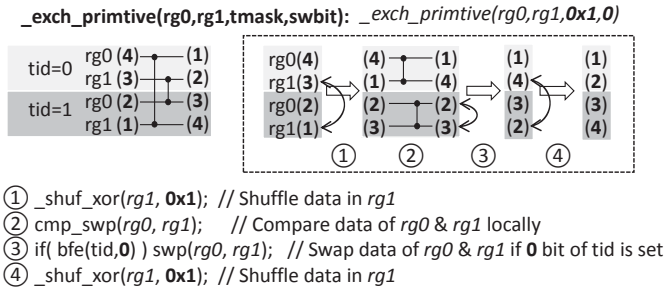


Figure 4.5: Primitive communication pattern and its implementation.

Figure 4.5 shows the data exchange primitive in the bitonic sorting network 4.3a with the details of implementation on GPU registers by using shuffle instructions. In this example, each thread holds two elements as the input: the thread 0 has 4 and 3 in its register rg0 and rg1; and the thread 1 has 2 and 1 accordingly. This situation corresponds to a 4-to-2 data-thread binding. The primitive is implemented in four steps. First, each thread needs to know the communicating thread by the parameter *tmask*. The line 12 of Algorithm 3 shows how to calculate its value, which is equal to the current cooperative thread group size minus 1. In this example, M is 2 because there are two threads in a cooperative group, and *tmask* is 1 (represented as 0x1 in the figure). This step uses the shuffle instruction `__shfl_xor(rg1, 0x1)`[2] to shuffle data in rg1: the thread 0 gets data from the rg1 of thread 1, and similar to the thread 1. This step makes the data changed from "4, 3, 2, 1" to "4, 1, 2, 3". Second, each thread will compare and swap data in rg0 and rg1 locally to change the data to "1, 4, 2, 3". After the second step, the rg0 in each thread has the smaller data and the rg1 has the larger one. Thus, the third step is necessary to exchange the data in thread 1 to make the smaller data in its rg1 and the larger data in it rg0 for the following shuffle, because the shuffle instruction can only exchange data in registers having the same variable name, i.e., rg1. In a more general case where there are more threads are involved, we use the parameter *swbit* to control which threads need to execute this local swap operation. The line 13 of Algorithm 3 shows how to calculate the value of *swbit*, which is equal to $\log \mathsf{coop\_thrd\_size} - 1$. In this case, the current cooperative thread group size is 2, the *swbit* is 0, and the thread 1 will do the swap when

[2]The *shuffle* operation `__shfl_xor(v, mask)` lets the calling thread x obtain register data `v` from thread `x ^ mask`, and the operation `__shfl(v, y)` lets the calling thread `x` fetch data `v` from thread `y`.

`bfe(1, 0)` returns 1. After the third step, we get "1, 4, 3, 2". Fourth, we shuffle again on rg1 to move the larger data of thread 0 to thread 1 and the smaller data of thread 1 to thread 0, and then get the output "1, 2, 3, 4".

Based on the primitive, we extend two patterns of the N-to-M binding to implement the bitonic sorting network, which are (1) *exch_intxn*: The differences of corresponding data indices for comparisons decrease as the register indices increase in the first thread. (2) *exch_paral*: The differences of corresponding data indices for comparisons keep consistent as the register indices increase in the first thread. The left hand sides of Figure 4.6a and Figure 4.6b show these two patterns. In these figures, each thread handling k elements, where $k = N/M$ of the N-to-M data-thread binding. These two patterns can be easily constructed from the primitive *exch_primitive* by simply swapping corresponding registers in each thread locally as shown in the right hand side of Figure 4.6a and Figure 4.6b. Different with these two patterns involving the inter-thread communication, a third pattern in the N-to-M data-thread binding only has the intra-thread communication. As shown in Figure 4.6d, we can directly use the compare and swap operation for the implementation without any shuffle-based inter-thread communication. We call this pattern *exch_local*. This pattern can be an extreme case of N-to-M binding, i.e., N-to-1, where the whole sorting network is processed by one thread. All of these three patterns are used in Algorithm 3 to implement a general N-to-M binding and the corresponding data communication for the bitonic sorting network.

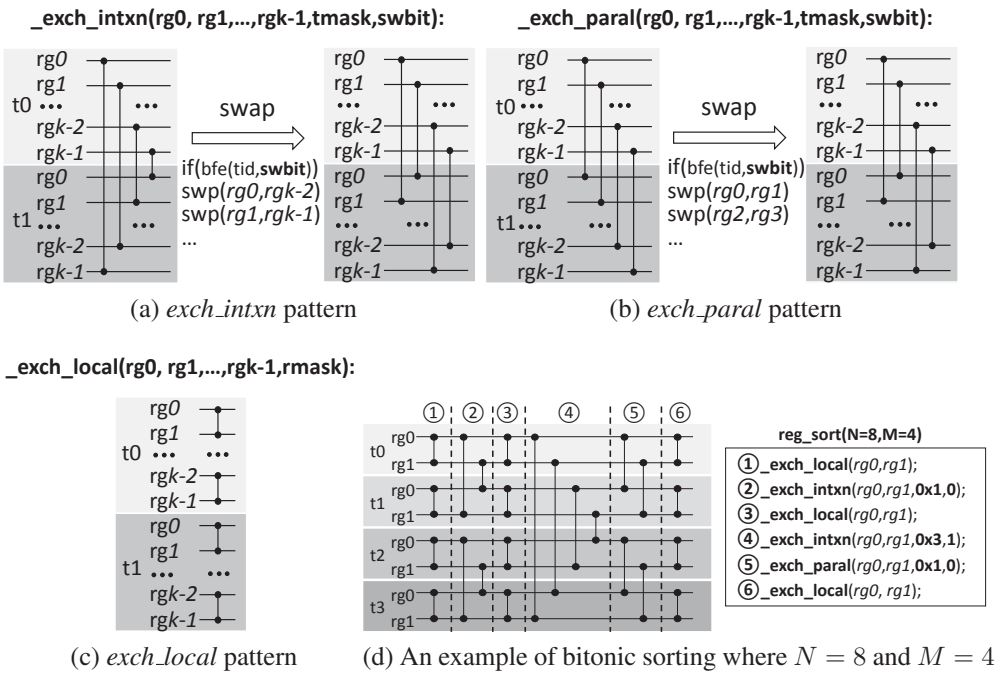
The pseudo-codes are shown in Algorithm 3, which essentially combine *exch_intxn* and *exch_paral* patterns (line 14 and line 28) in each iteration, and use *exch_local* when no inter-thread communication is needed. In each step where all comparisons can be performed in parallel, we group data elements as *coop_elem_num*, representing the maximum number of groups that the comparisons can be conducted without any interaction with elements in other group (line 4 and line 16); and the *coop_elem_size* represents how many elements in each group. For example, in the step 1 of Figure 4.6d, the data is put into 4 groups, each of which has 2 elements. Thus, *coop_elem_num* is 4 and *coop_elem_size* is 2. In the step 4 of this figure, there is only 1 group with the size of 8 elements. Thus, *coop_elem_num* is 1 and *coop_elem_size*is 8. Similarly, the algorithm groups threads into *coop_thrd_num* and *coop_thrd_size* for each step to represent the maximum number of

Algorithm 3: Reg-sort: N-to-M data-thread binding and communication for bitonic sort

```
   /* segment size N, thread number M, workloads per thread wpt = N/M, regList is a group of
      wpt registers.                                                                          */
1  int p = (int) log N;
2  int pt = (int) log M;
3  for l ←p; l >= 1; l-- do
4      int coop_elem_num = (int)pow(2, l − 1);
5      int coop_thrd_num = (int)pow(2, min(pt, l − 1));
6      int coop_elem_size = (int)pow(2, p − l + 1);
7      int coop_thrd_size = (int)pow(2, pt − min(pt, l − 1));
8      if coop_thrd_size == 1 then
9          int rmask = coop_elem_size − 1;
10         _exch_local(regList, rmask);
11     else
12         int tmask = coop_thrd_size − 1;
13         int swbit = (int) log coop_thrd_size − 1;
14         _exch_intxn(regList, tmask, swbit);
15     for k ← l + 1; k <=p; k + + do
16         int coop_elem_num = (int)pow(2, k − 1);
17         int coop_thrd_num = (int)pow(2, min(pt, k − 1));
18         int coop_elem_size = (int)pow(2, p − k + 1);
19         int coop_thrd_size = (int)pow(2, pt − min(pt, k − 1));
20         if coop_thrd_size == 1 then
21             int rmask = coop_elem_num − 1;
22             rmask = rmask − (rmask >> 1);
23             _exch_local(regList, rmask);
24         else
25             int tmask = coop_thrd_num − 1;
26             tmask = tmask − (tmask >> 1);
27             int swbit = (int)log coop_thrd_size − 1;
28             _exch_paral(regList, tmask, swbit);
```

cooperative thread groups and the number of threads in each group. For example, the step 1 of Figure 4.6d has 4 cooperative thread groups and each group has 1 thread. Thus, *coop_thrd_num* is 4 and *coop_thrd_size* is 1. In contrast, the step 4, *coop_thrd_num* is 1 and *coop_thrd_size* is 4 that means the 4 threads need to communicate with each other to get required data for the comparisons in this step.

If there is only one thread in a cooperative thread group (line 8 and line 20), the algorithm will switch to the local mode *exch_local* because the thread already has all comparison operands. Once there are more than one thread in a cooperative thread group, the algorithm uses *exch_intxn* and *exch_paral* patterns and calculates corresponding `tmask` and `swbit` to determine the thread for the communication pair and the thread that needs the local data rearrange aforementioned in the previous paragraph. In the step 1 of this figure, where *coop_thrd_size* is 1 and *exch_local* is executed, the algorithm calculates `rmask`, which controls the local comparison on registers (line 9). In the step 4, where *coop_elem_size* is 8 and *coop_thrd_size* is 4, all 8 elements will be



(a) *exch_intxn* pattern (b) *exch_paral* pattern

(c) *exch_local* pattern (d) An example of bitonic sorting where $N = 8$ and $M = 4$

Figure 4.6: Generic *exch_intxn*, *exch_paral* and *exch_local* patterns, shown in (a,b,c). The transformed patterns can be easily implemented by using primitives. A code example of Algorithm 3 for is shown in (d).

compared across all 4 threads. In this case, the `tmask` is 0x3 (line 12) and `swbit` is 1 (line 13). In the step 5 where the *exch_paral* pattern is used, the algorithm calculates *coop_elem_size* is 4 and *coop_thrd_size* is 2. Thus, the `tmask` is 0x1 and `swbit` is 0. Note that although our design in Algorithm 3 is for bitonic sorter, our ideas are also applicable to other sorting networks by swapping and padding registers based on the primitive pattern.

4.3.3 Smem-merge: Shared Memory-based Merge

As shown in Figure 4.4, for medium and large sized segments in the block bins and grid bin, multiple warps are launched to handle one segment and the sorted intermediate data from each warp need to merge. The *smem-merge* algorithm is designed to merge such data in the shared memory.

Our *smem-merge* method enables multiple warps to merge chunks having different numbers of elements. We assign first m warps with $x\text{-}warp\_size$ and the last m' warps with $y\text{-}warp\_size$

to keep load balance between warps as possible as we can. Inside each warp, we also try to keep balance among cooperative threads in the merge. We design a searching algorithm based on the MergePath algorithm [68] to search the splitting points to divide the target segments into balanced partitions for each threads.

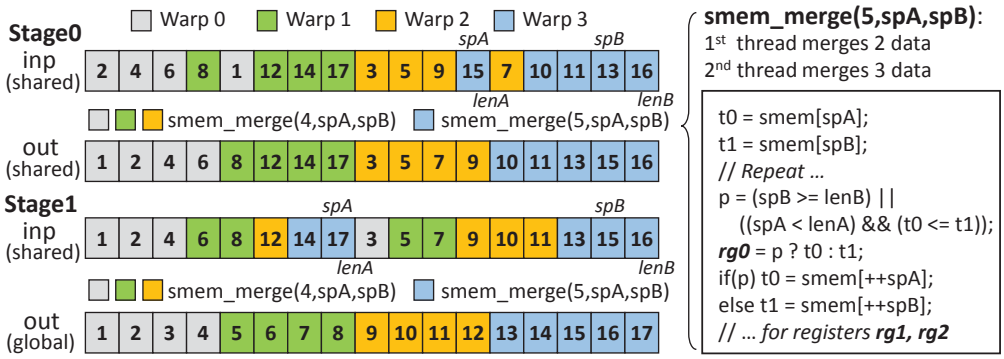


Figure 4.7: An example of warp-based merge using shared memory.

Figure 4.7 shows an example of *smem-merge*. In this case, there are 4 sorted chunks in the shared memory belonging to a segment. They have 4, 4, 4, and 5 elements, respectively. We assume each warp has 2 threads. As a result, the first 3 warps will merge 4 elements, and each thread in these warps will merge two elements; while the last warp will work on 5 elements, and the first thread will merge 2 elements and second thread will merge 3 elements. In the figure, the splitting points `spA` and `spB` for the second thread of warp 3 (in blue color) are computed by the MergePath algorithm. The right part of the figure shows the merge codes executed by this thread that process 3 elements. Our merge method first loads data from `spA` and `spB` to two temporary registers `t0` and `t1`. By checking if `spA` and `spB` is out-of-bound and comparing `t0` and `t1`, the merge algorithm selects the smaller element to fill first result register `rg0`. The algorithm continues loading the next element to fill `t0` or `t1` from corresponding chunks pointed by `spA` or `spB`, until assigned number of elements is encountered. After that, the merged data in registers, e.g., *rg0, rg1* for first thread, and *rg0, rg1, rg2* for the second thread, will be stored back to shared memory for another iteration of merge. As shown in the figure, two iterations of *smem-merge* are used to merge four chunks belonging to one segment.

4.3.4 Other Optimizations

Load data from global memory to registers: we decouple the data load from global memory to registers and the actual segmented sort in registers. Our data load kernel uses successive threads to load contiguous data for coalesced memory access. Although we don't keep the original global indices of input data in registers, that doesn't matter because the input data is unsorted and the global indices are not critical for the following sort routine.

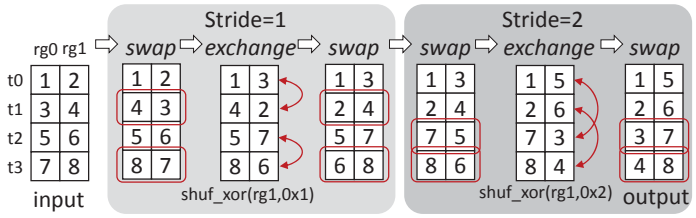


Figure 4.8: An example of in-register transpose.

Store data from registers to global memory: when the segments in the warp bins are sorted, we directly store them back to global memory without the merge stage. However, because the sorted data may distributed into different registers of threads in a warp, directly storing them back will lead to uncoalesced memory access. When the number of elements per thread grows, the situation will become even worse. Therefore, as well as keeping the direct store back method, which is labeled as *orig* in our evaluation, we design the **striped write** method, which is labeled as *strd* in the evaluation. We implement an in-register transpose method to scatter the sorted data in registers of threads. The transpose method starts from shuffling registers by the stride of 1, then doubles the stride to 2, and finishes the shuffles until $\log 32 = 5$ iterations, where 32 is the warp size. After that, the successive threads can write data to global memory in a cyclic manner for coalesced memory access. Figure 4.8 shows an example of using 4 threads to transpose data in their two registers. This example shows that the number of iterations for the in-register transpose depends on the number of threads but not on the number of elements each thread has[3]. As a result, after $\log 4 = 2$ iterations, the successive elements are scattered to these threads. In the evaluation, we will investigate the best scenarios for *orig* and *strd*, considering the *orig* method has the uncoalesced memory access

[3]The number of elements each thread has will determine how many shuffle instructions are needed in each iteration.

problem, while the *strid* method has the in-register transpose overhead.

4.4 Performance Results

We conduct the experiments on two generations of NVIDIA GPUs K80-Kepler and TitanX-Pascal. Tab. 4.1 lists the specifications of the two platforms. The input dataset is two arrays holding key and value pairs separately. The total dataset size is fixed at 2^{28} and segment numbers are varied accordingly to the target segment sizes. We report the throughput in the experiments equal to $2^{28}/t$ pairs/s, where t is the execution time.

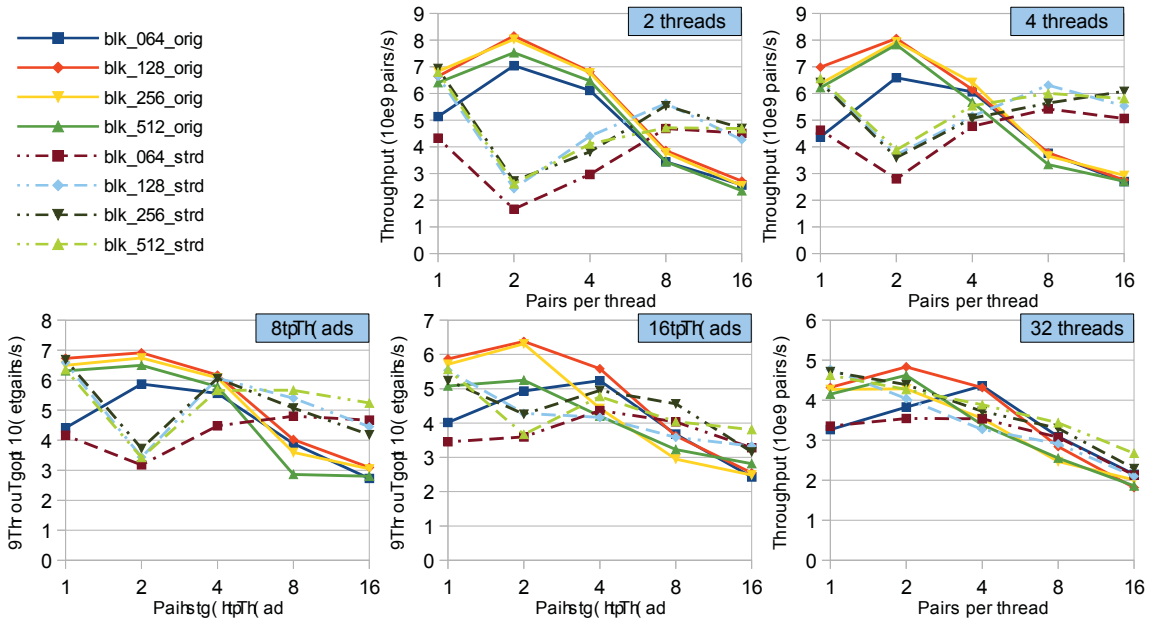
Table 4.1: Experiment Testbeds

	Tesla K80 (Kepler-GK210)	**TitanX (Pascal-GP102)**
Cores	2496 @ 824 MHz	3584 @ 1531 MHz
Register/L1/LDS per core	256/16/48 KB	256/16/48 KB
Global memory	12 GB @ 240 GB/s	12 GB @ 480 GB/s
Software	CUDA 7.5	CUDA 8.0

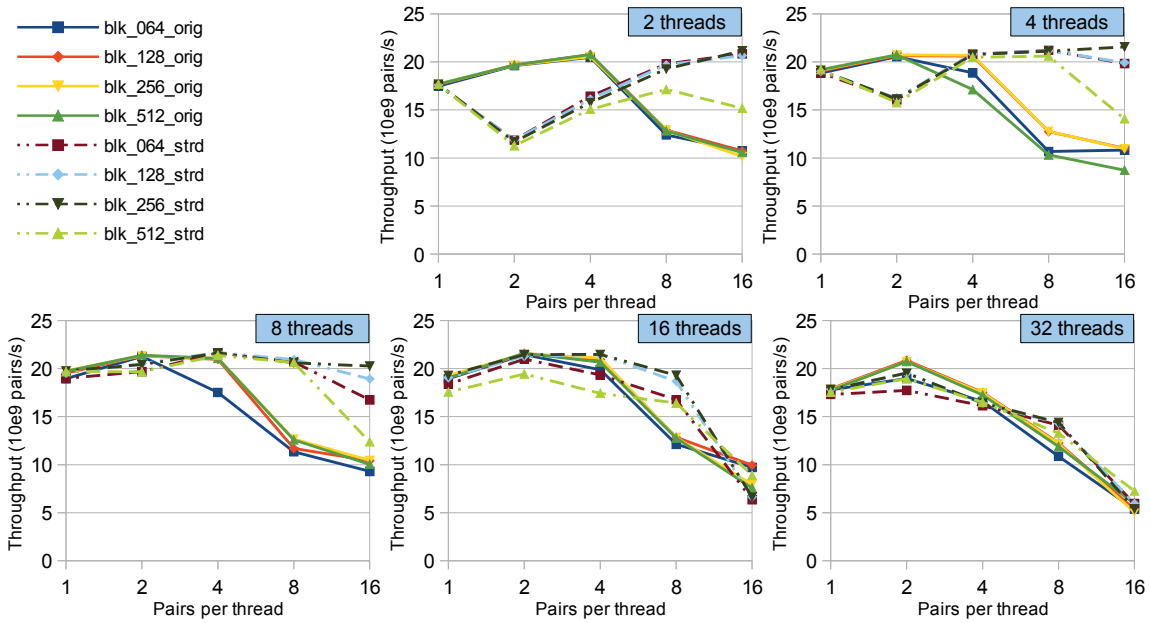
4.4.1 Kernel Performance

For the reg-sort kernels, we alternate the thread group size M in $\{2, 4, 8, 16, 32\}$. At the same time, each thread varies its bound data N/M in $\{1, 2, 4, 8, 16\}$ (N/M is labeled as pairs per thread *ppt* in figures). We use the maximum bound data of 16 because we observe the performance deteriorates obviously for higher numbers than 16. Thus, the target segment size N is a variable number ranging from 2 (i.e., 2 threads each bind 1 pair) to 512 (i.e., 32 threads each bind 16 pairs). Since reg-sort might exhaust register resources, we also vary the block sizes as 64, 128, 256, and 512 for maximizing occupancy. Figure 4.9 shows the diversified performance numbers of reg-sort kernels, which actually demonstrates that choosing a single solution for all segments would lead to suboptimal performance even among GPUs from the same vendor.

In Figure 4.9, we notice that the impact of data-thread binding policies varies widely depending on the GPU devices. For example, when the segment size N equals to 16, the possible candidates are 2(threads):8(ppt), 4:4, 8:2, and 16:1. On K80-Kepler GPU, the highest perfor-



(a) K80-Kepler



(b) TitanX-Pascal

Figure 4.9: Performance of reg-sort routines with different combinations of data-thread binding policies, write methods, and block sizes.

mance is given by 8:2, achieving 30% speedups over the slowest policy of 2:8. In contrast, the TitanX-Pascal GPU shows very similar performance for these policies. This insensitivity to regis-

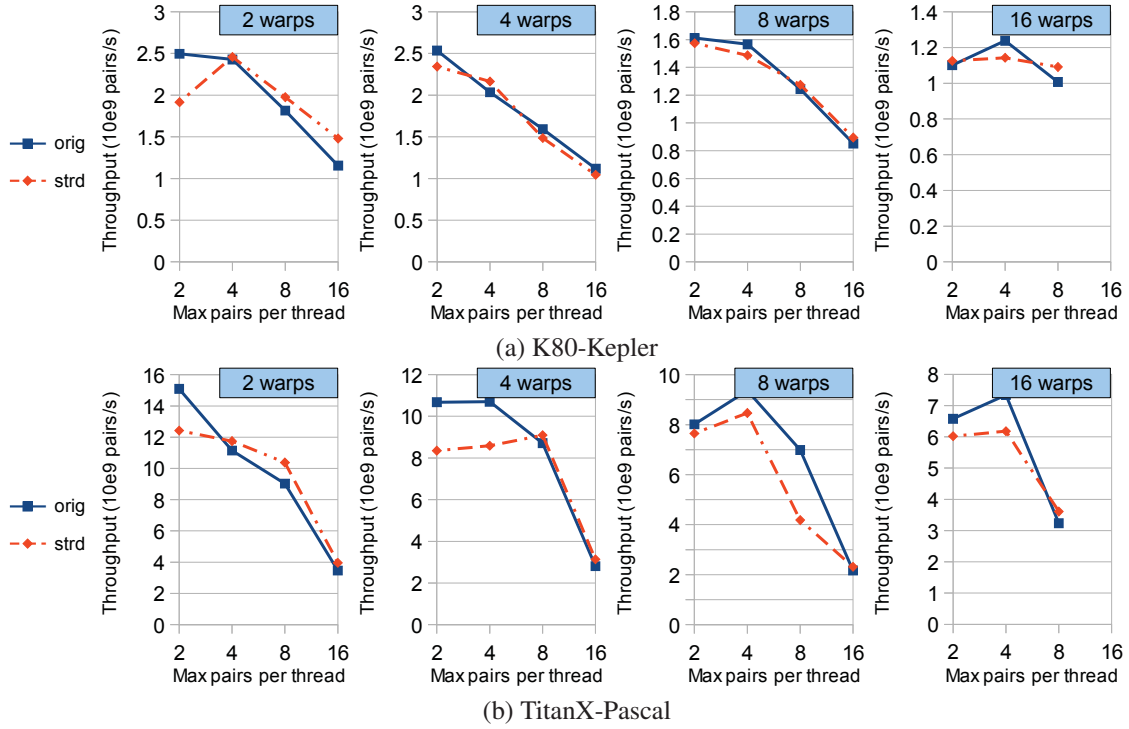
ter resources exhibited on Pascal architecture can contribute to its larger available register files per thread. Note, the policies of each thread binding only 1 pair is equivalent to the method proposed in [57]. This method actually wastes computing resources, because each comparison over two operands is conducted twice by two threads, resulting in suboptimal performance numbers.

The striped write method (*strd*) in reg-sort kernels is particularly effective when each thread binds more than 4 pairs. This is because when the ppt is small ($<$4), consecutive threads can still access almost consecutive memory locations for coalesced memory transaction. However, larger ppt indicates the thread access locations are highly scattered, causing inefficiently memory transaction instead. In this case, the striped write method is of great necessity. On the other hand, the block size also has the effect on the performance and the optimal one is usually achieved by using 128 or 256 threads.

For the smem-merge kernels, Figure 4.10 shows the performance numbers of changing the number of cooperative warps and ppt. The warp numbers are in $\{2, 4, 8, 16\}$, while the ppt varies among $\{2, 4, 8, 16\}$. In this scenario, the target segment size N ranges from 128 (i.e., 2 warps each merge 32x2 pairs) to 4096 (i.e., 16 warps each merge 32x8 pairs). Differed from reg-sort kernels, the best data-thread binding policies are more consistent for smem-merge on the two devices. For example, to merge 1024-length segment, both devices prefer to use 8 warps with 4 ppt. Considering memory access, the striped method provides similar performance with original method, which directly exploits random access of shared memory in order to achieve coalesced transaction on global memory. Note, in our implementation, we carefully select shared memory dimensions and sizes to avoid band conflicts.

Now, we can select best kernels for different segments with length of the power of two. Other segments can be handled by directly padding them to the nearest upper power of two in reg-sort kernels. For smem-merge kernels, we use different ppt for different warps to minimize padding values. Since our method is based on the pre-defined sorting networks, the characteristics of input datasets will cause negligible to the selection of best kernels. Moreover, for each GPU, we only need to conduct the offline selection once. The results show that we only need 13 bins to handle all the segments. The first 8 and 9 bins use reg-sort to handle up to 256 pairs on Kepler GPU and

512 on Pascal respectively. Then, other segments less than 2048 can be handled by smem-merge kernels using 3 and 2 bins on the two platforms. Finally, our grid-bin kernels can efficiently sort the segments longer than 2048, which are all assigned to the last bin.



(a) K80-Kepler

(b) TitanX-Pascal

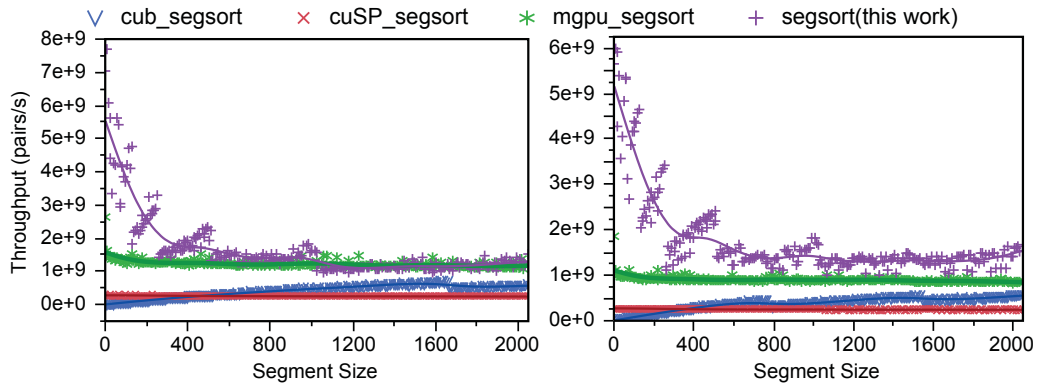
Figure 4.10: Performance of smem-merge routines with different combinations of data-thread binding policies and write methods. The results for 16 warps with 16 ppt are not available due to exhaustion of shared memory resources.

4.4.2 Segmented Sort Performance

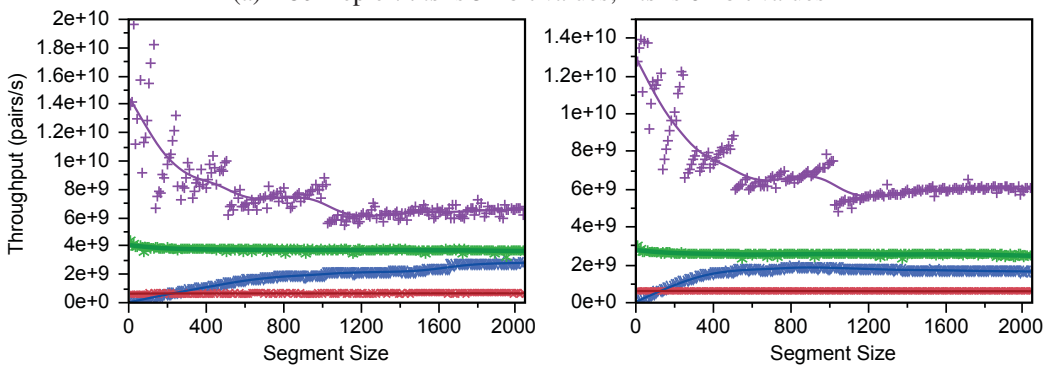
We conduct performance comparison of our best kernels over three existing tools: (1) *cusp-segsort* from CUSP [53] library that extends the segment pointer *seg_ptr* to form an another layer of primary keys, attaches it to the input keys and values, and performs the global sort from Thrust library [71] on the new data structure; (2) *cub-segsort* [122] that assigns each block to order a segment and uses radix sort scheme; and (3) *mgpu-segsort* [22] that evolves from the global merge sort with runtime segment delimiter checking, thus can ensure the sort only occurs within segment.

Datasets of uniform distribution: We test a batch of uniform segments to evaluate the per-

formance of our segsort kernels, which are plotted in Figure 4.11. Since the cusp-segsort and mgpu-segsort are designed from the global sort, their performance is determined by the total input size. This "one-size-fit-all" philosophy ignores the characteristics of the segments and thus shows the plateau performance. For the short segments (<256 on Kepler and <512 on Pascal), our segsort can achieve an average of 13.4x and 3.1x speedups over cusp-segsort and mgpu-segsort respectively on Kepler. These speedups rise to 15.5x and 3.2x on Pascal. For the other segments, our segsort provides an average of 5.6x and 1.2x improvements on Kepler (10.0x and 2.1x on Pascal). The performance improvements are mainly from our reg-sort and smem-merge, which minimize shared/global memory transactions.



(a) K80-Kepler: *lhs* is 32-bit values; *rhs* is 64-bit values



(b) TitanX-Pascal: *lhs* is 32-bit values; *rhs* is 64-bit values

Figure 4.11: Performance of different segsort over segments with uniform distribution.

In contrast, although cub-segsort conducts a more "real" segmented sort with each block working on one segment, this strategy falls short when the segments are of great amount and of short lengths. The maximum number of segments can be processed in parallel in cub-segment

is only 65535 (limitation of gridDim.x), which requires multiple rounds of calling if the segment number is too large. Furthermore, assigning a block to handle one segment may waste computing resources severely, especially when the segments are very short. Thus, our segsort can achieve an average of 211.2x and 30.4x speedups on Kepler and Pascal devices respectively. As the segment size increases, we can still keep 3.7x and 3.2x average improvements on the two platforms. Note, the staircase-like performance of our segsort is caused by the padding used in our method.

Datasets of power-law distribution: In this test, we use a collection of synthetic power-low data. Since segment lengths are non-uniform, we include binning overhead and use overall wall time for our segsort. The data generation tool is from PowerGraph [66] and its generated samples follow a Zipf distribution [7]. The equation of $P(l) \propto l^{-\alpha}$ shows the probability of segments with length l is proportional to $l^{-\alpha}$, where α is a positive number. This implies that the higher α will result in high skewness to shorter segments. For different segment bounds, we vary the skewness to test our segsort. We vary α from 0.1 to 1.6 (with stride of 0.1) and limit maximum segment size from 50 to 2000 (with stride of 50). Therefore, each method is tested with 640 sampling points of different parameter configurations.

Figure 4.12 plots the speedups of our segsort over the existing tools on both 32-bit and 64-bit values. For cusp-segsort and mgpu-segsort, we fix the total key-value pairs as 2^{28}. However, for the cub-segsort, we set the segment number to 65535. Otherwise, multiple rounds of calling are required. Figure 4.12(a,b) show the speedups of our segsort over the cub-segsort. Because of high cost for radix sort on short segments, our segsort can provide significant speedups, reaching up to 63.3x and 86.1x (top-left corner). For longer segment lengths with power-law distribution, our method can keep 1.9x speedups on both GPU devices due to better load balancing strategy.

Compared to cusp-segsort in Figure 4.12(c,d), our segsort achieves up to 12.5x and 16.5x improvements on Kepler and Pascal, and compared with mgpu-segsort in Figure 4.12(e,f), our method gets up to 3.0x and 3.8x performance gains. In both situations, we can notice that the top-left corners are blue, indicating less speedups (vs. cusp-segsort) or similar performance (vs. mgpu-segsort). This is because that the condition of $\alpha = 1.6$ and segment bound of 50 make almost all the segments to be 1. Thus, the segmented sorts become a copying procedure and exhibit the

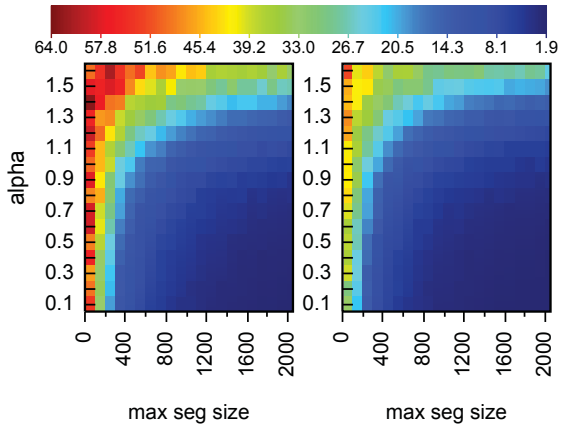
similar performance. On the other hand, we observe that the Pascal shows higher performance benefits over Kepler. The reasons are two-fold: faster atomic operations for binning and larger register files for sorting. Moreover, for the 64-bit values, we usually get higher performance gains compared to 32-bit values. This is mainly because we can handle the permutation indices more efficiently in the registers rather than the shared memory or global memory. To evaluate the binning overhead, we calculate the arithmetic mean for the ratio of binning to overall kernel time, which are only 5.9% for Kepler and 3.4% for Pascal.

4.5 SegSort in Real-World Applications

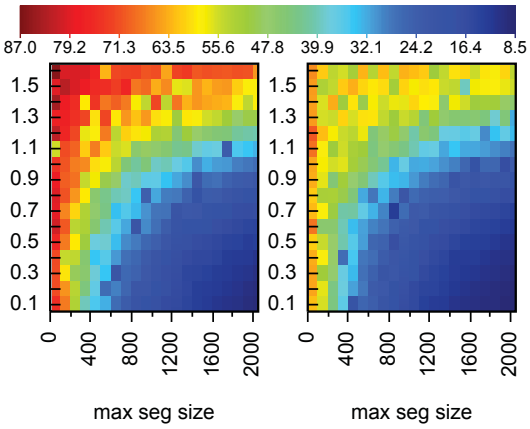
In order to further evaluate our approach, we choose two real-world applications, characterized by skewed segment distribution: (i) The suffix array construction to solve problems of pattern matching, data compression, etc. in text processing and bioinformatics [154, 63]. (ii) The sparse general matrix-matrix multiplication (SpGEMM) to solve graph and linear solver problems, e.g., sub-graphs, shortest paths, and algebraic multigrid [148, 30, 23]. We use our approach to optimize the applications and compare the results with state-of-the-art tools, i.e., skew/DC3-SA [154, 69], ESC(CUSP) [23, 53], cuSPARSE [121], and bhSPARSE [102].

4.5.1 Suffix Array Construction

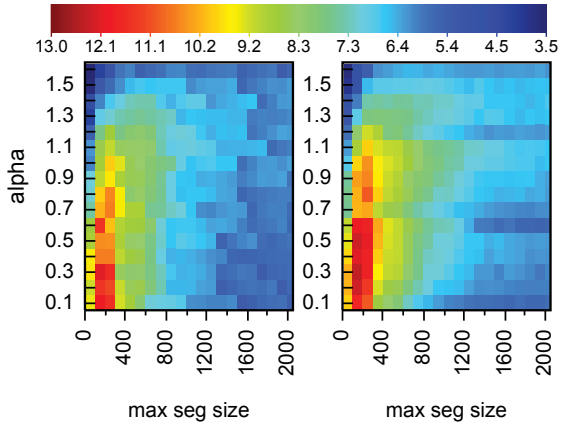
The suffix array stores lexicographically sorted indices of all suffixes of a given sequence. Our approach for the suffix array construction is based on the *prefix doubling* algorithm [109] with the computational complexity of $O(Nlog(N))$, where N is the length of input sequence. The main idea is that we can deduce the orders of two same $2h$-size strings S_i and S_j, if the orders of all h-size strings are known, which is stored in an unfinished suffix array h-SA. For example, we treat S_i as two concatenated h-size prefixes S_{ia} and S_{ib}. Similarly, S_j is split into two S_{ja} and S_{jb}. Then, by looking up the known h-SA, the comparison rule for S_i and S_j becomes: if the prefix S_{ia} differs from S_{ja}, we can directly determine the order of S_i and S_j accordingly; otherwise, we need to check the order of S_{ib} and S_{jb}. That way, if they are different, the order of S_i and S_j can also be



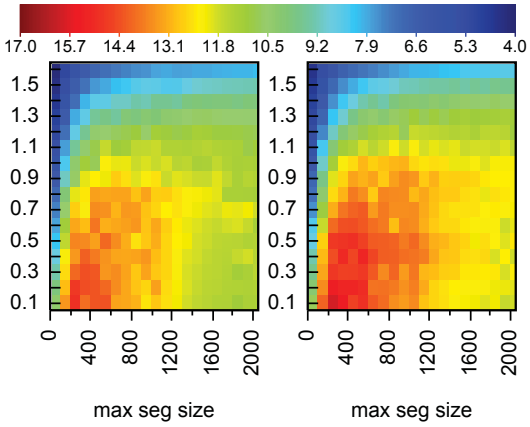
(a) K80-Kepler: speedups over cub_segsort (32- and 64-bit values)



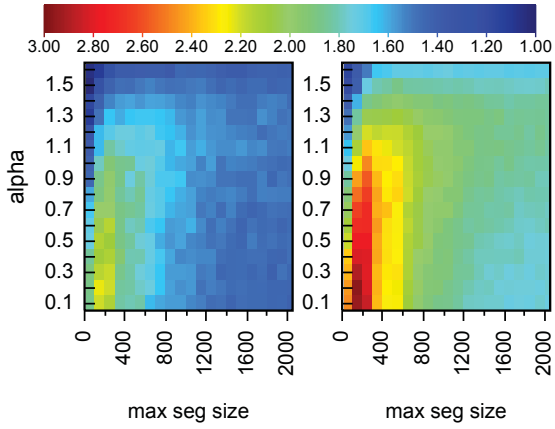
(b) TitanX-Pascal: speedups over cub_segsort (32- and 64-bit values)



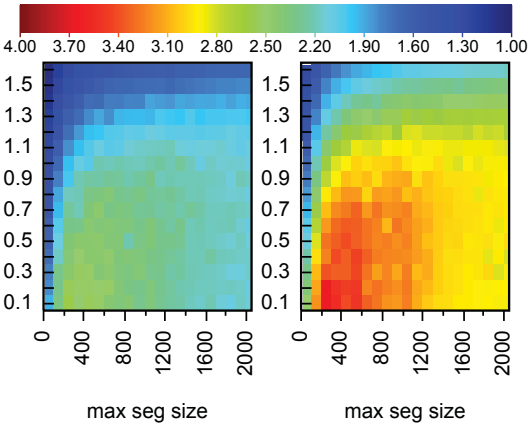
(c) K80-Kepler: speedups over cusp_segsort (32- and 64-bit values)



(d) TitanX-Pascal: speedups over cusp_segsort (32- and 64-bit values)



(e) K80-Kepler: speedups over mgpu_segsort (32- and 64-bit values)

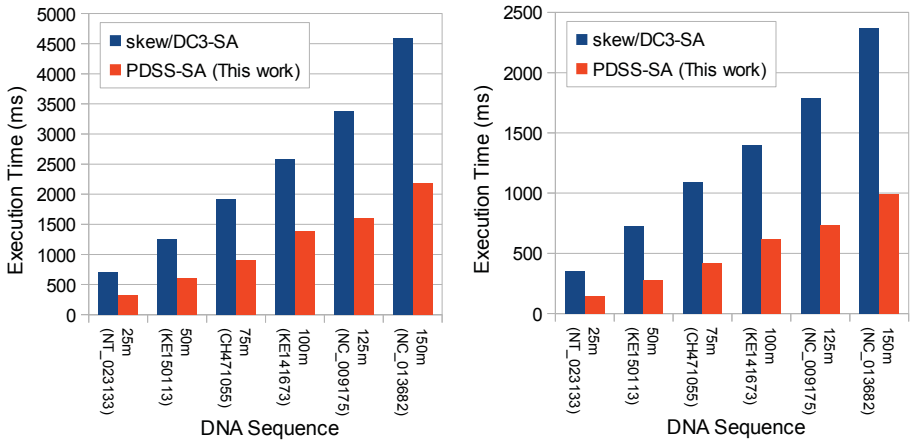


(f) TitanX-Pascal: speedups over mgpu_segsort (32- and 64-bit values)

Figure 4.12: Segmented sort v.s. existing tools over segments of power-law distribution.

induced. However, if they are same, we have to mark S_i and S_j unsolved and put them under the same category (i.e., updating h-SA to $2h$-SA with position i and j storing the same value). The ordering will proceed for $O(log(N))$ iterations by doubling the prefix length. This is a segmented sort with the customized comparison, and a segment corresponds to a category that contains a group of suffixes whose orders are not determined.

We use our method to optimize the prefix doubling, i.e., *PDSS-SA*, and use the baseline from the cuDPP library [69], which is based on the DC3/skew algorithm on GPUs [154]. Figure 4.13 presents the performance comparison over six DNA sequences of different lengths from NCBI NR datasets [115]. Our PDSS-SA can provide up to 2.2x and 2.6x speedups over the baseline on the K80-Kepler and TitanX-Pascal platforms, respectively. The improvement can be explained in two folds. First, although the baseline takes a linear time algorithm, the prefix doubling exhibits more ease of performing parallelization, e.g., global radix sort, prefix sum, etc. Second, for the dominant kernels (taking up 30% to 60% of total time), we use efficient segmented sort to handle the considerable amounts of short segments iteratively, while the baseline uses more expensive sort and merge kernels recursively.



(a) K80-Kepler (b) TitanX-Pascal

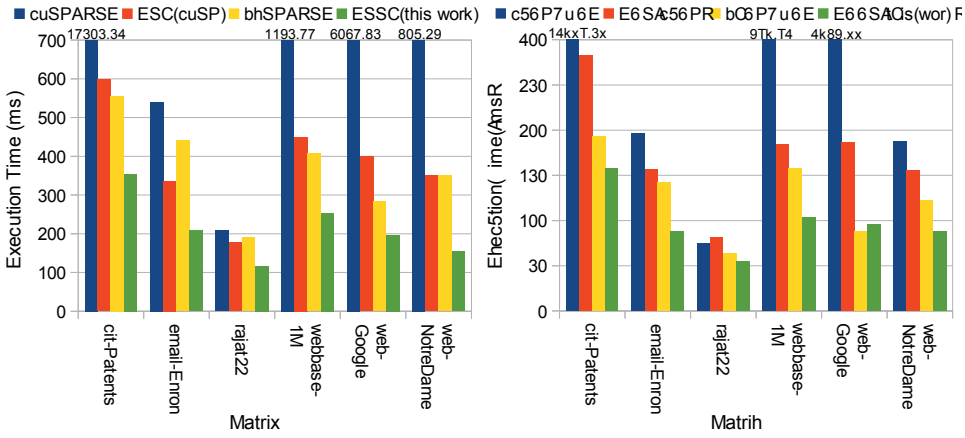
Figure 4.13: Performance of suffix array construction using our segmented sort.

4.5.2 Sparse Matrix-Matrix Multiplication

The SpGEMM operation multiplies a sparse matrix A with another sparse matrix B and obtains a resulting sparse matrix C. This operation may be the most complex routine in sparse basic linear algebra subprograms because all the three involved matrices are sparse. The *expansion*, *sorting* and *compression* (ESC) algorithm developed by Bell et al. [23] is one of several representative methods that aim to utilize wide SIMD units for accelerating SpGEMM on GPUs [23, 121, 102]. The ESC method includes three stages: (1) expanding all candidate nonzero entries generated by the necessary arithmetic operations into an intermediate sparse matrix $\widehat{C}$, (2) sorting $\widehat{C}$ by its indices of rows and columns, and (3) compressing $\widehat{C}$ into the resulting matrix C by fusing entries with duplicate column indices in each row.

We use the ESC SpGEMM in the CUSP library [53] as the baseline, and replace its original sort in the second stage of its ESC implementation with our segmented sort by mapping row-column information of the intermediate matrix $\widehat{C}$ to segment-element data in the context of segmented sort. Since we have segmented sort instead of sort in the ESC method, we call our method ESSC (*expansion*, *segmented sorting* and *compression*). Note that to better understand the effectiveness of segmented sort for SpGEMM, all other stages of the SpGEMM code remain unchanged. We select six widely used sparse matrices *cit-Patents*, *email-Enron*, *rajat22*, *webbase-1M*, *web-Google* and *web-NotreDame* from the University of Florida Sparse Matrix Collection [56] as our benchmark suite. Squaring those matrices will generate a $\widehat{C}$ including rows in power-law distribution. We also include two state-of-the-art SpGEMM methods in cuSPARSE [121] and bhSPARSE [102] into our comparison.

Figure 4.14 plots the performance of the four participating methods. It can be seen that our ESSC method achieves up to 2.3x and 1.9x speedups over the ESC method on the Kepler and Pascal architectures, respectively. The performance gain is from the fact that the sorting stage takes up to 85% (from 45%) overall cost of ESC SpGEMM, and our segmented sort is up to 8x (from 3.2x) faster than the sort method used in CUSP. Compared with the latest libraries cuSPARSE and bhSPARSE, our ESSC method brings performance improvement of up to 86.5x and 2.3x, respectively. The performance gain is mainly from the irregularity of the row distribution of $\widehat{C}$.



(a) K80-Kepler (b) TitanX-Pascal

Figure 4.14: Performance of SpGEMM using our segmented sort.

4.6 Chapter Summary

In this chapter, we have presented an efficient segmented sort mechanism that adaptively combine or split segments of different sizes for load balanced processing on GPUs, and have proposed a register-based sort algorithm with N-to-M data-thread binding and in-register communication for fast sorting networks on multiple memory hierarchies. The experimental results illustrate that our mechanism is greatly faster than existing segmented sort methods in vendor support libraries on two generations of GPUs. Furthermore, our approach improves overall performance of applications SAC from bioinformatics and SpGEMM from linear algebra.

Chapter 5

SIMD Operations in Sequence Alignment

5.1 Introduction

The pairwise sequence alignment algorithms, e.g., Smith-Waterman (SW) [140] and Needleman-Wunsch (NW) [116], are important computing kernels in bioinformatics applications ([132, 142, 105]) to quantify the similarity between pairs of DNA, RNA, or protein sequences. This similarity is captured by a matching score, which indicates the minimum number of deletion, insertion, or substitution operations with penalty or award values to transform one sequence to another. To boost their performance on modern multi- and many-core processors, it is crucial to utilize the vector processing units (VPU), which essentially conduct the single instruction, multiple data (SIMD) operations. However, the strong data dependencies among neighboring cells prevent such algorithms from taking advantage of compiler auto-vectorization. Thus, programmers need to explicitly vectorize their code or even resort to writing assembly code to attain better performance. 5codes.

The manual vectorization of such algorithms often relies on two strategies: (1) *iterate* [62], [142], [105]: partially ignore the dependencies in one direction, vectorize computations, and may compensate the results by using multiple rounds of corrections; (2) *scan* [87]: completely ignore the dependencies in one direction, vectorize computations, and recalculate the results with "weighted scan" operations and another round of correction. Either strategy has its own bene-

fits depending on selected algorithms (e.g., SW or NW), gap systems (linear or affine), and input sequences.

There are two main challenges facing programmers. First, the manual vectorization requires huge coding efforts to handle the idiosyncratic vector instructions. For applications having complex data dependencies, the expert knowledge of vector instruction sets and proficient skills to organize vector instructions is necessary to achieve desired functionality. Moreover, current vector ISAs evolve very fast and some versions are not backwards compatible [127]. Porting existing vectorized codes to another platforms becomes a boring and tedious task. Second, even the highly optimized vector codes may not get the optimal performance at the application level. For the pairwise sequence alignment, the combinations of algorithms, vectorization strategies, configurations (gap penalty systems), and input sequences at runtime may lead to significantly variable performance. It increases the complexity to optimize applications on modern multi- and many-core processors. Therefore, looking for a way to get around these obstacles is of great importance.

In this chapter, we propose a framework AAlign to automatically vectorize pairwise sequence alignment algorithms across ISAs. Our framework takes sequential algorithms, which need to follow our generalized paradigm for the pairwise sequence alignment, as the input and generate vectorized computing kernels as the output by using the formalized vector code constructs and linking to the platform-specific vector primitives. Two vectorizing strategies are formalized as the striped-iterate and striped-scan in our framework. In addition, a hybrid mechanism is introduced to take advantage of both of them. That means the hybrid mechanism can automatically switch between the striped-iterate and striped-scan based on the context of runtime, and then provide better performance than the basic mechanisms.

The major contributions of our work include the following. First, we propose the AAlign framework that can automatically generate parallel codes for pairwise sequence alignment with combinations of algorithms, vectorizing strategies, and configurations. Second, we identify the existing vectorizing strategies cannot always provide the optimal performance even the codes are highly vectorized and optimized. As a result, we design a hybrid mechanism to take advantages of two vectorizing strategies. Third, using AAlign, we generate various parallel codes for the

combinations of algorithms (SW and NW), vectorizing strategies (striped-iterate, striped-scan, and hybrid), and configurations (linear and affine gap penalty systems) on two x86-based platforms, i.e., the Advanced Vector eXtension (AVX2) supported multicore and the Initial Many Core Instructions (IMCI) supported manycore.

We conduct a serial of evaluations of the generated vector codes. Compared to the optimized sequential codes on Haswell CPU, our codes using the striped-scan can deliver 4 to 6.2-fold speedups, while switching to the striped-scan, our codes can provide 4.7 to 10-fold speedups. The vector codes continue showing performance advantages on Intel MIC, and can achieve 9.1 to 16-fold speedups using striped-scan and 9.5 to 25.9-fold speedups using striped-iterate over the optimized sequential counterparts, respectively. We also compare the proposed hybrid mechanism with the striped-iterate and striped-scan mechanisms, and demonstrate the hybrid mechanism can achieve better performance on both platforms. After wrapping our vector codes with the multi-threading, we compare our codes using the hybrid vectoring strategy with the highly optimized sequence alignment tools SWPS3 [142] on CPU and SWAPHI [105] on MIC. While aligning the given query sequences to a whole database, our codes can achieve up to 2.5-fold speedup over SWPS3 on CPU and 1.6-fold speedup over SWAPHI on MIC.

5.2 Motivation and Challenges

This section describes a brief overview of the pairwise sequence alignment algorithms and the motivation for this work.

5.2.1 Pairwise Sequence Alignment Algorithms

The pairwise sequence alignment is to quantify the best-matching score between piecewise or whole region of two input sequences of DNA, RNA, or protein. Specifically, the alignment uses the edit distance to describe how to transform one sequence into another by using minimum number of predefined operations, including insertion, deletion, and substitution, with associate penalty or award. One common technique is the dynamic programming using tabular computations shown

in Figure 5.1. If the input sequences are query Q_m with m characters and subject S_n with n characters, we need a $m * n$ table T, and every cell $T_{i,j}$ in the table stores the optimal score of matching the substring Q_i and S_j. To assist in the computation, we define three additional tables: $L_{i,j}$, $U_{i,j}$, $D_{i,j}$ denoting the optimal scores of matching with substring Q_i and S_j but ending with the insertion, deletion, and substitution respectively. We can derive:

$$T_{i,j} = max(L_{i,j}, U_{i,j}, D_{i,j}) \tag{5.1}$$

Figure 5.1 also shows the data dependencies. Visually, $L_{i,j}$, $U_{i,j}$, $D_{i,j}$ rely on its left, upper, diagonal neighbors. Although the algorithm takes $O(m * n)$ time and space complexity, by using the double-buffering technique shown in the two solid rectangles of the figure, we lower the space complexity to $O(m)$ assuming the computation goes along the Q_m.

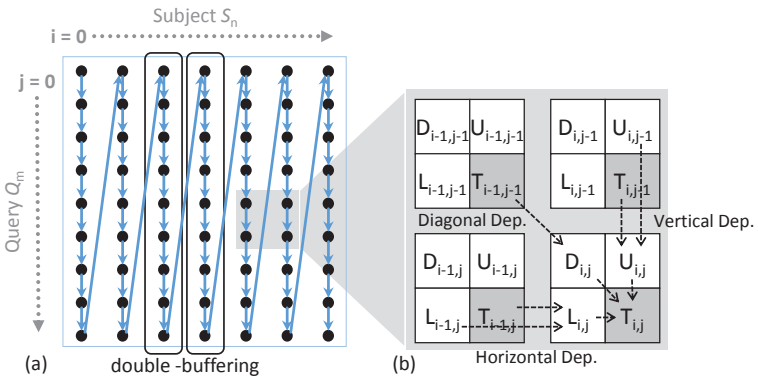


Figure 5.1: Data dependencies in the alignment algorithms using dynamic programming.

There are two major classes of pairwise sequence alignment algorithms, i.e. the local and global alignment. For the global alignment, the Needleman-Wunsch algorithm [116] can find the best-matching score regarding the entire sequences. For the local alignment, the Smith-Waterman algorithm [140] can quantify the optimal score regarding the partial regions. Both algorithms have multiple variants by using linear or affine gap penalties. We will show the generalized paradigm for the pairwise sequence alignment algorithms in § 5.3.

5.2.2 Challenges

Algorithm 4 shows the sequential code of SW with the affine gap penalty system. Though writing the sequential code is relatively simple, vectorizing such an algorithm is nontrivial due to the strong data dependencies among the neighbors shown in Figure 5.1.

Algorithm 4: Sequential Smith-Waterman following the paradigm (§ 5.3)

```
   /* GAP_OPEN and GAP_EXT are constants; BLOSUM62 is a substitution matrix; ctoi is a
      user-defined function to map given character to the index number in the substitution
      matrix                                                                          */
1  for i ←0; i < n+1; i++ do
2  |  T_{0,i} = U_{0,i} = L_{0,i} = 0;
3  for j ←0; j < m+1; j++ do
4  |  T_{j,0} = U_{j,0} = L_{j,0} = 0;
5  for i ←1; i < n+1; i++ do
6  |  for j ←1; j < m+1; j++ do
7  |  |  L_{i,j} = max(L_{i-1,j} + GAP_EXT, T_{i-1,j} + GAP_OPEN);
8  |  |  U_{i,j} = max(U_{i,j-1} + GAP_EXT, T_{i,j-1} + GAP_OPEN);
9  |  |  D_{i,j} = T_{i-1,j-1} + BLOSUM62_{ctoi(Q_{j-1}),ctoi(S_{i-1})};
10 |  |  T_{i,j} = max(0, L_{i,j}, U_{i,j}, D_{i,j});
11 // resultant score is the maximum value in T
```

We already introduce the two vectoring strategies to re-construct the data dependencies in § 5.1. We describe the major differences in this section: (1) *iterate* [62], **partially** ignores the vertical dependencies in Figure 5.1, and processes the vertical cells simultaneously along the column. This round of computations only ensures a part of the results are correct, leading to potentially multiple rounds of corrections. (2) *scan* [87], originally designed for GPU, **completely** ignores the vertical dependencies at the beginning. The vertical cells can be processed in a SIMD way, giving us the preliminary results. After that, a parallel max-scan operation will be conducted on the preliminary results, and the scan results will be applied to correct the results in another round of computation. The fundamental difference in these two strategies is in the correction: *iterate* may not need any correction, or finish the correction with one or several steps of re-computations once reach convergence, while *scan* will always take two rounds of re-computations, i.e., the scan on all vertical cells and then a round of much lighter correction.

Comparing the vector codes in Algorithm 5 and Algorithm 6[1] to the sequential code in Algorithm 4, we can find writing vector codes involves expert knowledge of the algorithms and the

[1]Although we use our formalized codes as the examples, the hand-written vector codes presented in previous research, e.g., [62], are similar to ours.

platform-specific ISAs, even though the detailed low-level intrinsics are hidden by our formalized codes. As a result, the first question we want to answer is whether we can automatically vectorize these types of applications with multiple combinations of parameters.

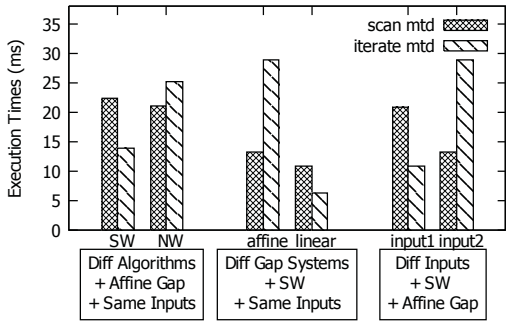


Figure 5.2: Example of comparing two vectorizing strategies under various conditions on MIC (the cases are from § 5.5).

On the other hand, the differences in the two strategies indicate they would have their own benefits. Figure 5.2 takes some evaluation numbers from § 5.5 to show our another motivation: because the algorithms, configurations, and input sequences at runtime can affect the performance and no one combination can always provide best performance, the second question in this chapter is whether we can design a mechanism to automatically select the favorable vectorization strategies at runtime.

5.3 Generalized Pairwise Alignment Paradigm

In the section, we present our generalized paradigm for the pairwise sequence alignment algorithms with adjustable gap penalties. Any sequential codes following the paradigm can be processed by our framework to generate real vector codes.

$$T_{i,j} = max \begin{cases} 0 \\ max_{0 \leqslant l < j}(T_{i,l} + \theta_{i,l} + \sum_{k=l+1}^{j} \beta_{i,k}) \\ max_{0 \leqslant l < i}(T_{l,j} + \theta'_{l,j} + \sum_{k=l+1}^{j} \beta_{k,j}) \\ T_{i-1,j-1} + \gamma_{i,j} \end{cases} \tag{5.2}$$

In the paradigm in Equation(2), the T is the working-set table and $T_{i,j}$ stores the suboptimal score. 0 is optional and used only in local alignment. $\theta_{i,l}$ ($\theta'_{l,j}$) is the gap penalty of initiating a gap at the position l of Q_m (S_n). $\beta_{i,k}$ ($\beta'_{k,j}$) is the gap penalty of continuing a gap at the position k of Q_m (S_n). $\gamma_{i,j}$ is the substitution score of matching base j of Q_m and base i of S_n. In bioinformatics, the substitution scores are usually from the scoring matrix, such as BLOSUM62. Both $\theta_{i,l}$ ($\theta'_{l,j}$) and $\beta_{i,k}$ ($\beta'_{k,j}$) can be configured to be either constants or variables. By using the dynamic programming, one can use three assistant symbols, i.e., $U_{i,j}$, $L_{i,j}$, $D_{i,j}$, to represent the influence from $T_{i,j}$'s upper, left, and diagonal neighbors. Therefore, the paradigm is equivalent to Equation (3-6).

$$T_{i,j} = max \begin{cases} 0 \\ U_{i,j} \\ L_{i,j} \\ D_{i,j} \end{cases} \quad (3)$$

$$U_{i,j} = max \begin{cases} U_{i,j-1} + \beta_{i,j} \\ T_{i,j-1} + \theta_{i,j-1} + \beta_{i,j} \end{cases} \quad (4)$$

$$L_{i,j} = max \begin{cases} L_{i-1,j} + \beta'_{i,j} \\ T_{i-1,j} + \theta'_{i-1,j} + \beta'_{i,j} \end{cases} \quad (5)$$

$$D_{i,j} = T_{i-1,j-1} + \gamma_{i,j} \quad (6)$$

Now, we can fit the real algorithms into the paradigm. *Smith-Waterman*: Because it is a local alignment algorithm, we need to keep 0 as the initial. If we simply use the linear gap penalty, the $\theta_{i,l}$ ($\theta'_{l,j}$) is set to 0 and $\beta_{i,k}$ ($\beta'_{k,j}$) is the gap penalty value. If we use affine gap penalty, the $\theta_{i,l}$ ($\theta'_{l,j}$) is the gap open penalty value and $\beta_{i,k}$ ($\beta'_{k,j}$) is the gap extension penalty value. If these parameters are variables, other gap penalty systems can be used. *Needleman-Wunsch*: Because it is a global alignment algorithm, we don't need the 0. The configuration of other parameters is similar with the SW. Actually, line 7 to line 10 in Algorithm 4 follow the paradigm with necessary initialization codes in line 1 to line 4.

5.4 AAlign Framework

The AAlign framework adopts the "striped-iterate" and "striped-scan" as the basic vectorization strategies. We make a few modifications to the original methods derived from [62] and [87] to fit our framework. Figure 5.3 illustrates the overview of AAlign. The framework can accept any kind of sequential codes following our generalized paradigm in § 5.3. After analyzing the Abstract

Syntax Tree (AST) of the sequential code, AAlign can obtain the required information, such as the type of the given alignment algorithm and the selected gap penalty system. Then, AAlign will input the information to the "vec code constructs" which are formalized according to the aforementioned vectorizing strategies. Finally, the framework can generate real codes by using proper vector modules. These modules include primitive vector operations whose implementation is ISA-specific.

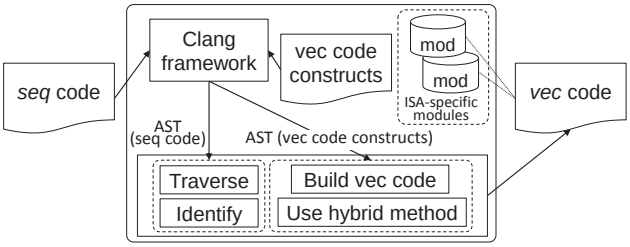


Figure 5.3: High-level overview of the structure of AAlign framework.

5.4.1 Vector Code Constructs

In this section, we will first describe the SIMD-friendly data layout used in AAlign. Based on it, we will present two vector code constructs containing the vector modules (§ 5.4.3) and the configurable parameters (§ 5.4.4).

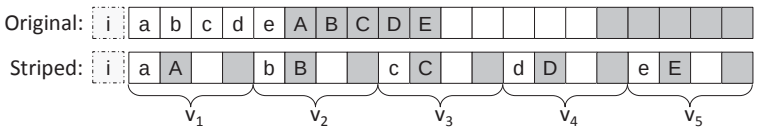


Figure 5.4: The original and SIMD-friendly striped layouts.

Striped layout: AAlign always conducts the tabular computation along the query sequence Q_m. After loading the data from the same column in Figure 5.1 to the buffer, AAlign transforms the data layout to the striped format, which is SIMD-friendly because the data dependency among adjacent elements are eliminated. Figure 5.4 shows the data layouts before and after the striped transformation. In the original buffer, we have 20 elements from the same column of the tabular; and each element depends on its preceding neighbor (the vertical direction in Figure 5.1). If we

load the elements directly into five vectors, the data dependencies will hinder efficient vector operations. By rearranging the buffer into the striped format, dependent elements are distributed to different vectors, making the interaction happening among vectors rather than within vectors.

Algorithm 5: Vector code constructs for striped-iterate

```
   /* m is the aligned length of Q, n is the length of S, k is number of vectors in Q, equal
      to m/vec_len. If the linear gap penalty system is taken, the AAlign will ignore the
      asterisked statements                                                              */
1  vec vT_dia, vT_left, vT_up, vT;
2  vec vT_max = broadcast(INT_MIN);
3  vec vGapT_left = broadcast(GAP_LEFT);
4  vec vGapT_up = broadcast(GAP_UP);
5  *vec vL, vU;
6  *vec vGapL = broadcast(GAP_LEFT_EXT);
7  *vec vGapU = broadcast(GAP_UP_EXT);
8  *vec vZero = broadcast(0);
9  for i ←0; i < n; i++ do
10     vT_dia = rshift_x_fill(arr_T1 + (k − 1) ∗ vec_len, 1, INIT_T);
11     vT_up = set_vector(m, INIT_T, GAP_UP);
12     vT_up = add_vector(vT_up, vGapT_up);
13     *vU = set_vector(m, INIT_U, GAP_UP_EXT);
14     *vU = add_vector(vU, vGapU);
15     *vU = max_vector(vU, vT_up);
16     for j ←0; j < k; j++ do
17        vT_dia = add_array(prof + ctoi(S_i) ∗ m + j ∗ vec_len, vT_dia);
18        vT_left = add_array(arr_T1 + j ∗ vec_len, vGapT_left);
19        *vL = add_array(arr_L + j ∗ vec_len, vGapL);
20        *vL = max_vector(vL, vT_left);
21        *store_vector(arr_L + j ∗ vec_len, vL);
22        vT = max_vector(vT_dia, MAX_OPRD);
23        store_vector(arr_T2 + j ∗ vec_len, vT);
24        vT_max = max_vector(vT_max, vT);
25        vT_dia = load_vector(arr_T1 + j ∗ vec_len);
26        vT_up = vT;
27        vT_up = add_array(vT_up + vGapT_up);
28        *vU = add_vector(vU + vGapU);
29        *vU = max_vector(vT_up, vU);
30     REC_UP = rshift_x_fill(REC_UP, 1, REC_FILL);
31     int j = 0;
32     vT = load_vector(arr_T2 + j ∗ vec_len);
33     while influence_test(REC_UP, REC_CRT) do
34        vT = max_vector(vT, REC_UP);
35        store_vector(arr_T2 + j ∗ vec_len, vT);
36        vT_max = max_vector(vT_max, vT);
37        REC_UP = add_vector(REC_UP, REC_UP_GAP);
38        if ++j >= k then
39           REC_UP = rshift_x_fill(REC_UP, 1, REC_FILL);
40           j=0;
41        vT = load_vector(arr_T2 + j ∗ vec_len);
42     swap(arr_T1, arr_T2);
```

Striped-iterate: This vectorizing strategy is based on [62]. The modified vector code constructs are shown in Algorithm 5. We use two m-element buffers $\mathsf{arr}_{\mathsf{T1}}$ and $\mathsf{arr}_{\mathsf{T2}}$ to store the best-matching scores. Additionally, a m-element buffer $\mathsf{arr}_{\mathsf{L}}$ stores the scores denoting best-matching with ending gap in Q. The scores denoting best-matching with ending gap in S are stored in the

vector register T_{up} or vU if affine gap penalty system is taken. In this strategy, we first partially ignore the data dependencies within the buffer (along the Q) and use the predefined vectors (line 11 and line 13) to set lower bounds. In the predefined vectors (T_{up} or vU), only first elements come from the real initialization expressions ($INIT\_T$ and $INIT\_U$), while other elements are derived from them and corresponding gap penalties (GAP_UP and GAP_UP_EXT). As a result, the first round of preliminary computations (line 16 to line 29) only ensures the first elements in each vector are correct (a-e cells in Figure 5.4).

We need to correct the results if the updated predefined vectors affect the results (line 33). The re-computations of correction (line 34 to line 41) will take at most $\mathsf{vec}_{\mathsf{len}}$-1 times to ensure all the other elements in the vectors are correct. After that, we continue the for loop (line 9) to process the next character in S, which corresponds to another column in Figure 5.1.

Algorithm 6: Vector code constructs for striped-scan

```
   // m is the aligned length of Q, n is the length of S, k is number of vectors in Q, equal
      to m/vec_len. If the linear gap penalty system is taken, the AAlign will ignore the
      asterisked statements
1  vec vT_dia, vT_left, vT_up, vT;
2  vec vT_max = broadcast(INT_MIN);
3  vec vGapT_left = broadcast(GAP_LEFT);
4  *vec vL;
5  *vec vGapL = broadcast(GAP_LEFT_EXT);
6  *vec vZero = broadcast(0);
7  for i ←0; i < n; i++ do
8      vT_dia = rshift_x_fill(arr_T1 + (k − 1) * vec_len, 1, INIT_T);
9      for j ←0; j < k; j++ do
10         vT_dia = add_array(prof + ctoi(S_i) * m + j * vec_len, vT_dia);
11         vT_left = add_array(arr_T1 + j * vec_len, vGapT_left);
12         *vL = add_array(arr_L + j * vec_len, vGapL);
13         *vL = max_vector(vL, vT_left);
14         *store_vector(arr_L + j * vec_len, vL);
15         vT = max_vector(vT_dia, MAX_OPRD);
16         store_vector(arr_T2 + j * vec_len, vT);
17         vT_dia = load_vector(arr_T1 + j * vec_len);
18     wgt_max_scan(arr_T2, arr_Scan, m, INIT_T, GAP_UP_EXT, GAP_UP);
19     for j ←0; j < k; j++ do
20         vT_up = load_vector(arr_Scan + j * vec_len);
21         vT = load_vector(arr_T2 + j * vec_len);
22         vT = max_vector(vT, vT_up);
23         vT_max = max_vector(vT_max, vT);
24         store_vector(arr_T2 + j * vec_len, vT);
25     swap(arr_T1, arr_T2);
```

Striped-scan: The scan strategy in AAlign is based on the GPU method [87]. We modify it by using the striped format on x86-based platforms, shown in Algorithm 6. Similar with the striped-iterate, we define three m-element buffers $\mathsf{arr}_{\mathsf{T1}}$, $\mathsf{arr}_{\mathsf{T2}}$, and $\mathsf{arr}_{\mathsf{L}}$. In addition, an extra

buffer $\mathsf{arr}_{\mathsf{scan}}$ is used to store the scan results. In this strategy, we first completely ignore the data dependencies within the buffer (along the Q) to do the tentative computation (line 9 to line 17). Unlike the striped-iterate, we conduct "weighted" scan over the tentative results $\mathsf{arr}_{\mathsf{T2}}$ and store the scan results to $\mathsf{arr}_{\mathsf{scan}}$ (line 18). Finally, we use the values in $\mathsf{arr}_{\mathsf{scan}}$ to correct the results (line 19 to line 24). After that, we continue to process the next character in S (line 7).

5.4.2 Hybrid Method

As we discussed in § 5.2.2, no one combination of the algorithms (SW or NW), vectoring strategies (iterate or scan), gap penalty systems (linear or affine) can always provide optimal performance for different pairs of input sequences. Before we provide a better solution, we investigate the reason under what circumstances a specific combination can win. We test various query sequences, whose lengths range from 100 to 36k characters. We fix the algorithm to SW and the gap penalty system to the affine gap, and change the vectoring strategies. We find that the striped-scan strategy performs better when the number of re-computations in striped-iterate is around 1.5 times more on MIC, and 2.5 times on Haswell (For other combinations of algorithms and gap systems, the ratios are similar due to the similar computational pattern and workloads). Generally, if the best-matching score before the re-computations is high, meaning that the two input sequences may be close to each other, the striped-iterate has to carefully and iteratively check each position with more re-computation steps in order to eliminate the false negative; while in striped-scan, no matter what the matching scores are, the fix number of re-computations are needed. Paradoxically, we cannot rely on this observation to determine which strategy should be taken, because unless we finish the alignment algorithm and get the real matching scores, we don't know how similar or dissimilar in the input pair of sequences, or even in a specific rang of pairs.

In the chapter, we propose an input-agnostic hybrid method that can automatically select the efficient vectorizing strategy at the runtime. Our hybrid method starts from the striped-iterate strategy, in which we maintain a counter to record the number of re-computations. When the counter exceeds the configured threshold, the method will switch to the striped-scan. For example, based on the experiments for the combination of SW with the affine gap presented in the previous para-

graph, we set the threshold to be 2 for MIC and 3 for Haswell CPU. However, switching back from striped-scan to striped-iterate is nontrivial, because we don't know the amount of re-computations for striped-iterate when the algorithm is working in the striped-scan mode. Alternatively, we design a solution to "probe" the re-computation overhead at a configurable interval *stride*. That way, after processing *stride* characters in the subject sequence using the striped-scan, we tentatively switch back to the striped-iterate and rely on the counter to determine the next switch. Once the counter is above the threshold, we switch back again to the striped-scan for another round of processing *stride* characters. Otherwise, our method will stay in the striped-iterate mode and continue checking the counter.

Figure 5.5 shows an example of the hybrid method. If we only rely on the striped-iterate method, the re-computations in the middle part of the subject sequence will kill the performance due to the overhead of re-computations. In contrast, if we only use the striped-scan, the benefits of the head and tail parts in the striped-iterate will be wasted. Our hybrid method uses the counter to find the amount of re-computations is above the threshold around processing the 800-th character, and thus switch to striped-scan method. Then, it will probe the counter periodically by going back to the iterate method until the counter drops below the threshold or the end of the sequence S is achieved.

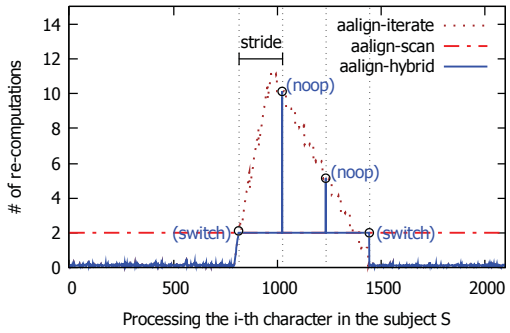


Figure 5.5: The mechanism of the hybrid method.

One may wonder why the hybrid mechanism starts from the striped-iterate, conservatively switches from striped-iterate to striped-scan only when the counter exceeds the threshold, and aggressively switches back by using the proactive probe. The reason is related with the characteristics of sequence search: although the sequence alignment is designed to find similar sequences

of databases for the input query, it cannot identify too many similar sequences because statistically most of the sequences of databases are dissimilar with a specific input. Even if a sequence is determined similar to the input, their exactly match regions are few. Considering the much faster convergence speed of striped-iterate for dissimilar pairs, we prefer it, and conservatively switch to striped-scan only when we find current aligned regions are highly matched.

5.4.3 Vector Modules

We've already seen the usage of the vector modules in Algorithm 5 and Algorithm 6. These vector modules are designed to express the required primitive vector operations in our vector code constructs and hide the ISA-specific vector instruction details. Therefore, when the platform changes, AAlign only needs to re-link the vector code to the proper set of vector modules. Table 5.1 defines the vector primitive modules. The first group of modules are designed to conduct basic vector operations over given arrays or vectors. Specifically, they are wrapper functions of the directly-mapped ISA intrinsics. As a contrast, the second group of modules carry out an application-specific operations, customized to our formalized vector code constructs.

Table 5.1: The vector modules in AAlign

Module Name	Description
Basic Vector Operation API	
load_vector(void **ad*); store_vector(void **ad*, vec *v*);	Load/store a vector from/to the memory address *ad*, which can be `char*`, `short*`, or `int*` (the same below)
add_vector(vec *va*, vec *v*); add_array(void **ad*, vec *v*);	Add a vector of *va* or from the memory address *ad* by vector *v*,
max_vector(vec *v1*, ...);	Take any count of input vectors, and return the vector with largest integers in each aligned position
App-specific Vector Operation API	
set_vector(int *m*, int *i*, int *g*);	Init a new vector, in which *i* is the default $T_{i,j}$ or $F_{i,j}$ value when j=0, *g* is their corresponding gap $\beta_{i,j}$ or $\theta_{i,j}$
rshift_x_fill(vec *v*, int *n*, ...); rshift_x_fill(void **ad*, int *n*, ...);	Right shift the vector of *v* or loaded from *ad* by *n* of positions and fill the gaps with specified values
influence_test(vec *va*, vec *vb*);	Check if vector *va* can affect the values in *vb*
wgt_max_scan(void **in*, void **out*, int *m*, int *i*, int *g*, int *G*);	"weighted" max-scan over the values in *in* of the striped format, store the results to *out*. *i* is the default $T_{i,j}$ value when j=0, *g*, *G* are the corresponding $\beta_{i,j}$, $\theta_{i,j}$

set_vector: is to set the lower-bound vector in the striped-iterate strategy. Figure 5.6 shows that AAlign will set the first value of the lower-bound vector to be the initial value i Then, the lower-bound values of the rest are set to be $i + l * k * g$, where l is the element's index, k is the

total number of vectors, and g is the associate gap penalty. The implementation of the module is to use the proper `_mm256/512_set` instrinsics.

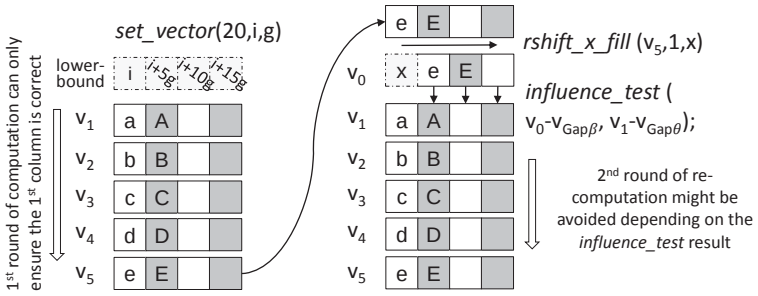


Figure 5.6: Vector modules used in the striped-iterate.

rshift_x_fill: is to right shift the vector elements with the value x filled. AAlign uses this module to adjust the data dependencies between vectors. As shown in Figure 5.6, the 1st round of computation can ensure the values in the first column (a-e cells) are correct, since they are calculated based on the real initial value i. Therefore, the test of the need for correction is required. Before that, we observe that in the 2nd round, the current "true" value e would affect A according to the original layout in Figure 5.4. As a result, we shift the vector v_5 to right by 1 position and fill the gap using a small enough number x to make sure there is no influence caused by it.

The implementation is essentially a combination of data-reordering operations. However, the selection of instructions is quite different because of different ISAs and desired data types. Figure 5.7 shows how to achieve the same functionality with different intrinsics. Because the shortest integer data type supported by IMCI is 32-bit, we only show IMCI with 32-bit int, which uses a combination of the cross 128-bit lane *permutevar* and *swizzle* intrinsics. As a contrast, we directly *insert* the value x after the *permutevar* completes on AVX2 with 32-bit int. If we work on the 16-bit values, there is no equivalent *permutevar* intrinsics so that we use *shufflehi/hi*, *permute8x32* and *blend* intrinsics for this functionality, followed by the *insert*.

influence_test: is to check if an extra re-computation of correction is necessary in the striped-iterate method. Specifically, the module is a vector comparison. The comparison results containing 1s mean the 1st operand will affect the 2nd one. In IMCI, the results are stored in a 16-bit mask and then we simply check if this value is larger than 0 or not. However, in AVX2, the "mask" is stored

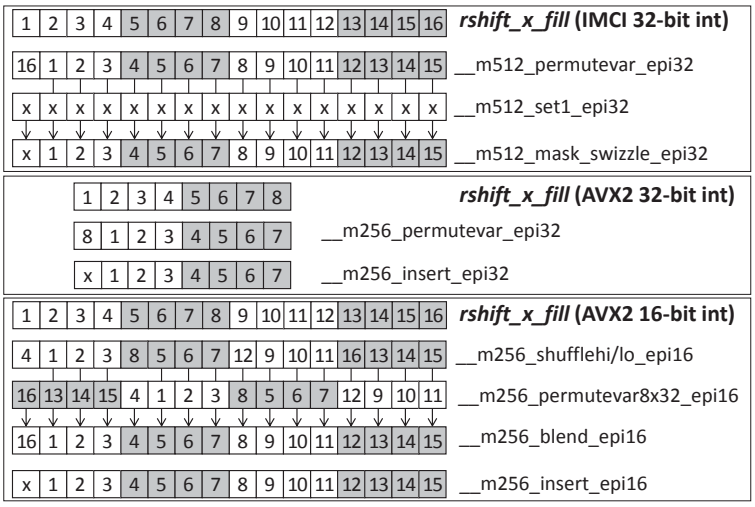


Figure 5.7: Example of chosen ISA intrinsics for *rshift_x_fill* (only *blend* operations are shown with arrows).

in a 256-bit vector, and there is no single instruction to peek how many set bits inside. Our solution is to split the vector to two 128-bit SSE vectors and use the intrinsic `_mm_test_all_zeros` to detect if there are set bits.

wgt_max_scan: is to implement the "weighted" scan along the buffer holding the tentative results (denoted as $\tilde{T}_{i,j}$ and stored in $\mathsf{arr}_{\mathsf{T1}}$ from line 18 of Algorithm 6). Mathematically, we perform the calculation of $max_{0 \leqslant l < j}(\tilde{T}_{i,l} + \theta_{i,l} + \sum_{k=l+1}^{j} \beta_{i,k})$. For simplicity, let's suppose $\theta_{i,l}$, $\beta_{i,k}$ are two constants θ and β and only use 8 characters as the example for the striped sequence shown in Figure 5.4. In Figure 5.8, we use three steps to achieve the *wgt_max_scan*. First, we conduct a preliminary round of inter-vector weighted scan on v_1 and v_2 with initial weight $\theta + \beta$ and extensive weight β. The results will be stored in the intermediate vectors u_1 and u_2. Second, an intra-vector and exclusive weighted scan is performed on vector u_2 with the weight of $k * \beta$, where k is the total number of vectors. The results are stored in s. Third, the last round of inter-vector and exclusive weighted broadcast is performed on s, u_1 and u_2 with the weight of β. The final scan results are stored in $\mathsf{arr}_{\mathsf{T1}}$.

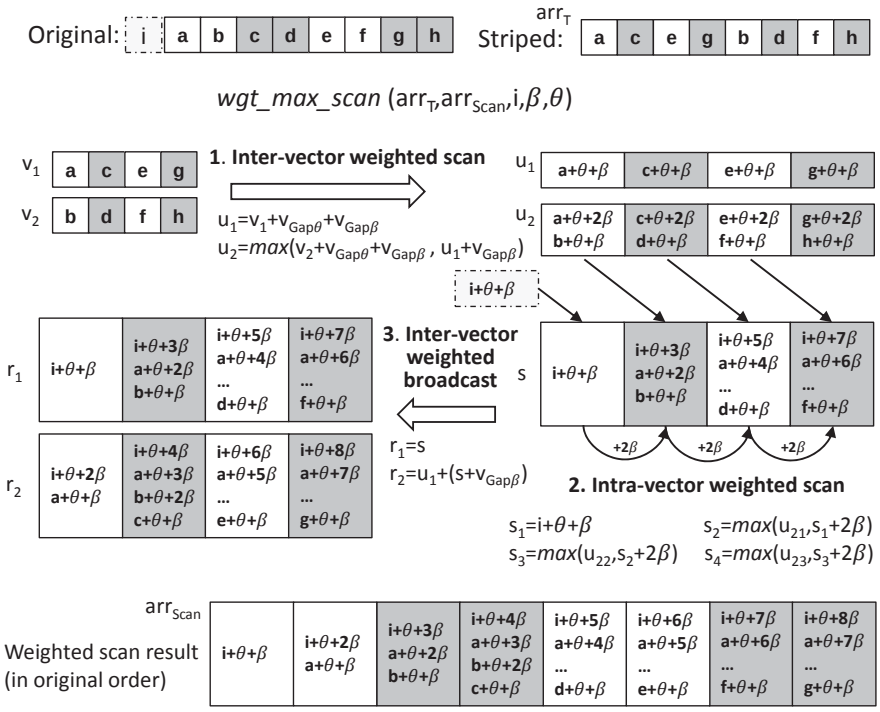


Figure 5.8: Orchestration mechanism in the *wgt_max_scan* (Maximum operations are applied on each cell).

5.4.4 Code Translation

The AAlign framework takes the sequential codes following our generalized paradigm as the input. After the analysis of the codes, the framework will decide how to modify the vector code constructs. We make use of Clang driver [3] to create the Abstract Syntax Tree (AST) for both the sequential codes and vector code constructs, shown in Figure 5.3. To traverse the trees, we build our Matcher and Visitor classes in Clang's AST Consumer class. Once the information from the AST nodes of interest is identified and retrieved, we use our Rewriter class to modify the AST tree of the vector code constructs with the information and its derivative results. Note, present framework only supports the constant gap penalties (e.g., $\beta_{i,k}$, $\theta_{i,l}$). We will leave it to future work to support variable penalties used in, for example, the dynamic time warping (DTW) algorithm.

Table 5.2 shows the configurable expressions in Algorithm 5 and Algorithm 6. The information can be retrieved from the sequential codes in four groups: 1. Identify which type of the pairwise alignment algorithm is used: local or global. This can be done by checking if there is a constant 0 set to T or not. 2. Identify what kind of gap penalty system is used. We can check

if θ is set to 0 or not (row 1-4 in Table 5.2). 3. Learn how to initialize the boundary values (row 5,6). 4. Derive other information of how to organize the vectors (row 7-11). After the vector code constructs have been rewritten, we use the hybrid method to generate our pairwise sequence alignment kernels.

Table 5.2: Configurable expressions in vector code contructs

Expression	Description & Format	Example*
GAP_LEFT	Gap penalty from the left T cell (i.e. $\theta'+\beta'$); constants or variables	$\mathsf{GAP}_{\mathsf{OPEN}}$ (line7)
GAP_UP	Gap penalty from the upper T cell (i.e. $\theta+\beta$); constants or variables	$\mathsf{GAP}_{\mathsf{OPEN}}$ (line8)
GAP_LEFT_EXT	Gap penalty from the left L cell (i.e. β'); constants or variables	$\mathsf{GAP}_{\mathsf{EXT}}$ (line7)
GAP_UP_EXT	Gap penalty from the upper U cell (i.e. β); constants or variables	$\mathsf{GAP}_{\mathsf{EXT}}$ (line8)
INIT_T	Upper boundary value of T cell; func(i)	0 (line2)
INIT_U	Upper boundary value of U cell; func(i)	0 (line2)
MAX_OPRD	Operands required by the max operation; vec variables	vU, vL, vZero
REC_FILL	Value to fill the right shifted gap; constant	$\mathsf{GAP}_{\mathsf{OPEN}}$ (line8)
REC_UP	Operand for checking the re-computation; vec variable	vU
REC_UP_GAP	Gap operand for REC_UP; vec variable	vGapU
REC_CRIT	Criterion for checking re-computation; vec variable	$\mathsf{vGapT}_{\mathsf{up}}$-vGapU

*: The examples are fetched or derived from Algorithm 4

5.4.5 Multi-threaded version

The AAlign framework can also utilize the thread-level parallelism of the multi- and many-cores to align a given query with all subject sequences in a database. We first assign the generated kernel to each thread, and a thread will get a subject sequence from the database to conduct the alignment until all subject sequences are aligned. After we sort the database by the subject sequence length, this dynamic binding mechanism is extremely efficient because of the load balance among threads. For the implementation, we don't need to create the profile array of substitution matrix for the query every time (prof in line 17 of Algorithm 5 or line 10 of Algorithm 6). Therefore, the only change of the kernel is to extract the part of building profile array and perform it once before launching multiple threads.

5.5 Evaluation

In the section, we evaluate the AAlign-generated pairwise sequence alignment codes on Haswell CPU and Knights Corner MIC. For Haswell, we use 2 sockets of E5-2680 v3, which totally contain

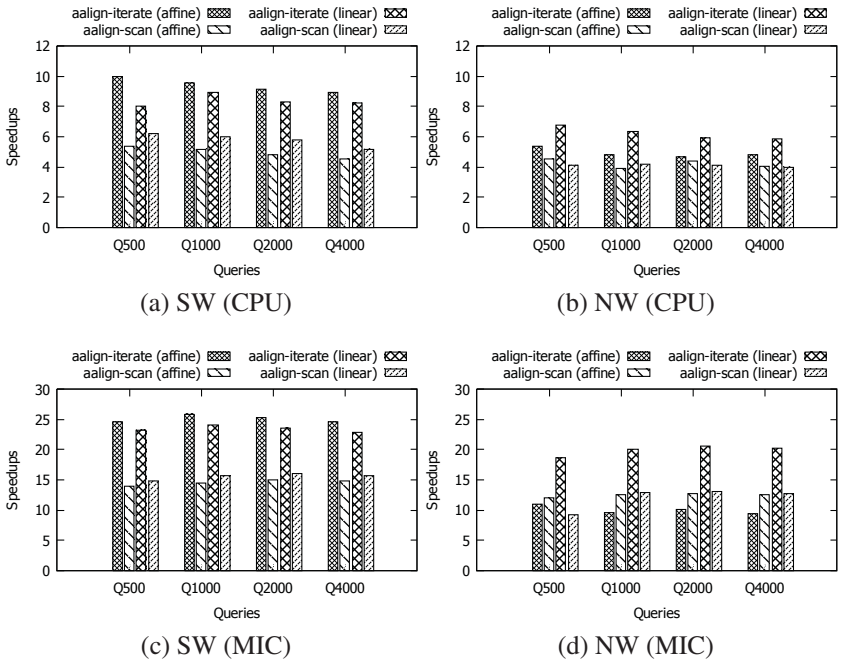
24 cores running on 2.5 GHz with 128 GB DDR3 memory. Each core has 32 KB L1, 256 KB L2, and shares 30 MB L3 cache. For MIC, we use the Intel Xeon Phi 5110P coprocessor in the *native* mode. The coprocessor consists of 60 cores running on 1.05 GHz with 8 GB GDDR5 memory, and each core includes 32 KB L1 and 512 KB L2 cache. We use $icpc$ in Intel compiler 15.3 with *-O3* option to compile the codes. To specialize the desired vector ISA, we also include *-xCORE-AVX2* for CPU and *-mmic* for MIC. All the sequences are from NCBI-protein database [4]. The number of characters is integrated into the query name.

Our objectives include: (1) Compare AAlign-generated codes with the optimized sequential codes. (2) Compare the proposed hybrid method with the iterate and scan method, respectively. (3) Compare multi-threaded versions of AAlign-generated codes with the existing tools.

5.5.1 Speedups from Our Framework

We first compare the AAlign-generated codes (32-bit int) with the sequential codes (32-bit int) to evaluate the vectorization efficiency. The subject sequence is a Q282. The sequential codes are following the same logic of the vector codes. We also add "#pragma vector always" in the inner-loop of the codes. The speedups, shown in Figure 5.9, are the performance benefits brought by the AAlign using striped-iterate and striped-scan respectively. By using the striped-scan, the SW and NW can achieve an average of 4.8 and 13.6-fold speedups over the sequential codes on CPU and MIC respectively. In contrast, the speedups of the striped-iterate SW and NW vary in a wider range of 4.7 to 10-fold on CPU and 9.5 to 25.9-fold on MIC. The superlinear speedups of the striped-iterate are mainly because the striped-iterate avoids a considerable amount of computation along the Q if the $influence\_test$ fails.

We can see that the performance variance of the striped-scan is smaller than the striped-iterate. For example, though the SW approximates the NW in terms of computational workloads, the performance of the striped-iterate SW-affine (Figure 5.9c) and NW-affine (Figure 5.9d) changes a lot, while the striped-scan keeps relatively consistent. Actually, the performance difference of the two methods depends on the processed numerical values which are affected by the algorithms, gap systems, and input sequences.



(a) SW (CPU) (b) NW (CPU)

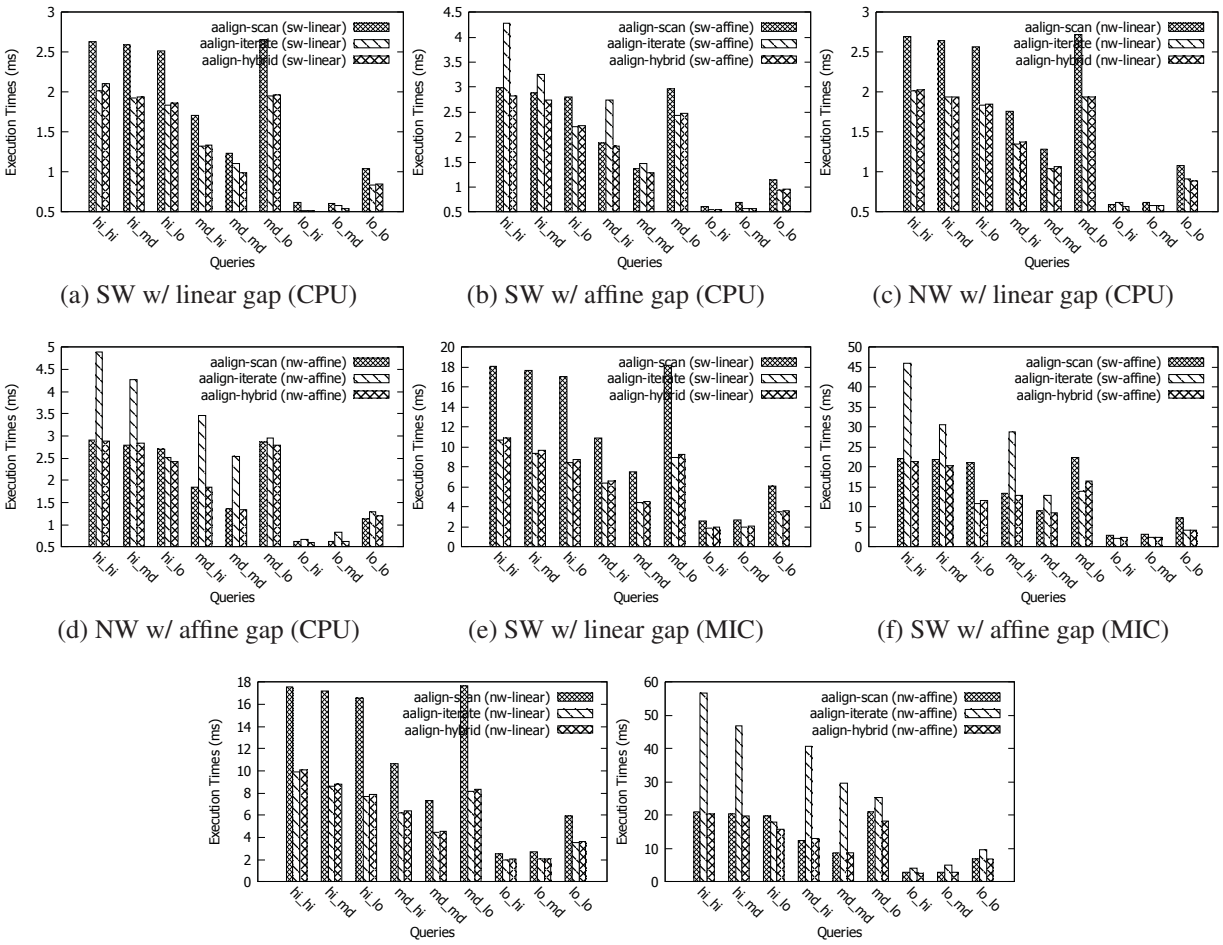
(c) SW (MIC) (d) NW (MIC)

Figure 5.9: AAlign codes vs. Baseline sequential codes. The baselines are different and they are optimized to follow the similar logic with the corresponding AAlign codes.

5.5.2 Performance for Pairwise Alignment

In the preceding section, we observe that the algorithm and gap penalty system will affect the choice of the better vectorizing strategy. This section changes the input sequences. We first borrow the concepts of query coverage (QC) and max identity (MI) [1] from the bioinformatics community to describe the similarity of the input sequences. QC means the percent of query sequence Q overlapping the subject S, while the MI is the percentage of the similarity between Q and S over the length of the overlapped area. Additionally, we define three ranges of hi (>70%), md (70%-30%), and lo (<30%). That way, we have 9 combinations of QC_MI to represent the similarity and dissimilarity of two input sequences. For example, lo_hi means only a small portion of two sequences overlaps each other, but the overlapped areas are highly similar. In the experiment, we use Q2000 against the "nr" database using NCBI-BLAST [1] and pick out 9 typical subjects for the aforementioned criteria.

Figure 5.10 shows the performance of AAlign using different vectorizing strategies, including



(a) SW w/ linear gap (CPU) (b) SW w/ affine gap (CPU) (c) NW w/ linear gap (CPU)

(d) NW w/ affine gap (CPU) (e) SW w/ linear gap (MIC) (f) SW w/ affine gap (MIC)

(g) NW w/ linear gap (MIC) (h) NW w/ affine gap (MIC)

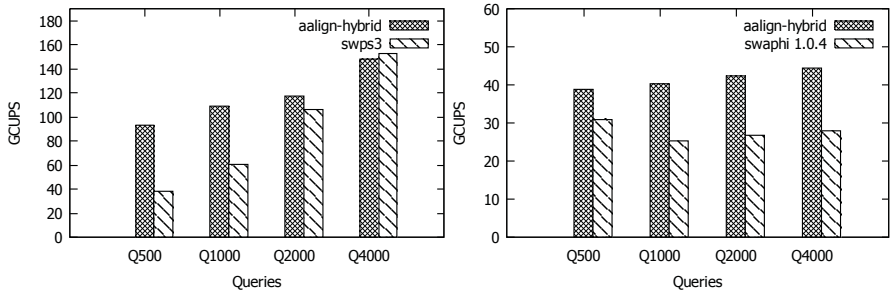
Figure 5.10: AAlign codes using striped-iterate, striped-scan, and hybrid method. The x-axis represents the similarity of the two sequences using the format of QC_MI in which the query coverage (QC) and max identity (MI) metrics are in three levels: high (>70%), medium (70%-30%), and low(<30%).

striped-iterate, striped-scan, and hybrid, on CPU and MIC. For the alignment algorithms with linear gap penalty, the striped-iterate method always outperforms the striped-scan, because the effects of the zero θ cause the number of re-computations falling into a very small number. The results also show that with the linear gap penalty, our hybrid method will fall back to the striped-iterate and has very similar performance with it. For the algorithms with affine gap penalty, the striped-scan is better than the striped-iterate when two sequences have high or medium scores of QC and MI, meaning that the input sequences are very similar. For example, for the sequences labeled as *hi_hi*, *hi_md*, *md_hi*, *md_md*, in Figure 5.10b, 5.10d, 5.10f, 5.10h, the striped-scan is the

better solution, thanks to its fixed rounds of re-computation. In the cases of the NW with the affine gap, the striped-scan can deliver up to 3.5 fold speedup on MIC and up to 1.9 fold speedup on CPU over the striped-iterate. For other inputs (dissimilar input sequences), the striped-iterate is better. Because the hybrid method can automatically switch to the better solution, in most test cases, the hybrid method has better performance than either of the striped-iterate and striped-scan method. In the corner cases, the hybrid method approximates to the better solution instead of the worse one.

5.5.3 Performance for Multi-threaded Codes

In the section, we compare AAlign's multi-threaded SW with affine gap penalty system with the tools of SWPS3 and SWAPHI. The database is the "swiss-prot" containing more than 570k sequences [6]. SWPS3 [142] uses a modified version of the striped-iterate method working on CPUs. The buffers of the table T are of char and short data types. SWAPHI [105] supports both inter-sequence and intra-sequence vectorization in the multi-threaded on MIC. In the experiment, we only focus on their intra-sequence method of int data type. Correspondingly, we use our generated kernel of short and int data type on CPU and MIC respectively.



(a) vs. SWPS3 on CPU (b) vs. SWAPHI on MIC

Figure 5.11: AAlign Smith-Waterman w/ affine gap vs. existing highly-optimized tools.

Figure 5.11 presents the results of AAlign SW algorithms comparing with the two highly-optimized tools. On the CPU, the generated AAlign codes can outperform the SWPS3 for up to 2.5 times, especially for the short query sequences. However, in Figure 5.11a, for the long sequences Q4000, SWPS3 is better. This mainly because rather than working entirely on the short data type (16 bits), SWPS3 also uses the char-type (8 bits) buffers. Only when the overflow occurs,

the tool will switch to the short. This is especially beneficial for long query sequences by lowering the cache pressure. For the MIC, we can outperform the SWAPHI on an average of 1.6 times, thanks to our hybrid method and the efficient vector modules.

5.6 Chapter Summary

The AAlign framework can generate the vector codes based on "striped-iterate" and "striped-scan". Moreover, we design an input-agnostic hybrid method, which can take advantage of both the vectorization strategies. The generated codes will be linked to a set of platform-specific vector modules. To do this, the AAlign only needs the input sequential codes following our generalized paradigm. The results show that the vector codes can deliver considerable performance gains over the sequential counterparts by utilizing the data-level parallelism and decreasing the amount of computation. We also demonstrate that our hybrid method is able to automatically switch to the better vectorization strategy at runtime. Finally, compared to the existing highly-optimized multi-threaded tools, the multi-threaded AAlign codes can also achieve competitive performance.

Chapter 6

Data Dependencies in Wavefront Loops

6.1 Introduction

Modern accelerators, e.g., GPUs, feature wide vector-like compute units and a complex memory hierarchy. If parallel applications can be organized to follow the SIMD processing paradigm, coalesced memory access patterns, and data reuse at different levels of the memory hierarchy, GPUs can often deliver superior performance over CPUs. However, *wavefront loops*, which can be found in many scientific applications, including partial differential equation (PDE) solvers and sequence alignment tools, are exceptions. Because their computations (including the association operator and the distribution operator, discussed in § 6.2.1) update each entry of a two-dimensional (2-D) matrix based on the already-updated values from its upper, left, and (optional) diagonal neighbors, this strong data dependency hinders the optimizing of computation and memory access on GPUs at the same time. That is, if data is stored in a row- or column-major order, the data can be processed in parallel but from non-contiguous memory addresses. In other words, data dependencies prevent consecutively stored data to be processed in parallel. Alternatively, if the data is stored in an anti-diagonal-major order, parallel computation can naturally follow the data dependency, but the exposed parallelism may result in severe load imbalance and underutilization of the compute units.

Figure 6.1 provides an overview of existing approaches to optimize waverfront loops on

GPUs. In ①, the parallelism on anti-diagonal data is directly exposed by applying loop transformation techniques, e.g., loop skewing and loop interchange [20, 156]. These methods may lead to severe performance penalties due to non-contiguous memory access and load imbalance. In ②, the tiling-based methods [110, 59, 24] block the reusable data in local memories, e.g., caches, to reduce the overhead of uncoalesced memory access. However, as their computation still strictly follows the anti-diagonal order, the load imbalance occurs at the beginning and ending anti-diagonals, causing the underutilization of computing resources. Although the recent tiling-based method called PeerWave [24] can mitigate the problem, it incurs extra overhead to transform the data layout. In contrast, instead of mainly optimizing memory accesses on GPUs, the recent studies [62, 87] in ③ focus on accelerating computations and resolving the load imbalance. The core idea is to ignore the data dependency along a row at first, compute data entries in a row in parallel, and finally correct the intermediate results. We call this method as a **compensation-based** parallelism for wavefront loops. However, this method requires expensive global synchronizations within and between processing each row, leading to frequent global synchronizations and the loss of data reuse. More importantly, because this method does not follow the original data dependency and has changed the sequence of computation operators on data entries, the domain knowledge from developers is required for the correctness of the final results, which makes it a privilege for experienced users only.

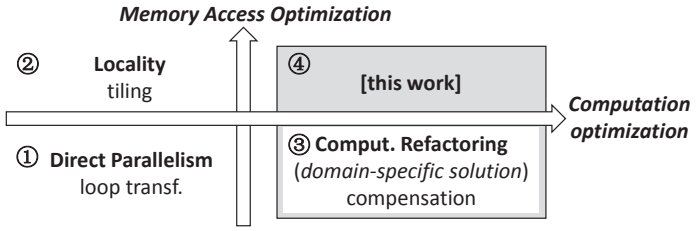


Figure 6.1: Parallelization landscape for wavefront loops.

In this chapter, we first investigate under which circumstances, the compensation-based method that breaks through the data dependency and changes the sequence of computation operators properly can be used to optimize wavefront loops. We prove that if the accumulation operator is associative and commutative and the distribution operator is either distributive over or same with the

accumulation operator, changing the sequence of operators properly doesn't affect the correctness of results. We also analyze that several popular algorithms, including a successive over-relaxation (SOR) solver [58], Smith-Waterman (SW) algorithm [140], summed-area table (SAT) computation [118], and integral histogram (IHist) algorithm [125], satisfy such requirements. Due to its generality, we design a highly efficient compensation-based solution for wavefront loops on GPUs: we propose a weighted scan-based method to accelerate the computation and combine it with the tiling method to optimize memory access and reduce global synchronization overhead.

In the evaluation, we first compare the performance of the weighted scan-based GPU kernels with those based on widely-used libraries, i.e., Thrust [71] and ModernGPU [22], and our kernels can deliver an average of 3.5x and 4.7x speedups on NVIDIA K80 and P100 GPUs, respectively. We also use our methods to optimize SOR, SW, SAT, and IHist application kernels, yielding up to 22.1x (43.3x) speedups on K80 (P100) over state-of-the-art optimizations [24]. Even for their best scenarios, we can still obtain an average of 1.1x (1.8x) improvements on K80 (P100). The key contributions of this chapter are summarized below.

- We prove that in wavefront loops, if the accumulation operator is associative and commutative and the distribution operator is either distributive over or same with the accumulation operator, breaking through the data dependency and changing the sequence of computation operators properly does not affect the correctness of results. This provides the guidance for developers under which circumstances, the compensation-based method can be used. (In ③ and ④ of Figure 6.1.)

- We design a highly efficient compensation-based method on GPUs. Our method provides the weighted scan-based GPU kernels to optimize the computation, and combines with the tiling method to optimize the memory access and synchronization. (In ④ of Figure 6.1.)

- We carry out a comprehensive evaluation on both kernel level and application level to demonstrate the efficiency of our method over the state-of-the-art research for wavefront loops.

6.2 Motivation

6.2.1 Wavefront Loops and Direct Parallelism

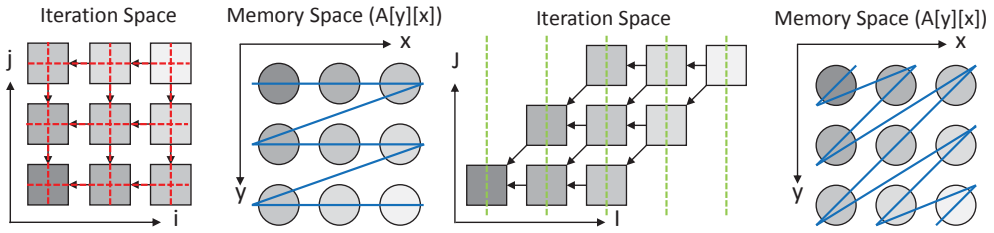
When loop-carried dependencies are present, compilers are oftentimes hard to parallelize the loops effectively, even with the auto-vectorization technologies and user-provided directives [108]. Algorithm 6.1 shows an example of the original loop nests with such data dependencies. The corresponding iterations and memory spaces are shown in Figure 6.2a. This loop can be parallelized for neither the *i*-loop nor *j*-loop, if we admit the row-major memory access pattern[1]. Fortunately, the parallelism-inhibiting dependencies can be 'eliminated' by applying loop transformation techniques, e.g., loop skewing and loop interchange [156, 20]. The transformed loop is also shown in Algorithm 6.1. Thereafter, the potential parallelism can be exposed from the iteration space in Figure 6.2b. However, this approach has two significant drawbacks: (a) load imbalance, especially at the beginning and the ending iterations (shown in the iteration space); (b) non-contiguous memory access (shown in the memory space).

Algorithm 6.1 Original & transformed loop nests with wavefront parallelism.

```
1 // Original
2 for(int i = 0; i < m; i++)
3   for(int j = 0; j < n; j++)
4     A[i][j] = A[i][j-1] * 0.5 + A[i-1][j] * 0.5;
5 // Transformed via loop skewing and interchange
6 for(int I = 0; I < m+n-1; I++)
7   for(int J = max(0, I-n+1); J < min(m, I+1); J++)
8     A[J][I-J] = A[J][I-J-1] * 0.5 + A[J-1][I-J] * 0.5;
```

Algorithm 6.1 also shows there are two types of binary operators in wavefront loops. The *distribution operator* will distribute a part of the value of one entry to another. For example, in this example, the multiplication "$*$" is the distribution operator, which distributes a portion of `A[i][j-1]` and `A[i-1][j]` to `A[i][j]`. Another operator is the *accumulation operator*, which will accumulate incoming values into an entry. Here, the accumulation operator is "+". Most existing studies strictly follow the sequence of operators, which means an entry will be updated by the accumulation operator only after receiving the distributed values from all prerequisites.

[1] By default, we assume the row-major layout for all arrays.



(a) Original (b) Transformed

Figure 6.2: Exposed parallelism and corresponding memory access pattern of the two forms of loop nests in Algorithm 6.1. In the iteration space, the arrow represents the data dependency, e.g., a←b means iteration b depends on a.

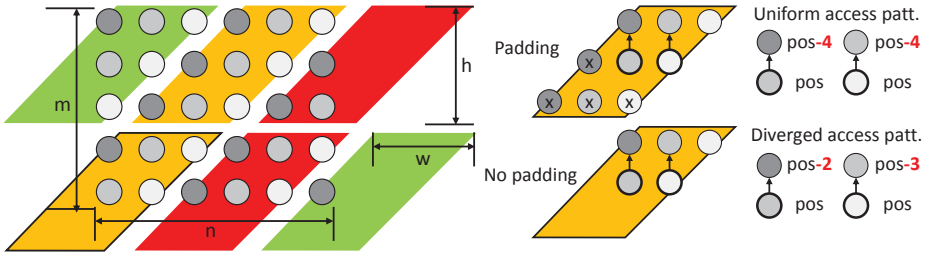
6.2.2 Tiling-based Solutions and Their Limitations

To reduce the cost of load imbalance and amortize the overhead of non-contiguous memory access, many studies [24, 152, 110] apply the tiling-based methods, where the spatial locality can be improved and the expensive synchronization among each entry will convert to the synchronization among tiles. However, there are two other issues.

Data layouts: In a basic design using the tiling optimizations, one can divide the working set into tiles and follow the anti-diagonal direction to parallelize the computation. The overhead of accessing non-contiguous data is mitigated by the cache. However, the non-contiguous data access still exists inside each tile, and at the beginning and the ending of anti diagonals, there are no enough entries that can be executed in parallel. This motivates the anti-diagonal major storage and *hyperplane* on GPUs [59, 24]. However, there are two another problems emerging:

(1) Wasted memory and computing resources. Suppose the dimensions of the working matrix `A` are m by n and it is divided into *hyperplane* tiles of h by w, shown in Figure 6.3a. To store the array `A`, we need to allocate $\lceil \frac{m}{h} \rceil \cdot \lceil \frac{n+h-1}{w} \rceil$ *hyperplane* tiles, where $n + h - 1$ is to process n row entries plus the longest preceding padding entries that is equal to $h - 1$. As a result, the actual memory usage for all *hyperplane* tiles must be larger than $m \cdot (n + h)$. Therefore, in the *hyperplane* mode, the percentage of effective memory usage is approximately $n/(n + h)$. Apparently, the padding overhead is not negligible when $n + h$ is sufficiently larger than n.

One might wonder that the padding could be removed by adding rules to skip out-of-bound access. However, this strategy will break the uniform access that is preferred by GPUs. Figure 6.3b



(a) Hyperplane tiles (b) Access patterns

Figure 6.3: Splitting the array `A` into hyperplane tiles and their access patterns w/ and w/o padding.

demonstrates the diverged access in the padding-free scenario. Compared to the uniform pattern, each highlighted element needs to use different indexing formulas to obtain their north neighbors, e.g., $pos - 2$ and $pos - 3$ in the figure. Therefore, the padding-free strategy may greatly increase the complexity of indexing and lead to more branches in GPU kernels, resulting in the performance degradation.

(2) Layout transformation overhead. To remove the non-contiguous memory access, the data layout can be transformed from the row-major to the anti-diagonal major [33, 24]. However, this conversion not only requires developers refactoring the implementations, but causes significant transformation overhead. In the evaluation, we have observed the transformation time makes up to 31˜60% and 40˜72% of computation time on on NVIDIA K80 and P100 GPUs, respectively (§ 6.6.2).

Task scheduling: The tile-based solutions, e.g., [110, 24, 160], assigns a complete row of tiles to one compute unit, e.g., a Multiprocessor (MP) of GPU. In that way, the sequential execution order by a MP naturally satisfies the dependency between tiles on the same row; while the dependencies between tiles on different rows can be satisfied via lightweight local synchronizations, e.g., the spin-lock, leading to a pipeline-like execution mode. This methodology works very well for square matrices (e.g., $m \approx n$), because the load balance among compute units can be quickly achieved by the large amounts of parallel tiles along the anti-diagonal. However, for rectangular matrices, especially when $m \ll n$, such a methodology may lose the efficiency, since there are no sufficient tiles in most anti-diagonals.

6.2.3 Compensation-based Solutions and Their Limitations

In recent years, several studies [87, 76, 62] have offered another type of solutions for parallelizing wavefront loops. Overall, the computation is conducted in a row-by-row manner. The contiguous data entries in a row are divided into groups and scheduled to different compute units. Figure 6.4 shows three main steps on processing the bottom row: (1) each compute unit ignores the horizontal data dependency and computes its data entries in parallel to generate the intermediate results; (2) a compensation step is performed to compute ignored data for each entry in a scan-like process; (3) each compute unit corrects the intermediate results with the compensations for the final results. Each step of this method can be parallelism-friendly, and also easy to balance entries between compute units.

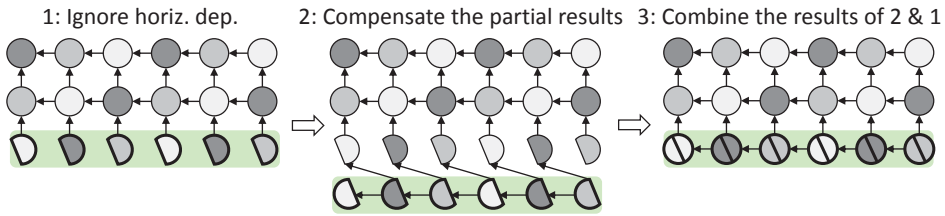


Figure 6.4: Compensation-based solutions decompose the processing into three steps, each of which can be parallelism-friendly and load balanced.

However, this solution requires multiple expensive global synchronizations within and between processing each row. Within a row, in the step 2, after a compute unit finish its local computation on the ignored data, it has to wait for the finish of all preceding compute units to get their compensation results, because the data dependency is propagated from the start to the end along a row. Between rows, only after the third step finishes at all compute units, they can continue processing the next row to avoid the data dependency between rows. Thus, the performance might deteriorate without a highly optimized compensation design. More importantly, previous research illustrates the compensation-based parallelism works well for string matching operators, e.g., *max* and +, but its generality to other domains is still unclear. As a consequence, in this chapter, we will determine the boundary of the compensation-based method and answer the question: under which circumstances, can the compensation-based method be used to optimize wavefront loops?

6.3 Compensation-based Computation – Theory

6.3.1 Standard Wavefront Computation Pattern

We define our target wavefront computation pattern by capturing the key operations and formalizing their data dependencies.

Definition 1. (*Wavefront Pattern*) Let $A = (A_{i,j})$ be a m by n matrix to store the output of the wavefront computation. For any entry $A_{i,j}$, where $0 < i < m$ and $0 < j < n$, the relationships with its neighbors, e.g., $A_{i,j-1}$, are defined by applying two generic binary operators $\diamond$ and $\circ$ as shown in Equation I. Besides, a constant or variable value b can be applied on the operator $\circ$. Note, when $i = 0$ or $j = 0$, $A_{i,j}$ can be predefined according to application-specific rules.

$$A_{i,j} = (A_{i,j-1} \circ b_0) \diamond (A_{i-1,j} \circ b_1) \diamond (A_{i-1,j-1} \circ b_2) \tag{I}$$

In the definition, $\circ$ is the distribution operator and $\diamond$ is the accumulation operator, while we will use these symbolic representations in the proof. This abstracted definition can cover various real-world wavefront loops by transforming the generic operators into concrete ones. The practical cases will be discussed in § 6.4.

6.3.2 Compensation-based Computation Pattern

We can present the compensation-based computation pattern with the generic operators in Equation I. First, the data dependencies along the j-direction are ignored and the partial results are represented as $\tilde{A}_{i,j}$, leading to Equation II-1. Second, an additional compensation step of Equation II-2 is carried out to produce the compensation values $B_{i,j}$. Third, the compensation values are used to correct the partial results $\tilde{A}_{i,j}$ for the loss caused by the loosened dependencies, as shown in Equation II-3. The symbols $\prod$ and $\sum$ represent the iterative binary operations $\circ$ and $\diamond$, respectively.

$$\tilde{A}_{i,j} = (A_{i-1,j} \circ b_1) \diamond (A_{i-1,j-1} \circ b_2) \tag{II-1}$$

$$B_{i,j} = \begin{cases} \sum_{u=0}^{j-1}(\tilde{A}_{i,u} \circ \prod_{v=u}^{j-1} b_0) & \text{when } \circ \neq \diamond \\ \sum_{u=0}^{j-1}(\tilde{A}_{i,u} \diamond b_0) & \text{when } \circ = \diamond \end{cases} \tag{II-2}$$

$$A_{i,j} = \tilde{A}_{i,j} \diamond B_{i,j} \tag{II-3}$$

Obviously, this new pattern has changed the computation ordering in Equation I. Thus, to show the validity, we need to prove that under which circumstances, the Equation II-3 (with the Equation II-1 and Equation II-2) is equivalent to the Equation I.

Theorem 1. The compensation-based computation shown in Equation II-3 (incl. Equation II-1 and II-2) is equivalent with the original computation in Equation I, provided the binary operators $\diamond$ is associative and commutative, and (1) $\circ$ has the distributive property over $\diamond$, or (2) $\circ$ is same with $\diamond$ (where, for brevity, we only use $\diamond$).

Proof. We use the induction method to prove the equivalence of the two equations. First, we focus on a *base case* to prove the statement holds for updating the first element $A_{1,1}$. Starting from Equation II-3, we have $A_{1,1} = \tilde{A}_{1,1} \diamond B_{1,1}$. According to Equation II-2 and $A_{1,0}$ is predefined, the item $B_{1,1} = \tilde{A}_{1,0} \circ b_0 = A_{1,0} \circ b_0$, no matter $\circ$ is same with $\diamond$ or not. Then, putting Equation II-1 and Equation II-2 into Equation II-3, we can get $A_{1,1} = (A_{0,1} \circ b_1) \diamond (A_{0,0} \circ b_2) \diamond (A_{1,0} \circ b_0)$. Since $\diamond$ has the commutative property, this is equal to $(A_{1,0} \circ b_0) \diamond (A_{0,1} \circ b_1) \diamond (A_{0,0} \circ b_2)$, which is $A_{1,1}$ defined by Equation I. Thus the statement is true for the base case.

Then, we focus on the *inductive step*: if the statement holds for $j = k - 1$, then it also holds for $j = k$.

In the case of $\circ \neq \diamond$, based on the Equation II-2, we know $B_{i,k} = \sum_{u=0}^{k-1}(\tilde{A}_{i,u} \circ \prod_{v=u}^{k-1} b_0)$. We unfold $\sum$ to get:

$$B_{i,k} = \sum_{u=0}^{k-2}(\tilde{A}_{i,u} \circ \prod_{v=u}^{k-1} b_0) \diamond (\tilde{A}_{i,k-1} \circ b_0) \tag{6.1}$$

Since $\diamond$ has the commutative and associative properties, this can be transformed to:

$$B_{i,k} = (\tilde{A}_{i,k-1} \circ b_0) \diamond (\sum_{u=0}^{k-2}(\tilde{A}_{i,u} \circ \prod_{v=u}^{k-1} b_0)) \tag{6.2}$$

Since $\circ$ has the distributive property over $\diamond$, we can "factor out" a b_0 from each term and get:

$$B_{i,k} = (\tilde{A}_{i,k-1} \diamond (\sum_{u=0}^{k-2}(\tilde{A}_{i,u} \circ \prod_{v=u}^{k-2} b_0))) \circ b_0 \tag{6.3}$$

Using Equation II-2 when $j = k - 1$, Equation 6.3 can be simplified to:

$$B_{i,k} = (\tilde{A}_{i,k-1} \diamond B_{i,k-1}) \circ b_0 \tag{6.4}$$

Because the induction hypothesis that $j = k - 1$ holds, meaning $A_{i,k-1} = \tilde{A}_{i,k-1} \diamond B_{i,k-1}$ is true, we can get:

$$B_{i,k} = A_{i,k-1} \circ b_0 \tag{6.5}$$

Then, putting Equation II-1 and Equation 6.5 to Equation II-3, we get $A_{i,k} = (A_{i-1,k} \circ b_1) \diamond (A_{i-1,k-1} \circ b_2) \diamond (A_{i,k-1} \circ b_0)$. Due to the commutative property of $\diamond$, this is equal to Equation I. Therefore, we demonstrate the statement also holds for $j = k$ in the case of $\circ \neq \diamond$.

Now, we consider the case of $\circ = \diamond$, where $B_{i,k} = \sum_{u=0}^{k-1}(\tilde{A}_{i,u} \diamond b_0)$. Then, due to the associative and commutative property of $\diamond$, $B_{i,k}$ can be transformed to:

$$B_{i,k} = (\tilde{A}_{i,k-1} \diamond (\textstyle\sum_{u=0}^{k-2}(\tilde{A}_{i,u} \diamond b_0))) \diamond b_0 \tag{6.6}$$

Equation 6.6 can be simplified by using Equation II-2 when $j = k - 1$.

$$B_{i,k} = (\tilde{A}_{i,k-1} \diamond B_{i,k-1}) \diamond b_0 \tag{6.7}$$

Using the induction hypothesis that $j = k - 1$ holds, we can get $B_{i,k} = A_{i,k-1} \circ b_0$. Then, similar to the case of $\circ \neq \diamond$, we can prove Equation II-3 is equal to Equation I for the case of $\circ = \diamond$ in $j = k$.

Since both the base and inductive cases have been performed, the statement holds for all natural numbers j. □

Now, we compare the complexity of the proposed compensation-based method with the original one. Obviously, the key difference of the two methods relies on how to satisfy the dependencies along the j-direction. In the original method, it is done by $A_{i,j-1} \circ b_0$ with $O(1)$ complexity, while in the proposed method, Equation II-2 is used for the same purpose, leading to $O(n)$ complexity. Nevertheless, Equation II-2 can be also optimized to $O(1)$ by using dynamic programming techniques, i.e., $B_{i,j} = (B_{i,j-1} \circ b_0) \diamond (\tilde{A}_{i,j-1} \circ b_0)$ for $\circ \neq \diamond$ or $B_{i,j} = B_{i,j-1} \diamond (\tilde{A}_{i,j-1} \circ b_0)$ for $\circ = \diamond$. However, updating $B_{i,j}$ is still more expensive than the original $A_{i,j-1} \circ b_0$ operation. We will show the proposed method can expose more parallelisms in § 6.5, and thus it provides better performance in § 6.6.

6.4 Compensation-based Computation – Practice

In this section, we discuss four representative wavefront loops from real-world applications. In addition, we demonstrate whether and how these loops can be expressed in the compensation-based parallelism patterns shown in § 6.3.

SOR Solver (SOR) [58]: The method of successive over-relaxation (SOR) conducts a stencil-like computation to solve a linear system of equations in an iterative fashion. As shown below, `A[i][j]` represents a discrete gridpoint and its new value depends on its neighbors: some are the most recently updated (i.e., `A[i][j-1], A[i-1][j]`), while others are from the previous time step (i.e., `A[i][j], A[i+1][j], A[i][j+1]`), resulting in a wavefront computation pattern.

```
A[i][j] = (A[i][j] + A[i][j-1] + A[i-1][j] + A[i+1][j] + A[i][j+1]) / 5;
```

To express the computation in the compensation-based parallelism pattern in § 6.3, we map ($\diamond$, $\circ$) to $(+, \cdot)$ and b_0, b_1, b_2 to 0.2. Obviously, the operator $+$ and $\cdot$ satisfy the requirements of the Theorem 1.

Smith-Waterman (SW) [140]: It is a well-known algorithm to align the input sequences a and b. `A[i][j]` stores the maximum score for aligning the sub-sequences 0-i of a and 0-j of b. The `s(i,j)` is the substitution function (i.e., b_2) to check if the corresponding amino acids are same at i of a and j of b. The constant `2` is the insertion/deletion penalty (i.e., b_0 and b_1). For the operators, we map ($\diamond$, $\circ$) to $(max, +)$. Note, max is a binary operator, but for brevity, we put four operands in this form.

```
A[i][j] = max(A[i][j-1] - 2, A[i-1][j] - 2, A[i-1][j-1] + s(i,j), 0);
```

Summed-area Table (SAT) [118]: It is used to accelerate texture filtering in image processing, where `A[i][j]` stores the sum of all pixels above and to the left of the point (i,j). Thus, `p[i][j]` is the pixel value (i.e., b_2). In addition, the operator $\circ$ is equal to $\diamond$ and is $+$; b_0 and b_1 are both 0. Note that, the computation order in the compensation-based method discussed in the previous section is along the j-direction. In this case, all entries at the row $i-1$ have been updated when processing the row i. As a result, the negation on `A[i-1][j-1]` will not affect the correctness.

```
A[i][j] = p[i][j] + A[i][j-1] + A[i-1][j] - A[i-1][j-1];
```

Integral Histogram (IHist) [125]: It extends the SAT and enables the multi-scale histogram-based search. In this method, `A[i][j]` is the histogram position for a bin z of its top-left sub-image, and thus, `Q(i,j,z)` checks if the pixel (i,j) belongs to bin z or not (i.e., b_2). The other parts are similar with SAT: $\circ$ is equal to $\diamond$ and is $+$; b_0 and b_1 are both 0.

```
A[i][j] = A[i][j-1] + A[i-1][j] - A[i-1][j-1] + Q(i,j,z);
```

6.5 Design and Implementation on GPUs

This section presents our efficient design of the compensation-based parallelism on GPUs. Because Equation II-1 and Equation II-3 naturally present no dependencies between neighboring entries and are easy to parallelize, we focus on Equation II-2, the compensation step.

6.5.1 Compensation-based Computation on GPUs

For the compensation step, based on Equation II-2, we can transform the computation to a fixed number of operations, i.e., $B_{i,j} = (B_{i,j-1} \circ b_0) \diamond (\tilde{A}_{i,j-1} \circ b_0)$ for the case of $\circ \neq \diamond$, and$B_{i,j} = B_{i,j-1} \diamond (\tilde{A}_{i,j-1} \circ b_0)$ for the case of $\circ = \diamond$. This method is used by previous research [87]. However, it will make every cell in B to depend on all preceding entries, causing strong data dependencies. Besides, the two formulas indicate different strategies to cope with parallelization, setting obstacles for the implementations on GPUs.

Therefore, we propose the efficient **scan-** and **weighted scan-based** methods to process the compensation computation. For the case of $\circ = \diamond$, because all previous $\tilde{A}$ contribute equally to current B, they only need to add a single b_0 on the operator ($\circ$). For example, to calculate $B_{1,3}$, $\tilde{A}_{1,0} \circ b_0$, $\tilde{A}_{1,1} \circ b_0$, and $\tilde{A}_{1,2} \circ b_0$ are used and each of them only needs to add a single b_0 on the operator without the consideration of the index. This actually corresponds to a typical **scan** operation, which has be well understood on GPUs [163]. However, the case $\circ \neq \diamond$ is much complicated, because each previous $\tilde{A}$ has different impacts on current B. For example, to calculate $B_{1,3}$, $\tilde{A}_{1,0} \circ 3b_0$,

$\tilde{A}_{1,1} \circ 2b_0$, and $\tilde{A}_{1,2} \circ b_0$ are applied[2]. Thus, the index (or distance) information of each operand has to be considered. We call this as the **weighted scan** pattern and its parallel design is shown in Figure 6.5.

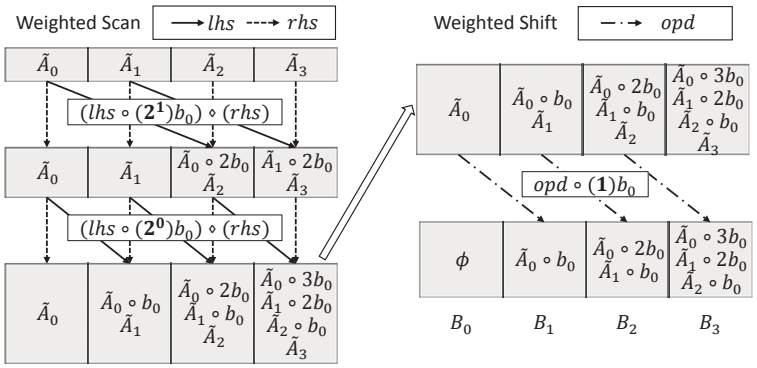


Figure 6.5: Parallel design of the weighted scan-based compensation computation. The operands lhs and rhs represent the left-hand side and the right-hand side of the $\diamond$ operator.

In the design, we conduct the compensation computation in two stages. (1) the weighted scan. This is an iterative computation to consider the effects of each preceding operand lhs on the current operand rhs. Suppose the size of the input array $\tilde{A}$ is n, we need $log_2 n$ steps (from $log_2 n - 1$ to 0) to finish the weighted scan. In each step, an entry lhs will contribute the weight $(2^i)b_0$ to the entry rhs with the distance (2^i). For example, as shown in the figure, when the input array is 4, there are 2 steps in the weighted scan. In the step $i = 1$, the lhs operand $\tilde{A}_0$ contributes $(2^1)b_0$ to the rhs operand $\tilde{A}_2$, leading to $(\tilde{A}_0 \circ (2^1)b_0) \diamond \tilde{A}_2$ on the position of $\tilde{A}_2$. Note, if the position of rhs is less than the current distance 2^i, the lhs doesn't need to contribute anything to rhs. (2) the weighted shift. According to Equation II-2, $B_{i,j}$ stores only the summation of previous weighted $\tilde{A}_{i,u}$ with u up to $j - 1$. Thus we need to compensate the previous results from the weighted scan to eliminate the effects of current operand. This can be achieved by shifting each item and add an additional weight b_0, as shown in the right part of Figure 6.5.

In order to better fit the underlying architecture of GPU, our implementation carries out the warp-aware SIMD computation by explicitly operating data at the register level. Algorithm 6.2 shows our weighted scan GPU kernel for the operator $(+, \cdot)$. The function `blk_wscan` in line 2

[2]In the following sections, for brevity, the coefficient before b_0 means the number of $\circ$ operations on b_0, e.g., $2b_0$ equals to $\prod_{k=1}^{2} b_0$.

processes data assigned to each block. First, we load current data to `rhs` (line 8-9), followed by a series of warp-level shuffle operations to realize the inner-warp weighted scan (line 11-15) using the scaling weight `w` with the distance `i` (line 13). The intermediate results are stored on shared memory. Then, a single thread will handle the inter-warp weighted scan over the intermediate results, where the weight grows by the distance of a `WRP_SIZE` (line 18-26). Finally, we broadcast the values on shared memory (representing the effects from preceding warps) to the local value in current warp; still, the weight should be scaled up with the local index (line 29). Note, in line 31, `f` determines if an additional weight `w` is needed for modifying the current value (corresponding to the aforementioned weighted shift).

The function `cpst_based_comput` in line 34 is a recursive function to deal with the data exceeding a block size and the basic case is identified in line 39. The function `blk_wreduce` is a variant of `blk_wscan` to perform weighted reduction operations over the data for each block, whose intermediate results are stored in `part_d`. Then, we recursively call the weighted scan to process `part_d` using the weight with the distance of `BLK_SIZE` (line 45). At last, `blk_wscan` is used to carry out the inner-block weighted prefix sum (line 47).

Algorithm 6.2 Weighted prefix sum for the operator $(+, \cdot)$

```
1 __global__ // BLK_SIZE: block size; WRP_SIZE: warp size
2 void blk_wscan(float *in, float *out, int n,
3      float w, float *partial, bool f) {
4   __shared__ float smem[BLK_SIZE/WRP_SIZE];
5   // gid: global idx; tid: thread idx; bid: block idx;
6   // lid: lane idx; wid: warp idx;
7   float rhs, lhs;
8   if(tid == 0) rhs = partial[bid];
9   else rhs = (gid<n)?in[gid-1]:0;
10  /* Inner-warp weighted prefix-sum */
11  for(int i = (WRP_SIZE>>1); i >= 1; i >>= 1) {
12    lhs = __shfl_up(rhs, i);
13    if(lid >= i) rhs = lhs*__powf(w, i) + rhs;
14  }
15  if(lid == WRP_SIZE-1) smem[wid] = rhs;
16  /* Inter-warp weighted prefix-sum */
17  __syncthreads();
18  if(tid == 0) {
19    float lhs2 = smem[0]*w, rhs2;
20    smem[0] = partial[0];
21    for(int i=1; i<BLK_SIZE/WRP_SIZE; i++) {
22      rhs2 = smem[i];
23      smem[i] = lhs2;
24      lhs2 = lhs2*__powf(w,WRP_SIZE)+rhs2*w;
25  }}
26  __syncthreads();
27  /* Inner-warp broadcast */
```

```
28   rhs = rhs +
29         ((!lid)?smem[wid]:smem[wid]*__powf(w, lid));
30   /* Modification */
31   if(f) if(tid != 0) rhs *= w;
32   if(gid < n) out[gid] = rhs;
33 }
34 void cpst_based_comput(float *in, float *out, int n,
35     float base, float w, bool f=true) {
36   dim3 blks(BLK_SIZE, 1, 1);
37   dim3 grds(CEIL_DIV(n, BLK_SIZE), 1, 1);
38   // Device malloc part_d for partial results
39   if(dimGrid.x == 1) {
40     // D2H: copy base to part_d
41     blk_wscan<<<grds, blks>>>(in, out, n, w, part_d, f);
42     return;
43   }
44   blk_wreduce<<<grds, blks>>>(in, out, n, w, part_d, f);
45   cpst_based_comput(part_d, part_d, grds.x,
46     base, pow(w,BLK_SIZE), false);
47   blk_wscan<<<grds, blks>>>(in, out, n, w, part_d, f);
48 }
```

One can also implement the weighted scan-based compensation method on GPUs by leveraging the scan functions from GPU library, e.g., Thrust and ModernGPU. The appendix shows how to prepare corresponding customized comparators for the library-based scan functions, which will be used as one of the baselines in the evaluation.

6.5.2 Synchronizations on GPUs: Global vs. P2P

As discussed in § 6.2.3, the compensation-based method may encounter the performance degradation due to the synchronizations within and between rows. The weighted scan method can mostly mitigate the synchronization overhead within a row; while between rows, the synchronization still affects the performance, because existing compensation-based solutions [62, 87] schedule thread blocks in a row-by-row manner, and each thread block have to wait for the finish of all others before processing the next row. We call this scheduling method as the global synchronization. On the contrary, the tile-based solutions [24, 110] use a pipeline-like mode: the tiles in a same row will be assigned to a thread block and be processed sequentially, which guarantees the horizontal dependencies; when the computations on a tile finish, it will trigger the processing on tiles below through a lightweight peer-to-peer (P2P) synchronization, e.g., spin locks, which guarantees the vertical dependencies.

Figure 6.6 exhibits these two methods to solve Algorithm 6.1 over different working matrices

by varying their dimensions of m and n. When m and n are close to each other, sufficient tile-level parallelism can be exposed, since there are many parallel tiles along the anti-diagonal. Thus, in the tile-based method, the poor performance of sequential processing at the beginning and the ending diagonals can be effectively hidden. In this scenario, i.e., m and n are close, processing the matrix row by row would cause high overhead due to the frequent global synchronizations. This is shown in the right part of Figure 6.6. On the other hand, when m is much larger than n, the portion of the beginning and ending diagonals is not negligible in the anti-diagonal-major method; in the extreme case, the whole computation would be serialized. For example, each row/anti-diagonal only contains a single tile, and thus, no tiles can be processed in parallel. In this scenario, as shown in the left part of Figure 6.6, the row-major method with the global synchronization can provide much better performance.

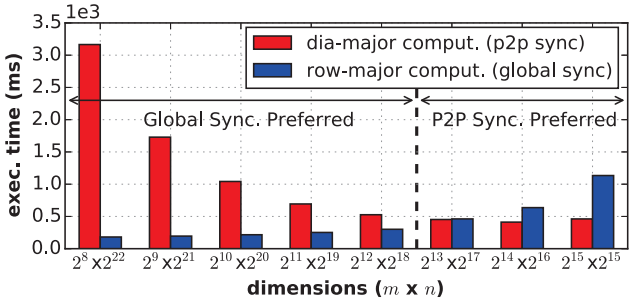


Figure 6.6: Performance comparison between the row-major computation with global sync. and the anti-diagonal-major computation with peer-to-peer (p2p) sync. The row-major kernel is based on our weight scan based method, while the diagonal-major kernel is based on a tiled solution [24].

6.5.3 Putting Them All Together

As discussed in the previous subsection, there is no single scheduling and synchronization method that can fit in different scenarios of workloads. Therefore, we propose a two-level hybrid method for wavefront problems, as shown in Figure 6.7. We use the compensation-based computation for the matrices which can expose sufficient parallelism for each row and the row number is limited to reduce the overhead of the global synchronization; otherwise, we switch to a tile and compensation hybrid method that organizes data into tiles and utilize p2p synchronization between tiles, while

inside each tile, still uses the compensation-based method to accelerate the computation. To find the optimal switching points, we build a simple offline auto-tuner based on the logistic regression to learn how the software and hardware configuration factors, including the operational intensity on each entry, the m and n of working matrix, and the generation of GPUs, determine the switch point. In the evaluation, we take 200 combinations of factors to determine the likelihood function offline.

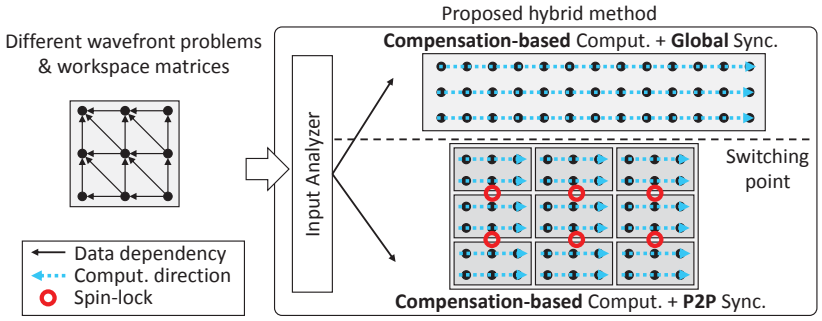


Figure 6.7: Proposed hybrid method to adapt the computation and synchronization to different wavefront problems and workspace matrices.

6.5.4 Library-based Implementations

It is invalid to use 'address-of' operator (&) for indexing on GPUs, since the data are controled explicitly in memory hierarchies with different address spaces. Moreover, restricted by the interface of comparators, we can only define the behavior of two given operands rather than the scan itself. Based on these, we implement the custom comparator in Algorithm 6.3. A new data structure `concat_t` is introduced to associate the original value `v`, its index `i`, and the flag `f` to mark if its distance needs modification (weighted shift). Then, the comparator is shown from line 6: `k` is the distance between the two operands to add weights on `lhsv` in line 11 or 14. `f` makes sure the weight is modified if the weighted shift occurs (e.g., line 11). The branch in line 10 guarantees the `lhsv` is always preceding `rhsv`.

Algorithm 6.3 Custom comparator in library-based solution for Algorithm 6.1, where the operator combination is $(+, \cdot)$ and the weight is 0.5.

```
1 typedef struct {
```

```
2   float v; int i; bool f = false;
3 } concat_t;
4 template<typename T=concat_t> struct bin_opt {
5   __device__
6   T operator()(const T &lhs, const T &rhs) const {
7     int k = abs(rhs.i - lhs.i);
8     float lhsv, rhsv;
9     int idx = max(lhs.i, rhs.i);
10    if(lhs.i < rhs.i) {
11      lhsv = lhs.v * __powf(0.5, lhs.f?k:k+1);
12      rhsv = rhs.v * (rhs.f?1:0.5);
13    } else {
14      lhsv = rhs.v * __powf(0.5, rhs.f?k:k+1);
15      rhsv = lhs.v * (lhs.f?1:0.5);
16    }
17    T res(lhsv+rhsv, index, true);
18    return res;
19 }};
```

6.6 Evaluation

We conduct the experiments on two generations of NVIDIA GPUs, i.e., Tesla K80 and P100. The specifications are listed in Table 6.1. First, we evaluate the performance of the core kernel in the compensation-based solution. Then, we investigate how the tile sizes affect the performance of our hybrid method. Finally, we report on the performance of the wavefront problems solved by our method compared with state-of-the-art optimizations.

Table 6.1: Experiment Testbeds

	Tesla K80-Kepler	Tesla P100-Pascal
Cores	2496 @ 824 MHz	3584 @ 405 MHz
Multiprocessors (MP)	13	56
Reg/Smem per MP	256/48 KB	256/48 KB
Global memory	12 GB @ 240 GB/s	12 GB @ 720 GB/s
Software	CUDA 7.5	CUDA 8.0

6.6.1 Performance of Compensation-based Kernels

We first study the weighted scan performance in the compensation-based solution by comparing the performance of our own design in Algorithm 6.2 with the scan functions based on Thrust and ModernGPU. The customized comparators for the library-based solutions are shown in appendix. As indicated by the four wavefront problems in § 6.4, we only need three combinations of binary

operators, i.e., $(+, \cdot)$, $(max, +)$, and $(+, +)$. The input matrices contain random sizes of m and n varying from 2^{14} to 2^{28}.

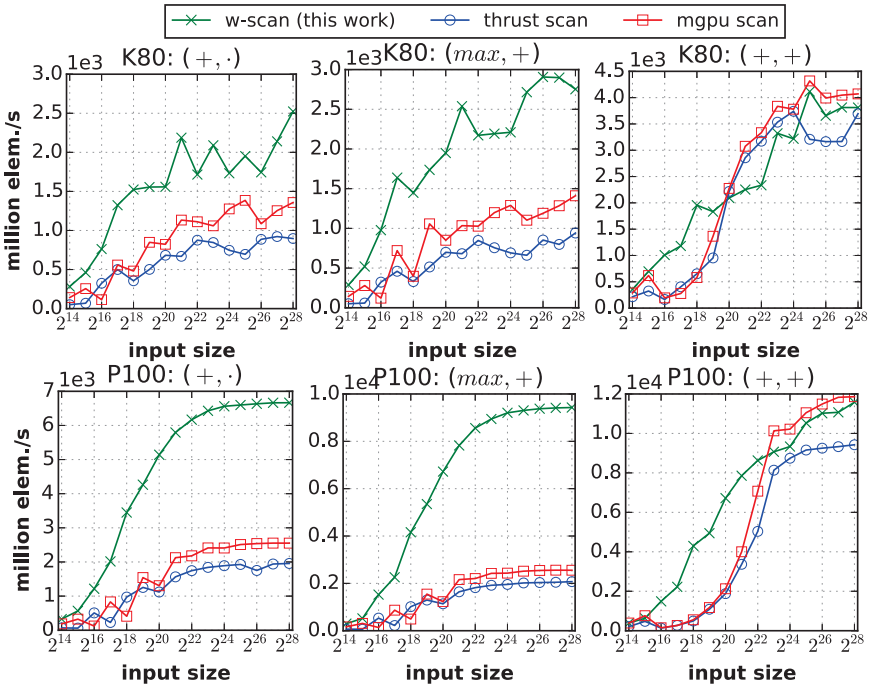


Figure 6.8: Throughput comparison of the weighted scan kernels.

As shown in Figure 6.8, for the case of $\circ \neq \diamond$, i.e., $(+, \cdot)$, $(max, +)$, our design can yield significant performance improvements over Thrust and ModernGPU, achieving an average of 3.5x (4.7x) and 2.4x (3.9x) speedups on K80 (P100). The implementation of ModernGPU explicitly exploits GPU register for data reuse and permutation[3]. However, our implementations not only take advantage of GPU registers, but also optimize the performance due to the following two reasons: (1) we calculate the distance-related weights more efficiently within the kernels; while the library-based methods put all the checking and calculating in the comparators, leading to redundant computations; (2) our algorithm directly operates on the original data and keep track of their location in GPU kernels; while the library-based design has to pack and unpack such information before and after the actual computation, causing extra performance penalty.

For the case of $\circ = \diamond$, i.e., $(+, +)$, where there is no need to deal with the varied weights,

[3]Since ModernGPU is an open source library, our analysis for library-based solutions is mainly based on it.

our solution falls back to a typical scan algorithm and can achieve comparable performance to the highly-optimized library codes. We also observe that our design is particularly effective for the middle range of input sizes. For example, it can deliver an average 5.5x improvements for the inputs ranging from 2^{16} to 2^{22} on P100. This is due to the different parallel strategies. In ModernGPU, each thread is "coarsened" to handle multiple data elements to better utilize the on-chip memory. However, this might result in the degraded GPU occupancy that less threads can be running in a multiprocessors (MP) to hide memory latency for middle-sized inputs. As a contrast, considering the potential heavy use of registers for the weight computation, we schedule a thread to process one element at one time. Besides, this thread-data scheduling strategy can also avoid uncoalesced memory transaction.

6.6.2 Performance of Hybrid Kernels

Optimal Tile Sizes

Our hybrid method conducts compensation-based computation in tiles when sufficient parallelism is available on anti-diagonals. Thus we first investigate how the tile sizes influence the performance. In the experiment, a large square matrix of 2^{15}x2^{15} is used to represent the case with sufficient anti-diagonal parallelism. In addition, we maximize the shared memory usage (40 KB for each block and other 8 KB for the inter-warp weighted prefix-sum), by using the *persistent thread block* mechanism [24, 160], where each MP only hosts one thread block to avoid deadlock for the spin-lock in the p2p synchronization. The tile sizes (height * width) are shown in Figure 6.9. The width corresponds to the thread block size, meaning threads will perform the row-major compensation-based computation. In Figure 6.9, we observe that our hybrid method prefers rectangular tiles, because they allow more threads to handle entries in parallel and the resources of registers and shared memory for intermediate values can be more efficiently utilized. For the SOR and SW, the complex weight computation needs more thread warps per block to exploit the high parallelism and data reuse in registers. In contrast, for the SAT and IHist, the computation is relatively simple and small blocks are sufficient. In the following experiments, we set tile sizes

to 10x1024 for SOR and SW on K80 and P100, and 20x512 (40x256) for SAT and IHist on K80 (P100). For the other tiling-based solutions, i.e., *tile* and *hypertile* method in § 6.6.2, similar tuning procedures are performed, and we select the best tile sizes for them (i.e., the tiles 80x128 or 40x256).

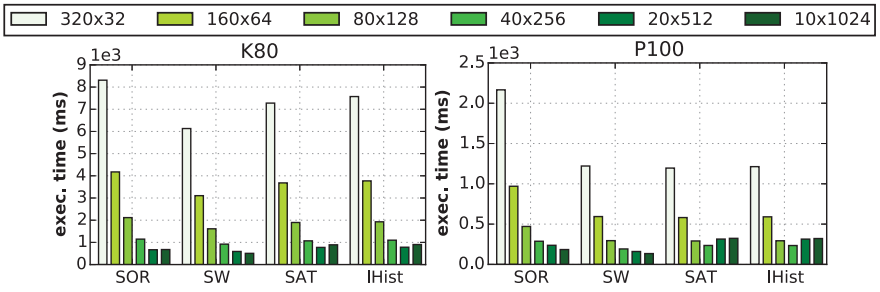


Figure 6.9: Performance of our hybrid method with varying tile sizes (height * width).

Comparison to Previous Work

Now, we evaluate the wavefront problems optimized by our method and state-of-the-art solutions. We fix the total size of the working matrix A to 2^{30} with varying dimensions as shown in Figure 6.10. The dashed vertical lines mark the switching points in our hybrid method to use the global and p2p synchronization, whose calculation is based on the auto-tuner presented in § 6.5.3. For the tiling-based methods, the *tile* kernel uses the original row-major data layout, while the *hypertile* uses the hyperplane tiles with the anti-diagonal major layout via the affine transformation. Most of the codes can be found in previous research [24]. For the library-based strategy, *lib-mgpu* and *lib-thrust* are the compensation-based solution with the global synchronization using ModernGPU and Thrust libraries, respectively.

We first focus on the left parts of the vertical dashed lines, where our hybrid methods use the weighted scan with the global synchronization. For SOR and SW, the library-based solutions can achieve an average of 3.7x (4.3x) speedups over tiling-based ones on K80 (P100), because a matrix (height * width) with the longer width can expose more parallelism in a row and at the same time the shorter height places less demands on global synchronization. These scenarios will cause severe serialization of tiling-based solutions, which explains the drastic improvements from our

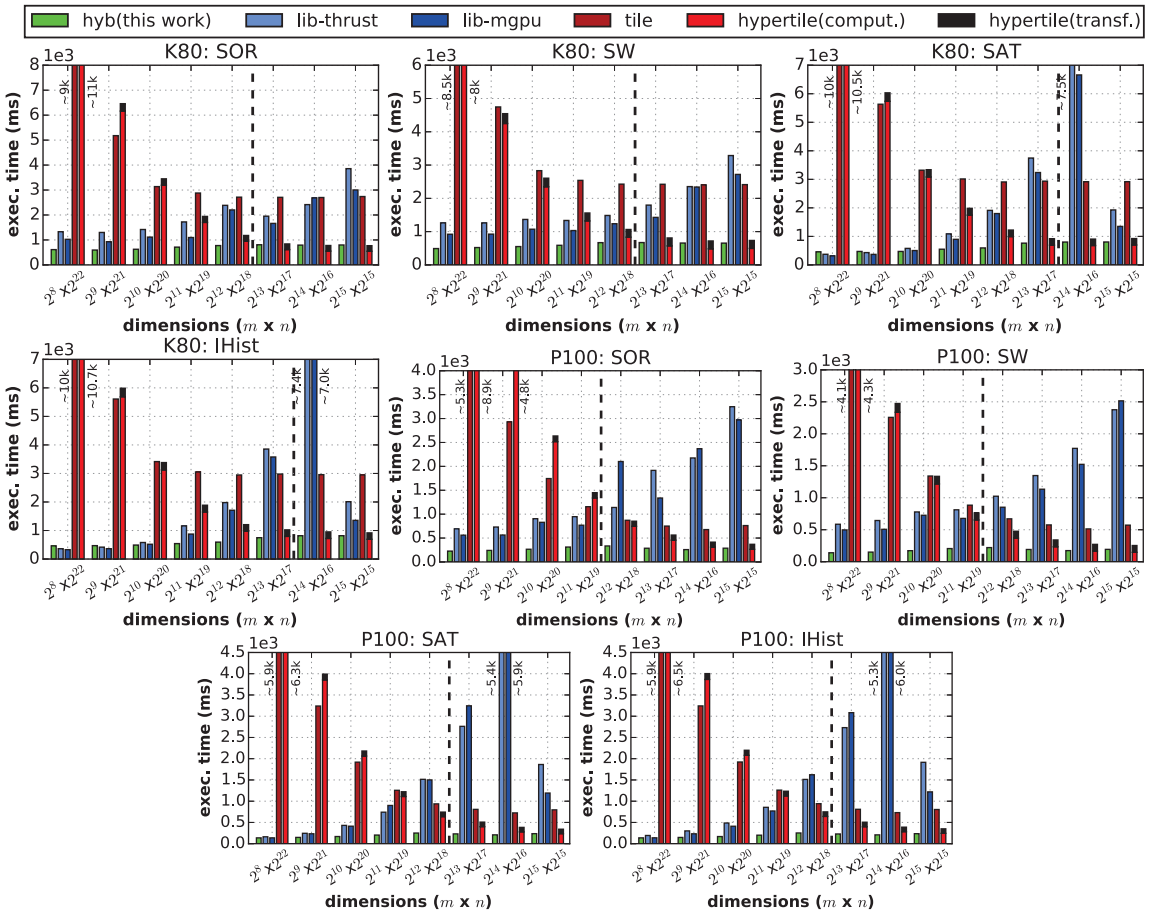


Figure 6.10: Performance comparison of the library-based (*lib-thrust, lib-mgpu*), tiling-based (*tile, hypertile*) and our hybrid method on different input matrices (height * width). The transformation of data layouts in *hypertile* is also presented. The vertical dashed lines indicate the switch points in our method.

solution of up to 22.1x (43.3x) speedups on K80 (P100). Compared to the library-based solutions, our design can provide an average of 1.9x and 3.1x speedups on K80 and P100, respectively. This can mainly attribute to our native support to the complex weight computation and elimination of pack and unpack overhead, as discussed in § 6.6.1. For SAT and IHist, our design can deliver the significant speedups: up to 4.8x (6.5x) speedup over the library-based solutions on K80 (P100) when the width of the input matrix falls into the range of 2^{16} to 2^{22}, which is consistent with the results in § 6.6.1.

Then, we focus on the right parts of the dashed lines, where our hybrid methods switch to the p2p synchronization. In these cases, the performance of library-based solution deteriorates

significantly, as more expensive global synchronizations are required. As a contrast, the tiling-based solutions exhibit superior performance as the square-like matrices contain more parallel tiles along anti-diagonals. Compared to the *tile* kernel, our solution takes advantage of both row-major computation and lightweight local synchronization, achieving an average of 3.6x (3.1x) speedups on K80 (P100). For the *hypertile* kernel, its computation is improved significantly for the square-like matrices due to the increased number of entries in each tile that exposes more parallelism opportunities; however, the transformation overhead becomes non-negligible. Our method, by contrast, is able to provide an average of 1.1x speedup on K80 if the transformation overhead is considered, and on P100 we can achieve to an average of 1.8x speedup. Even if we only consider the computation part, our method can still yield an average of 1.3x speedups on P100.

6.6.3 Discussion

Precision: For the integer datatype, our compensation-based method can obtain exactly same results with the original methods, e.g., tiling-based ones. However, we also need to consider the precision for float and double datatypes, because changing the computation order may lead to different rounding results. In SOR experiments, we observe the small relative errors are around 10^{-6} if the `float` datatype is used. The error can be further reduced to 10^{-8} for `double` datatype. We believe this is acceptable to the applications using float and double datatypes.

Generality: Theorem 1 poses the requirements on the operators $\circ$ and $\diamond$ along the horizontal dimension; however, in practice, the requirements can be loosened or differentiated along the vertical and diagonal dependencies (e.g., the negation in SAT from § 6.4). On the other hand, the proof demonstrates this compensation-based method only relies on standard properties of binary operators. Therefore, it could benefit applications in a more general data dependency (e.g., FSM [85]) than the wavefront pattern that only has the dependencies with the horizontal, vertical, and anti-diagonal neighbors.

6.7 Chapter Summary

In this chapter, we target on the compensation-based parallelism for wavefront loops on GPUs. We prove that for the compensation-based method, if the accumulation operator is associative and commutative, and the distribution operator is either distributive over or same with the accumulation operator, breaking through the data dependency and changing the sequence of computation operators properly will not affect the correctness of results. We also propose a highly efficient design of the compensation-based parallelism on GPUs, which uses the weighted scan-based GPU kernels to accelerate the computation and the tiling method to optimize data access and synchronization. Experiments demonstrate that our work can achieve significant performance improvements for four wavefront problems on various input workloads.

Chapter 7

Data Reuse in Stencil Computations

7.1 Introduction

Spatial blocking is a critical memory-access optimization that seeks to put spatially reusable data in fast memory (e.g., L1 cache, scratchpad memory, or registers) before actual computation. It has been proven to be effective in utilizing the parallel computing potential of modern accelerators, especially for stencil kernels, where the kernels perform the same computations and data-access patterns over each cell in a multi-dimensional grid. Extensive research efforts have been taken to explore different blocking schemes and develop high-performance stencil programs [117, 61, 128, 129].

In stencil computations, each cell is visited multiple times by its neighbors with the computation sweeping over a spatial grid. Consequently, cache blocking should be done to avoid unnecessary off-chip DRAM loads. Besides global memory, modern GPUs come with multiple low-latency cache levels within each compute unit (CU): (1) *L1 cache*: hardware-managed cache; (2) *scratchpad memory*: fast, programmable memory that is shared by threads assigned to the same CU, but which developers must *explicitly* manage; and (3) *registers*: fastest memory that can be accessed by each thread. In addition, recent GPUs support data exchange between threads in the same wavefront [9] or warp [120].

Different spatial blocking techniques have been proposed for these caches. By using scratch-

pad memory, one can explicitly load the requisite stencil data into cache from global memory. Then, all the working threads synchronize before doing the actual computation. After the computation, the results are stored back to global memory [117]. With the regularity of access pattern in stencils (based on Cartesian grids), simply relying on the L1 cache can also provide competitive performance [149, 111, 80]. That way, developers only need to focus on the workload partitioning and thread organization. In addition, the advent of register-based data exchange between threads enables each thread to load data into its individual registers and then directly communicate with the threads who own its neighboring data [61, 25, 73].

However, optimizing stencil kernels via spatial blocking introduces three major challenges. *First*, writing blocking code requires substantial coding effort – especially when using registers, as developers must handle the complex and convoluted data communication patterns amongst threads. For stencils with different dimensionalities, where communication patterns must change accordingly, developers must possess extensive coding expertise to reorganize the threads and recalculate the data exchange patterns. *Second*, different GPU architectures have different ISAs, specifications, and run-time configurations – all of which impact the communication patterns, and in turn, lead to rewriting of the stencil codes. For example, the sizes of hardware scheduling unit (e.g., wavefront) and data exchange instructions differ between AMD and NVIDIA GPUs, causing issues with code portability for the stencil kernels. *Third*, even when a selected stencil is mapped onto a selected GPU, the redesign of the kernel still requires changes in the target cache levels (e.g., scratchpad memory or registers) or blocking strategies (e.g., 2D, 2.5D, or 3D blocking schemes).

While existing stencil frameworks for parallel code generation and performance auto-tuning focus on mapping an entire stencil computation onto an accelerator with dedicated blocking optimizations [147, 112], we focus on a *cross-platform framework* called GPU-UNICACHE that automatically generates spatial blocking codes for different stencils, GPU architectures, and cache levels, while still allowing developers the option to change their desired stencils. That is, GPU-UNICACHE analyzes the characteristic parameters of both stencils and GPUs as input and generates highly-optimized blocking codes for the designated cache level. For example, for register-based methods, the GPU-UNICACHE framework handles the distribution of grid data to minimize

register conflict and realizes the communication patterns of given blocking strategies by minimizing the number of permute/shuffle instructions.

The contributions of our work include the following: (1) GPU-UNICACHE, a framework to automatically generate spatial blocking codes for stencil kernels on GPUs, and (2) a comprehensive evaluation of the GPU-UNICACHE framework on AMD and NVIDIA GPUs. GPU-UNICACHE not only improves programming productivity by unifying the interfaces of spatial blocking for different stencils, GPU architectures, and cache levels; but it also provides high performance by optimizing data distribution, indexing conversion, thread communication, and synchronization to facilitate data access in GPU kernels. Compared to hardware-managed memory (L1 cache), with single-precision arithmetic, our automatically-generated codes deliver up to 1.7-fold and 1.8-fold speedups at the scratchpad memory level and register level, respectively, when running on an AMD GCN3 GPU and up to 1.6-fold and 1.8-fold, respectively, when running on a NVIDIA Maxwell GPU. For double precision, it delivers up to a 1.3-fold speedup on both GPU platforms. Compared to the state-of-the-art benchmarks (incl. Parboil [141], PolyBench [126], SHOC [54]), it can also provide up to 1.5-fold improvement.

7.2 Motivation and Challenges

7.2.1 Stencil Computation

A stencil computation defines the point p in a multi-dimensional grid at time t (stored in v) that is updated based on a function f of surrounding grid points $\mathbb{P}$ at the previous time step $t-1$ (stored in u). It sweeps the stencil computation over all the points at t before moving to the next time step $t+1$ and then the next. The stencil order h defines the distance between the central point p and its farthest neighbor $q \in \mathbb{P}$. The stencil size N is $|p \bigcup \mathbb{P}|$. Equation (7.1) shows a stencil computation pattern in a 2-dimensional (i.e., 2D) grid; its h is 1; and N is 5. For brevity, we refer to this stencil as "2D5Pt."

$$v_{i,j} = f(\mathbb{P}) = a_0 u_{i-1,j} + a_1 u_{i+1,j} + a_2 u_{i,j} + a_3 u_{i,j-1} + a_4 u_{i,j+1} \tag{7.1}$$

Due to the application-specific *f*, there exists no common libraries for stencils that users can directly use without defining the specific stencil patterns. Thus, to evaluate the potential benefits of our GPU-UNICACHE library framework, we collect a benchmark of stencils representing different dimensionalities and memory-access patterns, as noted in Table 7.1. Although we distinguish between low and high data-reuse kernels for each dimensionality, their arithmetic intensities (AI), defined as FLOPS/byte [155], are similar.[1] Additionally, data-access patterns differ in that one-dimensional (i.e., 1D) stencils make unit-stride access, whereas higher-dimensional stencils make non-contiguous access of memory. Irrespective of the access pattern, if the data can be ideally cached and reused, the stencil computation will benefit with respect to performance.

Table 7.1: Summary of the stencil computations

Name	**jacobi-1d** [126]	**gaussian X7** [177]	**jacobi-2d** [126]	**seidel-2d** [126]	**heat-3d** [126]	**jacobi-3d** [55]
Stencil	1D3Pt	1D7Pt	2D5Pt	2D9Pt	3D7Pt	3D27Pt
h	1	3	1	1	1	1
N	3	7	5	9	7	27
#FLOPS	5	13	9	17	13	53
Bytes	12	28	20	36	28	108
AI	0.42	0.46	0.45	0.47	0.46	0.49

7.2.2 Spatial Blocking Schemes

In spatial cache-blocking optimizations, one needs to load data into the cache, and then do the stencil computation using the cached data before the results are stored back to global memory. Figure 7.1 shows examples of different blocking schemes. A 2D stencil can be optimized by using 2D tiles. Likewise, for 3D stencils, a 3D block is a natural way to buffer data for high reuse. Alternatively, one can use a 2D-slice layout, allowing stencil computations to be carried out from the bottom to the top (i.e., 2.5D blocking). In addition, temporal blocking [117], consisting of multiple rounds of spatial blocking within the cache, can also be used.

Figure 7.1 shows which data domains are loaded into the cache. However, when designing real GPU kernels, one must explore implementation details, such as how to load domain data. As shown in the figure, when loading a 2D-square tile, the task of loading boundary points (Bs) outside the

[1] We use the *Roofline* model with emphasis on loading data from memory of a machine model without cache.

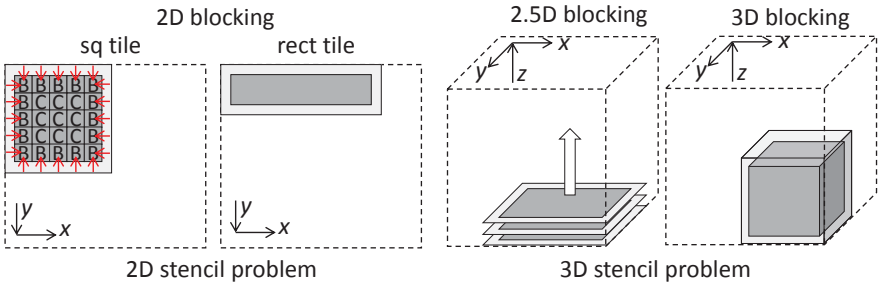


Figure 7.1: Blocking schemes for 2D and 3D stencils.

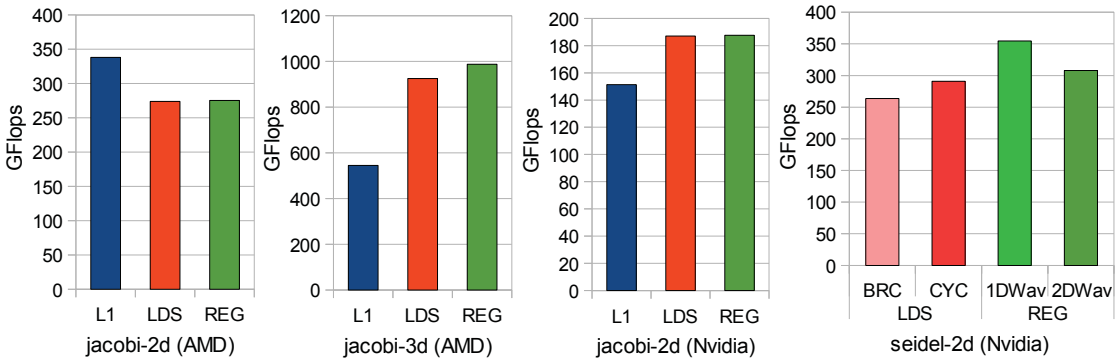
current tile is assigned to point Bs rather than Cs. This method introduces branch divergence to the GPU kernels. Alternatively, with an (additional) amount of remapping calculation, the data can be evenly assigned to threads (not shown in the figure). In addition, when loading data, one must decide on either a square tile for high data reuse or a rectangle tile for more regular memory access. All the above choices will affect the later realization of fetching data from caches, which in turn, produces significant performance differences (as captured in Figure 7.2b).

On the other hand, the temporal cache-blocking essentially adds another dimension (i.e. time) to the spatial blocking by conducting multiple rounds of computations over reusable data (loaded in cache). This procedure also follows a fixed or predictable pattern, which matches the idea of our GPU-UNICACHE framework. However, considering that the spatial blocking is more fundamental and essential in blocking techniques, we focus on analyzing the patterns in spatial blocking for stencils in this paper. Our idea is general and can be used to construct temporal blocking as reported in previous research [117, 129].

7.2.3 Challenges

Performance It is critical to take advantage of the cached memory hierarchy in a GPU via blocking optimizations. Though modern GPUs provide different options, such as L1 cache, scratchpad memory, and registers, it is still unclear where data should be cached for the different stencils. Figure 7.2a shows two types of stencils (i.e., jacobi-2d and jacobi-3d) that prefer *different* cache options for the same platform. On the other hand, for the different blocking strategies, developers need to adjust the optimizations to achieve best performance. Figure 7.2b shows the diversified

performance of "seidel-2d" stencil on two types of caches (i.e., LDS and registers), for each of which we use different loading styles. These simple examples illustrate the challenges encountered by programmers when implementing stencil codes on GPUs. They also demonstrate that choosing a "one-size-fits-all" optimization strategy for any kind of stencil or GPU would be ineffective.



(a) Diff Stencils/GPUs (b) Diff Variants

Figure 7.2: Diversified performance of stencils under different situations. BRC and CYC refer to different loading modes, while 1DWav and 2DWav mean different wave layouts (discussed in § 7.5).

Programmability The second challenge encountered by developers is the programmability issue. They might be involved in complex implementation details, where, for example, one needs to figure out how to efficiently organize domain data into individual registers across threads in a wave while using registers as cache. Many other factors can affect how GPU kernel codes are written, including stencil types, GPU architectures, and blocking strategies. To address these issues, we present a framework called GPU-UNICACHE to automatically generate spatial-blocking codes that manage data reuse within a GPU.

7.3 GPU-UniCache Framework

Figure 7.3 highlights the major components of our GPU-UNICACHE framework: (1) feature extraction, (2) code generation, and (3) stencil buffer library. Feature extraction discovers the user-defined conigurations, the stencil types, and the underlying GPU platforms. Code generation automatically produces stencil codes for the different cached systems, i.e. L1 cache, scratchpad

memory, and registers. In essence, the codes focus on loading from and storing to the global store, during which GPU-UNICACHE needs to deal with problems like indexing, synchronization, workload partition, and thread communications. Finally, the stencil buffer library wraps the generated codes inside a set of functions with uniform interfaces. We provide details of how our GPU-UNICACHE library framework works below.

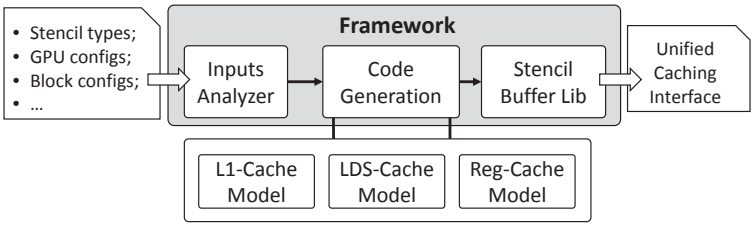


Figure 7.3: An overview of the GPU-UNICACHE framework.

At the first step, the inputs analyzer component conducts analysis on the user input parameters and some features extracted from the underlying GPU platforms. This information includes stencil types (e.g., stencil order, stencil size), block configs (e.g., block and warp dimensions, blocking strategies), GPU specifications (e.g., built-in warp size, ISAs about data exchange). These parameters assist the framework in realizing the generalized stencil patterns.

The code generation component uses three models on each cache level for given stencils. In L1-cache model, it mainly uses the hardware's capability to access the contiguous data. In scratchpad memory model, it solves the problem of eliminating branches, index conversion under different blocking strategies. In register model, it solves how the data are distributed into registers of each thread, and how the threads communicate with each other to obtain desired neighbors. The generated codes are for three purposes: cache declaration, which allocates required space for scratchpad or registers; cache initialization, which loads central and halo data from global store; cache fetch, which fetches the desired data by using the offsets away from the current point. The codes are finally wrapped into a set of functions by the stencil buffer library component. Developers can call the functions through unified interfaces to design dedicated stencil kernels for efficacy.

7.3.1 GPU-UNICACHE API

The GPU-UNICACHE library provides the operation functions for moving data between on-chip storage and off-chip DRAM memory for stencil computations. Figure 7.1 lists the cache classes and their core member functions. The GPU-UNICACHE API is object oriented. The base class defines interface to initialize the cache, i.e. `init()`, and access the locations with given relative offsets, i.e. `fetch()`. Since all these member functions are executed on GPU devices, we have `__device__` qualifiers for NVIDIA GPU, and `[[hc]]` attribute specifiers for AMD GCN3 GPU. Internally, the classes use `_load()` and `_store()` to access locations in cache using local indices. Sub-classes are devised for different cache storage.

Algorithm 7.1 Interface of GPU-UNICACHE functions

```
template<class T>
class GPU_UniCache
{
protected:
  virtual T _load(int z, int y=0, int x=0)=0;
  virtual void _store(T v, int z, int y=0, int x=0)=0;
public:
  virtual void init(T *in, int off, int mode=CYCLIC)=0;
  virtual T fetch(int z, int y, int x, int tc_i=0)=0;
};
// Derived classes
class L1Cache : public GPU_UniCache{ ... };
class LDSCache: public GPU_UniCache{ ... };
class RegCache: public GPU_UniCache{ ... };
```

In Table 7.2, we list the member functions and corresponding descriptions. Note, all the member functions need location information of the running thread, such as global or local index. For NVIDIA GPU, no specific arguments need to be transferred to the functions, since CUDA supports built-in constants regarding the thread index. For AMD GCN3 GPU, we need to explicitly transfer such information of `tiled_index` by reference. For brevity, we don't list them in the table. In practice, developers create a inherited GPU-UNICACHE object (e.g., LDSCache) within a device kernel to declare an empty cache space. After the data have been stored into cache, they can use the object to get data in neighbors. We present a working example to show how the GPU-UNICACHE API works.

Table 7.2: GPU-UNICACHE and its subclass member functions

Function Name	Description
(Constructor)(**int** dz, **int** dy, **int** dx, **int** h, **int** tc)	Constructs a specific cache, initializing its attributes of the stencil domain dimensions(dz, dy, dx), order(h), and thread coarsening factor(tc).
_load(**int** z, **int** y, **int** x)	Loads data from cache using local indices(z, y, x). If only z is set, z is the register index.
_store(**T** v, **int** z, **int** y, **int** x)	Stores data(v) to cache using local indices(z, y, x). If only z is set, z is the register index.
init(**T** *in, **int** off, **int** mode)	Initializes the cache from source *in*. The target domain will be located using info. got from the constructor or user-defined offset *off*. Workload distribution can switch by mode, which currently supports CYCLIC and BRANCH styles.
fetch(**int** z, **int** y, **int** x)	Fetches data using the offsets(z, y, x) away from the central point.

7.3.2 GPU-UNICACHE Example

We use an example of 2D5Pt GPU kernel (Equation 7.1) in Figure 7.2 to illustrate how to use the API. This stencil simply uses a 2D blocking optimization strategy and registers as cache. In line 4, the kernel declares a RegCache with thread coarsening factor of 4, which means each thread will perform 4 iterations of stencil computation over 4 points. It is demonstrated that using thread coarsening is useful for stencils [25, 107] and we will discuss it in details in § 7.4. Then, we fill in the register cache by calling `init()` member function. Here, we use the loading mode as CYCLYC in line 5, which means the kernel will distribute all the domain data evenly into each thread in a round-robin fashion. While performing the actual stencil computation (line 8 to 12), users only need to provide relative offsets of target neighbors and the `fetch()` will figure out where to get the data.

The GPU-UNICACHE APIs aim to facilitate the process of accessing cached data in stencils on Cartesian grids and allow GPU programmers to develop efficient kernel codes optimized by different blocking strategies. The codes can be easily changed to work on another cache levels for more efficiency. We can also use multiple types of caches at the same time by declaring different GPU-UNICACHE objects. This could benefit programs which place significantly high resource pressure on a single type of cache. More importantly, the kernel codes are portable across different GPU platforms. We will cover how the GPU-UNICACHE framework assists in automatically generating the codes for these functions in § 7.4.

Algorithm 7.2 Example of 2D5Pt stencil CUDA kernel using RegCache APIs with thread coarsening factor *csr_fct* of 4, which means each thread will update 4 cell points. The current cell point is located by using *csr_id*.

```
__global__
void kern_2d5pt(float *in, float *out, float a0-4)
{
  RegCache<float> buf(m, n, h, 4);
  buf.init(in, 0, CYCLIC);
  // each thread processes 4 points since csr_fct = 4;
  for (csr_id = 0; csr_id < 4; csr_id++)
    out[/*global_idx w/ offset csr_id*/] = a0 * buf.fetch(-1, 0, csr_id)+
                                           a1 * buf.fetch( 0,-1, csr_id)+
                                           a2 * buf.fetch( 0, 0, csr_id)+
                                           a3 * buf.fetch( 0, 1, csr_id)+
                                           a4 * buf.fetch( 1, 0, csr_id);
}
```

7.4 Code Generation

In this section, we put emphasis on the register and scratchpad memory methods, since both methods need to explicitly handle how to access the data. For the member functions of sub-classes in § 7.3, we generate the real codes based on our generalized code constructs and algorithms.

7.4.1 Input Parameters

The input parameters are used by the framework to understand the features of target stencil and running environment. Table 7.3 shows the list of the required parameters in three types. Among them, the crs_fct and crs_dim are used specifically for thread coarsening in RegCache methods. RegCache methods handle the computation based on the unit of wave, whose thread number is usually much smaller than a thread block, meaning we need to load more halo data. In contrast, thread coarsening [107] is an optimization technique to increase the workload of each thread and enable loaded data to be more reused. Therefore, we use thread coarsening to compensate low data-reuse rate in RegCache methods.

Table 7.3: List of input parameters for the code generation

Parameter Name	Description
	User-defined thread layout
blk_dim[3]	Thread block dimensions in exponent notations (with a base of 2). The least significant dimension is blk_dim[0].
wav_dim[3]	Wave dimensions in exponent notations (with a base of 2). The least significant dimension is wav_dim[0].
crs_fct	Thread coarsening factor. It defines the number of iterative process the wave will conduct.
crs_dim	Thread coarsening occurs along which dimension.
	Stencil computation characteristics
h	Stencil order.
N	Stencil size.
sten_dim	Stencil dimensionality.
	GPU architecture characteristics
blk_sync()	Built-in block-level synchronization barrier.
wav_size	Number of threads in a wave.
wav_shfl(v, id)	Machine-dependent register-level data exchange instruction. Data exchange occurs between calling thread and thread *id* on value *v*.

7.4.2 RegCache Methods

We first look at a specific example of "2D9Pt" stencil and analyze its data distribution and computation patterns. Figure 7.4 shows a wave with thread layout of $2 \times 4 = 8$ (i.e., $wav\_dim[1] = 1$ and $wav\_dim[0] = 2$) loads required grid points of $4 \times 6 = 24$ (i.e., $h = 1$). The 24 points are distributed evenly into registers of each thread in a round-robin fashion, meaning $\lceil 24/8 \rceil = 3$ iterations and registers are needed. To achieve this CYCLIC loading method, we map these threads to assigned points by using $(y, x) = ((i \cdot wav\_size + tid)/(2^{wav\_dim[0]} + 2h), (i \cdot wav\_size + tid)\%(2^{wav\_dim[0]} + 2h))$, where tid is thread index and i is iteration number. Therefore, for example, thread 0 will deal with points (0,0), (1,2), (2,4) and store them in register r, s, t respectively.

The destination points (gray area) are updated by fetching registers from their neighbors. However, this raises two further questions: 1, which threads to communicate with; 2, which registers store the desired neighbors. We observe from Figure 7.4 that these information can be calculated from neighbors of thread 0 in the wave (located in the red circle). For example, when handling the northeast (NE) neighbors, we need to know the first neighbor is stored in register 0 (r) of thread 2. Then, the other neighbor thread index and register index can be calculated by each thread applying $(tid + 2 + tid/2^{wav\_dim[0]} \cdot 2h)\%wav\_size$ and $0 + (2 + tid/2^{wav\_dim[0]} \cdot 2h + tid)/wav\_size$ respectively. That way, thread 0 will interact with thread 2 on register 0 (r), and simultaneously, thread 4 will fetch value of register 1 (s) of thread 0.

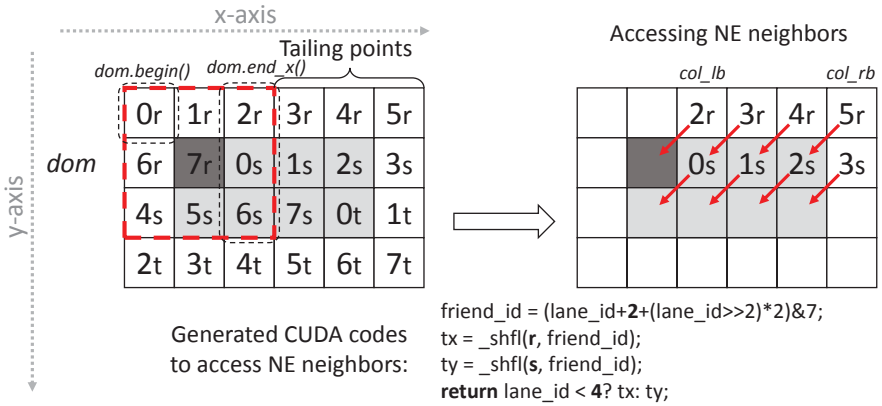


Figure 7.4: Example of data exchange for "2D9Pt" stencil. The figure illustrates 2 steps of *loading* and *computing*. For *loading*, all the data are distributed across threads in a 2x4 wave. The '2r' in cell, for example, means the corresponding value will be stored in register r of thread 2. For *computing*, the wave updates the gray area. The thread 0 in the wave is in deep gray. The CUDA codes below are to access the NE neighbors.

With the variety of stencils and options (e.g., thread coarsening factors and neighbor directions), manually calculating these parameters is a painful task. As size and complexity of the target stencil grow, so does the development cost. Therefore, in our framework, we first generalize the stencil computation in registers by means of code constructs. Then, we calculate the parameters using our proposed formula and algorithm.

Method `init()`: we only support loading method of CYCLIC rather than BRANCH in RegCache. The reasons are two-fold: (1) BRANCH mode will make boundary threads hold too many registers and thus all the other threads in the same wave have to keep same number of "idle" registers, leading to register pressure problem; (2) While accessing neighbors, extra branches are needed to distinguish the meaningful from these "idle" registers. Code constructs in Figure 7.3 show how we distribute the DRAM data to registers. The remapping occurs in line 3 to 6 and the fetched data are stored to registers (line 8).

Algorithm 7.3 Code constructs for RegCache `init()` method

```
1 // **** CYCLIC ****
2 int it = _lane_id();
3 c_0 = (wav_id0<<wav_dim[0]) + it%(2^{wav_dim[0]} + 2h);
4 c_1 = (wav_id1<<wav_dim[1]) + it/(2^{wav_dim[0]} + 2h)
5                              %(2^{wav_dim[1]} + 2h);
6 c_2 = (wav_id2<<wav_dim[2]) + it%(∏_{k=0}^{1}(2^{wav_dim[k]} + 2h));
7 reg_id = 0;
8 _store(in(off,c_2,c_1,c_0),reg_id++);
```

```
9 it += wav_size;
10 // repeat for ⌈∏_{k=0}^{2}(2^{wav_dim[k]} + 2h)/wav_size⌉ times
```

Method `fetch()`: Figure 7.4 exhibits the generalized data exchange code constructs to fetch data in given direction. The neighbor thread index is represented by *friend_id*, which depends on the parameter `F`. The registers of interest are ranged from *reg*`N1` to *reg*`N3`. Parameter `M` is the cut-off marker to select values from different registers. Here, we only use up to three data exchange operations to fetch the data, since this number fits in our benchmark of stencils and different wave dimensions. For other stencils with higher stencil order, for example, it is easy to extend the pattern to support more data exchange operations.

Algorithm 7.4 Code constructs for RegCache `fetch()` method

```
1 // **** Fetch a given neighbor ****
2 friend_id = (lane_id+F+
3             ((lane_id>>wav_dim[0]*2*h))&(wav_size-1);
4 tx = wav_shfl(regN1, friend_id);
5 ty = wav_shfl(regN2, friend_id);
6 tz = wav_shfl(regN3, friend_id);
7 return ((lane_id < M1)? tx: ((lane_id < M2)? ty: tz));
```

Figure 7.5 shows the pseudo code of calculating the parameters based on given inputs from Table 7.3 (Each direction of neighbors need a set of the parameters). We define a *domain* as a set of points surrounding the first thread in a wave. Since threads might be coarsened by the factor of `crs_fct`, there are multiple domains stored in `dom` (line 1 and 17). In the function calculating parameter `F` (line 1), the identifier of the starting point in each domain is computed in line 8. Then, we sweep all the other points and record the relative order within the wave (line 9). The order is the parameter `F`, which can be used later by other threads in the wave to find neighbors towards the same direction (line 2 in Figure 7.4). Additionally, we record the round number in line 10, indicating how many registers we have already used in each thread. Note, the out-of-domain points should be skipped in line 12 to 14.

Subsequently, we need to calculate which registers are used to store the target neighbors in the wave and how to select data from these registers. This can be achieved by computing parameters `N` and `M` through the function in line 17. The register identifier in line 24 indicates the register storing the first value of neighbors toward the given direction. Then, we can calculate the boundaries of

neighbors of the entire wave (line 27 and 28, also shown in Figure 7.4) which will be used to skip other irrelevant points. If an incoming point is identified as using a new register in line 36 and it is within the boundary in line 38, the new register is recorded with the counter `cnt` showing the cut-off location.

Algorithm 7.5 Algorithms to calculate the `F`, `N`, and `M`.

```
1 void calculate_F(domain* dom)
2 { // compute param F used in friend_id formula
3   for(int c = 0; c < csr_fct; c++)
4   {
5     for(auto pt: dom[c]) // each point in domain
6     {
7       if(pt == dom[c].begin())
8         id = c * ∏_{k=0}^{csr_dim-1}(2^{wav_dim[k]} + 2h);
9       pt.F  = id % wav_size;
10      pt.rid = id / wav_size;
11      id++;
12      if(pt == dom[c].end_x()) // skip tailing points
13        id += 2^{wav_dim[0]};
14      if(pt == dom[c].end_yx()) //skip tailing lines
15        id += 2^{wav_dim[1]} · (2^{wav_dim[0]} + 2h);
16 } } }
17 void calculate_NM(domain* dom)
18 { // compute param N M used in data exchange patterns
19   for(int c = 0; c < csr_fct; c++)
20   {
21     for(auto pt: dom[c]) // each point in domain
22     {
23       int i = 1, j = 1;
24       int reg_id = pt.rid;
25       pt.N[i++] = reg_id;
26       int skipped_pts = pt.F + reg_id * wav_size;
27       int col_lb = skipped_pts % (2^{wav_dim[0]} + 2h);
28       int col_rb = lb + 2^{wav_dim[0]};
29       int cnt = 1;
30       bool reg_update = false;
31       while(cnt < wav_size)
32       {
33         skipped_pts++;
34         int col_id = skipped_pts % (2^{wav_dim[0]} + 2h);
35         int wav_id = skipped_pts % wav_size;
36         if(wav_id == 0) // end of current wave
37           reg_id++, reg_update = true;
38         if(col_lb <= col_id && col_id < col_rb)
39         {
40           if(reg_update) // mark the divergence
41           {
42             pt.N[i++] = reg_id;
43             pt.M[j++] = cnt;
44             reg_update = false;
45 } } } } } }
```

After we calculate these parameters, we replace $wav\_shfl()$ with "__shfl()" for NVIDIA GPUs and "amdgcn_bs_bpermute()" for AMD GCN3 GPUs. Note, for AMD GCN3 GPUs, we

need to right shift the *friend_id* by 2 (§ 2.1.2). If the datatype is double precision number, we will first split the value into two 32-bit ones, perform two data exchange instructions, and then concatenate the results.

7.4.3 LDSCache Methods

Method `init()`: The major problem encountered by using scratchpad memory is conditional branching, since the sizes of thread block and working data domain don't match each other. In LDSCache, we support two loading modes: BRANCH, boundary threads handle more workloads (i.e. halo points); CYCLIC, threads address the data domain in a round-robin fashion by remapping themselves. This way, we can minimize the branches at the expense of more index conversion calculation. The code constructs of BRANCH are comprised of multiple conditional statements to assign additional workloads to boundary threads. The CYCLIC code constructs are similar with RegCache method (in Figure 7.3), but replaced with the granularity of *blk_size* rather than *wav_size*. In addition, the destination locations are changed to scratchpad memory. Note, we need to use an explicit synchronization *blk_sync()* at the end of loading.

Method `fetch()`: This method is straightforward and we only need to use the thread local index to fetch desired data, since the loaded points follow original data layouts and are same by using BRANCH or CYCLIC mode.

7.5 Evaluation

7.5.1 Experiment Setup

In the section, we evaluate the stencil codes using GPU-UNICACHE library. The details of the two platforms are listed in Table 7.4. We conduct the tests using both single precision and double precision numbers.

The benchmark of stencils are listed in § 7.2.1. We optimize them using the GPU-UNICACHE APIs with different blocking strategies. The blocking strategies used in 1D and 2D stencils are

Table 7.4: Experiment Testbeds

	AMD	**NVIDIA**
Model	Radeon R9 Nano	GeForce GTX 980
Codename	Fiji XT	GM204(Maxwell)
Cores	4096	2048
Core frequency	1000 MHz	1126 MHz
Register file size	256 kB*	256 kB
L1/LDS/L2	16/64/1024 kB	-/96/2048 kB
Memory bus	HBM	GDDR5
Memory capacity	4096 MB	4096 MB
Memory BW	512 GB/s	224 GB/s
GFLOPS float/double	8192/512	4612/144
Software	HCC/ROCM 1.2	CUDA 7.5

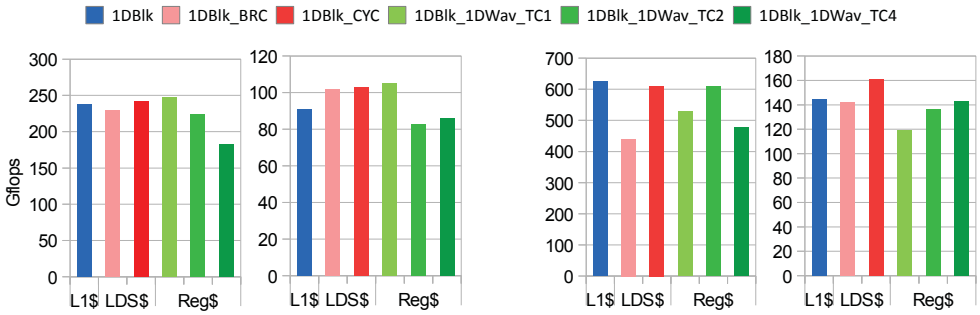
* Each CU has 256 kB vector registers and an additional 8 kB scalar registers.

straightforward, while in 3D kernels, we use 2.5D and 3D blocking [117] (labeled as 1DBlk, 2DBlk, 2.5DBlk, and 3DBlk). For LDSCache version, we try both loading modes: BRANCH and CYCLIC (as BRC and CYC), while for RegCache, we vary dimensionalities of wave: 1D and 2D (as 1DWav and 2DWav). The 1DWav is 64×1 for AMD and 32×1 for NVIDIA, while the 2DWav is 8×8 and 8×4 on the two platforms. The sizes of data set are 2^{25}, $2^{12} \times 2^{12}$, $2^8 \times 2^8 \times 2^8$ for 1D, 2D, 3D stencils respectively. The test iterates for 100 times. The metric we use is GFLOPS calculated by $(FLOPS \cdot dim2 \cdot dim1 \cdot dim0)/time$.

7.5.2 AMD GCN3 GPU

We use the best speedup achievable when the kernel is optimized by RegCache or LDSCache, if not mentioned otherwise. For 1D stencils, the performance numbers are shown in Figure 7.5. Different cache levels show very similar performance. Using LDSCache or RegCache do not show significant improvements over L1Cache, because 1D stencil has unit-stride memory access pattern where the data can be effectively put into cache by hardware. The optimal achieved with RegCache leads to 15% improvement; LDSCache achieves up to 13% improvement. We also notice that performance deteriorates with LDSCache in BRANCH mode for gaussianX7 stencil, due to extra loading operations to perform data transfer between L1 cache to scratchpad memory and overhead of branches, which can be offset by using CYCLIC mode.

For 2D stencils, L1Cache methods still exhibit competitive performance on AMD GCN3 GPUs (shown in Figure 7.6a and 7.6b). The maximum speedups with LDSCache and RegCache

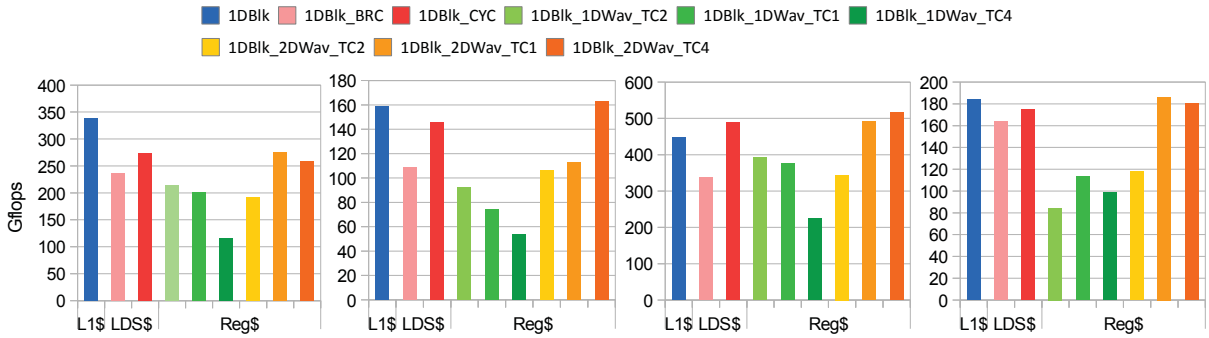


(a) jacobi-1d *l*: single; *r*: double (b) gaussianX7 *l*: single; *r*: double

Figure 7.5: 1D stencils with HCC by GPU-UNICACHE on AMD GPU.

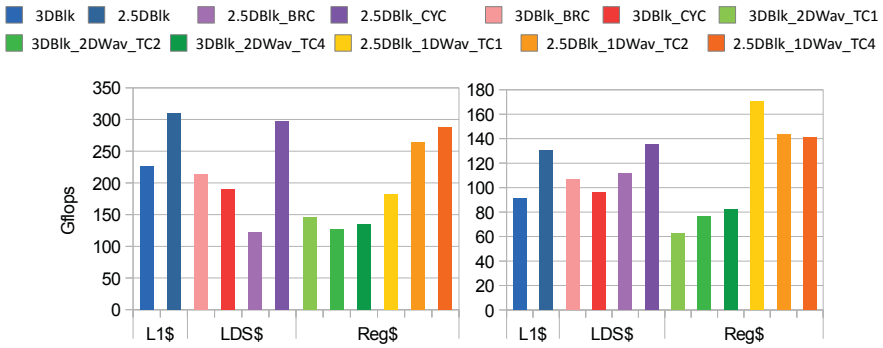
surpass L1Cache when data reuse grows in seidel-2d stencil. In the LDSCache solution, we first observe that 2D stencils are more sensitive to the loading mode, where CYCLIC mode is generally superior to BRANCH mode averaging 25% better performance, since more branches are needed to load surrounding data in 2D stencils. The maximum improvement of LDSCache over L1Cache is 9%. In RegCache solution, using 1D wave variant is particularly effective over 2D wave. 1D wave have longer dimension while conducting memory access, which can better utilize the hardware bandwidth but at the expense of relatively low data reuse. 2D wave, by contrast, exhibits high data reuse rate. For example, considering the wave size of 64 on AMD GPUs, if we organize the wave threads as 64 by 1, we have to load 66*3=198 elements for the 2D problem with stencil order of 1. However, if we organize them as 8 by 8, we only need to load 9*9=81 elements. On the other hand, the former thread layout can load the data in less memory transactions, leading to its superior performance. If we consider the effect of thread coarsening on performance, 2D wave can barely benefit from it, because the narrowed access stride makes it bound by memory latency. We record the best speedup of RegCache is 15% over L1Cache.

Figure 7.7 shows more significant and diversified speedups with RegCache and LDSCache. Differences in the performance are first reflected in the speedups of the 2.5D and 3D blocking. 2.5D blocking gives a speedup of 1.58x over 3D blocking on average. This is because 3D blocking has smaller dimensions for the block than 2.5D blocking if we assume the blocks have same number of threads. That way, uncoalesced memory access would occur even though it has better data reuse rate. In the low data-reuse kernels (heat-3d), L1Cache solution is similar with LDSCache,



(a) jacobi-2d *l*: single; *r*: double (b) seidel-2d *l*: single; *r*: double

Figure 7.6: 2D stencils with HCC by GPU-UNICACHE on AMD GPU.



(a) heat-3d *l*: single; *r*: double

(b) jacobi-3d *l*: single; *r*: double

Figure 7.7: 3D stencils with HCC by GPU-UNICACHE on AMD GPU.

while the additional gain is achieved from using RegCache, resulting in up to 30% improvement. This is mainly because of the elimination of explicit synchronization of RegCache in this iterative 2D method. For high data-reuse kernels (jacobi-3d), LDSCache or RegCache are critical to get optimal performance. The best improvements are 1.70x for LDSCache and 1.81x for RegCache over their L1Cache counterpart. Moreover, we prefer CYCLIC mode in LDSCache in high data-reuse kernels, observing that the overhead of branches is significantly high, because, for example,

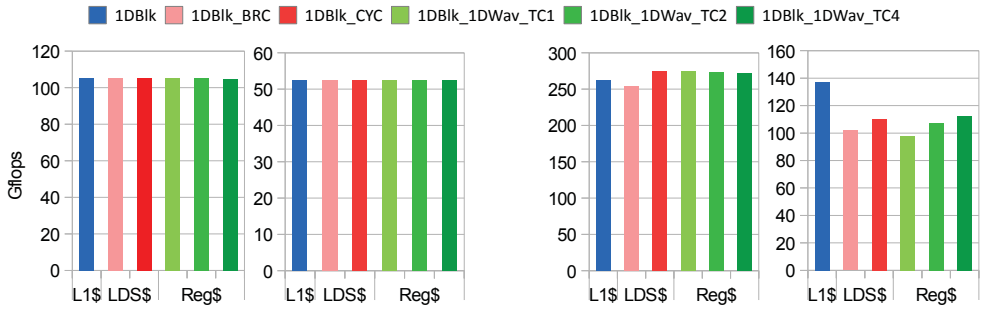
the jacobi-3d stencil has nearly 4 times more halo elements to load than the heat-3d stencil. For RegCache solutions, we only record the performance of 2D wave in 3D blocking, because using 1D wave instead would be equivalent to the 2.5D blocking with 1D wave. Also, we only show the results of 1D wave for 2.5D blocking, since this strategy is preferable and has been demonstrated. Similarly, 3D blocking encounters higher memory latency, making itself benefit little from thread coarsening. As a contrast, 2.5D blocking with 1D wave improves significant after applying thread coarsening.

The speedups given by thread coarsening in the cases of double precision numbers are less consistent, where the optimal thread coarsening factors are only 1 or 2, since operating doubles requires more space from register files and register pressure would be more easily reached. Furthermore, because the built-in data exchange of 64-bit data is not supported, we need more operations to achieve the same functionality, i.e. split the data into two 32-bit data, do two permutes, and concatenate the two data together.

7.5.3 NVIDIA Maxwell GPU

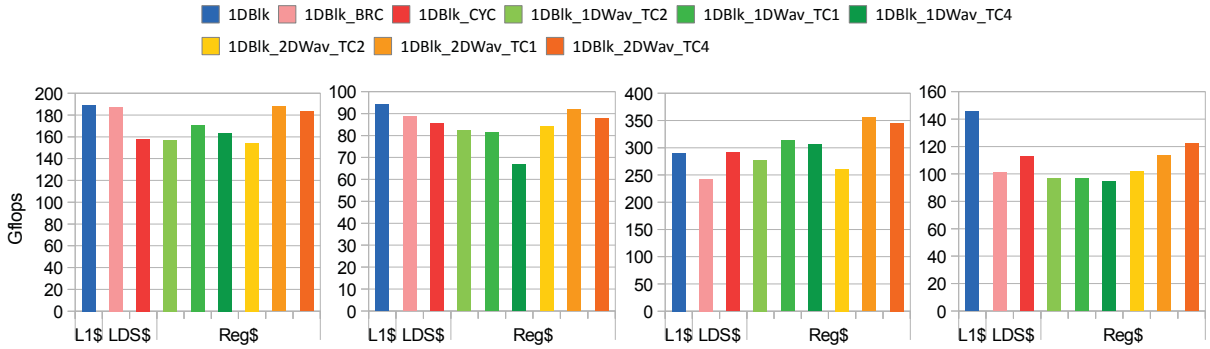
On NVIDIA GPU, Figure 7.8, 7.9b and 7.9c show the performance of 1D and 2D stencils on Maxwell GPU. For low data reuse kernels, the optimal is achieved by simply using L1Cache. This demonstrates the need to "opt-in" to enable the global caching in the Maxwell GPU (§ 2.1.2), which is particularly effective for solving 1D and 2D arrays. The benefits of using LDSCache or RegCache become obvious when there are high data reuse, achieving up to 5% and 20% improvements for gaussianX7 and seidel-2d stencils respectively. However, for double datatype, we observe a slowdown experienced by LDSCache and RegCache. For LDSCache, since the shared memory banking in Maxwell only supports 4 bytes width per bank, overhead of accessing 8-byte data is accordingly higher; for RegCache, more instructions are needed to conduct every data exchange operations for 8-byte data. Similar to 2D stencils on GCN3 GPU, CYCLIC mode is of necessity in seidel-2d kernels and 1D wave is preferable because all the threads in the same wave are able to access consecutive locations to achieve a coalesced memory transaction.

The performance of 3D stencils shown in Figure 7.10 shows diversified speedups after apply-



(a) jacobi-1d *l*: single; *r*: double (b) gaussianX7 *l*: single; *r*: double

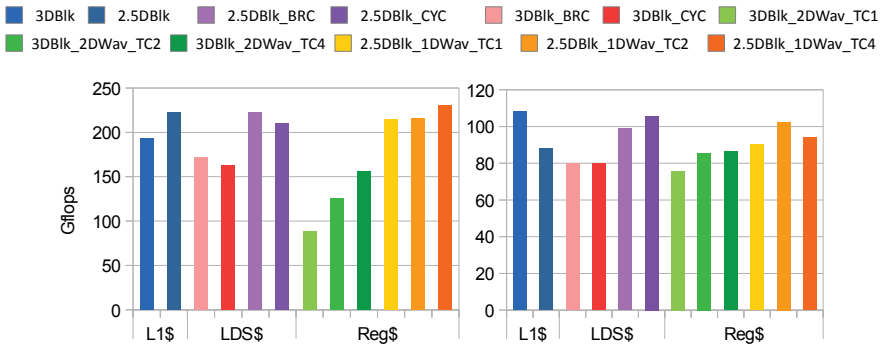
Figure 7.8: 1D stencils with CUDA by GPU-UNICACHE on NVIDIA GPU.



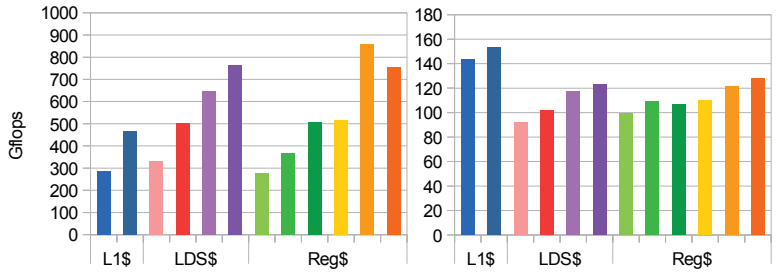
(b) jacobi-2d *l*: single; *r*: double (c) seidel-2d *l*: single; *r*: double

Figure 7.9: 2D stencils with CUDA by GPU-UNICACHE on NVIDIA GPU.

ing different cache levels. Speedups of using LDSCache or RegCache range from a few percent on the low data reuse kernels up to 1.64x and 1.83x for high data reuse kernels with LDSCache and RegCache respectively. The 2.5D blocking is still preferred in the 3D stencils and for float datatype, we record 4% to 12% improvements of the best RegCache over LDSCache. 2.5D blocking needs to iteratively load a 2D slice before conducting actual computation, which will result in overhead of block-level synchronization. In contrast, RegCache can eliminate this explicit synchronization, leading to better performance. For double datatype, using our L1Cache interface can provide competitive performance, mainly because the overhead of operating doubles in RegCache and LDSCache is relatively high in Maxwell.



(a) heat-3d *l*: single; *r*: double

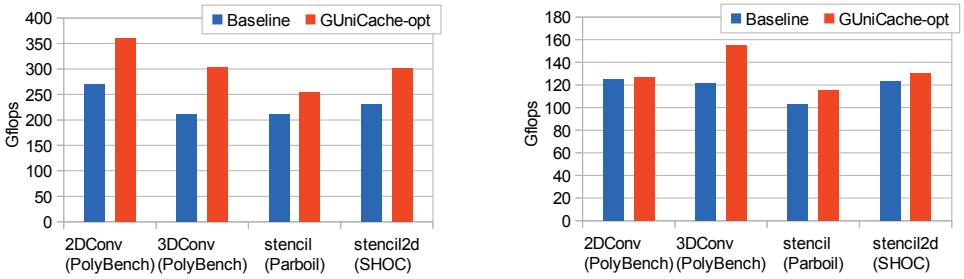


(b) jacobi-3d *l*: single; *r*: double

Figure 7.10: 3D stencils with CUDA by GPU-UNICACHE on NVIDIA GPU.

7.5.4 Speedups to Existing Benchmarks

In the section, we optimize third-party benchmarks by using GPU-UNICACHE. They have been optimized via different spatial blocking strategies: *2DConv* and *3DConv* (PloyBench [126]) use 2D and 3D blocking with L1 cache respectively; *stencil* (Parboil [141]) is a "3D7Pt" stencil optimized by 2.5D blocking with shared memory; *stencil2d* (SHOC [54]) adopts 2D blocking with shared memory. We optimize these kernels by using GPU-UNICACHE and only report the best performance. Figure 7.11 presents the results of the comparisons on NVIDIA GPU (There are no equivalent benchmarks using HCC yet). For single datatype, all the optimal GPU-UNICACHE codes are using RegCache and can outperform the baselines for up to 1.5x. For double datatype, GPU-UNICACHE selects L1Cache for 2D stencils and LDSCache for 3D stencils, mainly because the overhead of register shuffle on double grows. The best improvement is as high as 1.3x speedup.



(a) single datatype (b) double datatype

Figure 7.11: GPU-UNICACHE optimized codes vs. existing stencil benchmarks optimized by spatial blocking on NVIDIA Maxwell GPU.

7.5.5 Discussion

Running Parameters In the experiments, we use the same settings for the kernels to evaluate the performance for two main reasons. First, we can limit the variables to the options of cache levels and focus on the correlation between performance and different GPU-UNICACHE functions. One exception is that we need to shrink the total number of threads as the thread coarsening factor grows up in RegCache kernels. Second, the GPU-UNICACHE APIs are also designed to enable GPU programmers access to the different caches simultaneously, especially when the programs encounter high pressure on one single type of resource. Therefore, we need to test the APIs under the same circumstances. Despite of this, we still observe the diversified speedups, indicating an auto-tuning framework is of necessity [106, 72]. We leave this as our future work.

Register Pressure Using too many registers in GPU programs could reduce the active waves per CU. Table 7.5 shows the profiling numbers of register usage from the jacobi-3d stencil which exhibits the highest data reuse rate. First, as the coarsening factor doubles, the number of registers increases logarithmically, because coarsening technique can improve data reuse. Additionally, 2.5D blocking generally uses more registers than 3D blocking as we discussed in § 7.5.2. The kernel of 2.5D blocking with the best performance can attain 40% occupancy on both GPU platforms. However, considering its better memory access and high FLOPs, the active waves can still effectively utilize the computing resources.

Table 7.5: Register usage of jacobi-3d stencil*

GPU	L1Cache		LDSCache				RegCache					
	3DBlk	2.5DBlk	3DBlk BRC	3DBlk CYC	2.5DBlk BRC	2.5DBlk CYC	3DBlk 2DWav TC1	3DBlk 2DWav TC2	3DBlk 2DWav TC4	2.5DBlk 1DWav TC1	2.5DBlk 1DWav TC2	2.5DBlk 1DWav TC4
AMD	37	101	21	18	40	31	19	31	48	48	59	74
NVIDIA	32	32	30	28	31	31	32	40	56	42	56	80

* Collected by CodeXL 2.2 for AMD GPU and nvprof tool for NVIDIA GPU

7.6 Chapter Summary

In the chapter, we propose a framework GPU-UNICACHE to automatically generate the library codes to access cached data L1, scratchpad memory, and registers of the spatial blocking optimizations for stencils computations. The codes to achieve these functionalities are automatically generated by our GPU-UNICACHE framework based on the information of stencils and underlying architectures. The GPU-UNICACHE has facilitated efficiently accessing cache-loaded data without a tedious code rewrite, a major advantage in designing different stencil codebases. The evaluation demonstrates that we can get up to 1.8x improvements by only changing the GPU-UNICACHE API calls on different GPU platforms.

Chapter 8

Conclusion and Future Work

The key goal of this dissertation is to develop and propose innovative solutions to performance portability issues in parallel computing. In this dissertation, we have successfully applied the methodology built on parallel patterns to solve critical issues in accelerating scientific kernels. This methodology leads to a series of novel and effective automation frameworks that takes into consideration characteristics of both algorithm and hardware. For example, our propose ASPaS framework [75] features the fast search for the optimal instruction combinations based on given data-reordering patterns and ISAs.

This dissertation is motivated by the increasing complexity and cost encountered by developers to manage efficient and portable parallel codes for various parallel systems. We launch from two aspects to discuss our approaches using parallel patterns: (1) domain-specific languages to tackle the vector data reordering on x86-based platforms and the data-thread binding in GPU environments [75, 78, 73]; (2) algorithmic skeletons to deal with SIMD operations, data dependencies, and data reuse problems in different computational kernels [76, 79, 77]. These work is realized by performing extensive analysis to uncover and identify the performance/programmability issues, formalizing patterns from target algorithms, and building corresponding frameworks to generate efficient codes across platforms. We demonstrate, in this dissertation, improved performance efficiency from various modern accelerator without the need for developers programming the low level.

8.1 Summary

The growing demand for extreme performance from modern accelerators is driving the use of programming models defined by each vendor (e.g., CUDA, HCC) with low level codes (e.g., compiler intrinsics). This will inevitably lead to the complexity and cost on application development, and even worse, result in the performance portability issues, meaning the efficient codes for one platform may be not portable to another. *This dissertation along this line handles the above issues by introducing an additional abstraction layer with parallel patterns to exploit both characteristics of algorithms and hardware, to generate efficient codes across platforms, and to mitigate the tricky details in programming accelerators.*

We first focus on the research problem on data reordering in parallel sort, a commonly used kernel in many scientific applications. We present an automation framework ASPaS [75, 78] to generate highly efficient vector codes on x86-based processors, i.e., CPUs and MICs. In the framework, we formalize the base cases (e.g., sorting networks) as sequences of comparing and reordering operators using DSL. ASPaS performs fast search for the vector instructions, by exploiting the symmetric features of data reordering patterns, and from our proposed pools storing ISA-friendly operation primitives. The ASPaS-generated codes can outperform compiler optimized codes and deliver substantial performance gains over existing sorting tools from libraries, e.g., STL, BOOST, and TBB, on Ivy Bridge, Haswell, and Knights Corner architectures.

Second, we target the segmented sort and associative data-thread binding issues in GPUs. The demand for processing a large amount of independent workloads is on the rise in the "big data" era. Segmented sort is one example. The existing parallel solutions on GPUs often fail to take full advantage of computing resources. We first uncover the performance issue of their processing the skewed data, a common input scenario in real-world applications. Then, we present a novel and efficient segmented sort [73] that can adaptively adjust its solutions to process different segments. In this mechanism, we generalize the register-based sort algorithm as a N-to-M data-thread binding and communication problem: how to assign the target data to threads and how the data reordering changes correspondingly. We formalize the binding patterns and look for the best realization to perform the sorting networks on multiple memory hierarchies of registers, shared memory, etc.

We show the performance improvements over previous tools from CUB, ModernGPU, CUSP. The performance advantage is also demonstrated by integrating our segmented sort into real-world applications in bioinformatics and linear algebra.

Third, we tackle the performance portability problem caused by vectorizing sequence alignment algorithms. The existing vectorization strategies—"striped-iterate" and "striped-scan"—all require explicit programming with low-level and platform-specific vector intrinsics. Thus, we present AAlign framework [76], where the SIMD operations are generalized as a series of vector modules to hide the underlying platform-specific vector implementations. In addition, we reveal that the existing vectorization strategies can only deliver optimal performance on certain inputs. Based on these findings, AAlign supports a new input-agnostic hybrid method to achieve optimal performance regardless input scenarios. AAlign accepts sequential codes following our generalized patterns and transforms them into efficient parallel codes. The results show our hybrid method can yield superior performance over other sequence alignment tools, such as SWPS3 and SWAPHI.

This dissertation then looks at a more general computation—wavefront loops—extended from the sequence alignment algorithms. For the parallel strategies, we claim using a compensation-based computational order is preferable over the traditional anti-diagonal major ones on GPUs. The compensation-based method is originally from parallelizing alignment algorithms in bioinformatics. Therefore, we first need to answer the research question: under which circumstances, can the computation-based method be used to optimize general wavefront loops?. In this work [79], we uncover the boundary of this method by resorting to the relations of our introduced distributive and accumulative operators. Based on the findings, we present a highly efficient "weighted" scan as the skeleton to parallelize different wavefront problems on GPUs. Experiments demonstrate that our optimized four wavefront loops can achieve high performance not only for various input workloads, but also across generations of GPUs.

Finally, this dissertation tackles the data reuse problem in stencil computations. Modern accelerators feature multiple levels of memory hierarchy (including L1 cache or local memory), meaning efficient data reuse strategies should be carefully designed to take full advantage of them. The framework GPU-UniCache [77] is therefore proposed to automatically generate library codes

to access cached data in L1, local memory, and registers in GPU settings. The code generation is based on the information of given stencils and underlying architectures. These library codes have uniform APIs to lift the burden from developers on designing separate stencil codes for different memory hierarchies as cache. With GPU-UNICACHE, the performance improvements can be achieved by only substituting the data access codes with GPU-UNICACHE API calls.

8.2 Future Directions

This dissertation focuses on the research on parallel patterns: how parallel patterns bridge the gap between practical applications and modern accelerators. The research questions for future work can be stated in terms of two aspects: (1) Platforms, how can we efficiently exploit other emerging accelerators, heterogeneous platforms, clusters, and clouds?; (2) Applications, how can we extend this methodology to other hot applications (e.g. computer security, deep learning, data mining, etc.)? In the following, we discuss the potential future directions from these two aspects.

8.2.1 Utilizing Heterogeneous Accelerators and Clusters

The heterogeneous platforms are increasingly adopted in state-of-the-art supercomputers [5]. However, developers might encounter two major scheduling problems, since their jobs usually need to share computing resources or migrate among computing units (e.g., job consolidation [181] and job pipeline on GPU+CPU [51, 52]. It is known that efficient solution requires careful design of scheduling algorithms for given parallel platforms and applications, which paradoxically hinders the usage of supercomputers or heterogeneous platforms. In the short term, we plan to (1) study and understand the scheduling patterns in applications, and (2) abstract these patterns to automate the code generation of scheduling that aims to exploiting different underlying resources.

As enterprises are increasingly moving their infrastructure services to the cloud with the goal of reducing the administration cost, advanced techniques should be used to enhance the quality of service experience of data-intensive applications [151, 44, 43, 42, 17, 45, 13, 14, 15, 16, 191] and reduce the cost for both cloud service providers and tenants [46, 47]. Unfortunately, such tech-

niques also require significant efforts from developers, forming a huge gap between applications and underlying distributed clusters; thus, we plan to apply our methodology so that we can generalize the essential operations as an intermediate layer, which can enable an application-managed hardware design [42].

It is also promising to build a software hardware infrastructure to measure function level performance and power consumption of HPC applications [93, 92, 95]. It enables the measurement of power consumption of system components such as CPU, memory, and PCI-E based accelerators (e.g. Intel Xeon Phi, GPU). To reduce the total energy consumption of data centers, a decentralized replica selection mechanism can be used to allocate data intensive workload across geographically located data centers [96, 97]. To improve the energy efficiency of supercomputers, different throttling techniques [94, 32, 31, 159] and their impact on performance and power consumption should be investigated.

8.2.2 Accelerating Big Data Applications for Parallel Platforms

We are particularly interested in designing solutions for the big data applications in security, data mining, and pattern matching. Mining security risks are becoming more challenging as the amount of data grows rapidly. New techniques [100, 98, 28, 40, 137, 99] are needed to detect, prioritize and measure the security risks from massive datasets. Within these designs, many search/sort/alignments algorithms are used to compose tools for detecting security risks, which actually can directly benefit from our optimized kernels to exploit parallel platforms.

Another area we are interested in is to accelerate computation patterns in data mining. These patterns oftentimes involve irregular computation and memory access, which is challenging for parallel computing. For example, many tree-based data structures are used in the Data-Aware Vaccination problem to find healthy people based on the data from social media (e.g., Twitter) [189, 190]. Another example is the CT (computed tomography) image reconstruction in medical imaging diagnostics, which needs to operate over large-scaled sparse matrices [173, 174]. The problem is challenging when applying our methods to optimize the irregular data processing, since its patterns are usually unpredictable and dynamic. One potential solution is to find the patterns by

preprocessing the data, which could benefit from our methods.

Pattern matching is an another kernel to filter information from large-scaled data. It is challenging to parallelize it onto the parallel architectures, e.g., GPU and the emerging AP (Automata Processor) [119], which involves many tricky designs, such as fine-grained data parallelism, speculative computation, and mastering low level instructions [169, 172, 168, 167, 166]. We believe many core computations can be formalized by using parallel patterns. In reality, Robotomata [171, 170] is a recent work on abstracting the pattern matching for APs, which is able to provide not only superior performance but also user-accessible programmability.

Bibliography

[1] BLAST. http://blast.ncbi.nlm.nih.gov/Blast.cgi.

[2] Clang 6: Clang Language Extensions. https://clang.llvm.org/docs/LanguageExtensions.html.

[3] Clang: a C Language Family Frontend for LLVM. http://clang.llvm.org/.

[4] NCBI-protein. http://blast.ncbi.nlm.nih.gov/protein.

[5] TOP500 Supercomputing List. https://www.top500.org/.

[6] UniProt: Universal Protein Resource. http://www.uniprot.org/.

[7] L. A. Adamic and B. A. Huberman. Zipf's Law and the Internet. *Glottometrics 3*, 2002.

[8] S. W. Al-Haj Baddar and K. W. Batcher. *Designing Sorting Networks: A New Paradigm*. Springer, 2011.

[9] AMD. *Graphics Core Next Architecture, Generation 3 Reference Guide*, 2016. Rev.1.1.

[10] S. E. Amiri, L. Chen, and B. A. Prakash. Segmenting Sequences of Node-Labeled Graphs. In *IEEE Int. Conf. on Data Mining Workshops (ICDMW)*, 2016.

[11] S. E. Amiri, L. Chen, and B. A. Prakash. SnapNETS: Automatic Segmentation of Network Sequences with Node Labels. In *AAAI Conf. on Artif. Intell.*, 2017.

[12] S. E. Amiri, L. Chen, and B. A. Prakash. Automatic Segmentation of Dynamic Network Sequences with Node Labels. *IEEE Trans. on Knowledge Data Engineer.*, 2018.

[13] A. Anwar, Y. Cheng, and A. R. Butt. Towards Managing Variability in the Cloud. In *IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, 2016.

[14] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. Taming the Cloud Object Storage with MOS. In *ACM Parallel Data Storage Workshop (PDSW)*, 2015.

[15] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. MOS: Workload-aware Elasticity for Cloud Object Stores. In *ACM Int. Symp. High-Perform. Parallel Distrib. Comput. (HPDC)*, 2016.

[16] A. Anwar, Y. Cheng, H. Huang, and A. R. Butt. ClusterOn: Building Highly Configurable and Reusable Clustered Data Services Using Simple Data Nodes. In *USENIX Workshop Hot Top. Storage File Syst. (HotStorage)*, 2016.

[17] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt. Improving Docker Registry Design based on Production Workload Analysis. In *USENIX Conf. File Storage Technol. (FAST)*, 2018.

[18] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A View of the Parallel Computing Landscape. *Commun. ACM*, 2009.

[19] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *IEEE Int. Conf. Parallel Archit. Compil. Tech. (PACT)*, 2015.

[20] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *Int. Conf. Compiler Constr. (CC), Springer*, 2010.

[21] K. E. Batcher. Sorting Networks and Their Applications. In *AFIPS Spring Joint Comput. Conf.*, 1968.

[22] S. Baxter. *ModernGPU 2.0: A Productivity Library for General-purpose Computing on GPUs*. `https://github.com/moderngpu/moderngpu`.

[23] N. Bell, S. Dalton, and L. N. Olson. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM J. Sci. Comput.*, 2012.

[24] M. E. Belviranli, P. Deng, L. N. Bhuyan, R. Gupta, and Q. Zhu. PeerWave: Exploiting Wavefront Parallelism on GPUs with Peer-SM Synchronization. In *ACM Int. Conf. on Supercomput. (ICS)*, 2015.

[25] E. Ben-Sasson, M. Hamilis, M. Silberstein, and E. Tromer. Fast Multiplication in Binary Fields on GPUs via Register Cache. In *ACM Int. Conf. Supercomput. (ICS)*, 2016.

[26] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N. Vasilache. Tiling and Optimizing Time-iterated Computations on Periodic Domains. In *IEEE Int. Conf. Parallel Archit. Compil. Tech. (PACT)*, 2014.

[27] R. C. Bose and R. J. Nelson. A Sorting Problem. *J. ACM*, 1962.

[28] A. Bosu, F. Liu, D. D. Yao, and G. Wang. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *ACM Asia Conf. Comput. Commun. Secur. (Asia CCS)*, 2017.

[29] B. Bramas. Fast Sorting Algorithms using AVX-512 on Intel Knights Landing. *CoRR*, 2017.

[30] T. M. Chan. More Algorithms for All-pairs Shortest Paths in Weighted Graphs. In *ACM Symp. Theory Comput. (STOC)*, 2007.

[31] H. C. Chang, B. Li, G. Back, A. R. Butt, and K. W. Cameron. LUC: Limiting the Unintended Consequences of Power Scaling on Parallel Transaction-Oriented Workloads. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2015.

[32] H. C. Chang, B. Li, M. Grove, and K. W. Cameron. How Processor Speedups Can Slow Down I/O Performance. In *IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2014.

[33] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In *ACM/IEEE Int. Conf. High Perf. Comput., Netw., Storage and Anal. (SC)*, 2011.

[34] L. Chen, S. E. Amiri, and B. A. Prakash. Automatic Segmentation of Data Sequences. In *AAAI*, 2018.

[35] L. Chen, K. S. M. T. Hossain, P. Butler, N. Ramakrishnan, and B. A. Prakash. Flu Gone Viral: Syndromic Surveillance of Flu on Twitter Using Temporal Topic Models. In *IEEE Int. Conf. on Data Mining ICDM*, 2014.

[36] L. Chen, K. S. M. T. Hossain, P. Butler, N. Ramakrishnan, and B. A. Prakash. Syndromic Surveillance of Flu on Twitter using Weakly Supervised Temporal Topic Models. *Data Min. Knowl. Discov.*, 2016.

[37] L. Chen, N. Muralidhar, S. Chinthavali, N. Ramakrishnan, and B. A. Prakash. Segmentations with Explanations for Outage Analysis. *Computer Science Technical Reports TR-18-02, VTechWorks*, 2018.

[38] L. Chen and B. A. Prakash. Modeling Influence using Weak Supervision: A Joint Link and Post-level Analysis. *Computer Science Technical Reports TR-18-03, VTechWorks*, 2018.

[39] L. Chen, X. Xu, S. Lee, S. Duan, A. G. Tarditi, S. Chinthavali, and B. A. Prakash. HotSpots: Failure Cascades on Heterogeneous Critical Infrastructure Networks. In *CIKM*, 2017.

[40] L. Cheng, F. Liu, and D. D. Yao. Enterprise Data Breach: Causes, Challenges, Prevention, and Future Directions. *Wiley Interdiscip. Rev.: Data Min. Knowl. Discovery*, 2017.

[41] L. Cheng, K. Tian, and D. D. Yao. Orpheus: Enforcing Cyber-physical Execution Semantics to Defend against Data-oriented Attacks. In *ACM Computer Secur. App. Conf.*, 2017.

[42] Y. Cheng, F. Douglis, P. Shilane, G. Wallace, P. Desnoyers, and K. Li. Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality. In *USENIX Annu. Tech. Conf. (ATC)*, 2016.

[43] Y. Cheng, A. Gupta, and A. R. Butt. An In-memory Object Caching Framework with Adaptive Load Balancing. In *ACM Eur. Conf. Comput. Syst. (EuroSys)*, 2015.

[44] Y. Cheng, A. Gupta, A. Povzner, and A. R. Butt. High Performance In-memory Caching Through Flexible Fine-grained Services. In *ACM Ann. Symp. Cloud Comput. (SOCC)*, 2013.

[45] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. CAST: Tiering Storage for Data Analytics in the Cloud. In *ACM Int. Symp. High-Perform. Parallel Distrib. Comput. (HPDC)*, 2015.

[46] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. Pricing Games for Hybrid Object Stores in the Cloud: Provider vs. Tenant. In *USENIX Workshop Hot Top. Cloud Comput. (HotCloud)*, 2015.

[47] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. Provider versus Tenant Pricing Games for Hybrid Object Stores in the Cloud. *IEEE Internet Comput.*, 2016.

[48] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *Proc. VLDB Endow. (PVLDB)*, 2008.

[49] M. Codish, L. Cruz-Filipe, M. Nebel, and P. Schneider-Kamp. Applying Sorting Networks to Synthesize Optimized Sorting Libraries. In *Int. Symp. Logic-Based Program Synth. Transform. (LOPSTR)*, 2015.

[50] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[51] X. Cui, T. R. Scogland, B. R. de Supinski, and W.-C. Feng. Directive-based pipelining extension for openmp. In *IEEE Int. Conf. Cluster Comput. (CLUSTER)*, 2016.

[52] X. Cui, T. R. Scogland, B. R. de Supinski, and W.-c. Feng. Directive-based Partitioning and Pipelining for Graphics Processing Units. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2017.

[53] S. Dalton, N. Bell, L. Olson, and M. Garland. CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, 2014. `http://cusplibrary.github.io/` v.0.5.0.

[54] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *ACM Workshop Gen. Purpose Process. Graphics Process. Unit (GPGPU)*, 2010.

[55] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning the 27-point Stencil for Multicore. In *Int. Workshop Autom. Perform. Tuning (iWAPT)*, 2009.

[56] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 2011.

[57] J. Demouth. Shuffle: Tips and Tricks, 2013. GTC'13 Presentation.

[58] P. Di and J. Xue. Model-driven Tile Size Selection for DOACROSS Loops on GPUs. In *Parallel Process.: Int. Eur. Conf. Parallel Distrib. Comput. (EuroPar)*, 2011.

[59] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue. Automatic Parallelization of Tiled Loop Nests with Enhanced Fine-Grained Parallelism on GPUs. In *Int. Conf. on Parallel Process. (ICPP)*, 2012.

[60] I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W.-m. Hwu. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2016.

[61] T. L. Falch and A. C. Elster. Register Caching for Stencil Computations on GPUs. In *Int. Symp. Symbolic Numer. Algorithms Sci. Comp. (SYNASC)*, 2014.

[62] M. Farrar. Striped Smith-Waterman Speeds Database Searches Six Times over other SIMD Implementations. *Oxford Bioinf.*, 2007.

[63] P. Flick and S. Aluru. Parallel Distributed Memory Construction of Suffix and Longest Common Prefix Arrays. In *ACM Int. Conf. High Perf. Comput., Netw., Storage and Anal. (SC)*, 2015.

[64] F. Franchetti, F. Mesmay, D. Mcfarlin, and M. Püschel. Operator Language: A Program Generation Framework for Fast Kernels. In *IFIP TC 2 Working Conf. Domain-Specific Lang. (DSL), Springer*, 2009.

[65] T. Furtak, J. N. Amaral, and R. Niewiadomski. Using SIMD Registers and Instructions to Enable Instruction-level Parallelism in Sorting Algorithms. In *ACM Symp. Parallel Alg. Arch. (SPAA)*, 2007.

[66] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX Symp. Oper. Syst. Des. Impl. (OSDI)*, 2012.

[67] M. W. Green. Some Improvements in Non-adaptive Sorting Algorithms. In *Annu. Princeton Conf. Inf. Sci. Syst.*, 1972.

[68] O. Green, R. McColl, and D. A. Bader. GPU Merge Path: A GPU Merging Algorithm. In *ACM Int. Conf. Supercomput. (ICS)*, 2012.

[69] M. Harris, J. D. Owens, S. Sengupta, S. Tzeng, Y. Zhang, A. Davidson, R. Patel, L. Wang, and S. Ashkiani. *CUDPP: CUDA Data-Parallel Primitives Library*. `http://cudpp.github.io/`.

[70] T. N. Hibbard. An Empirical Study of Minimal Storage Sorting. *Commun. ACM*, 1963.

[71] J. Hoberock and N. Bell. *Thrust: A Parallel Algorithms Library*, 2015. `https://thrust.github.io/`.

[72] K. Hou, W.-c. Feng, and S. Che. Auto-Tuning Strategies for Parallelizing Sparse Matrix-Vector (SpMV) Multiplication on Multi- and Many-Core Processors. In *IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, 2017.

[73] K. Hou, W. Liu, H. Wang, and W.-c. Feng. Fast Segmented Sort on GPUs. In *ACM Int. Conf. Supercomput. (ICS)*, 2017.

[74] K. Hou, H. Wang, and W.-c. Feng. Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study. In *Int. Conf. Parallel Process. Workshops (ICPPW)*, 2014.

[75] K. Hou, H. Wang, and W.-c. Feng. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In *ACM Int. Conf. Supercomput. (ICS)*, 2015.

[76] K. Hou, H. Wang, and W.-c. Feng. AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-Based Multi-and Many-Core Processors. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2016.

[77] K. Hou, H. Wang, and W.-c. Feng. GPU-UniCache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs. In *ACM Conf. Comput. Front. (CF)*, 2017.

[78] K. Hou, H. Wang, and W.-c. Feng. A Framework for the Automatic Vectorization of Parallel Sort on x86-based Processors. *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 2018.

[79] K. Hou, H. Wang, W.-c. Feng, J. Vetter, and S. Lee. Highly Efcient Compensation-based Parallelism for Wavefront Loops on GPUs. In *IEEE Int. Parallel and Distrib. Process. Symp. (IPDPS)*, 2018.

[80] K. Hou, Y. Zhao, J. Huang, and L. Zhang. Performance Evaluation of the Three-Dimensional Finite-Difference Time-Domain(FDTD) Method on Fermi Architecture GPUs. In *Int. Conf. Algorithms Archit. Parallel Process. (ICA3PP), Springer*, 2011.

[81] X. Huo, B. Ren, and G. Agrawal. A Programming System for Xeon Phis with Runtime SIMD Parallelization. In *ACM Int. Conf. Supercomput. (ICS)*, 2014.

[82] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *ACM Int. Conf. Parallel Arch. Compil. Tech. (PACT)*, 2007.

[83] H. Inoue and K. Taura. SIMD- and Cache-friendly Algorithm for Sorting an Array of Structures. *Proc. VLDB Endow. (VLDB)*, 2015.

[84] Intel Co. *Intel Xeon Phi Coprocessor System Software Developers Guide*, 2012. Doc. ID: 488596.

[85] P. Jiang and G. Agrawal. Combining SIMD and Many/Multi-core Parallelism for Finite State Machines with Enumerative Speculation. In *ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2017.

[86] W. Jung, J. Park, and J. Lee. Versatile and Scalable Parallel Histogram Construction. In *IEEE Int. Conf. Parallel Archit. Compil. Tech. (PACT)*, 2014.

[87] A. Khajeh-Saeed, S. Poole, and J. B. Perot. Acceleration of the Smith-Waterman Algorithm using Single and Multiple Graphics Processors. *J. Comput. Phys., Elsevier*, 2010.

[88] S. Lee, L. Chen, S. Duan, S. Chinthavali, M. Shankar, and B. A. Prakash. URBAN-NET: A Network-based Infrastructure Monitoring and Analysis System for Emergency Management and Public Safety. In *IEEE Int. Conf. on Big Data (BigData)*, 2016.

[89] N. Leischner, V. Osipov, and P. Sanders. GPU Sample Sort. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2010.

[90] A. Li, W. Liu, M. R. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song. Exploring and Analyzing the Real Impact of Modern On-package Memory on HPC Scientific Kernels. In *ACM/IEEE Int. Conf. High Perf. Comput., Netw., Storage and Anal. (SC)*, 2017.

[91] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal. Locality-Aware CTA Clustering for Modern GPUs. In *ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2017.

[92] B. Li, H.-C. Chang, S. Song, C.-Y. Su, T. Meyer, J. Mooring, and K. Cameron. Extending PowerPack for Profiling and Analysis of High-Performance Accelerator-Based Systems. *Parallel Process. Lett.*, 2014.

[93] B. Li, H. C. Chang, S. Song, C. Y. Su, T. Meyer, J. Mooring, and K. W. Cameron. The Power-Performance Tradeoffs of the Intel Xeon Phi on HPC Applications. In *IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, 2014.

[94] B. Li and E. A. León. Memory Throttling on BG/Q: A Case Study with Explicit Hydrodynamics. In *USENIX Workshop Power-Aware Comput. Syst. (HotPower)*, 2014.

[95] B. Li, E. A. León, and K. W. Cameron. COS: A Parallel Performance Model for Dynamic Variations in Processor Speed, Memory Speed, and Thread Concurrency. In *ACM Int. Symp. High-Perform. Parallel Distrib. Comput. (HPDC)*, 2017.

[96] B. Li, S. Song, I. Bezakova, and K. W. Cameron. Energy-Aware Replica Selection for Data-Intensive Services in Cloud. In *IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2012.

[97] B. Li, S. L. Song, I. Bezakova, and K. W. Cameron. EDR: An Energy-aware Runtime Load Distribution System for Data-intensive Applications in the Cloud. In *IEEE Int. Conf. Cluster Comput. (CLUSTER)*, 2013.

[98] F. Liu, H. Cai, G. Wang, D. D. Yao, K. O. Elish, and B. G. Ryder. MR-Droid: A Scalable and Prioritized Analysis of Inter-App Communication Risks. In *Mobile Secur. Technol. (MoST)*, 2017.

[99] F. Liu, X. Shu, D. Yao, and A. R. Butt. Privacy-preserving scanning of big content for sensitive data exposure with MapReduce. In *ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, 2015.

[100] F. Liu, C. Wang, A. Pico, D. Yao, and G. Wang. Measuring the Insecurity of Mobile Deep Links. In *USENIX Secur. Symp. (Security)*, 2017.

[101] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solvers. In *Parallel Process.: Int. Eur. Conf. Parallel Distrib. Comput. (EuroPar)*. Springer Berlin Heidelberg, 2016.

[102] W. Liu and B. Vinter. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. *J. Parallel Distrib. Comput. (JPDC)*, 2015.

[103] W. Liu and B. Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ACM Int. Conf. Supercomput. (ICS)*, 2015.

[104] W. Liu and B. Vinter. Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. *Parallel Comput. (ParCo)*, 2015.

[105] Y. Liu and B. Schmidt. SWAPHI: Smith-Waterman Protein Database Search on Xeon Phi Coprocessors. In *IEEE Int. Conf. Appl.-spec. Syst. Archit. Process. (ASAP)*, 2014.

[106] Y. Luo, G. Tan, Z. Mo, and N. Sun. FAST: A Fast Stencil Autotuning Framework Based On An Optimal-solution Space Model. In *ACM Int. Conf. Supercomput. (ICS)*, 2015.

[107] A. Magni, C. Dubach, and M. F. P. O'Boyle. A Large-scale Cross-architecture Evaluation of Thread-coarsening. In *ACM/IEEE Int. Conf. High Perf. Comput., Netw., Storage and Anal. (SC)*, 2013.

[108] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An Evaluation of Vectorizing Compilers. In *IEEE Int. Conf. Parallel Archit. Compil. Tech. (PACT)*, 2011.

[109] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 1993.

[110] N. Manjikian and T. S. Abdelrahman. Exploiting Wavefront Parallelism on Large-Scale Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 2001.

[111] N. Maruyama and T. Aoki. Optimizing Stencil Computations for NVIDIA Kepler GPUs. In *Int. Workshop High-Perform. Stencil Comput. (HiStencils)*, 2014.

[112] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers. In *ACM Int. Conf. High Perf. Comput., Netw., Storage and Anal. (SC)*, 2011.

[113] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. In *ACM Int. Conf. Supercomput. (ICS)*, 2011.

[114] D. Merrill and A. Grimshaw. High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing. *Parallel Process. Lett.*, 2011.

[115] NCBI. *Genbank*. `ftp://ftp.ncbi.nlm.nih.gov/genbank`.

[116] S. B. Needleman and C. D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *J. Mol. Bio., Elsevier*, 1970.

[117] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In *ACM/IEEE Int. Conf. High Perf. Comput., Netw., Storage and Anal. (SC)*, 2010.

[118] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007.

[119] M. Nourian, X. Wang, X. Yu, W.-c. Feng, and M. Becchi. Demystifying Automata Processing: GPUs, FPGAs or Micron's AP? In *ACM Int. Conf. Supercomput. (ICS)*, 2017.

[120] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture Kepler GK110*, 2014. v1.0.

[121] NVIDIA. cuSPARSE lib, 2016. `https://developer.nvidia.com/cuSPARSE`.

[122] NVIDIA Research. *CUB 1.6.4*, 2016. `http://nvlabs.github.io/cub/`.

[123] M. Pharr and W. Mark. ISPC: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Comput. (InPar)*, 2012.

[124] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Lib.* Prentice Hall PTR, 2000.

[125] F. Porikli. Integral Histogram: a Fast Way to Extract Histograms in Cartesian Spaces. In *IEEE Comput. Soc. Conf. on Comput. Vision Pattern Recognit. (CVPR)*, 2005.

[126] L.-N. Pouchet. Polybench: The Polyhedral Benchmark Suite, 2015. `http://web.cse.ohio-state.edu/pouchet/software/polybench`.

[127] R. Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[128] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Resource Conscious Reuse-Driven Tiling for GPUs. In *ACM Int. Conf. Parallel Archit. Compil. (PACT)*, 2016.

[129] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, and P. Sadayappan. Effective Resource Management for Enhancing Performance of 2D and 3D Stencils on GPUs. In *ACM Workshop Gen. Purpose Process. Graphics Process. Unit (GPGPU)*, 2016.

[130] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., 2007.

[131] G. Ren, P. Wu, and D. Padua. Optimizing Data Permutations for SIMD Devices. In *ACM SIGPLAN Conf. Program. Lang. Design Impl. (PLDI)*, 2006.

[132] P. Rice, I. Longden, A. Bleasby, et al. EMBOSS: the European Molecular Biology Open Software Suite.

[133] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2009.

[134] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2010.

[135] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications? In *IEEE Int. Symp. Comput. Archit. (ISCA)*, 2012.

[136] B. Schling. *The Boost C++ Libraries*. XML Press, 2011.

[137] X. Shu, F. Liu, and D. (Daphne) Yao. Rapid Screening of Big Data Against Inadvertent Leaks. In *Big Data Concepts, Theories, and Applications*. Springer, 2016.

[138] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief Announcement: The Problem Based Benchmark Suite. In *ACM Symp. Parallel Alg. Arch. (SPAA)*, 2012.

[139] E. Sintorn and U. Assarsson. Fast Parallel GPU-sorting Using a Hybrid Algorithm. *J. Parallel Distrib. Comput. (JPDC)*, 2008.

[140] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *J. Mol. Bio., Elsevier*, 1981.

[141] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and HPC*, 2012.

[142] A. Szalkowski, C. Ledergerber, P. Krhenbhl, and C. Dessimoz. SWPS3 fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and 86/SSE2. *BMC Res Notes*, 2008.

[143] K. Tian, Z. Li, K. Bowers, and D. D. Yao. FrameHanger: Evaluating and Classifying Iframe Injection at Large Scale. In *EAI Int. Conf. Secur. Privacy Commun. Networks (SecureComm)*, 2018.

[144] K. Tian, G. Tan, D. D. Yao, and B. G. Ryder. ReDroid: Prioritizing Data Flows and Sinks for App Security Transformation. In *ACM Workshop Forming an Ecosyst. Around Software Transform.*, 2017.

[145] K. Tian, D. Yao, B. G. Ryder, and G. Tan. Analysis of Code Heterogeneity for High-precision Classification of Repackaged Malware. In *IEEE Secur. Privacy Workshops (SPW)*, 2016.

[146] K. Tian, D. D. Yao, B. G. Ryder, G. Tan, and G. Peng. Detection of Repackaged Android Malware with Code-heterogeneity Features. *IEEE Trans. Dependable Secure Comput.*, 2017.

[147] D. Unat, X. Cai, and S. B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *ACM Int. Conf. Supercomput. (ICS)*, 2011.

[148] V. Vassilevska, R. Williams, and R. Yuster. Finding Heaviest H-subgraphs in Real Weighted Graphs, with Applications. *ACM Trans. Algorithms*, 2010.

[149] A. Vizitiu, L. Itu, C. Ni, and C. Suciu. Optimized Three-dimensional Stencil Computation on Fermi and Kepler GPUs. In *High Perform. Extreme Comput. Conf. (HPEC)*, 2014.

[150] H. Wang, W. Liu, K. Hou, and W.-c. Feng. Parallel Transposition of Sparse Data Structures. In *ACM Int. Conf. Supercomput. (ICS)*, 2016.

[151] J. Wang, P. Shang, and J. Yin. DRAW: A New Data-gRouping-AWare Data Placement Scheme for Data Intensive Applications with Interest Locality. In *Cloud Comput. Data-Intensive Appl.* Springer, 2014.

[152] J. Wang, X. Xie, and J. Cong. Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2017.

[153] J. Wang and S. Yalamanchili. Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications. In *IEEE Int. Symp. Workload Charact. (IISWC)*, 2014.

[154] L. Wang, S. Baxter, and J. D. Owens. Fast Parallel Suffix Array on the GPU. In *Parallel Process.: Int. Eur. Conf. Parallel Distrib. Comput. (EuroPar)*. Springer Berlin Heidelberg, 2015.

[155] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Comm. ACM*, 2009.

[156] M. Wolfe. Loops Skewing: the Wavefront Method Revisited. *Int. J. Parallel Program.*, 1986.

[157] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, 2014.

[158] H. Wu, D. Li, and M. Becchi. Compiler-Assisted Workload Consolidation for Efficient Dynamic Parallelism on GPU. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2016.

[159] X. Wu, C. Lively, V. Taylor, H. C. Chang, C. Y. Su, K. Cameron, S. Moore, D. Terpstra, and V. Weaver. MuMMI: Multiple Metrics Modeling Infrastructure. In *ACIS Int. Conf. Software Eng., Artif. Intell. Networking Parallel/Distrib. Comput.*, 2013.

[160] S. Xiao, A. M. Aji, and W. c. Feng. On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit. In *IEEE Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2009.

[161] T. Xiaochen, K. Rocki, and R. Suda. Register Level Sort Algorithm on Multi-core SIMD Processors. In *ACM Workshop Irregular App.: Arch. Alg. (IA3)*, 2013.

[162] K. Xu, K. Tian, D. Yao, and B. G. Ryder. A Sharper Sense of Self: Probabilistic Reasoning of Program Behaviors for Anomaly Detection with Context Sensitivity. In *IEEE/IFIP Int. Conf. Dependable Systems Networks (DSN)*, 2016.

[163] S. Yan, G. Long, and Y. Zhang. StreamScan: Fast Scan Algorithms for GPUs Without Global Barrier Synchronization. In *ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2013.

[164] S. Yang, H. Chung, X. Lin, S. Lee, L. Chen, A. Wood, A. L. Kavanaugh, S. D. Sheetz, D. J. Shoemaker, and E. A. Fox. PhaseVis1: What, When, Where, and Who in Visualizing the

Four Phases of Emergency Management through the Lens of Social Media. In *Int. ISCRAM Conf.*, 2013.

[165] Y. Yang and H. Zhou. CUDA-NP: Realizing Nested Thread-level Parallelism in GPGPU Applications. In *ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2014.

[166] X. Yu. *Deep Packet Inspection on Large Datasets: Algorithmic and Parallelization Techniques for Accelerating Regular Expression Matching on Many-core Processors*. University of Missouri-Columbia, 2013.

[167] X. Yu and M. Becchi. Exploring Different Automata Representations for Efficient Regular Expression Matching on GPUs. *ACM SIGPLAN Not.*, 2013.

[168] X. Yu and M. Becchi. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *ACM Conf. Comput. Front. (CF)*, 2013.

[169] X. Yu, W.-c. Feng, D. D. Yao, and M. Becchi. O3FA: A Scalable Finite Automata-based Pattern-Matching Engine for Out-of-Order Deep Packet Inspection. In *ACM Symp. Archit. Networking Commun. Syst. (ANCS)*, 2016.

[170] X. Yu, K. Hou, H. Wang, and W.-c. Feng. A Framework for Fast and Fair Evaluation of Automata Processing Hardware. In *IEEE Int. Symp. Workload Charact. (IISWC)*, 2017.

[171] X. Yu, K. Hou, H. Wang, and W.-c. Feng. Robotomata: A Framework for Approximate Pattern Matching of Big Data on an Automata Processor. In *IEEE Int. Conf. on Big Data (BigData)*, 2017.

[172] X. Yu, B. Lin, and M. Becchi. Revisiting State Blow-Up: Automatically Building Augmented-FA While Preserving Functional Equivalence. *IEEE J. Selected Areas Commun.*, 2014.

[173] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao. cuART: Fine-Grained Algebraic Reconstruction Technique for Computed Tomography Images on GPUs. In *IEEE/ACM Int. Symp. Cluster Cloud Grid Comput. (CCGrid)*, 2016.

[174] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao. An Enhanced Image Reconstruction Tool for Computed Tomography on GPUs. In *ACM Conf. Comput. Front. (CF)*, 2017.

[175] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow. (PVLDB)*, 2013.

[176] D. Zhang, H. Wang, K. Hou, J. Zhang, and W.-c. Feng. pDindel: Accelerating InDel Detection on a Multicore CPU Architecture with SIMD. In *IEEE Int. Conf. Comput. Adv. Bio Med. Sci. (ICCABS)*, 2015.

[177] F. Zhang. Automatic Loop Tuning and Memory Management for Stencil Computations, 2014. `http://scholarcommons.sc.edu/etd/3012` (Doctoral Dissertation).

[178] H. Zhang, D. Yao, and N. Ramakrishnan. Causality-based Sensemaking of Network Traffic for Android Application Security. In *ACM Workshop on Artif. Intell. Secur. (AISec)*, 2016.

[179] H. Zhang, D. Yao, N. Ramakrishnan, and Z. Zhang. Causality Reasoning about Network Events for Detecting Stealthy Malware Activities. *Computers & Security*, 2016.

[180] H. Zhang, D. D. Yao, and N. Ramakrishnan. Detection of Stealthy Malware Activities with Traffic Causality and Scalable Triggering Relation Discovery. In *ACM Symp. on Inf. Comput. Commun. Secur.*, 2014.

[181] J. Zhang, H. Lin, P. Balaji, and W.-c. Feng. Consolidating Applications for Energy Efficiency in Heterogeneous Computing Systems. In *IEEE Int. Conf. High Perform. Comput. and Commun. (HPCC)*, 2013.

[182] J. Zhang, H. Lin, P. Balaji, and W.-c. Feng. Optimizing Burrows-Wheeler Transform-Based Sequence Alignment on Multicore Architectures. In *IEEE/ACM Int. Symp. Cluster Cloud Grid Comput. (CCGrid)*, 2013.

[183] J. Zhang, S. Misra, H. Wang, and W.-c. Feng. muBLASTP: Database-Indexed Protein Sequence Search on Multicore CPUs. *BMC Bioinf.*, 2016.

[184] J. Zhang, S. Misra, H. Wang, and W.-c. Feng. Eliminating Irregularities of Protein Sequence Search on Multicore Architectures. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2017.

[185] J. Zhang, H. Wang, and W.-c. Feng. cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on CPU+GPU. *IEEE/ACM Trans. Comput. Bio. Bioinf. (TCBB)*, 2016.

[186] J. Zhang, H. Wang, H. Lin, and W.-c. Feng. cuBLASTP: Fine-Grained Parallelization of Protein Sequence Search on a GPU. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2014.

[187] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-KV: A Case for GPUs to Maximize the Throughput of In-memory Key-value Stores. *Proc. VLDB Endow. (PVLDB)*, 2015.

[188] Y. Zhang and F. Mueller. Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters. *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 2013.

[189] Y. Zhang and B. A. Prakash. DAVA: Distributing Vaccines over Networks under Prior Information. In *SIAM Int. Conf. Data Mining*, 2014.

[190] Y. Zhang and B. A. Prakash. Scalable Vaccine Distribution in Large Graphs given Uncertain Data. In *ACM Int. Conf. Inf. Knowledge Manage.*, 2014.

[191] N. Zhao, A. Anwar, Y. Cheng, M. Salman, D. Li, J. Wan, C. Xie, X. He, F. Wang, and A. R. Butt. Chameleon: An Adaptive Wear Balancer for Flash Clusters. In *IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2018.

[192] M. Zuluaga, P. Milder, and M. Püschel. Computer Generation of Streaming Sorting Networks. In *ACM Des. Autom. Conf. (DAC)*, 2012.