

Accelerating the “Motifs” in Machine Learning on Modern Processors

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Jiyuan Zhang

B.S., Electrical Engineering, Harbin Institute of Technology

Carnegie Mellon University

Pittsburgh, PA

December 2020

©Jiyuan Zhang, 2020

All Rights Reserved

Acknowledgments

First and foremost, I would like to thank my advisor — Franz. Thank you for being a supportive and understanding advisor. You encouraged me to pursue projects I enjoyed and guided me to be the researcher that I am today. When I was stuck on a difficult problem, you are always there to offer help and stand side by side with me to figure out solutions. No matter what the difficulties are, from research to personal life issues, you can always consider in our shoes and provide us your unreserved support. Your encouragement and sense of humor has been a great help to guide me through the difficult times. You are the key enabler for me to accomplish the PhD journey, as well as making my PhD journey less suffering and more joyful.

Next, I would like to thank the members of my thesis committee – Dr. Michael Garland, Dr. Phil Gibbons, and Dr. Tze Meng Low. Thank you for taking the time to be my thesis committee. Your questions and suggestions have greatly helped shape this work, and your insights have helped me understand the limitations that I would have overlooked in the original draft. I am grateful for the advice and suggestions I have received from you to improve this dissertation. This thesis would not have been possible without the guidance and patience of you.

In addition, I am extremely grateful for the endless support from the

Spiral group and colleagues at A-Level. Thank you professor James Hoe. You are such a wonderful professor to have at A-level. Your encouragement for us to think critically and to express our thoughts no matter how immature they are, has been a big encouragement for me. I am also extremely thankful for professor Tze Meng Low. It was such a luck to have you in the group when I just started graduate school. You are a great help in my research journey. Those discussions with you were very inspirational for a early-year PhD. They definitely shaped my understanding of research and taught me how to be a responsible researcher. I would also like to thank Dr. Scott McMillan for those helpful discussions and advice on my research and guiding me into the world of graph-related motifs. I cannot adequately express how thankful I am. Additionally, thank you my wonderful colleagues at A-level — Richard, Thom, Daniele, Fazle, Marie, Joe, Guanglin, Zhipeng, Ke, Senbo, Anuva. My experience in the graduate school was greatly enhanced as I was surrounded by so many wonderful friends and colleagues. I would never forget those research-related and non-research-related discussions with you guys. It has been so much fun and thought-provoking. Also, thank you my friends — Yi Lu, Zhuo Chen, Yang Gao, Qichen Huang, Chenglei Fang, Yang li, for being great friends, the source of great emotional support.

I would also like to thank my funding resources. The research in this thesis is based upon work funded and supported by DARPA under agreement HR0011-20-9-0018, DE-AC02-05CH11231, HR00111320007 and FA87501220291. This thesis is also based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Last but not least, I owe an especially deep gratitude to my parents for their never-ending love, patience, and encouragement.

JIYUAN ZHANG

Carnegie Mellon University

December 2020

Abstract

The booming growth in AI and machine learning is drastically reshaping the landscape of high performance computing. Traditional HPC addresses scientific problems that are driven by simulation, modeling and analysis in science domains. Algorithms like linear algebra and methods like differential equations are at the core of solutions to such problems. However, emerging machine learning tasks rest on a new set of algorithms and models. The computation and data movement patterns inherent in learning algorithms cannot be directly mapped to the computation motifs in the physics, chemistry, or biology simulation problems. As a result, the high performance libraries that originate in the traditional scientific domains thus cannot be straightforwardly applied to emerging ML tasks to deliver required performance.

This thesis focuses on performance optimizations of computation kernels in emerging machine learning applications, spanning across a diverse range from dense, regular to sparse and irregular kernels. In this work, we demonstrate how code specialization and generation together with expert-built performance models and learned dispatch strategies can together enable ML motifs to achieve better performance on modern processors.

First, we investigate the performance optimization of dense kernels with a focus on the convolutional neural networks (CNN). The computation of convolution layers in deep neural networks typically relies on high performance matrix-multiplication routines to improve performance. However, these routines are not optimized for performing convolution. Extra memory overhead is incurred due to data transformation, and the performance obtained is also less than conventionally expected. We demonstrate that direct convolution, when implemented *correctly*, eliminates all memory overhead, and yields performance that is between 10% to 4x better than existing high performance im-

plementations on conventional and embedded CPU architectures. We present a model-guided optimization approach which utilizes the characteristics of system architectures to guide the optimization choices of loop ordering, blocking, and memory layout transformation. We show that a high performance direct convolution exhibits better performance scaling than expert-tuned matrix implementation, i.e. suffers less performance drop, when increasing the number of threads.

Sparse kernel is an equally important computation kernel appearing in many machine learning applications such as graph analytics and genetic sequencing. One factor that prevents sparse kernels from achieving high performance on modern processors results from the prohibitively large number of different implementations and data structures for sparse problems. We start with the observation that the complicated sparse computations can be distilled into primitive set of operators such as join, merge, and difference. To accelerate those operators on modern processors with data parallelism, we propose a vectorization and code specialization approach which can eliminate the control divergences of these operators. Next, we explore the design space for vectorization on CPUs with various vector width, based on which we present the code generation algorithm that takes the data width and operations as input and generates various implementations. We then demonstrate the acceleration of the General Sparse Matrix-Matrix Multiplication (SpGEMM) on GPUs. We show how the SpGEMM implementation can leverage join/merge operators to compose a variety of implementations. Another challenge when optimizing sparse kernels is that their performance behavior is data dependent, while the input characteristics may change online during iterative updates. To leverage the different implementations offered by the code generator, we propose a low-

overhead mechanism that collects the data characteristic information to learn online dispatch decisions over iterations.

Overall, in this thesis, we demonstrate the interplay of code specialization and generation, together with performance modeling, learned dispatch, can enable high performance kernels for the emerging machine learning applications.

Contents

| | |
|--|------------|
| Acknowledgments | iii |
| Abstract | vi |
| List of Figures | xv |
| List of Tables | xxi |
| Chapter 1 Introduction | 1 |
| 1.1 Motivation | 1 |
| 1.2 Preliminaries | 4 |
| 1.2.1 Domains and Computation Kernels | 4 |
| 1.2.2 Frameworks and Libraries | 5 |
| 1.2.3 Code Generation | 8 |
| 1.2.4 Performance Optimization | 8 |
| 1.2.5 Hardware Architectures | 9 |
| 1.3 Dense Computational Kernels in Neural Networks | 10 |
| 1.4 Sparse Computational Kernels in Data Analytics | 12 |

| | | |
|------------------|--|-----------|
| 1.5 | Thesis Overview | 14 |
| Chapter 2 | Accelerating Dense Convolutions | 17 |
| 2.1 | Overview | 18 |
| 2.2 | Introduction | 20 |
| 2.3 | Background | 23 |
| 2.4 | High Performance Direct Convolution | 27 |
| 2.4.1 | Strategy for Mapping Loops to Architecture | 27 |
| 2.4.2 | Parallelism | 33 |
| 2.5 | Convolution-Friendly Data Layout | 34 |
| 2.5.1 | Input/Output Layout | 34 |
| 2.5.2 | Kernel Layout | 35 |
| 2.5.3 | Backward Compatibility | 36 |
| 2.6 | Experiments | 38 |
| 2.6.1 | Experimental Setup | 38 |
| 2.6.2 | Performance | 39 |
| 2.6.3 | Parallel Performance | 40 |
| 2.7 | Chapter Summary | 43 |
| Chapter 3 | Sparse Computational Kernels: A Preliminary Study | 45 |
| 3.1 | Expressing Computation Kernels in Sparse Applications | 47 |
| 3.2 | Accelerating Sparse Computation: A Case Study with Triangle Counting | 49 |

| | | |
|-------|--|----|
| 3.2.1 | Introduction | 49 |
| 3.2.2 | Algorithm | 51 |
| 3.2.3 | Parallel Triangle Counting | 52 |
| 3.2.4 | Preliminary Exploration on Optimizations | 53 |
| 3.2.5 | Profiling and Analysis | 56 |
| 3.2.6 | Understanding the Optimizations | 64 |
| 3.2.7 | Performance Results | 65 |

| | | |
|------------------|---|-----------|
| Chapter 4 | Accelerating Sparse Computational Kernels on CPUs: A Fast SIMD Set Intersection Approach | 69 |
| 4.1 | Abstract | 70 |
| 4.2 | Introduction | 71 |
| 4.3 | Background and Related Work | 74 |
| 4.3.1 | Scalar Set Intersection Approaches | 76 |
| 4.3.2 | Accelerating Set Intersections | 78 |
| 4.4 | The FESIA Approach | 79 |
| 4.4.1 | Overview | 80 |
| 4.4.2 | Data Structure | 80 |
| 4.4.3 | Intersection Algorithm | 82 |
| 4.4.4 | Theoretical Analysis | 83 |
| 4.5 | Bitmap-Level Intersection | 85 |
| 4.6 | Segment-Level Intersection | 87 |
| 4.6.1 | Runtime Dispatch | 89 |
| 4.6.2 | Specialized vs. General Set Intersection Kernels | 91 |

| | | |
|-------|---|-----|
| 4.6.3 | Implementing Specialized Intersection Kernels with SIMD | 92 |
| 4.7 | Discussion | 96 |
| 4.8 | Experiments | 99 |
| 4.8.1 | Experimental Setup | 100 |
| 4.8.2 | Result of Specialized Intersection Kernels | 101 |
| 4.8.3 | Effect of Varying the Input Size | 103 |
| 4.8.4 | Effect of Varying the Selectivity | 105 |
| 4.8.5 | Performance of Two Sets with Different Sizes | 107 |
| 4.8.6 | Performance on Real-World Datasets | 110 |
| 4.9 | Conclusion | 111 |

Chapter 5 Accelerating Sparse Computational Kernels on GPUs 113

| | | |
|-------|----------------------------------|-----|
| 5.1 | Abstract | 114 |
| 5.2 | Introduction | 115 |
| 5.3 | Background and Related Work | 118 |
| 5.3.1 | GPUs. | 118 |
| 5.3.2 | SpGEMM and Masked-SpGEMM Problem | 121 |
| 5.3.3 | Sparse Data Structure | 122 |
| 5.3.4 | SpGEMM Computations | 123 |
| 5.4 | The Family of SpGEMM Algorithms | 124 |
| 5.4.1 | Matrix-Vector SpGEMM | 125 |
| 5.4.2 | Inner-Product-Based SpGEMM | 127 |
| 5.4.3 | Outer-Product-Based SpGEMM | 129 |

| | | |
|------------------|--|------------|
| 5.5 | Join-Based SpGEMM | 130 |
| 5.5.1 | The Baseline Join-Based SpGEMM | 130 |
| 5.5.2 | The Data-Parallel Join-Based SpGEMM | 131 |
| 5.5.3 | Join Using Wider Data Types | 133 |
| 5.6 | Join-Based SpGEMM Using Hash Methods | 137 |
| 5.7 | Online Scheduling | 139 |
| 5.8 | Experiments | 142 |
| 5.8.1 | Experimental Setups | 142 |
| 5.8.2 | Performance of GPU Joins with Very Sparse Matrices . | 145 |
| 5.8.3 | Performance of Join-Based SpGEMM | 146 |
| 5.8.4 | Comparison on Different Densities | 147 |
| 5.8.5 | Performance on the Hash-Based Join Kernels | 149 |
| 5.8.6 | Performance on Real-World Dataset | 153 |
| 5.8.7 | Performance on Real-World Graph Datasets with Skewed Distribution | 156 |
| Chapter 6 | Summary and Future Directions | 161 |
| 6.1 | Summary | 162 |
| 6.2 | Takeaways and Reflections | 164 |
| 6.3 | Future Directions | 167 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | The structure of a convolutional layer. | 11 |
| 1.2 | Sparse data is underlying many data analytics tasks such as social network analytics, recommendation system, sparse machine learning models. | 12 |
| 2.1 | High performance direct convolution implementation achieves higher performance than a high performance matrix multiplication routine, whereas matrix-multiplication based convolution implementations suffers from packing overheads and is limited by the performance of the matrix multiplication routine | 19 |

| | | |
|-----|--|----|
| 2.2 | The 5×5 input image with 3 different channels (denoted with different colors) is convolved with two separate kernels to obtain a 3×3 output with two output channels. Packing is performed to turn three dimensional input images (left) into a two dimensional matrix (right) in order to utilize a high performance matrix multiplication routine. As C_o and/or $(H_o \times W_o)$ are often less than $H_f \times W_f \times C_i$, performance of standard matrix-matrix multiplication in many BLAS libraries are often sub-optimal. | 25 |
| 2.3 | Convolution-friendly layout for input/output (left) and kernel weights (right). The output data is organized into sequential blocks of $H_o \times W_o \times C_{o,b}$, where in each block, the fastest dimension is in the channel dimension, followed by the column and row dimension of the output. The kernel weights are organized into blocks of $H_o \times W_o \times C_{o,b} \times C_{i,b}$. The fastest dimension is the blocked output channel, followed by the blocked input channels, kernel width and height, input channels and then the output channels. | 36 |

| | | |
|-----|---|----|
| 2.4 | Performance of direct convolution against existing high performance FFT-based and SGEMM -based convolution implementations. Performances of all implementations are normalized to the performance of SGEMM + <code>im2col</code> routine. Direct convolution is highly competitive against all other implementations achieving between 10% and 400% improvement in performance even against a BLAS library (Intel MKL) that optimizes for matrix shapes arising from convolution layers. | 41 |
| 2.5 | Scaling behavior with increasing number of threads. Our direct convolution implementation retains high GFLOPs per core performance as we increase the number of threads from 1 to the number of available cores. This is indicative of an efficient parallelized algorithm. When the number of threads exceeds the number of cores, excessive contention results in a significant drop in performance per core. In contrast, SGEMM has poor scalability even when the number of threads is low (e.g. 2). . . | 42 |
| 3.1 | SIMD algorithm assuming simd width is 4 elements. | 55 |
| 3.2 | SIMD algorithm illusion. | 56 |
| 3.3 | Degree distribution of cit-Patents. | 58 |
| 3.4 | Degree distribution of friendster. | 59 |
| 3.5 | Degree distribution of graph500-scale23. | 60 |
| 3.6 | Theoretical computation distribution and non-zero element distribution on graphs with different skewness. | 62 |

| | | |
|-----|---|-----|
| 3.7 | Number of Non-zero element distribution and computation time distribution across rows. | 63 |
| 4.1 | Illustrating the data structure and the set intersection algorithm in FESIA. There are two steps in the set intersection: (1) the bitmaps are used to filter out unmatched elements, and (2) a segment-by-segment comparison is conducted to compute the final set intersection using specialized SIMD kernels. | 75 |
| 4.2 | Illustrating the difference between a general and a specialized SIMD set intersection kernels. A general kernel is shown on the upper part of this figure, which is implemented with SSE-128 instructions and can be used for any intersection with input size less than 4-by-4. A specialized 2-by-4 kernel is shown on the bottom, which reduces unnecessary computation and memory accesses (highlighted in purple). | 88 |
| 4.3 | Illustrating the specialized kernels: (1) a 2-by-7 intersection kernel (small-by-large), (2) a 4-by-5 intersection kernel (small- by-large and S_b is slightly larger than V), and (3) a 6-by-6 intersection kernel (large-by-large). | 93 |
| 4.4 | Performance of SSE kernels | 102 |
| 4.5 | Performance of AVX kernels | 102 |
| 4.6 | Performance comparison with a varying input size | 104 |
| 4.7 | Performance on different selectivity | 106 |
| 4.8 | Performance on different selectivity (AVX-512) | 106 |

| | | |
|------|--|-----|
| 4.9 | Performance of three-way intersection | 106 |
| 4.10 | Performance comparison on varying skew | 108 |
| 4.11 | Results on the database query task | 109 |
| 4.12 | Results on the triangle counting task | 111 |
| 5.1 | The family of SpGEMM algorithms | 119 |
| 5.2 | Complexity comparison on different sparse matrices. N denotes the maximum dimension of the sparse matrix (i.e., maximum nonzero value). k denotes the average nonzeros per row/column — for tall-skinny matrix, k is the average nonzeros per row. If the matrix is short and fat, k is average nonzeros per column. | 120 |
| 5.3 | All-pair comparison based join implementation: a 32-by-32 im- plementation. | 133 |
| 5.4 | All-pair comparison based join implementation: a 8-by-8 im- plementation. | 134 |
| 5.5 | Hash-based SpGEMM implementation: parallel across columns. | 138 |
| 5.6 | Hash-based SpGEMM implementation: parallel across rows. . | 139 |
| 5.7 | Online scheduling strategy. | 141 |
| 5.8 | Comparison of join implementations on very sparse matrices. . | 146 |
| 5.9 | Comparison of join implementations on relatively dense syn- thetic data in which the densities are 0.3. | 148 |
| 5.10 | Comparison of join implementations on relatively sparse syn- thetic data in which the densities are 0.01. | 148 |
| 5.11 | Comparison of join implementations on varying sparsities. . . | 149 |

| | | |
|------|---|-----|
| 5.12 | Comparison of hash-based join implementations on datasets of density 0.05. The hash table size is 1KB per row. $\text{tblsz} = 1\text{K}/2\text{K}/4\text{K}$ refers to building one hashtable for 1/2/3 rows respectively. | 151 |
| 5.13 | Comparison of hash-based join implementations on datasets of density 0.05. The hash table size is 2KB per row. We build a hashtable for 1/2/3 rows simultaneously. $\text{tblsz} = 1\text{K}/2\text{K}/4\text{K}$ refers to building one hashtable for 1/2/3 rows respectively. . . | 152 |
| 5.14 | Comparison of hash-based join implementations on relatively sparse synthetic data in which the densities are 0.3. | 153 |
| 5.15 | Comparison of hash-based join implementations on relatively sparse synthetic data in which the densities are 0.01. | 154 |
| 5.16 | Comparison of different thread assignment strategies on real-world graph datasets. | 160 |
| 6.1 | Overview of this dissertation | 162 |

List of Tables

| | | |
|-----|---|-----|
| 2.1 | Details of specific architecture used | 38 |
| 3.1 | Compression savings on Friendster dataset. Its original edge list size is 1,806,067,135. | 65 |
| 3.2 | Single-core performance scalar vs. SIMD | 67 |
| 3.3 | Small 28-core system performance | 67 |
| 3.4 | Performance on a HP Superdome X system with 16 sockets, 352 cores | 68 |
| 4.1 | The summary of our approach vs. state-of-the-art set intersec- tion approaches. | 72 |
| 4.2 | Hardware specifications | 100 |
| 4.3 | The details of each graph dataset | 110 |
| 5.1 | Performance comparison on iterative SpGEMM computations with real-world datasets. | 156 |
| 5.2 | Graph dataset characteristics. | 159 |

Chapter 1

Introduction

1.1 Motivation

Performance has always been of utmost importance in many computing tasks, which has been the driving force behind the research of high performance computing over the decades. Traditional HPC mainly focuses on the scientific computing tasks which usually arise from the simulation or modeling processes for biology, physics or chemistry problems. For example, typical scientific computing tasks include weather prediction, quantum mechanics simulation, and cosmopolitan analysis.

Solutions to these scientific tasks are usually based on mathematic methods in linear algebra or differential equations. Thus the computational kernels of these methods are centered around dense linear algebra or structured mesh. Over the years tremendous efforts are spent on researching and designing high performance implementations for those computing kernels in

scientific applications. This leads to a handful of libraries and frameworks. For example, BLAS/LAPACK libraries were developed for the computing kernels in linear algebra, FFTX library was proposed targeting spectral methods, and OSKI was proposed for the sparse linear algebra kernels in differential equations.

In recent years, machine learning has made significant breakthroughs which brought transformations across many aspects of research and industry. In many real-world problems, the conventional methodologies and algorithms are now replaced by ML-based models. For example, methods such as deep learning and graph analytics are now at the core of tasks such as image classification, voice recognition or object tracking. Despite their wide adoption, however, it is still ongoing research on how to make the emerging ML models run faster and efficiently on various platforms. The computation kernels of ML models are completely different from the conventional scientific kernels, Thus high performance libraries developed for scientific computing can not be directly applied to the machine learning field.

Achieve high performance for the emerging machine learning tasks is challenging due to two reasons. Firstly, computation kernels in machine learning tasks possess a broad spectrum of distinctions in the characteristics of computation patterns and data movements. At one end of the spectrum, there is dense applications, in which the computation kernels of such tasks have regular and structured patterns. The representative dense computation tasks include convolutional-neural-networks. At the other end of the spectrum is the sparse

applications, and such applications usually work with irregular and unstructured data. The representative sparse applications involve some analytics tasks in social networks or recommendation systems. Dense and sparse computation kernels present different computational characteristics, which result in different sets of challenges for software optimizations. The second performance challenge results from the complexities of the underlying hardware. In particular, with the growing computational demand for machine learning tasks, the underlying computer systems are evolving rapidly. Architecture features such as cache size, number of registers, memory bandwidth vary across platforms, which makes obtaining high performance consistently across platforms a non-trivial task.

With the growing prevalence of machine learning techniques and their widespread deployment onto a broad spectrum of systems, there is a pressing need for high performance machine learning libraries or frameworks on existing and future platforms.

The goal of this thesis is to provide understandings on the optimizations of the computation kernels of the emerging ML tasks. The computation kernels lay the backbone of understanding the interplay between software optimizations and hardware architecture, which is a fundamental step towards building a portable and efficient end-to-end ML system. In particular, in this work, we demonstrate how an analytical model, together with code specialization, and learned dispatch strategies can enable the high performance computational kernels of machine learning tasks. In the first part of this work, we

demonstrate the optimization of the dense kernels in machine learning using convolution neural networks as an example. We adopt a model-based analysis to provide a systematic understanding of how the software optimizations such as loop ordering, blocking, instruction scheduling, should be applied based on the hardware architectures. In the second part of this work, we investigate the optimizations of sparse kernels. Sparse kernels present a different set of challenges compared to dense computations. We present the technique that can improve the data-parallelism of the irregular computations of sparse kernels, as well as investigating the effects of algorithm and data structure choices on the performances.

1.2 Preliminaries

In this section, we provide a brief overview of the computational domains and the class of operations in these domains that we target.

1.2.1 Domains and Computation Kernels

Researchers from the University of California, Berkeley performed an investigation across different domains and distilled the computational kernels to characterize and represent the domain-specific computations [1]. We explain some kernels that are relevant to this thesis in the following.

Dense linear algebra These computations typically arise in the applications of scientific simulations. Dense linear algebra computation involves data

as dense matrices or vectors. The computation pattern is regular with mostly strided memory access.

Structured Mesh (Stencils). These computations typically arise in the computation of PDEs using finite-difference methods. The computation of structure mesh usually involves data represented by a regular grid of n -dimensional mesh. The connection in the grid represents the relationship between this data point and its neighbors.

Sparse linear algebra. Sparse linear algebra can appear both in scientific computing as well as machine learning applications. Many real-world data contain a significant number of zero values. Therefore such data is usually stored in compressed matrices to reduce the storage and bandwidth requirements to access the elements. The data format examples include Compressed-Sparse-Row (CSR), Compressed-Sparse-Column (CSC), Coordinate (COO) format, etc. Because of the compressed formats, data is generally accessed with indexed loads and stores.

1.2.2 Frameworks and Libraries

One approach to shifting the burden from the user is to implement libraries that contain commonly used functionality. This approach has been very successfully applied to dense computations and libraries like BLAS and LAPACK are frequently used in scientific codes.

Dense Linear Algebra

Dense linear algebra in scientific computing Basic Linear Algebra Subprograms (BLAS) specifies a set of standards for the common linear algebra operations. The linear algebra operations are categorized into Level-1, Level-2, and Level-3 routines in BLAS specification: Level-1 refers to vector operations such as vector addition, dot product. Level-2 refers to the general matrix-vector multiplication operation. Level-3 refers to the generalized matrix multiplication. A number of libraries have been proposed conforming to the BLAS specifications and are tuned for specific hardware. For example, AMD Core Math Library (ACML) has a high-performance BLAS routine implementation for AMD processors. Intel Math Kernel Library (MKL) is the BLAS implementation for Intel, and OpenBLAS provides high-performance BLAS implementations for a wide range of x86 architectures.

Dense linear algebra in machine learning Dense linear algebra kernels also appear in machine learning applications. Some computations in deep learning models can correspond to the BLAS operations. As a matter of fact, many machine learning frameworks heavily depend on high-performance BLAS routines. For example, the TensorFlow framework transforms the high-level machine learning computations into primitive matrix operators such as matrix-matrix-multiplication (convolution neural network), matrix-vector-multiplication (recurrent neural network), etc.

Sparse Linear Algebra

Sparse linear algebra in scientific computing Sparse linear algebra kernels appear in both traditional scientific computing and machine learning applications. For example, the finite discretization in fluid flow simulations can be formulated using a sparse grid. Libraries have been proposed to accelerate sparse linear algebra kernels in scientific problems. For example, the Optimized Sparse Kernel Interface (OSKI) [48] exposed an autotuning API along with providing memory hierarchy aware kernels to match modern architectures, which allows library users to pass domain knowledge to these building blocks and reap the performance benefit while minimizing the amount of expensive autotuning.

Sparse linear algebra in machine learning Recent years, a growing body of research is focusing on sparse problems arising from the social sciences. As a result, we are seeing the development of sparse libraries and frameworks designed to accelerate graph analytics — the tasks that are underneath many social analytics problems. There are a rich amount of graph frameworks that can target a general set of graph applications [2, 3, 4, 5, 6, 7]. Those frameworks have different frontend designs. For example, [2] is based on vertex-iterator programming model where a utility function is supplied and executed on each vertex, [3] uses linear algebra language to abstract away the graph algorithms, meanwhile, [4] uses a set of DSL to describe and optimize graph operations, etc.

1.2.3 Code Generation

Code generation can overcome the drawbacks of general compilers. The general compiler is capable of performing some optimizations such as auto-vectorization, peep-hole optimizations. However, their optimization capabilities are very limited. Firstly, the optimization effectiveness of the general compilers is highly dependent on accurate modeling of the underlying hardware, whereas it is usually hard to establish an exact model of the hardware architectures. More importantly, general compilers are unable to capture the high-level algorithmic optimizations. For example, the compiler is capable to perform auto-vectorization, but only when the code is inherently expressed as regular and iterative loops. Code generation offers an alternative approach that is capable to take into account the high-level algorithmic transformations while generating different implementations as specified by the programmers. The code generation approach can therefore ease the burden for programmers to explore different implementations, especially when facing the fast-changing and increasingly-complex hardware architectures.

1.2.4 Performance Optimization

Implementing software that performs satisfactorily across platforms and hardware has become an ever-growing challenging task with the evolving complexity of today's computer systems. In conventional high performance computing, there are two approaches to achieve high performance: analytic modeling and automatic-tuning. Analytic modeling requires accurate modeling of the un-

derlying computer architectures and analytically derive the best optimization strategies such as loop ordering, blocking strategies, parallelism, etc. Analytic modeling has been demonstrated to successfully optimize the implementation of BLAS libraries [8, 9]. Auto-tuning is another way to achieve high performance, especially when modeling the machine’s behavior accurately enough is impossible on today’s computers, auto-tuning can get the actual run time as feedback. By means of automatic empirical performance tuning, and apply search techniques to find the best implementation for a given target machine. Auto-tuning is applied in libraries such as Spiral [10], FFTW [11], and ATLAS [12].

1.2.5 Hardware Architectures

Vector instructions. A few years ago major vendors of general-purpose microprocessors have started to include short vector SIMD (single instruction, multiple data) extensions into their instruction set architecture (ISA) primarily to improve the performance of multimedia applications. Examples of SIMD extensions supporting both integer and floating-point operations include vectorized loads/stores, logic arithmetics, bit permutations, etc. SIMD instructions have now become the status quo on modern CPUs to exploit data parallelism. For example, Intel CPUs have SSE/AVX2 instructions to support 128-bit and 256-bit vector operations. Similarly, ARM processors have NEON instructions, and IBM Power processors have AltiVec instructions. The prevailing SIMD widths on modern processors are 128-bit and 256-bit. More

recently, the Intel Skylake architecture introduced AVX512 instructions.

Hierarchical memory system. The modern processor is typically built with hierarchical memory structures. The closest to the processor is the register memory. The register file holds the temporary elements that are used for arithmetic tasks. The next level is the L1-cache (built into the processor) followed by the L2-cache. L1 caches are usually fast but small. They directly access the L2 cache which is usually larger but slower. L1-cache and L2-cache are usually private to each core. In the next level is the L3-cache which is usually shared among all the cores on the chip. The L3-cache accesses main memory which—on architectures with virtual memory—exchanges data with disk storage.

1.3 Dense Computational Kernels in Neural Networks

Convolution neural networks have received significant attention over recent years, as they are becoming one of the most popular models for many deep learning tasks. They are widely deployed for deep learning tasks such as image classification and segmentation, object detection, video processing, etc.

With CNN’s wide adoption in varying deep learning tasks, as well as the ever-growing applications and scenarios, it also introduces new challenges and problems for computation and system designs. For example, there is a growing

need of running machine learning tasks on devices like smartphones, cars, drones, etc. On such devices, the compute capability and memory capacity available are often limited. This naturally limits the size of the deep neural nets that can be placed on the system.

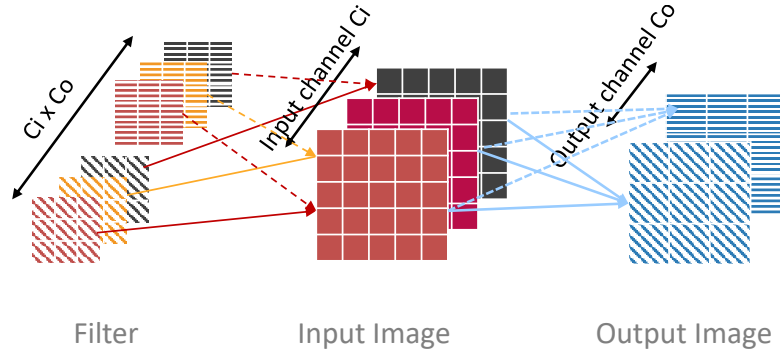


Figure 1.1: The structure of a convolutional layer.

The computation of convolution layers in deep neural networks typically relies on high performance routines that trade space for time by using additional memory (either for packing purposes or required as part of the algorithm) to improve performance. The problems with such an approach are two-fold. First, these routines incur additional memory overhead which reduces the overall size of the network that can fit on embedded devices with limited memory capacity. Second, these high performance routines were not optimized for performing convolution, which means that the performance obtained is usually less than conventionally expected. In this chapter, we demonstrate that direct convolution, when implemented *correctly*, eliminates all memory overhead, and yields performance that is between 10% to 400% times better than existing high performance implementations of convolution layers on conventional and

embedded CPU architectures. We also show that a high performance direct convolution exhibits better scaling performance, i.e. suffers less performance drop, when increasing the number of threads.

1.4 Sparse Computational Kernels in Data Analytics

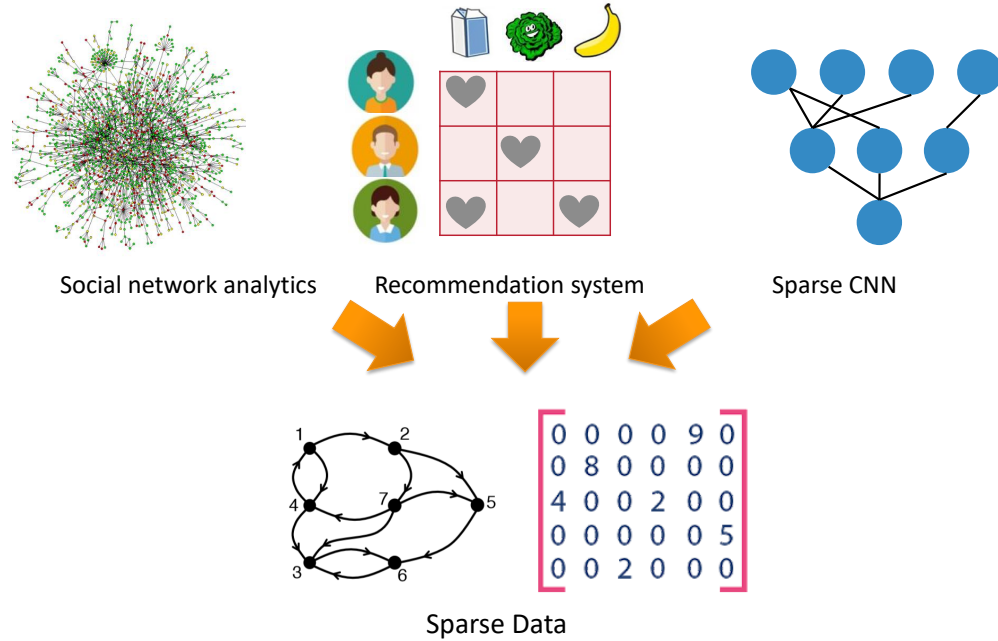


Figure 1.2: Sparse data is underlying many data analytics tasks such as social network analytics, recommendation system, sparse machine learning models.

Sparse computations are at the core of machine learning and data science applications. They have been employed in a broad range of tasks, such as graph analytics [13], neural networks compressing [14], genome sequenc-

ing [15], recommendation systems [16]. The Sparse matrices in these problems are usually represented and recorded using the indices of the nonzero. And the percentage of non-zero elements in these applications can vary drastically, ranging from $10^{-6}\%$ to 50% depending on the problem domain.

Sparse kernels are hard to attain high performance on modern processors. Unlike their counterpart in dense and regular applications, sparse applications have locally varying non-zero (NZ) patterns, leading to unpredictable control flows and unbalanced workloads which are detrimental to performance. In addition, sparse applications have irregular compute patterns that are inherently sequential, together with random and uncoalesced memory accesses. These issues can result in low occupancy as well as low resource utilization on modern processors.

On the other hand, there is a large number of different implementations and data structures for sparse problems. However, this is also a lack of understanding of the optimization techniques for sparse problems. It still remains an open question as to which algorithm or data structure best fits sparse data. One reason for this is due to the vast problem domains and the data characteristics (density, distribution) can vary drastically in different domains. Additionally, there are a large number of different methods to process and deal with sparse data, including different algorithms, parallelism schemes, etc.

In this dissertation, we start with the observation that the complicated sparse computations can be distilled into a primitive set of operators such as

join, merge, and difference. To accelerate those operators on modern processors with data parallelism, we first propose a vectorization and code specialization approach that can eliminate the control divergences of these operators. Next, we explore the design space for vectorization on CPUs with various vector width, based on which we present the code generation algorithm that takes the data width and operations as input and generates various implementations. We then demonstrate the acceleration of the General Sparse Matrix-Matrix Multiplication (SpGEMM) on GPUs.

1.5 Thesis Overview

The goal of the thesis is to provide understandings and optimizations on the computation kernels of the emerging ML tasks including neural networks, social networks, and data analytics applications. The computation kernels lay the backbone of understanding the interplay between software optimizations and hardware architecture, which is a fundamental step towards building a portable and efficient end-to-end ML system.

In particular, this work is focusing on two sets of computational kernels: the dense computational kernels in convolutional neural networks, and the sparse kernels in data analysis tasks such as social networks. These kernels are fundamental to a majority of machine learning tasks and have received tremendous attention recently in both academia and industry.

The motivating ideas behind this thesis are: the optimization to achieve high performance is a codesign between combinatorial factors such as algo-

rithm, implementation, platforms, etc. By correctly modeling the platform architectures, we are capable of deriving the "right" implementation that can achieve peak FLOPs on given platforms. Furthermore, different platforms can also exhibit drastic differences that are beyond parameter levels. Under such circumstances, we need to specialize in the implementations as well as algorithm choices to adapt to hardware attributes. Therefore, we demonstrate how an analytical model, together with code specialization, and learned dispatch strategies can enable the high performance for these computational kernels on modern hardware.

In Chapter 2, we demonstrate the optimization of the dense kernels in machine learning using convolution neural networks as an example. We propose a model-based analysis to provide a systematic understanding of how the software optimizations such as loop ordering, blocking, instruction scheduling, should be applied based on the hardware architectures. The work in this chapter was published in ICML 2018 [7].

In Chapter 3, we investigate the optimizations of sparse kernels. Sparse kernels present a different set of challenges compared to dense computations. We present the technique that can improve the data-parallelism of the irregular computations of sparse kernels, as well as investigating the effects of algorithm and data structure choices on the performances, using triangle counting application as an example. The work in this chapter was published in HPEC 2018 [17].

In Chapter 4, we investigate the vectorization and code specialization

approach for set intersections on CPUs. This chapter took a step forward from chapter 3 by focusing on intersections whose result sizes are much smaller than the input sizes—this property has been observed in many real-world intersection scenarios. In this chapter, we propose a specialized algorithm for the small-size intersection problem. We demonstrate the specialized algorithm has lower complexity than generalized intersection methods. Next we explore the vectorization accelerations for the specialized intersection algorithm on modern CPUs. We explore the design space for the fast vectorized intersection on CPUs with various vector widths, based on which we present the code generation algorithm that takes the data width and operations as input and generates various implementations. The work in this chapter was published in ICDE 2019 [18].

In Chapter 5, we investigate the optimization of Generalized-Sparse-Matrix-Matrix-Multiplication (SpGEMM) on GPU. We demonstrate how to break down the sparse matrix computation into a set of primitive operations of join and union. We further propose several optimizations for join-based SpGEMM GPU implementations. We perform experimental and theoretic analysis on various implementations with both synthetic and real-world datasets. Finally we propose an online scheduling algorithm that is trained upon a neural network.

In Chapter 6, we present the concluding remarks and discuss possible directions for future work.

Chapter 2

Accelerating Dense Convolutions

Contents

| | | |
|------------|--|-----------|
| 2.1 | Overview | 18 |
| 2.2 | Introduction | 20 |
| 2.3 | Background | 23 |
| 2.4 | High Performance Direct Convolution | 27 |
| 2.4.1 | Strategy for Mapping Loops to Architecture | 27 |
| 2.4.2 | Parallelism | 33 |
| 2.5 | Convolution-Friendly Data Layout | 34 |
| 2.5.1 | Input/Output Layout | 34 |
| 2.5.2 | Kernel Layout | 35 |
| 2.5.3 | Backward Compatibility | 36 |
| 2.6 | Experiments | 38 |
| 2.6.1 | Experimental Setup | 38 |

| | | |
|-------|--------------------------------|----|
| 2.6.2 | Performance | 39 |
| 2.6.3 | Parallel Performance | 40 |
| 2.7 | Chapter Summary | 43 |

2.1 Overview

The computation of convolution layers in deep neural networks typically rely on high performance routines that trade space for time by using additional memory (either for packing purposes or required as part of the algorithm) to improve performance. The problems with such an approach are two-fold. First, these routines incur additional memory overhead which reduces the overall size of the network that can fit on embedded devices with limited memory capacity. Second, these high performance routines were not optimized for performing convolution, which means that the performance obtained is usually less than conventionally expected. In this chapter, we demonstrate that direct convolution, when implemented *correctly*, eliminates all memory overhead, and yields performance that is between 10% to 400% times better than existing high performance implementations of convolution layers on conventional and embedded CPU architectures. We also show that a high performance direct convolution exhibits better scaling performance, i.e. suffers less performance drop, when increasing the number of threads.

Performance normalized to OpenBLAS GEMM on AMD PileDriver

4.0 GHz, 4/4 cores/threads

Normalized Performance [Gflop/s]

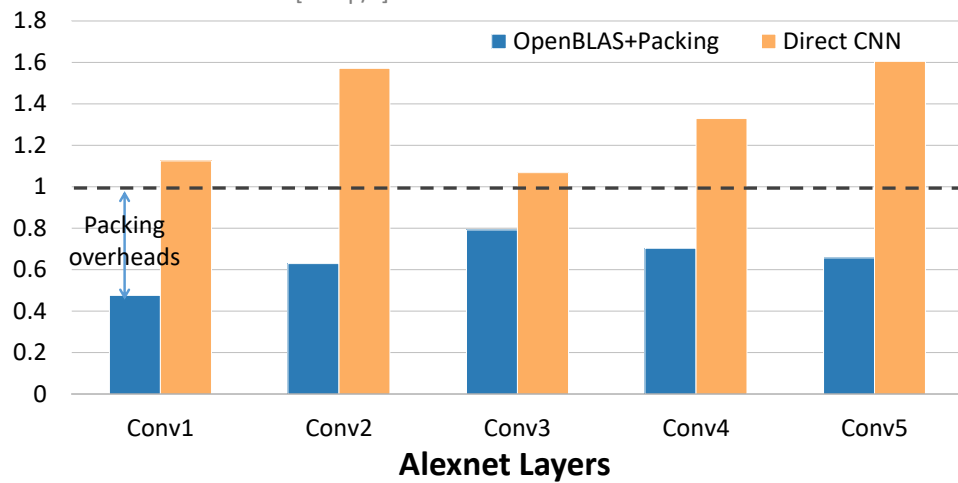


Figure 2.1: High performance direct convolution implementation achieves higher performance than a high performance matrix multiplication routine, whereas matrix-multiplication based convolution implementations suffers from packing overheads and is limited by the performance of the matrix multiplication routine

2.2 Introduction

Conventional wisdom suggests that computing convolution layers found in deep neural nets via direct convolution is not efficient. As such, many existing methods for computing convolution layers [19, 20] in deep neural networks are based on highly optimized routines (e.g. matrix-matrix multiplication) found in computational libraries such as the Basic Linear Algebra Subprograms (BLAS) [8]. In order to utilize the matrix-matrix multiplication routine, these frameworks reshape and selectively duplicate parts of the original input data (collectively known as packing); thereby incurring additional memory space for performance.

There are two problems with this approach: First, the additional work of reshaping and duplicating elements of the input data is a bandwidth-bounded operation that incurs an additional, and non-trivial time penalty on the overall system performance. Second, and more importantly, matrices arising from convolution layers often have dimensions that are dissimilar from matrices arising from traditional high performance computing (HPC) application. As such, the matrix-matrix multiplication routine typically does not achieve as good a performance on convolution matrices as compared to HPC matrices.

To illustrate these drawbacks of existing methods, consider the 4-thread performance attained on various convolution layers in AlexNet using an AMD Piledriver architecture shown in Figure 2.1. In this plot, we present performance of 1) a traditional matrix-multiply based convolution implementation linked to OpenBLAS [21] (blue) and 2) our proposed high performance di-

rect convolution implementation (yellow). Performance of both implementations are normalized to the performance of only the matrix-matrix multiplication routine (dashed line). This dashed line is the performance attained by matrix-matrix multiplication *if packing is free*. Notice that the performance of OpenBLAS + Packing achieves less than 80% of the performance of matrix multiplication itself. This implies that the packing routine degrades the overall performance by more than 20%. In contrast, our custom direct convolution implementation yields performance that exceeds the expert-implemented matrix-matrix multiplication routine, even if packing was free. In addition, we attained the performance *without* any additional memory overhead.

It is timely to revisit how convolution layers are computed as machine learning tasks based on deep neural networks are increasingly being placed on edge devices [22, 23]. These devices are often limited in terms of compute capability and memory capacity [24, 25]. This means that existing methods that trade memory capacity for performance are no longer viable solutions for these devices. Improving performance and reducing memory overheads also bring about better energy efficiency [26]. While many work have focused on reducing the memory footprint of the convolution layer through the approximation [27], quantilization[28], or sparsification of the weights [29], few work tackle the additional memory requirements required in order to use high performance routines.

Contributions. In this chapter, we make the following contributions:

- *High performance direct convolution.* We show that a high perfor-

mance implementation of direct convolution can out-perform a expert-implemented matrix-matrix multiplication based convolution in terms of amount of actual performance, parallelism, and reduced memory overhead. This demonstrates that that direct convolution is a viable means of computing convolution layers.

- *Data layouts for input/output feature maps and kernel weights.* We proposed new data layouts for storing the input, output and kernel weights required for computing a convolution layer using our direct convolution algorithm. The space required for these new data layouts is identical to the existing data storage scheme for storing the input, output and kernel weights *prior* to any packing or duplication of elements.

2.3 Background

In this section, we highlight the inefficiency of computing convolution with existing methods used in many deep learning frameworks.

Fast Fourier Transform-based Implementations

Fast Fourier Transform (FFT)-based implementations [30, 31] of convolution were proposed as a means of reducing the number of floating point operations that are performed when computing convolution in the frequency domain. However, in order for the computation to proceed, the kernel weights have to be padded to the size of the input image, incurring significantly more memory than necessary, specially when the kernels themselves are small (e.g. 3×3).

Alternative approaches have been proposed to subdivide the image into smaller blocks or tiles [32]. However, such approaches also require additional padding of the kernel weights to a convenient size (usually a power of two) in order to attain performance. Even padding the kernel weights to small multiples of the architecture register size (e.g. 8 or 16) will result in factors of 7 to 28 increase in memory requirement. This additional padding and transforming the kernel to the frequency domain can be minimized by performing the FFT on-the-fly as part of the computation of the convolution layer. This, however, incurs significant performance overhead, especially on embedded devices, as we will show in the performance section (Section 2.6).

Matrix Multiplication-based Implementations

Another common approach is to cast the inputs (both the image and kernel weights) into matrices and leverage the high performance matrix-matrix multiplication routine found in the Level 3 Basic Linear Algebra Subprogram (BLAS) [8] for computation. There are two major inefficiencies with this approach:

- *Additional memory requirements.* In order to cast the image into a matrix, a lowering operation is performed to cast the three dimensional image into a two dimensional matrix. Typically, this is performed via an operation conventionally called `im2col` that copies the $W_i \times H_i \times C_i$ image into a $(H_f \times W_f \times C_i) \times (H_o \times W_o)$ matrix which is then used as an input to the matrix-matrix multiplication call. During this lowering process, appropriate elements are also duplicated. The additional memory required grows quadratically with the problem size [20].

Cho and Brand [20] proposed an alternative lowering mechanism that is more memory efficient by reducing the amount of duplication required during the packing process. In their lowering routine, the memory footprint is reduced by an average factor of 3.2 times over `im2col`. This is achieved by eliminating the amount of duplication required at the expense of additional matrix-matrix multiplication calls. Nonetheless, this is still an additional memory requirement, and their computation still relies on a matrix-matrix multiplication that is often sub-optimal for matrices arising from convolution.

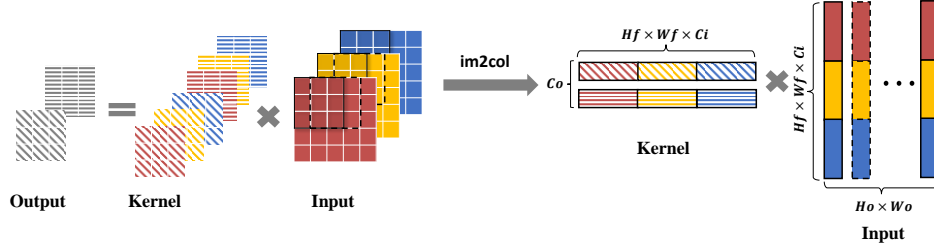


Figure 2.2: The 5×5 input image with 3 different channels (denoted with different colors) is convolved with two separate kernels to obtain a 3×3 output with two output channels. Packing is performed to turn three dimensional input images (left) into a two dimensional matrix (right) in order to utilize a high performance matrix multiplication routine. As C_o and/or $(H_o \times W_o)$ are often less than $H_f \times W_f \times C_i$, performance of standard matrix-matrix multiplication in many BLAS libraries are often sub-optimal.

- *Sub-optimal matrix matrix multiplication.* In most BLAS libraries (e.g. GotoBLAS [33], OpenBLAS [21], BLIS [34]), the matrix-matrix multiplication routine achieves the best performance when the inner dimensions, i.e. the dimension that is common between the two input matrices, of the input matrices are small compared to the overall dimensions of the output matrix. This particular set of matrix shapes is commonly found in scientific and engineering codes, for which these libraries are optimized. However, this particular set of shapes exercise only one out of six possible algorithms for matrix-matrix multiplication [33].

Recall that the `im2col` reshapes the input into a $(H_f \times W_f \times C_i) \times (H_o \times W_o)$ matrix. This means that the inner dimensions of the input matrices are often the larger of two dimensions (See Figure 2.2). As such, the performance of matrix matrix multiplication on this particular set of input shapes is often significantly below the best achievable performance. It

has been shown that alternative algorithms for computing matrix multiplications should be pursued for shapes similar to that arising from convolution layers [35].

Another reason that matrix-matrix multiplication is inefficient for convolution layers is that parallelism in existing BLAS libraries are obtained by partitioning the rows and columns of the input matrices [36]. This partitioning of the matrices skews the matrix shapes even farther away from the shapes expected by the matrix-matrix multiplication routine. As such, the efficiency of the routine suffers as the number of threads increases.

2.4 High Performance Direct Convolution

A naive implementation of direct convolution (See Algorithm 1) is essentially six perfectly-nested loops around a multiply-and-accumulate computational statement that computes a single output element. Any permutation of the ordering of the loops will yield the correct result. However, in order to obtain a high performance implementation of direct convolution, it is essential that these loops and their order are appropriately mapped to the given architecture.

Algorithm 1: Naive Convolution Algorithm

Input: Input \mathcal{I} , Kernel Weights \mathcal{F} , stride s ;
Output: Output \mathcal{O}
for $i = 1$ **to** C_i **do**
 for $j = 1$ **to** C_o **do**
 for $k = 1$ **to** W_o **do**
 for $\ell = 1$ **to** H_o **do**
 for $m = 1$ **to** W_f **do**
 for $n = 1$ **to** H_f **do**
 $\mathcal{O}_{j,k,\ell} += \mathcal{I}_{i,k \times s + m, \ell \times s + n} \times \mathcal{F}_{i,j,m,n}$

2.4.1 Strategy for Mapping Loops to Architecture

Our strategy for mapping the loops to a model architecture is similar to the analytical model for high performance matrix-matrix multiplication [37]. (1) We first introduce the model architecture used by high performance matrix-matrix multiplication. (2) Next, we identify loops that utilize the available computational units efficiently. (3) Finally, we identify the order of the outer loops in order to improve data reuse, which in turn will reduce the amount

of performance-degrading stalls introduced into the computation. In this discussion, we use the index variables show in Algorithm 1 (i, j, k, ℓ, m, n) to differentiate between the loops.

Model architecture

We use the model architecture used the analytical model for high performance matrix-multiplication [37]. The model architecture is assumed to have the following features:

- **Vector registers.** We assume that our model architecture uses single instruction multiple data (SIMD) instruction sets. This means that each operation simultaneously performs its operation on N_{vec} scalar output elements. We also make the assumption that N_{vec} is a power of two. When N_{vec} is one, this implies that only scalar computations are available. In addition, a total of N_{reg} logical registers are addressable.
- **FMA instructions.** We assume the presence of N_{fma} units that can compute fused multiply-add instructions (FMA). Each FMA instruction computes a multiplication and an addition. Each of these N_{fma} units can compute one FMA instruction every cycle (i.e., the units can be fully pipelined), but each FMA instruction has a latency of L_{fma} cycles. This means that L_{fma} cycles must pass since the issuance of the FMA instruction before a subsequent dependent FMA instruction can be issued.

- **Load/Store architecture.** We assume that the architecture is a load/store architecture where data has to be loaded into registers before operations can be performed on the loaded data. On architectures with instructions that compute directly from memory, we assume that those instructions are not used.

Loops to saturate computations

The maximum performance on our model architecture is attained when all N_{fma} units are computing one FMA per cycle. However, because each FMA instruction has a latency of L_{fma} cycles, this means that there must at least be L_{fma} independent FMA instructions issued to each computational unit. As each FMA instruction can compute N_{vec} output elements, this means that

$$\mathcal{E} \geq N_{\text{vec}} N_{\text{fma}} L_{\text{fma}}, \quad (2.1)$$

where \mathcal{E} is the minimum number of independent output elements that has to be computed in each cycle in order to reach the maximum attainable performance.

Having determine that at least \mathcal{E} output elements must be computed in each cycle, the next step is to determine the arrangement of these output elements within the overall output of the convolution layer. Notice that the output has three dimensions ($H_o \times W_o \times C_o$) where H_o and W_o are primarily a function of the input sizes, while C_o is a design parameter of the convolution layer. Since \mathcal{E} must be a multiple of N_{vec} , i.e. a power-of-two, and C_o can be chosen (and is the case in practice) to be a power-of-two, *the j loop is chosen*

as the inner-most loop.

As the minimum number \mathcal{E} is highly dependent on the number and capability of the FMA computation units, we want to ensure that there are sufficient output elements to completely saturate computation. As such, *the k loop that iterates over the elements in the same row of the output image is chosen to be the loop around the j loop*¹.

Loops to optimize data reuse

The subsequent loops are ordered to bring data to the computational units as efficiently as possible.

Recall that the inner two loops (j and k) iterate over multiple output elements to ensure that sufficient independent FMA operations can be performed to avoid stalls in the computation units. As our model architecture is a load/store architecture, this means that these output elements are already in registers. Therefore, we want to bring in data that allows us to accumulate into these output elements.

Recall that to compute a single output element, all $H_f \times W_f \times C_i$ weights are multiplied with the appropriate element from the input image and accumulated into the output element. This naturally means that *the next three loops in sequence from the inner-most to outer-most are the i, n, m loops*. This order of the loops is determined based on the observation that the input of most convolution layers is the output of another convolution layer. This means that

¹It should be noted that the choice of W_o over H_o is arbitrary as the analysis is identical.

Algorithm 2: Reorder Convolution Algorithm

Input: Input \mathcal{I} , Kernel Weights \mathcal{F} , stride s ;
Output: Output \mathcal{O}
for $\ell = 1$ **to** H_o **do**
 for $n = 1$ **to** H_f **do**
 for $m = 1$ **to** W_f **do**
 for $i = 1$ **to** C_i **do**
 for $k = 1$ **to** W_o **do**
 for $j = 1$ **to** C_o **do**
 $\mathcal{O}_{j,k,\ell} += \mathcal{I}_{i,k \times s + m, \ell \times s + n} \times \mathcal{F}_{i,j,m,n}$

it would be advisable if data from both the input and output are accessed in the same order. As such, we want to access the input elements in the channels (i) before rows (n), which gives us the i, n, m ordering of the loops.

Having decided on five of the original six loops, this means that *outermost loop has to be the l loop*. This loop traverses over the remaining through different rows of the output. The original loop order as shown in Algorithm 1 (i, j, k, l, m, n) is transformed to the (l, n, m, i, k, j) loop ordering as shown in Algorithm 2.

Blocking for the memory hierarchy

Register Blocking. The astute reader will recognize that we have conveniently ignored the fact that \mathcal{E} , the number of minimum output elements required to sustain peak performance, is upper bounded by the number of registers as described by the following inequality:

$$\mathcal{E} \leq N_{\text{reg}} N_{\text{vec}}. \quad (2.2)$$

This upper bound imposed by the number of available registers means that at most $N_{\text{reg}}N_{\text{vec}}$ elements can be kept in the registers. This means that instead of iterating over all $C_o \times W_o$ elements, loop blocking/tiling [38] with block sizes of $C_{o,b}$ ² and $W_{o,b}$ has to be applied to the two inner-most loops to avoid register-spilling that will degrade performance.

Applying loop blocking to the original j and k loops decomposes a row from each of the output channel into smaller output images, each of which having a row width and output channel of $W_{o,b}$, and $C_{o,b}$ respectively. Since loop blocking decomposes the overall convolution into smaller convolutions, the loop ordering previously described remains applicable. However, we now need to determine how to traverse over the smaller convolutions.

The final algorithm applying loop blocking is shown in Algorithm 3. The loops j' and k' iterate over the blocks in the channel and row dimensions of the output, respectively. We make the observation accessing input elements in the same row will require us to also access kernel weights in the same row. This suggest that the ordering of the loop should be similar to the loops traversing across the kernel weights. As such, the k' loop is nested between ℓ and n loops. The j' loop is set to be the outermost loop since it is a parallel loop that facilitates parallelization.

Cache Blocking. On architecture with more levels in the memory hierarchy, i.e. architectures with caches, we can further partition the input dataset into smaller partitions such that they fit into the appropriate levels of

² $C_{o,b}$ is chosen to be a multiple of the vector length N_{vec} so that SIMD instructions can be better used for computation.

the cache. Recall that the loops around jj and kk accumulates $H_f \times W_f \times C_i$ intermediate results into the output stored in the register. Since H_f and W_f , i.e. the size of the kernel weights, are typically smaller than C_i , we choose to partition the i loop which iterates over C_i input channels for the next level in the memory hierarchy.

The final algorithm for high performance direct convolution is shown in Algorithm 3.

Algorithm 3: Parallelized Direct Convolution Algorithm

Input: Input \mathcal{I} , Kernel Weights \mathcal{F} , stride s ;
Output: Output \mathcal{O}
for $j' = 1$ **to** $C_o/C_{o,b}$ **in Parallel do**
 for $i' = 1$ **to** $C_i/C_{i,b}$ **do**
 for $\ell = 1$ **to** H_o **do**
 for $k' = 1$ **to** $W_o/W_{o,b}$ **do**
 for $n = 1$ **to** H_f **do**
 for $m = 1$ **to** W_f **do**
 for $ii = 1$ **to** $C_{i,b}$ **do**
 for $kk = 1$ **to** $W_{o,b}$ **do**
 for $jj = 1$ **to** $C_{o,b}$ **do**
 $\mathcal{O}_{j'C_{o,b}+jj,k'W_{o,b}+kk,\ell} \quad +=$
 $\mathcal{I}_{i'C_{i,b}+ii,sk'W_{o,b}+kk+m,\ell s+n} \times$
 $\mathcal{F}_{i'C_{i,b}+ii,j' \times C_{o,b}+jj,m,n}$

2.4.2 Parallelism

In order to identify possible parallel algorithms, we first make the observation that all output elements can be computed in parallel. Since the output is a three dimensional object ($H_o \times W_o \times C_o$), this means that parallelism can be extracted in at least three different dimensions.

Our direct convolution implementation extracts parallelism in the output channel (C_o) dimension. Each thread is assigned a block of output elements to compute, where each block of output elements is of size $H_o \times W_o \times C_o/p$, where p is the number of threads used.

2.5 Convolution-Friendly Data Layout

We proposed new data layouts for the input and kernel data so that data is accessed in unit stride as much as possible. This improves data access and avoids costly stalls when accessing data from lower levels of the memory hierarchy. A key criteria in revising the layout is that the output and the input image should have the same data layout. This is because the input of most convolution layers is the output of another convolution layer. Keeping them in the same data layout will avoid costly data reshape between convolution layers. However, to ensure compatibility with original input images, we do not impose the proposed layout on the inputs to the first convolution layer.

2.5.1 Input/Output Layout

We want to access the output data in unit stride. Therefore, we determine the output data layout by considering how the elements are accessed using the loop ordering shown in Algorithm 3. Data accessed in the inner loops should be arranged closer together in memory than data accessed in the outer loops.

Five loops (j, k, ℓ, kk, jj) iterate over the output data, which suggests

a five-dimensional data layout. However, this is sub-optimal if we were to use it for the input data. This is because W_f elements in an input row is required to compute one output element. With the five-dimensional layout, a row of the input is blocked into blocks of $W_{o,b}$ elements. This means that output elements that require input elements from two separate $W_{o,b}$ blocks will incur a large penalty as these input elements are separated over a large distance in memory. As such we do not layout the data according to the kk loop.

The proposed input/output layout is shown in Figure 2.3 (left). The output data is organized into sequential blocks of $H_o \times W_o \times C_{o,b}$, where in each block, elements are first laid out in the channel dimension, before being organized into a $H_o \times W_o$ row-major-order matrix of pencils of length $C_{o,b}$.

2.5.2 Kernel Layout

Similar to the input/output layout, we use the loop ordering to determine how to order the kernel weights into sequential memory. Notice that the ℓ, k', kk loops in Algorithm 3 iterates over the height and width of the output in a single output channel. As all output elements in the same output channel share the same kernel weights, these loops provide no information as to how the kernel weights should be stored. As such, we only consider the remaining six loops.

The kernel layout proposed by the remaining six loops is shown in Figure 2.3 (right). The fastest dimension in the kernel layout is the blocked output channel ($C_{o,b}$) dimension, and is dictated by the inner-most loop. The

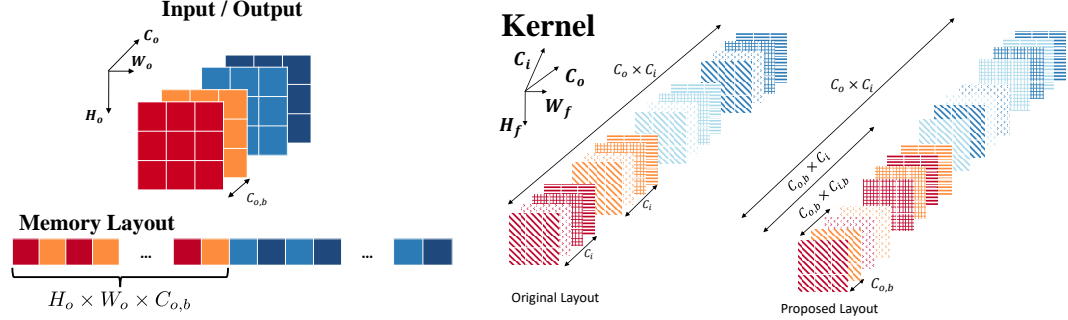


Figure 2.3: Convolution-friendly layout for input/output (left) and kernel weights (right). The output data is organized into sequential blocks of $H_o \times W_o \times C_{o,b}$, where in each block, the fastest dimension is in the channel dimension, followed by the column and row dimension of the output. The kernel weights are organized into blocks of $H_o \times W_o \times C_{o,b} \times C_{i,b}$. The fastest dimension is the blocked output channel, followed by the blocked input channels, kernel width and height, input channels and then the output channels.

remaining dimensions from fastest to slowest are the blocked input channel ($C_{i,b}$), followed by the columns (W_f) and rows (H_f) of the kernel, the input channels ($C_i/C_{i,b}$) and finally the output channels ($C_o/C_{o,b}$).

2.5.3 Backward Compatibility

Given the successful deployment of convolution neural nets (CNN) in the field, the proposed change in data layout will mean that trained networks are unable to directly benefit from our proposed direct convolution implementation. However, in order for a trained network to use our proposed algorithm, there is only a one-time cost of rearranging the kernel weights into the proposed data layout. Other network layers such as skip layers [39], and activation layers are point-wise operations that should not require any significant change in

the implementation. Nonetheless, reordering the loops used to compute these layers will likely yield better performance.

Table 2.1: Details of specific architecture used

| | Intel i7-4770K | AMD FX(tm)-8350 | ARM Cortex-A57 |
|--------------|-------------------|--------------------|-------------------|
| Architecture | Haswell | Piledriver | ARMv8 |
| Frequency | 3.5GHz | 4GHz | 1.1GHz |
| Cores | 4 | 4 | 2 |
| N_{vec} | 8 | 8 | 4 |

2.6 Experiments

In this section, we present performance results of our direct CNN implementation against existing convolution approaches on a variety of architecture. A mix of traditional CPU architectures (Intel and AMD) and embedded processor (ARM) found on embedded devices are chosen.

2.6.1 Experimental Setup

Platform We run our experiments on Intel Core i7-4770K, AMD FX(tm)-8350, ARM Cortex-A57 architectures. The architecture details of those platforms are shown in Table 2.1.

Software. We implement our direct convolution using techniques from the HPC community [40]. We compare performance our direct convolution implementation against matrix-multiplication based convolution linked to high performance BLAS libraries. For matrix-multiplication based convolution, the input data is first packed into the appropriate matrix using Caffe’s `im2col` routine before a high performance single-precision matrix-multiplication (`SGEMM`)

routine is called. The **SGEMM** routine used is dependent on the architecture. On Intel architecture, we linked to Intel’s Math Kernel Library (MKL) [41], while OpenBLAS [21] is used on the other two architectures. We also provide comparison against the FFT-based convolution implementation provided by NNPACK [32], a software library that underlies the FFT-based convolutions in Caffe 2 [42]. As NNPACK provides multiple FFT-based (inclusive of Winograd) implementations, we only report performance attained by the best (fastest) implementation. We use the benchmark program supplied by NNPACK to perform our tests.

Benchmarks. All implementations were ran against all convolution layers found in AlexNet [43], GoogLeNet [44] and VGG [45]. The different convolution layers in these three CNNs span a wide range of sizes of input, output and kernel weights. They are also commonly used as benchmarks for demonstrating the performance of convolution implementations.

2.6.2 Performance

The relative performance of the different implementations normalized to the **SGEMM**+ packing method are shown in Figure 2.4. Our direct convolution implementations out-performs all **SGEMM**-based convolutions on all architectures by at least 10% and up to 400%. Our direct convolution out-performs **SGEMM** even when the BLAS library (MKL) optimizes for the appropriate matrix shapes arising from convolution. Against a BLAS library (OpenBLAS) that only optimizes for HPC matrices, we see a minimum of 1.5 times performance

gain on 4 threads.

In comparison with the FFT-based implementations provided by NNPack, the direct convolution implementation significantly out-performs FFT-based implementations for all layers on the ARM. As FFTs are known to be memory-bandwidth bound, we suspect that the FFT may be the bottleneck on a smaller architecture such as the ARM where available bandwidth may be limited. On the Intel architecture, the results are similar with direct convolution outperforming FFT-based implementations. However, in this case the FFT-based implementations are able to out-perform the **SGEMM**-based approach only when the dataset is “sufficiently large” to amortize the cost of performing the FFT itself. The AMD architecture is not supported by NNPACK.

2.6.3 Parallel Performance

In Figure 2.5, we compare the scalability of our convolution performance by parallelizing the implementation with increasing number of threads. On all architecture, we report performance per core for multi-threaded implementations normalized to the performance attained on one thread. Notice that the performance per core for existing matrix-multiplication based convolutions decrease significantly as we increase the number of threads. This is an indication that as we increase the number of threads, the processors are utilized less efficiently by the existing matrix-multiplication based implementations. Our direct CNN implementation demonstrates minimal drop in performance per core as we increase the number of threads. It is only when the number of threads is twice

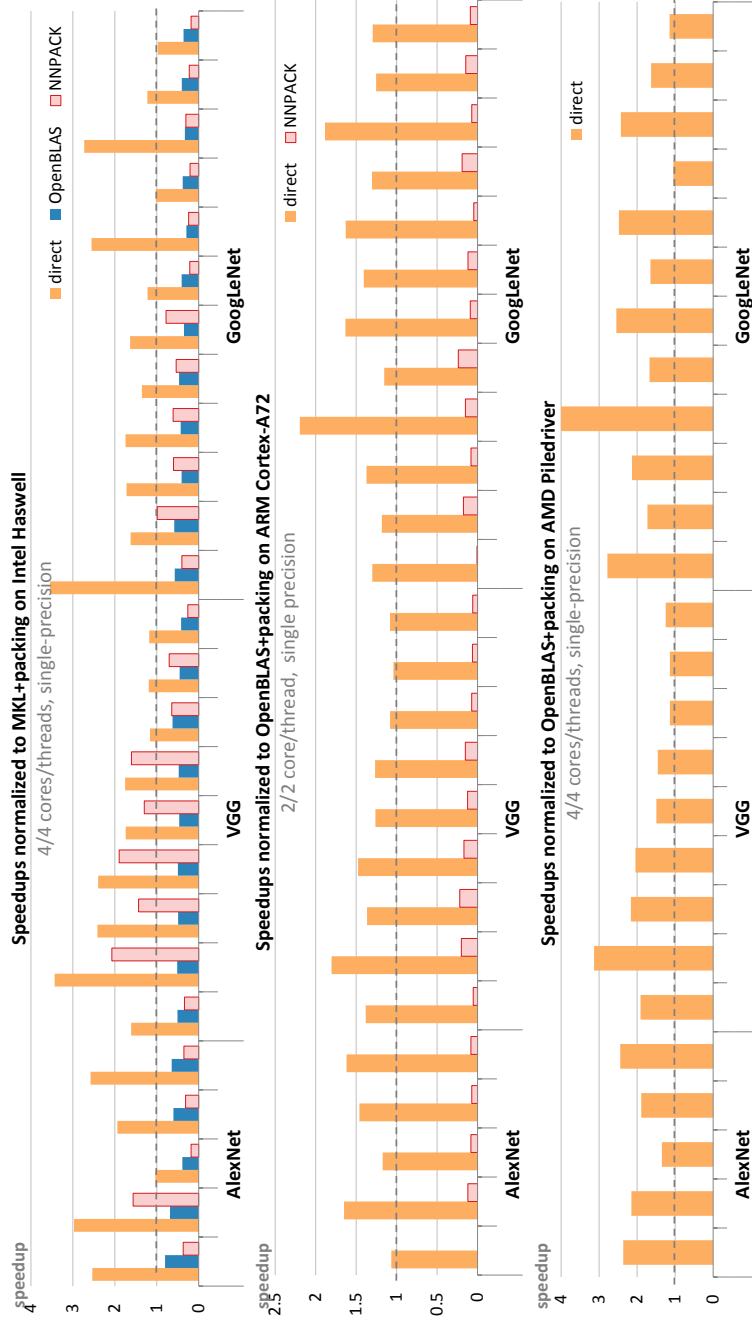


Figure 2.4: Performance of direct convolution against existing high performance FFT-based and SGEMM-based convolution implementations. Performances of all implementations are normalized to the performance of SGEMM + im2col routine. Direct convolution is highly competitive against all other implementations achieving between 10% and 400% improvement in performance even against a BLAS library (Intel MKL) that optimizes for matrix shapes arising from convolution layers.

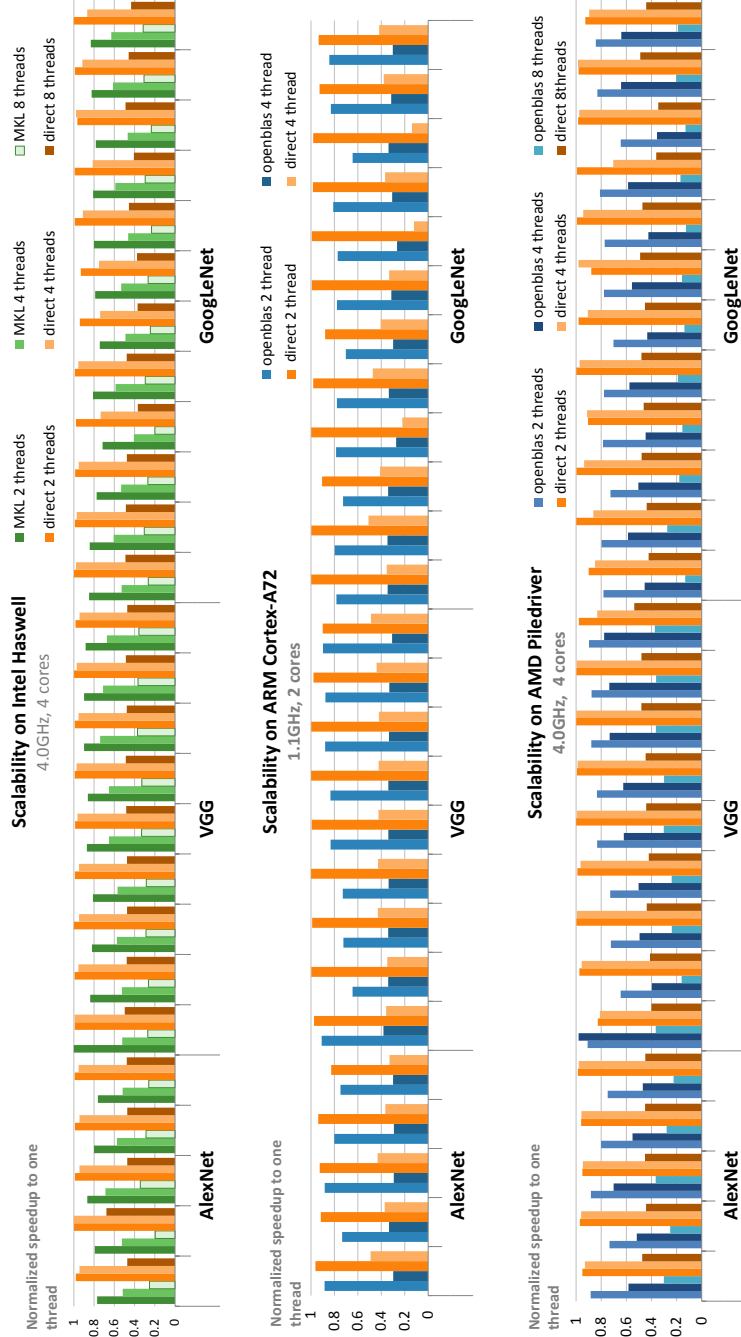


Figure 2.5: Scaling behavior with increasing number of threads. Our direct convolution implementation retains high GFLOPs per core performance as we increase the number of threads from 1 to the number of available cores. This is indicative of an efficient parallelized algorithm. When the number of threads exceeds the number of cores, excessive contention results in a significant drop in performance per core. In contrast, **SGEMM** has poor scalability even when the number of threads is low (e.g. 2).

as much as the number of physical cores does the performance per core of our implementation drops significantly. This is expected and important as it indicates that our implementation utilizes the compute units effectively and increasing the number of threads beyond the number of physical compute units creates excessive contention for the compute resources, thereby resulting in a sharp drop in performance per core.

2.7 Chapter Summary

In this chapter, we demonstrate that direct convolution, a computational technique largely ignored for computing convolution layers, is competitive with existing state of the art convolution layer computation. We show that a high performance direct convolution implementation not only eliminates all additional memory overhead, but also attains higher performance than the expert-implemented matrix-matrix-multiplication based convolution. We also show that our implementation scales to larger number of processors without degradation in performance as our implementation exploits the dimension of the kernel that has the highest amount of parallelism. In contrast, current high performance matrix-multiply based implementations do not scale as well to a larger number of processors.

Our direct convolution implementation currently attains 87.5%, 58.2% and 88.9% of the theoretical peak of the Intel, AMD, and ARM architecture, where as the **SGEMM** on HPC matrices attains peaks of 89% 54% and 92% the same architecture. While we have shown that our direct convolution imple-

mentation is competitive (within 3% of peak **SGEMM** performance), we believe that the performance gap between our direct convolution, and **SGEMM** on HPC matrices can be closed by taking an auto-tuning [46, 12] or analytical approach [47, 37] to identifying the blocking parameters of the different loops. These approaches will also allow the exploration of different combinations of parallelism to determine suitable parallelism strategies. This is something we intend to pursue in the near future.

Another possible direction arising from this work is to use similar design techniques to optimize the backward process to update both in image and kernel. Given the similarity of the forward and backward process, we believe that only minor changes to the loop ordering are required.

Finally, we believe that our direct convolution algorithm can be ported to the GPU. Our proposed data layouts are similar to the layout required for the **StridedBatchedGemm** operation [48]. As this operation and data layout is currently supported on Nvidia GPUs using cuBLAS 8.0 [49], this lends support to our belief that our algorithm can be easily ported to the GPU.

Chapter 3

Sparse Computational Kernels: A Preliminary Study

Contents

| | | |
|------------|---|-----------|
| 3.1 | Expressing Computation Kernels in Sparse Ap- plications | 47 |
| 3.2 | Accelerating Sparse Computation: A Case Study with Triangle Counting | 49 |
| 3.2.1 | Introduction | 49 |
| 3.2.2 | Algorithm | 51 |
| 3.2.3 | Parallel Triangle Counting | 52 |
| 3.2.4 | Preliminary Exploration on Optimizations | 53 |
| 3.2.5 | Profiling and Analysis | 56 |
| 3.2.6 | Understanding the Optimizations | 64 |
| 3.2.7 | Performance Results | 65 |

Sparse computations are important kernels used in many artificial intelligence and machine learning applications. For example, many social analytics tasks such as anomaly detection, community classification are built upon graph methods, and graph computations can be described from the perspectives of sparse matrix computations. Graph-convolutional-neural-networks (GNN) is an emerging machine learning models used in tasks such as learning molecular fingerprints, predicting protein interface. GNN works with graph structure data and such data is also an example of sparse matrix format. Sparse neural networks is another sparse application that has attracted tremendous attention in the machine learning tasks. Sparse neural networks can greatly reduce the model size of the original dense neural networks by pruning a large fraction of the connecting edges. As a result, the deep neural network models can fit into small systems or devices.

Sparse kernels are challenging to optimize compared to dense kernels due to their notorious irregular patterns of computations. In this chapter, we are going to demonstrate that many sparse computations are built upon set primitives such as intersections. And next we present strategies that can leverage the data parallelism in modern systems to accelerate those operations that were originally irregular and difficult to parallel.

3.1 Expressing Computation Kernels in Sparse Applications

Sparse kernels for graph applications Matrix algebra has been a useful tool in graph theory as an alternative way to formulate graph algorithms [50]. As a matter of fact, the graph representation as collections of vertices as well as matrices are considered interchangeable formats and equally important in the decades of graph theory research. Researchers have leveraged the duality between graph and matrix representation and proposed the standard for implementing graph libraries [51, 52]. The NIST Sparse Basic Linear Algebra Subprograms (BLAS) was one of the first standard for sparse problems which adopts similar standards of dense linear algebra subroutines (BLAS) [53]. It classifies the sparse operation into three levels — Sparse Vector (Level 1), Matrix Vector (Level 2), and Matrix Matrix (Level 3) operations. These BLAS were designed for solving the kinds of sparse linear algebra operations that arise in finite element simulation techniques. More recently, researchers extend such sparse algebra standards to represent more general graph algorithms — GraphBLAS. In particular, GraphBLAS extend the BLAS idea to address the needs of graph algorithms by generalizing the pair of operations involved in the computations to the notation of *semiring*. *semiring* refers to certain \oplus and \otimes combinations over certain sets of scalars. Some common Semiring examples include:

- standard arithmetics, where

$\otimes = \times$, $\oplus = +$, and $A, B, C \in R$

- max-plus semiring, where

$\otimes = \max$, $\oplus = +$, and $A, B, C \in -\infty \cup R$

- max-min semiring, where

$\otimes = \max$, $\oplus = \min$, $A, B, C \in \infty \cup R_{\leq 0}$

With this more general case of sparse matrix multiply, a wide range of graphs algorithms can be implemented. We will discuss this in more detail in section 3.2.

Sparse kernels for neural networks The primary mathematical operation performed by a DNN network can be captured using the sparse matrix operator as well [54]. In particular, the inference process in a deep neural network is a step executed repeatedly during training to determine both the weight matrices W_k and the bias vectors b_k of the neural network. The inference computes the following equation:

$$y_{k+1} = h(W_k y_k + b_k) \quad (3.1)$$

where h is a non-linear function applied to each element of the vector. An example of h function is the rectified linear unit (ReLU) given by $h(y) = \max(y, 0)$ which sets values less than 0 and leaves other values unchanged. When training a DNN, it is common to compute multiple y_k vectors at once, which can be denoted as the matrix Y_k . In matrix form, the inference step becomes $Y_{k+1} = h(W_k Y_k + B_k)$. Therefore, the inference computation can be

rewritten as a linear function over semirings $Y_{k+1} = W_k Y_k \otimes B_k \oplus 0$, where $\oplus = \max$ and $\otimes = +$.

3.2 Accelerating Sparse Computation: A Case Study with Triangle Counting

In this section, we use triangle counting task as an example to investigate the computational kernels in graph applications. In the next section, we will focus on the acceleration of the sparse computational kernels.

3.2.1 Introduction

As triangle counting is becoming a widely-used building block for a large amount of graph analytics applications, there is a growing need to make it run fast and scalable on large parallel systems. In this section we conduct a preliminary exploration on the optimizations of triangle counting algorithms on shared-memory system with large dataset.

Graph analytics is becoming increasingly important in a growing number of domains and there is a need to build fast and scalable systems for graph analytics. Nowadays, there is a growing trend for the design of graph analytics engines for shared-memory systems [4, 5, 6]. Shared-memory system has lower communication costs and lower data access latency. This can potentially lead to better performance compared with distributed memory systems. On top of that, a state-of-art high-end multi-core machine can integrate hundreds

of cores, and terabytes of memory [55]. This further enables shared memory systems to process large-scale datasets in memory.

In this section, we explore one of the basic applications in graph analytics: triangle counting. Triangle counting is one of the frequently computed building block operation in graph analytics, therefore it is important to make it run fast and scalable to large systems. There are rich amount of graph frameworks that can target a general set of graph applications. Those frameworks have different frontend designs[2, 3, 4, 5, 6, 7]. For example, [2] is based on vertex-iterator programming model where a utility function is supplied and executed on each vertex, [3] uses linear algebra language as primitives. And others are based on domain-specific languages, etc. However, those graph framework’s target are the broad set of graph applications, their backend optimizations may be too general to be effective for triangle countings.

In this section, we plan to study the optimizations specific for triangle counting on the shared-memory systems. There are different algorithms for triangle counting. And for these algorithms, one could have different implementations, including a variety of hashing and merging tweaks. In order to get optimal performance for triangle counting, it is essential to consider system characteristics, as well as input graph’s properties to find the effective optimization method. This chapter conducts a preliminary explorations on whether such optimizations can be effective for large-scale triangle counting on shared-memory multi-core systems. The structure of the chapter is as follows: the discussion of the triangle counting algorithm is presented in section 3.2.2.

Then the baseline parallel implementation is given in section 3.2.3. Section 3.2.4 discusses a variety of optimization techniques and analyzes whether they can be effective in practice. The performance results are presented in section 3.2.7.

3.2.2 Algorithm

The straight-forward algorithm for triangle counting using the linear algebra language can be illustrated as [56, 57]:

$$A^2 \circ A \tag{3.2}$$

where A is the adjacency matrix of the input graph. The square of A gives all the wedges (connected two edges) and the element-wise multiplication with A closes the wedge and forms a triangle. Afterwards, the algorithm sums over each element in the resultant matrix. However, in this algorithm each triangle will be counted repetitively for six times. An improvement to reduce the double counting in the baseline algorithm can use half the adjacency matrix. In this way, each triangle will only be counted once.

There is an algorithm that can further reduce the triangle counting complexity. The compact-forward algorithm in [58] can outperform the above two baseline algorithms by greatly reducing the number of false positives. The algorithm consists of two steps. The first step is direction assignment: it assigns a direction to each undirected edge such that the edge points from the

low-degree vertex to the high-degree vertex. Then the second step counts the number of triangles based on the directed graph. There are two benefits of introducing a direction to the edge: First of all, it can avoid double counting. Second, it reduces the (out-)degree of nodes, especially those with high degrees. As the counting has the quadratic complexity to the vertex's (out-)degree and the high (out-)degree vertex dominates the computation among all the vertices, this can greatly reduce the complexity.

3.2.3 Parallel Triangle Counting

The most time consuming part of the compact forward algorithm is the second step—counting triangles from the directed graph. The pseudocode for this step is shown in Algorithm 4. In this step, the algorithm iterates every vertex (v), and for each one of its (directed) neighbors u , sums up the number of common neighbors. We first run a baseline parallel triangle counting algorithm on the shared memory systems. The baseline implementation is based on [59]. The implementation simply parallelizes the loop among vertex v (line 2) and the subsequent loop among the neighborhood of v (line 3).

Algorithm 4: count triangles in directed graph

Input: directed graph G

Output: *count*: number of triangles

```

1 count  $\leftarrow 0$ 
2 for  $v \in G.vertices$  do
3   for  $u \in v.neighbors$  do
4      $count \leftarrow count + \text{common neighbors of } u \text{ and } v$ 
5 return count
```

In terms of the counting phase in line 4, the implementation has explored two basic ways to implement counting — hashing-based method and sort-merge based method. The author finds the sort-merge based implementation is faster than the hash-based method.

3.2.4 Preliminary Exploration on Optimizations

Counting the common elements of two vertex's neighbor list is the most time consuming computation. In this section, we will mostly focus on the optimizations for the counting phase in line 5. Counting the common elements of two neighbor lists is essentially founding out the intersecting elements between two sets. Therefore optimizations explored previously in set-intersection problems may be applied here.

Algorithm 5: count common elements in two sorted lists

```

1  $i \leftarrow 0, j \leftarrow 0, counts \leftarrow 0$ 
2 while  $i < A.size$  AND  $j < B.size$  do
3   if  $A[i] < B[j]$  then
4      $i++$ 
5   else
6     if  $A[i] > B[j]$  then
7        $j++$ 
8     else
9        $i++, j++, count++$ 
10 return  $count$ 

```

Hashing One technique to optimize the set intersection operation is to use hashing. The hashing method maps each vertex's neighborhood into a corre-

sponding hash table. To count the common elements in two vertex’s neighborhood, we can iterate through the elements in the smaller neighborhood list and probes into the hash table of the bigger neighborhood. The cost of the hashing based set intersection method is $\min(n_1, n_2)$, where n_1 is the size of small list and n_2 is the size of the large list.

Merging is another technique to count the common elements in two lists. It first sorts each vertex’s neighborhood list in increasing order and counts the common elements of two sorted list by linear scan through them. The baseline code to count the intersection of two sorted lists via merging method is shown in Algorithm 5. The cost for the merging based set intersection is $\max(n_1, n_2)$.

Exploiting binary search to accelerate merging When the two list size is quite different, we can binary search the smaller list in the large one. The cost of the counting phase can be reduced to $n_1 \log(n_2)$.

Exploiting SIMD to accelerate merging There has been plenty of work exploring how to utilize the SIMD unit to accelerate set intersection computation [60, 61, 62]. We implemented a baseline SIMD algorithm based on [62]. Assume the system SIMD width is four-elements, the SIMD intersection algorithm is illustrated with an example in Figure. 3.1. In this example, during the first iteration the first four-element of list A and B will be compared (the instructions used will be explained subsequently). Based on comparison of the tail elements (i.e. the 4th element), one of the pointers is decided to move.

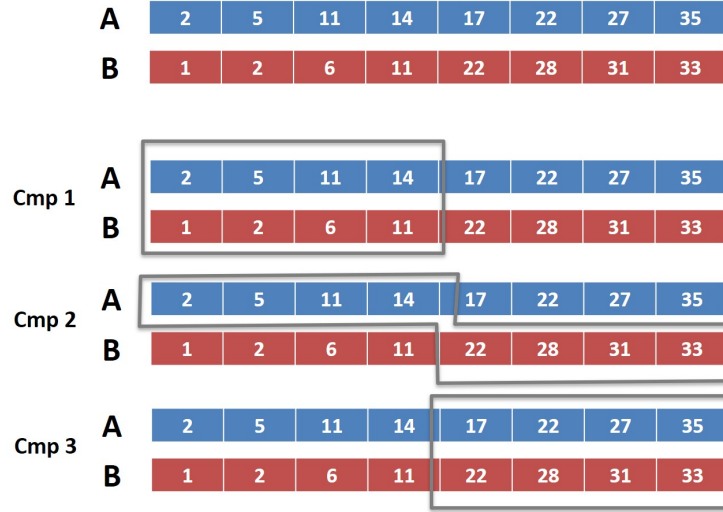


Figure 3.1: SIMD algorithm assuming simd width is 4 elements.

In the example case, pointer of the list B will advance to the next 4-element block because its 4th element is smaller than list A. In second iteration, the first four elements of A are compared with the second 4-element block of B and finally the second 4-element blocks of A and B are compared.

On CPUs with AVX instruction set architecture, the algorithm exemplified in Fig. 3.1 can be implemented using `_mm_cmpeq` and `_mm_shuffle` intrinsics. `_mm_cmpeq` can compare multiple elements at once and produce a resulting bit mask. In order to conduct all-to-all comparisons between A and B, only one `_mm_cmpeq` is not enough — the SIMD block of B needs to left-shift using `_mm_shuffle` intrinsics and then compare with A again till the right-most element in B's element block is compared with the left-most element of A. The process is illustrated in Fig. 3.2.

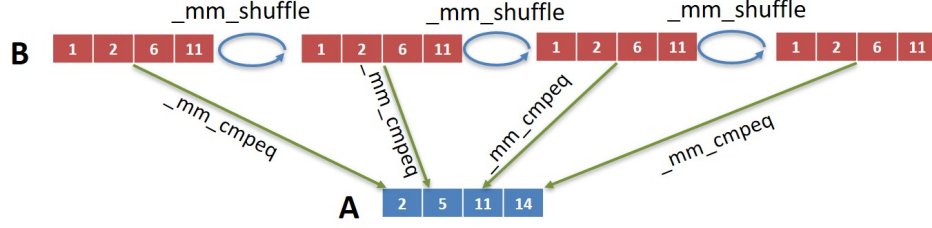


Figure 3.2: SIMD algorithm illusion.

hybrid hashing and merging A hybrid method represents a node's neighborhood with a hybrid structure between sparse list and hash table. The hybrid method partitions the data range into fix-sized (K) blocks. Each block is indexed by its base value and has an associated bit-vector of length K . Each bit of the bit-vector indicates the existence of an element on that position. For example, given a list $\{0, 1, 4, 52, 102, 493, 534\}$ and K as 64, the list can be compressed as $\{0, 64, 448, 512\}$. We can see the size of the list reduces from 8 to 4. In this way, the number of comparisons can be reduced subsequently. To count the common elements in the compressed block list, it firstly scan through the block lists to find blocks with common base values via the merge-sort methods and then count the common elements in the two blocks by comparing the two corresponding indicator vectors.

3.2.5 Profiling and Analysis

Whether the above algorithms will work largely depends on the property of the dataset. In this section, we start with the profiling results of some representative graphs datasets in order to exam the effectiveness of the above

techniques.

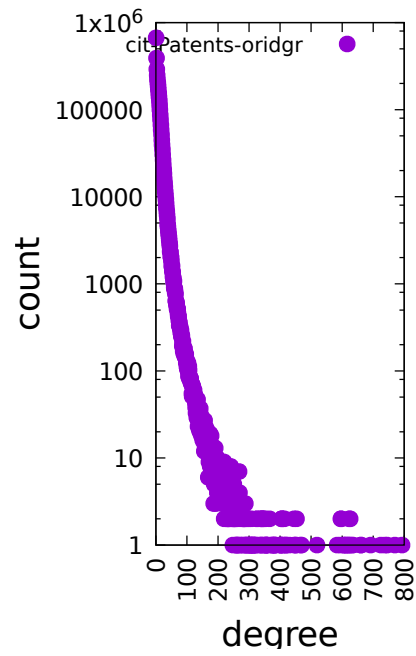
Cit-Patents. This dataset has 3,774,768 vertices and 16,518,947 edges. The original maximum degree 793. The maximum degree after sorting is 73. The degree distribution is shown in Fig. 3.3. We can see that after direction assignment, the out-degree distribution still follows a power-law distribution. Majority of vertices has small out-degree.

Friendster. This dataset has 65,608,366 vertices and 1,806,067,135 edges. The original maximum degree among all the nodes is 5214. The maximum out-degree in the directed graph is 2389. And their distribution is in Fig. 3.4.

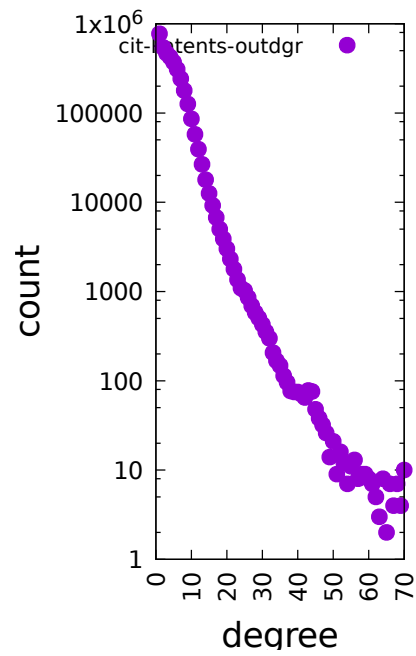
Graph500-scale23. This dataset has 4,606,314 vertices and 129,250,705 edges. The original maximum degree 272176. The maximum degree after sorting is 1376. The degree distribution is shown in Fig. 3.5.

We highlight the following observations based on the profilings:

Observation 1. In the directed graph, the number of nodes with high out-degrees is small. However, intersections between those high out-degree nodes' neighbor list dominates the total run-time. And this is not difficult to verify. If we assume that after reordering the rows from highest degree to lowest degree in the adjacent matrix of the graph, the number of non-zero element (nnz, equivalently out-degree) follows a power-law distribution over the rows, then the comparing cost per row will also be skewed and resemble a power-law distribution. As plotted out in Figure 3.6, we can see that if the nnz element follows a power-law distribution of x^{-1} , then in all the two list(a.k.a.

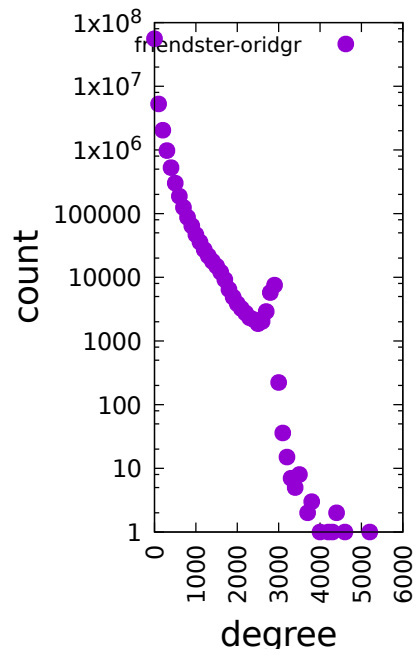


(a) Original degree distribution.

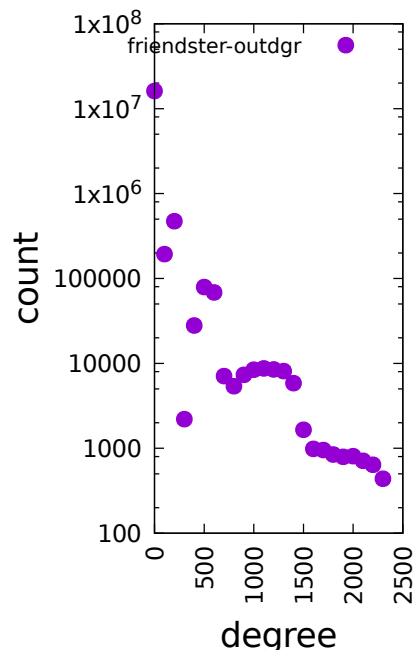


(b) Out-degree distribution after direction assignment

Figure 3.3: Degree distribution of cit-Patents.

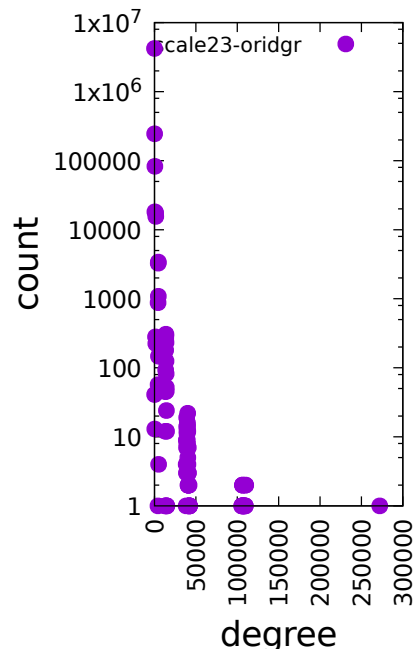


(a) Original degree distribution.

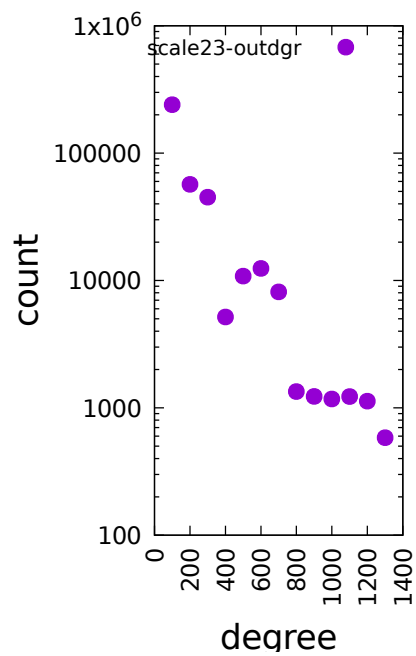


(b) Out-degree distribution after direction assignment.

Figure 3.4: Degree distribution of friendster.



(a) Original degree distribution.

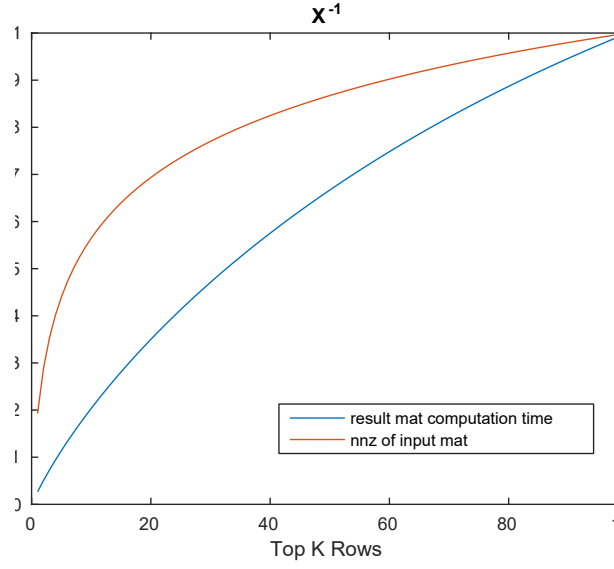


(b) Out-degree distribution after direction assignment.

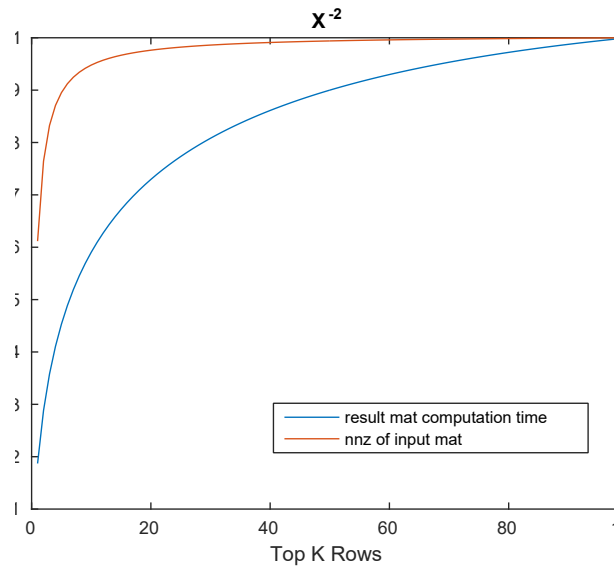
Figure 3.5: Degree distribution of graph500-scale23.

rows) comparisons, the top 40% rows takes 50% of the total comparison cost. If the nnz element distribution is x^{-2} , then the top 10% rows will take up over 50% of the total comparison cost. In general, more skewed the nnz element distribution is in the graph’s adjacent matrix, the more skewed the computation cost will be towards the dense part of the matrix. Figure 3.7 shows the real time and computation cost distribution on some given datasets. Consistent with the theoretical results, the computation cost is also mostly centered among the beginning rows with larger out degrees.

Observation 2. The density of the high-degree nodes’ neighbor lists is still very sparse. Each node’s neighbor list is a sequence of numbers whose value lies in the range from 0 to $N - 1$. We introduce the metric *Density* to measure the sparsity of a vertex’s neighborhood vector. Density is the number of elements divided by their maximum value. The smaller the density, the sparser the neighborhood list is. Knowing the density is helpful to determine the hashing parameters as for a sparser neighborhood list, it may require a larger hash table. In the directed graph, usually the vertices with higher degree have higher density. For the scale23 graph, the highest density neighborhood is around 0.763. The top 10% high out-degree vertex has density between 0.1 – 0.7. For the friendster graph, the highest density 0.01. The top 10 percent high out-degree vertex has density between 0.001 – 0.01. We can see that as a graph scales, the density of the vertex’s neighbor can be sparser.

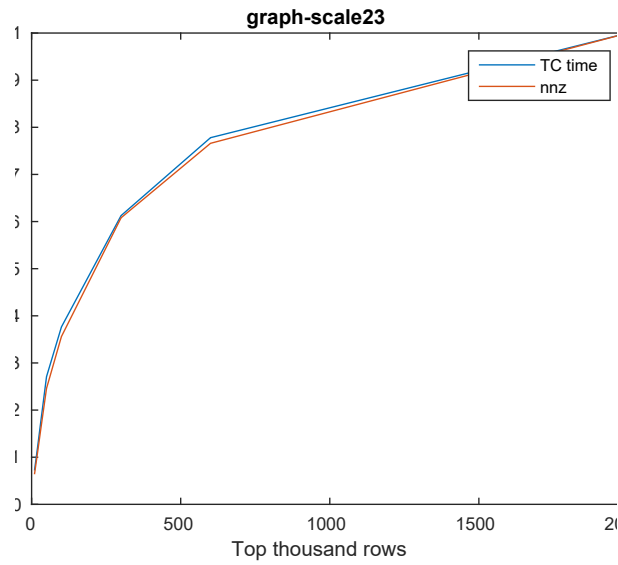


(a) Power-law parameter of 1.

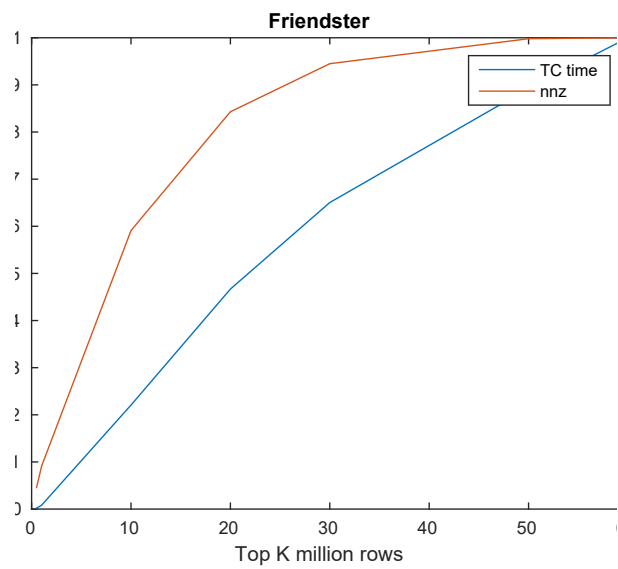


(b) Power-law parameter of 2.

Figure 3.6: Theoretical computation distribution and non-zero element distribution on graphs with different skewness.



(a) Scale23 graph dataset.



(b) friendster dataset.

Figure 3.7: Number of Non-zero element distribution and computation time distribution across rows.

3.2.6 Understanding the Optimizations

Hashing Using hashing method, we didn't see any improvement on the speed. Although the complexity of hashing method is $\min(n1, n2)$. However, this is most effective when there is at least an order of magnitude difference between nA and nB (i.e, $\max(n1, n1) \gg \min(n1, n2)$). But profiling shows that the comparison cases where $\max(n1, n2) > 10 * \min(n1, n2)$ only makes up a small part in all the two-pair list comparisons, according to our observation 1. In other words, majority of time is spent to compare lists $n1 \approx n2$ and both $n1$ and $n2$ are large. Therefore, hashing technique that is helpful to accelerate a dense list versus a sparse list is unable to observe performance improvement.

Exploiting binary search to accelerate merging Similar to the hashing method, the binary search method does not accelerate the computation, because the binary search also requires the size of two sorted lists to be very different in order to have observable speedup.

Exploiting SIMD to accelerate merging This method can be applied regardless of the relative size of the two lists. Therefore, it can be effective to accelerate the CountCommon process of two big lists. And we will show the results in the next section.

Hybrid hashing and merging Whether the Hybrid hashing and merging can be effective highly depends on the density of the neighbor list: whether

the list has elements condensed together that can be compressed. Otherwise the size reduction in the compressed block list is not sufficient. We studied the Friendster dataset and the result suggests the original neighbor list is not friendly for such compressing. As shown in Table 3.1, the base list size after 256bit compression only reduces from 1,806,067,135 elements to 1,690,408,139 elements (about 7% reduction). Moreover, the comparison operation then becomes more expensive when there is a match in the base value. One needs very delicate design in order to draw benefits from such methods.

Table 3.1: Compression savings on Friendster dataset. Its original edge list size is 1,806,067,135.

| | 64bit | 128bit | 256bit |
|-----------------|---------------|---------------|---------------|
| Cmpr. list size | 1,723,833,467 | 1,711,082,186 | 1,690,408,139 |
| Cmpr. Saving | 5% | 6% | 7% |

3.2.7 Performance Results

Hardware. The systems used in our experiments were supported by Pittsburgh Supercomputing Center [55, 63, 64]. The experiment was tested on three types of hardware settings:

- *Single Core*: A single core out of the small multicore system, used to study the scalar performance.
- *Small multi-core*: A small multicore system which has two Intel Haswell (E5-2695 v3) CPUs(sockets) and each CPU has 14 cores.

- Large multi-core: A large shared-memory multicore system from HPE Integrity Superdome X. It has 16 Intel Xeon E7-8880 v4 CPUs(sockets) with 22 cores per CPU socket, 55MB Last Level Cache.

Results. We present the performance results on the above three systems in Table 3.2-3.4.

Single Core: Table 3.2 shows the speedup of our SIMD-based set intersection over a scalar implementation on a single core (Intel Haswell (E5-2695 v3)) using the small dataset. We can see that SIMD can bring about 1.2x to over 3x times speedups over scalar implementation.

Small multi-core: Table 3.3 shows the SIMD speedup on the small multicore system with media-size dataset. Similar to single-core performance, the result shows that our SIMD implementation is about 1.4x to 3.6x speedup on this multicore system.

Large multi-core: Table 3.4 shows the performance on the large shared-memory multicore system with large dataset. Those large datasets are generated using the kronecker graph generator [65]. The kron35 dataset runs over 2-days and unfinished by the given submission time frame. We are unable give the execution time. We estimate the time will be between 60 hours to 100 hours.

Performance comparison with distributed system. Last year graph challenge champion Pearce, et al. [66] presented their triangle counting performance on distributed systems. They used up to 256 nodes where each node

has 24 cores. Their total number of cores is higher than ours. Therefore their overall execution time is shorter. When it comes to performance per-core. The *triangle processed per second per core* they achieved is **1.9MTPS** at highest (for WDC dataset, they counted 9.65T triangles in 808.7s on 256 node where each node has 24 cores). Our implementation can get **3.1MTPS** per core.

Table 3.2: Single-core performance scalar vs. SIMD

| Dataset | V | E | T | Scalar time(s) | SIMD time(s) | SIMD speedup |
|------------------|-----------|------------|---------------|-------------------|-----------------|-----------------|
| cit-HepTh | 27,770 | 352,285 | 1,478,735 | 0.08 | 0.03 | 1.22 |
| cit-Patents | 3,774,768 | 16,518,947 | 7,515,023 | 1.23 | 1.01 | 2.66 |
| flickrEdges | 105,938 | 2,316,948 | 107,987,357 | 1.57 | 0.569 | 2.76 |
| graph500-scale18 | 174,147 | 3,800,348 | 82,287,285 | 3.35 | 1.17 | 2.86 |
| graph500-scale19 | 335,318 | 7,729,675 | 186,288,972 | 8.51 | 2.84 | 3.01 |
| graph500-scale20 | 645,820 | 15,680,861 | 419,349,784 | 21.9 | 7.29 | 3.00 |
| graph500-scale21 | 1,243,072 | 31,731,650 | 935,100,883 | 55.6 | 19.8 | 2.80 |
| graph500-scale22 | 2,393,285 | 64,097,004 | 2,067,392,370 | 142 | 52.9 | 2.68 |

Table 3.3: Small 28-core system performance

| Dataset | V | E | T | Scalar time(s) | SIMD time(s) | SIMD speedup |
|------------------|------------|---------------|----------------|-------------------|-----------------|-----------------|
| Friendster | 65,608,366 | 1,806,067,135 | 4,173,724,142 | 96.7 | 67.1 | 1.44 |
| graph500-scale23 | 4,606,314 | 129,250,705 | 4,549,133,002 | 63 | 17.7 | 3.56 |
| graph500-scale25 | 17,043,780 | 523,467,448 | 21,575,375,802 | 259 | 72 | 3.60 |

Table 3.4: Performance on a HP Superdome X system with 16 sockets, 352 cores

| Dataset | V | E | T | Time(s) | Triangles/ (sec*core) |
|---------|---------------|-------------------|-------------------|---------|--------------------------|
| kron26 | 68,175,120 | 6,281,609,376 | 222,966,186,844 | 202 | 3.1M |
| kron31 | 1,090,801,920 | 33,501,916,672 | 1,380,824,051,328 | 2572 | 1.5M |
| kron35 | 1,380,546,180 | 1,256,845,342,648 | | >50h | |

Chapter 4

Accelerating Sparse Computational Kernels on CPUs: A Fast SIMD Set Intersection Approach

Contents

| | | |
|------------|------------------------------------|-----------|
| 4.1 | Abstract | 70 |
| 4.2 | Introduction | 71 |
| 4.3 | Background and Related Work | 74 |
| 4.3.1 | Scalar Set Intersection Approaches | 76 |
| 4.3.2 | Accelerating Set Intersections | 78 |
| 4.4 | The FESIA Approach | 79 |
| 4.4.1 | Overview | 80 |

| | | |
|------------|--|------------|
| 4.4.2 | Data Structure | 80 |
| 4.4.3 | Intersection Algorithm | 82 |
| 4.4.4 | Theoretical Analysis | 83 |
| 4.5 | Bitmap-Level Intersection | 85 |
| 4.6 | Segment-Level Intersection | 87 |
| 4.6.1 | Runtime Dispatch | 89 |
| 4.6.2 | Specialized vs. General Set Intersection Kernels . . . | 91 |
| 4.6.3 | Implementing Specialized Intersection Kernels with SIMD | 92 |
| 4.7 | Discussion | 96 |
| 4.8 | Experiments | 99 |
| 4.8.1 | Experimental Setup | 100 |
| 4.8.2 | Result of Specialized Intersection Kernels | 101 |
| 4.8.3 | Effect of Varying the Input Size | 103 |
| 4.8.4 | Effect of Varying the Selectivity | 105 |
| 4.8.5 | Performance of Two Sets with Different Sizes | 107 |
| 4.8.6 | Performance on Real-World Datasets | 110 |
| 4.9 | Conclusion | 111 |

4.1 Abstract

Set intersection is an important operation and widely used in both database and graph analytics applications. However, existing state-of-the-art set intersection methods only consider the size of input sets and fail to optimize for the case in which the intersection size is small. In real-world scenarios,

the size of most intersections is usually orders of magnitude smaller than the size of the input sets, e.g., keyword search in databases and common neighbor search in graph analytics. In this chapter, we present FESIA, a new set intersection approach on modern CPUs. The time complexity of our approach is $O(n/\sqrt{w} + r)$, in which w is the SIMD width, and n and r are the size of input sets and intersection size, respectively. The key idea behind FESIA is that it first uses bitmaps to filter out unmatched elements from the input sets, and then selects suitable specialized kernels (i.e., small function blocks) at runtime to compute the final intersection on each pair of bitmap segments. In addition, all data structures in FESIA are designed to take advantage of SIMD instructions provided by vector ISAs with various SIMD widths, including SSE, AVX, and the latest AVX512. Our experiments on both real-world and synthetic datasets show that our intersection method achieves more than an order of magnitude better performance than conventional scalar implementations, and up to 4x better performance than state-of-the-art SIMD implementations.

4.2 Introduction

Set intersection selects the common elements appearing in all input sets, which is a fundamental operation in database applications. For example, given a search query with multiple keywords, a list of documents containing all input keywords can be computed through a set intersection [71]. In addition, set intersection is becoming a critical building block for a wide range of new applications in graph analytics, such as triangle counting [72, 73], neighborhood

Table 4.1: The summary of our approach vs. state-of-the-art set intersection approaches.

| Methods | FESIA | BMiss ^[67] | Gallopings ^[68] | Hiera ^[69] | Fast ^[70] |
|-----------------------|-------------------|-----------------------|-------------------------------------|-----------------------|----------------------|
| Complexity | $n/\sqrt{w} + r$ | $n_1 + n_2$ | $n_1 \log n_2$ | $n_1 + n_2$ | $n/\sqrt{w} + r$ |
| Small Intersect | ✓ | ✓ | | | ✓ |
| SIMD | ✓ | ✓ | ✓ | ✓ | |
| Multicore | ✓ | | | | |
| $n_1 \ll n_2$ | $\min(n_1, n_2)$ | $n_1 + n_2$ | $n_1 \log n_2$ | $n_1 + n_2$ | $n/\sqrt{w} + r$ |
| k -way intersection | $kn/\sqrt{w} + r$ | $n_1 \cdots + n_k$ | $n_1(\log n_2 + \cdots + \log n_k)$ | $n_1 \cdots + n_k$ | $n/\sqrt{w} + kr$ |
| Portable | ✓ | ✓ | | | |

discovery [74], subgraph isomorphism [75], and community detection [76, 77].

For example, the common friends of two people on social networks can be computed through a set intersection as well.

Recent work focusing on accelerating set intersection operations in database systems [67, 69, 68] proposed the use of merge-based approaches, in which the runtime cost usually only depends on the size of the input sets. However, in real-world scenarios, the intersection size is usually dramatically smaller than the sizes of the input sets. For example, 90% of the intersections resulting from search queries in Bing is an order of magnitude smaller than the size of the input sets, and the intersection size of 76% of the queries is even two orders of magnitude smaller than the input [70]. A similar result is also observed in graph analytics [78], in which the size of over 90% of the intersections is smaller than 30% of their input size.

In this chapter, we propose FESIA, a fast and efficient set intersection algorithm targeting at modern CPU architectures. Our key insight is that a

large number of comparisons required by existing merge-based set intersection approaches are redundant. These redundancies result in significant overheads especially when the intersection size is small. To this end, FESIA accelerates set intersections by avoiding these redundant and unnecessary comparisons. Specifically, we take a two-step approach: it builds an auxiliary bitmap data structure for pruning unnecessary comparisons in the first step. Since the earlier pruning may leave some false positive matches, our approach further compares these elements to produce the final intersection in the second step.

Prior work [70] focusing on new data structures for set intersection achieves lower time complexity but does not take advantage of SIMD instructions available on modern processors. In contrast, other work [67, 69, 68] focusing on vectorizing set intersections with SIMD instructions has higher time complexity. FESIA is the first approach that considers both aspects at the same time. It achieves fast and efficient set intersections for two reasons: firstly, the new segmented-bitmap data structure and the course-grained filtering step can make the complexity depend on the intersection size instead of the size of input sets. In summary, the time complexity of our approach is $O(n/\sqrt{w} + r)$ as in [70], in which w indicates the SIMD width, and n and r indicate the size of the input set and the intersection size. Secondly, the data structure and intersection algorithm are designed with SIMD in mind, allowing our approach to exploit more data parallelism and be portable to different SIMD widths. This includes an efficient design of the course-grained filtering implementation with the bitwise operations, combined with our specialized

SIMD intersection functions for small input sizes, which are more efficient than existing vectorization methods [79, 67].

In summary, this chapter makes the following major contributions:

- We present FESIA, a fast and efficient set intersection approach targeting modern CPUs, which leverages the SIMD instructions and achieves better complexity simultaneously.
- We introduce a coarse-grain pruning approach which can be efficiently implemented with the bitwise SIMD operations.
- We design a code specialization mechanism to reduce the cost of fine-grained intersections required after the initial coarse-grained pruning step.
- We describe an implementation of FESIA for two different Intel platforms with SSE, AVX, and AVX512 instructions.
- Our experiments on both real-world and synthetic datasets show that our intersection method achieves more than a order of magnitude better performance than conventional scalar implementations, and up to 4x better performance than state-of-the-art SIMD implementations.

4.3 Background and Related Work

In this section, we describe conventional scalar set intersection approaches and introduce how SIMD instructions are used to accelerate set intersection.

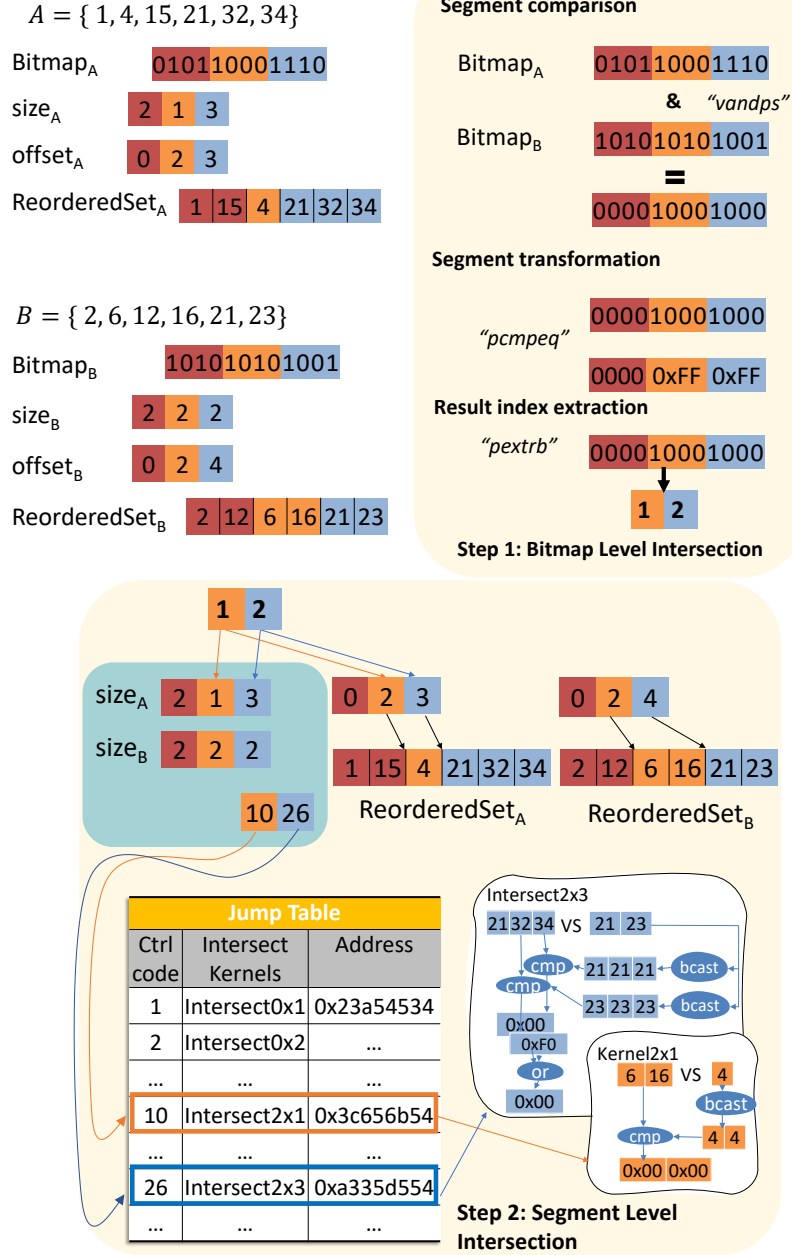


Figure 4.1: Illustrating the data structure and the set intersection algorithm in FESIA. There are two steps in the set intersection: (1) the bitmaps are used to filter out unmatched elements, and (2) a segment-by-segment comparison is conducted to compute the final set intersection using specialized SIMD kernels.

4.3.1 Scalar Set Intersection Approaches

Merge-based set intersection is the most common approach to compute the intersection of two (or more) sorted sets, and an example is shown in C in Listing 4.1. Suppose there are two sets L_1 and L_2 of size n_1 and n_2 respectively, and we use r to denote the number of common elements. The algorithm starts with two pointers that point to the beginning of the two sorted lists. Pointers are iteratively advanced based on the comparison result of their pointed elements. The algorithm finishes when one pointer points to the end of the list. The time complexity of merge-based set intersection is $O(n_1 + n_2)$ for two sets and $O(n_1 + \dots + n_k)$ for k -way set intersection. One limitation of merge-based set intersection is that it cannot be easily extended to exploit multicore parallelism. This is because there exists a loop-carried dependency when pointers are advanced.

```
1 int scalar_merge_intersection(int L1[],
2                             int n1, int L2[], int n2) {
3     int i = 0, j = 0, r = 0;
4     while (i < n1 && j < n2) {
5         if (L1[i] < L2[j]) {
6             i++;
7         } else if (L1[i] > L2[j]) {
8             j++;
9         } else {
10            i++; j++; r++;
11        }
12    }
13    return r;
14 }
```

Listing 4.1: A code example of scalar merge-based set intersection

Hash-based set intersection is another popular approach. It builds a hash table from the elements of one set and then probes the hash table with all elements from the other set. The time complexity of hash-based set intersection is $O(\min(n_1, n_2))$, which makes it the best method when one set is dramatically smaller than the other set. As we will discuss in later sections, the time complexity of our approach is the same as hash-based set intersection when two input sets have dramatically different sizes.

In addition, many other data structures have been proposed to compute set intersection, such as treap [80], skiplist [81], bitmap [82, 70], and adaptive data structures [83].

4.3.2 Accelerating Set Intersections

Data parallelism using SIMD: SIMD instructions have become the status quo on modern CPUs to exploit data parallelism. For example, Intel CPUs have SSE/AVX2 instructions to support 128-bit and 256-bit vector operations. Similarly, ARM processors have NEON instructions, and IBM Power processors have AltiVec instructions. The prevailing SIMD widths on modern processors are 128-bit and 256-bit. More recently, the Intel Skylake architecture introduced AVX512 instructions. Additionally, Intel SSE4.2 introduces the STTNI instruction for string comparisons and it has been used for all-pair comparisons between two vectors in parallel.

The state-of-the-art set intersection methods: Prior work has proposed different ways to accelerate set intersections, which are summarized and compared with our approach in Table 4.1. We now introduce each method in more detail.

BMiss [67] aims at reducing the number of branch mispredictions in merge-based set intersections and demonstrates this on Intel and IBM Power processors. The complexity of this method is the same as other merge-based methods. Note that BMiss performs better when the intersection size is small, in which mispredictions are more likely to occur.

Galloping [84] and its extension SIMDGalloping [68] are approaches based on binary search. Each element from the smaller set is looked up in the larger set through a binary search. This method usually performs better when the size of two input sets is significantly different. Note that the complexity

of k -way intersection with binary search is $n_1(\log n_2 + \dots + \log n_k)$, in which each element from the smallest set (L_1) is used as an anchor point and looked up in all other sets.

Hiera [69] is an approach that leverages the STTNI instruction to accelerate merge-based set intersections. Since it is a merge-based approach, its time complexity is $O(n_1 + n_2)$ on two sets. In addition, a hierarchical data structure is used in Hiera, since STTNI instruction supports only 8-bit and 16-bit data types. One limitation of Hiera is that its effectiveness highly depends on the data distribution. For example, it downgrades to a scalar approach when the elements in input sets are sparse. In addition, it is not portable to processors without the STTNI instruction.

Fast [70] is an approach that leverages bitmaps for fast comparisons and it performs better when the intersection size is small. Its time complexity is $O(n/\sqrt{w} + r)$, in which n , r , and w indicate the size of input sets, the intersection size, and the word size, respectively. This method has a better time complexity compared to other merge-based methods, however, it fails to consider SIMD instructions and may not have competitive performance compared to other merge-based methods with SIMD accelerations.

4.4 The FESIA Approach

In this section, we start with an overview of our FESIA approach. We next explain our data structure and the set intersection algorithm in detail. Finally, we present a theoretical analysis of our approach.

4.4.1 Overview

The intersection process is built upon our *segmented-bitmap* data structure. It first compresses and encodes all elements of a set into a *bitmap* in an offline phase. To exploit data-level parallelism, every s bits from the bitmap are further grouped into a *segment*. When performing online intersection, the bitmap serves as a data structure to quickly filter out the unmatched elements between two sets. The online intersection process therefore consists of two steps: (1) **bitmap intersection**: we compare the segmented bitmaps of two given sets using a **bitwise-AND** operator and output the segments that intersect, and (2) **segment intersection**: given a list of segments whose bitmap intersects with one another (i.e., the result of **bitwise-AND** is not zero), we go through their corresponding lists and compare the relevant elements from each segment to compute the final set intersection. A larger m can eliminate more false-positive intersections on the bitmap, but leads to more comparison time in step 1. The segment size s affects the intersection sizes at step 2. The m and s are chosen to minimize the total time. A theoretical analysis on the choice of m and s will be presented in Section 4.4.4.

We now introduce our data structure and set intersection algorithm in more details.

4.4.2 Data Structure

FESIA is built on our segmented bitmap data structure. This data structure encodes the elements of a set using a bitmap, and groups the bits as well as the

corresponding elements of the bitmap into segments. Specifically, given a set of n elements, its elements are mapped into a bitmap of size m with a universal hash function h . With a properly designed h , all elements can be uniformly distributed in the bitmap. For clarity, we now assume all bitmaps have the same size m and we will relax this assumption with a simple transformation at the end of Section 4.4.3. Every s elements of the bitmap are grouped as a segment. Since the size of the bitmap can be less than the number of elements in the set, more than one element can be mapped to the same location in the bitmap. Therefore, we associate a list to each segment such that elements mapped to this segment are inserted into the list. Note that all elements in a list are always kept in an increasing order.

We now introduce the details of our data structure, as shown in Fig. 4.1. **Bitmap** is a 0/1 binary bit vector of size m . **Size** is an array of size m/s , storing the size of each segment. **ReorderedSet** is an array having all elements of a set but with a different ordering. Intuitively, it is a concatenation of all segments and the elements are sorted in an increasing order within each segment. **offset** is an array of size m/s , storing the starting index of each segment in **ReorderedSet**. We now use the example in Fig. 4.1 to illustrate how the data structure works.

Example 1. Suppose there are two sets: $A = \{1, 4, 15, 21, 32, 34\}$, and $B = \{2, 6, 12, 16, 21, 23\}$. The size of the bitmap is 12, and we use function $f(x) = x \bmod 12$ as our hash function. After mapping all elements into the bitmap, we now have $\text{Bitmap}_A = \{010110001110\}$, $\text{size}_A = \{2, 1, 3\}$, $\text{offset}_A = \{0, 2, 3\}$,

$\text{ReorderedSet}_A = \{1, 15, 4, 21, 32, 34\}$, $\text{Bitmap}_B = \{101010101001\}$, $\text{size}_B = \{2, 2, 2\}$, $\text{offset}_B = \{0, 2, 4\}$, and $\text{ReorderedSet}_B = \{2, 12, 6, 16, 21, 23\}$.

4.4.3 Intersection Algorithm

For the ease of presentation, we discuss the intersection algorithm of two sets in this section. The k-way intersection will be discussed later in Section 4.7.

Our algorithm first compares the associated bitmaps to quickly eliminate unmatched segments between two sets. It streams through the two bitmaps, compares each bit to find the segment pairs that intersect. Next, it intersects the lists associated with those segments with our specialized intersection kernels. A kernel is a specialized intersection function block for a certain size. We now use Example 1 to demonstrate our intersection algorithm. In the first step, we compare the bitmaps of the two sets with a bitwise AND operation to get a list of segments. In Example 1, we compare Bitmap_A and Bitmap_B with a bitwise AND operation, and the result is $\{000010001000\}$. In the second step, we go through each segment to compute the final intersection result, as shown in Algorithm 6. This is because it's possible to have false positives, i.e., a segment whose bitmap intersects with another one but they do not have common elements. In Example 1, we focus on only non-zero segments, i.e., the second and the third segment. Note that the elements mapped to these two segments are $\{4\}$, $\{6, 16\}$ and $\{21, 32, 34\}$, $\{21, 23\}$. Finally, we use the 1-by-2 kernel to compare $\{4\}$ against $\{6, 16\}$, and the 2-by-3 kernel to compare $\{21, 23\}$ against $\{21, 32, 34\}$ to compute the final result of the

intersection. The details of these kernels will be discussed in Section 4.6.

Algorithm 6: The intersection algorithm in FESIA

Input: List L_A , List L_B , Bitmap $_A$, Bitmap $_B$, ReorderedSet $_A$ and ReorderedSet $_B$
Output: the intersection size r

```

1  $r = 0$ 
2  $N = m/s$  // the number of segments
3 for  $i \in 0 \dots N - 1$  do
4   if Bitmap $_{A_i} \&$  Bitmap $_{B_i} \neq 0$  then
5      $r = r + \text{Intersect}(\text{ReorderedSet}_{A_i}, \text{ReorderedSet}_{B_i})$ 
6 return  $r$ 
```

Different bitmap sizes: For any pair of sets, there may exist a pair of sets that have different bitmap sizes. When a bitmap has a different size from the other one, our algorithm requires that the bitmap size m_1 of the larger set can be divided by the bitmap size m_2 of the smaller set. Therefore, the choice of bitmap size in our algorithm is to round the bitmap size to the nearest power of two. When we compare two bitmaps with different sizes, the i th segment from the larger set only needs to compare with the k th segment from the smaller set, in which $k = i \bmod m_2$.

4.4.4 Theoretical Analysis

We now give an analysis of the time complexity of our intersection algorithm. For clarity, we start our analysis with lists L_1 and L_2 with the same size n . We use w to denote the word size and r to denote the intersection size.

Proposition 1. *The time complexity of Algorithm 6 is $O(n/\sqrt{w} + r)$.*

Proof. There are two steps in the algorithm: (1) bitwise **AND** on the two bitmaps, and (2) Computing the intersection on the matched segments. The time spent in bitwise **AND** on the two bitmaps is linearly proportional to m , the size of a bitmap. SIMD allows us to conduct the bitwise operation in parallel. Given the SIMD width w , the time for the bitwise operation is m/w . Let's now analyze the time spent in computing the intersection on the matched segments, which depends on the number of matched segments. There are two types of matches: (1) the true matches, and (2) the false positive matches. The number of true matches equals the intersection size r . A false-positive match happens when there are elements in both lists mapped to the same segment but they are not identical.

More precisely, the number of true matches is

$$E(I_T) = r$$

while the expected number of false positive matches is:

$$\begin{aligned} E(I_{FP}) &= \sum_{\substack{e_1 \in L_1, e_2 \in L_2 \\ e_1 \neq e_2}} P(h(e_1) = h(e_2)) \leq \binom{n}{2} \times \frac{1}{m} \\ &= \frac{n(n-1)}{2m} \end{aligned}$$

Note that $E(I_{FP})$ depends on the choice of m , which affects the time complexity of Algorithm 6. For example, when $m = 1$, the time spent in the second step is $O(n^2)$. When $m = n^2$, the time spent in the second step is $O(1)$. However,

the time spent in the first step now becomes $O(n^2)$. When $m = n\sqrt{w}$, $E(I_{\text{FP}})$ equals to n/\sqrt{w} . Therefore, we have the total number of both true and false positive matches:

$$E(I) = E(I_T) + E(I_{\text{FP}}) = n/\sqrt{w} + r$$

In summary, when $m = n\sqrt{w}$, the time complexity of Algorithm 6 is $O(n/\sqrt{w} + r)$. It achieves the same theoretical bound as in Fast [70]. As we will show in later sections, due to the use of SIMD instructions, our approach also has the best performance in practice. \square

4.5 Bitmap-Level Intersection

In this section, we discuss how to use SIMD instructions to conduct the bitmap-level intersection, which prunes out the unmatched elements. The discussion of segment-level intersection that computes the final intersection will be discussed in Section 4.6.

Given two sets and the corresponding two bitmaps Bitmap_A and Bitmap_B , there are three steps in the bitmap-level intersection to generate a list of segments where one bitmap intersects with the other:

Step 1: Bitwise AND on bitmaps: The key idea is to perform a bitwise AND operation on bitmaps Bitmap_A and Bitmap_B with SIMD instructions. The SIMD instruction that we use is the `vandps` instruction, which operates on w bits at the same time. Note that w is the SIMD width of the processor and

it's a multiple of the segment size s . For example, if $w = 256$ and $s = 8$, a single `vandps` instruction can perform the bitwise AND operation on $256/8 = 32$ segments in parallel.

Step 2: Segment transformation: We next summarize the output of bitwise AND operation from the step above by applying a transformation on each segment, i.e., every s bits in the bitwise AND output. A collection of SIMD comparison instructions (e.g., `pcmpeqw`, `pcmpeqq`, etc) are provided on Intel processors such that comparisons can be performed on every 8, 16, 32, or 64 bits at the same time. Similar instructions are also provided on other processors, such as ARM or IBM Power processors.

With the support of these SIMD instructions, we can use a single instruction to compare s bits with integer 0 for multiple segments at the same time. Note that the SIMD instruction we use depends on the value of s , i.e., different values of s lead to the use of different SIMD instructions. The output for each segment also has s bits. Suppose $s = 16$ and if one segment intersects with the other one, the result is 0xFFFF. Otherwise, the result is 0x0000.

Step 3: Non-zero segment index extraction: We now pick the segments with value 0xFFFF. Given a list of segments, we use the `pextrb` instruction to extract the non-zero segments. The instruction uses 1-bit to represent the output for each segment. Suppose the segment size $s = 16$ and there are two segments 0xFFFF0000. We will get 0x2 (i.e., 10 in the binary format) after applying the instruction on the two segments above.

We next use the `tzcnt` instruction to extract the index of bits that

are set to one. Given an integer, the `tzcnt` instruction returns the number of trailing zeros (i.e., the index of the least-significant 1-bit). Then we set this bit to zero. We iteratively apply the above process until all the ones are extracted.

4.6 Segment-Level Intersection

In the previous section, we discussed how to use bitmap intersection to generate a list of pairs of segments that may have common elements. In this section, we focus on using SIMD instructions to accelerate intersection for each pair of segments.

Suppose there are N segments from the output of bitmap intersection, which are indexed by $i_1 \dots i_N$. The task in this step is to find the intersection size with elements from `ReorderedSetA,i1 ... ReorderedSetA,iN` and `ReorderedSetB,i1 ... ReorderedSetB,iN`. Prior work has proposed different ways to vectorize set intersections with SIMD instructions, but most of them target the general intersection problem in which the input size is sufficiently large. However, in our segment-level intersection, the number of elements in each segment are usually very small. Therefore, a solution to the general intersection problem may suffer from undesirable performance. As a result, we design specialized SIMD intersection kernels for inputs with small sizes. These specialized kernels are more efficient, since they are able to avoid unnecessary computations. For example, if one set has two elements and the other set has four elements, a general SIMD set intersection approach usually assumes that

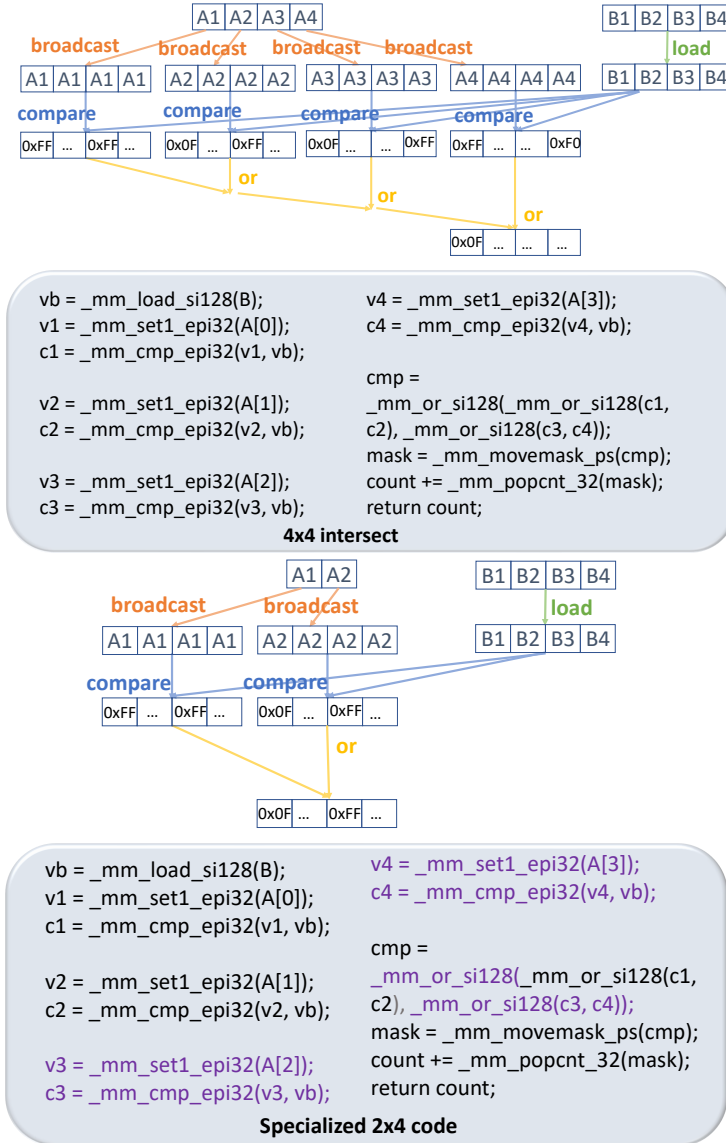


Figure 4.2: Illustrating the difference between a general and a specialized SIMD set intersection kernels. A general kernel is shown on the upper part of this figure, which is implemented with SSE-128 instructions and can be used for any intersection with input size less than 4-by-4. A specialized 2-by-4 kernel is shown on the bottom, which reduces unnecessary computation and memory accesses (highlighted in purple).

both sets have four elements, which introduces unnecessary computations.

In this section, we will first discuss how to dispatch each segment to the corresponding specialized intersection kernel in Section 4.6.1. The implementation of each kernel will be described later in Section 4.6.2.

4.6.1 Runtime Dispatch

To achieve the best performance, we have implemented and compiled set intersection kernels in advance for all possible scenarios. For example, if there are at most \mathcal{S} elements in a segment, we will have $(\mathcal{S} + 1)^2$ different set intersection kernels, i.e., from 0-by-0 up to n-by-n set intersection. Note that some kernels may never be used, e.g., kernel 0-by- i ($0 \leq i \leq n$). With all set intersection kernels, a runtime dispatch mechanism is needed, which allows us to use the correct set intersection kernel given different input segment sizes (i.e., `sizeA,ik` and `sizeB,ik`). We now describe how the runtime dispatch mechanism is implemented with switch statement. Note that the GCC compiler will generate a table in assembly instead of a sequence of if-else statements, which are prohibitively expensive.

```

1 /* the dispatch control code */
2 int ctrl = ((Sa << 3) | Sb);
3 /* jump to a specialized intersection kernel */
4 switch(ctrl & 0x7F){
5 /* Each specialized kernel is a macro */
6     case 0: break; //kernel0x0
7     case 1: break; //kernel0x1
8     case 2: break; //kernel0x2
9     /* case 3 to 62 are omitted .... */
10    case 63: Kernel7x7; break;
11    default: GeneralIntersection(); break;
12 }

```

Listing 4.2: The jump table for specialized kernels

We implement and place all set intersection kernels in memory. Their starting addresses are referenced by a jump table as shown in Algorithm 4.1. The entries of the jump table are managed in a way such that a correct entry can be easily located through a control code, which is computed through simple arithmetic computation given a pair of segment sizes. Given the k th pair of segments A and B from a list of segments, we use S_a and S_b to denote their sizes size_{A,i_k} and size_{B,i_k} . The control code for the jump table is computed by concatenating the two integers S_a and S_b .

We now use a concrete example to show how we compute the control code. Assume that there are 64 different set intersection kernels (from 0-by-0 up to 7-by-7 set intersection) in Listing 4.2. Since the maximum segment size is 7, it's sufficient to use three bits to represent S_a or S_b . As a result,

we concatenate S_a and S_b into one integer through the following statement:
 $\text{ctrl} \leftarrow (S_a \lll \lceil \log_2^S \rceil) | S_b$.

In summary, bits 3 to 5 of the control code indicate size S_a . Similarly, bits 0 to 2 of the control code indicate size S_b .

4.6.2 Specialized vs. General Set Intersection Kernels

The reasons that a specialized set intersection kernel achieves better performance than a general one are twofold: (1) less computations, and (2) more efficient memory access. We illustrate the difference between a general and specialized set intersection kernel in Fig. 4.2. We show a general 4-by-4 intersection kernel on the left-hand side of the figure. The general kernel is implemented with the SSE-128 instructions and can be used for any pair of sets with input size smaller than 4-by-4 (e.g., 2-by-4). However, it is not as efficient as the specialized 2-by-4 intersection kernel, which is shown on the right-hand side of the figure. The specialized kernel on the right-side of Fig. 4.2 is also implemented with the SSE-128 instructions but it avoids unnecessary computations and memory accesses (highlighted in purple in the figure).

In addition, specialized kernels can take more advantage of data reuse. For example, when the set size is larger than the vector length, some elements may be used in multiple comparisons. With a specialized kernel, these elements can be put in registers to avoid redundant memory accesses and minimize the cost of comparisons at the same time.

4.6.3 Implementing Specialized Intersection Kernels with SIMD

We now describe how we implement the specialized intersection kernels with SIMD for different input sizes. We start with a scenario in which the size of both input sets equals to the vector length (i.e., $S_a = S_b = V$). Note that S_a and S_b denote the sizes of the input sets and $V = w/S_e$, in which w is the SIMD word width and S_e is the size of each element in the set.

V -by- V set intersection ($S_a = S_b = V$): We first describe the key idea of implementing a $V \times V$ intersection kernel, which performs a complete all pair comparisons between all V elements of the two input sets. Our SIMD implementation iteratively picks one element from set A and compares it with with all V elements in set B simultaneously. Note that the comparison results of all V elements of set A are combined together into a single vector (e.g., `cmp` as shown in Fig. 4.2).

There are four types of SSE instructions on 32-bit integers in our SIMD implementation, namely, (1) load: `_mm_load_si128`, (2) broadcast: `_mm_set1_epi32`, (3) compare: `_mm_cmp_epi32`, and (4) bitwise OR: `_mm_or_si128`. We now describe the details as follow. First, the V elements of set B are loaded into one vector register v_b . Second, each element of set A is broadcast to a different vector v_i ($i = 1, 2, 3, 4$), and is compared with v_b . The comparison result between v_i and set v_b is stored in vector c_i . Finally, the four comparison results c_1, \dots, c_4 are combined together into one mask register with a bitwise OR instruction.

| | | |
|--|--|--|
| Function: Intersect4by6(l, j) 1 $v_a = \text{_mm_load_si128}(A)$ 2 $v_1 = \text{_mm_set1_epi32}(B_1)$ 3 $c_1 = \text{_mm_cmp_epi32}(v_1, v_a)$ 4 $v_2 = \text{_mm_set1_epi32}(B_2)$ 5 $c_2 = \text{_mm_cmp_epi32}(v_2, v_a)$ 6 $v_3 = \text{_mm_set1_epi32}(B_3)$ 7 $c_3 = \text{_mm_cmp_epi32}(v_3, v_a)$ 8 $v_4 = \text{_mm_set1_epi32}(B_4)$ 9 $c_4 = \text{_mm_cmp_epi32}(v_4, v_a)$ 10 $v_5 = \text{_mm_set1_epi32}(B_5)$ 11 $c_5 = \text{_mm_cmp_epi32}(v_5, v_a)$ 12 $v_6 = \text{_mm_set1_epi32}(B_6)$ 13 $c_6 = \text{_mm_cmp_epi32}(v_6, v_a)$ 14 $t_1 = \text{_mm_or_si128}(c_1, c_2)$ 15 $t_2 = \text{_mm_or_si128}(c_3, c_4)$ 16 $t_3 = \text{_mm_or_si128}(c_5, c_6)$ 17 $\text{cmp} = \text{_mm_or_si128}(t_1, t_2)$ 18 $\text{cmp} = \text{_mm_or_si128}(\text{cmp}, t_3)$ 19 $\text{mask} = \text{_mm_movemask_ps}(\text{cmp})$ 20 $\text{count} += \text{_mm_popcnt_32}(\text{mask})$ 21 return count | Function: Intersect2by7(i, j) 1 $v_b = \text{_mm_load_si128}(B)$ 2 $v_1 = \text{_mm_set1_epi32}(A_1)$ 3 $c_1 = \text{_mm_cmp_epi32}(v_1, v_b)$ 4 $v_2 = \text{_mm_set1_epi32}(A_1)$ 5 $c_2 = \text{_mm_cmp_epi32}(v_2, v_b)$ 6 $\text{cmp} = \text{_mm_or_si128}(c_1, c_2)$ 7 $\text{mask} = \text{_mm_movemask_ps}(\text{cmp})$ 8 $v_b = \text{_mm_load_si128}(B + 4)$ 9 $c_1 = \text{_mm_cmp_epi32}(v_1, v_b)$ 10 $c_2 = \text{_mm_cmp_epi32}(v_2, v_b)$ 11 $\text{cmp} = \text{_mm_or_si128}(c_1, c_2)$ 12 $\text{mask} = \text{_mm_movemask_ps}(\text{cmp})$ 13 $\text{count} += \text{_mm_popcnt_32}(\text{mask})$ 14 return count | Function: Intersect6by6(l, j) 1 $v_b = \text{_mm_load_si128}(B)$ 2 $v_1 = \text{_mm_set1_epi32}(A_1)$ 3 $c_1 = \text{_mm_cmp_epi32}(v_1, v_b)$ 4 $v_2 = \text{_mm_set1_epi32}(A_1)$ 5 $c_2 = \text{_mm_cmp_epi32}(v_2, v_b)$ 6 $v_3 = \text{_mm_set1_epi32}(B_3)$ 7 $c_3 = \text{_mm_cmp_epi32}(v_3, v_b)$ 8 $v_4 = \text{_mm_set1_epi32}(B_4)$ 9 $c_4 = \text{_mm_cmp_epi32}(v_4, v_b)$ 10 $v_5 = \text{_mm_set1_epi32}(B_5)$ 11 $c_5 = \text{_mm_cmp_epi32}(v_5, v_b)$ 12 $v_6 = \text{_mm_set1_epi32}(B_6)$ 13 $c_6 = \text{_mm_cmp_epi32}(v_6, v_b)$ 14 $t_1 = \text{_mm_or_si128}(c_1, c_2)$ 15 $t_2 = \text{_mm_or_si128}(c_3, c_4)$ 16 $t_3 = \text{_mm_or_si128}(c_5, c_6)$ 17 $\text{cmp} = \text{_mm_or_si128}(t_1, t_2)$ 18 $\text{cmp} = \text{_mm_or_si128}(\text{cmp}, t_3)$ 19 $v_b = \text{_mm_load_si128}(B_4)$ 20 $c_5 = \text{_mm_cmp_epi32}(v_5, v_b)$ 21 $c_6 = \text{_mm_cmp_epi32}(v_6, v_b)$ 22 $\text{cmp} = \text{_mm_or_si128}(c_5, c_6)$ 23 $\text{mask} = \text{_mm_movemask_ps}(\text{cmp})$ 24 $\text{count} += \text{_mm_popcnt_32}(\text{mask})$ 25 return count |
|--|--|--|

Figure 4.3: Illustrating the specialized kernels: (1) a 2-by-7 intersection kernel (small-by-large), (2) a 4-by-5 intersection kernel (small-by-large and S_b is slightly larger than V), and (3) a 6-by-6 intersection kernel (large-by-large).

We next describe how we implement the specialized intersection kernels for other input sizes. Without loss of generality, we divide all intersection kernels into three categories based their input sizes: (1) small-by-small set intersection ($S_a \leq S_b \leq V$): the size of both input sets is less than the vector length, (2) small-by-larger set intersection ($S_a \leq V < S_b$): the size of one input size is less than the vector length, and the size of the other input size is larger than the vector length, and (3) large-by-large set intersection ($V < S_a \leq S_b$): the size of both input sets is larger than the vector length.

Small-by-small set intersection ($S_a \leq S_b \leq V$): When S_a and S_b are both less than the vector length V , the implementation of our specialized kernels removes unnecessary operations, as shown in Fig. 4.2. Compared to the general 4-by-4, the specialized 2-by-4 kernel does not need to perform complete 4-by-4

comparisons. Instead, it only compares between the two elements in set A and the four elements in set B , which only takes two broadcasts, two comparisons and one bitwise **OR** instruction. In summary, the number of comparisons and memory access operations in the specialized kernel is only half of that as in the general 4-by-4 kernel.

Small-by-large set intersection ($S_a \leq V < S_b$): The kernels for small-by-small intersections can be used to build the specialized kernels for small-by-large intersections, in which the size of one set is less than the vector length V and the size of the other set is larger than the vector length V ($S_a \leq V \leq S_b$). The key idea is that we first compare all elements in set A with the first V elements of set B .

We now use the 2-by-7 set intersection (as shown in the left side of Fig. 4.3) as an example. We start with comparing the two elements in set A against the first four elements in set B . In particular, two elements A_1 and A_2 are broadcast into vector registers v_1 and v_2 . They are next compared with the first 4 elements in set B in vector register v_b . Line 7-9 combines the comparison results together through a bitwise **OR** operation. We next apply the the same process to compare the two elements in set A with the remaining three elements in set B , by loading the three elements into one vector register all together, as shown at line 10-15. Note that two elements A_1 and A_2 can be reused from registers v_1 and v_2 .

Compared to the general kernel, this specialized kernel reduces computation, has better data reuse and avoids redundant loads. In summary, the

number of broadcasts in the specialized kernel equals to the size of the small set (i.e., $S_a = 2$). The number of loads equals to the size of the large set divided by the SIMD width (i.e., $\lceil S_b/w \rceil = \lceil 7/4 \rceil = 2$). The number of comparisons is $2S_a$, since each element in the small set requires two comparisons with the large set.

However, this specialized kernel may be sub-optimal for some cases due to more comparisons. For example, when the size of set A is 4 and the size of set B is only slightly larger than the vector size V (e.g., $S_b = 5$), the fifth element in set B requires four comparisons against set A . This is because the four elements A_1, \dots, A_4 are in four different vector registers due to broadcast operations. The Intersect4by5 example in the middle of Fig. 4.3 illustrates a better way to implement such scenario. First, set A is loaded into a vector register. Second, elements in set B are broadcast and compared with this register. In total, the number of comparisons is only five instead of eight.

Large-by-large set intersection ($V < S_a \leq S_b$): The implementation of our specialized kernels is built on top of small-by-large and small-by-large kernels, when both S_a and S_b are larger than the vector length V .

Let's now take the 6-by-6 set intersection (as shown in the right side of Fig. 4.3) as an example. We first apply a 4-by-4 intersection to compare the first V elements between set A and set B . There are two scenarios in the second step: (1) $A_4 \leq B_4$: To maximize data reuse, it is more efficient to load all elements from set B into one vector and broadcast all elements from set A . Therefore, we apply a 2-by-6 intersection between the fifth and sixth element

of set A and all six elements of set B . The example in the right side of Fig. 4.3 corresponds to this scenario. (2) $A_4 > B_4$: This is the symmetric scenario. Therefore, it is more efficient to load all elements from set A into one vector and broadcast all elements from set B . Note that we have implemented both kernels above, and use the correct one based on the comparison result between A_4 and B_4 in runtime.

4.7 Discussion

In this section, we first describe how our approach is used for k -way intersection and its time complexity. We next discuss the case for input with dramatically different sizes and how to extend our approach to exploit multicore parallelism and wider vector width.

k -way intersection: Similar to the process of set intersection between two sets, our approach can also leverage the segmented bitmaps to quickly prune the mismatches among k sets. Given k sets L_1, L_2, \dots, L_k , we now describe the two-step set intersection as follows: (1) for each set L_i , there is a corresponding Bitmap_i . In this step, we compare the k bitmaps (i.e., $\text{Bitmap}_1 \dots \text{Bitmap}_k$) using a bitwise **AND** operation to quickly filter out segments whose bitmaps do not have intersection with others. As discussed in Section 4.4, the output is a list of non-zero segments; (2) we next apply the specialized k -way intersection kernels to the list of segments to perform the intersections over their associated elements. Since the expensive k -way intersection kernels are only performed on the matched segments (i.e., output from the first step), the complexity

of the k -way set intersection with our data structure is proportional to the intersection size, instead of the entire input size. The performance advantage is more significant when the final intersection size is small.

We now give a theoretical analysis of the time complexity of our k -way intersection approach. For clarity, we assume the size of each set is n and we use w to denote the word size and r to denote the intersection size.

Proposition 2. *The time complexity of the k -way intersection algorithm is $O(kn/\sqrt{w} + r)$.*

Proof. There are two steps in the algorithm: (1) bitwise **AND** on k bitmaps, and (2) computing the intersection on the matched segments. The analysis for the time spent in each step is similar to what we discussed in Proposition 1. Therefore, the time for the bitwise operation with SIMD instructions is $O(k * m/w)$. The time spent in the second step depends on the number of false positive matches. As in Proposition 1, we have the expected number of false positive matches:

$$E(I_{\text{FP}}) = \sum_{e_1 \in L_1, \dots, e_k \in L_k} P(h(e_1) = \dots = h(e_k)) = \frac{n^k}{m^{k-1}}$$

In summary, when $m = n\sqrt{w}$, we have the total number of both true and false positive matches:

$$E(I) = E(I_T) + E(I_{\text{FP}}) = n/\sqrt{w}^{k-1} + r$$

In summary, the time complexity of the k -way intersection algorithm is

$O(kn/\sqrt{w} + r)$. □

Input with dramatically different sizes: When two sets have dramatically different sizes (i.e., $n_1 \ll n_2$), we adapt our approach to a different strategy such that we can achieve the same time complexity as in a hash-based method, i.e. $O(\min(n_1, n_2)) = O(n_1)$. The key idea is that we go through each element in the smaller set and check the existence of the element in the larger set. Note that if the larger set has the element, then the element must be in segment $v \bmod m_2$, in which m_2 is the size of larger set's bitmap. If the corresponding bit in the bitmap is not set, meaning the element is not in the larger set, all subsequent comparisons are avoided. Otherwise, the element is compared against all the elements mapped to that position.

Multicore parallelism: Our set intersection approach can be easily extended to exploit more parallelism on multicore processors, since there are no cross-iteration dependencies.

The bitmaps of input sets can be partitioned and distributed onto different CPU cores such that each core can independently perform the bitwise AND operation on its partition and use specialized kernels to compute the intersection result on the matched segments. Note that when the two input sets have dramatically different sizes, we only partition and distribute the elements in the small set so that each core can next perform our approach against the bitmap of the large set independently.

Wider vector width: Intel is introducing the AVX512 instructions the Skylake and Cannonlake architecture. As discussed earlier, there is a bitwise AND

operation in the first step, which can have linear speedups with wider SIMD width. In addition, our specialized kernels are also designed to support arbitrary vector length V . However, directly applying our approach with wider SIMD instructions leads to significant performance degradation. This is because increasing segment size s leads to more elements in a segment on average. As the segment size goes up, more specialized intersection kernels for larger sizes are needed. As a result, the complexity of the jump table goes up as well. When the size of the jump table exceeds the instruction cache size, severe performance degradation will happen.

To reduce the number of specialized intersection kernels in the jump table, some specialized intersection are omitted and not implemented. In particular, instead of enumerating intersection kernels for all size pairs, we only implement intersection kernels at some sampled sizes (e.g., the even sizes). For a segment whose size falls in between those sampled sizes, its size is rounded up to the next larger size, meaning we will use a slightly larger specialized kernel for this segment. Although this causes some redundant computations, it significantly reduces the number of intersection kernels in the jump table.

4.8 Experiments

In this section, we study the performance of our approach compared to the state-of-the art set intersection algorithms on both synthetic and real-world datasets.

Table 4.2: Hardware specifications

| | Processor 1 | Processor 2 |
|-----------------|---------------|--------------------|
| Processor Model | Intel E5-2695 | Intel i7-7820X |
| Architecture | Haswell | Skylake |
| Frequency | 2.3 GHz | 4.3 GHz |
| Cores | 28 | 8 |
| L1 cache/core | 32 KB | 32 KB |
| L2 cache/core | 256 KB | 1 MB |
| L3 cache | 35 MB | 11 MB |
| SIMD feature | SSE, AVX2 | SSE, AVX2, AVX-512 |

4.8.1 Experimental Setup

Platforms: We implement our algorithms on platforms with SSE/AVX/AVX-512 instructions and compare with state-of-the-art set intersection methods. We use two Intel platforms in our evaluation: (1) Intel E5-2695, which is a Haswell architecture and supports SSE(128-bit) and AVX(256-bit) instructions, and (2) Intel i7-7820X, which is a Skylake architecture and has the latest AVX-512 instruction support. All SSE/AVX experiments are performed on the Haswell processor, and all AVX-512 experiments are performed on the Skylake processor. The detailed hardware specifications are given in Table 4.2.

Datasets: We perform the evaluation on both synthetic and real-world datasets, focusing on how the following three key factors: (1) input size (n), (2) selectivity (r/n), and (3) skew in the input sizes (n_1/n_2).

Methods: We study and compare the performance of our approach with the following state-of-the-art implementations: (1) Scalar: This is an optimized scalar merge implementation similar to the code in Listing 4.1, but it

replaces the expensive if-else branch statements with conditional moves, (2) Shuffling [79]: This is a SIMD implementation for set intersection and it is similar to what is presented in Fig. 4.2. It uses the SIMD `shuffling` instruction to perform all pair-wise comparisons between two input vectors by creating all variations of one vector. (3) scalarGallop [84]: This is a binary-search based intersection method. (4) SIMDGallop [68]: A SIMD optimized version of scalarGallop, and (5) BMiss [67]: A merge-based intersection approach with SIMD and optimizations on reducing branch mispredictions. Note that the data structure is built offline. The built time is not included in our experiments.

4.8.2 Result of Specialized Intersection Kernels

As discussed in Section 4.6, our system adopts different strategies and generates specialized kernels for intersections with different sizes. The specialized kernels can take advantage of SIMD instructions, which perform fewer memory accesses, shuffle and comparison computations.

We now study the performance of our specialized intersection kernels compared to the generalized SIMD implementation. Fig. 4.4 shows the result of SSE kernels. We generated kernel sizes from 1-by-1 up to 7-by-7, which is twice larger than the SSE SIMD width. We can observe that our specialized kernels are up to 70% faster than the general SIMD intersection implementation. Similarly, Fig. 4.5 shows the result of AVX kernels. The kernel size goes up to 15-by-15 and the specialized kernels are faster than the general AVX

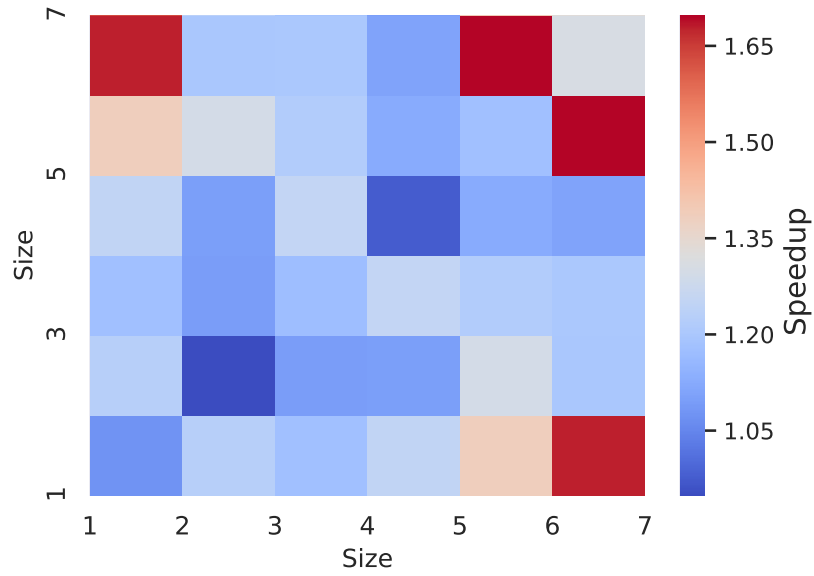


Figure 4.4: Performance of SSE kernels

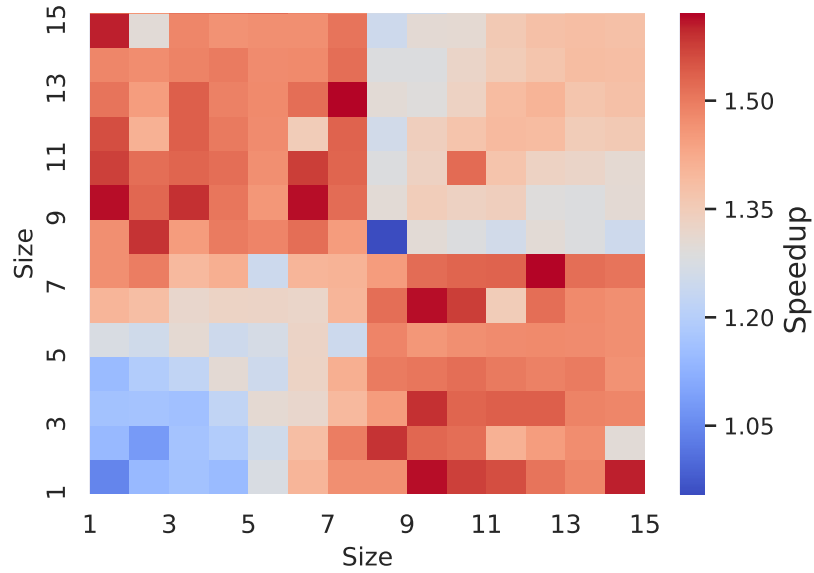


Figure 4.5: Performance of AVX kernels

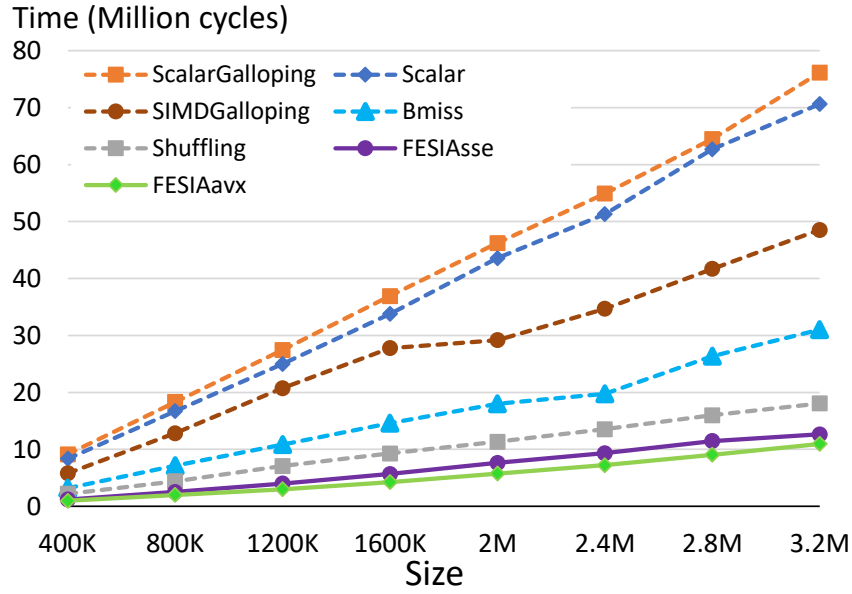
intersection implementation in all scenarios. The performance advantage is even more significant when the size of one set is much larger than the other

set.

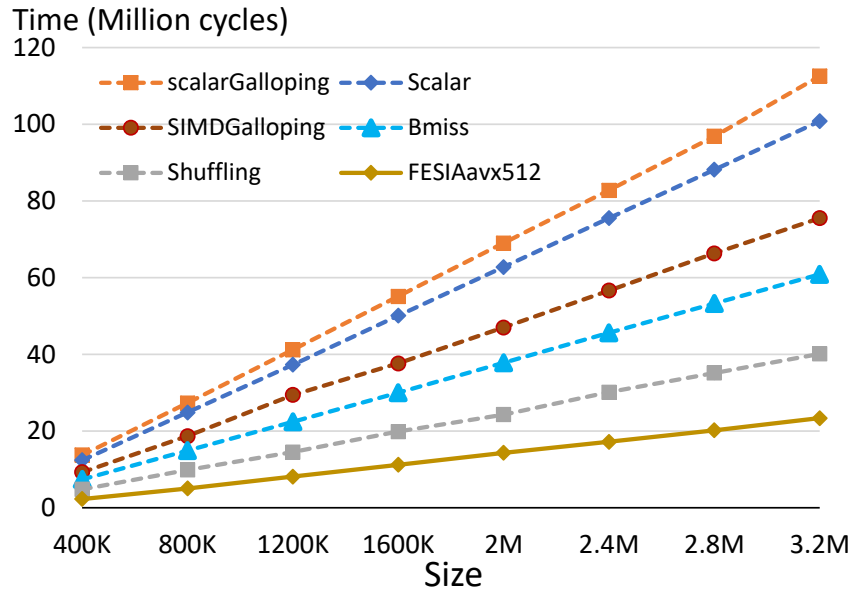
4.8.3 Effect of Varying the Input Size

We next study the performance of different intersection approaches with a varying input size. In this experiment, we evaluate each intersection method with two synthetic sets. In addition, we make the size of input sets identical and their intersection size be 1% of the input size. We vary the number of elements in input from 400K to 3.2M. Note that our approach is implemented on two different Intel processors with three different SIMD instruction sets: (1) SSE, (2)AVX, and (3)AVX-512. The performance of our SSE, AVX implementation is measured on an Intel Haswell architecture, which is reported in Fig. 4.6a. The performance of our AVX-512 implementation is measured on an Intel Skylake architecture, which is reported in Fig. 4.6b.

In Fig. 4.6, the y-axis shows the CPU time in million cycles (i.e., the lower, the better). We can observe that the relative performance of these methods remains consistent as we increase the input size. On the Haswell architecture, our SSE and AVX implementation can achieve up to 7.6x speedup compared to scalar methods, and 1.4x-3.5x speedup compared to other methods with SIMD. On the Skylake architecture, our AVX-512 implementation can achieve 2-4x speedup compared to other SIMD methods. On both architectures, Scalar and ScalarGalloping are the slowest, since scalar implementations cannot leverage data-level parallelism. SIMDGalloping performs poorly as well. This is because it is based on binary search, which has higher



(a) Performance comparison on Intel Haswell



(b) Performance comparison on Intel Skylake

Figure 4.6: Performance comparison with a varying input size

complexity when two input sets have similar size.

4.8.4 Effect of Varying the Selectivity

Selectivity is the metric that describes how large the intersection size is relative to the input size. It is defined as the ratio of the intersection size divided by the input size (i.e., r/n). We now study the performance of different intersection approaches when varying the selectivity. In this experiment, we fix the size of the two input sets to one million and report the relative speedup of different approaches to the Scalar intersection method in Fig. 4.7 and Fig. 4.8.

Fig. 4.7 shows the performance of each approach on the Haswell architecture using SSE and AVX instructions. We observe that our approach can achieve up to 7.6x speedups compared to state-of-the-art Scalar intersection methods, and 1.8x speedups compared to state-of-the-art SIMD intersection methods. Fig. 4.8 shows the performance of each approach on the Skylake architecture using AVX-512 instructions. We observe that our approach can achieve up to 6x speedups compared to state-of-the-art Scalar intersection methods, and 1.4-3x speedups compared to state-of-the-art SIMD intersection methods.

In addition, we see that the our method’s speedup is higher when the selectivity becomes lower. Note that in most real-world scenarios, the intersection size is usually less than 10% of the input set (i.e., selectivity is less than 0.1).

K-way intersection: We also study the performance of each method on

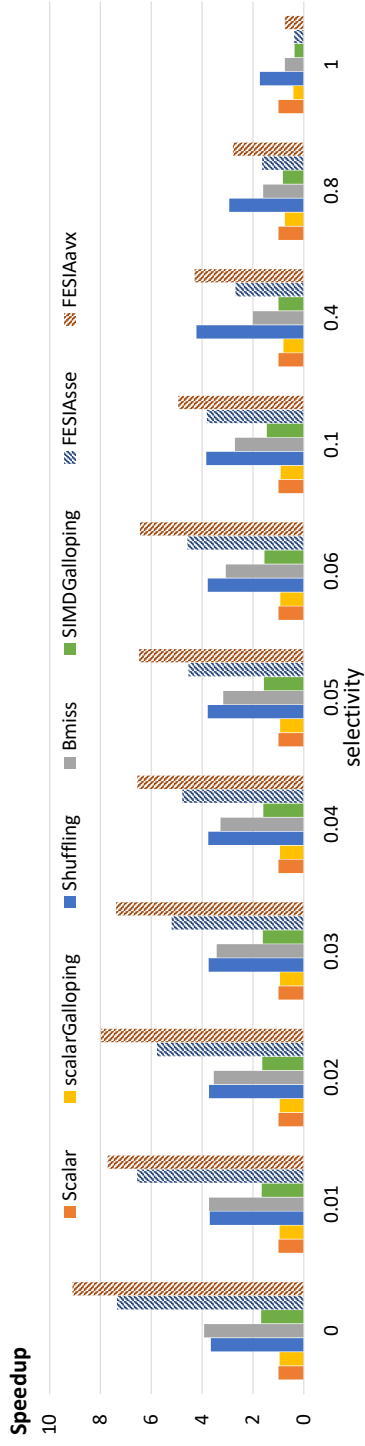


Figure 4.7: Performance on different selectivity

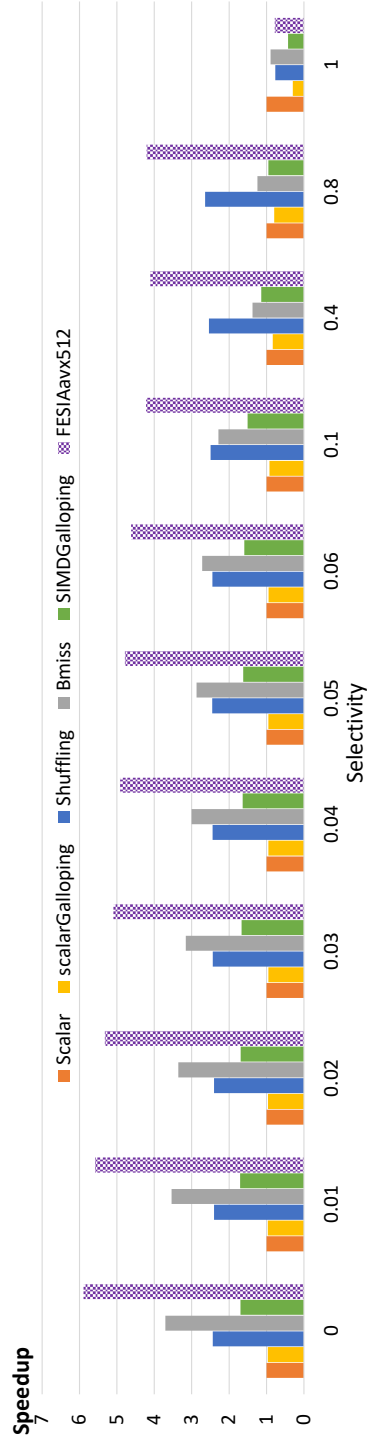


Figure 4.8: Performance on different selectivity (AVX-512)

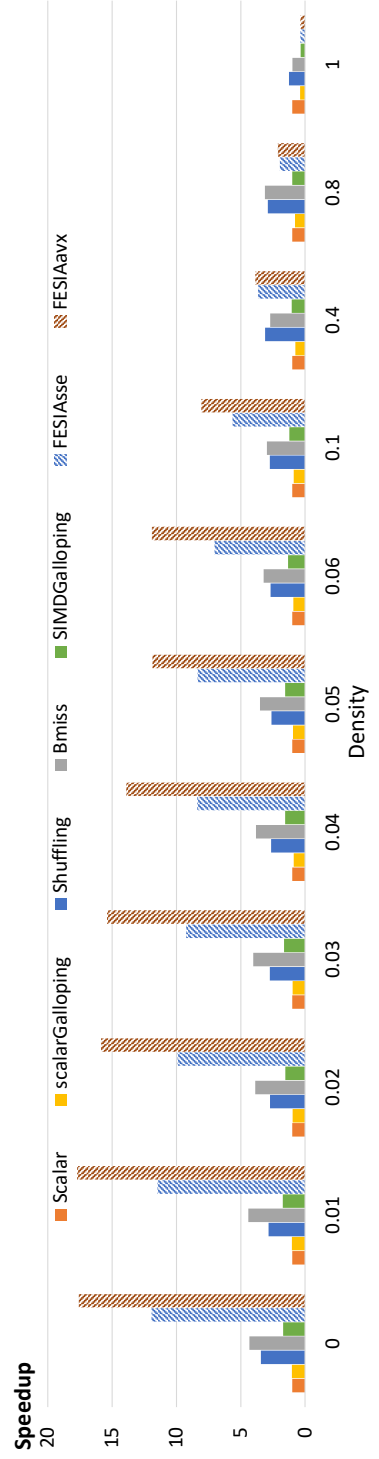


Figure 4.9: Performance of three-way intersection

three-way intersection. We fix the input size to one million as in the experiment above, and report the result in Fig. 4.9. The y-axis is the relative speedup to the Scalar method. The x-axis shows the set density, which describes how clustered the elements are distributed among a given range. For example, elements in a dense set are randomly drew from a small range, while a sparse set have elements drew from a larger range. Density can affect the intersection size. For example, two dense sets are more likely to have common elements. When $k = 3$, the selectivity is proportional to the third power of set density.

We can observe that FESIA can achieve up to 17.8x speedups compared to scalar intersection methods on 3-way intersection, and up to 4.8x speedups compared to state-of-the-art SIMD set intersection approaches. The speedup is higher when the density is lower. When the density is zero, the maximum speedup achieved in 3-way intersection is higher than that in 2-way intersection, which shows the speedup of our approach is more prominent with a larger k . This is because multi-way comparisons are more expensive for larger k . However, our approach can avoid the cost of unnecessary multi-way comparisons through cheap bitmap intersections.

4.8.5 Performance of Two Sets with Different Sizes

We now study the performance of each approach when the two input sets have different sizes. In this experiment, we fix the size of the larger input size to one million and the selectivity to 0.1. The x-axis is the skew, i.e., the ratio of the smaller set size to the larger set size (n_1/n_2). The y-axis is the relative

Speedups on skewness

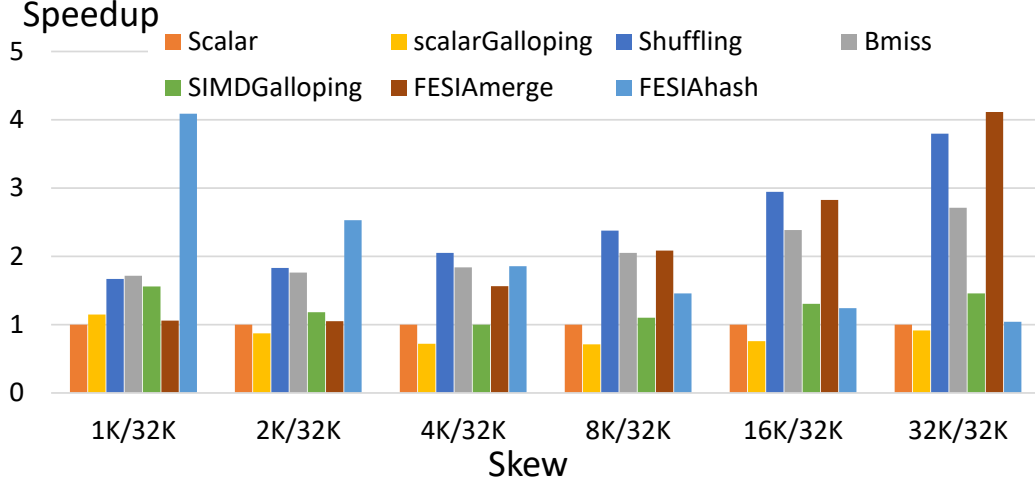


Figure 4.10: Performance comparison on varying skew

speedup to the Scalar method. Note that our approach can adapt to different strategies depending on the value of skew. We use $\text{FESIA}_{\text{merge}}$ to denote the strategy we use when the input has a similar size and $\text{FESIA}_{\text{hash}}$ to denote the strategy we use when the input has dramatically different sizes. We report the performance of both strategies in Fig. 4.10.

Theoretically, when the skew is small ($n_1 \ll n_2$), the hash-based method has the lowest average complexity and the merge based method has the highest average complexity. The complexity of the binary search method is between the two above. When the skew is large ($n_1 \sim n_2$), the average complexity of the merge method is comparable with the hash-based method, and the binary search method has the highest complexity.

As shown in Fig. 4.10, the actual performance of the intersection methods can match the theoretical analysis. When the skew is small, our method

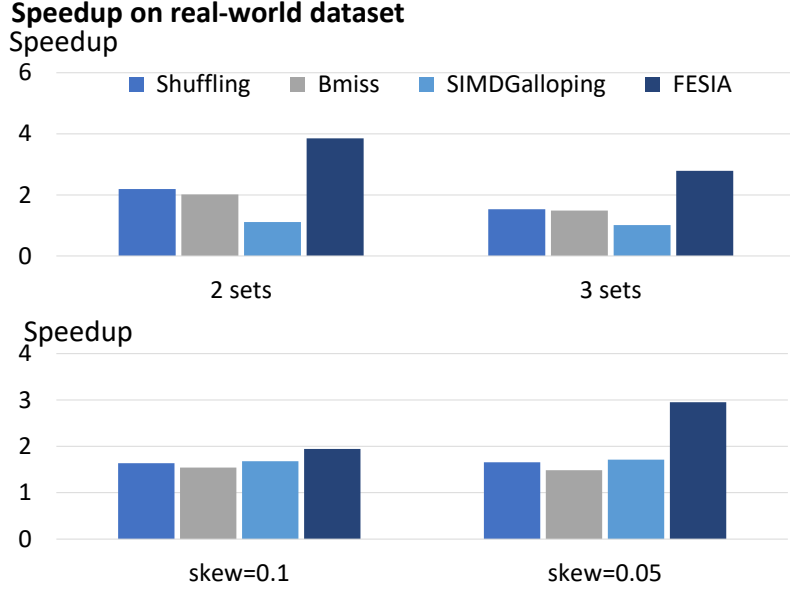


Figure 4.11: Results on the database query task

based on hash (i.e., $\text{FESIA}_{\text{hash}}$) has the best performance, which is 2-3x faster than the binary-search based method SIMDGallop, while SIMDGallop is faster than the two SIMD merge-based methods (Shuffling and BMiss). As the skew goes up to more than $1/4$, $\text{FESIA}_{\text{merge}}$ starts to outperform $\text{FESIA}_{\text{hash}}$ and it achieves the best performance among all approaches. For other methods, the SIMD merge-based method can outperform methods based on binary search.

In summary, FESIA can adapt to different strategies given different skew in input, and achieves better performance than other approaches. All other approaches only perform well when the skew is either small or large.

Table 4.3: The details of each graph dataset

| Dataset | # of nodes | # of edges |
|-------------|------------|------------|
| Patents | 3,774,768 | 16,518,948 |
| HepPh | 34,546 | 421,578 |
| LiveJournal | 3,997,962 | 34,681,189 |

4.8.6 Performance on Real-World Datasets

We next study the performance of each intersection approach on two real-world tasks: (1) a database query task, and (2) a triangle counting task in graph analytics.

The database query task: In Fig. 4.11, we report the result of each approach on a real-life dataset called WebDocs from the Frequent Itemset Mining Dataset Repository [85]. The WebDocs dataset is a web crawl dataset built from a collection of web HTML documents. The whole collection has about 1.7 million documents with 5,267,656 distinct items.

To simulate the low selectivity of real-world queries, we generate random queries from the dataset and keep the set intersection size below 20% of the input size. We show the result of set intersection with two input sets and three input sets on the top of Fig. 4.11. We observe that our approach achieves close to 4x speedup compared to the Scalar method, 2x speedup compared to the SIMD Shuffling, and 3.8x speedup compared to SIMDGallop. The bottom of Fig. 4.11 shows the performance of each approach when the sizes of input sets are skewed. Overall, we observe that our approach has an up to 3x speedup compared to other approaches.

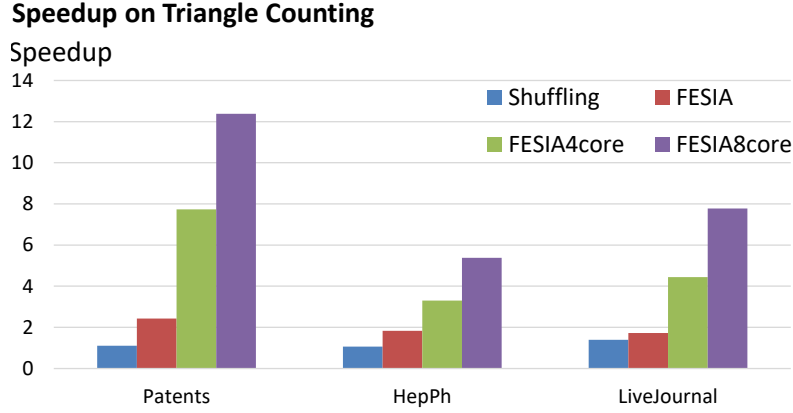


Figure 4.12: Results on the triangle counting task

The triangle counting task: We now study the performance of each approach on the triangle counting task with three graph analytics datasets. The three datasets are from the Stanford Large Network Dataset Collection [86] and we report the details of each dataset in Table 4.3. In Fig. 4.12, we can observe that our approach can achieve up to 12x speedup compared to the Scalar method and up to 1.7x speedup compared to the SIMD Shuffling approach. In addition, the speedup can scale linearly with the number of CPU cores.

4.9 Conclusion

In this chapter, we presented FESIA, an efficient SIMD-vectorized approach for set intersection on modern CPUs. In many real-world tasks such as database queries and graph analytics, the intersection size is usually orders of magnitude smaller than the size of the input sets. FESIA leverages this observation by adopting a bitmap-based approach to efficiently prune necessary compar-

isons and refine a list of smaller segments with intersecting elements. These segments are later processed by specialized SIMD kernels for final intersection. Experiments on both real-world and synthetic datasets show that our approach can achieve an order of magnitude speedup compared to scalar set intersection methods and be up to 4x faster than state-of-the-art SIMD implementations.

Chapter 5

Accelerating Sparse Computational Kernels on GPUs

Contents

| | | |
|------------|--|------------|
| 5.1 | Abstract | 114 |
| 5.2 | Introduction | 115 |
| 5.3 | Background and Related Work | 118 |
| 5.3.1 | GPUs | 118 |
| 5.3.2 | SpGEMM and Masked-SpGEMM Problem | 121 |
| 5.3.3 | Sparse Data Structure | 122 |
| 5.3.4 | SpGEMM Computations | 123 |
| 5.4 | The Family of SpGEMM Algorithms | 124 |
| 5.4.1 | Matrix-Vector SpGEMM | 125 |
| 5.4.2 | Inner-Product-Based SpGEMM | 127 |

| | | |
|------------|--|------------|
| 5.4.3 | Outer-Product-Based SpGEMM | 129 |
| 5.5 | Join-Based SpGEMM | 130 |
| 5.5.1 | The Baseline Join-Based SpGEMM | 130 |
| 5.5.2 | The Data-Parallel Join-Based SpGEMM | 131 |
| 5.5.3 | Join Using Wider Data Types | 133 |
| 5.6 | Join-Based SpGEMM Using Hash Methods . . . | 137 |
| 5.7 | Online Scheduling | 139 |
| 5.8 | Experiments | 142 |
| 5.8.1 | Experimental Setups | 142 |
| 5.8.2 | Performance of GPU Joins with Very Sparse Matrices | 145 |
| 5.8.3 | Performance of Join-Based SpGEMM | 146 |
| 5.8.4 | Comparison on Different Densities | 147 |
| 5.8.5 | Performance on the Hash-Based Join Kernels | 149 |
| 5.8.6 | Performance on Real-World Dataset | 153 |
| 5.8.7 | Performance on Real-World Graph Datasets with Skewed Distribution | 156 |

5.1 Abstract

Accelerating applications that involve sparse and irregular computational pattern has received increasing attention in recent years. Sparse computational kernels are appearing in a variety of different domains from social sciences to machine learning. Data in these domains can have drastically distinct sparsity characteristics. This may lead to different algorithm/implementation choices. Meanwhile, with the advent of processor technologies, modern processors have

more computing power and higher memory bandwidth. But sparse computational kernels tend to perform poorly on modern processors due to irregular computational patterns and random memory accesses. There is a lack of understanding on how the sparse problem should be accelerated on modern processors with massize scale of parallelism. In this chapter, we investigate the performance of parallel sparse-Matrix-Multiplication(SpGEMM) implementations on GPUs. We focus on the (masked-)sparse-matrix-matrix multiplication (SpGEMM) used in many social network analytics tasks. We demonstrate how the sparse matrix computation can be broken down into a primitive set of join/union operations, based on which we propose several optimizations for join-based SpGEMM implementations. We perform experimental and theoretic analysis with both synthetic and real-world datasets. Finally, we propose an online scheduling algorithm that performs a light-weight calculation online to adaptively pick out the right implementation for iterative SpGEMM problems.

5.2 Introduction

Sparse computations are at the core of a variety of domains. They have been employed in a broad range of tasks, such as graph analytics [13], neural networks compressing [14], genome sequencing [15], recommendation systems [16]. Many applications in the above domains work with irregular and nonuniform data that can be represented and structured as sparse matrices. Meanwhile the percentage of non-zero elements in these application can vary drastically,

ranging from $10^{-6}\%$ to 50% depending on the problem domain.

With the growing popularity of sparse computations in various domains, there is a growing effort to optimize and accelerate sparse computations on modern processors. For example, there are a growing number of graph analytics systems that are proposed recently to tackle social network-related sparse computations [87, 88, 89, 90, 91, 92, 93]. A common feature in these frameworks is that they accelerate high-level sparse problems by providing the interface to a set of primitive sparse operations. At the heart of these frameworks is the primitive sparse operation implementations such as sparse-matrix-vector computation, generalized sparse-matrix-matrix-multiplication. For example, graphBLAS defined a specification that can formulate many sparse computations as sparse matrix computations over semirings. Such specification simplifies the sparse problem and provides a unified interface that separates the high-level problem and the design and optimization of underlying implementations. Among them, sparse-matrix-times-sparse matrix (SpGEMM) and masked-sparse-matrix-times-sparse-matrix are two important operators.

One challenge in optimizing sparse computations is that they are hard to attain high performance on modern processors. Unlike their counterpart in dense and regular applications, sparse applications have locally varying non-zero (nnz) patterns, leading to unpredictable control flows and unbalanced workloads which are detrimental to performance. In addition, sparse applications have irregular compute patterns that are inherently sequential, together with random and uncoalesced memory accesses. These issues can result in low

occupancy as well as low resource utilization on GPUs. Meanwhile, modern GPUs are usually designed to attain peak performance when high throughputs and parallelism can be achieved.

On the other hand, there is a lack of understanding of the optimizations for the sparse matrix-multiplication problems. It still remains an open question as to which algorithm or data structure best fits sparse data. One reason for this is due to the vast problem domains and the data characteristics (e.g., density, distribution) which can vary drastically in different domains. Additionally, there are a large number of different methods to process and deal with sparse data, including different algorithms, parallelism schemes, etc. This broad design space naturally leads to questions such as which algorithm is best suited for sparse data, how the parallelism schemes should adapt to the algorithms, and how we should make cheap online decisions for iterative SpGEMM, etc

In this work, we investigate the optimizations of sparse computations on GPU platforms. Existing work has proposed different optimizations on accelerating sparse computation on general platforms such as GPU, CPU, or ASIC designs [94, 95, 96, 97]. However, they are mainly focused on designing new data structures to resolve the load unbalancing issues, or designing new hardware to improve the data flow. In this work, we take a different perspective. We focus on the algorithm choices. We investigate and compare the different SpGEMM implementations on GPU. We explore how different algorithms such as join, merge, as well as different implementations can affect

performance.

In this work, we focus on the sparse-matrix-times-sparse-matrix (identical matrix) problem, The contributions of this chapter include:

- We investigate different methods to compute sparse matrix multiplication, including union-based methods, and intersection-based methods
- We propose and implement the data-parallel join-based SpGEMM methods on GPU
- We propose and implement the hash-based SpGEMM join methods on GPU
- We propose a sparsity-aware scheduler that is capable of choosing the best method according to the data distribution for iterative SpGEMM computations

5.3 Background and Related Work

5.3.1 GPUs.

Modern GPUs relies on largescale multithreading to attain high computational throughput and hide memory access time. A GPU contains many “streaming multiprocessors” (SMs) with up to hundreds of arithmetic logic units (ALUs). A SM is a “multiprocessor” that contains many cuda cores while each cuda core is an execute unit for integer and float numbers of one thread. Each

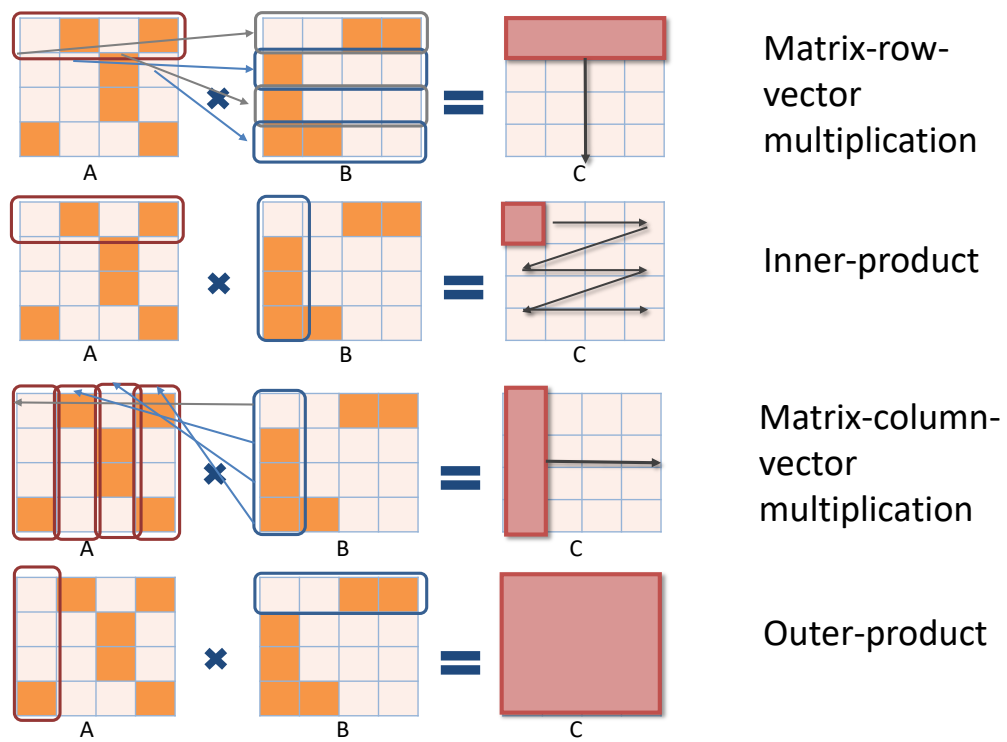


Figure 5.1: The family of SpGEMM algorithms



| | Matrix-row-vector multiplication | Inner-product | Outer-product |
|---|-------------------------------------|---------------|---------------|
| tall and thin (n rows, k nonzeros/row)  | $O(nnk)$ | $O(nnk)$ | $O(nnk)$ |
| short and fat (n columns, k nonzeros/col)  | $O(nkk)$ | $O(nkk)$ | $O(nkk)$ |
| Skewed distribution (βx^α) | $< nz(\alpha)$ | $< O(n^3)$ | $< O(n^3)$ |

Figure 5.2: Complexity comparison on different sparse matrices. N denotes the maximum dimension of the sparse matrix (i.e., maximum nonzero value). k denotes the average nonzeros per row/column — for tall-skinny matrix, k is the average nonzeros per row. If the matrix is short and fat, k is average nonzeros per column.

SM contains hierarchical memories, including shared memory for fast data interchange between threads, and L1, texture, constant caches. For example, the latest Nvidia Turing architecture has 72 SMs. Each Turing SM includes 4 warp-scheduler units, each SM provides 64 FP32 cores, 64 INT32 cores.

GPU programs are called kernels, which run a large number of threads in parallel in a single-program, multiple-data (SPMD) fashion. The threads are arranged in hierarchies: Each kernel is split into a **grid** of threads called thread **blocks** or concurrent thread arrays (CTA). A CTA is a basic workload unit assigned to an SM in a GPU. Threads in the same block have access to shared memory and their execution can be synchronized. Threads in a CTA are sub-grouped into **warps**, the smallest execution unit sharing the same program counter, which contains 32 threads. These 32 threads execute

the same instruction on each clock cycle in lockstep. For a program to achieve high performance on GPU, a good implementation needs to consider coalesced memory access, because the memory bandwidth can be maximized only when threads are accessing consecutive memory addresses. Secondly, divergence is another factor that can degrade performance. Divergence can break the control flow, introducing extra branches and scheduling overheads.

5.3.2 SpGEMM and Masked-SpGEMM Problem

SpGEMM computes the problem of:

$$C_{ij} = \sum_k A_{ik} \cdot B_{kj} \quad (5.1)$$

The computation involves multiplications between every row/column pair of matrix A and matrix B . While masked-SpGEMM computes the problem of:

$$C_{ij} = M_{ij} \sum_k A_{ik} \cdot B_{kj} \quad (5.2)$$

Different from general SpGEMM, masked-SpGEMM applies a matrix M on top of the $A \times B$ result. Elements in the $A \times B$ result matrix will be filtered out if their corresponding positions are zeros in M . In other words, the computation of masked-SpGEMM only involves multiplications between a fraction of row/column pairs between A and B . There are many real-world problems that can be formulated as SpGEMM or masked-SpGEMM problems. For example, triangle counting is a fundamental operator widely used in many graph

analytic tasks in social network applications [98]. The number of triangles can be an approximation of the closeness between neighbors. The algorithm to compute triangle counting can be formulated as a masked-SpGEMM problem. Suppose A is the adjacency matrix of the given graph, the i -th row of A has all the edges adjacent to node i , and the j -th column of A^T contains all the edges incident to node j . The multiplication between the i -th row of A and j -th column of A^T generates all the “wedges” between node i and node j . The subsequent masking operation will filter out the wedges that do not have closing edges. Therefore the total number of triangles between every node pairs can be obtained by computing $A \cdot A^T$. K -truss is another computation that is widely used in graph analytics. K -truss finds out a subgraph in which each node is connected to at least K other nodes. K -truss is computed by iteratively applying triangle counting procedure — each iteration of triangle counting outputs the nodes that are part of a triangle in the input graph and these nodes will be the new input graph for the next iteration.

5.3.3 Sparse Data Structure

Existing literature has proposed different schemes to store sparse data. Among them, compressed sparse row (CSR) format is one of the most popular storage schemes. CSR format stores the column indices and values of nonzeros by row orders in the sparse matrix. It represents the sparse matrix using three arrays: an array storing the (accumulative) length of each row (*col_index*), one array that stores the nonzero column indices (*col_index*), and the third array

that stores the nonzero values (*val*). CSR format requires $n + 2\mathbf{nnz}$ memory for storage, where n is the number of rows and \mathbf{nnz} is the number of nonzeros of the sparse matrix. Similarly, if the matrix is stored in column-major order — when successive elements in the same column are contiguous in memory — it is called compressed sparse column (CSC) format.

5.3.4 SpGEMM Computations

Researchers have proposed a variety of optimizations for sparse computations over the past years. The early works are mainly focused on accelerating sparse matrix-vector multiplication problem (SpMV) [99, 100, 101, 102, 103, 104, 105]. It has been investigated that there can be different computations methods for SpMV problems. For example, previous work have proposed the *push-or-pull* optimizations for SpMV computation. “Push” and “pull” are methods that compute the matrix-vector product in two different orderings, and they can result in different performance according to the input sparsity. Existing literature has investigated how push and pull methods can affect the performance on different platforms [106, 107, 108].

Recently, more attention has been given to SpMM (sparse-matrix-times-dense matrix) and SpGEMM computations [109, 110, 111, 112, 113]. Many literatures are using merge-based methods to compute SpGEMM [114, 115, 116, 117, 118]. The merge-based method iterates the sparse matrix in rows. It takes one row in matrix A (C_i) and multiplies with the matrix B . This generates the result that corresponds to one row in matrix C (C_i). Specifically,

the sparse row-vector-matrix-multiplication is performed by iterating through each element in row i , and multiplies this element with the corresponding row in B . The results will be merged together. This merge-based method is good for a sparse matrix that is tall and skinny.

Notation nnz is used to denote the number of non-zero elements in the sparse matrix. Lowercase n is used to denote the row dimension. The subscript number represents the row of the matrix (e.g, A_i represents the i -th row of matrix A). And A_i^T denotes the i -th row of the transpose matrix A^T , which is also the column vector of A .

5.4 The Family of SpGEMM Algorithms

In this section, we will discuss in detail the three ways to compute SpGEMM as illustrated in Figure 5.1. The complexity of SpGEMM method depends on the number of non-zero elements of the sparse matrices. Note that in this work, we primarily focus on the SpGEMM problem with two identical sparse matrices, i.e., $A = B$. In other words, the SpGEMM problem we are interested is $A \cdot A^T$. The subsequent complexity analysis are all for this problem.

The distribution of the non-zero elements in the sparse matrix is an important factor that can affect the amount of computation performed. For example, the non-zero elements could be centered among a few columns. At other times, some graph datasets may have unevenly-distributed elements where the non-zero elements are centered around some hub nodes. We categorize the

sparse matrix into three cases: the *tall-and-thin* sparse matrices are matrices whose row dimension is much larger than the column dimension (i.e., the average number of nonzeros per row); the *short-and-fat* sparse matrices are matrices whose column dimensions are much larger than the rows. In both tall-and-thin and short-and-fat sparse matrices, the nonzeros are uniformly distributed. The third case refers to matrices that nonzeros are skewedly distributed (and we use the power-law distribution to simulate the skewed-distribution, because it has been widely acknowledged that many skewedly-distributed graph datasets usually follow such distribution [119, 120]). And the later analysis will target these cases separately.

5.4.1 Matrix-Vector SpGEMM

The first way to compute SpGEMM is based on vector-matrix multiplication. This method breaks down the matrix multiplication into a sequence of *row-vector-matrix-multiplication* between a row of matrix A and the matrix B . The row-vector-matrix-multiplication generates the result corresponding to one row in matrix C .

Specifically, row A_i multiplies matrix B will generate row C_i . The computation of the vector-matrix multiplication leverages the merge operation: to compute sparse-vector-sparse-matrix-multiplication, the algorithm iterates every non-zero element in the row of matrix A , indexing into the corresponding row vector of matrix B . The resulting row is obtained by merging all those rows together. The pseudocode in algorithm 7 illustrate the idea of the matrix-

vector based computation.

For *matrix-vector-multiplication* based SpGEMM implementation, when the non-zero elements are uniformly distributed and the sparse matrices are tall-and-skin, the computational complexity is $O(nkk)$ where n is the number of rows of A , k is the average number of non-zero elements per row. Specifically, the vector-matrix-multiplication method computes matrix C row by row. For each row of matrix C , the process involves iterating each non-zero element in matrix A . And this therefore would involve in total nk iterations. And inside each iteration is the merging process that merges the row B_{ele} into row C_i . Usually the merging takes time that is linear as the list size — k in this case. The total time is therefore $O(nkk)$. For the second case in which the sparse matrix is short-and-fat and the nonzeros are uniformly distributed among the columns, the computational complexity is $O(nkk)$. As there are in average k nonzeros in each column, and the k nonzeros are uniformly distributed among the dimension N . Therefore the average elements per row is k . each The outer two loops that iterate along each row and the nonzeros of that row would take nk iterations considering that there are n rows and each row has in average k elements. The merging operation inside the loop takes k as well. For the third case where the distribution follows a skewed distribution, and the distribution can be simulated by a power-law curve (βx^α), the complexity of this method is less than $n\zeta(\alpha)$. Note that this complexity is a rough estimation and gives a very rough upper bound.

Algorithm 7: matrix-vector based SpGEMM

Input: Sparse matrix A stored in csr format: `OffsetA`, `ColA`

Sparse matrix B stored in csr format: `OffsetB`, `ColB`

Output: the result matrix C

```
1 for  $i \in N$  do
2   for  $ele \in A_i$  do
3      $C_i \leftarrow \text{merge}(C_i, B_{ele})$ 
4   end
5 end
```

5.4.2 Inner-Product-Based SpGEMM

The second way to compute SpGEMM is what we call *inner-product-based* method. This method is based on join operations. It breaks down the matrix multiplication into a sequence of *inner-products* between one row of matrix A and one column of matrix B . The row-vector-matrix multiplication generates the result corresponding to one element in matrix C . This algorithm is illustrated in Algorithm 8.

For *inner-product-based SpGEMM* implementation, when the non-zero elements are uniformly distributed and the sparse matrices are tall-and-skin, the computational complexity is $O(nkk)$. Specifically, the method computes matrix C element by element (C_{ij} for $i \in 1$ to n and $j \in 1$ to n), and this process involves in total n^2 iterations. And inside each iteration is the inner-product that merges the row A_i with column B_j . The inner-product between

Algorithm 8: inner-product based SpGEMM

Input: Sparse matrix A stored in csr format: `OffsetA`, `ColA`

Sparse matrix B stored in csr format: `OffsetB`, `ColB`

Output: the result matrix C

```
1 for  $i \in N$  do
2   for  $j \in N$  do
3      $C_{ij} \leftarrow \text{join}(A_i, B_j)$ 
4   end
5 end
```

two sparse lists are essentially a *join* operation, and join implementation takes time that is linear as the list size — k in this case. The total time is therefore $O(nnk)$. For the second case in which the sparse matrix is short-and-fat and the nonzeros are uniformly distributed among the columns, the computational complexity is $O(nkk)$. As the row dimension of A is k , the dimension of $A \cdot B$ is $k \cdot k$. The outer two loops that iterate along the rows(i) and columns(j) of matrix C would therefore have k^2 iterations. The complexity of the join between A_i and B_j is n , as there are in average n nonzeros in A_i and B_j . Therefore the total complexity is $O(nkk)$. For the third case where the distribution follows a skewed distribution, and the distribution can be simulated by a power-law curve (βx^α), the complexity of this method is less than $O(n^3)$. Note that this complexity is a rough estimation and gives a very rough upper bound.

The complexity of this method depends on the matrix size N , which is irrelevant to the number of non-zero elements because it requires to iterate

every element in the resulting matrix.

Compared to the above matrix-vector based SpGEMM method, when the sparse matrix size is small, the ininter-product-based SpGEMM may not have advantage over the inner-product based method. However, the inner-product method may have the advantage when the matrix is dense. On the other hand, the join operation may be more efficient than merge operation, because it requires less intermediate memory, and it is also prone to better data parallelism. Moreover, if the problem is mask-SpGEMM, the inner-product method complexity is linear to the number of non-zero elements, because it can skip intersections of elements that are outside the masks.

5.4.3 Outer-Product-Based SpGEMM

The third way to compute SpGEMM is based on outer products. It breaks down the matrix multiplication into a sequence of *outer-products* between one column vector of matrix A and one row vector of matrix B . The result is later merged into matrix C . Algorithm 9 illustrates the pseudocode of this method.

For *outer-product-based SpGEMM* implementation, when the non-zero elements are uniformly distributed and the sparse matrices are tall-and-skin, the computational complexity is $O(nkk)$. Specifically, the method computes matrix C by iterating every column-row vector pairs of A and A^T , and this process involves in total n iterations (cause there are n different columns). And inside each iteration is the outer-product between column A_i and row B_j . The outer-product takes $k \cdot k$ time (because the column length of A (A_i) is

Algorithm 9: Outer-product based SpGEMM

Input: Sparse matrix A stored in csr format: `OffsetA`, `ColA`

Sparse matrix B stored in csr format: `OffsetB`, `ColB`

Output: the result matrix C

```
1 for  $i \in N$  do
2   for  $j \in N$  do
3      $C \leftarrow (A_i \times B_j) \cup C$ 
4   end
5 end
```

in average k). Therefore the total time is therefore $O(nkk)$. For the second case in which the sparse matrix is short-and-fat and the nonzeros are uniformly distributed among the columns, the computational complexity is $O(nkk)$. The analysis is similar with the first case. For the third case where the distribution follows a power-law curve (βx^α), the complexity of this method is less than $O(n^3)$.

5.5 Join-Based SpGEMM

5.5.1 The Baseline Join-Based SpGEMM

Suppose the matrix has N rows, and M non-zero elements. The baseline join-based SpGEMM iterates through each element in matrix C by rows and columns. And the element c_{ij} is obtained by intersecting the i -th row of A (A_i) with the j -th column of B (B_j). [Listing 5.1](#) shows the merge-based intersection

method.

```
1 int scalar_merge_intersection(int L1[],
2                               int n1, int L2[], int n2) {
3     int i = 0, j = 0, r = 0;
4     while (i < n1 && j < n2) {
5         if (L1[i] < L2[j]) {
6             i++;
7         } else if (L1[i] > L2[j]) {
8             j++;
9         } else {
10            i++; j++; r++;
11        }
12    }
13    return r;
14 }
```

Listing 5.1: A code example of scalar merge-based set intersection

5.5.2 The Data-Parallel Join-Based SpGEMM

Previous research focusing on CPU intersection has proposed methods to leverage the SIMD instructions for accelerations. In order to leverage the SIMD instruction, the method performs all-pair comparisons between elements held in two vector registers. This all-pair comparison introduces more computations, yet it is actually faster than the scalar-based intersection method on CPU with vector instructions. The reasons are two-fold: first, the all-pair comparisons

can replace the *compare-and-advance* operations in scalar intersection. The all-pair comparison operation is inherently data-parallel, which is capable of reducing the number of branch instructions executed, and thus reducing the mispredictions that can cause expensive pipeline stalls. Second, it is capable of leveraging the SIMD instructions which can introduce more compute capabilities than only using scalar instructions. In the following, we introduce two different data-parallel join-based implementations for intersection-based SpGEMM on GPUs.

All-pair comparisons. The first method allocates 32 threads in a block to perform join using 32x32 all pair comparisons. Specifically, the 32 threads in the thread block first load 32 elements from both row i of matrix A and column j of matrix A^T , and reserve the loaded element in the local register. Next, the thread block compares the 32 elements in a round-robin fashion. In the first iteration, the first thread broadcasts its local value A_0 to all the other threads using the shuffle instruction *shlf*. The other threads will compare A_0 with their local element B_i (for thread i). If $A_0 == B_i$, they will increment their local counter by 1. This process will be repeated for all 32 elements. After the 32x32 comparison is done, the algorithm will proceed to the next 32 elements. To decide which list gets the next 32 elements, the algorithm compares the A_{31} with B_{31} , and the pointer of the list that has a smaller value will proceed to the next 32 elements. After either iterator reaches the end of the two lists, the comparison will stop. Afterward, the local counter of all threads in this block will be aggregated together as the final result and

stored into C_{ij} . Figure 5.4 illustrates this method.

There can also be different blocking choices for the all-pair comparison based joins. For example, we can launch 64 threads in a block and arrange them as 8-by-8 pairs. Figure 5.4 illustrates this method.

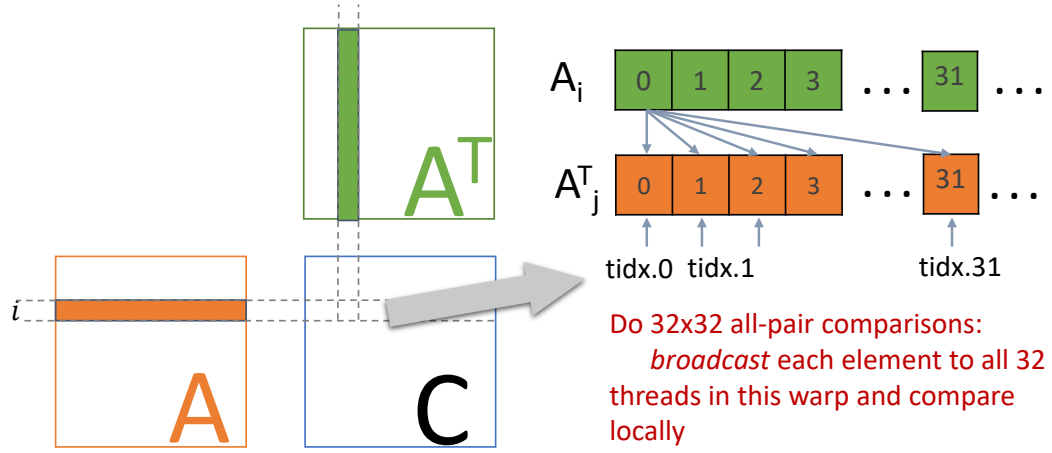


Figure 5.3: All-pair comparison based join implementation: a 32-by-32 implementation.

5.5.3 Join Using Wider Data Types

CUDA also provides wider instructions (e.g. `int2`, `int3`, `int4`). Those instructions can simultaneously manipulate data types that are larger than the conventional 64-bit. With these instructions, wider memory bandwidth can be achieved. As the scalar-based join method is memory bound, we can further leverage those wider instructions for acceleration.

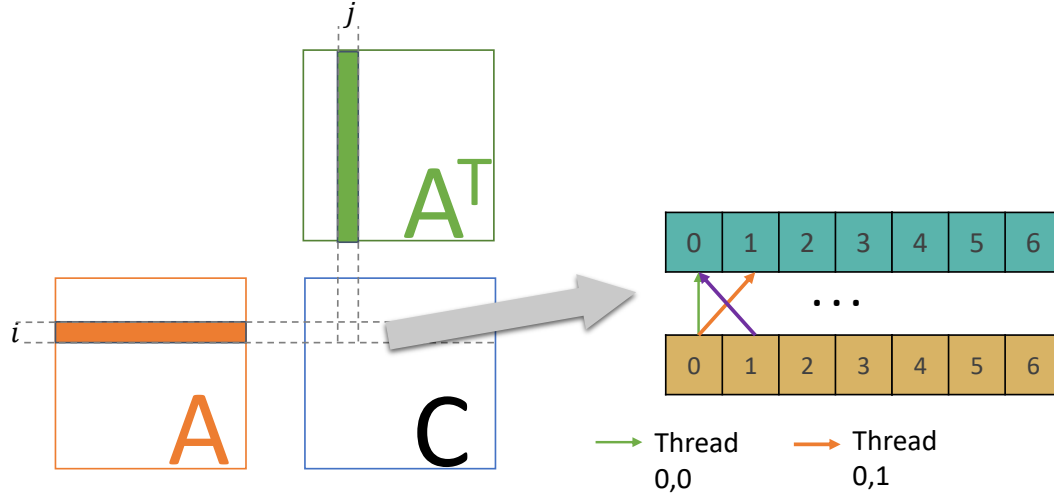


Figure 5.4: All-pair comparison based join implementation: a 8-by-8 implementation.

The wider integer type requires the array to be memory-aligned. To fulfill this requirement, one solution is to pad the matrix. However, padding comes with extra overheads, and padding may not always be guaranteed or available if the input is provided via an external interface. In such situations, we need to manually take care of the data alignment. We break down the neigh list into three regions: the *middle* region has the aligned starting and ending addresses, the *prolog* region is the region from the beginning of the list to the first memory-aligned point, the *epilog* region is the region from the last memory-aligned point to the end of the list. The idea is to compute the *middle* region using int4 data types with all-pair data parallelism comparisons and to compute the prolog and epilog region using the scalar intersection.

Listing 5.4 shows how the intersection using int4 data types is computed. Firstly, we need to compute the prolog of the two lists. The *compute_prolog*

```
1 __device__ uintptr_t compute_prelog(int* addr){
2     char* p = reinterpret_cast<char*> (addr);
3     uintptr_t p_ru = (((uintptr_t)(p+15))>>4)<<4);
4     return p_ru;
5 }
```

Listing 5.2: function to get the aligned address of the list start

```
1 __device__ uintptr_t compute_epilog(int* addr){
2     char* p = reinterpret_cast<char*> (addr);
3     uintptr_t p_rd = (((uintptr_t)p)>>4)<<4);
4     return p_rd;
5 }
```

Listing 5.3: function to get the aligned address of the list end

function in Listing 5.2 obtains the prolog region by rounding up the list’s starting address to the closest point that is memory aligned. Elements from the beginning to the aligned rounding point of the two lists are intersected using scalar methods. Similarly, the *compute_epilog* function in Listing 5.3 obtains the epilog region by rounding down the list’s ending address to the closest memory aligned point. The epilog regions are intersected using scalar intersections. The middle regions that are between the memory-aligned starting and ending points are perfectly aligned, therefore they can be intersected using the 4-by-4 all-pair data-parallel intersections.

```

1 __global__ int scalar_merge_intersection(int L1[],
2     int n1, int L2[], int n2) {
3     int i = 0, j = 0, r = 0;
4     int tid;
5     uintptr_t prelogi = compute_prelog(i);
6     uintptr_t prelogj = compute_prelog(j);
7     uintptr_t epilogi = compute_prelog(i);
8     uintptr_t epilogj = compute_prelog(j);
9
10    //compute prelog via scalar-intersection
11    while (i < prelogi && j < prelogj) {
12        if (L1[i] < L2[j]) {
13            i++;
14        } else if (L1[i] > L2[j]) {
15            j++;
16        } else {
17            i++; j++; r++;
18        }
19    }
20
21    //compute the middle using wide data types
22    //convert the pointer to int4* pointer
23    int4* pint4_prelogi = reinterpret_cast<int4*> prelogi;
24    int4* pint4_prelogj = reinterpret_cast<int4*> prelogj;
25    int4* pint4_epilogi = reinterpret_cast<int4*> epilogi;
26    int4* pint4_epilogj = reinterpret_cast<int4*> epilogj;
27    while(pint4_prelogi < pint4_epilogi && pint4_prelogj <
28        pint4_epilogj){
29        int4 n1 = *pint4_prelogi;
30        int4 n2 = *pint4_prelogj;
31        tt += (n1.x==n2.x) + (n1.x == n2.y) + (n1.x == n2.z)
32            + (n1.x == n2.w);
33        tt += (n1.y==n2.x) + (n1.y == n2.y) + (n1.y == n2.z)
34            + (n1.y == n2.w);
35        tt += (n1.z==n2.x) + (n1.z == n2.y) + (n1.z == n2.z)
36            + (n1.z == n2.w);
37        tt += (n1.w==n2.x) + (n1.w == n2.y) + (n1.w == n2.z)
38            + (n1.w == n2.w);
39    }
40
41    //compute the epilog using scalar intersection
42    //convert the int4 pointer back to int, start from
43    where is left
44    int* pi = (int*) pint4_prelogi;
45    int* pj = (int*) pint4_prelogj;
46    while(pi < colA + indA[i+1] && pj < colB +
47        return r;
48 }

```

5.6 Join-Based SpGEMM Using Hash Methods

We have discussed above the merge-based SpGEMM methods. Note that the primitive operations in the join between rows of matrix A and matrix B . The previous section mainly focuses on using the pointer-based merge method. Such method takes zero additional memory overheads but is inherently sequential. We have also explored different ways to make the scalar join method parallel across warps. Another way to compute sparse data is by using a hash table. And there are different ways to optimize the hash table. The hash table is based on the linear-probing open address method, where we have a fixed size hash table for each row. One way to leverage hash is to build a hash table for a row each time, and use the elements in the multiplicand matrix to probe the hash table. complete the row-matrix-multiplication.

There are different ways to implement hash-based join SpGEMM depending on how the threads are assigned. The first way of hash-based SpGEMM is to assign all the threads in a block to process the same row. Figure 5.5 shows the details of this process. This process composes of three steps. The first step builds a hash table for K rows of the matrix B (K is the blocking parameter). In order to differentiate elements from different rows, we use $rowid \cdot N + columnid$ as the hash key. Afterward, the algorithm performs the multiplication between each row of A and the block of B by indexing in the hash table. Then the results of all the threads in this block will be aggregated

together into C_{ij} .

Another way is to assign all the threads in a block to process different rows. It has a similar step one as the above method, which builds a hash table for K rows of matrix B (K is the blocking parameter). But in step two of indexing the nonzeros in A , this approach assigns each thread to process a different row of A . In addition, in step three, unlike the previous method, this method does not need to perform block-wise reduction — the local sum at each thread corresponds to the nonzero count of one row. Figure 5.6 shows the details of this process.

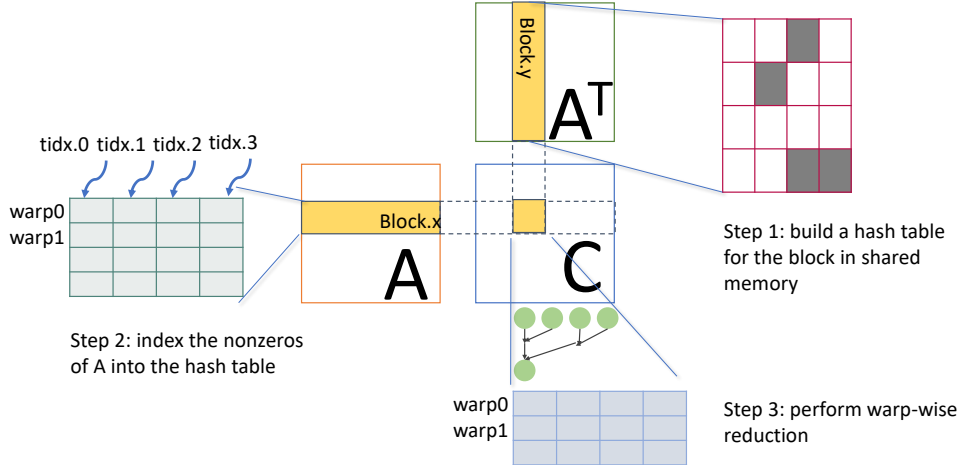


Figure 5.5: Hash-based SpGEMM implementation: parallel across columns.

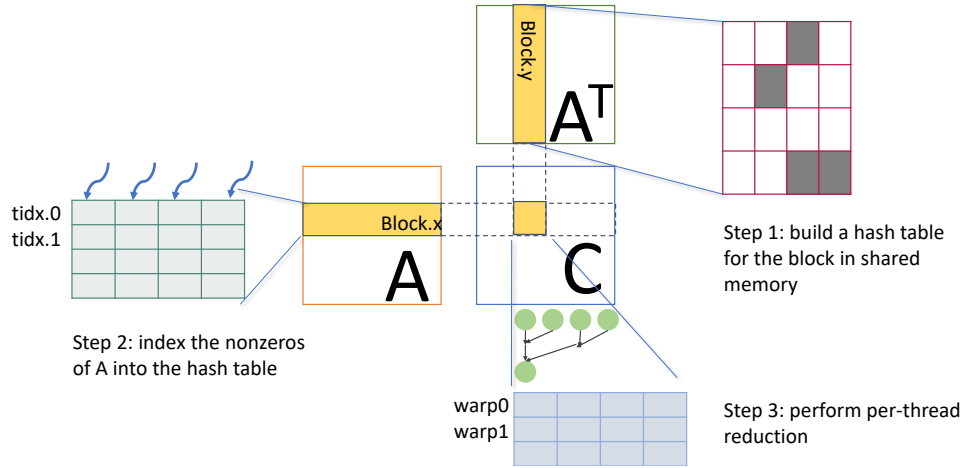


Figure 5.6: Hash-based SpGEMM implementation: parallel across rows.

5.7 Online Scheduling

As we discussed in section 5.3.2, many applications require to perform SpGEMM iteratively. As the density of the sparse matrix input can change across iterations, the SpGEMM algorithm should be able to adapt accordingly to the density patterns of the input matrix. Therefore, we need an online scheduling mechanism that is capable of picking the best performant SpGEMM algorithm in each iteration.

The design of the online scheduling mechanism should take into account two important aspects: Firstly, as the scheduling process is happening online, the process should be light-weight, i.e., the time to compute the scheduling decision should be negligible compared to the SpGEMM computation time. Secondly, the online scheduling mechanism should respond appropriately to the change of input density characteristics.

There are two different approaches to design the online scheduling mechanism: The first approach is based on a theoretical analytic solution. Through theoretical modeling, one can build the model to work out which SpGEMM can perform best under a multitude of situations, and only need to pick the solution suggested by the analysis during the online computation. The advantage of this approach is that it has the assurance of a correct outcome, but the disadvantage is also that it requires the construction of analytic modeling. And the computation of online analytics may be too expensive. Meanwhile, more recently, a lot of research is investigating using machine learning models to replace traditional analytic models. For example, machine learning based approaches have been applied in many database tasks to replace traditional data structures in tasks such as indexing, searching, etc. Comparing to analytical modeling, learning-based approach does not that strong correctness guarantees. But the advantages of using machine learning models are: first, the computation of the machine learning model is usually regular. Second, using a machine learning model is proof of the noise in the input data.

In this work, we plan to use machine learning based scheduling, based on two considerations: first, there are a variety of different SpGEMM algorithms and data structures, and it is impractical to construct the analytic solution for all the combinatorial cases. Secondly, the computation of machine learning model can be fast and light-weight. Previous work have demonstrated that it can be used to substitute the complex analytic models for online scheduling.

The online scheduling procedure is illustrated in Figure 5.7. We use

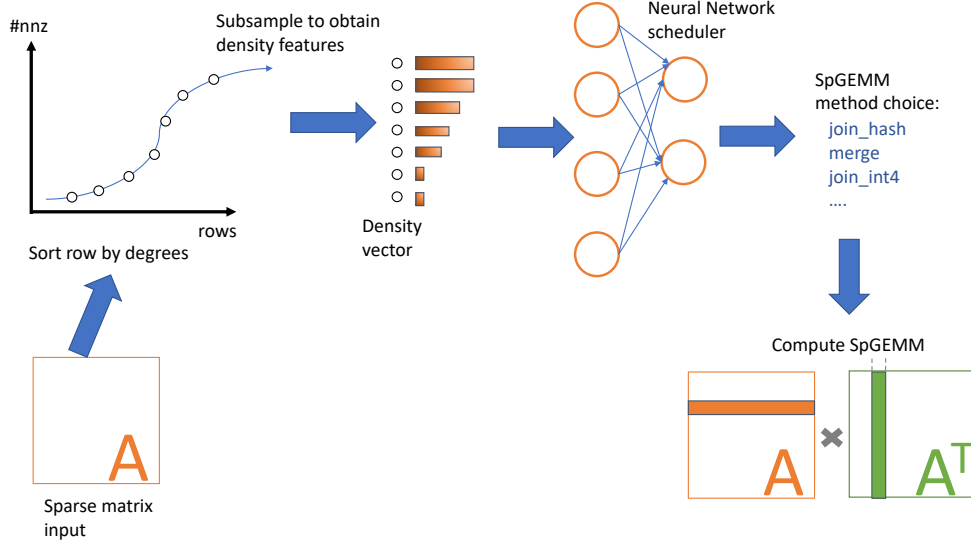


Figure 5.7: Online scheduling strategy.

a fully-connected neural network as our scheduling model. The reasons are two-fold: a fully-connected neural network is an important component to construct a classifier. Additionally, it is simple and easy to compute, making it a light-weight choice for online scheduling. The input to the scheduling model is the sparse matrix density information. If the sparse matrix is pre-sorted by row degrees, the power-law parameter can be approximated by the row density histograms. Therefore, the number of elements per row can be used to approximate the distribution. But as the neural network requires the input dimension to be fix-sized, while the data set dimension can vary. To ensure the input dimension, we subsample the rows of the input matrices uniformly so that we take a fixed number of rows, and use their densities (the number of nonzeros divided by N), as the input to the NN model. The model acts as

a simple classifier and outputs a single value which represents the SpGEMM method choice. Then the corresponding SpGEMM implementation will be selected to compute the sparse matrix multiplication problem.

5.8 Experiments

5.8.1 Experimental Setups

Dataset

In the experiments, we use both synthetic and real-world datasets. The synthetic datasets are generated using the boost power-law graph generator [121]. The power-law generator generates graph whose nodes degree follow an exponential distribution:

$$P(x) = \beta x^{(-\alpha)} \quad (5.3)$$

The parameter α controls the skewness of the data distribution. The larger the α , the more skewed the dataset is. When $\alpha = 0$, the generated matrix has uniformly distributed nonzeros. β controls the sparsity of the matrix by changing the average number of elements per row.

Methods

In the experiments, we compare our four different join-based SpGEMM implementations with a baseline merge-based SpGEMM implementation from cusparse. The four join-based SpGEMM implementations are join_baseline,

join_int4, join_dp, and join_hash. These four implementations are all based on the SpGEMM-using-intersection method illustrated in Algorithm 8.

More specifically, we evaluate two SpGEMM computations in our experiments: AA^T (*non-masked-SpGEMM*) and $A \odot AA^T$ (*masked-SpGEMM*). These two computations are primitive operations appearing in many graph analytics tasks where A is the input graph which is stored as a sparse matrix. In our experiments, we assume A is stored in CSR format. To compute AA^T , the join-based approach first allocates a dense matrix to store the temporal result. The dense matrix takes $O(N^2)$ space where N is the number of vertices. Next, the join-based approach computes the intersection for every row-column pair between A and A^T . Note that the i -th column of matrix A^T is the i -th row of matrix A . So no transpose is needed. How the four join-based SpGEMM methods perform the intersection are summarized below. After intersecting every row-column pair, the result matrix C which is temporarily stored in the dense $O(N^2)$ matrix will be compactified and transform into CSR format. To compute the *masked-SpGEMM* $A \odot AA^T$, join-based SpGEMM methods first allocate the memory space of $O(\text{nnz})$ in which **nnz** the number of nonzeros in matrix A . Next, the method performs the intersection on the positions of A that have nonzeros elements.

- **Join_baseline.** The join_baseline method computes SpGEMM by intersecting every row and column of the matrices A and B as illustrated in Algorithm 8. The intersection is based on the sequential intersection method as shown in listing 5.1. The computation is parallelized such that

a warp is assigned to compute one row in matrix C , and each thread is assigned to a different column. In the following experiments, without special notice, the block size is set to 64, and the grid size is set to 1024.

- **Join_int4.** The `join_int4` method is similar to the `join_baseline`. It uses the same join-based SpGEMM idea as illustrated in Algorithm 8, but when performing the two list intersection, it uses the wider data types (int4) as discussed in section 5.5.3. Listing 5.4 illustrates the pseudocode of this method.
- **Join_dp.** The `join_dp` refers to the methods that use the data-parallel all-pair comparison intersection scheme as described in section 5.5.2. Among them, **join_dp8x8** assigns all 32 threads in a warp to performs 8-by-8 all pair comparisons. **join_dp32x1** assigns all 32 threads to perform 1-by-32 all pair comparisons.
- **Join_hash.** The `join_hash` is the join-based SpGEMM method using hashtable for the intersection as described in 5.6.
- **Merge_cuspars.** The `merge_cuspars` refers to the cuspars library implementation. The cuspars implementation is based on the merge method.

System

The experiments in this section are run on a Linux workstation with a TITAN V GPU. The TITAN V GPU has 12 GB HBM2 memory with 5120 cores, 80

streaming multiprocessors. The total memory bandwidth is 652.8 GB/s. The CPU is a 4-core Intel Core i7-4770K architecture, running at a frequency of 3.5GHz.

The GPU programs were compiled with NVIDIA’s nvcc compiler (version 8.0.44). The C code was compiled using gcc 4.9.3. All results ignore the transfer time (from disk-to-memory and CPU-to-GPU). The merge path operation is from the Modern GPU library [122]. The version of cuSPARSE used is 8.0 [123].

5.8.2 Performance of GPU Joins with Very Sparse Matrices

In this experiment, we compare our different join-based SpGEMM implementations on very sparse matrices (i.e., the average nonzeros per row is fixed and of a limited amount). The methods compared in the experiment include the sequential join, the two data-parallel join variants, and the merge-based SpGEMM. In the experiment, we fix the matrix dimensions N to 1000 while varying the average NNZ per row from 0 to 50. The experiment is based on masked-SpGEMM. Figure 5.8 shows the performance result.

We can see from the result that the merge is slower than join-based approaches. There are two reasons for this: firstly, the merge approach does not efficiently utilize all the threads in a block, when the number of nonzeros per row is small, as the merge-based method assigns the entire thread block to process one row. When the nonzeros in a row are smaller than the thread

block size, some threads will be idle. Secondly, when the nonzeros per row are very small, the hash merge overheads kick in, and iterating the hash table takes time linear to the hashtable size, which is much larger than the number of non-zeros.

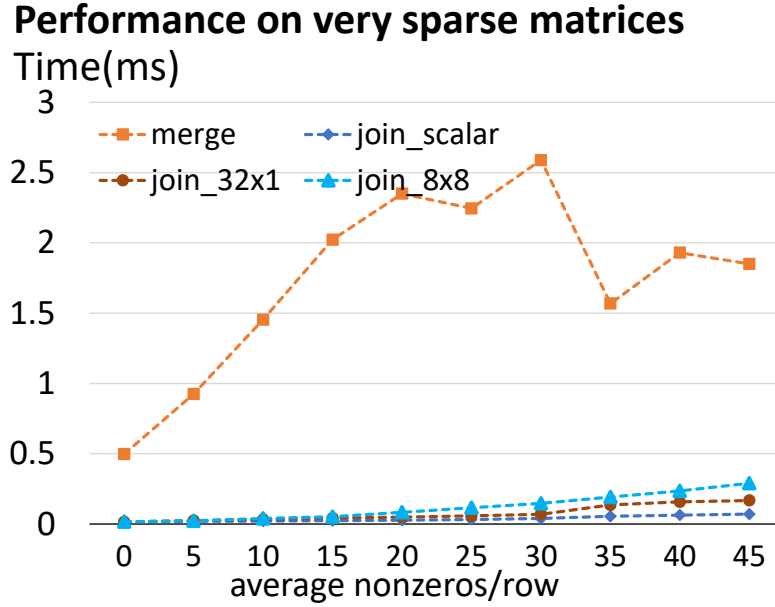


Figure 5.8: Comparison of join implementations on very sparse matrices.

5.8.3 Performance of Join-Based SpGEMM

We have presented the analysis on how the computational complexities of those algorithms are affected by the density of the matrix in [Figure 5.2](#). In [Figure 5.9](#) and [Figure 5.10](#), we show the execution time comparisons of those algorithms on synthetically generated power-law matrices with different densities. We control the densities of the sparse matrices by changing the β value

of the power-law generator when generating these synthetic datasets. From the results, we can see that the relative performance can be quite different for different SpGEMM methods depending on the matrix densities. Figure 5.9 shows the execution time on sparse matrices of different sizes whose density is 0.3. For this density, we can see that join-based methods generally perform better than the merge-based methods. We can see that the sequential join-based method is faster than the merge-based method (cusparse) by 40% ~ 10x. The optimized join method leveraging the wider data types (join_int4) can be up to 2x faster than the join sequential. Figure 5.10 shows the execution time result on sparse matrices whose density is 0.01. For this density, we can see that the performance of join-based methods is not as good as the merge-based methods. This is because join-based SpGEMM would have much higher complexity when the matrices are sparse. They need to check and perform many unnecessary joins on the elements that do not exist in the result.

5.8.4 Comparison on Different Densities

Figure 5.11 shows the performance of different SpGEMM methods on varying sparsities. The result is based on generalized SpGEMM (without mask). We can see that for sparse matrices in which densities are less than 0.1, merge-based methods are better. This is because their complexity is lower than join methods, which needs to perform many unnecessary joins even the elements are zero in the result. When density is above 0.1, join-based SpGEMM methods are faster than merge-based methods. This is because the amount of compu-

Performance on varying sizes (sparsity = 0.3)

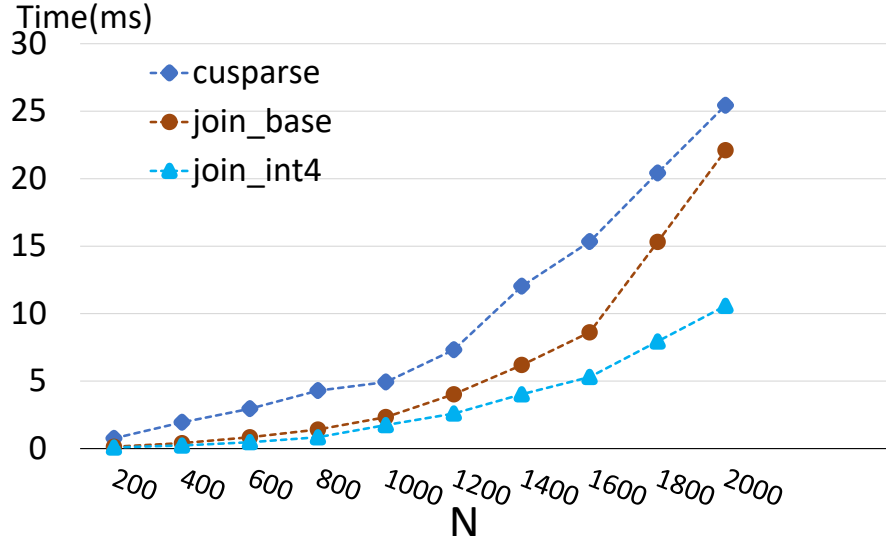


Figure 5.9: Comparison of join implementations on relatively dense synthetic data in which the densities are 0.3.

Performance on varying sizes (sparsity = 0.01)

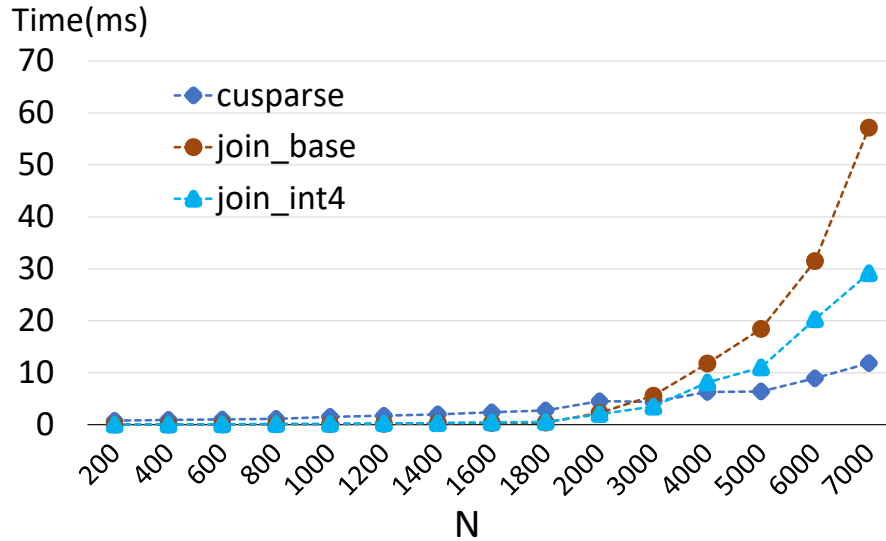


Figure 5.10: Comparison of join implementations on relatively sparse synthetic data in which the densities are 0.01.

tation performed by the merge method and join method comparable when the matrices are relatively dense. On top of that, the merge method would incur overheads. In particular, when the result is relatively dense, there would be many contentions in the hash table merge operation.

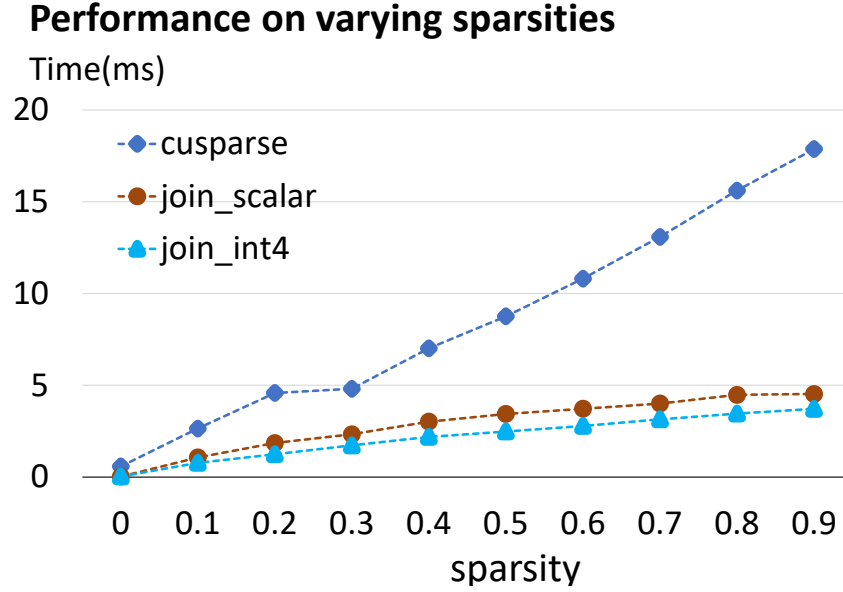


Figure 5.11: Comparison of join implementations on varying sparsities.

5.8.5 Performance on the Hash-Based Join Kernels

In this section, we compare the performance of our hash-based SpGEMM implementations. The results are shown in [Figure 5.12](#) and [Figure 5.13](#). *Hash1* refers to the method where all the threads in a warp collectively process the columns across the same row, and *Hash2* refers to the method where threads in a block process different rows. Experiments are conducted on synthetic

datasets whose densities are 0.05.

Our implementation uses an open-addressing hash. In the implementation, we build 1K hash table for each row, and warps build hashtables for blocks of rows simultaneously. The result in [Figure 5.12](#) and [Figure 5.13](#) demonstrates the impact of different blocking parameters on performance. Both experiments have similar settings, but the hash table size per row changes. [Figure 5.13](#) has larger hash table size per row. We can see from the result that *Hash2* (threads parallel across different rows) is faster than *Hash1* (threads access different columns). And increasing the hash table size can help improve performance. This is because a larger hash table can reduce contention in the hash building process.

Performance on the Hash-Based Join Kernels with Varying Densities

In this experiment, we compare the performance of our hash-based SpGEMM implementations against the merge method. The general SpGEMM (without mask) problem is implemented in this experiment. The results are shown in [Figure 5.14](#) and [Figure 5.15](#). [Figure 5.14](#) shows the performance on synthetic datasets whose densities are 0.3. We can see that hash-based implementation can be 10x faster than merge based method. [Figure 5.15](#) shows the performance on datasets whose densities are 0.01. We can see that for such sparse datasets, merge based cusparse is faster than hash-based implementations. This is because for very sparse datasets, the complexity of the join-based methods is related to N^2 , which is much higher than that of merge based

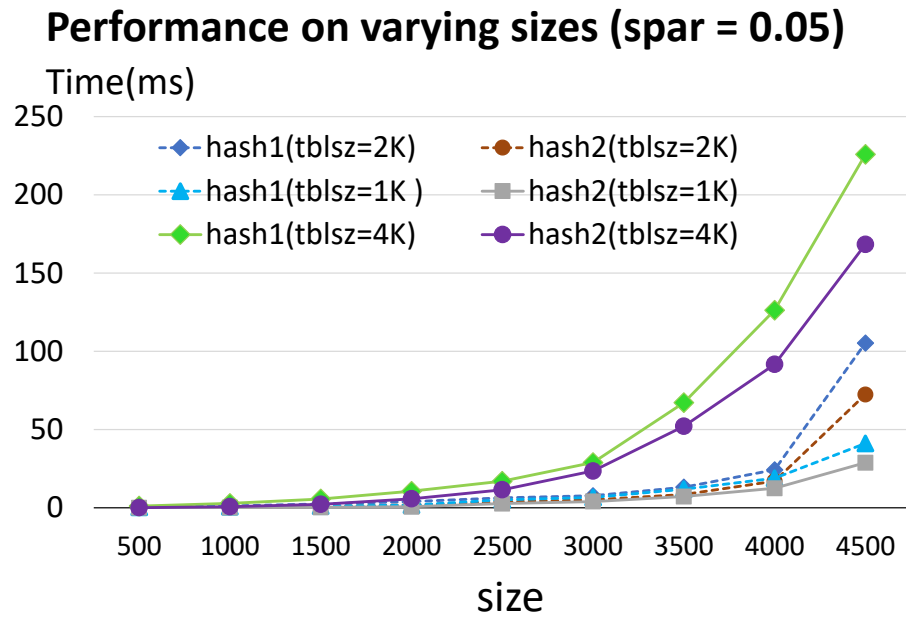


Figure 5.12: Comparison of hash-based join implementations on datasets of density 0.05. The hash table size is 1KB per row. tblsz = 1K/2K/4K refers to building one hashtable for 1/2/3 rows respectively.

Performance on varying sizes (spar = 0.05)

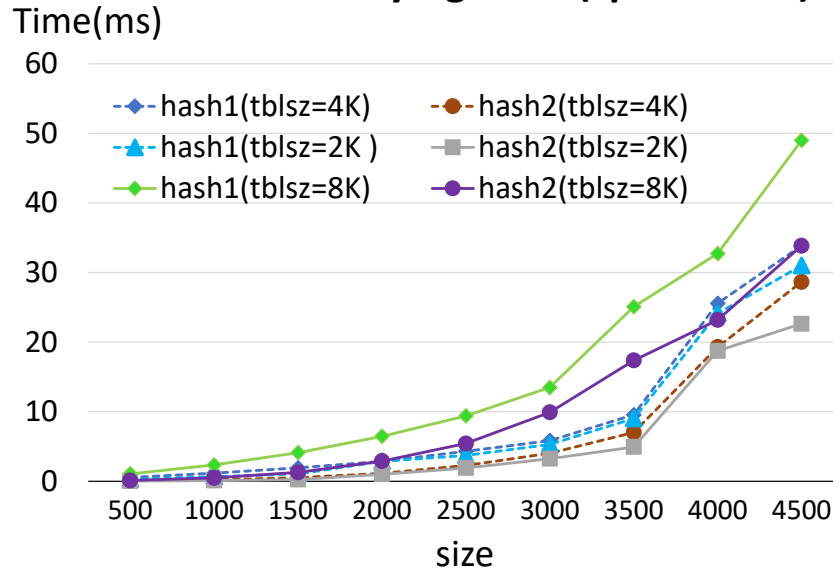


Figure 5.13: Comparison of hash-based join implementations on datasets of density 0.05. The hash table size is 2KB per row. We build a hashtable for 1/2/3 rows simultaneously. tblsz = 1K/2K/4K refers to building one hashtable for 1/2/3 rows respectively.

method.

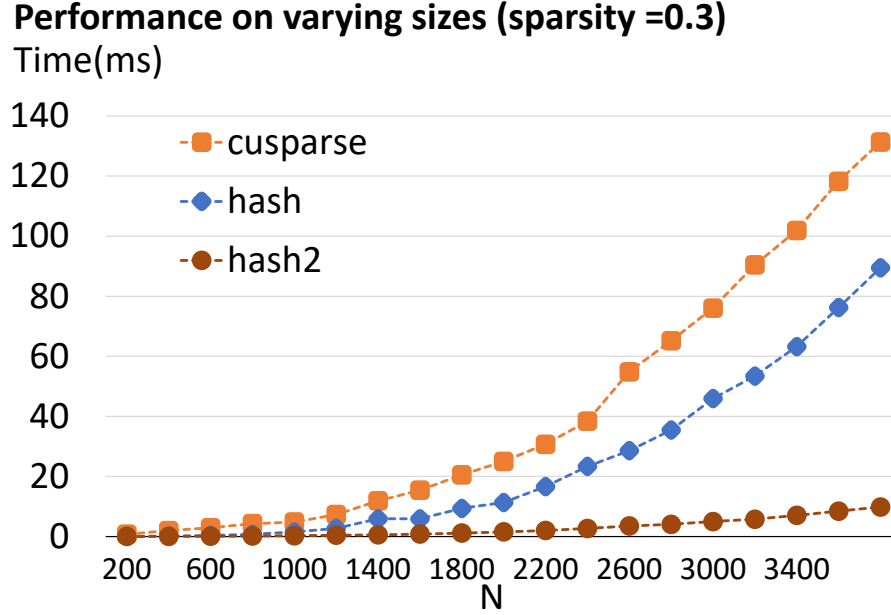


Figure 5.14: Comparison of hash-based join implementations on relatively sparse synthetic data in which the densities are 0.3.

5.8.6 Performance on Real-World Dataset

We evaluate our implementations on real-world datasets. These real-world datasets are collected from the SuiteSparse Matrix Collection. Their details are shown in [Table 5.1](#). In this experiment, we compare our iterative SpGEMM implementations with machine-learning based scheduling against the baseline method which keeps using the merge-based SpGEMM implementations cross iterations.

Specifically, our scheduling method is based on the neural network

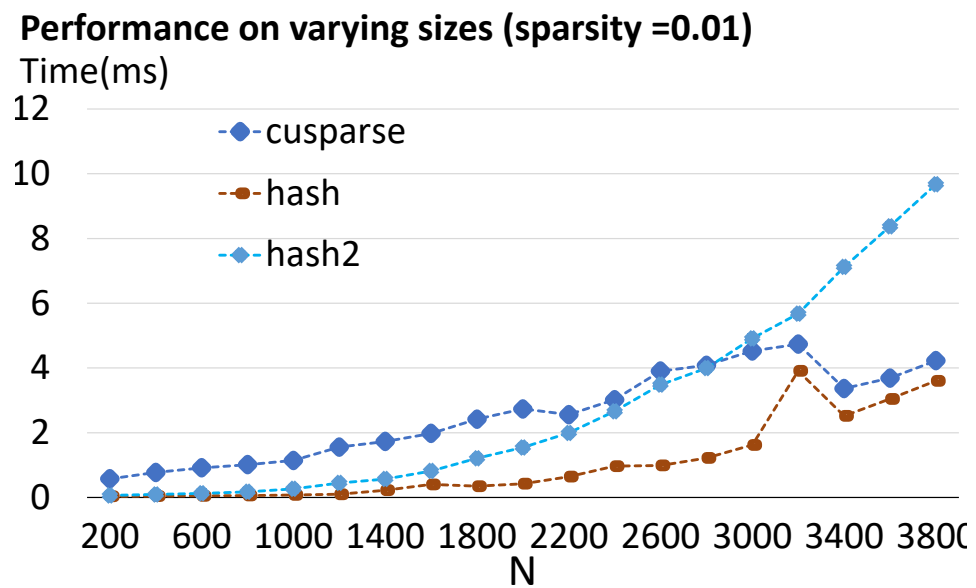


Figure 5.15: Comparison of hash-based join implementations on relatively sparse synthetic data in which the densities are 0.01.

scheduler as described in section 5.7. The scheduler is a two-layer fully connected network. Each layer has 100 neurons, followed by a softmax layer in the end. The output of this neural network is a scalar, whose value will be rounded to an integer between 0 to 5, representing the `join_baseline`, `join_int4`, `join_dp`, `merg_cusparse`, and `join_hash` based method respectively. In each iteration, we first execute the scheduling neural network. And based on the output of the scheduling neural network, we chose the corresponding SpGEMM method.

From the experiment result, we can see that our scheduling-based implementation can be up to 3.6x faster than the non-scheduling one, after running for three iterations. The speedups come from the fact that the matrix is progressively becoming denser and denser after several iterations, and our scheduling mechanism is capable of capturing the density change and switch to a faster SpGEMM implementation accordingly. For example, the *bcsstk13* dataset has a sparse input matrix A which has on average 20 elements per row. The first AA^T iteration is performed using the merge-based method. But after the first iteration, the density of the resulting matrix is significantly increased. Our scheduling mechanism is capable to adapt such change in density and switch to a hash-based SpGEMM method (`join_hash`) for the subsequent iterations. This can result in over 3x speedups comparing to the baseline method that sticks to merge-based implementations. On the other hand, we didn't observe speedups for dataset *bcsstk17* and *cant*. This is because these two datasets didn't trigger the change in SpGEMM method, either because their density didn't change much across iterations, or the neural network scheduler

Table 5.1: Performance comparison on iterative SpGEMM computations with real-world datasets.

| Dataset | No. nodes | No. edges | baseline | scheduling | speedup |
|----------|-----------|-----------|-------------|-------------|---------|
| bcsstk13 | 2003 | 42943 | 23.777984 | 6.475968 | 3.675 |
| bcsstk17 | 10974 | 428650 | 79.097119 | 89.09 | 0.88 |
| cant | 62451 | 2034917 | 4088.257021 | 4085.100191 | 1.00 |
| opt1 | 15450 | 973052 | 4681.783642 | 951.42 | 4.92 |
| msc | 10849 | 620313 | 7175.93 | 522.25 | 13.74 |

didn’t produce the correct prediction. That is why there are no speedups or and there is even a performance drop because of the extra overheads incurred when running the scheduler.

5.8.7 Performance on Real-World Graph Datasets with Skewed Distribution

Many graph datasets have skewed distributions, in which some rows can have a lot more elements than others, and this would introduce the load balancing problem in sparse matrix multiplication computation. To tackle the load balancing problem, we introduce a diverse set of thread assignment strategies. Specifically, we propose to introduce block strategies to divide the input matrices into regions of different row densities. Then we can assign different thread blocks to process different regions. For example, we could group rows whose density is more than 50% together, and assign large thread blocks to process such rows. And for rows whose density is rather sparse, we should assign smaller thread blocks to achieve better resource utilization. Therefore,

we introduce the following assignment plans to tackle the skewed datasets: we assume that the input matrix is presorted by row degrees, and we have a pool of pre-defined thread assignment strategies as illustrated below. At run time, A light-analysis analysis will be performed on the input matrix densities and the best-fit thread assignment strategy will be selected based on the analysis:

- Assignment strategy 1. The whole matrix is treated the same. Each row is assigned with one thread block of 32 threads.
- Assignment strategy 2. The whole matrix is treated the same. Each row is assigned with one thread and one thread block has 32 threads.
- Assignment strategy 3. The matrix is divided into a “dense” region and a “sparse” region. The “dense” region consists of rows at the top twenty percent (after sorting by row degrees in descending order), and the “sparse” region is the rest of the rows. The top twenty percent rows are assigned with thread blocks of size 256, and the rest of the rows are assigned with thread blocks of size 32.
- Assignment strategy 4. The matrix is divided into a “dense” region and a “sparse” region. The “dense” region consists of rows at the top ten percent after degree sorting, and the “sparse” region is the rest of rows. The top ten percent rows are assigned with thread blocks of size 256, and the rest of rows are assigned with a single thread per row and the thread block size is 32.

The experiment in this section is based on *masked-SpGEMM* problem.

We extend the `join_baseline` with the above assignment strategies. Figure 5.16 shows the performance of these different assignment strategies on real-world graph data collected from the SNAP datasets [86]. The graph dataset statistics are shown in listing 5.2. We can see that different datasets can favor different assignment strategies. There is no one-size-fits-all strategy that fits all datasets.

In order to pick the best assignment strategy, we leverage the neural network scheduler to perform a light-weight analysis on the densities of the input matrices. We re-train the neural network scheduler with these amended assignment options. The input to the neural network is the same as before: a 1x100 vector that represents the densities of 100 rows subsampled from the sparse matrix. The output now becomes an integer value from 0 ~ 9, instead of from 0 ~ 5. Value 6, 7, 8, and 9 refers to the assignment strategy 1, 2, 3, and 4 respectively. After the retraining, the accuracy of our neural network predictor drops from 83.8% to 72.2%, but it is able to incorporate different assignment strategies for skewed datasets.

Note that the above experiment is a preliminary step towards solving the load balancing issue with the skewed datasets. In the experiment, we have only demonstrated four different assignment strategies and on a `join_baseline` implementation, but we believe that this idea can be easily extended to more assignment strategies, and we can incorporate into more join- or merge-based SpGEMM methods. Additionally, how to better design the neural network scheduler for better accuracy and speedup is also an interesting open question

Table 5.2: Graph dataset characteristics.

| | N | M | avg nnz/row |
|--------------|-----------|-----------|-------------|
| cit-HepPh | 34,546 | 421,578 | 12.2 |
| cit-HepTh | 27,770 | 352,807 | 12.7 |
| web-Stanford | 281,903 | 2,312,497 | 8.2 |
| com-dblp | 317,080 | 1,049,866 | 3.3 |
| ca-CondMat | 18,772 | 198,110 | 10.5 |
| web-Google | 875,713 | 5,105,039 | 5.8 |
| roadNet-PA | 1,088,092 | 1,541,898 | 1.4 |
| p2p-Gnutella | 10,876 | 39,994 | 3.6 |

that can be left as future work.

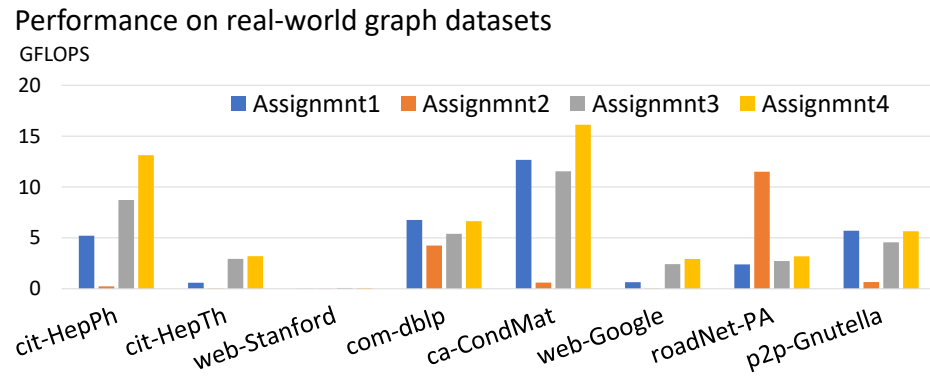


Figure 5.16: Comparison of different thread assignment strategies on real-world graph datasets.

Chapter 6

Summary and Future Directions

Contents

| | | |
|------------|----------------------------------|------------|
| 6.1 | Summary | 162 |
| 6.2 | Takeaways and Reflections | 164 |
| 6.3 | Future Directions | 167 |

In this dissertation, we investigate the acceleration and performance tuning of the motifs in machine learning tasks on modern processors. The motifs form the backbones of understanding the interplay between software optimizations and hardware architecture, which is a fundamental step towards building a portable and efficient end-to-end ML systems. In particular, this work is focusing on two sets of computational kernels: the dense computational kernels in convolutional neural networks, and the sparse kernels in data analytical tasks such as social networks. These kernels are fundamental to a majority of machine learning tasks, and have received tremendous attention

recently in both academia and industry. Figure 6.1 demonstrates the overview of this dissertation.

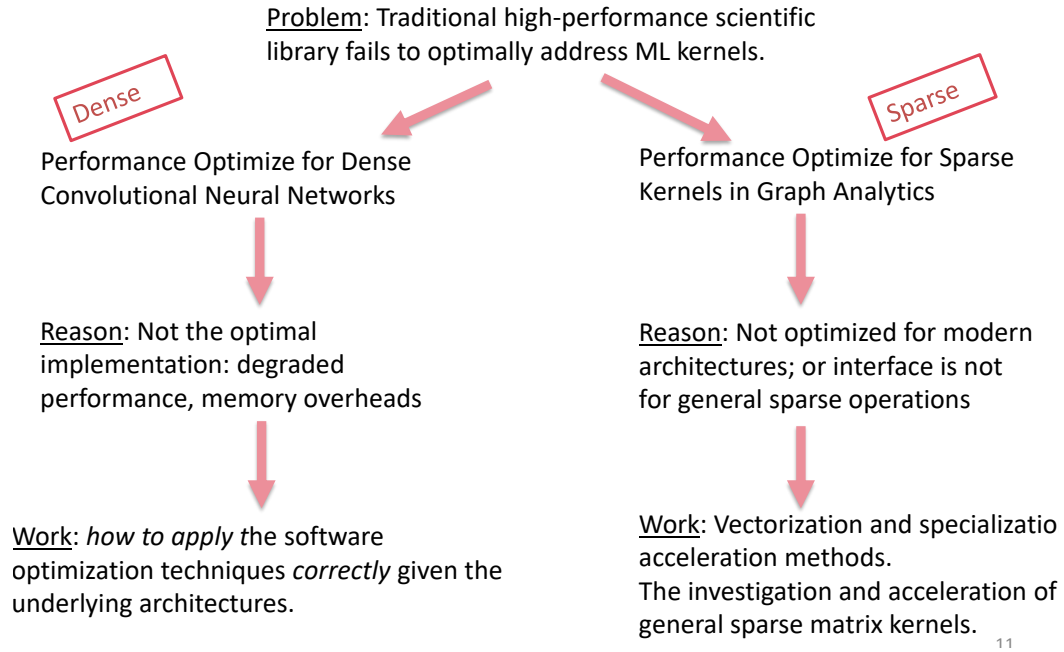


Figure 6.1: Overview of this dissertation

6.1 Summary

In chapter 2, we investigate the optimization of dense kernels in machine learning using convolutional neural networks as an example. The computation of convolution layers in deep neural networks typically rely on high performance routines that trade space for time by using additional memory (either for packing purposes or required as part of the algorithm) to improve performance. The contributions are:

- We demonstrate that a high performance direct convolution implementation can not only eliminate all additional memory overhead, but also attain higher performance than the expert-implemented matrix-matrix-multiplication based convolution.
- We also show that our implementation scales to larger number of processors without degradation in performance as our implementation exploits the dimension of the kernel that has the highest amount of parallelism. In contrast, current high performance matrix-multiply based implementations do not scale as well to a larger number of processors.

In chapter 4 and chapter 5, we investigate the optimization of sparse kernels. Sparse kernels present a different set of challenges compared to dense computations. We present the technique that can improve the data-parallelism of the irregular computations of sparse kernels, as well as investigating the effects of algorithm and data structure choices on the performances. Overall, in these chapters, we demonstrate how to accelerate the notoriously irregular sparse operations on modern processors (i.e., CPU, GPU).

- We primarily focus on Sparse Matrix multiplication with another sparse matrix (SpGEMM) as it is a dominant and versatile component used in social science analytics tasks very low fractions of peak processor performance.
- We demonstrate how the implementation of SpGEMM can be broken down into different combinations of primitive operations such as

merge/join, and explored how the different implementation choices can affect performance under varying input.

6.2 Takeaways and Reflections

Algorithm and implementation codesign

Algorithms. Algorithm choice is an important factor that constitute the foundation for computation complexity. For sparse applications, we demonstrate that there are no one-size-fits-all algorithms, and that algorithms should adapt for different input.

Implementations. Despite the fact that the choice of algorithms is a fundamental factor affecting the total complexity, it is not the only determinant factor. Implementation has an equally important influence. In performance optimization and tunings, implementation sometimes can overtune the algorithm choices — some low complexity may end up performing poorly on modern architectures. The specific evidence have been demonstrated in both our dense and sparse works — in the optimizations of dense convolution neural networks, we demonstrated that the algorithms with lower complexities such as FFT-based implementations, Winograd-based implementations do not have advantages over direct convolution methods, in both aspects of memory and performance. This is because despite of low complexity, FFT and Winograd implementations have computational pattern that fails to take the most out of the computational resources on today’s processors, and also they come

with extra memory overheads which can result in inefficiencies. Similarly for the sparse applications, we also see that sometimes all-pair comparisons (the inner-product based method) that have higher complexity can be faster than merge-based methods on real systems. The results obtained in this dissertation emphasize the importance of algorithm and implementation co-designs when we are tuning the performance on modern processors.

AI vs. scientific computational motifs

Scientific computational kernels are appearing in the domains of linear algebra, signal processing, differential equations, etc. There are decades of research investigating the computational motifs in the above domains. AI and machine learning as an emerging domain, shares similarities as well as differences with the traditional scientific computing in computational motifs. Broadly speaking, the motifs of these two domains can be categorized based on their access patterns as dense and sparse. For example, the neural networks are similar as traditional linear algebra computations in scientific domains, because they are both dense and regular. Graph or data analytic tasks in learning and data science domains are similar to the traditional sparse scientific problems such as differential systems, in that they both work on sparse data that has random and irregular computational patterns. Despite the dense/sparse similarities, however, when it comes the computation details, AI/machine learning tasks and scientific tasks are not exactly the same. For example, the data they are working on can have different dimensions, the computation can have dif-

ferent orderings, or the data types can be different. All of these factors can result in different data reuse, blocking choices, etc. The optimized implementation strategies therefore would be different. Thus the traditional libraries designed for scientific applications can not be directly used for the emerging AI applications. But on the other hand, the general lessons/methods learned from optimizing scientific applications can be adapted to the emerging domains when it comes to the general design principles, performance models, optimization approaches.

Performance tuning

Performance comes from co-designs of algorithms and implementations. To attain high performance on a given platform, traditional approaches can be divided into two categories: the auto-tuning approach which treats the application as a black-box and enumerates all the possible design parameters to find the best configuration. Another approach is analytical modeling. The analytical modeling approach constructs the model for the underlying systems, and attempts to derive the best configuration based on the analytics. The auto-tuning approach can be inefficient, because there can be explosive range of search space which is impractical to enumerate, especially with the growing complexity of nowadays hardware designs. In contrary, an analytical model can leverage expert knowledge to alleviate the search burden. But for new applications/systems, it may fail to accurately capture the systems or application characteristics. Additionally, it can not incorporate the actual system

response. Model verification can be a nontrivial task. Therefore, a combination of analytical modeling and auto-tuning may be a more practical choice for the future performance tuning research.

6.3 Future Directions

We foresee a variety of directions and topics that could potentially benefit from the work and ideas in this dissertation. In the last section, we provide discussion and suggestions on future search opportunities.

Accelerating diverse machine learning “motifs”

With the booming growth of machine learning research, learning models are vigorously progressing. New learning models are coming out one of another, and existing models are also constantly updated and optimized, and evolving towards different scenarios. For example, a lot of research work have proposed different ways to prune and compress CNN models. Some work are focusing fine-grained pruning that prune out the neurons of small values in the network. Fine-grained pruning can result in irregular layer structures where nonzeros show up randomly. Some work are focusing on regular pruning where they prune out an entire set of rows/columns/layers of neurons. In addition, there are also a great number of research over these years on how to compress the networks by using fix-point numerical numbers. On top of this, the network model structures are also evolving to adapt to different domains. For example,

there are 3D neural networks, or networks with mixed filter shapes. All these models would require more acceleration and performance tuning. We believe there are a plenty of opportunities for hardware/software codesigns. And the acceleration would need to consider the hardware features and computational patterns. We suspect that these followups can benefit from the optimization principles discussed in this thesis.

Implications for AI/ML accelerator design

There is an increasing number of work focused on the design of accelerators for learning applications. Moreover, in industry, hardware companies have released new architectures/accelerator engines for learning applications. Many of these accelerator engine designs are intended for matrix or tensor operations — as matrix/tensor multiplication is believed to be at the heart of machine learning models such as neural networks. Our work in this dissertation can provide insights for the accelerator designs. We have provided the model-based analysis on the convolution neural networks, demonstrated the data reuse pattern, parallelism design. These can be directly applied when designing the memory hierarchies, and the execution engines.

Implications for cross-stack system optimizations

Over recent years, a handful of machine learning libraries and systems have been built and released such as pytorch, tensorflow, graphlab, etc. An efficient system would benefit from cross-stack optimizations. The reasons are

two-fold: cross-stack optimization enables the maximum customization from hardware, compiler to algorithm, and the thorough customization ensures the resources are being fully utilized. Secondly, cross-stack design can enable performance portability. Performance portability across platforms and processors is a nontrivial task, because different platforms and processors can have distinct architecture characteristics, and there is hardly a one-size-fits-all implementation that can overcome all the differences. Secondly, different platforms and systems have their specific languages and instructions. Programs optimized for one platform cannot be directly migrated to another system. But in our work we have demonstrated that the implementation strategies can share some similarities across processors (different GPUs, and also across CPU and GPU). Moreover in the optimization for sparse applications, we demonstrate that how a computation kernel (SpGEMM) can be broken down into a series of primitive operations (join, union, etc), which can result in a variety of implementations. Therefore, there is a chance for the use of high-level DSL that is capable to express different algorithms, and transform the algorithm choice to the lower-level implementation methods, and then platform-specific code. In addition, the layered DSL transformation enables the separation of low-level implementations from high-level algorithms, facilitating the process of performance portability across platforms.

Bibliography

- [1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, “The landscape of parallel computing research: A view from berkeley,” 2006. [1.2.1](#)
- [2] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, “Graphmat: High performance graph analytics made productive,” *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015. [1.2.2](#), [3.2.1](#)
- [3] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011. [1.2.2](#), [3.2.1](#)
- [4] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, “Emptyheaded: A relational engine for graph processing,” *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 4, p. 20, 2017. [1.2.2](#), [3.2.1](#)
- [5] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing frame-

- work for shared memory,” in *ACM Sigplan Notices*, vol. 48, pp. 135–146, ACM, 2013. [1.2.2](#), [3.2.1](#)
- [6] D. Nguyen, A. Lenharth, and K. Pingali, “A lightweight infrastructure for graph analytics,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 456–471, ACM, 2013. [1.2.2](#), [3.2.1](#)
- [7] J. Zhang, F. Franchetti, and T. M. Low, “High performance zero-memory overhead direct convolutions,” in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pp. 5771–5780, 2018. [1.2.2](#), [1.5](#), [3.2.1](#)
- [8] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, “A set of level 3 basic linear algebra subprograms: Model implementation and test programs,” *ACM Trans. Math. Soft.*, vol. 16, pp. 18–28, March 1990. [1.2.4](#), [2.2](#), [2.3](#)
- [9] F. G. Van Zee, *libflame: The Complete Reference*. www.lulu.com, 2012. [1.2.4](#)
- [10] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, “Spiral: Code generation for dsp transforms,” *Proceedings of the IEEE*, vol. 93, pp. 232–275, Feb 2005. [1.2.4](#)

- [11] M. Frigo and S. G. Johnson, “The design and implementation of fftw3,” *Proceedings of the IEEE*, vol. 93, pp. 216–231, Feb 2005. [1.2.4](#)
- [12] R. C. Whaley and J. J. Dongarra, “Automatically tuned linear algebra software,” in *Proceedings of SC’98*, 1998. [1.2.4](#), [2.7](#)
- [13] H. Kwak, C. Lee, H. Park, and S. Moon, “What is twitter, a social network or a news media?,” in *Proceedings of the 19th International Conference on World Wide Web*, WWW ’10, (New York, NY, USA), p. 591–600, Association for Computing Machinery, 2010. [1.4](#), [5.2](#)
- [14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “Eie: efficient inference engine on compressed deep neural network,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016. [1.4](#), [5.2](#)
- [15] M. S. Waterman, *Introduction to computational biology: maps, sequences and genomes*. CRC Press, 1995. [1.4](#), [5.2](#)
- [16] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, *et al.*, “The youtube video recommendation system,” in *Proceedings of the fourth ACM conference on Recommender systems*, pp. 293–296, 2010. [1.4](#), [5.2](#)
- [17] J. Zhang, D. G. Spampinato, S. McMillan, and F. Franchetti, “Preliminary exploration of large-scale triangle counting on shared-memory multicore system,” in *HPEC*, pp. 1–6, 2018. [1.5](#)

- [18] J. Zhang, Y. Lu, D. G. Spampinato, and F. Franchetti, “Fesia: A fast and simd-efficient set intersection approach on modern cpus,” in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1465–1476, 2020. [1.5](#)
- [19] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, ACM, 2014. [2.2](#)
- [20] M. Cho and D. Brand, “MEC: memory-efficient convolution for deep neural network,” *CoRR*, vol. abs/1706.06873, 2017. [2.2](#), [2.3](#)
- [21] <http://www.openblas.net>, 2015. [2.2](#), [2.3](#), [2.6.1](#)
- [22] M. Schuster, “Speech recognition for mobile devices at google,” in *Pacific Rim International Conference on Artificial Intelligence*, pp. 8–10, Springer, 2010. [2.2](#)
- [23] K. H. Lee and N. Verma, “A low-power processor with configurable embedded machine-learning accelerators for high-order and adaptive analysis of medical-sensor signals,” *IEEE Journal of Solid-State Circuits*, vol. 48, no. 7, pp. 1625–1637, 2013. [2.2](#)
- [24] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, “A 240 g-ops/s mobile coprocessor for deep neural networks,” in *Proceedings*

- of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 682–687, 2014. [2.2](#)
- [25] A. Dundar, J. Jin, V. Gokhale, B. Krishnamurthy, A. Canziani, B. Martini, and E. Culurciello, “Accelerating deep neural networks on mobile processor with embedded programmable logic,” [2.2](#)
 - [26] J. Zhang, T. M. Low, Q. Guo, and F. Franchetti, “A 3d-stacked memory manycore stencil accelerator system,” 2014. [2.2](#)
 - [27] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, “Compression of deep convolutional neural networks for fast and low power mobile applications,” *arXiv preprint arXiv:1511.06530*, 2015. [2.2](#)
 - [28] Y. Gong, L. Liu, M. Yang, and L. Bourdev, “Compressing deep convolutional networks using vector quantization,” *arXiv preprint arXiv:1412.6115*, 2014. [2.2](#)
 - [29] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “Eie: efficient inference engine on compressed deep neural network,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 243–254, IEEE Press, 2016. [2.2](#)
 - [30] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, “Fast convolutional nets with fbfft: A gpu performance evaluation,” *arXiv preprint arXiv:1412.7580*, 2014. [2.3](#)

- [31] M. Mathieu, M. Henaff, and Y. LeCun, “Fast training of convolutional networks through ffts,” *CoRR*, vol. abs/1312.5851, 2013. 2.3
- [32] M. Dukhan, “Nnpack.” <https://github.com/Maratyszczka/NNPACK>. 2.3, 2.6.1
- [33] K. Goto and R. van de Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Trans. Math. Soft.*, vol. 34, pp. 12:1–12:25, May 2008. 2.3
- [34] F. G. Van Zee and R. A. van de Geijn, “Blis: A framework for rapidly instantiating blas functionality,” *ACM Trans. Math. Softw.*, vol. 41, pp. 14:1–14:33, June 2015. 2.3
- [35] J. A. Gunnels, G. M. Henry, and R. A. Van De Geijn, “A family of high-performance matrix multiplication algorithms,” in *International Conference on Computational Science*, pp. 51–60, Springer, 2001. 2.3
- [36] T. M. Smith, R. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. V. Zee, “Anatomy of high-performance many-threaded matrix multiplication,” in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14*, (Washington, DC, USA), pp. 1049–1059, IEEE Computer Society, 2014. 2.3
- [37] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, “Analytical modeling is enough for high-performance blis,” *ACM Transactions*

- on *Mathematical Software (TOMS)*, vol. 43, no. 2, p. 12, 2016. [2.4.1](#), [2.4.1](#), [2.7](#)
- [38] M. Wolfe, “Iteration space tiling for memory hierarchies,” in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, (Philadelphia, PA, USA), pp. 357–361, Society for Industrial and Applied Mathematics, 1989. [2.4.1](#)
 - [39] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015. [2.5.3](#)
 - [40] R. Veras, D. T. Popovici, T. M. Low, and F. Franchetti, “Compilers, hands-off my hands-on optimizations,” in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, WPMVP ’16, (New York, NY, USA), pp. 4:1–4:8, ACM, 2016. [2.6.1](#)
 - [41] Intel, “Math Kernel Library.” <https://software.intel.com/en-us/intel-mkl>, 2015. [2.6.1](#)
 - [42] caffe2, “caffe2.” <https://caffe2.ai/>. [2.6.1](#)
 - [43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012. [2.6.1](#)

- [44] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, June 2015. [2.6.1](#)
- [45] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014. [2.6.1](#)
- [46] J. Bilmes, K. Asanović, C. whye Chin, and J. Demmel, “Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology,” in *Proceedings of International Conference on Supercomputing*, (Vienna, Austria), July 1997. [2.7](#)
- [47] K. Yotov, X. Li, M. J. Garzarán, D. Padua, K. Pingali, and P. Stodghill, “Is search really necessary to generate high-performance BLAS?,” *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”*, vol. 93, no. 2, 2005. [2.7](#)
- [48] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka, “Tensor contractions with extended blas kernels on cpu and gpu,” in *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*, pp. 193–202, IEEE, 2016. [2.7](#)
- [49] Nvidia, “cuBLAS.” <https://developer.nvidia.com/cublas>, 2017. [2.7](#)
- [50] R. B. Bapat, *Graphs and Matrices*. Universitext, 2014. [3.1](#)

- [51] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. USA: Society for Industrial and Applied Mathematics, 2011. [3.1](#)
- [52] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, “Standards for graph algorithm primitives,” in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–2, Sep. 2013. [3.1](#)
- [53] <http://math.nist.gov/spblas/>. [3.1](#)
- [54] J. Kepner, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, and H. Tufo, “Enabling massive deep neural networks with the graphblas,” *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep 2017. [3.1](#)
- [55] N. A. Nystrom, M. J. Levine, R. Z. Roskies, and J. R. Scott, “Bridges: A uniquely flexible hpc resource for new communities and data analytics,” in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, XSEDE ’15, (New York, NY, USA), pp. 30:1–30:8, ACM, 2015. [3.2.1](#), [3.2.7](#)
- [56] S. Samsi, V. Gadepally, M. B. Hurley, M. Jones, E. K. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and

- J. Kepner, “Graphchallenge.org: Raising the bar on graph analytic performance,” *CoRR*, vol. abs/1805.09675, 2018. 3.2.2
- [57] T.-M. Low, V. Rao, M. Lee, T. Popovici, F. Franchetti, and S. McMillan, “First look: Linear algebra-based triangle counting without matrix multiplication,” 3.2.2
- [58] T. Schank and D. Wagner, “Finding, counting and listing all triangles in large graphs, an experimental study,” in *International workshop on experimental and efficient algorithms*, pp. 606–609, Springer, 2005. 3.2.2
- [59] J. Shun and K. Tangwongsan, “Multicore triangle computations without tuning,” in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pp. 149–160, IEEE, 2015. 3.2.3
- [60] D. Lemire, L. Boytsov, and N. Kurz, “SIMD compression and the intersection of sorted integers,” *Software: Practice and Experience*, vol. 46, no. 6, pp. 723–749, 2016. 10
- [61] B. Schlegel, T. Willhalm, and W. Lehner, “Fast sorted-set intersection using SIMD instructions,” in *ADMS@ VLDB*, pp. 1–8, 2011. 10
- [62] “Fast intersection of sorted lists using SSE instructions.”
["https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/"](https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/). 10
- [63] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R.

- Scott, and N. Wilkins-Diehr, “Xsede: Accelerating scientific discovery,” *Computing in Science and Engineering*, vol. 16, pp. 62–74, Sept.-Oct. 2014. [3.2.7](#)
- [64] W.-D. N, S. Sanielevici, J. Alameda, J. Cazes, L. Crosby, M. Pierce, and R. Roskies in *High Performance Computer Applications 6th International Conference, ISUM 2015, Revised Selected Papers Gitler, Isidoro, Klapp, Jaime (Eds.)*, pp. 3–13, Springer International Publishing, 2015. [3.2.7](#)
- [65] J. Kepner, S. Samsi, W. Arcand, D. Bestor, B. Bergeron, T. Davis, V. Gadepally, M. Houle, M. Hubbell, H. Jananthan, *et al.*, “Design, generation, and validation of extreme scale power-law graphs,” *arXiv preprint arXiv:1803.01281*, 2018. [3.2.7](#)
- [66] R. Pearce, “Triangle counting for scale-free graphs at scale in distributed memory,” [3.2.7](#)
- [67] H. Inoue, M. Ohara, and K. Taura, “Faster set intersection with SIMD instructions by reducing branch mispredictions,” *PVLDB*, vol. 8, no. 3, pp. 293–304, 2014. [4.1](#), [4.2](#), [4.3.2](#), [4.8.1](#)
- [68] D. Lemire, L. Boytsov, and N. Kurz, “SIMD compression and the intersection of sorted integers,” *Softw., Pract. Exper.*, vol. 46, no. 6, pp. 723–749, 2016. [4.1](#), [4.2](#), [4.3.2](#), [4.8.1](#)
- [69] B. Schlegel, T. Willhalm, and W. Lehner, “Fast sorted-set intersection

- using SIMD instructions,” in *ADMS@VLDB*, pp. 1–8, 2011. [4.1](#), [4.2](#), [4.3.2](#)
- [70] B. Ding and A. C. König, “Fast set intersection in memory,” *PVLDB*, vol. 4, no. 4, pp. 255–266, 2011. [4.1](#), [4.2](#), [4.3.1](#), [4.3.2](#), [4.4.4](#)
- [71] L. A. Barroso, J. Dean, and U. Hölzle, “Web search for a planet: The Google cluster architecture,” *IEEE Micro*, no. 2, pp. 22–28, 2003. [4.2](#)
- [72] S. Chu and J. Cheng, “Triangle listing in massive networks and its applications,” in *KDD*, pp. 672–680, 2011. [4.2](#)
- [73] X. Hu, Y. Tao, and C. Chung, “Massive graph triangulation,” in *SIGMOD Conference*, pp. 325–336, 2013. [4.2](#)
- [74] N. Wang, J. Zhang, K. Tan, and A. K. H. Tung, “On triangulation-based dense neighborhood graphs discovery,” *PVLDB*, vol. 4, no. 2, pp. 58–68, 2010. [4.2](#)
- [75] W. Han, J. Lee, and J. Lee, “Turbo_{iso}: towards ultrafast and robust subgraph isomorphism search in large graph databases,” in *SIGMOD Conference*, pp. 337–348, 2013. [4.2](#)
- [76] D. Eppstein, M. Löffler, and D. Strash, “Listing all maximal cliques in sparse graphs in near-optimal time,” in *ISAAC*, pp. 403–414, 2010. [4.2](#)
- [77] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, “Online search of overlapping communities,” in *SIGMOD Conference*, pp. 277–288, 2013. [4.2](#)

- [78] S. Han, L. Zou, and J. X. Yu, “Speeding up set intersections in graph algorithms using SIMD instructions,” in *SIGMOD Conference*, pp. 1587–1602, 2018. [4.2](#)
- [79] I. Katsov, “Fast intersection of sorted lists using SSE instructions.” <https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/>, 2012. [4.2](#), [4.8.1](#)
- [80] G. E. Blelloch and M. Reid-Miller, “Fast set operations using treaps,” in *SPAA*, pp. 16–26, 1998. [4.3.1](#)
- [81] W. Pugh, “A skip list cookbook,” tech. rep., 1990. [4.3.1](#)
- [82] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O’Hara, F. Saint-Jacques, and G. S. Y. Kai, “Roaring bitmaps: Implementation of an optimized software library,” *Softw., Pract. Exper.*, vol. 48, no. 4, pp. 867–895, 2018. [4.3.1](#)
- [83] E. D. Demaine, A. López-Ortiz, and J. I. Munro, “Adaptive set intersections, unions, and differences,” in *SODA*, pp. 743–752, 2000. [4.3.1](#)
- [84] J. L. Bentley and A. C.-C. Yao, “An almost optimal algorithm for unbounded searching,” *Information processing letters*, vol. 5, no. SLAC-PUB-1679, 1976. [4.3.2](#), [4.8.1](#)
- [85] <http://fimi.cs.helsinki.fi/data/>. [4.8.6](#)

- [86] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection.” <http://snap.stanford.edu/data>, June 2014. 4.8.6, 5.8.7
- [87] D. Nguyen, A. Lenharth, and K. Pingali, “A lightweight infrastructure for graph analytics,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 456–471, 2013. 5.2
- [88] A. Kyrola, G. Blelloch, and C. Guestrin, “Graphchi: Large-scale graph computation on just a pc,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI’12, (USA), p. 31–46, USENIX Association, 2012. 5.2
- [89] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, “Graphlab: A new framework for parallel machine learning,” 2014. 5.2
- [90] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens *ACM Trans. Parallel Comput.*, August. 5.2
- [91] C. Yang, A. Buluç, and J. D. Owens, “Design principles for sparse matrix multiplication on the gpu,” in *Euro-Par 2018: Parallel Processing* (M. Aldinucci, L. Padovani, and M. Torquati, eds.), (Cham), pp. 672–687, Springer International Publishing, 2018. 5.2
- [92] C. Yang, A. Buluç, and J. D. Owens, “Implementing push-pull efficiently

- in graphblas,” ICPP 2018, (New York, NY, USA), Association for Computing Machinery, 2018. [5.2](#)
- [93] C. Yang, Y. Wang, and J. D. Owens, “Fast sparse matrix and sparse vector multiplication algorithm on the gpu,” in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 841–847, 2015. [5.2](#)
- [94] G. Huang, G. Dai, Y. Wang, and H. Yang, “Ge-spmv: General-purpose sparse matrix-vector multiplication on gpus for graph neural networks,” 2020. [5.2](#)
- [95] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, “Exploiting locality in graph analytics through hardware-accelerated traversal scheduling,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, 2018. [5.2](#)
- [96] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “Extensor: An accelerator for sparse tensor algebra,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO ’18*, (New York, NY, USA), p. 319–333, Association for Computing Machinery, 2019. [5.2](#)
- [97] Z. Zhang, H. Wang, S. Han, and W. J. Dally, “Sparch: Efficient architecture for sparse matrix multiplication,” in *26th IEEE International*

Symposium on High Performance Computer Architecture (HPCA), 2020.

5.2

- [98] A. Azad, A. Buluç, and J. Gilbert, “Parallel triangle counting and enumeration using matrix algebra,” in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 804–811, 2015. [5.3.2](#)
- [99] A. Azad and A. Buluc, “A work-efficient parallel sparse matrix-sparse vector multiplication algorithm,” *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017. [5.3.4](#)
- [100] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, “Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication,” *SIAM Journal on Scientific Computing*, vol. 38, p. C624–C651, Jan 2016. [5.3.4](#)
- [101] D. Merrill and M. Garland, “Merge-based parallel sparse matrix-vector multiplication,” in *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 678–689, 2016. [5.3.4](#)
- [102] D. Merrill, M. Garland, and A. Grimshaw, “Scalable gpu graph traversal,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’12, (New York, NY, USA), p. 117–128, Association for Computing Machinery, 2012. [5.3.4](#)
- [103] C. Yang, A. Buluç, and J. D. Owens, “Implementing push-pull efficiently

- in graphblas,” *Proceedings of the 47th International Conference on Parallel Processing*, Aug 2018. [5.3.4](#)
- [104] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, “To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC’17)*, ACM, Jun. 2017. [5.3.4](#)
- [105] S. Beamer, K. Asanovic, and D. Patterson, “Direction-optimizing breadth-first search,” in *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–10, 2012. [5.3.4](#)
- [106] S. Beamer, K. Asanovic, and D. Patterson, “Locality exists in graph processing: Workload characterization on an ivy bridge server,” in *2015 IEEE International Symposium on Workload Characterization*, pp. 56–65, 2015. [5.3.4](#)
- [107] C. Yang, A. Buluç, and J. D. Owens, “Implementing push-pull efficiently in graphblas,” *ICPP 2018*, (New York, NY, USA), Association for Computing Machinery, 2018. [5.3.4](#)
- [108] O. Green, M. Dukhan, and R. Vuduc, “Branch-avoiding graph algorithms,” *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, Jun 2015. [5.3.4](#)

- [109] C. Yang, A. Buluc, and J. D. Owens, “Graphblast: A high-performance linear algebra-based graph framework on the gpu,” 2019. [5.3.4](#)
- [110] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, “The tensor algebra compiler,” *Proc. ACM Program. Lang.*, vol. 1, pp. 77:1–77:29, October 2017. [5.3.4](#)
- [111] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe, “taco: A tool to generate tensor algebra kernels,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 943–948, Oct 2017. [5.3.4](#)
- [112] S. Chou, F. Kjolstad, and S. Amarasinghe, “Format abstraction for sparse tensor algebra compilers,” *Proc. ACM Program. Lang.*, vol. 2, pp. 123:1–123:30, October 2018. [5.3.4](#)
- [113] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe, “Tensor algebra compilation with workspaces,” pp. 180–192, 2019. [5.3.4](#)
- [114] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Regul, N. Sakharnykh, *et al.*, “Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods,” *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. S602–S626, 2015. [5.3.4](#)
- [115] W. Liu and B. Vinter, “A framework for general sparse matrix–matrix

- multiplication on gpus and heterogeneous processors,” *Journal of Parallel and Distributed Computing*, vol. 85, p. 47–61, Nov 2015. [5.3.4](#)
- [116] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, “Speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis,” PPOPP ’20, (New York, NY, USA), p. 362–375, Association for Computing Machinery, 2020. [5.3.4](#)
- [117] P. Jiang, C. Hong, and G. Agrawal, “A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus,” in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’20, (New York, NY, USA), p. 376–388, Association for Computing Machinery, 2020. [5.3.4](#)
- [118] F. G. Gustavson, “Two fast algorithms for sparse matrices: Multiplication and permuted transposition,” *ACM Trans. Math. Softw.*, vol. 4, p. 250–269, Sept. 1978. [5.3.4](#)
- [119] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, “Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication,” in *Knowledge Discovery in Databases: PKDD 2005* (A. M. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, eds.), (Berlin, Heidelberg), pp. 133–145, Springer Berlin Heidelberg, 2005. [5.4](#)
- [120] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, “Kronecker graphs: An approach to modeling networks,” *J. Mach. Learn. Res.*, vol. 11, p. 985–1042, March 2010. [5.4](#)

- [121] https://www.boost.org/doc/libs/1_68_0/libs/graph/doc/plod_generator.html. 5.8.1
- [122] <https://moderngpu.github.io/library.html>. 5.8.1
- [123] <https://docs.nvidia.com/cuda/cusparses/index.html>. 5.8.1