論文 / 著書情報
Article / Book Information

題目(和文)	メニーコアアーキテクチャにおける疎行列計算の性能最適化
Title(English)	Performance Optimization of Sparse Matrix Kernels for Many-core Architectures
著者(和文)	長坂 侑亮
Author(English)	Yusuke Nagasaka
出典(和文)	学位:博士(理学), 学位授与機関:東京工業大学, 報告番号:甲第11066号, 授与年月日:2019年3月26日, 学位の種別:課程博士, 審査員:松岡 聡,遠藤 敏夫,増原 英彦,額田 彰,横田 理央
Citation(English)	Degree:Doctor (Science), Conferring organization: Tokyo Institute of Technology, Report number:甲第11066号, Conferred date:2019/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

TOKYO INSTITUTE OF TECHNOLOGY

Performance Optimization of Sparse Matrix Kernels for Many-core Architectures

by
Yusuke Nagasaka

A thesis submitted in partial fulfillment
for the degree of Doctor of Science
in the
Department of Mathematical and Computing Science

Supervisor: Prof. Satoshi Matsuoka

February, 2019

Abstract

Many-core architectures such as Graphic Processing Units (GPUs) and Many Integrated Core architectures (MICs) are becoming mainstream in high performance computing platform, accelerating the performance of various kinds of applications such as simulations, Big Data analytics and Machine Learning. Currently, areas for improving application performance on many-core processors include operations on sparse matrices, which are mainly occupied by zero elements. Sparse matrices are used in various fields, such as the problem matrix in simulations or the adjacency matrix in graph processing, and it is crucial to accelerate the kernels for processing sparse matrices in a wide range of applications. However, the kernels for sparse matrix operations on many-core processors still have several performance issues.

There are two main issues with optimizing sparse matrix kernels on many-core processors. The first issue deals with memory access. A sparse matrix is usually compressed using sparse matrix format, holding only non-zero elements with value and indices information. This increases byte per flop ratio while the byte per flop ratio of many-core processors is relatively low compared to modern multi-core CPUs, resulting in much waste of computing capability of many-core processors. Furthermore, a kernel based on a sparse matrix format yields indirect memory access resulting in frequent cache misses, especially on many-core processors, whose cache capacity is small. The second issue is concerned with load balancing. The computation complexity is determined by the pattern or number of non-zero elements in the sparse matrix. Non-sophisticated task assignment causes load imbalance wasting the massive parallelism of many-core architectures. In addition, another load-balancing issue emerges when processing many kernels for sparse matrices in parallel.

Firstly, for sparse matrix vector multiplication (SpMV), we propose the Non-Uniformly Segmented (NUS) format and the Adaptive Multi-level Blocking (AMB) format to tackle the memory access and load balancing issues of sparse matrix operations on many-core architectures. By dividing the matrix along the column, the memory access to input vector elements attains better cache locality. In the AMB format, furthermore, the number of bits for holding the indices of non-zero elements is reduced, and the contiguous non-zero elements in the matrix is treated as “one block” in order to reduce the amount of bytes accessed, thereby alleviating high byte per flop ratio. Performance evaluation using

a variety of sparse matrix datasets shows that our proposed sparse matrix formats and algorithms for SpMV achieve significant speedups of up to 1.4x compared to existing state-of-the-art algorithms.

Secondly, for sparse general matrix-matrix multiplication (SpGEMM), we propose a hash table-based algorithm optimized for GPU with perfect load balance. In SpGEMM, memory allocation is also one of the issues due to the large memory required for temporary use and storing output matrix despite the limited memory capacity of many-core processors. Our approach efficiently uses shared memory on the GPU for hash table, and the evaluation on NVIDIA GPU shows significant speedups of up to 4.4x compared to existing approaches for GPU. The proposed optimization strategy for SpGEMM has also been applied to Intel Xeon Phi. we executed microbenchmarks on Intel Xeon Phi in order to expose the performance bottlenecks of SpGEMM, and optimize hash-based and heap-based approaches for Intel Xeon Phi. We built the performance model of SpGEMM and the guide for selecting the best algorithm for specific input and scenario based on empirical analysis.

Thirdly, for processing many sparse matrix multiplications (SpMMs) mainly between small matrices, we propose Batched SpMM. Recently proposed Graph Convolutional Networks (GCN) can deal with the graph structure as input of the neural networks. In GCN applications, the graph structure is expressed as adjacency matrix, and many sparse matrix multiplication kernels are executed. However, the graph sizes are often very small and the parallelism of many-core processors is hardly exploited. To improve the occupancy of many-core processors and achieve high performance, the Batched SpMM executes tens or hundreds of SpMM kernels with a single kernel launch. The evaluation results show that the Batched SpMM largely accelerates the GCN application and the speedup in training is up to 1.59x.

This thesis provides several contributions to performance optimizations across a wide-area of computer science from traditional simulations to Bigdata and Machine Learning related to the kernels for sparse matrix.

Acknowledgement

I would like to express my deep appreciation for everyone who made this thesis possible.

First of all, I would like to sincerely thank my supervisor, Prof. Satoshi Matsuoka. His significant advice has encouraged my research, and a lot of invaluable opportunities he brought to me allow me to make progress to reach here. I am grateful to my mentor, Prof. Akira Nukada, for his kind and magnificent support. I never complete this thesis without his help. I would like to express my appreciation to the members of Tokyo Institute of Technology, especially Matsuoka lab for providing excellent help for research and an enjoyable time.

I would like to thank my collaborators, Dr. John Shalf, Dr. Ken-ichi Miura and Dr. Aydın Buluç at Lawrence Berkeley National Laboratory and Prof. Azad Ariful at Indiana University, for providing me a special opportunity to work as internship and accepting me. This collaboration and experience let me broaden my research dramatically. I show my appreciation to the members at Lawrence Berkeley National Laboratory for invaluable three months during my internship and providing excellent environment to grow up my capability, not limited to research itself. I would like to thank my collaborators, Prof. Ryosuke Kojima at Kyoto University and Mr. Akira Naruse at NVIDIA, for providing a seed and significant support for the research. This collaboration is a great opportunity which allows me to see my experience so far can be extended to another research area.

I am grateful to my friends who made my journey enjoyable. Special thanks to the alumni of the club where I belonged in my undergraduate age, the members met through orienteering, and those who have been encouraging me.

Finally, I would like to show my sincere thankful to my family.

Yusuke Nagasaka
February, 2019

Publications

Conference (Refereed)

- Yusuke Nagasaka, Akira Nukada and Satoshi Matsuoka, "Cache-aware sparse matrix formats for Kepler GPU", 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), Hsinchu, Taiwan, December 2014.
- Yusuke Nagasaka, Akira Nukada and Satoshi Matsuoka, "Adaptive Multi-level Blocking Optimization for Sparse Matrix Vector Multiplication on GPU", International Conference on Computational Science 2016 (ICCS 2016), San Diego, California, USA, June 2016.
- Yusuke Nagasaka, Akira Nukada and Satoshi Matsuoka, "High-performance and Memory-saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU", International Conference on Parallel Processing 2017 (ICPP 2017), Bristol, UK, August 2017.

Workshop (Refereed)

- Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad and Aydın Buluç, "High-performance sparse matrix-matrix products on Intel KNL and multicore architectures", International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), held in conjunction with International Conference on Parallel Processing 2018 (ICPP 2018), Eugene, Oregon, USA, 2018.

Posters

- Yusuke Nagasaka, Akira Nukada and Satoshi Matsuoka, "Cache-aware Sparse Matrix Format for GPU", International Superconputing Conference (ISC'14) HPC in Asia Posters, Leipzig, Germany, June 2014.

- Yusuke Nagasaka, Akira Nukada and Satoshi Matsuoka, “Multi-Level Blocking Optimization for Fast Sparse Matrix Vector Multiplication on GPUs”, The International Conference for High Performance Computing, Networking, Storage and Analysis (SC15) Technical Program Posters, Austin, Texas, USA, November 2015.
- Yusuke Nagasaka, “Fast Sparse Matrix Vector Multiplication with Highly-Compressed Sparse Format”, GPU Technology Conference (GTC2016), San Jose, CA, USA, April 2016.
- Yusuke Nagasaka, Akira Nukada and Satoshi Matsuoka, “Fast Sparse General Matrix-Matrix Multiplication on GPU with Low Memory Usage”, The International Conference for High Performance Computing, Networking, Storage and Analysis (SC16) Technical Program Posters, Salt Lake City, Utah, USA, November 2016.
- Yusuke Nagasaka, “Fast and Memory-saving SpGEMM Algorithm for New Pascal Generation GPU”, GPU Technology Conference (GTC2017), San Jose, CA, USA, May 2017.

Talks

- Yusuke Nagasaka “Exploiting GPU Caches in Sparse Matrix Vector Multiplication”, GPU Technology Conference (GTC2015), San Jose, CA, USA, March 2015.
- Yusuke Nagasaka, Akira Nukada and Satoshi Matsuoka, Ariful Azad and Aydın Buluç, “Efficient Sparse General Matrix-Matrix Multiplication Algorithms for Many-Core Processors”, SIAM PP 2018, Tokyo, Japan, March 2018.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the decades following the invention of the first computer, there is no doubt a great deal of technological change and benefits have emerged. As computers are requiring more computing capability, systems are becoming more dense and necessitating more parallelism to attain better performance. What is boosting this stream is the emergence of many-core architectures, which implement tens or hundreds of computing cores with low frequency. The Graphic Processing Unit (GPU) is one such many-core architecture. GPUs are designed to deal with graphic processing where each operation itself is trivial, but is needed to execute in parallel. They are used for not only graphic purposes, but general ones as well. Much effort for utilizing many-core processors including GPUs enables computers to acquire more scalability and computing power, and currently many-core processors have become mainstream in high performance computing platform.

During the transition of computer architectures, what computers can offer to the world also largely changes. Using today's hardware, enormous computations can be completed in reasonable time, so simulations in the scientific and engineering fields are becoming more sophisticated and accurate. Two other significant changes came about as well - the first of which is Big Data. The computer is a common tool in human society, and given our growing population and reliance on technology, much more data is being accumulated. By utilizing accumulated data, new values have been emerging, and hence new findings and knowledges are discovered. The other change comes from the growth of computing power happening in Machine Learning field. After the "Deep Learning" approach was proposed in 2012 [1], the field has enjoyed phenomenal progress. What supports this progress is high computing capability with many-core processors.

In these progress of computer science, the applications including irregular memory access have

been emerging, and it is important to achieve high performance with many-core processors. One of the common themes among these changing applications is the sparse matrix, where most elements are zero. Sparse matrices appear, for example, as a system of simultaneous equations discretized by implicit solvers in simulations, or as adjacency matrix to express a graph structure for processing Big Data. Currently, sparse matrices also appear in the machine learning field as the expression of parameters sparsified by network pruning [2, 3], or to deal with data which has a graph structure in a graph convolutional network (GCN) [4]. It is crucial to know how to treat sparse matrices found in a variety of areas as a key component.

1.2 Problem Statement

It is crucially important to accelerate data intensive applications including irregular memory access with exploiting many-core processors. In many applications, calculations of sparse matrices appear to become a performance bottleneck. Therefore, accelerating them to improve the performance of applications is indispensable. However, it is not a trivial task, especially when one wants to boost compute kernels for sparse matrices while exploiting the potential power of computing architectures or many-core processors.

The first issue is about memory access. A sparse matrix only holds non-zero elements which are needed for calculation. As a result, a sparse matrix holds not only values of non-zero elements but also indices which indicate where the non-zero element is in matrix. Since the amount of memory access required for computing sparse matrix kernel increases, the performance of sparse matrix operations is limited by memory bandwidth. Also, for example, sparse matrix vector multiplication includes indirect memory access due to compression of the sparse matrix. This indicates memory access is random and causes frequent cache misses. In this case, the latency for memory access becomes a key performance factor. Although recent many-core processors show high computing capability, the ratio of memory bandwidth to compute power is low and the capacity of cache per core is relatively small compared to conventional multi-core CPUs. Thus, the operations of sparse matrices hardly exploits the potential of many-core processors, and the issue about memory access becomes more serious. Although memory access in sparse matrix operations is complex in rather simple Level-2 BLAS, it is extremely hard to devise the algorithm for better memory access in Level-3 BLAS where operations are between matrices.

The second issue is about exploitation of high parallelism of many-core processors. It is imperative that assigned tasks to each thread are to be equal as possible in many-core processors. In operations on a sparse matrix, the computation complexity is determined by the pattern or number of non-zero elements in the matrix. This pattern largely affects the computation complexity of each row (or column). As the result, simple task assignment driven by row (or column) based algorithms to each thread causes load imbalance. In this case, the threads which finish their own tasks earlier than others

end up in an idle state, and limits the performance improvement with many-core processors is limited. There is another scenario of sparse matrix operation - not only computing a single sparse matrix, but also processing many (small) sparse matrices in parallel. In this case, the kernel for only one sparse matrix cannot exploit high parallelism of many-core processors. However, when we try to deal with multiple sparse matrices in parallel, there is a major difference. That is, the number or pattern of non-zero elements of each sparse matrix carries varied computational complexity. Similarly, many small sparse matrices are also introduces to the issue of load imbalance. This causes a hierarchical load imbalance issue as well.

Although just each issue is serious enough, considering both issues at the same time is more important for applications. Additional memory access for improving load-balance is resulting in that memory access limits the performance and failure of exploiting many-core processors. In the data intensive applications which include irregular memory access, the load balance of not only computation complexity but also memory access is important.

1.3 Contribution

To improve the performance of computing kernels for sparse matrices, we propose state-of-the-art algorithms for many-core processors. This thesis focuses on accelerating sparse matrix vector multiplication (SpMV) and sparse general matrix-matrix multiplication (SpGEMM), which are crucial in traditional HPC applications and recent graph algorithms. Furthermore, the thesis proposes a new computing routine, "Batched sparse matrix multiplication (Batched SpMM)", which is designed for recent algorithms and applications emerging in the machine learning field.

Sparse Matrix Vector Multiplication

- One of the issues about memory access is frequent cache miss caused by indirect memory access. In SpMV, the memory access to vector elements is indirect, and the performance is largely affected by the memory access latency. To overcomes this issue, we propose Non-Uniformly Segmented (NUS) Format, which divides the matrix along the column. NUS format is designed to improve the locality of memory access to input vector elements and bring a significant improvement of SpMV. The evaluation results on NVIDIA GPU and Intel Xeon Phi show meaningful speedups with NUS format compared to existing work for SpMV.
- Sparse matrix format, which compresses a matrix and imposes indices information as additional data, aggravates the byte per flop ratio, and thus the performance of SpMV is limited by memory bandwidth. We propose a new sparse matrix format named Adaptive Multi-level Blocking (AMB). AMB reduces the number of bits for holding the indices of

non-zero elements with keeping better locality of memory access to input vector elements in SpMV. This is achieved by dividing the matrix along the column as well as NUS format. In addition to bit reduction, AMB format treats contiguous non-zero elements in the matrix as "one block", thus reducing more memory access. Through these compressions, the index data for sparse matrices can be largely reduced, and the evaluation result on NVIDIA GPU shows the performance improvement of SpMV up to 1.4x compared to existing state-of-the-art algorithms.

- These proposed sparse matrix formats, NUS and AMB, truly shine when their parameters are appropriately selected to suit to input matrix. The iterative method is one of the cases where the format conversion offers performance gain in SpMV operations. We propose the conversion scheme for both NUS and AMB format to search the best combination of parameters with minimal cost. Evaluation result of CG method, which is one of the iterative methods, with auto-tuning scheme shows that the cost of format conversion is negligible. Our proposed scheme achieves significant speedups compared to CSR format without format conversion.

Sparse General Matrix-Matrix Multiplication

- In SpGEMM, not only memory access but also memory allocation is a crucial issue since the output matrix is also sparse. SpGEMM exacerbates load balance of many threads. To cope with these issues, we propose a new approach of SpGEMM for GPU with load balancing method and hash table exploiting cache, which enables fast memory access. The evaluation on NVIDIA GPU shows significant speedups up to over 10x from existing approaches for GPU.
- Xeon Phi also meets same issues of SpGEMM. Xeon Phi, which has more memory capacity, is required to accelerate SpGEMM of larger sparse matrices. Firstly, we identify the bottlenecks of SpGEMM on Xeon Phi using various benchmarks. Next the SpGEMM algorithms with hash table or heap data structure are optimized for Xeon Phi. Our optimized algorithms for Intel Xeon Phi achieve much higher performance compared to existing libraries including Intel MKL.
- We develop the performance model of SpGEMM on Intel Xeon Phi. We also evaluate the performance of SpGEMM with optimized algorithms, changing the various parameters of input matrices, and prove the correctness of the performance model. Furthermore, we clarify which algorithm is useful for specific application scenarios.

Batched Sparse Matrix Multiplication

- In order to achieve high throughput for processing sparse matrix multiplication between small matrices on GPU, we propose Batched SpMM, which processes tens or hundreds of SpMM operations with single kernel exploiting high parallelism and computing power of

GPU. Performance evaluation on TSUBAME3.0 implementing NVIDIA Tesla P100 GPUs shows that applying Batched approaches to GCN application achieves prominent speedups, up to 1.59x in training and 1.37x in inference.

1.4 Thesis Outline

The rest of the thesis follows as below.

Chapter 2.

In this chapter, we provide the background knowledge of my work. Firstly, we describe NVIDIA GPU and Intel Xeon Phi, including both hardware and software. Next we show what a sparse matrix is and how sparse matrices are used for real world applications.

Chapter 3.

In this chapter, we discuss the cache locality of sparse matrix vector multiplication (SpMV). Firstly, we show the issue of memory access locality in SpMV with performance profiling results. Next we propose new sparse matrix format named Non-Uniformly Segmented format designed for improving the reusability of input vector elements in the cache, and thus performance. Performance evaluation includes the results both on GPU and Intel Xeon Phi.

Chapter 4.

In this chapter, we discuss the issue of expensive byte/flop of SpMV. We propose the state-of-the-art sparse matrix format named Adaptive Multi-level Blocking (AMB) format reducing memory access. This chapter includes the efficient auto-tuning mechanism for the format conversion to AMB. The chapter shows the performance evaluation and profiling results of SpMV for 32 matrix datasets, and also includes the results of CG method with auto-tuning mechanism as the practical case .

Chapter 5.

In this chapter, we discuss the optimization strategy of sparse general matrix-matrix multiplication (SpGEMM) on GPU. Firstly, we describe what SpGEMM is and what the issues are for SpGEMM, especially on GPU. Next, we propose the efficient SpGEMM algorithm for GPU in terms of both speed and memory usage. Performance of proposed algorithm and existing approaches is evaluated with several kinds of input matrices including the matrices used in graph analysis, which is hard to deal with on GPU. Evaluation results on brand-new GPU show that the proposed algorithm achieves significant speedups compared to other libraries.

Chapter 6.

In this chapter, we discuss the optimization and performance analysis of SpGEMM on Intel

architectures including Intel Xeon Phi. Firstly, we show the microbenchmarks on Intel KNL in order to expose the performance bottlenecks of SpGEMM. Next, we propose the optimization strategy of SpGEMM along with the architecture characteristic of Intel architectures. Performance evaluation includes various input data and application scenarios. This chapter also proposes the performance model of SpGEMM and develops the recipes for selecting the best algorithm for specific input and scenario, based on the model and empirical analysis.

Chapter 7.

In this chapter, we discuss about processing many sparse matrix operations between small matrices. Firstly, we propose new computing approach, Batched SpMM, for the application of Graph Convolutional Networks (GCN). Next, we show the preliminary performance evaluation of the batched approaches, followed by the performance boost of GCN application with the batched approaches.

Chapter 8.

In this chapter, we wrap up my all contributions of accelerating sparse matrix kernels for many-core processors.

Chapter 2

Background

This chapter shows the background of this thesis. I mention about many-core processors, which play an important role of the performance improvement of computers, in terms of both hardware and software. Next, we show the basic knowledge of sparse matrix operations.

2.1 Many-core Processors

Many-core processors was devised in order to achieve high computing performance by densely implementing computing resources. Although each core in many-core processors has less computing capability, combining tens or hundreds of cores and executing them in parallel bring high performance to the applications. The strategy, where each core/thread executes same instructions but handles different data, significantly works for the applications, which require more parallelism to solve the larger problem. General Purpose GPU (GPGPU), where Graphics Processing Unit (GPU) designed for processing graphic-related operations is used more general such as for scientific simulations, is one of the ways of many-core processors. GPUs from NVIDIA or AMD are widely used, and both two vendors provide the platform for GPGPU. Besides GPU, Many Integrated Cores (MIC) from Intel is widely known as many-core processor. These many-core processors with high-bandwidth memory carries the benefit in the scientific applications, which require memory bandwidth. Recent architectures have high-bandwidth memory called HBM. The many-core processors with the HBM have significantly accelerated the applications, not only compute-bound but also memory-bound.

In order to achieve high performance on many-core processors, taking account of software is also important. Especially for NVIDIA's GPU, using CUDA has been de facto standard. In addition, OpenACC only with inserting directive also has been widely used to acquire high portability to

many-core processors. Basically, time-consuming parts are re-implemented for many-core processors to accelerate them. When we implement the program for many-core processors, it is crucial to consider the parallelism with abundant threads. The hierarchy of parallelism consists of three levels.

- thread (CUDA), workitem (OpenCL): Minimum unit in execution.
- thread block (CUDA), workgroup (OpenCL): Bunch of threads. Each block (group) is assigned to the bunch of computing cores (streaming multiprocessor on NVIDIA GPU) in terms of hardware.
- grid: Set of thread blocks.

The number of threads or workitems need to be tuned for each program and architectures. On the other hand, the software standard for Intel MIC follows the traditional one for multi-core architectures. The threads are assigning to each task by implementing with OpenMP or other libraries. However, the researchers and developers carefully implement the time-consuming part for the performance since Intel MIC provides high parallelism and memory bandwidth, which simply changes the characteristics of existing algorithms. Not only the parallelism but also memory access is also key factor for the performance. On NVIDIA GPU, 32 threads, called a warp, are working cooperatively and accessing same cache line by all threads in same warp is necessary for high performance. Also, each many-core processor has its own memory hierarchy with several levels.

2.2 Sparse Matrix

Sparse matrix, which is occupied by zero elements, appears various kinds of computing area, such as numerical applications, graph processing and, currently, machine learning. This section discussed what kinds of computation for sparse matrix is executed in the applications and what the problem is.

Numerical applications often require solving extremely large sparse linear equations. Direct method, which decomposes the problem matrix and solve the equations, requires $\mathcal{O}(\mathrm{N}^2)$ memory space and $\mathcal{O}(\mathrm{N}^3)$ computational complexity for the matrix with N rows/columns. Therefore, larger problem needs much computing time and memory usage. However, such kinds of large matrix often sparse, and iterative method becomes effective method for large sparse matrix. The iterative method can solve the equations in $\mathcal{O}(\mathrm{nnz} * \mathrm{itr})$ for the problem matrix with nnz number of non-zero elements. The iterative methods include the operation between sparse matrix and vectors.

In graph processing, graph structure is expressed as adjacency matrix, and it is usually sparse. The algorithms for graph analysis such as Pagerank, which is for search engine, can be expressed as the combinations of the operations between matrices and vectors. The multiplication between sparse matrix and vector (SpMV) becomes an bottleneck in Pagerank [5], and the multiplication between sparse matrices is time-consuming part in clustering applications or triangle counting algorithm.

Sparse matrix appears to take an attention from machine learning fields. The expression of parameters are sparsified by network pruning and expressed as sparse matrix [2, 3]. More recently, data with a graph structure can be dealt with in a graph convolutional networks (GCNs) [4], and the multiplication between sparse matrix and dense matrix is the key kernel for the GCNs applications.

2.3 Sparse Matrix Formats

Expressing sparse matrix as dense matrix causes extra memory usage for holding many of zero elements. Sparse matrix format compresses the sparse matrix, holding only non-zero elements. This compression can largely reduce memory usage and calculations. Many sparse matrix format have been proposed to improve the performance of the operations of sparse matrix for specific inputs. This section introduces widely used sparse matrix formats, Coordinated (COO), Compressed Sparse Row (CSR) and ELLPACK.

2.3.1 COO Format

Figure 2.1 shows an example of COO format. COO format holds the tuple of arrays for value data, row index and column index of each non-zero element in the matrix. Memory requirement is in proportion to the number of non-zero elements. Although non-zero elements in COO format are sorted by column and row indices in Figure 2.1, they are not necessarily sorted.

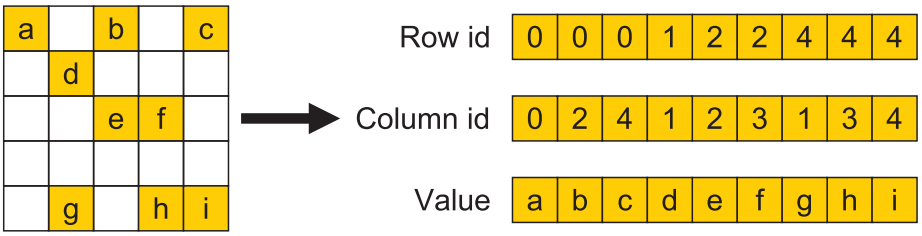


Figure 2.1: Example of COO Format

2.3.2 CSR Format

Figure 2.2 shows an example of CSR format. CSR format accumulates the non-zero elements which have same row index, and manages them by holding the pointers to the beginning point of each row. Figure 2.2 shows sorted array by column indices, but as well as COO format, it is not necessary. There is alternative sparse matrix format named Compressed Sparse Column (CSC) format, which manages the non-zero elements per each column and hold the column pointers.

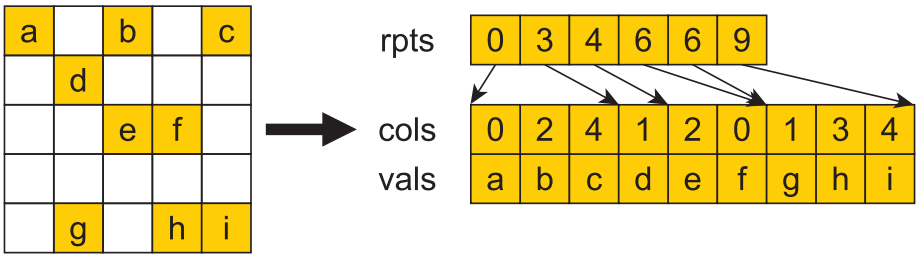


Figure 2.2: Example of CSR Format

2.3.3 ELLPACK Format

ELLPACK format is designed to be efficient for parallel computation on vector and many-core processors. The matrix data is stored in column major ordering as shown in Figure 2.3. In ELLPACK, the numbers of elements are made to be the same for all rows, with the rows embodying the largest number of non-zero elements determining the row size, and for other (smaller) rows zeros are filled in. This simplifies the SpMV computation with contiguous access at the cost of increasing the storage size as well as number of memory accesses. ELLPACK is efficient if the rows have similar number of non-zero elements, but becomes very inefficient otherwise when they are not balanced. There have been several sparse matrix formats extended from ELLPACK to alleviate this problem [6–8].

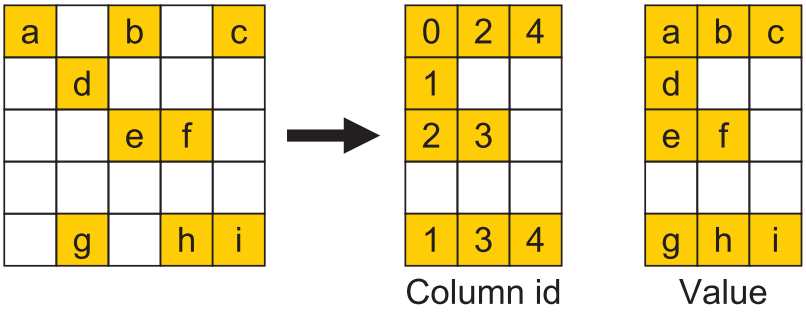


Figure 2.3: Example of ELLPACK Format

Chapter 3

Improving Cache Locality in Sparse Matrix Vector Multiplication

3.1 Introduction

Many-core architectures such as (GP)GPUs and MICs are becoming mainstream in various IT platforms, being lead by the successes in graphics and supercomputers, and making their way to smartphones, tablets *etc.* in the next few years. Such processors integrate numerous processing cores onto a single chip, and facilitate very high memory bandwidth to match the increase in compute capabilities, and often exhibit several factors power-performance advantage over conventional multi-core, latency optimized processors such as the Intel Xeon.

Many-core architectures, especially GPUs, are architecturally optimized as so-called *throughput processing* in SMT (Simultaneous MultiThreading) model, in that all threads execute same computation for different data that are streamed in and out of the processor. Such is regarded as an extension of vector processors of the past, with more flexible execution order of the elements. As such, many-core processors do not embody a rich and deep cache hierarchy of latency processors today, but rather have small caches or explicitly managed internal memory (*e.g.*, somewhat inappropriately termed as *shared memory* for NVIDIA GPUs), with memory systems designed for high streaming data throughput but less for latency, which are aimed to be hidden with SMT (Simultaneous Multi-Threading).

However, much of the algorithms and the code today are optimized for latency-optimized processors. When porting such algorithms to many-core processors, especially to the streaming SMT model, in some cases such as stencil computations [9], it is largely fine, as memory access patterns are simple

and fairly regular. But when the computational kernel involves many random and indirect memory accesses, performance drops significantly despite still being memory access dominant—a known but critical weak point of modern many-core processors that lack hardware support for such random accesses. In particular, Sparse Matrix-Vector multiplication (SpMV), the lowest-level kernel of major portions of various numerical algorithms, including CG as well as other algorithms such as page ranks [5], unfortunately exhibits this property, as matrix elements are stored in a compressed fashion, and requires numerous random access to the input vectors. Although there have been various work in the past to improve the performance of SpMV computations on GPUs and other many-core processors, they have not been able to fully exploit the performance of those processors in a stable fashion, mainly due to the random access and small size of the cache on such processors.

We propose a family of new sparse matrix storage formats and an associated, small cache-friendly, bandwidth-reducing parallel algorithms to best exploit the cache under most circumstances, and thus achieve high and stable SpMV performance and thus higher-level algorithms dependent on the SpMV kernel such as CG solvers [10], without the need for expensive hardware. Instead, our family of new sparse matrix formats for many-core processors significantly increases the cache hit ratio by segmenting the matrix along the columns, dividing the work among the many core up to the internal cache capacity, and aggregating the result later on. Performance studies show that we achieve up to 2.0x speedup for real datasets, 3.0x for synthetic matrices in SpMV, and 1.68x in multi-node CG. Our studies compared our format with top-of-the-line vendor libraries and competing new formats that have been recently proposed such as SELL-C-σ.

3.2 Numerical Background

3.2.1 Conjugate Gradient Solver and SpMV

Many large-scale scientific simulations solve linear equations represented by sparse matrices generated as a result of numerical discretization, such as FEM. Conjugate Gradient (CG) method is often used (typically with some preconditioner) as an effective solver of the positive definite and symmetrical equations, which in turn embodies SpMV and level-1 BLAS of AXPY and dot products that are basically memory-bound. The level-1 BLAS involves sequential memory accesses that are efficient on most processor architectures, especially many-core processors. On the other hand, SpMV performs random access to the input vectors which lead to numerous cache misses—the situation is further aggravated, as the accesses are intermixed with sequential access to matrix data that is much larger than the vector data, thus polluting the cache access. As a result, SpMV now occupies dominant percentage of the total execution time of the CG method on modern processors, typically more than 90%. Various research have attempted to address this issue by developing efficient implementation of SpMV, to reduce this overhead [11–14].

3.2.2 Sparse Matrix Formats

Initial design motivation of sparse matrix format was data compression only, as the matrix in a typical scientific simulations could involve billions of data points, but most of them are zero, with a very low number of non-zero elements in each row or column as a result of discretization, so their direct storage is an immense waste of memory as well as floating point operations involved. Historically popular sparse matrix formats have been CSR and COO: CSR (Compressed Sparse Row) format compresses the non-zero elements in each row, and adds indices that point the starting position of the rows. COO (Coordinated) format simply lists the tuple of (row, column, value) for the non-zero elements. More recently, in order to achieve better performance in SpMV computations, various sparse matrix formats have been proposed for specific types of matrices, or for specific architectures [15–18]. For example DIA and ELL formats are suitable for diagonal matrices, while JDS (Jagged Diagonal Storage) format was proposed to allow for continuous streaming memory access on vector processors. Since modern many-core processors with extensive SIMT capabilities such as GPUs can be regarded as generalizations of classic vector processors, their memory access characteristics are very similar to those of vector processors, and as a result JDS format usually works efficiently [19, 20]. ELLPACK is also designed to work well with vector processors and many-core accelerators; however, as later experiments show, performance of sparse matrix formats may fluctuate significantly depending on the matrix form and shape, especially for large matrices of modern problems.

JDS

JDS format compresses the non-zero elements of rows first in a CSR format. After compression, JDS reorders the rows by number of non-zero elements per row as shown in Figure 3.1. By this reordering, run length is maximized for each column. For vector processors with list vector or gather instructions, the iteration count of the inner-most loop is lengthened, improving performance. At the same time, the order of storing matrix data is also transposed. Reading the input vector uses gather instructions, and writing the output vectors to the main memory uses scatter instructions to store the elements in the original ordering. However, modern many-core processors are optimized for sequential streaming access only, and they lack the sophisticated and costly memory hardware to accelerate random memory access performance, and as a result, typically lack hardware scatter-gather instructions. Where random access is required it is only within the realm of their (rather small) cache memory. As such, generalized gather / scatter operation performances are poor on such processors, hindering JDS SpMV performance significantly.

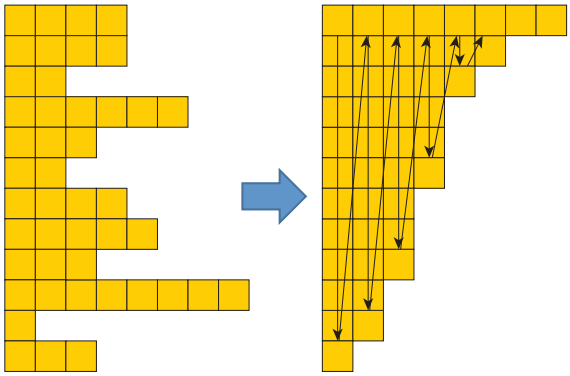


Figure 3.1: JDS format

We can avoid the JDS SpMV scatter operation algorithmically by storing the output vector once, and performing the reordering using the gather operation afterwards. Another optimization method is possible when SpMV is used as a part of iterative solvers such as CG, in that if we reorder the vector data for a given matrix before starting the main iterations, no reordering is required during the iterations. Figure 3.2 shows this case. In general, this achieves the best performance, although there are still overhead and performance instability. Henceforth, we will use this reordering version as the baseline in the successive performance evaluations.

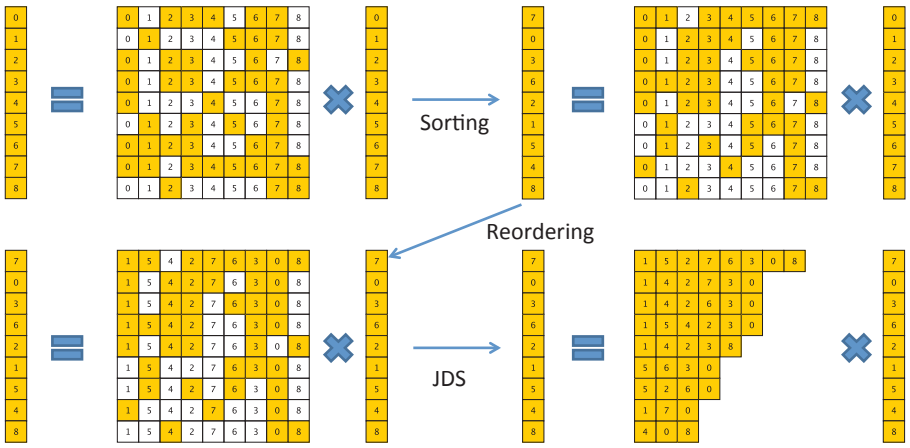


Figure 3.2: Reordering input and output vectors for JDS format

ELLPACK family Format

Moritz Kreutzer, *et al.* recently proposed the SELL-C-σ format to alleviate the disadvantage of ELLPACK format [21]. Figure 3.3 illustrates the format, where, C and σ are parameters to the layout.

SELL-C-σ sorts every σ rows by the number of non-zero elements per row. After the sorting, every C rows are converted to ELLPACK. By the initial sorting, the number of zeros filled in the ELLPACK format is reduced. The sorting scope 'σ' is usually smaller than the matrix size, as global reordering may decrease memory access locality present in the original ordering. The parameter C is the chunk size, which should be set to architectural property of the processor, such as vector register length, warp size, wavefront size, etc., or their multiples.

Nathan Bell, *et al.* proposed the *Hybrid* format which combines ELLPACK and COO formats [22], where the majority of the matrix data is covered by the ELLPACK format, whereas some of the outliers that has the potential of ballooning the ELLPACK zero fill-ins are represented with COO. NVIDIA's cuSPARSE library supports CSR format as well as the Hybrid format, and also provides a subroutine to optimize the number of elements to be covered by ELLPACK, as it is the most important parameter in the Hybrid format.

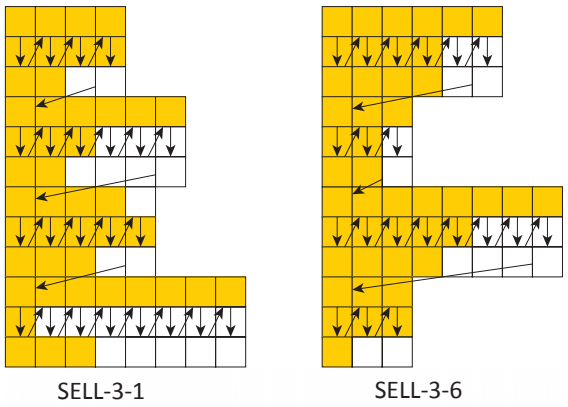


Figure 3.3: SELL-C-σ format

3.3 New Cache-Friendly Matrix Formats

3.3.1 Low Cache Hit Ratio of Previous Formats

Although various sparse matrix formats have been proposed as we have seen, they all have the commonality in that they require random access to the input vectors, being inefficient in modern many-core processors that lack efficient random access capabilities such as scatter-gather. Although some processor families have expressed such instruction extensions, at least for some foreseeable time they are instruction definitions only, and likely not accompanied by fundamental hardware-level improvements, contrary to vector processors of the past with costly hardware features such as 32,768-way memory interleaving on the NEC SX-9 [23]. Caches do not help in this case as the access

is too sparse, and in fact aggravates the situation by mandating fetching of the entire cache line in order to fetch one word of memory. In addition, cache misses cause pipeline stalls, reducing the opportunity of latency hiding of successive memory access operations, even with extensive SIMT.

In order to analyze the effect of cache hit ratio versus their effect of performance, we conduct the following experiment of SpMV performance. We randomly generate various sparse matrices, changing the number of columns to observe the effect of the cache hit ratio. The number of rows is set to 4M and number of columns is set to powers of two. Number of non-zero elements per row is set to 16.

Figure 3.4 and 3.5 shows the performance on NVIDIA Tesla K20X GPU using JDS format. The number of threads per thread block is set to 32, 64, 128, 256, and 512. In the case of 32 or 64 threads per thread block, the performance is limited to about 13 or 25 GFlops respectively, due to insufficiency in the number of threads to hide the latency when accessing the matrix data. For other cases, there are two large performance gaps, where each corresponds to the capacity of read-only cache and the L2 cache, respectively. This roughly indicates that, if we limit the range of random access to the input vector to be within the caches, we could reduce the random access to memory and thus achieve high performance.

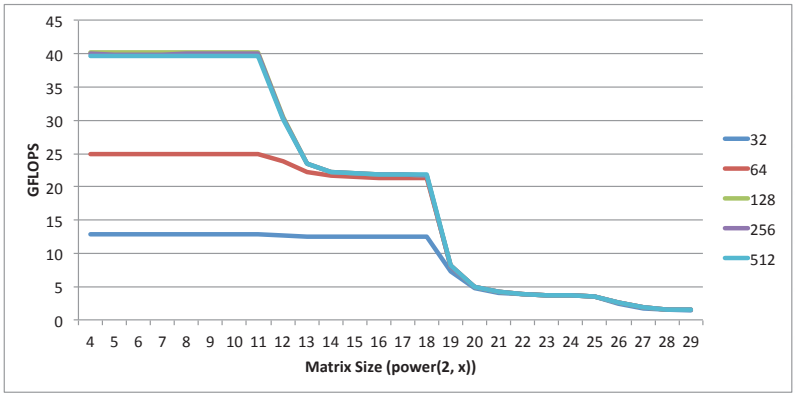


Figure 3.4: Performance of SpMV for random matrix with various column sizes

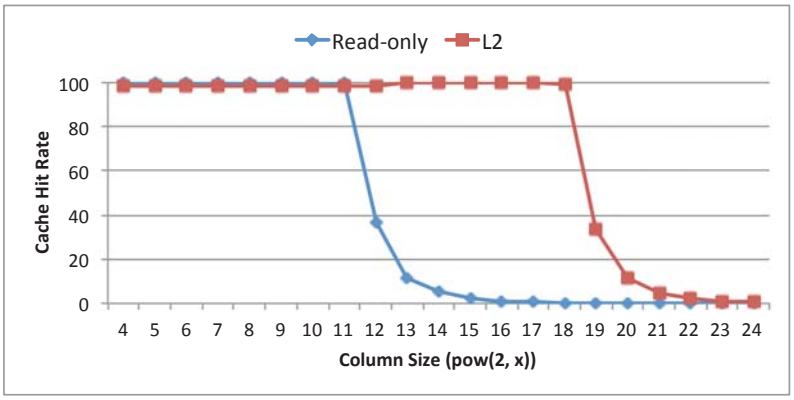


Figure 3.5: Cache hit ratio of SpMV for random matrix with various column sizes. The number of threads per thread block is set to 512.

Given this observation, we next propose a family of new SpMV algorithms and matrix formats based on column-segmentation of the matrix. The column-wise segmentation also divides the input vector into short vectors, which improves the cache hit ratio and thus performance in a stable fashion, without the need for expensive scatter-gather hardware. The sub-matrix format could be various formats proposed in the past, such as JDS or SELL-C-σ; we analyze their characteristics and tradeoffs in the subsequent sections.

3.3.2 A Family of Segmented Sparse Matrix Formats

Figure 3.6 shows the SpMV computation using our S-JDS (Segemented-JDS), which uses JDS format for the underlying sub-matrices. We can also use SELL-C-σ format for sub-matrices or any other formats, but these formats have favorable properties for modern many-core processors. The SpMV computation consists of two phases. The first phase computes SpMV for sub-matrices and sub-vectors, and stores the resulting vectors into the memory. The second phase accumulates the intermediate vectors, where reordering of the vectors is performed. Since the reordering pattern is not common to all the sub-matrices, these permutations cannot be eliminated by reordering of input and output vectors.

Segmentation improves the cache hit ratio of accesses to the input vectors in the first phase. On the other hand, it increases the number of sequential writes, as there has to be writes to intermediate vectors. In the second phase, the algorithm does increase the number of random reads of intermediate vectors, as well as sequential reads of permutation indices. However, the random reads in the second phase are not so critical because there is certain memory access locality. Overall, the advantage of the segmented formats depends on the trade-off between the improved cache hit ratio versus memory

operations on intermediate vectors.

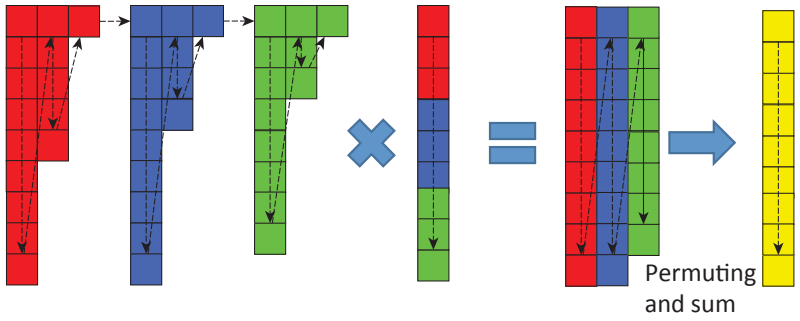


Figure 3.6: SpMV computation in S-JDS format

3.3.3 Non-uniformly Segmented Formats

The sizes of sub-matrices, that is, segmentation width, need not be identical across all the sub-matrices. Since many-core processors have multi-level cache hierarchy, we can mix-and-match a variety of segmentation widths. In fact, matrices from irregular problems typically have divergence in the element density within the matrix, i.e., some columns may have more non-zero elements than others. For performance we should place such high density part into the L1 cache or the read-only cache. Contrastingly, large segmentation width would be more appropriate for the low density part, because we expect low reusability for those elements, possibly placing them in lower level caches (L2 in the case of K20X). Since most many-core processors today currently employ a two-level cache system, we propose non-uniform segmentation as follows:

$$N = S_1 \times K_1 + S_2 \times K_2 + S_3.$$

where S_1 and S_2 are segmentation width suitable for L1 cache and L2 caches respectively, and K_1 and K_2 are number of segments of those widths. S_3 is the remainder of the matrix not suitable for segmentation.

Figure 3.7 illustrates the procedure for converting a matrix into the NUS-JDS (Non-Uniformly Segmented JDS) format: (1) columns are reordered by the number of non-zero elements in columns; (2) input vector is also reordered, and the column indices are updated; (3) rows are exchanged so that the ordering of output vector becomes the same as the input vector; (4) the matrix is segmented by columns, and sub-matrices are converted into JDS (SELL-C-σ in case of NUS-SELL-C-σ). Currently, optimal segmentation widths are determined by exhaustive search. Optimizations that reduces the search space based on a cache performance model is a subject of future research.

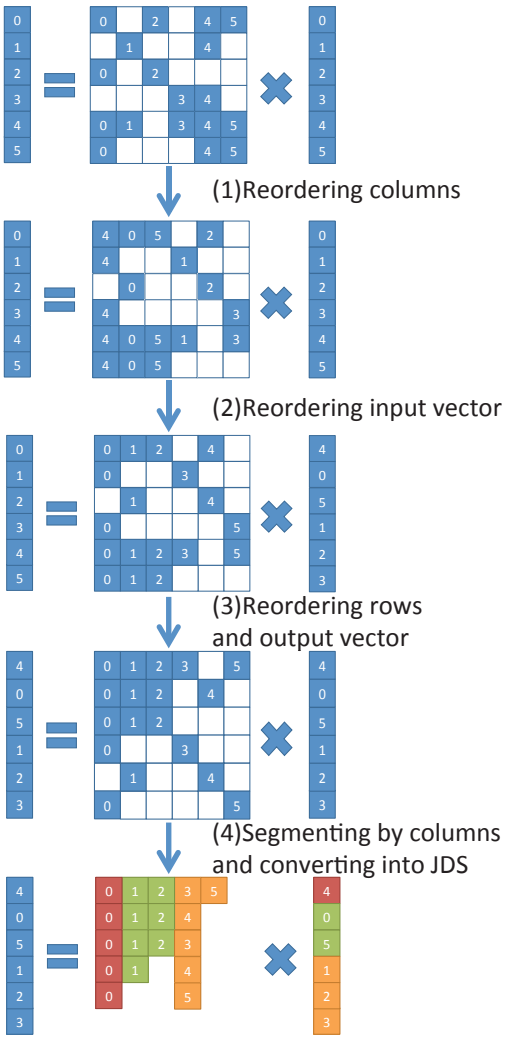


Figure 3.7: Procedure of converting a matrix into the NUS-JDS format

3.3.4 Auto Parameter Tuning Mechanism for NUS Format

NUS format with better performance requires appropriate parameter setting about column-wise division, the number of segments (seg_num) and size of segment (seg_size). However, the parameter setting is difficult because the performance is hardly predicted before the format conversion. We propose auto parameter tuning mechanism for NUS format in practical use case such as iterative method. Figure 3.8 shows the flowchart of this auto parameter tuning mechanism. Format conversion and iterative method are executed in parallel by utilizing task-parallel functionality of OpenMP. One thread manages iterative method executed on GPU from the beginning, and the other thread

converted the matrix to NUS format with a parameter set. After the format conversion, the matrix data is transferred to GPU and the performance of SpMV is evaluated once. If the performance of NUS format with new parameter is better than one with old parameter, the matrix data is replaced with new one. This cycle is repeated until the mechanism finishes parameter search or the iterative method converges. This approach can explore the best parameter set in the parameter space without additional execution time in iterative method compared to original format and parameters.

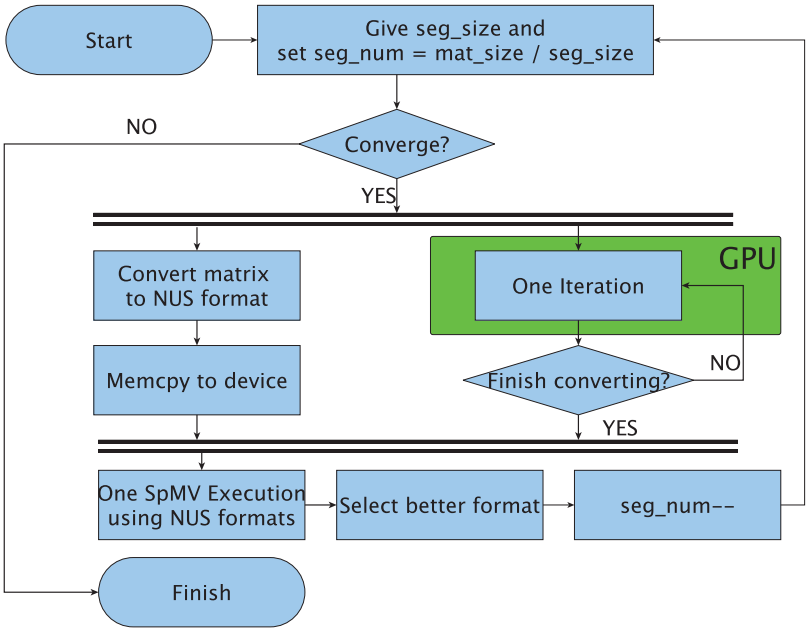


Figure 3.8: Chart of auto parameter tuning mechanism

3.4 Performance Evaluations

We now compare the performance of our family of sparse matrix formats optimized for SpMV computations compared to existing formats. Performance values are in GFlops assuming SpMV computation requires $(2 * \text{nnz})$ floating operations, and an iteration of CG method requires $(2 * \text{nnz} + 11 * \text{N})$ likewise, where N is the square matrix size, and nnz is the total number of non-zero elements in the matrix. All experiments are done in single precision arithmetic although in principle the method is perfectly applicable to double precision with similar performance trends. The matrix and vector values are stored as the `float` type, and indices for matrix uses the 32bit integer type.

For our performance evaluations, we utilize the matrices from the Sparse Matrix Collection at University of Florida [24]; the properties and the sizes of the matrices are summarized in Table 3.1. From the collection, we selected real, symmetric matrices with high non-zero element density.

Table 3.1: Matrices from Sparse Matrix Collection at University of Florida. N is the number of rows (or columns), NNZ is the number of non-zero elements.

Matrix	N	NNZ	Application info	NNZ / N
audikw_1	943,695	77,651,847	structural problem	82.28
nd12k	36,000	14,220,946	2D/3D problem	395.02
crankseg_2	63,838	14,148,858	structural problem	221.63
crankseg_1	52,804	10,614,210	structural problem	201.01
nd24k	72,000	28,715,634	2D/3D problem	398.83
mouse_gene	45,101	28,967,291	undirected weighted graph	642.28
F1	343,791	26,837,113	structural problem	78.06
F2	71,505	5,294,285	structural problem	74.04
bmw7st_1	141,347	7,339,667	structural problem	51.93
TSOPF_FS_b300_c3	84,414	13,135,930	power network problem	155.61
inline_1	503,712	36,816,342	structural problem	73.09

3.4.1 Performance of Segmented Formats on Single NVIDIA GPU

We first compared the performance of our segmented formats on NVIDIA's Tesla K20X GPU installed on TSUBAME-KFC, which was recognized as the greenest supercomputer on the Green500 list in Nov. 2013. The GPU has 6 GBytes of device memory, being sufficient for storing all the matrices used in the performance evaluations. The Kepler-generation GPU has 1.5 MByte L2 cache memory shared by 14 SMXs. Each SMX is said to have a 48KB read-only cache memory, but in reality SMX itself is composed by four parts, each of which has a capacity of 12 KBytes. For this reason, the segment size in our sparse matrix formats is chosen assuming cache sizes of either 12KB or 1.5MB. GPU codes are implemented in CUDA 5.5 and are executed on CentOS Linux 6.5 OS 64bit system. In SpMV, read-only cache memory of Kepler GPUs is used only for random accesses to input vectors in both the first and the second phases, and all the other coalesced memory accesses to matrix values and indices directly go to the L2 cache.

Figure 3.9 shows the performance of SpMV computations for Florida matrices on Tesla K20X, using NVIDIA's cuSPARSE library as well as our implementations of the original JDS and SELL-C-σ formats, compared to our proposed formats. The cuSPARSE library is tested with both CSR and HYBRID formats. The CSR format in particular demonstrates relatively stable high performance for all matrices.

To compare with such existing formats, we devise four versions of segmented matrix formats: uniformly segmented or non-uniformly segmented, combine with JDS or SELL-C-σ as the underlying sub-matrix format after the segmentation. For NVIDIA GPUs, the parameter C of SELL-C-σ is

set to 32 derived from the warp size, whereas the parameter σ are set to 25600, resulting in better performance compared to the recommended parameters in the original paper [21]. Here, we can observe that our NUS-SELL-C-σ format is the best choice, in that it is both high-performing and very stable, demonstrating the best performance for most of the matrices, and even if not almost matching the best. The segmented formats achieve a (geometric) mean speedup of 1.13x compared to non-segmented formats.

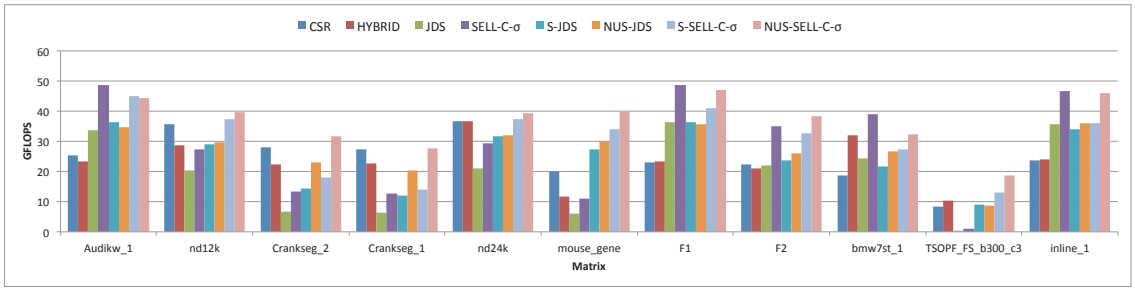


Figure 3.9: Performance of SpMV computation for Florida matrices on Tesla K20X

Figure 3.10 shows the cache hit ratio during the SpMV computations with CSR, SELL-C-σ, S-SELL-σ and NUS-SELL-C-σ formats. For large matrices where the vector size is larger than the L2 cache, the cache hit ratio of the L2 cache is improved by the segmented format, while the cache hit ratio of the read-only cache is improved for small matrices. One exception is the matrix 'TSOPF_FS_b300_c3', which is small in size, but has the rows with much large number of non-zero elements compared to other rows; in this case, the L2 cache hit ratio becomes more important.

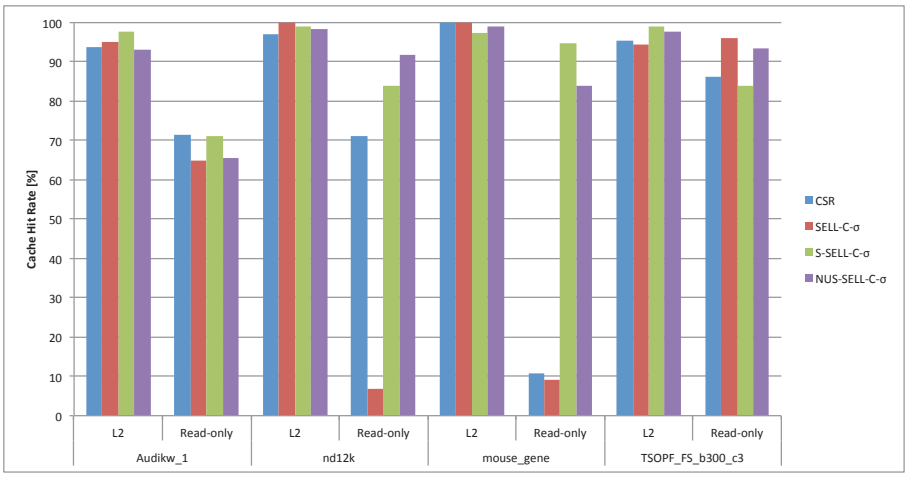


Figure 3.10: Cache hit ratio in SpMV computation for Florida matrices on Tesla K20X

Table 3.2 shows the optimal width of the segmentation for each matrix. These sizes are based on the matrix sizes and cache sizes of target architecture. We note that it does not depend on the internal sub-sparse matrix format of JDS or SELL-C-σ. Since large matrices cause many L2 cache misses, the segmentation width should fit the L2 cache. For matrices crankseg_1 and nd24k, the optimal segmentation width patterns are included in the uniform segmentation. For matrix nd24k, the width is also the same, but we observe that NUS- series exhibits marginal performance superiority over the S- series this is because the NUS- series performs sorting by the number of non-zero elements per column, thus improving cache hit rate.

Table 3.2: Optimal width of the segmentation.

Unit of the width is not in bytes but in number of float elements.

Matrix	Uniformly Segmented	Non-uniformly Segmented
audikw_1	256k	128k*2+665k
nd12k	2k	2k*10+3k*5
crankseg_2	1k	3k*16+4k*5
crankseg_1	1k	2k*26
nd24k	3k	3k*24
mouse_gene	1k	1k*35+2k*5
F1	512k	128k*1+207k
F2	32k	32k*2+6k
bmw7st_1	32k	16k*3+90k
TSOPF_FS_b300_c3	1k	1k*40+8k*5
inline_1	256k	128k*1+363k

To ascertain the *optimality* of our approach, we calculate the *effective memory bandwidth* of our our algorithm as follows: classic vector processor systems had no cache memory for the vector processing units, including NEC machines up to SX-8, as well as Cray machines up to T90. They also facilitated special scatter/gather hardware units to help with random memory accesses. SpMV computation on such architectures would require $(3 * \text{nnz} + 2 * \text{N}) \times$ sizeof(float and int) bytes of memory accesses in total. Now, assuming that it would be the inherent memory accesses required, Table 3.3 shows the effective memory bandwidth during the SpMV computation using NUS-SELL-C-σ format on a Tesla K20X GPU, achieving up to 284GB/s effective bandwidth with our segmented format and algorithm. This is higher than NEC SX-6's peak single node memory bandwidth of 256GByte/s, which in reality would be difficult to achieve due to stalls in the memory system, caused by various factors such as bank conflicts.

Table 3.3: Effective memory bandwidth (GB/s) during the SpMV computation using NUS-SELL-C-σ format.

Matrix	Bandwidth
audikw_1	267.5
nd12k	238.9
crankseg_2	190.0
crankseg_1	166.7
nd24k	236.9
mouse_gene	240.3
F1	284.0
F2	231.5
bmw7st_1	196.9
TSOPF_FS_b300_c3	112.5
inline_1	278.7

Our segmented format should have a performance advantage for modern-day large problems, which cause low cache hit rates. Such matrices are unfortunately not publicly available; so in addition to the matrices from Florida Matrix Collection, we investigate larger, randomly generated matrices. The matrix size is 1M, 1.5M, 2M, 2.5M, and 3M. The density of non-zero elements is set to 0.0001%, 0.0002%, and 0.0005%. The diagonal elements are always non-zero. The other elements are completely random, and there is no restriction except being symmetric. Such matrices puts us at a disadvantage, since there is no structural locality semantics that could have been present in real problem, but nonetheless this is likely the worst case, and if our proposal works well, it should perform even better in a real problem.

Figure 3.11 shows the performance of SpMV computation for these randomly generated matrices. The name of matrices is exhibited as 'rand_<size>_<density>'. S- series uses 256K for segmentation width, to improve the L2 cache hit ratio. For smaller and low density matrices, JDS and SELL-C-σ work efficiently. But as the matrices become larger, their performance degrades quickly due to cache capacity misses. By contrast, the performance of segmented formats is stable, simply depending on the density of the non-zero elements, which is roughly equivalent to the reusability of cache memory. The segmented formats achieve up to 3x speedup compared to non-segmented formats.

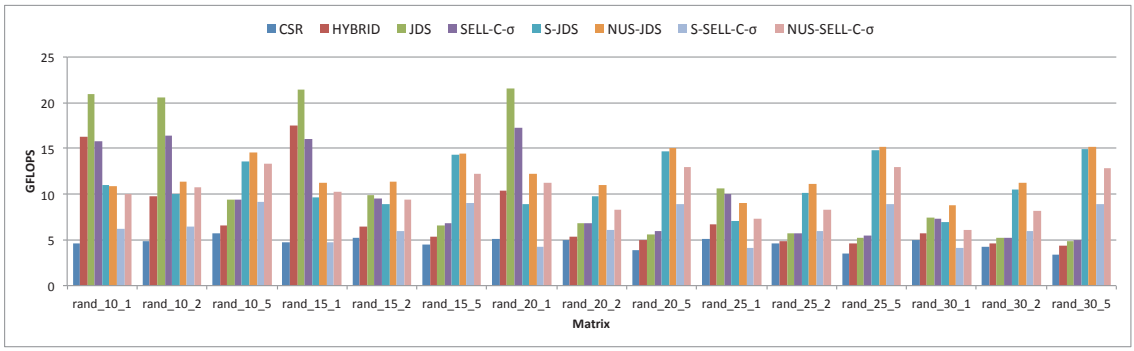


Figure 3.11: Performance of SpMV computation for randomly generated matrices on Tesla K20X

We now apply our segmented format to a CG method to investigate the advantages, where we can expect speed-ups as SpMV occupies large percentage of the total execution time. The other routines use NVIDIA's cuBLAS library. Figure 3.12 shows the performance of the various CG methods. The matrices 'mouse_gene' , 'F1', 'F2' and 'TSOPF_FS_b300_c3' are removed from this result because they are not positive definite matrices. In Figure 3.12, we can observe similar performance improvement to SpMV, up to 1.12x, compared to the best of the existing formats in most of the matrices.

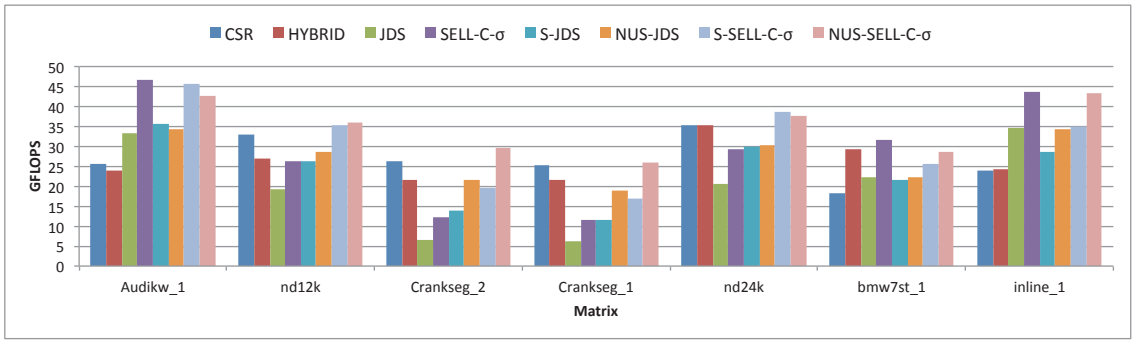
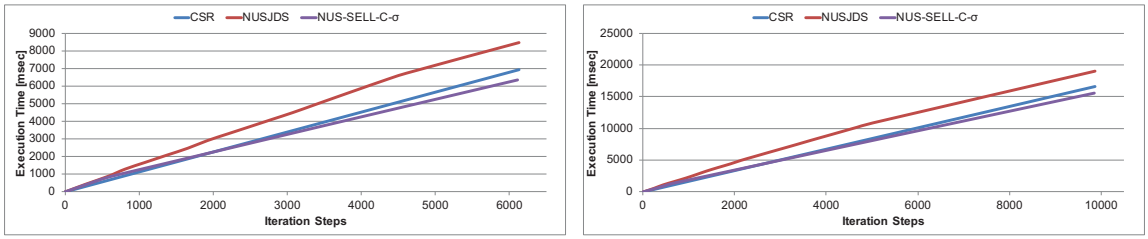


Figure 3.12: Performance of CG method for the Florida matrices on Tesla K20X

As the more practical evaluation, we show the execution time of CG method with auto-parameter tuning mechanism for NUS format. Figure 3.13 shows the cumulative execution time until convergence on 'crankseg2' and 'nd24k'. In the beginning of iterations, our approach takes more execution time compared to CSR format since the parameters are not optimized yet. However, our approach is gradually getting better after it finds best combination of parameters, and finally achieves convergence in less execution time. Our auto-parameter tuning mechanism efficiently executes iterations of CG method with converting format with better parameters.



(a) crankseg2 (b) nd24k

Figure 3.13: Result of auto parameter tuning CG method for NUS formats

3.4.2 Performance of Segmented Formats on Xeon Phi

We evaluate the performance of the segmented formats on Intel Xeon Phi 7120X processor, or so-called MIC architecture. Xeon 7120X is top-of the line, embodying 61 cores and 16GByte onboard memory featuring 350GByte/s theoretical peak bandwidth. Each core has 32KByte L1 data cache and 512KByte L2 cache, for 30MByte L2 cache in aggregate total. But unlike GPUs, the data first must be transferred to the local L2 cache when a core accesses the data residing on L2 cache of the other cores, and as a result, capacity miss will become serious as the cache line is shared.

Our SpMV implementation for MIC uses 512-bit SIMD instructions (16-way vector for 32bit data) including gather operations. For accesses to matrix data, non-temporal option is specified in SIMD load and store instructions. The MIC *native mode* is used, meaning that it boots and runs its own operating system as an independent node. ECC function of the device memory is disabled to maximize performance. The code is written with MIC intrinsics in addition to OpenMP `#pragmas`, and the number of threads is set to 244 in an attempt to hide the memory latency as much as possible. In SELL-C-σ format, the parameters C and σ are set to 16 and 25600 on MIC.

Recall that, as shown in Figure 3.4, on GPUs performance largely degrades when the vector size exceeds the capacity of read-only cache and/or the L2 cache. Figure 3.14 shows the result of the comparative experiments on MIC. For JDS format, performance degradation starts after 32KByte (L1 data cache size). The second degradation seen on the GPU when we exceed the capacity of L2 cache is not as evident on MIC (512KByte), due to the existence of the L2 caches of other cores likely holding onto the data. On the other hand, SELL-C-σ behaves somewhat oddly, and peaks at 8K elements (=32KB). This could be due to the following reason: for GPUs, it is possible to isolate the random access to the input vector from sequential accesses to the matrix as well as other accesses via the use of the read-only cache, which we do on the current implementation; however on MIC, they are all forced to share the same L1 cache, resulting in thrashing and thus performance instability.

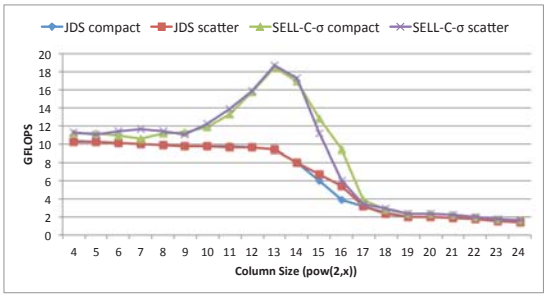


Figure 3.14: Performance of SpMV for random matrices with various column sizes.

Figure 3.15 shows the performance of SpMV computation of the Florida matrices on MIC. On MIC, SELL-C-σ without segmentation works efficiently for more matrices than GPUs. This is because the segmentation width is set to 8K on MIC, and thus it works efficiently only if the characteristics of the matrix matches this size. On GPUs, the matrices 'nd12k' and 'nd24' selected 2K and 3K segmentation size respectively; they are relatively close to 8K and thus those matrices show higher performance with segmented formats on MIC as well. The matrix 'TSOPF_FS_b300_c3' used 1K for GPU, and this also works well on MIC. One thing to note is that JDS formats nor its segmented variations are not suitable on MIC, as the JDS format on one hand reduces the storage size while on the other hand it requires additional branch instructions in its processing. On GPUs, such divergence is acceptable because there are many other threads which can be executed as the computation is still very much bandwidth dominant, and GPUs can launch hundreds of threads per each SM. But on MIC, only up to four threads are running on each core, and as a result it becomes harder to hide the effect of branch instructions.

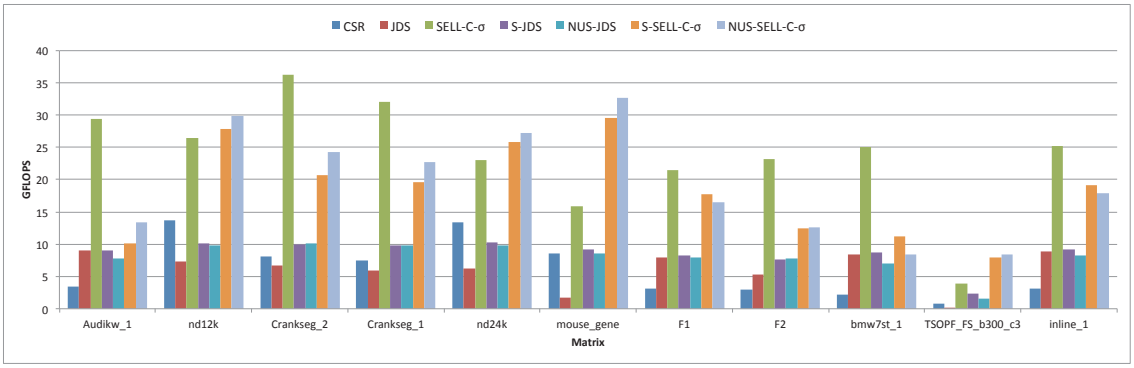


Figure 3.15: Performance of SpMV computations for the Florida matrices on MIC.

Figure 3.16 shows the performance of CG method for the Florida matrices on MIC. Intel MKL library [25] is used for the other level-1 BLAS routines. The segmented formats achieves the highest

performance for several matrices, while SELL-C-σ is the best for many of the others. Further detailed analysis of the MIC cache behavior to identify the segmentation overhead, and auto-selection of segmented/non-segmented versions is the subject of future work.

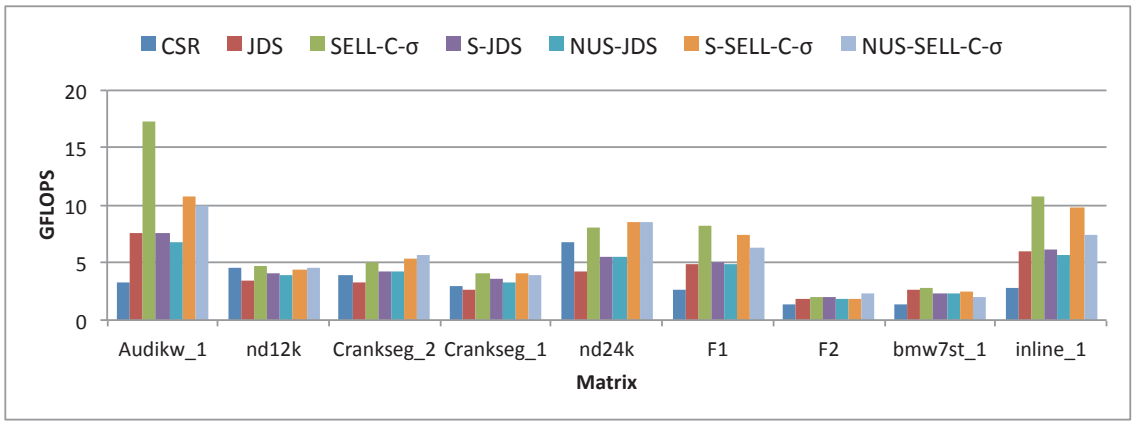


Figure 3.16: Performance of CG method for the Florida matrices on MIC.

3.4.3 Multi-Node Application of Segmented Formats

We test the viability of our segmented formats in a multi-node, distributed setting, employing multiple GPU devices on multiple supercomputer nodes to allow for larger problem sizes and speedup. The concern here is whether the algorithm will be able to exploit the locality especially in strong scaling situations, as the effective sparsity of the matrix will become more severe, possibly negating the effect of the caches. Our initial tests with the Florida Matrix Library verified this to some extent, as we found the problems to be too small for multi-node strong scaling. Thus, we generate larger matrices of various densities that are more realistic for modern-day machines. We still conduct our tests as strong scaling and not weak scaling, as the latter result will be fairly obviously to our favor.

Figure 3.17 shows the performance of CG method using multiple nodes under strong scaling. The matrices of 8M× 8M are randomly generated with 0.0001%, 0.0002% or 0.0005% non-zero element density. We used up to 8 nodes from TSUBAME-KFC compute nodes. On each node, only one GPU is used in our performance measurement. The data transfer between nodes uses OpenMPI library 1.7.2 over FDR InfiniBand network. The matrices are divided by row, and distributed to each node / GPU. Since the density of non-zero elements determines the reusability of cache line, we observe that our segmented format works efficiently especially in higher density matrices achieving up to x1.68 speed up compared to the non-segmented SELL-C-σ. Additionally, we observe that, in the case of the matrices with lower density, the data transfer time between nodes takes relatively longer, and as a result, the performance difference becomes unnoticeable unlike the single node case. As such,

employing the segmented version will likely be either parity or advantageous for all the distributed memory cases, so that one will not lose by its usage.

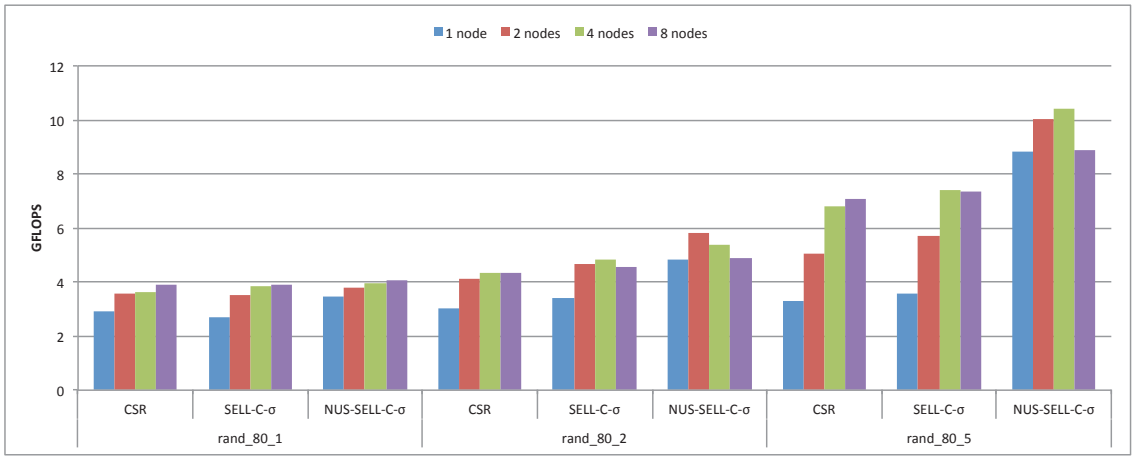


Figure 3.17: Performance of CG method using GPUs on multiple nodes

3.5 Related Work

There have been several past work addressing low cache hit rate caused by random accesses to the input vectors in SpMV computation. Eun-Jin Im, *et al.* proposed a cache blocking algorithm for SpMV computation on CPUs [26]. To improve the cache hit rate, they divided the matrix into small tiles. For a given matrix, they selected the best size of small tiles of fixed square dimensions from 32K× 32K to 128K× 128K. Since the target processor was a generic CPU, there were very few threads per core, and each core could dominate the L1 cache exclusively for easier control. Contrastingly for GPUs, many threads share the cache memory, and the control becomes more difficult and thus requires more probabilistic handling and varying of the segment width as well as the underlying sub-matrices storage format depending on the distribution of non-zero elements. As a note, one could also devise a family of 2-D versions of our segmented format and algorithm, but our initial experiments showed slowdowns compared to 1-D segmentation we have proposed. In addition, unlike GPUs, the number of registers on CPU is limited, requiring temporal output vectors to be stored in the cache memory while on a GPU it can be a part of the thread state. These store operations to the L1 cache are not as critical to the performance.

Weizhi Xu, *et al.* proposed the BCSR format [27]. BCSR also divides a matrix into tiles, skipping the all-zero area. They evaluate the performance of BCSR on Fermi-generation GPU (GeForce GTX 480). Unlike the current Kepler-generation GPU, Fermi GPU's L1 cache can cache the data read from global memory. For this reason, their tile width is set to 16KB or 48KB, which are the natural sizes

for Fermi. Also, our segmentation method stores temporal output vector onto global memory, while BCSR uses synchronization within the thread block, and continues the computation of the next tile for the same range of rows. This seemingly avoids the increase of memory access instructions, but in reality it only contributes to the improvement of cache hit rate at the thread block level and not globally, and as a result, memory access locality in the L2 cache is not considered. In Figure 3.18, we compare the performance of BCSR as indicated in the original paper versus our segmented format, using the same matrices (Table 3.4). The original paper used GeForce GTX 480, while we used the Tesla M2070 GPU. Both GPUs are 40nm Fermi-generation, but the GTX480 has more CUDA cores (480 vs. 448) as well as higher clock (1400Mhz vs. 1150Mhz), and as a result the peak performance is much faster on the GTX480 than M2070 (1344 GFlops vs. 1030 GFlops), and more importantly, the memory bandwidths are much higher with 177.4GB/s and 148GB/s respectively. Still, we observe that our S-JDS format achieves much higher performance, using a much slower GPU.

Table 3.4: Matrices used in the BCSR paper.

	N	nnz	category	nnz/N
shipsec1	140,874	7,813,404	structural problem	55.46
consph	83,334	6,010,480	2D/3D problem	72.13

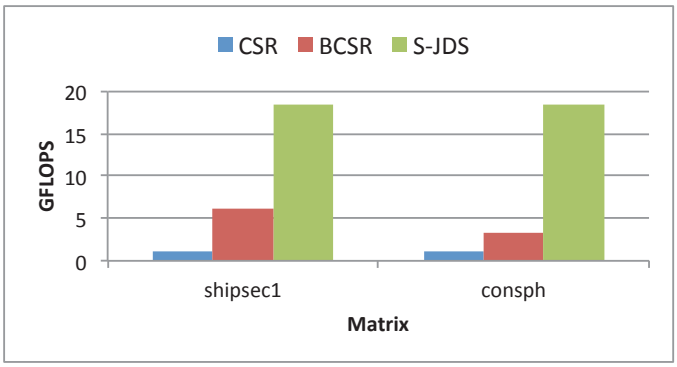


Figure 3.18: Performance comparison between CSR, BCSR, and our SJDS using Fermi-generation GPUs

3.6 Conclusion

SpMV computation is a dominant kernel in many scientific simulations to achieve high performance, and thus very important to achieve high performance in many-core processors, used from the very top class supercomputers in the world, to future tablets and cell phones. Since the capacity of the

cache memory of these accelerators are smaller than that of generic CPUs, and due to the lack of expensive scatter-gather hardware, random access inherent in SpMV computation degrades their performance significantly. We propose a family of new bandwidth reducing sparse matrix formats and the associated algorithms using column-wise segmentation. Segmentation is also applied to the input vector, avoiding random access and reducing cache misses significantly, allowing us to achieve good and stable performance, making good use of the available high sequential memory bandwidth. While there is some tradeoff due to random access of the output vector as well as increase in the sequential memory accesses, overall memory access locality is improved. Performance evaluations show that segmented format on NVIDIA Kepler GPU generally shows superior performance to competing formats and tuned vendor libraries, sometimes by a significant margin. On an Intel Xeon Phi, we find that there is a tradeoff between segmented and no-segmented formats as to which works better, so an auto-selection methods would likely work best overall, with SELL-C-σ being the baseline format.

Chapter 4

Bandwidth Reducing Algorithm for Sparse Matrix Vector Multiplication

4.1 Introduction

Many-core processors such as GPU have numerous computing cores and high-bandwidth memory to increase computational capability. A vast number of codes and algorithms have been ported to many-core processors and successfully accelerated. Especially in case of GPUs, many threads are executed concurrently in order to hide the latency of accessing the device memory. Although this works efficiently for contiguous memory access, random or indirect memory access degrades the performance. Sparse matrix vector multiplication (SpMV) is a memory intensive kernel that is dominant in various numerical algorithms such as conjugate gradient (CG) method and page ranks [5]. SpMV typically requires random memory access to input vector element and, thus, exhibits poor performance. Although there has been many existing work to improve the performance of SpMV computation on GPU or other many-core processors, their improvements are limited to memory bandwidth and they do not exploit the computational capability of many-core processors.

We propose a state-of-the-art sparse format that reduces memory traffic to dramatically accelerate SpMV computation on GPU. Our new format is called the Adaptive Multi-level Blocking (AMB) format and employs multiple levels of division and blocking. The AMB format improves the reusability of input vector elements in GPU cache by column-wise segmentation of the matrix and compresses the column index of each non-zero element. The AMB format requires very little memory traffic in total,

and this is crucial for memory bound kernels. We evaluate the performance of our AMB format for 32 matrix datasets taken from the Sparse Matrix Collection at University of Florida [24], and AMB achieves a maximum speedup of 2.92x and an average of 1.74x over NVIDIA's cuSparse [28]. AMB also reports 1.40x and 1.13x for peak and average speedups, respectively, over yaSpMV [29]—the recently proposed and most notably fast SpMV computation library. To utilize our sparse format in practical situation, we also propose fast auto-tuning mechanism that searches the best set of parameters of AMB for each matrix dataset by estimating the memory traffic and predicting the performance of SpMV computation.

4.2 Sparse Matrix Formats

Sparse matrices in various scientific and engineering applications are generated by using a kind of discretization such as FEM and FDM. Matrix sizes are very large, however, most of the elements are actually zeros. For these matrices, sparse matrix formats are designed to reduce memory usage as well as amount of computations. Coordinated (COO) and Compressed Sparse Row (CSR) are widely-used sparse formats. More recently, numerous sparse formats have been proposed for specific processor architectures or matrices in order to achieve better performance of SpMV computation [11–14]. In particular, many researchers have been optimizing SpMV computation for many-core processors such as GPUs [15, 18, 30]. While CSR format has been extended for GPU [22, 31–34], many sparse matrix formats that store the matrix data in column major ordering have been developed since they have favorable properties for vector processors and many-core processors. JDS (Jagged Diagonal Storage) format [35], which was designed for vector processors, has also been optimized for GPU [19, 20], and ELLPACK [36] based sparse formats have been proposed. Existing sparse formats focused on the issues specific to many-core processors. Some sparse formats alleviate the load imbalance caused by concurrent execution of many threads [8, 37], while other sparse formats improved the locality of memory access to input vector [38, 39].

The ELLPACK format compresses the non-zero elements along each row, and the numbers of elements are made to be same with zero-filling in order to store the matrix data in column major ordering. ELLPACK is suitable if the rows have similar numbers of non-zero elements like a banded matrix. In contrast, it is inefficient if the numbers are not balanced because the number of zero-filling is much larger. There have been several sparse formats extended from ELLPACK to alleviate this problem.

Bell *et al.* proposed the Hybrid format, which combines the ELLPACK and COO formats [22]. The matrix data is basically covered by ELLPACK and remaining part is represented in COO format in order to reduce the number of zero-fillings. Hybrid format is implemented in the cuSparse library provided by NVIDIA as well as the CSR format optimized for NVIDIA's GPU. The library also

provides a subroutine to optimize the number of elements to be covered by ELLPACK, which is the most important parameter in the Hybrid format.

Kreutzer *et al.* proposed the SELL-C-σ format [40], which sorts every σ rows by the number of non-zero elements per row and converts every C rows to the ELLPACK format. By sorting and conversion, SELL-C-σ format effectively reduces the number of zero-fillings. The sorting scope 'σ' is usually smaller than the matrix size to keep memory access locality in the original ordering. The parameter 'C' is the chunk size, which should be set to an architectural property such as vector register length, warp size or wavefront size.

Segmented formats are our previously proposed formats, which focus on the locality of memory access to input vector in SpMV computation [38]. Segmented formats divide a matrix along the column uniformly and each sub-matrix is converted to JDS or SELL-C-σ format, which has favorable properties for many-core processors. Although our Segmented formats are able to improve the reusability of the input vector elements in caches, they proportionally increase the number of additional memory access as the number of the segmentations and total memory traffic in SpMV computation becomes larger. To alleviate this problem, we also proposed Non-Uniformly Segmented (NUS) formats, which reorder a matrix by the number of non-zero elements per row and segments the matrix with a variety of segmentation widths. NUS formats can reduce the number of segmentations with keeping high reusability of the input vector elements in caches. For large size matrices, however, the additional memory access strongly affects the performance down because of its sparsity. Therefore, NUS formats can improve the performance of SpMV only for specific small size matrices.

4.3 Proposal

Our previous proposals, Segmented and NUS formats, successfully improved the locality of memory access to input vector in SpMV computation on GPU. However, total memory traffic became huge and the performance of SpMV computation did not improve for large size matrices. The performance of SpMV computation is strongly limited by the memory bandwidth. Therefore, the reduction of the memory traffic is important to alleviate the memory bound. We propose AMB format, which largely reduces memory traffic not only to matrix data but also to input vector. First, we explain how AMB format reduces memory traffic. Next, we explain the kernel of SpMV computation and estimation of memory traffic. Finally, we explain the scheme of format conversion with auto-parameter tuning.

4.3.1 AMB Format

AMB format aims to reduce the memory traffic in SpMV computation by being constructed in three levels of division and blocking; first level is column-wise segmentation, second level is row-wise slicing

and third level is blocking of column indices. We call column-wise division 'segmentation' and row-wise division 'slicing'. Figure 4.1 shows the state of each level of AMB format.

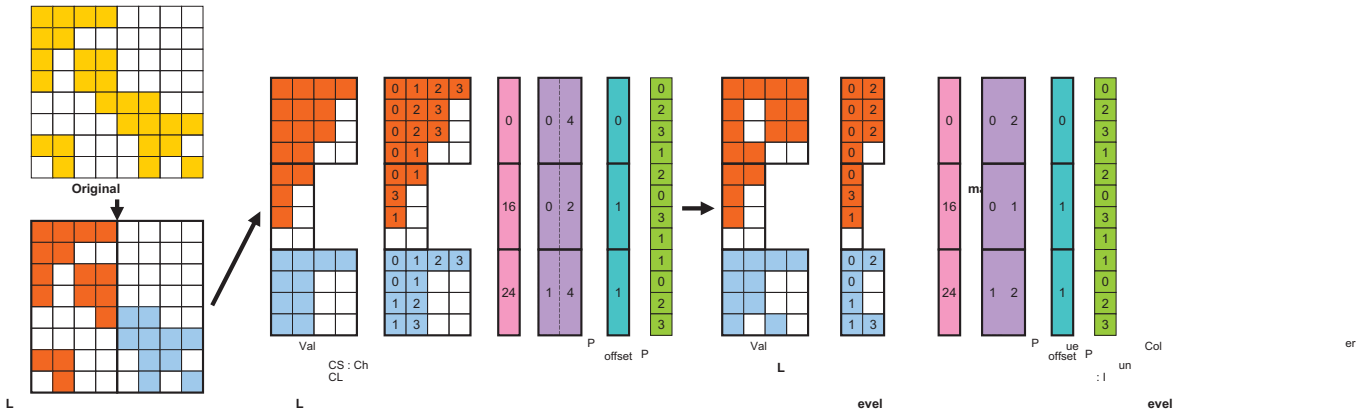


Figure 4.1: AMB format

First level is column-wise segmentation. AMB format sets one segmentation width and divides a matrix along the column uniformly. By executing SpMV computation for each sub-matrix, the locality of memory access to input vector is improved and the memory traffic can be reduced. Although Segmented formats set the segmentation width to match the cache size, AMB format limits the segmentation width to 65,536 ($= 2^{16}$) columns, which is not an architecture specific value, for the compression of column index in the second level division.

Second level is row-wise slicing. Each sub-matrix generated by segmentation in first level is converted into extended SELL-C-σ format. In the same way as SELL-C-σ, AMB format also sorts each σ rows by the number of non-zero elements per row and converts each C rows into ELLPACK in order to reduce zero-filling. AMB format stores the starting offset and the length of each chunk and the original row index of non-zero elements per row represented in 'CS', 'CL' and a pair of 'Permutation offset' and 'Permutation' arrays in Figure 4.1 respectively. There are three extensions from original the SELL-C-σ format. One is the elimination of the empty chunk which has no non-zero elements. Second is the compression of column index by representing 16-bit integer. The AMB format stores the remainder when the original index is divided by segmentation width as column index. The quotient is stored to upper 16-bit of the 'CL' array, while the chunk length can be represented in lower 16-bit integer since segmentation width is not greater than 2^{16} and the chunks, whose length is equal to zero, are removed. Third is the compression of permutation array. The AMB format sets sorting scope as 32,768 ($= 2^{15}$) since the zero-filling is enough reduced without setting larger σ. The AMB format stores the remainder when the original row index is divided by σ as permutation offset and the quotient is stored to 'Permutation'. This extended SELL-C-σ

extremely reduces the memory usage and the memory traffic in SpMV computation from original SELL-C-σ format.

Third level is blocking of column indices along row. Since AMB sorts the rows in second level, blocking is limited in a row. One block size is set for each matrix data and AMB compresses the contiguous column indices. Although the compression rate of column index can be improved when block size is larger, the number of zero-filling in value array may increase in some circumstances. Therefore, the block size should be carefully chosen for each matrix.

Being different from NUS format, the AMB does not reorder the input matrix. Reordering the input matrix can gather the non-zero elements, and thus improve the memory access locality and reduce the amount of zero-paddings. However, the AMB already attain enough memory access locality by dividing the matrix along the column with 2^{16} columns. Since the matrix in AMB format with many zero-paddings comes from blocking optimization of column indices, the reordering optimization is not remedy for increasing zero-padding. Although choosing a block size chunk-by-chunk can reduce the amount of zero-paddings, each chunk needs and stores the block size of each and this increases memory usage and memory access. Also, selecting best block size for each chunk causes non-negligible format conversion cost.

4.3.2 SpMV Kernel in CUDA and Estimation of Memory Traffic

The SpMV kernel of each format affects the performance of SpMV computation and should be carefully implemented to match the architecture and programming language. We propose the SpMV kernel for AMB format in CUDA environment. SpMV computation of AMB is constructed in two CUDA kernels. The first kernel initializes the output vector with zero, and the second kernel executes matrix vector multiplication. In the second kernel, one thread is assigned to each row, and the result of each row is accumulated in the output vector by atomic operation since column-wise segmentation causes conflicting memory accesses to the same output vector elements. If the block size set in the third level is larger, the chunk length of the column index becomes shorter and the number of iteration during the matrix vector multiplication of each row is reduced. This has the same advantage as loop unrolling.

Since SpMV computation is a memory intensive kernel, the performance of SpMV can be predicted by appropriately estimating the memory traffic. We estimate the memory traffic of SpMV computation for the AMB format. The size of matrix is assumed as $M \times N$. We set the variants after format conversion as follows; v is a sizeof floating point number, nz_{zf} is a number of value elements with zero-filling, b is a block size in third level, C is a chunk size in second level and n_C is a number of the chunk. The estimated memory traffic for the matrix data is a sum of 'value', 'column index', 'CS', 'CL', 'Permutation offset' and 'Permutation' since they are read only once. Because of column-wise segmentation in the first level, accessed input vector elements will be in the cache and the number of

global memory access to each input vector element is expected to be one. For the output vector, in addition to initialization in first kernel of SpMV computation, atomicAdd, which requires one read and one write memory accesses, is executed for each row. In the result, the estimated memory traffic in SpMV computation for AMB format is: $nz_{zf} * (v + 2/b) + n_C * (2 * v * C + 2 * C + 10) + (M + N) * v$.

4.3.3 Format Conversion for GPU with Parameter Auto-Tuning

For AMB format, the set of parameters such as block size affects the performance of SpMV computation. However, searching all combination of the parameters with executions of SpMV requires expensive preprocessing and it is inefficient in case of real applications. We propose the conversion scheme for the AMB format to minimize the cost of searching the parameter space. We devise a fast format conversion using GPU and auto-tuning mechanism, which searches the best set of parameters of AMB format by estimating the memory traffic and the performance of the SpMV computation. Algorithm 1 shows the auto-tuning mechanism for AMB format. In this scheme, the matrix data in CSR format is stored on GPU as input data and is converted into AMB format. The conversion is done from first level segmentation to third level blocking. In third level, this scheme searches the best block size for each matrix. The scheme converts the matrix data into the AMB format with setting the block size from 1 to 10 and estimates the memory traffic. When the mechanism searches the best block size in the third level, it does not require redundant conversion of first level segmentation and second level slicing. The block size with minimum estimated memory traffic is regarded as best block size. After the mechanism finds out the best block size, it starts searching for best thread block size, which affects the performance of GPU executions. The mechanism evaluates the performance of the SpMV computation by setting the thread block size as 64, 128, 256 or 512 and executing the kernel.

Algorithm 1 Format conversion scheme with parameter auto-tuning

Input: Matrix data in CSR

Output: Matrix data in AMB with Best Set of Parameters

1: Column-wise segmentation (segment width = 2^{16})
2: Convert each segment to SELL-C-σ ($C = 32$, $\sigma = 2^{15}$)
3: Eliminate empty chunks and compress column index and permutation to 16bit
4: block_size $\leftarrow 1$
5: **while** block_size ≤ 10 **do**
6: Blocking with block_size
7: Estimate the memory access and update best block size
8: block_size $\leftarrow$ block_size $+ 1$
9: **end while**
10: Blocking with best block size
11: thread_block_size $\leftarrow 64$
12: **while** thread_block_size ≤ 512 **do**
13: Execute SpMV and update best thread block size
14: thread_block_size $\leftarrow$ thread_block_size $* 2$
15: **end while**

4.4 Performance Evaluation

We compare the performance of our AMB format and major existing sparse formats, and evaluate the cost of conversion to AMB format. For our performance evaluations, we utilize the matrices from the Sparse Matrix Collection at University of Florida [24]. The name and size of the matrices are summarized in Table 4.1. We selected 32 positive definite symmetric matrices whose row size is larger than 131,072 ($= 2^{17}$) from the collection.

Table 4.1: Matrix data

Matrix	Row size	Non-zeroes	nz/row
af_0_k101	503,625	17,550,675	34.85
af_shell3	504,855	17,588,875	34.84
apache2	715,176	4,817,870	6.74
audikw_1	943,695	77,651,847	82.28
BenElechi1	245,874	13,150,496	53.48
bmw7st_1	141,347	7,339,667	51.93
bmwcra_1	148,770	10,644,002	71.55
bone010	986,703	71,666,325	72.63
boneS10	914,898	55,468,422	60.63
Dubcova3	146,689	3,636,649	24.79
ecology2	999,999	4,995,991	5.00
Emilia_923	923,136	41,005,206	44.42
Fault_639	638,802	28,614,564	44.79
Flan_1565	1,564,794	117,406,044	75.03
G2_circuit	150,102	726,674	4.84
G3_circuit	1,585,478	7,660,826	4.83

Matrix	Row size	Non-zeroes	nz/row
Geo_1438	1,437,960	63,156,690	43.92
hood	220,542	10,768,436	48.83
Hook_1498	1,498,023	60,917,445	40.67
inline_1	503,712	36,816,342	73.09
ldoor	952,203	46,522,475	48.86
msdoor	415,863	20,240,935	48.67
offshore	259,789	4,242,673	16.33
parabolic_fem	525,825	3,674,625	6.99
pwtk	217,918	11,634,424	53.39
Serena	1,391,349	64,531,701	46.38
shipsec1	140,874	7,813,404	55.46
shipsec5	179,860	10,113,096	56.23
StocF-1465	1,465,137	21,005,389	14.34
thermal2	1,228,045	8,580,313	6.99
thermomech_dM	204,316	1,423,116	6.97
tmt_sym	726,713	5,080,961	6.99

We evaluate the performance of SpMV on NVIDIA Tesla K20X GPU installed on TSUBAME-KFC. The GPU has 6 GBytes of device memory, being sufficient for storing all the matrices used in this performance evaluations. The bandwidth of device memory is up to 250 GB/sec and ECC is disabled. The Kepler-generation GPU has 1.5 MByte L2 cache memory shared by 14 SMX, and each SMX has 48 KByte read-only cache. Input vector elements and arrays representing the starting offset and size of each chunk in SELL-C-σ and AMB format (CL and CS arrays in Figure 4.1) and 'Permutation offset' in AMB format are assigned to read-only cache. Memory accesses to the other arrays bypass the read-only cache. GPU codes are implemented in CUDA 7.0 and are executed on CentOS Linux 6.4 OS 64bit.

4.4.1 Performance of SpMV

For the performance evaluation of SpMV computation, we utilize two sparse matrix libraries, cuSparse and yaSpMV [29], and also we prepare SELL-C-σ, Segmented-SELL-C-σ and NUS-SELL-C-σ formats. We select CSR, Hybrid and BSR (Block CSR) from cuSparse library, and the best result of three formats is represented as the performance of cuSparse. The block size of BSR's parameter is set to from 1 to 4. BCCOO format and its conversion scheme are provided in yaSpMV library by Yan *et al.* We add yaSpMV library for the performance evaluation because it achieved large performance gain compared to existing SpMV library including cuSparse. In the evaluation, yaSpMV library searches the best parameters for each matrix data, and the result of SpMV using those best parameters is used as

the performance of yaSpMV. The kernel in double precision is not provided by yaSpMV. We evaluated the performance of SELL-C-σ with various sorting scopes and select best one for each matrix. For AMB format, SpMV is executed for all set of parameters and the best result is regarded as AMB's performance; segmentation width in first level is 65536 ($= 2^{16}$), block size in third level is set from 1 to 10, and thread block size is set as 64, 128, 256 or 512.

Figure 4.2 shows the performance of SpMV computations in single precision. Matrices are listed in ascending order of the average number of non-zero elements per row (= nz/row). If nz/row is larger than 24, the performance of SpMV computation is high, and especially the speedup of AMB format is superior to other formats. Our AMB format achieves speedups of 2.92x on maximum and 1.74x on average compared to cuSparse library, and 1.40x on maximum and 1.13x on average compared to yaSpMV library. Figure 4.3 shows the result of double precision. As well as single precision, our AMB format achieves high speedups compared to existing formats. However, since the memory traffic of value data is larger and the compression rate becomes lower, the speedups of AMB format are relatively small.

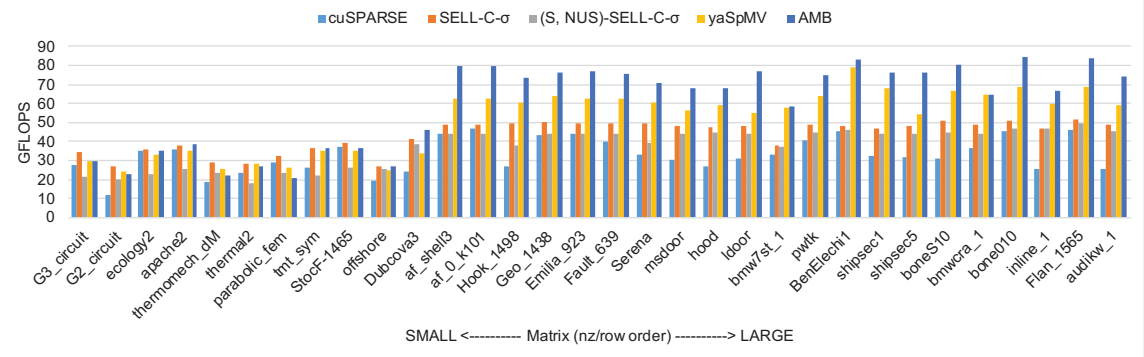


Figure 4.2: Performance of SpMV computation in single precision

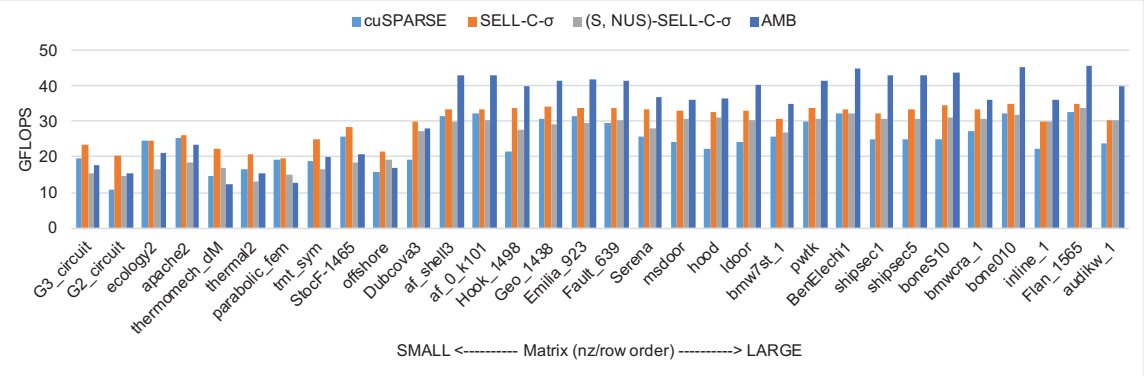


Figure 4.3: Performance of SpMV computation in double precision

Figure 4.4 shows the speedups of each level compared to SELL-C-σ format in single precision. For Seg-SELL-C-σ, the segmentation width is set to 2^{16}. Seg-SELL-C-σ employs only first level segmentation and second level slicing. For the large size matrices utilized in this paper, Seg-SELL-C-σ does not outperform SELL-C-σ because the additional memory traffic, which Seg-SELL-C-σ format requires, is larger than the reduced memory traffic for input vector by column-wise segmentation. AMB (Level 1, 2) sets the block size in third level to 1. For most of matrices, AMB (Level 1, 2) achieves speedups compared to SELL-C-σ format. We find the elimination of needless chunks and compression of the column index to 16-bit integer contribute the speedups of the performance of SpMV. AMB selects best block size in third level, and shows further speedups. The result also shows the blocking optimization in third level does not improve the performance when nz/row is small. This is because there are not enough non-zero elements which have contiguous column indices in each row.

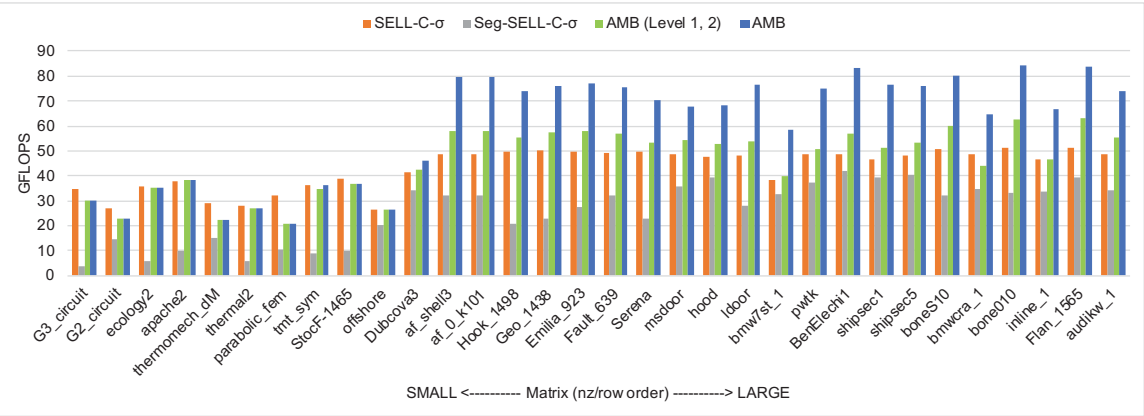


Figure 4.4: Speedup of each level compared to SELL-C-σ format in single precision

Figure 4.5 shows the ratio of the memory traffic compared to CSR format for each matrix data. We evaluate the memory traffic of SELL-C-σ, BCCOO from yaSpMV and AMB format. The result shows AMB format effectively reduces the memory traffic if nz/row is larger. That is why AMB format shows superior performance to other formats. On the other hand, in smaller nz/row, the memory traffic of AMB format is not minimum in compared formats, and SELL-C-σ is most efficient format in terms of memory traffic. For these matrices, AMB format requires atomicAdd operations for output vector, which occupy large percentage in total memory traffic. For most of matrix data, the format with minimum memory traffic shows best performance of SpMV computation compared to other sparse formats.

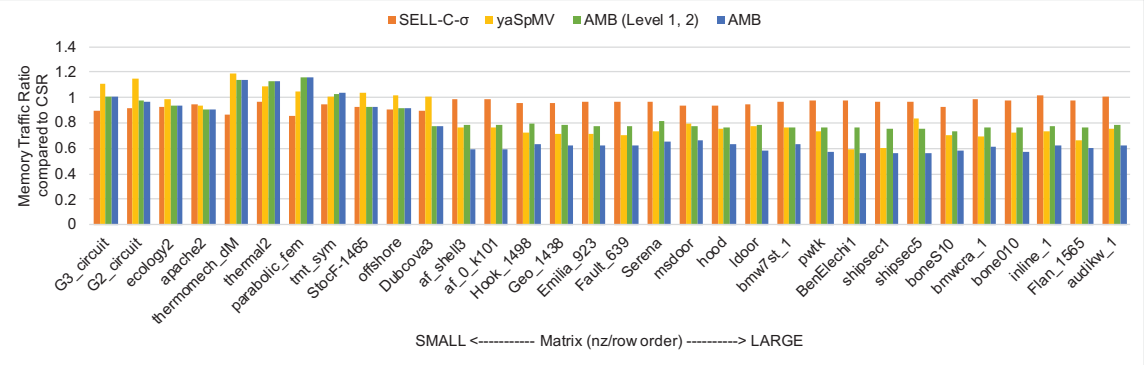


Figure 4.5: Memory Access Ratio in single precision

4.4.2 Evaluation of CG method

The execution time of SpMV occupies large amount of that of CG method, and we expect that accelerating SpMV kernel can reduce total execution time for CG method. We evaluate the performance of CG method on Kepler generation GPU with CSR, SELL-C-σ and AMB formats in double precision. Selected matrices are 'audikw_1', 'bmwcra_1', 'bone010', 'Serena', 'Flan_1565' and 'af_shell3'. Table 4.2 shows the number of iterations until convergence in CG method for each matrix.

Table 4.2: Iterations for the convergence in CG method in double precision

Matrix	#Iterations
audikw_1	112,310
bmwcra_1	18,488
bone010	23,196
Serena	69,851
Flan_1565	33,829
af_shell3	5,001

Figure 4.6 shows the ratio of execution time including format conversion to that of CSR format. Although our AMB format requires slightly more execution time for format conversion, total execution time is largely reduced with AMB format compared to CSR and SELL-C-σ formats. The result shows almost same speedups as SpMV performance without format conversion. The cost of format conversion on 'af_shell3', where the number of iterations until convergence is small, is relatively larger, while other matrices require negligible format conversion cost. Our AMB format can accelerate iterative method including format conversion cost.

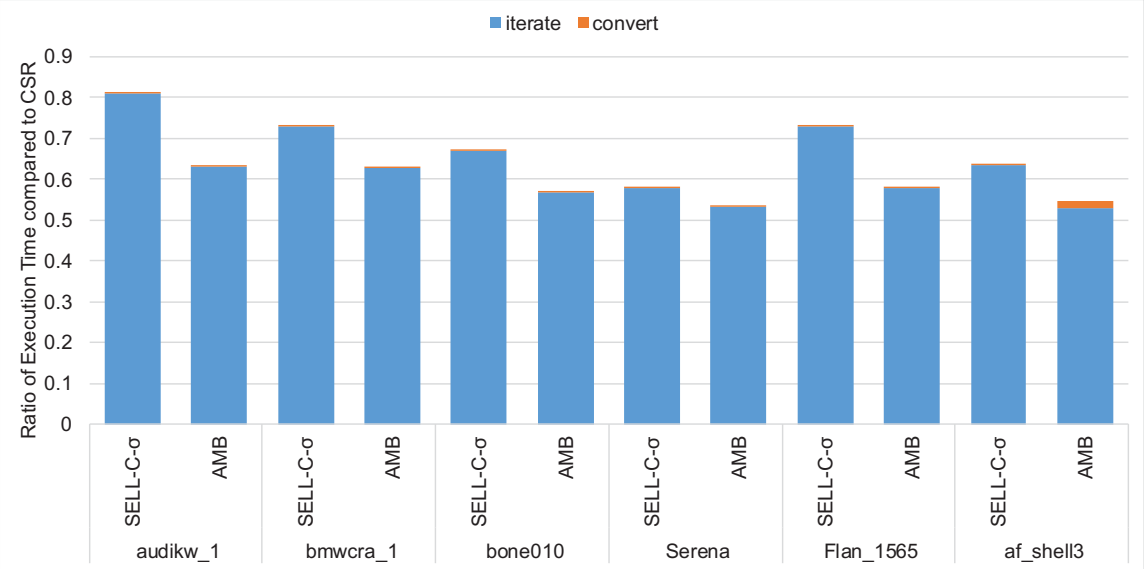


Figure 4.6: Execution time of CG method on Kepler generation GPU

4.4.3 Evaluation of Format Conversion

Auto parameter tuning mechanism determines the parameters of AMB format based on the estimation of memory traffic. We first evaluate the correctness of the estimation of the memory traffic and evaluate the adequacy of the choice of the parameters, that is, how the estimation affects the performance of SpMV computation. Finally, we evaluate the cost of conversion including parameter tuning.

Figure 4.7 shows the measured and estimated memory traffic of AMB format with best set of parameters. As the correlation coefficient between the measured and estimated memory traffic is 0.999, which is enormously high score, the estimation of the memory traffic is correct. Although the memory traffic is estimated smaller compared to measured one in some cases, this problem comes from cache miss of access to output vector.

We evaluate whether the estimation of memory traffic predicts the performance of SpMV. Figure 4.8 shows the estimation of the memory traffic and execution time of SpMV computation for AMB format with best set of parameters. The performance of SpMV computation depends on the memory traffic and we are able to predict the performance of SpMV computation by estimating the memory traffic. For several matrices, the execution time of SpMV computation is longer than predicted. We find load imbalance degrades the performance of SpMV computation for these matrices. In addition, we evaluate the estimation of memory traffic and the performance of SpMV computation by changing the block size in third level. Table 4.3 shows the result of 'audikw_1' and 'Dubcova3'. The block size which requires smaller memory traffic basically shows better performance of SpMV computation. Therefore, we are able to predict the best block size in third level by estimating the memory traffic. However, if the difference between the estimation of memory traffic is small such as 'Dubcova3', the prediction of best block size in third level may fail.

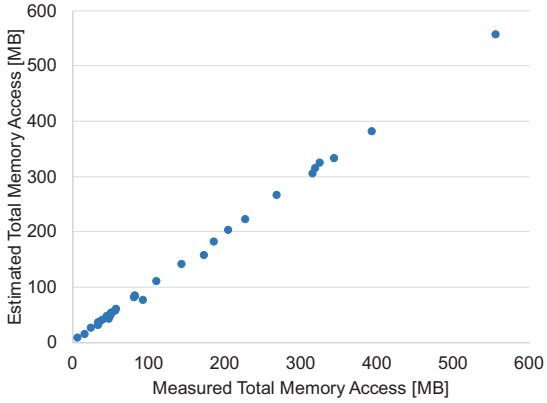


Figure 4.7: Measured and estimated memory traffic

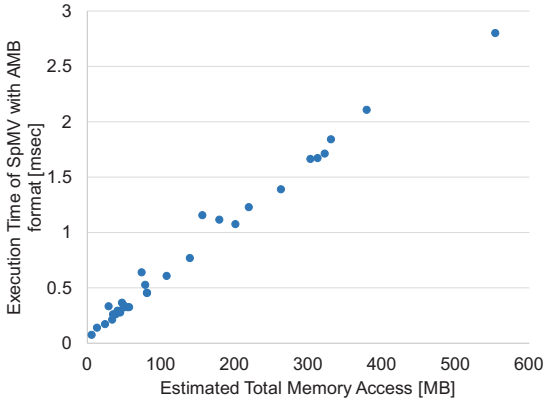


Figure 4.8: Memory traffic and execution time

Table 4.3: Memory traffic and execution time for different block size

Matrix	Block size	Memory trafic [MB]	Execution time [msec]
audikw_1	1	484.41	2.819
	2	491.32	2.671
	3	385.31	2.180
	4	479.30	2.600
	5	559.71	3.014
	6	571.99	3.057
	7	643.70	3.395
	8	704.83	3.733
Dubcova3	1	25.78	0.171
	2	26.06	0.160
	3	33.74	0.196
	4	35.53	0.202
	5	41.72	0.234
	6	47.01	0.261
	7	52.73	0.286
	8	56.75	0.309

We evaluate the performance of SpMV computation with auto-tuning mechanism for 32 matrix data. In the result, the mechanism selects best set of parameters for 29 matrix data in single precision and 28 in double precision. The performance down is up to 8.01% in case that the best block size for matrix data is not precisely selected. We conclude that our auto-tuning mechanism selects enough appropriate parameters. The average cost of format conversion into AMB format is 304 times execution of SpMV computation for AMB format, which can be negligible in iterative method such as CG method.

We evaluate the cost of format conversion with auto-tuning mechanism. Figure 4.9 shows the breakdown of format conversion for each matrix data. 'Level 1' includes the scheme of column-wise segmentation. 'Level 2' includes the row-wise slicing and sorting scheme before row-wise slicing corresponds to 'Sorting'. The scheme about the elimination of needless chunks and the compression of column index into 16-bit integer corresponds to 'Compression'. Since segmentation width in first level is set to 2^{16} for all matrix data used in this paper, the schemes from 'Level 1' to 'compression' are executed only once for each matrix data. In 'Level 3', the matrix data is converted with changing the block size from 1 to 8. The mechanism searches the best block size with format conversion and estimation of memory traffic in SpMV computation. In 'Thread Block Tuning', SpMV computations are executed with changing thread block size, and the mechanism searches best thread block size. Thread block size is set as 64, 128, 256 or 512, and SpMV computation is executed twice for each

thread block size. The average rate of practical parameter search about block size in third level and thread block size is only 23% of the total format converting time. Therefore, our auto-tuning mechanism works efficiently for format conversion. The result shows the average conversion cost is 662 times execution of SpMV computation for AMB format, which can be negligible in iterative method such as CG method. In order to reduce the conversion cost, we should optimize sorting phase.

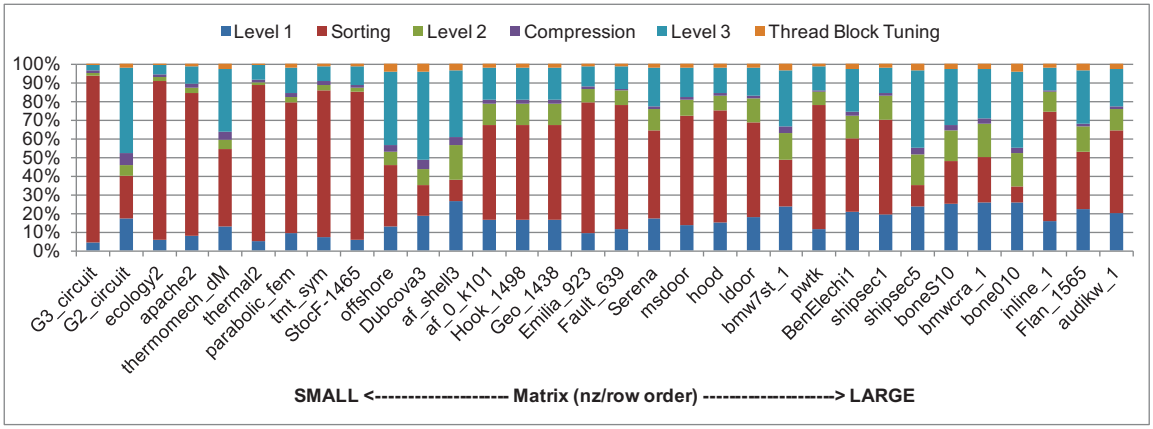


Figure 4.9: Breakdown of format conversion cost

4.5 Related work

There have been several past work focusing on reducing memory traffic on GPUs in SpMV computation.

Maggioni *et al.* proposed Compressed Adaptive ELL (CoAdELL) format, where column index is represented in 16-bit or 8bit integer [41]. CoAdELL format is based on delta encoding technique to utilize the column distances between contiguous non-zero elements in each row. If the difference of column indices between non-zero elements is small, the distances can be represented in 16-bit or 8-bit integer and the memory footprint is reduced. CoAdELL format makes a judgmental decision of compression for the column indices in warp granularity. Since CoAdELL cannot compress the column index when the non-zero elements is scattered, the compression rate of CoAdELL is not high. AMB format ensures the compression to 16-bit integer for any matrices.

Tang *et al.* proposed Bit-Representation-Optimized (BRO) compression schemes for GPU [16]. BRO format is also based on delta encoding and the difference of column indices is represented in more precise number of bits. Although these bit representation techniques have been proposed for CPU, they do not work efficiently on GPU because the schemes sequentially execute. BRO format is optimized for GPU by removing these processing. Although BRO format successfully reduces the

memory footprint, the performance of SpMV computation is not improved so much since it requires additional decompression schemes.

Yan *et al.* proposed BCCOO format, which focuses on both reduction of memory traffic for matrix data and improving load balance [29]. BCCOO stores the matrix data in extended COO format with blocking the matrix to reduce the memory usage of the column and row index. Row index is represented in a bit-flag being different from original COO format. When the computation result of each block is accumulated in SpMV computation, local memory such as shared memory on GPU is effectively utilized to avoid the access to global memory. Since BCCOO format requires the precise setting of many parameters such as block size and strategy in SpMV computation, yaSpMV library provides the auto-tuning mechanism. However, the cost of preprocessing of parameter tuning is extremely huge since the parameter space of BCCOO format is large. Our performance evaluation shows AMB format reduces the memory traffic and achieve better performance in SpMV computation compared to BCCOO format since BCCOO format itself is based on COO format, which requires high memory bandwidth.

4.6 Conclusion

Accelerating SpMV computation, which is a dominant kernel in many scientific calculations, is one of the critical issues for scientific and engineering applications. We propose a sparse matrix format, called AMB, that overcomes the two major bottlenecks of SpMV computation: memory intensive kernel and high cache miss rates caused by random memory accesses to input vector elements. AMB employs multiple levels of division and blocking optimization techniques to compress column indices and improve input vector access locality. This results in lower cache miss rates and significantly reduced memory traffic. AMB achieves superior performance to competing formats on NVIDIA Kepler GPU, demonstrating up to 2.92x speedup over NVIDIA's cuSparse and up to 1.40x over the notable yaSpMV. We also find out the performance of SpMV computation strongly depends on the memory traffic, and we devise the auto-tuning mechanism for AMB format with best set of parameters by estimating memory traffic precisely.

Chapter 5

Optimization Strategy of Sparse General Matrix Matrix Multiplication on GPU

5.1 Introduction

Numerical applications such as scientific simulations and graph processing often require sparse matrix computation. Algebraic multigrid (AMG) method [42] for preconditioner of iterative method in solving equations and some graph algorithms such as graph clustering [43] and breadth-first search (BFS) [44] compute matrix multiplication of sparse matrices. Generally, sparse matrix data is compressed by holding only non-zero elements to reduce memory usage and amount of computation. In sparse general matrix-matrix multiplication (SpGEMM), input and output matrices should be compressed. However, the number and pattern of non-zero elements in the output matrix depend on the input matrices and are unknown before execution. This property makes it hard to allocate memory of the output matrix. Furthermore, unlike dense matrix computations, SpGEMM causes frequent cache misses because of indirect memory access to the input matrix [45].

Recently, Graphics Processing Unit (GPU) is utilized for general purpose computations. The computational kernels are accelerated by GPU's high computing power. Sparse matrix computations are also ported to GPUs [22, 31, 32], especially, sparse matrix vector multiplication (SpMV) is highly accelerated due to high parallelism and memory bandwidth of GPU. Although SpGEMM is also accelerated by GPU [42, 46], existing approaches require much larger memory compared to final usage of output matrix because it is not easy to manage the memory allocation in massive thread

executing. As a result, the applicable matrix data is limited by the capacity of GPU's device memory. Furthermore, since high parallelism and various types of memory on GPU are not utilized efficiently, SpGEMM is not fully accelerated on GPU devices.

We propose a state-of-the-art algorithm which not only accelerates SpGEMM computation on GPU but also reduces memory usage to compute various matrix data. Our approach utilizes GPU's on-chip shared memory and assigns GPU resources in a step-by-step manner by grouping the rows based on the number of non-zero elements or computational complexity. We evaluate the performance of our SpGEMM algorithm for various kinds of 12 matrix datasets taken from University of Florida Sparse Matrix Collection [24]. Compared to existing sparse libraries: CUSP, cuSPARSE and BHSPARSE, we achieve the maximum speedups of 4.3x in single precision and 4.4x in double precision on brand new NVIDIA's Pascal generation GPU. The memory usage of our approach is reduced from other sparse matrix libraries for all matrix datasets, and we achieve 14.7% reduction on average in single precision and 10.9% in double precision. We also examine the performance of SpGEMM for large size matrix data generated in graph analysis. BHSPARSE and CUSP cannot handle some of these large size matrix data since they consume much memory space. For these matrices, our proposal shows significant speedups up to 11.6x compared to cuSPARSE with low memory usage.

5.2 Background

We first introduce sparse matrix format which reduces both memory usage and computations. Then we describe the sequential SpGEMM method and discuss the problems of SpGEMM including GPU case.

5.2.1 Sparse Matrix Format

When a matrix has many zero elements, the matrix is compressed by removing zero elements and the sparse matrix format holds only non-zero elements and pointers/indices. Coordinated (COO) and Compressed Sparse Row (CSR) are widely-used sparse matrix formats. COO format simply contains the array of tuples of row, column and value for each non-zero element. CSR format has an array of pointers to the beginning of each row, called row pointer (rpt), instead of holding row indices of each non-zero element. CSR format requires less memory usage compared to COO format.

In order to accelerate SpMV, many of sparse matrix formats have been proposed [11, 47], and recently some sparse matrix formats show the outstanding performance of SpMV on GPUs [40, 48]. These sparse matrix formats usually require format conversion from standard format such as CSR or COO format. In iterative methods which require many times of SpMV, the acceleration of SpMV with converting matrix to other sparse matrix format is an useful way since the cost of format conversion

can be negligible. On the other hand, when the execution time of sparse matrix computation is relatively small, the computation should be executed without format conversion since the cost of format conversion becomes relatively large.

5.2.2 Sparse General Matrix-Matrix Multiplication

We first describe the sequential basic SpGEMM. Let A, B be input matrices, and SpGEMM computes a matrix C such that $C = AB$. The input and output matrices are sparse and they are stored in sparse matrix format. When the matrices are stored in CSR format, SpGEMM computation is executed for each row of output matrix. Algorithm 2 shows the pseudo code of SpGEMM in CSR format. Let a_{ij} be the element in i-th row and j-th column of matrix A and a_{i*} be the i-th row of matrix A. SpGEMM is executed for each row of matrix A, that is, matrix C. The row of matrix B corresponding to each non-zero element of matrix A is read, and each non-zero element of output matrix C is calculated.

Algorithm 2 Pseudo code of SpGEMM

```
set matrix C to ∅
for all a_{i*} in matrix A do
  for all a_{ik} in row a_{i*} do
    for all b_{kj} in row b_{k*} do
      value ← a_{ik} b_{kj}
      if c_{ij} ∈ c_{i*} then
        insert (c_{ij} ← value) to c_{i*}
      else
        c_{ij} ← c_{ij} + value
      end if
    end for
  end for
end for
```

SpGEMM computation has three critical issues unlike dense matrix multiplication. First issue is indirect memory access to matrix data. According to Algorithm 2, the memory access to matrix B depends on the non-zero elements of matrix A. Therefore, the memory access to each non-zero element of matrix B is indirect, and the cache miss may occur frequently. Second is that the pattern and the number of non-zero elements of output matrix are not known beforehand. For this reason, the memory allocation of output matrix becomes hard, and we need to select from two strategies. One is a two-pass strategy, which counts the number of non-zero elements of output matrix first, and then allocates memory and computes. The other is that we allocate enough large memory space for output matrix and compute. The former requires more computation cost, and the latter uses much more memory

space. Finally, third issue is about combining the intermediate products (*value* in Algorithm 2) to non-zero elements of output matrix. Since the output matrix is also sparse, it is hard to efficiently combine intermediate products. This procedure becomes a performance bottleneck of SpGEMM computation.

SpGEMM has been accelerated on GPU from CPUs by exploiting its high parallelism and computing power [49]. In GPU case, we should additionally consider how to improve the load balance between threads. Since the computation cost of each row largely varies, the threads should be carefully assigned to rows. Bell et al. proposed ESC algorithm, which decomposes SpGEMM computation to GPU friendly procedures in order to improve load balance [42]. The ESC algorithm consists of three phases; expansion, sorting and contraction. First, the algorithm generates the list of tuple of row index, column index and value for all intermediate products of matrix-matrix multiplication. Next, the entries in the list are sorted by the row and column indices. Finally, the intermediate products with same row and column indices are accumulated, and the algorithm outputs the sets of row index, column index and value of each non-zero element. Each phase of ESC algorithm exploits high parallelism of GPU. However, since ESC algorithm handles extremely large amount of intermediate data, the performance of SpGEMM is low and the large memory usage is required. For these problems, Dalton et al. improved ESC algorithm by utilizing on-chip shared memory [50]. Nevertheless, the performance is still low since the base of the algorithm is ESC and the problem is not fundamentally solved.

5.3 Proposal

Many of researchers have proposed new SpGEMM algorithms for GPUs. However, the acceleration comes from requiring large amount of working memory. We propose the state of the art algorithm which accelerates SpGEMM on GPU and reduces memory usage. Our algorithm allocates hash table on fast shared memory and appropriately divides the rows into groups in order to exploit GPU resource. Figure 5.1 shows the flow of our proposed SpGEMM algorithm. Our proposal consists of two phases not to require redundant working memory. First phase for counting the number of non-zero elements of output matrix is from (1) to (4), and second phase for calculating the output matrix is (6) and (7) in Figure 5.1. In order to exploit GPU resource more efficiently including better load balancing, the rows are divided into groups based on the number of intermediate products in (2) and on the number of non-zero elements in (6).

We describe three key phases: grouping, counting the number of non-zero elements and calculation of output matrix.Finally, we describe the parameters setting for each group. In the following of this paper, let A, B be input matrices and C be output matrix, and we compute $C = AB$. All input and output matrices are stored in CSR format.

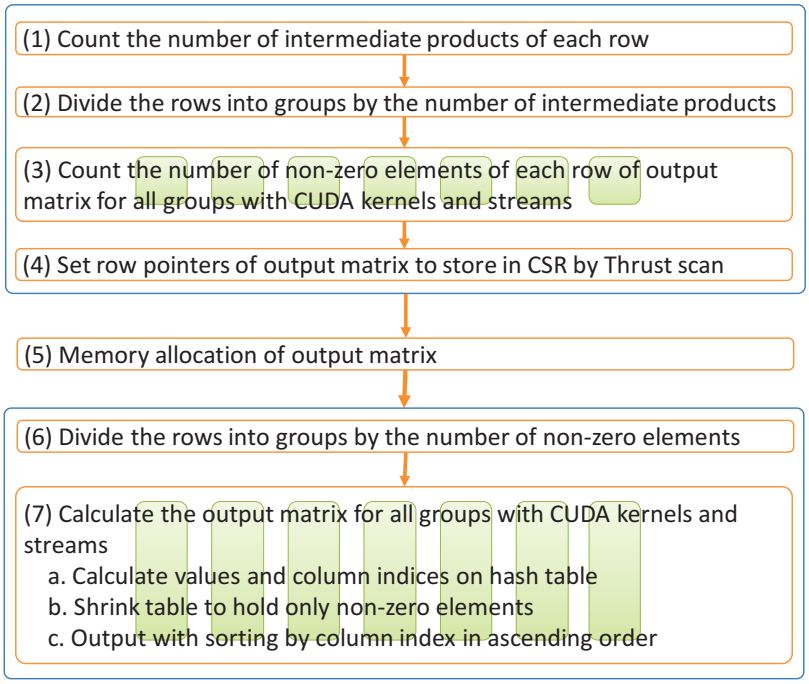


Figure 5.1: Flow of our SpGEMM algorithm

5.3.1 Grouping

Since the computational cost of each row varies in SpGEMM computation, simple parallelization may cause load imbalance. In order to improve the load balance and to exploit GPU resources such as shared memory, our algorithm divides the rows into groups by the number of intermediate products or non-zero elements of output matrix. The number of non-zero elements of each row in output matrix is unknown at grouping phase (2). Therefore, after counting the intermediate products of each row in (1), the rows are divided into groups by the number of intermediate products. In grouping phase (6), the rows are divided into groups based on the number of non-zero elements, which are counted in (3). Since the execution cost of reordering whole input matrix is too large, the grouping is done by generating the array of gathered row indices corresponding to each group. This array size is additional memory usage on GPU in our algorithm.

Counting the number of intermediate products

Algorithm 3 shows how to calculate the number of intermediate products of each row in (1). The number of intermediate products is an upper limit of the number of non-zero elements of output

matrix. Since this procedure requires only the row pointers and column indices of matrix A and the row pointers of matrix B, the execution cost is relatively small compared to whole SpGEMM execution.

Algorithm 3 Count the number of intermediate products of i-th row

$n_{prod} \leftarrow 0$
for $j = rpt_A[i]$ to $rpt_A[i+1]$ **do**
  $n_{prod} \leftarrow n_{prod} + (rpt_B[col_A[j]+1] - rpt_B[col_A[j]])$
end for

5.3.2 Counting the number of non-zero elements

The number of intermediate products is typically much larger than the number of non-zero elements in output matrix. This is because same column may be counted for multiple times. We count the number of non-zero elements of each row in output matrix with considering overlapping of the column indices. We improve memory access to input matrices on GPU, and employ hash tables to manage the column indices with considering overlapping. In order to exploit GPU resources and improve the load balance, we set different hash table size, thread assignment and thread block size for each group. Our approach launches multiple CUDA kernels with different CUDA streams for each group to execute the kernels in parallel.

Thread Assignments and Memory Access Optimization

We combine two-way thread assignments; PWARP/ROW and TB/ROW, where partial warp is bundle of 4 threads, and TB means thread block. Our algorithm sets better one for each group to improve the load balance. The PWARP/ROW assigns 4 threads for each row in matrix A, one thread for each non-zero element of matrix A and corresponding row of matrix B. Algorithm 4 shows how to count the number of non-zero elements by PWARP/ROW. We did preliminary evaluation with changing the number of threads per row as 1, 2, 4, 8 and 16. In the result, 4 threads per row stably shows best performance. Thus, we select 4 threads as PWARP. Since the PWARP/ROW executes the procedure of each row by only 4 threads, the PWARP/ROW is selected for the groups with small numbers of intermediate products or non-zero elements. The warp shuffle function is utilized for accumulating the numbers of non-zero elements of each thread in same PWARP, and the numbers of non-zero elements in each row are output.

Algorithm 4 Count the number of non-zero elements of i-th row by PWARP/ROW

```
tid ← threadIdx%4
for j ← rpt_A[i] to rpt_A[i + 1] stride 4 do
  d ← col_A[j + tid]
  for k ← rpt_B[d] to rpt_B[d + 1] stride 1 do
    //hash operation
  end for
end for
```

The TB/ROW assigns one thread block for each row of matrix A, one warp for each non-zero element of matrix A and one thread for each non-zero element of matrix B. Algorithm 5 shows how to count the number of non-zero elements by TB/ROW. Since the procedure of each row is executed by many threads, the TB/ROW is selected for the groups which have large numbers of intermediate products or non-zero elements. To calculate the number of non-zero elements of each row, counted numbers of non-zero elements by each thread in same warp are accumulated by the warp shuffle function as well as PWARP/ROW, and then the TB/ROW accumulates the sum of each warp by utilizing shared memory.

Algorithm 5 Count the number of non-zero elements of i-th row by TB/ROW

```
tid ← threadIdx%warpsize
wid ← threadIdx/warpsize
wnum ← blockDim/warpsize
for j ← rpt_A[i] + wid to rpt_A[i + 1] stride wnum do
  d ← col_A[j]
  for k ← rpt_B[d] + tid to rpt_B[d + 1] stride 32 do
    //hash operation
  end for
end for
```

Hash Table

It is most time-consuming part in SpGEMM computation to combine the intermediate products with same row and column indices into one non-zero element. Our proposal utilizes hash table to accelerate SpGEMM computation. The hash table is utilized not only for counting the numbers of non-zero elements of each row in (3), but also for calculation of values of output matrix in (7). Algorithm 6 shows hash algorithm of our proposal. The column indices are inserted into hash table as keys. The hash table is initialized by storing -1 since the column indices are no less than 0. The column index is multiplied by constant number, $HASH\_SCAL$, and divided by hash table size to compute the

remainder. The *hash* in Algorithm 6 is the result of this calculation. The hashing algorithm is based on linear probing. When the hash is collided, the hashing algorithm tries next entry until occurring no hash collision. Since the entry in hash table would be accessed by multiple threads simultaneously, the operation should be executed exclusively by compare-and-swap (atomicCAS).

Algorithm 6 Hash Algorithm

```
(Hash table is initialized: table[] ← −1)
(nz is initialized: nz ← 0)
(k comes from Algorithm 4, 5)
key ← col_B[k]
hash ← (key * HASH_SCAL)%t_size
while true do
  if table[hash] = key then
    break
  else if table[hash] = −1 then
    old ← atomicCAS(table + hash, −1, key)
    if old = −1 then
      nz ← nz + 1
      break
    end if
  else
    hash ← (hash + 1)%t_size
  end if
end while
```

The size of hash table (t_{size} in Algorithm 6) is set based on the number of intermediate products or non-zero elements. The hash table can be allocated on shared memory except the group for large number of non-zero elements. For the group with large number of intermediate products, the process of counting the number of non-zero elements is executed in two phases. First phase tries to count the number of non-zero elements of each row, using the maximum hash table size for shared memory. When a row finds larger number of non-zero elements than the hash table size, the thread block records the row index, and immediately terminates its execution. After this first phase, our algorithm counts the numbers of non-zero elements for recorded rows by allocating enough large hash table on global memory. The size of hash table on global memory is set based on the number of intermediate products. Since the number of non-zero elements is usually much smaller than the number of intermediate products, most of rows complete in the first phase.

5.3.3 Calculation of Output Matrix

Calculation of output matrix C consists of three phases. First phase computes the values and column indices of output matrix by utilizing hash table as well as counting the number of non-zero elements of each row in C. Our algorithm employs another table for the calculation of values. After the calculation of each row finishes, the hash table contains the computed non-zero elements. Second phase gathers non-zero elements in the hash table. Finally, third phase sorts the non-zero elements in ascending order of column indices. The thread is assigned to each non-zero element in hash table, and computes the order of the column index of the non-zero element by comparing the other non-zero elements in same hash table. According to this order, the non-zero elements are output to global memory and SpGEMM computation finishes. When the hash table can be allocated on shared memory, all phases of calculating output matrix are executed on shared memory.

5.3.4 Parameter Setting for Each Group

Table 5.1: Parameter setting for each group on Tesla P100

Group ID	**(3) flop of row**	**(6) nnz of row**	**Assignment**	**Thread Block size**	**#TB**
0	8193-	4097-	TB/ROW	1024	2
1	4097-8192	2049-4096	TB/ROW	1024	2
2	2049-4096	1025-2048	TB/ROW	512	4
3	1025-2048	513-1024	TB/ROW	256	8
4	513-1024	257-512	TB/ROW	128	16
5	33-512	17-256	TB/ROW	64	32
6	0-32	0-16	PWARP/ROW	512	4

In order to improve the load balance and exploit shared memory, it is important to set appropriate parameters for our proposal. We describe how to set the hash table size and thread block size for each group. Table 5.1 shows the parameter setting on Tesla P100 GPU. NVIDIA GPUs have few dozen streaming multiprocessors (SMs). Each SM implements 64 CUDA cores and on-chip shared memory. The size of shared memory per SM is 64 KB and maximum shared memory size per thread block is 48 KB on P100. Parameter setting can be statically done based on hardware information such as shared memory size and maximum thread block size.

First, we determine the group of the largest hash table that can be allocated on shared memory. Algorithm 6 requires modulus operation of hash table size, t_{size}. Since the modulus operation is expensive, we utilize lightweight bit operations by setting t_{size} to powers of two. In double precision, the hash tables need 8 bytes for each value data, and 4 bytes for each column index. When computing

output matrix by TB/ROW, each thread block requires $t_{size} \times 12$[Byte] for hash table. Therefore, the largest table size is set as $t_{size} = 48$[KB]/12[Byte] = 4096, respected to Group 1. When counting the number of non-zero elements of each row in step (3), the hash table size is set larger than that of step (7) since the table for value data is not necessary. The thread block size is set as upper limit, 1024. The Group 0 handles the rows, which require larger hash tables than the size of shared memory, and the hash tables are allocated on global memory.

After the upper limit size of hash table is set, we set the one-step smaller hash table size. The hash table size and thread block size are halved. This enables to increase the number of concurrently executing thread blocks on each SM, and improves the GPU resource usage and occupancy. When the number of thread blocks executing in each SM is larger than fixed max thread blocks / multiprocessor (= 32), the algorithm finishes making row group with TB/ROW. Finally, the PWARP/ROW is assigned to the rows with few non-zero elements. Each borderline between TB/ROW and PWARP/ROW is set as 32 for (3) and 16 for (7).

5.4 Performance Evaluation

We compare the performance of our proposal and major existing sparse libraries. Our evaluation is based on FLOPS performance of squaring matrix, which is twice the number of intermediate products divided by execution time.

We evaluate the performance of SpGEMM on NVIDIA Tesla P100 PCI-e GPU, which is Pascal generation. The GPU has 16 GBytes of device memory and its bandwidth is up to 732 GB/sec. The machine has Intel Xeon CPU E5-2650 v3. GPU codes are implemented in CUDA 8.0 and are executed on CentOS Linux release 7.2.1511.

We use three sparse matrix libraries, cuSPARSE [51], CUSP [52] and BHSPARSE [49]. The cuSPARSE employs two-phases SpGEMM algorithm. First phase counts the number of non-zero elements and second phase computes the output matrix. The CUSP adopts ESC algorithm. The SpGEMM kernel of BHSPARSE is optimized for irregular matrices, which likely cause load imbalance.

For our performance evaluations, we selected various kinds of 12 square matrices which are often used for the evaluation of sparse matrix computation on GPUs [22] from the Sparse Matrix Collection at University of Florida [24]. The name and size of each matrix data are summarized in Table 5.2. The matrix size, the number of non-zero elements and the pattern of non-zero elements are largely different by matrix. Based on the average number of non-zero elements per row (Nnz/row), top eight matrices are called as High-Throughput Matrices and the other four matrices as Low-Throughput Matrices in this paper. Since the number of intermediate products is much larger than the number of non-zero elements of output matrix, some existing SpGEMM algorithms, which increase the memory usages

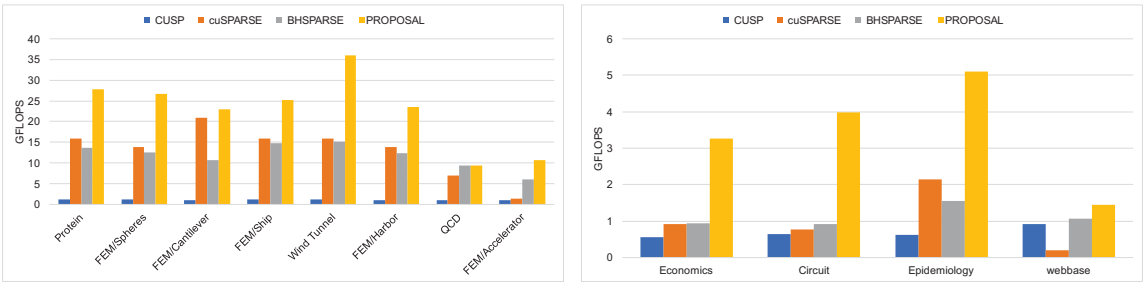
depending on the numbers of intermediate products, may require much memory usage. The dataset 'webbase' is distinctive matrix; maximum number of non-zero elements per row (Max nnz/row) is much larger than Nnz/row, that is, only some rows have many non-zero elements and most rows have very few non-zero elements. We additionally selected large size matrices, which are generated in graph analysis. These selected three matrices have highly irregularity and SpGEMM computation on GPU for these matrices often causes terrible load imbalance.

Table 5.2: Matrix data

Name	**Rows**	**Non-zero**	**Nnz/row**	**Max nnz/row**	**FLOP**(A^2)	**Nnz of** A^2
Protein	36,417	4,344,765	119.3	204	555,322,659	19,594,581
FEM/Spheres	83,334	6,010,480	72.1	81	463,845,030	26,539,736
FEM/Cantilever	62,451	4,007,383	64.2	78	269,486,473	17,440,029
FEM/Ship	140,874	7,813,404	55.5	102	450,639,288	24,086,412
Wind Tunnel	217,918	11,634,424	53.4	180	626,054,402	32,772,236
FEM/Harbor	46,835	2,374,001	50.7	145	156,480,259	7,900,917
QCD	49,152	1,916,928	39.0	39	74,760,192	10,911,744
FEM/Accelerator	121,192	2,624,331	21.7	81	79,883,385	18,705,069
Economics	206,500	1,273,389	6.2	44	7,556,897	6,704,899
Circuit	170,998	958,936	5.6	353	8,676,313	5,222,525
Epidemiology	525,825	2,100,225	4.0	4	8,391,680	5,245,952
webbase	1,000,005	3,105,536	3.1	4700	69,524,195	51,111,996
cage15	5,154,859	99,199,551	19.2	47	2,078,631,615	929,023,247
wb-edu	9,845,725	57,156,537	5.8	3841	1,559,579,990	630,077,764
cit-Patents	3,774,768	16,518,948	4.4	770	82,152,992	68,848,721

5.4.1 Performance of SpGEMM

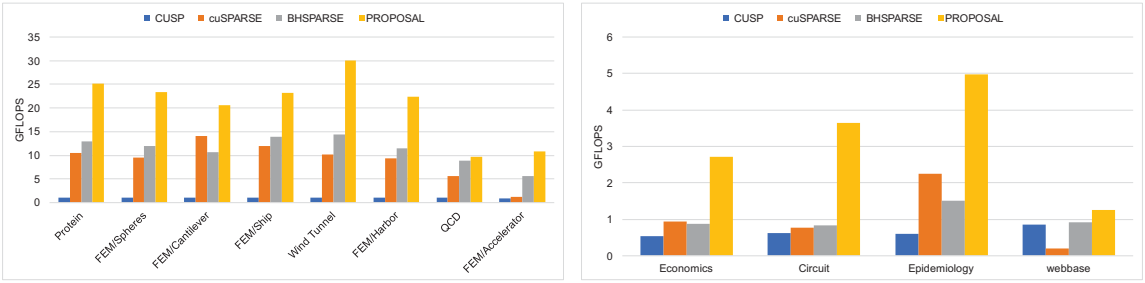
Figure 5.2 shows the performance of SpGEMM computation in single precision. The CUSP achieves constant performance for all matrices. The performance is independent to the matrix size and the pattern of non-zero elements. The BHSPARSE shows superior performance compared to cuSPARSE for low Nnz/row matrices. On the other hand, cuSPARSE achieves higher performance for denser or regular matrices. Our proposal shows best performance in the evaluation for all evaluated matrices. The appropriate thread assignments and the construction of hash table with utilizing shared memory make large performance gains for both High- and Low-Throughput Matrices. Our proposal achieves significant speedups of 32.3x, 8.1x and 4.3x on maximum compared to CUSP, cuSPARSE and BHSPARSE respectively, and 15.7x, 3.2x and 2.3x on average respectively. Performance evaluation shows the speedup is up to 4.3x compared to best of existing sparse matrix libraries.



(a) High-Throughput Matrices (b) Low-Throughput Matrices

Figure 5.2: Performance of SpGEMM computation in single precision

Figure 5.3 shows the performance of SpGEMM computation in double precision. Performance trend is similar to single precision, and our approach also shows good speedups compared to existing sparse matrix libraries. Our proposal achieves speedups of 28.7x, 8.7x and 4.4x on maximum compared to CUSP, cuSPARSE and BHSPARSE respectively, and 15.1x, 3.3x and 2.2x on average respectively. Performance evaluation shows the speedup is up to 4.4x compared to best of existing sparse matrix libraries.



(a) High-Throughput Matrices (b) Low-Throughput Matrices

Figure 5.3: Performance of SpGEMM computation in double precision

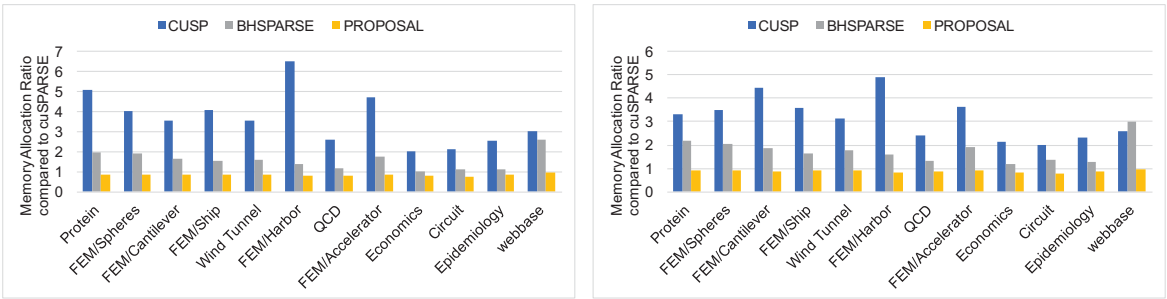
Table 5.3 shows the performance of SpGEMM computation for large size matrix data. The CUSP and BHSPARSE cannot execute SpGEMM of 'cage15' and 'wb-edu' because of large amounts of memory use for temporary results. On the other hand, the cuSPARSE executes SpGEMM of these matrices. However, the performance of cuSPARSE for such irregular matrices is really poor. Our proposal executes SpGEMM with low memory usage and achieves significant speedups up to 11.6x.

Table 5.3: Performance of SpGEMM computation for large size graph data [GFLOPS]

	Matrix	CUSP	cuSPARSE	BHSPARSE	PROPOSAL	Speedup
single	cage15	-	0.519	-	5.955	x11.5
	wb-edu	-	2.348	-	5.403	x2.3
	cit-Patents	0.837	0.028	0.880	3.351	x3.8
double	cage15	-	0.491	-	5.684	x11.6
	wb-edu	-	2.145	-	4.618	x2.2
	cit-Patents	0.780	0.028	0.813	2.980	x3.7

5.4.2 Memory Usage

Our proposal aims to both accelerate SpGEMM computation and reduce memory usage on GPU. We evaluate the maximum memory usage during SpGEMM computation with existing SpGEMM libraries and our proposal. Figure 5.4 shows the ratio of maximum memory usage to cuSPARSE in single and double precision. Memory usage of our approach is much lower than other existing libraries for any matrix data in any precisions. The memory usages are reduced by 14.7% in single precision and 10.9% in double precision on average. This result indicates that our proposal accelerates SpGEMM computation of large matrix data on GPUs' limited device memory. Although BHSPARSE achieves superior performance to cuSPARSE for irregular matrices, BHSPARSE requires much larger memory. In contrast, our proposal shows higher performance compared to BHSPARSE for irregular matrices and also reduces memory usage by 67.7% on maximum.



(a) Single precision

(b) Double precision

Figure 5.4: Maximum memory usage in SpGEMM computation

5.4.3 Performance Analysis

We show the detailed performance analysis of our SpGEMM algorithm on P100.

First, we describe the effectiveness of CUDA stream. Our proposal launches multiple CUDA kernels with CUDA streams for each group to execute concurrently. Since some groups handle few rows less than 10, the CUDA stream enables to utilize GPU resource. For the matrix 'Cuircuit', one of the groups handles only 8 rows for counting and 9 rows for calculation. We confirmed that our proposal with CUDA stream achieves 1.3x speedups compared to the proposal without CUDA stream.

Next, we show the effectiveness of PWARP/ROW. The PWARP/ROW efficiently processes the rows with few non-zero elements. Especially, a matrix with low Nnz/row has many of rows with few non-zero elements. For the matrix 'Epidemiology', whose Nnz/row is 4.0, the PWARP/ROW significantly improves the performance, and we confirmed that the speedup is 3.1x compared to the proposal without PWARP/ROW.

Finally, we show the execution time breakdown of cuSPARSE and our proposal. Figure 5.5 and 5.6 shows the ratio of execution time of each phase, where the total execution time of cuSPARSE is set as 1. The result is divided into 4 phases; setup phase for grouping in our proposal, count phase, calculation phase and cudaMalloc time of output matrix. For High-Throughput Matrix, our proposal largely reduces the 'calculation' time from cuSPARSE though the cost of 'count' is roughly same in both cuSPARSE and our proposal. The 'setup' phase, which is an overhead in our proposal, is negligible for almost matrices. The cost of cudaMalloc on Pascal GPU becomes larger compared to previous generation GPUs, and the cudaMalloc phase becomes considerable for sparser matrices. The GPU memory is also allocated even in setup phase. For the matrix 'Epidemiology', which is highly sparse and regular, the execution time of 'count' and 'calculation' are largely reduced from cuSPARSE. However, the large cost of cudaMalloc in setup phase and of cudaMalloc for output matrix prevents the performance improvement.

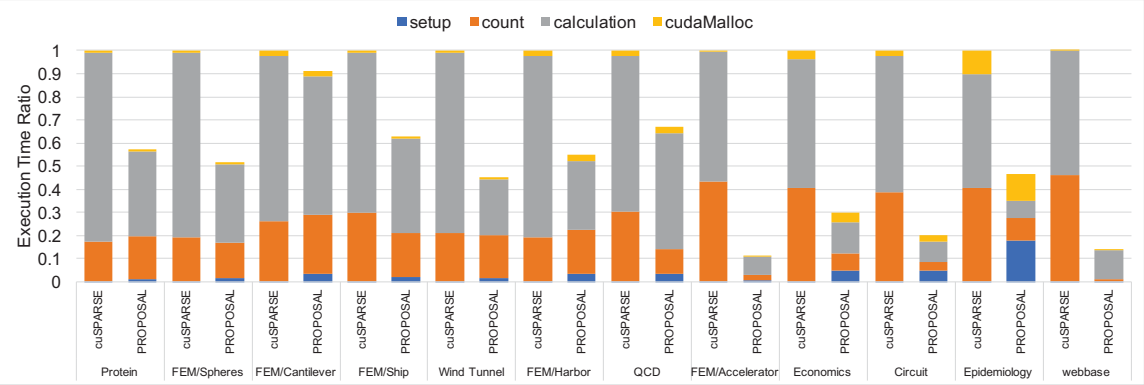


Figure 5.5: Performance breakdown comparing with cuSPARSE in single precision

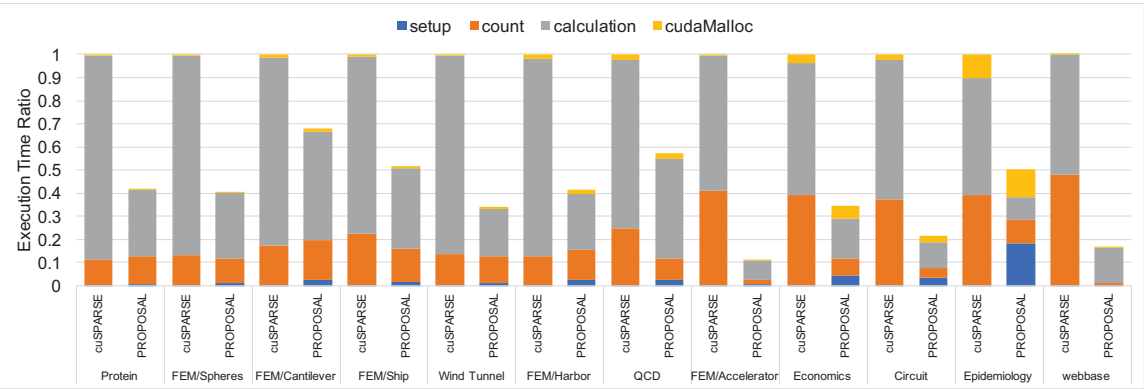
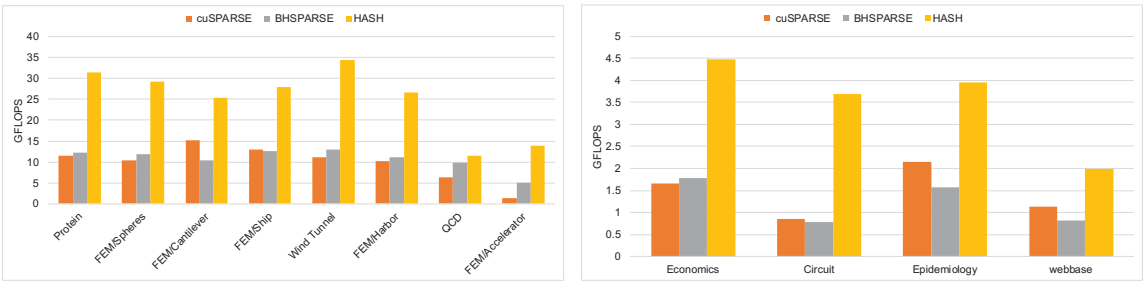


Figure 5.6: Performance breakdown comparing with cuSPARSE in double precision

5.4.4 Practical Benchmark Results

Our approach is not specialized for A^2 SpGEMM operation, generalized for $A * B$. Here, we show the practical performance evaluation results, not limited to A^2.

Figure 5.7 shows the results of $A^T * A$ in double precision. This form of SpGEMM is used in graph analysis to find connected components. The performance trend of $A^T * A$ is similar to A^2, and our proposal achieves best performance. Our approach shows the speedups up to 4.3x and 4.7x for cuSPARSE and BHSPARSE, respectively.



(a) High-Throughput Matrices

(b) Low-Throughput Matrices

Figure 5.7: Performance of SpGEMM between A^T and A in double precision

In numerical applications, the AMG method is used for preconditioner of iterative method in solving equations. For the setup phase in AMG method, the Galerkin product, which is the product of three sparse matrices expressed as $R * A * P$, is executed. Figure 5.8 shows the execution time of SpGEMM operations in AMG method. We used sample program, which solves a 5-point Poisson equation discretized on 2048 by 2048 grid, in CUSP library [52]. To make coarse problem matrix, four times of $R * A * P$ are executed, and we compare the SpGEMM performance of our approach to cuSPARSE and BHSPARSE. The evaluation result shows our HASH algorithm largely reduces the execution time of SpGEMM, especially for larger matrix, and achieves 3.0x performance improvements compared to cuSPARSE.

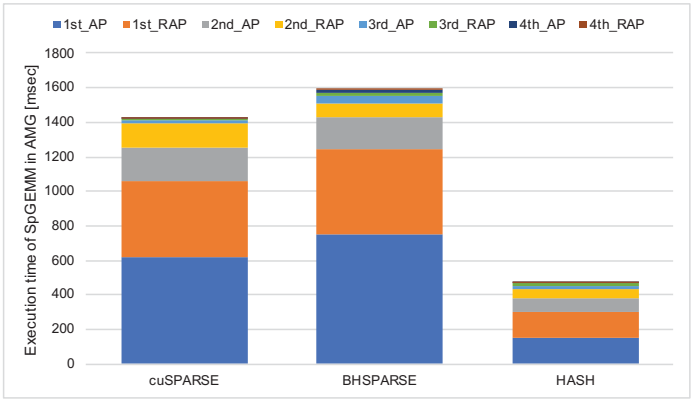


Figure 5.8: Performance of SpGEMM between three sparse matrices following $R * A * P$ in AMG method

5.5 Related work

There has been several past work of SpGEMM computation on GPUs.

Demouth proposed two-phases SpGEMM algorithm implemented in cuSPARSE library [53]. The SpGEMM kernel of cuSPARSE allocates hash table on shared memory and global memory. If the insertion to the hash table on shared memory does not success, the algorithm tries for global memory. This algorithm causes many random global memory access and do not efficiently utilize fast shared memory. On the other hand, our proposal appropriately divides the rows into groups, and the execution of most of rows can be finished on shared memory.

Gremse et al. proposed Iterative Row Merge method, which efficiently combines intermediate products among warp [54]. Part of threads in warp is assigned to each row of input matrix A, and each thread generates the table of non-zero elements in the row of input matrix B corresponding to the non-zero element of A. By merging tables among threads, the values and column indices of non-zero elements in output matrix are computed. The merge operation itself is fast since it's executed among warp. However, the operation can be applicable for the rows which have few non-zero elements. The Iterative Row Merge method often requires decomposition of given matrix in order to execute, and this decomposition causes additional cost of memory usage and memory access.

Liu et al. proposed the SpGEMM algorithm for irregular matrix data like graph data on GPUs [49], and this algorithm is implemented in BHSPARSE library. The execution time of SpGEMM in each row largely varies, and depends on the number of intermediate products or non-zero elements. In order to improve the load balance of combining intermediate products, this algorithm assigns the rows to multiple bins based on the number of intermediate products, and selects appropriate algorithms for each bin. They use heap method, bionic ESC method and mergepath method for combining intermediate products. They evaluate the performance of GPUs and CPU, and show the performance advantage of GPUs. The aims of grouping rows of our proposal are the improvement of parallelism and exploiting GPU resource by appropriate thread assignments and the coordination of hash table size.

Anh et al. proposed Balanced Hash, which improves both the load balance and the process of combining intermediate products [55]. They improve the load balance by generating the worklist, which is the list of pairs of indices of non-zero elements required in the computation of intermediate products. Multiple rows are bundled based on the number of intermediate products, and are assigned one thread block to. Balanced Hash utilizes hash table to combine intermediate products. The hash table is on shared memory and the size is fixed. When the hash collisions occur, the intermediate products are stored into queue to execute later. After the calculation finishes once, the non-zero elements in hash table are stored to global memory. Then, the table is refreshed and the calculation is done again if there are still any intermediate products in the queue. The process continues until

the queue becomes empty. The Balanced Hash algorithm accelerates SpGEMM computation since most of process can be executed on shared memory. On the other hand, the worklist requires much memory usage since it is stored in memory once. The queue for hash collision also requires additional memory usage and memory access. When the number of non-zero elements is small, the Balanced Hash algorithm uses more shared memory than necessary and cannot exploit the GPU resource. By constructing hash table with different size for each group, our algorithm assigns more thread blocks to each SM and fully utilizes GPU resource. Furthermore, since our algorithm does not require memory access when the hash collision occurs, the execution of most of rows can finish on shared memory and our algorithm achieves high SpGEMM performance.

5.6 Conclusion

Accelerating SpGEMM computation, which is a bottleneck of the performance in the preconditioner of iterative method such as AMG method and graph algorithms, is one of the critical issues for scientific and engineering applications. Although SpGEMM is accelerated on many-core processors such as GPUs, which are widely installed on supercomputers, the memory access and utilization of shared memory are not well optimized. Furthermore, since existing work requires much memory usage to accelerate SpGEMM, the applicable matrix is limited on GPUs with small device memory. We propose the state of the art SpGEMM algorithm in order to both accelerate SpGEMM and reduce memory usage by appropriate grouping and efficient utilization of shared memory. Our proposal achieves superior performance to competing sparse matrix libraries on NVIDIA Pascal generation GPU, demonstrating up to 4.3x speedup in single precision and 4.4x speedup in double precision over CUSP, cuSPARSE and BHSPARSE libraries. Our proposal successfully reduces memory usage from existing sparse matrix libraries. Maximum memory usage is reduced by 14.7% in single precision and 10.9% in double precision on average.

Chapter 6

Optimization and Performance Analysis of SpGEMM on Intel Architectures

6.1 Introduction

Multiplication of two sparse matrices (SpGEMM) is a recurrent kernel in many algorithms in machine learning, data analysis, and graph analysis. For example, bulk of the computation in multi-source breadth-first search [56], betweenness centrality [57], Markov clustering [58], label propagation [59], peer pressure clustering [60], clustering coefficients [61], high-dimensional similarity search [62], and topological similarity search [63] can be expressed as SpGEMM. Similarly, numerical applications such as scientific simulations also use SpGEMM as a subroutine. Typical examples include the Algebraic Multigrid (AMG) method for solving sparse system of linear equations [64], volumetric mesh processing [65], and linear-scaling electronic structure calculations [66].

The extensive use of SpGEMM in data-intensive applications has led to the development of several sequential and parallel algorithms. Most of these algorithms are based on Gustavson's row-wise SpGEMM algorithm [67] where a row of the output matrix is constructed by accumulating a subset of rows of the second input matrix (see Algorithm 7 for details). It is the accumulation (also called merging) technique that often distinguishes major classes of SpGEMM algorithms from one another. Popular data structures for accumulating rows or columns of the output matrix include heap [68], hash [69], and sparse accumulator (SPA) [70].

Recently, researchers have developed parallel heap-, hash-, and SPA-based SpGEMM algorithms

for shared-memory platforms [46, 50, 54, 55, 71]. These algorithms are also packaged in publicly-available software that can tremendously benefit many scientific applications. However, when using an SpGEMM algorithm and implementation for a scientific problem, one needs answer to the following questions: (a) what is the best algorithm/implementation for a *problem* at hand? (b) what is the best algorithm/implementation for the *architecture* to be used in solving the problem? These practically important questions remain mostly unanswered for many scientific applications running on highly-threaded architectures. This paper answers both questions in the context of existing SpGEMM algorithms. That means our focus is not to develop new parallel algorithms, but to characterize, optimize and evaluate existing algorithms for real-world applications on modern multi-core and many-core architectures.

Firstly, previous algorithmic work did not pay close attention to architecture-specific optimizations that have big impacts on the performance of SpGEMM. We fill this gap by characterizing the performance of SpGEMM on shared-memory platforms and identifying bottlenecks in memory allocation and deallocation as well as overheads in thread scheduling. We propose solutions to mitigate those bottlenecks. Using microbenchmarks that model SpGEMM access patterns, we also uncover reasons behind the non-uniform performance boost provided by the MCDRAM on KNL. These optimizations result in efficient heap-based and hash-table-based SpGEMM algorithms that outperform state-of-the-art SpGEMM libraries including Intel MKL and Kokkos-Kernels [72] for many practical problems.

Secondly, previous work has narrowly focused on one or two real world application scenarios such as squaring a matrix and studying SpGEMM in the context of AMG solvers [72, 73]. Different from the literature, our evaluation also includes use cases that are representative of real graph algorithms, such as the multiplication of a square matrix with a tall skinny one that represents multi-source breadth-first search and the multiplication of triangular matrices that is used in triangle counting. While in the majority of the cases the hash-table-based SpGEMM algorithm is dominant, we also find that different algorithms dominate depending on matrix size, sparsity, compression factor, and operation type. This in-depth analysis exposes many interesting features of algorithms, applications, and multithreaded platforms.

Thirdly, while many SpGEMM algorithms keep nonzeros sorted within each row (or column) in increasing column (or row) identifiers, this is not universally necessary for subsequent sparse matrix operations. For example, CSparse [74, 75] assumes none of the input matrices are sorted. Clearly, if an algorithm accepts its inputs only in sorted format, then it must also emit sorted output for fairness. This is the case with the heap-based algorithms. However, hash-table-based algorithm do not need their inputs sorted. In this case, we see a significant performance benefit due to skipping the sorting of the output as well.

Fourthly, based on these architecture- and application-centric optimizations and evaluations, we

make a recipe for selecting the best-performing algorithm for a specific application scenario. We also build a performance model for hash-table and heap-based algorithms, and show detailed profiling result of them. These performance models and profiling results support the correctness of the recipe both theoretically and empirically. With this recipe, we switch the SpGEMM algorithm inside a large-scale protein clustering application named HipMCL, and show the positive performance impact of selecting the appropriate algorithm in real applications.

This paper brings various SpGEMM algorithms and libraries together, analyzes them based on algorithm, application, and architecture related features and provides exhaustive guidelines for SpGEMM-dependent applications.

6.2 Background and Related Work

Let A, B be input matrices, and SpGEMM computes a matrix C such that $C = AB$. When analyzing algorithms in this paper, we assume n-by-n matrices for simplicity. The input and output matrices are sparse and they are stored in a sparse format. The number of nonzeros in matrix A is denoted with $nnz(A)$. Figure 7 shows the skeleton of the most commonly implemented SpGEMM algorithm, which is due to Gustavson [67]. When the matrices are stored using the Compressed Sparse Rows (CSR) format, this SpGEMM algorithm proceeds row-by-row on matrix A (and hence on the output matrix C). Let a_{ij} be the element in i-th row and j-th column of matrix A and a_{i*} be the i-th row of matrix A. The row of matrix B corresponding to each non-zero element of matrix A is read, and each non-zero element of output matrix C is calculated.

Algorithm 7 Gustavson's Row-wise SpGEMM [1]

Input: Sparse matrices A and B

Output: Sparse matrix C

```
1: set matrix C to ∅
2: for all a_i* in matrix A in parallel do
3:   for all a_ik in row a_i* do
4:     for all b_kj in row b_k* do
5:       value ← a_ik b_kj
6:       if c_ij ∉ c_i* then
7:         insert (c_ij ← value) to c_i*
8:       else
9:         c_ij ← c_ij + value
10:      end if
11:    end for
12:  end for
13: end for
```

SpGEMM computation has two critical issues unlike dense matrix multiplication. Firstly, the pattern and the number of non-zero elements of output matrix are not known beforehand. For this reason, the memory allocation of output matrix becomes hard, and we need to select from two strategies. One is a two-phase method, which counts the number of non-zero elements of output matrix first (symbolic phase), and then allocates memory and computes output matrix (numeric phase). The other is a one-phase method, where we allocate large enough memory space for output matrix and compute. The former requires more computation cost, and the latter uses much more memory space. Second issue is about combining the intermediate products (*value* in Fig. 7) to non-zero elements of output matrix. Since the output matrix is also sparse, it is hard to efficiently accumulate intermediate products into non-zero elements. This procedure is a performance bottleneck of SpGEMM computation, and it is important to devise and select better accumulator for SpGEMM.

Since each row of C can be constructed independently of each other, Gustavson's algorithm is conceptually highly parallel. For accumulation, Gustavson's algorithm uses a dense vector and a list of indices that hold the nonzero entries in the current active row. This particular set of data structures used in accumulation are later formalized by Gilbert et al. under the name of sparse accumulator (SPA) [70]. Consequently, a naive parallelization of Gustavson's algorithm requires temporary storage of $O(n\,t)$ where t is the number of threads. For matrices with large dimensions, a SPA-based algorithm

[1] The **in parallel** keyword does not exist in the original algorithm but is used here to illustrate the common parallelization pattern of this algorithm used by all known implementations.

can still achieve good performance by "blocking" SPA in order to decrease cache miss rates. Patwary et al. [73] achieved this by partitioning the data structure of B by columns.

Sulatycke and Ghose [71] presented the first shared-memory parallel algorithm for the SpGEMM problem, to the best of our knowledge. Their parallel algorithm, dubbed *IKJ method* due to the order of the loops, has a double-nested loop over the rows and the columns of the matrix A. Therefore, the IKJ method has work complexity $O(n^2 + \text{flop})$ where flop is the number of the non-trivial scalar multiplications (i.e. those multiplications where both operands are nonzero) required to compute the product. Consequently, the IKJ method is only competitive when flop $\geq n^2$, which is rare for SpGEMM. Several GPU algorithms that are also based on the row-by-row formulation are presented [49, 69]. These algorithms bin the rows based on their density due to the peculiarities of the GPU architectures. Then, a poly-algorithm executes a different specialized kernel for each bin, depending on its density. Two recent algorithms that are implemented in both GPUs and CPUs also follow the same row-by-row pattern, only differing on how they perform the merging operation. ViennaCL [76] implementation, which was first described for GPUs [54], iteratively merges sorted lists, similar to merge sort. KokkosKernels implementation [72], which we also include in our evaluation, uses a multi-level hash map data structure.

The CSR format is composed of three arrays: row pointers array (*rpts*) of length $n + 1$, column indices (*cols*) of length *nnz*, and values (*vals*) of length *nnz*. Array *rpts* indexes the beginning and end locations of nonzeros within each row such that the range $cols[rpts[i] \dots rpts[i + 1] - 1]$ lists the column indices of row i. The CSR format does not specify whether this range should be sorted with increasing column indices; that decision has been left to the library implementation. As we will show in our results, there are significant performance benefits of operating on unsorted CSR format. Table 6.1 lists high-level properties of the codes we study in this paper. Heap and Hash are based on our prior work [68, 69]. Since MKL code is proprietary, we do not know the accumulator.

Table 6.1: Summary of SpGEMM codes studied in this paper

Algorithm	**Phases**	**Accumulator**	**Sortedness (Input/Output)**
MKL	2	-	Any/Select
MKL-inspector	1	-	Any/Unsorted
KokkosKernels	2	HashMap	Any/Unsorted
Heap	1	Heap	Sorted/Sorted
Hash/HashVector	2	Hash Table	Any/Select

6.3 Microbenchmarks on Intel KNL

Our experiments mainly target Intel Xeon Phi architecture. We conducted some preliminary experiments to tune optimization parameters to expose the performance bottlenecks. These microbenchmarks are especially valuable for designing an algorithm of SpGEMM for these architectures. Details of evaluation environment are summarized in Table 6.3.

6.3.1 Scheduling Cost of OpenMP

When parallelizing a loop, OpenMP provides three thread scheduling choices: *static*, *dynamic* and *guided*. Static scheduling divides loop iterations equally among threads, dynamic scheduling allocates iterations to threads dynamically, and guided scheduling starts with static scheduling with smaller iterations and switches to dynamic scheduling later. Here, we experimentally evaluate the cost of these three scheduling options on KNL processors by running a simple program, which performs iterations of an empty loop. We measure the time during loop iterations and the result is shown in Figure 6.1. In the load balanced case with a large number of iterations, static scheduling has little scheduling overhead compared to dynamic scheduling on KNL, as expected. The guided scheduling is also as expensive as dynamic scheduling. Based on these evaluations, we opt to use static scheduling in our SpGEMM algorithms since the scheduling overhead of dynamic or guided scheduling becomes a bottleneck.

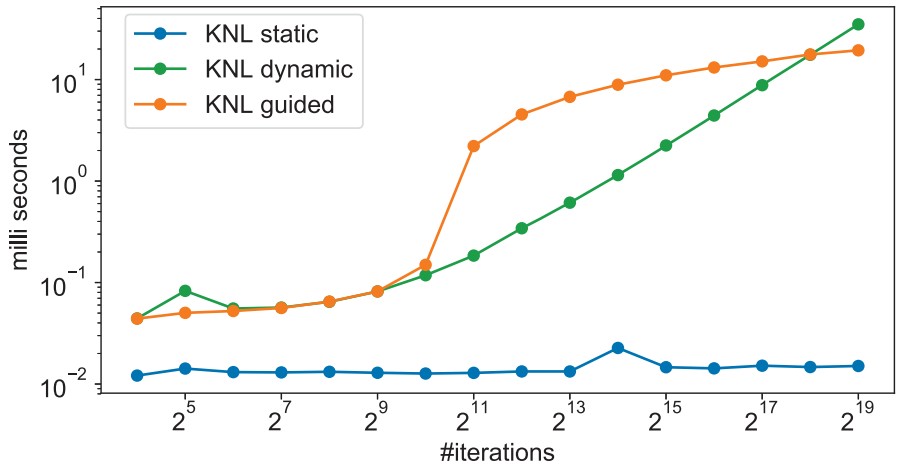


Figure 6.1: OpenMP Scheduling Cost on KNL

6.3.2 Memory Allocation/Deallocation

To find a suitable memory allocation/deallocation scheme on KNL, we performed a simple experiment: allocate a memory space, access elements on the allocated memory and then deallocate it. In section, we consider two aspects of memory allocation. One is parallelism. Either each thread independently allocates its memory space (called *each* in this paper), or one master thread allocates space for all threads (called *master* in this paper).

The other aspect is the allocator used. We examined three allocators: new/delete of C++, aligned allocation (_mm_malloc/_free), and scalable_malloc/_free provided by Intel TBB (Thread Building Block). Figure 6.2 shows the results of deallocation cost with 256 threads on KNL. Since aligned allocation showed nearly the same performance as C++, we show only the results of C++ and TBB. All allocators by *master* have extremely high cost for large memory: over 100 milliseconds for the deallocation of 1GB memory space. The *each* deallocation for large memory chunks is much cheaper than *master* deallocation. The cost of *each* deallocation suddenly rises at 8GB (C++) or 64GB (TBB), where each thread allocates 32MB or 256MB, respectively. These thresholds match those of *master* deallocation. On the other hand, the cost of *each* deallocation for small memory space becomes larger than that of *master* deallocation since *each* deallocation causes the overheads of OpenMP scheduling and synchronization. From this result, we compute the amount of memory required by each thread and allocate this amount of thread-private memory in each thread independently in order to reduce deallocation cost in SpGEMM computation, which requires temporally memory allocation and deallocation. In the following experiments in this paper, the TBB is used for both *master* and *each* memory allocation/deallocation to simply have performance gain.

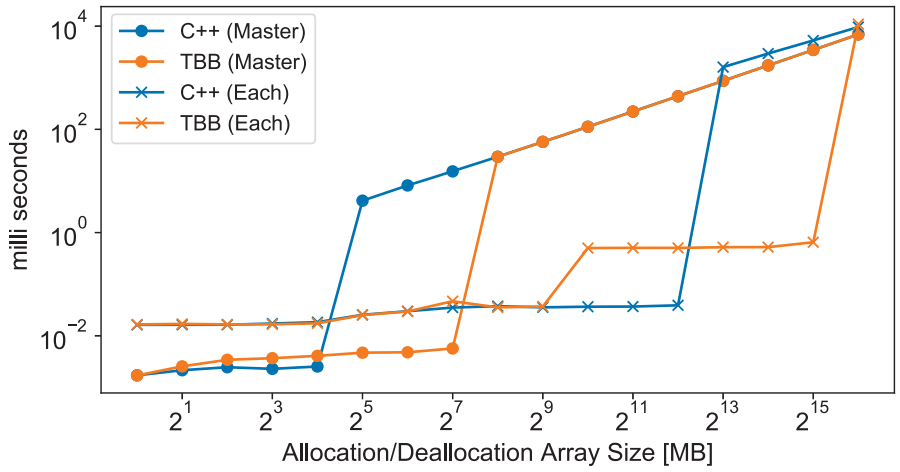


Figure 6.2: Cost of deallocation on KNL

6.3.3 DDR vs. MCDRAM

Intel KNL implements MCDRAM, which can accelerate bandwidth-bound algorithms and applications. While MCDRAM provides high bandwidth, its memory latency is larger than that of DDR4. In row-wise SpGEMM (Algorithm 7), there are three main types of data accesses for the formation of each row of C. First is a unit-stride streaming access pattern for indices and value data of A and the sparse output vector c_{i*}. Second is a stanza-like memory access to rows of B. Small blocks (stanzas) of consecutive elements are randomly accessed. Finally, updates to the accumulator exhibit different access pattern depending on the type of the accumulator (a hash-table, SPA, or heap). The streaming access to the input vector is usually the cheapest of the three and the accumulator access depends on the data structure used. Hence, stanza access pattern is the most canonical of the three and provides a decent proxy to study.

To evaluate the stanza bandwidth, we used a custom microbenchmark. Figure 6.3 shows a comparison result between DDR only and use of MCDRAM as Cache with scaling the length of contiguous memory access. When the contiguous memory access is wider, both DDR only and MCDRAM as Cache achieve their peak bandwidth, and especially MCDRAM as Cache shows over 3.4x superior bandwidth compared to DDR only. However, the use of MCDRAM as Cache is incompatible with fine-grained memory access. When the stanza length is small, there is little benefit of using MCDRAM. This benchmark hints that it would be hard to get the benefits of MCDRAM on very sparse matrices.

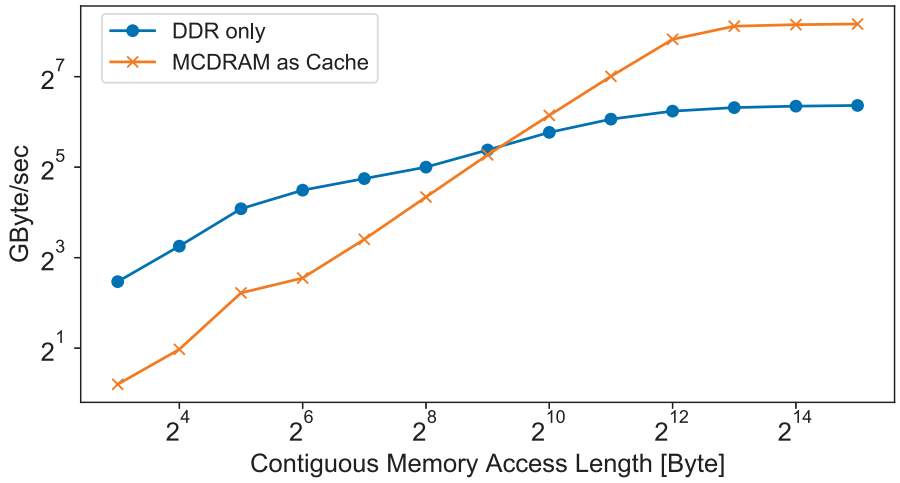


Figure 6.3: Benchmark result of random memory access with DDR only or MCDRAM as Cache

6.4 Architecture Specific Optimization of SpGEMM

Based on our findings of performance critical bottlenecks on Intel KNL, we design SpGEMM algorithms taking account in architecture specific issues. Firstly, we show light-weight thread scheduling scheme with load-balancing for SpGEMM. Next, we show the optimization schemes for hash-table-based SpGEMM, which is proposed for GPU [69], and heap-based shared-memory SpGEMM algorithms [68]. Additionally, we extend the Hash SpGEMM with utilizing vector registers of Intel Xeon Phi. Finally, we show which accumulator works well for target scenario from the theoretical point of view by estimating each accumulation cost.

6.4.1 Light-weight Load-balancing Thread Scheduling Scheme

To achieve good load-balance with static scheduling, the bundle of rows should be assigned to threads with equal computation complexity before symbolic or numeric phase. Algorithm 8 shows how to assign rows to threads. We count flop of each row, then do prefix sum. Each thread can find the start point of rows by binary search. LOWBND($vec, value$) in line 16 finds the minimum id such that $vec[id]$ is larger than or equal to $value$. Each of these operations can be executed in parallel.

Algorithm 8 RowsToThreads

Input: Sparse matrices A and B

Output: Array $offset$

```
1: {1. Set FLOP vector}
2: for i ← 0 to m in parallel do
3:    flop[i] ← 0
4:    for j ← rpts_A[i] to rpts_A[i + 1] do
5:       rnz ← rpts_B[cols_A[j] + 1] − rpt_B[cols_A[j]]
6:       flop[i] ← flop[i] + rnz
7:    end for
8: end for
9: {2. Assign rows to thread}
10: flop_ps ← ParallelPrefixSum(flop)
11: sum_flop ← flop_ps[m]
12: tnum ← omp_get_max_threads()
13: ave_flop ← sum_flop / tnum
14: offset[0] ← 0
15: for tid ← 1 to tnum in parallel do
16:    offset[tid] ← lowbnd(flop_ps, ave_flop ∗ tid)
17: end for
18: offset[tnum] ← m
```

6.4.2 Symbolic and Numeric Phases

We optimized two approaches of accumulation for KNL. One is hash-table-based algorithm and the other is heap-based algorithm. Furthermore, we add another version of Hash SpGEMM, where hash probing is vectorized with AVX2 or AVX-512 instructions.

Hash SpGEMM

We use hash-table for accumulator in SpGEMM computation, based on GPU work [69]. Algorithm 9 shows the algorithm of Hash SpGEMM for multi- and many-core processors. We count a flop per row of output matrix. The upper limit of any thread's local hash-table size is the maximum number of flop per row within the rows assigned to the thread. Each thread once allocates the hash-table based on its own upper limit and reuses that hash-table throughout the computation by reinitializing for each row. Next is about hashing algorithm we adopted. A column index is inserted into hash-table as key. Since the column index is no less than 0, the hash-table is initialized by storing -1. The column index is

multiplied by constant number and divided by hash-table size to compute the remainder. In order to compute modulus operation efficiently, the hash-table size is set as 2^n (n is an integer). The hashing algorithm is based on linear probing. Figure 6.4-(a) shows an example of hash probing on 16 entries hash-table.

Algorithm 9 Hash SpGEMM

Input: Sparse matrices A and B

Output: Sparse matrix C

1: $offset \leftarrow$ RowsToThreads(A, B)
2: {Determine hash-table size for each thread}
3: $tnum \leftarrow$ omp_get_max_threads()
4: **for** $tid \leftarrow 0$ to $tnum$ **in parallel do**
5: $size_t \leftarrow 0$
6: **for** $i \leftarrow offset[tid]$ to $offset[tid + 1]$ **do**
7: $size_t \leftarrow$ max$(size_t, \text{flop}[i])$
8: **end for**
9: {Required maximum hash-table size is N_{col}}
10: $size_t \leftarrow$ min$(N_{col}, size_t)$
11: {Return minimum 2^n so that $2^n > size_t$}
12: $size_t \leftarrow$ lowest_p2$(size_t)$
13: **end for**
14: Symbolic$(rpts_C, A, B)$
15: Numeric(C, A, B)

In symbolic phase, it is enough to insert keys to the hash-table. In numeric phase, however, we need to store the resulting value data. Once the computation on the hash-table finishes, the results are sorted by column indices in ascending order (if necessary), and stored to memory as output. The Hash SpGEMM for multi/many-core processors computes each row of output matrix by single thread. On the other hand, multiple threads are assigned to a row of output matrix in the GPU version of Hash SpGEMM in order to exploit massive number of threads on GPU. Due to this design for GPU version, the Hash SpGEMM on GPU requires some form of mutual exclusion since multiple threads access the same entry of the hash-table concurrently. We were able to remove this overhead in our present Hash SpGEMM for multi/many-core processors.

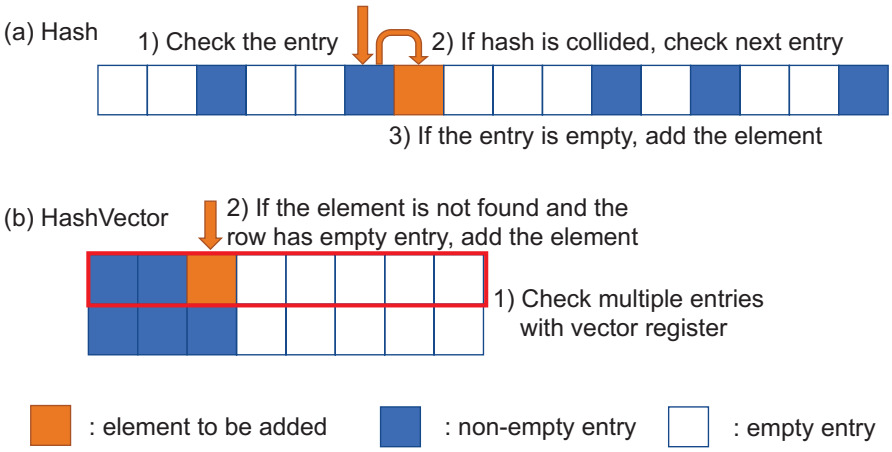


Figure 6.4: Hash Probing in Hash and HashVector SpGEMM

HashVector SpGEMM

Intel Xeon or Xeon Phi implements 256 and 512-bit wide vector register, respectively. This vector register reduces instruction counts and brings large benefit to algorithms and applications, which require contiguous memory access. However, sparse matrix computation has indirect memory access, and hence it is hard to utilize vector registers. In this paper, we utilize vector register for hash probing in our Hash SpGEMM algorithm. The vectorization of hash probing is based on Ross [77]. Figure 6.4-(b) shows how HashVector algorithm works hash probing. The size of hash-table is 16, same as (a), and it is divided into chunks based on vector width (=256-bit in this case). A chunk consists of 8 entries when a key (= column index) is represented as 32-bit. In HashVector, the hash indicates the identifier of target chunk in hash-table. In order to examine the keys in the chunk (Figure-6.4-(b)-(1)), we use comparison instruction with vector register. If the entry with target key is found, the algorithm finishes the probing for the element in symbolic phase. In numeric phase, the target entry in chunk is identified by __builtin_ctz function, which counts trailing zeros, and the multiplied value is added to the value of the entry. If the algorithm finds no entry with the key, the element is pushed to the hash-table as Figure 6.4-(b)-(2) shows. In HashVector, new element is pushed into the table in order from the beginning. The entries in chunk are compared with the initial value of hash-table, -1, by using vector register. The beginning of empty entries can be found by counting the number of bit flags of comparison result. When the chunk is occupied with other keys, the next chunk is to be checked in accordance with linear probing. Since HashVector SpGEMM can reduce the number of probing caused by hash collisions, it can achieve better performance compared to Hash SpGEMM. However, HashVector requires a few more instructions for each check. Thus, HashVector may degrade the performance when the collisions in Hash SpGEMM are rare.

Heap SpGEMM

In another variant of SpGEMM [68], we use a priority queue (heap) – indexed by column indices – to accumulate each row of C. To construct c_{i*}, a heap of size $nnz(a_{i*})$ is allocated. For every nonzero a_{ik}, the first nonzero entry in b_{k*} along with its column index is inserted into the heap. The algorithm iteratively extracts an entry with the minimum column index from the heap, accumulates it to c_{i*}, and inserts the next nonzero entry from the last extracted row of B into the heap. When the heap becomes empty, the algorithm moves to construct the next row of C.

Heap SpGEMM can be more expensive than hash- and SPA- based algorithms because it requires logarithmic time to extract elements from the heap. However, from the accumulator point of view, Heap SpGEMM is space efficient as it uses $O(nnz(a_{i*}))$ memory to accumulate c_{i*} instead of $O(\text{flop}(c_{i*}))$ and $O(n)$ memory used by hash- and SPA-based algorithms, respectively.

Our implementation of Heap SpGEMM adopts the one-phase method, which requires larger memory usage for temporally keeping the output. In parallel Heap SpGEMM, because rows of C are independently constructed by different threads, this temporary memory use for keeping output is thread-independent and the memory allocation/deallocation is done by each thread. Thread-private heaps are also managed by each thread. As with the Hash algorithm, Heap SpGEMM estimates flop via a symbolic step and uses it to balance computational load evenly among threads. Each thread is assigned multiple rows with the equal number of floating operations.

Performance Estimation for A Recipe

To estimate how our algorithms would perform in practice, we analytically analyze their computation costs. As described, Heap SpGEMM requires logarithmic time to extract elements to the heap. The complexity of Heap SpGEMM is:

$$T_{heap} = \sum_{i=1}^{n} (\text{flop}(c_{i*}) \cdot \log nnz(a_{i*})) \tag{6.1}$$

On the other hand, Hash SpGEMM has $O(1)$ cost to explore its hash-table if there are no hash collisions. We introduce the collision factor c, which is the average number of probes needed to find or insert a key. When $c = 1$, no hash collision occurs during SpGEMM computation. In addition to this hash probing cost, Hash SpGEMM requires sorting, which takes $O(n \log n)$ time, if the application needs sorted output. The computational complexity of Hash SpGEMM is:

$$T_{hash} = \text{flop} \cdot c + \sum_{i=1}^{n} (nnz(c_{i*}) \cdot \log nnz(c_{i*})) \tag{6.2}$$

From (6.1) and (6.2), Hash SpGEMM tends to achieve superior performance to Heap SpGEMM when $nnz(c_{i*})$ or $\text{flop}(c_{i*})/nnz(c_{i*})$ is large. Denser input matrices make output matrix denser too.

Also, the multiplication of input matrices with regular non-zero patterns outputs a regular matrix, and in that case, $\text{flop}(c_{i*})/nnz(c_{i*})$ is large. Thus, we can guess that Hash becomes a better choice when the input matrices are dense or have regular structures.

6.5 Experimental Setup

6.5.1 Input Types

We use two types of matrices for the evaluation. We generate synthetic matrices using matrix generator, and take matrices from SuiteSparse Matrix Collection [24]. For the evaluation of unsorted output, the column indices of input matrices are randomly permuted. We use 26 sparse matrices used in [49,61,72]. The matrices are listed in Table 6.2.

We use R-MAT [78], the recursive matrix generator, to generate two different non-zero patterns of synthetic matrices represented as ER and G500. ER matrix represents Erdős-Rényi random graphs, and G500 represents graphs with power-law degree distributions used for Graph500 benchmark. These matrices are generated with R-MAT seed parameters; a=b=c=d=0.25 for ER matrix and a=0.57, b=c=0.19, d=0.05 for G500 matrix. A scale m matrix represents 2^m-by-2^m. The *edge factor* parameter for the generator is the average number of non-zero elements per row (or column) of the matrix. In other words, it is the ratio of nnz to n.

Table 6.2: Matrix data used in our experiments (all numbers are in millions)

Matrix	**n**	***nnz***(A)	**flop**(A^2)	***nnz***(A^2)
2cubes_sphere	0.101	1.65	27.45	8.97
cage12	0.130	2.03	34.61	15.23
cage15	5.155	99.20	2,078.63	929.02
cant	0.062	4.01	269.49	17.44
conf5_4-8x8-05	0.049	1.92	74.76	10.91
consph	0.083	6.01	463.85	26.54
cop20k_A	0.121	2.62	79.88	18.71
delaunay_n24	16.777	100.66	633.91	347.32
filter3D	0.106	2.71	85.96	20.16
hood	0.221	10.77	562.03	34.24
m133-b3	0.200	0.80	3.20	3.18
mac_econ_fwd500	0.207	1.27	7.56	6.70
majorbasis	0.160	1.75	19.18	8.24
mario002	0.390	2.10	12.83	6.45
mc2depi	0.526	2.10	8.39	5.25
mono_500Hz	0.169	5.04	204.03	41.38
offshore	0.260	4.24	71.34	23.36
patents_main	0.241	0.56	2.60	2.28
pdb1HYS	0.036	4.34	555.32	19.59
poisson3Da	0.014	0.35	11.77	2.96
pwtk	0.218	11.63	626.05	32.77
rma10	0.047	2.37	156.48	7.90
scircuit	0.171	0.96	8.68	5.22
shipsec1	0.141	7.81	450.64	24.09
wb-edu	9.846	57.16	1,559.58	630.08
webbase-1M	1.000	3.11	69.52	51.11

6.5.2 Experimental Environment

We evaluate the performance of SpGEMM on a single node of the Cori supercomputer at NERSC. Cori system consists of two partitions; one is Intel Xeon Haswell cluster (Haswell), and another is Intel KNL cluster. We use nodes from both partitions of Cori. Details are summarized in Table 6.3. Each performance number in the following part is the average of ten executions.

The Haswell and KNL processors provide hyperthreading with 2 or 4 threads for each core. We set the number of threads as 68, 136, 204 or 272 for KNL, and 32 or 64 for Haswell. For the

evaluation of Kokkos, we set 128 or 256 threads instead of 136 or 272 threads whenever the execution with Kokkos fails on non-powers of two threads. We show the result with the best thread count. We set "quadrant" cluster mode, and mainly "Cache" memory mode. To select DDR4 or MCDRAM with "Flat" memory mode, we use "numactl -p". The thread affinity is set as "KMP_AFFINITY='granularity=fine',scatter".

Table 6.3: Overview of Evaluation Environment (Cori system)

	Haswell cluster	**KNL cluster**
CPU	**Intel Xeon Processor E5-2698 v3**	**Intel Xeon Phi Processor 7250**
#Sockets	2	1
#Cores/socket	16	68
Clock	2.3GHz	1.4GHz
L1 cache	32KB/core	32KB/core
L2 cache	256KB/core	1MB/tile
L3 cache	40MB per socket	-
Memory		
DDR4	128GB	96GB
MCDRAM	-	16GB
Software		
OS	SuSE Linux Enterprise Server 12 SP3	
Compiler	Intel C++ Compiler (icpc) ver18.0.0	
Option	-g -O3 -qopenmp	

6.5.3 Preliminary Evaluation

DDR vs MCDRAM

We examine the benefit of using MCDRAM over DDR memory by squaring G500 matrices with or without using MCDRAM on KNL. Figure 6.5 shows the speedup attained with the Cache mode against the Flat mode on DDR for various matrix densities. We observe that Hash SpGEMM algorithms can be benefitted, albeit moderately, from MCDRAM when denser matrices are multiplied. This observation is consistent with the benchmark shown in Figure 6.3. The limited benefit stems from the fact that SpGEMM frequently requires indirect fine-grained memory accesses often as small as 8 bytes. On denser matrices, MCDRAM can still bring benefit from contiguous memory accesses of input matrices. By contrast, Heap SpGEMM is not benefitted from high-bandwidth MCDRAM because of its fine-grained memory accesses. The performance of Heap SpGEMM even degrades when edge factor is 64 at which point the memory requirement of Heap SpGEMM surpasses the capacity of MCDRAM.

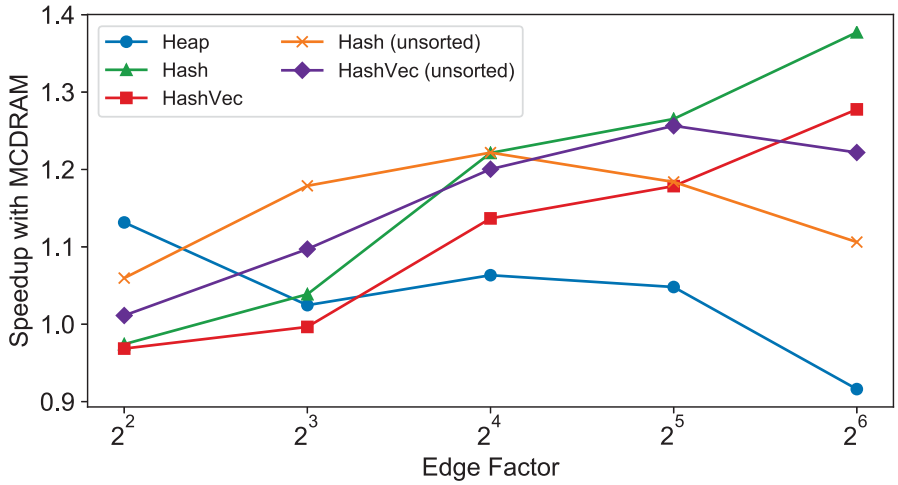


Figure 6.5: Speedups attained with the use of Cache mode on KNL compared to Flat mode on DDR4. G500 (scale 15) matrices are used with different edge factors.

Advantage of Performance Optimization on KNL for SpGEMM

We examined performance difference between OpenMP scheduling and ways to allocate memory. Figure 6.6 shows the performance of Heap SpGEMM for squaring G500 matrices with edge factor 16. When simply parallelizing SpGEMM by row, we cannot achieve higher performance because of load imbalance with static scheduling or expensive scheduling overhead with dynamic/guided scheduling. On the other hand, our light-weight load-balancing thread scheduling scheme based on the number of floating operations (flop) with static scheduling works well on SpGEMM. For larger inputs, Heap SpGEMM temporally requires large memory usage, whose deallocation causes performance degradation. Memory management scheme by each thread reduces the overhead of memory deallocation for temporal memory use compared to that by one master thread, and keeps high performance on larger size input matrices. We are also showing more detailed thread behavior in Figure 6.7. We evaluate the execution time of each thread with static, dynamic or guided option parallelized by row and with static option parallelized by flop on 272 threads. The input matrix is G500 with scale=13 and edge factor=16. Since we focus on the load-balance in Figure 6.7, the memory management scheme by one master thread (static(flops), master) in Figure 6.6 is excluded. The execution time of the busiest thread in "static (flop)" is set as 1, and the relative execution time of each thread to the baseline is plotted in descending order. The scheduling algorithms, which are parallelizing SpGEMM by row, assign the task to each thread without considering the actual computational cost. As a result, the "static (row)" shows terrible load imbalance, and the performance of "static (row)" is determined by the longest run among all threads even though some

threads finish their tasks in 1% of the execution time of the slowest thread. The execution time of almost all threads with "dynamic (row)" algorithm is about the same, while the execution time is longer on the whole due to the overhead of thread scheduling. The "guided (row)" algorithm can reduce the overhead of thread scheduling compared to "dynamic (row)" algorithm, and achieves better load balance than "static (row)". However, some of threads still take much more execution time compared to other threads, that we can see the same issue in the "static (row)". Our approach parallelized by flop with static scheduling shows not only good load balance but also less overhead of thread scheduling.

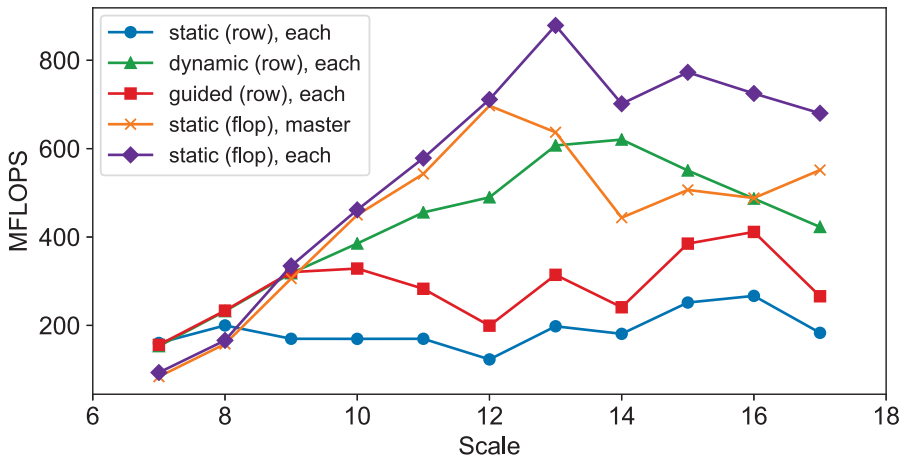


Figure 6.6: Performance of Heap SpGEMM scaling with size of G500 inputs on KNL with Cache mode

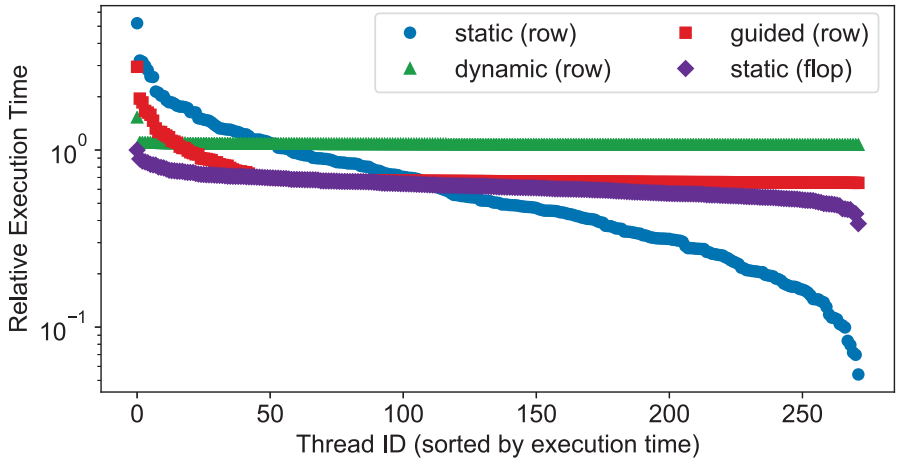


Figure 6.7: Execution time of each thread for SpGEMM with G500 matrix (scale = 13, edge factor=16)

6.6 Experimental Results

Different SpGEMM algorithms can dominate others depending on the aspect ratio (i.e. ratio of its dimensions), density, sparsity structure, and size (i.e. dimensions) of its inputs. To provide a comprehensive and fair assessment, we evaluate SpGEMM codes under several different scenarios. For the case where input and output matrices are sorted, we evaluate MKL, Heap and Hash/HashVector, and for the case where they are unsorted, we evaluate MKL, MKL-inspector, KokkosKernels (with 'kkmem' option) and Hash/HashVector.

6.6.1 Squaring a matrix

Multiplying a sparse matrix by itself is a well-studied SpGEMM scenario. Markov clustering is an example of this case, which requires A^2 for a given doubly-stochastic similarity matrix. We evaluate this case extensively, under using real and synthetically generated data. For synthetic data, we provide experiments with varying density (for a fixed sized matrix) and with varying size (for a fixed density).

Scaling with Density

Figure 6.8 shows the result of scaling with density. When output is sorted, MKL's performance degrades with increasing density. When the output is not sorted, increased density generally translates into increased performance. The performance of all codes except MKL increases significantly as the

ER matrices get denser until edge factor is 16. On the other hand, the evaluation result on G500 matrices with edge factor 32 shows further performance boost. In SpGEMM operation between ER matrices with edge factor 32, the average non-zero elements per row of output matrix or hash table size for each row is about 32^2. As a result, the memory accesses to the accumulators appear to cause L1 cache misses, degrading the performance of SpGEMM compared to the case between matrices with edge factor 16. For G500 matrices, Hash shows superior performance on KNL while HashVector is the best performer on Haswell.

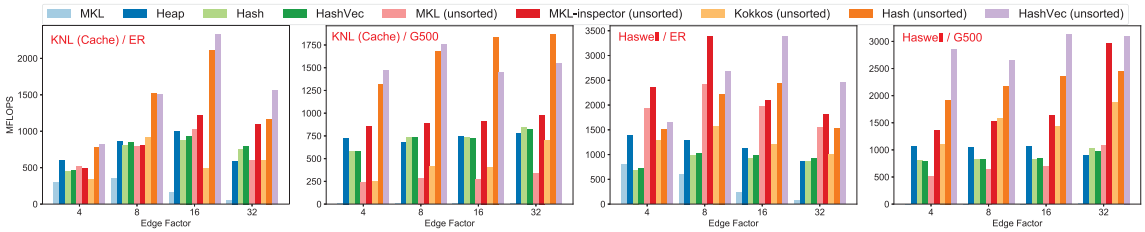


Figure 6.8: Scaling with increasing density (scale 16) on KNL

Scaling with Input Size

Evaluation is running on ER and G500 matrices with scaling the size from 7 to 20 or 17 respectively. The edge factor is fixed as 16. Figure 6.9 shows the results on KNL. MKL family with unsorted output shows good performance for ER matrices with small scale. However, for large scale matrices, MKL goes down, and Heap and Hash outperform. Especially, Hash and HashVector keep high performance even for large scale matrices. When the scale is about 13, the performance gap between sorted and unsorted is large, and it becomes smaller when the scale is getting large. This is because the cost of computation with hash-table or heap becomes larger, and the advantage of removing sorting phase becomes relatively smaller. For G500 matrices, whose non-zero elements of each row are skewed, the performance of MKL is terrible even if its output is unsorted. Since there is no issue about load-balance in Heap and Hash kernels, they show stable performance as ER matrices.

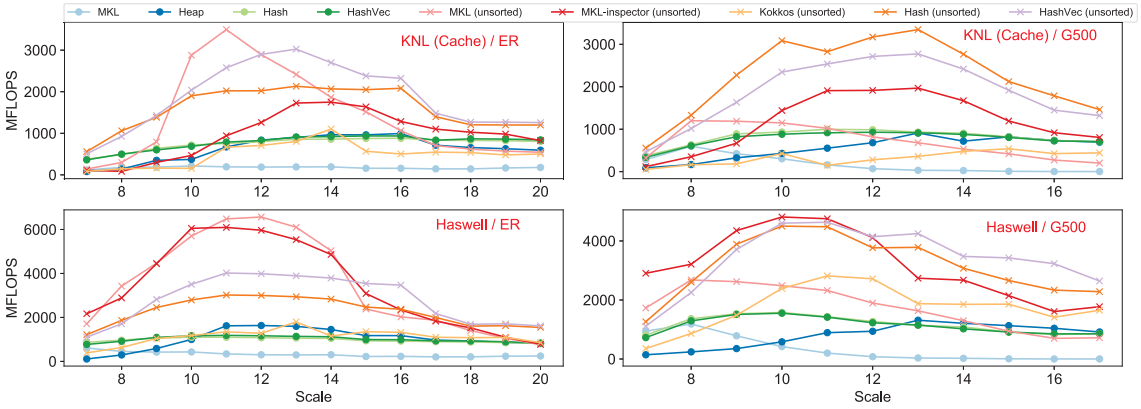


Figure 6.9: Scaling with size on KNL with edge factor 16

Scaling with Thread Count

Figure 6.10 shows the scalability analysis of KNL on ER and G500 matrices with scale=16 and edge factor=16. Each kernel is executed with 1, 2, 4, 8, 16, 32, 64, 68, 128, 136, 192, 204, 256 or 272 threads. We do not show the result of MKL with sorted output since it takes much longer execution time compared to other kernels. All kernels show good scalability until around 64 threads, but MKL with unsorted output has no improvement over 68 threads. On the other hand, Heap and Hash/HashVector get further improvement over 64 threads. This is because our thread scheduling scheme based on the number of floating operations in each row works well and brings better load-balancing.

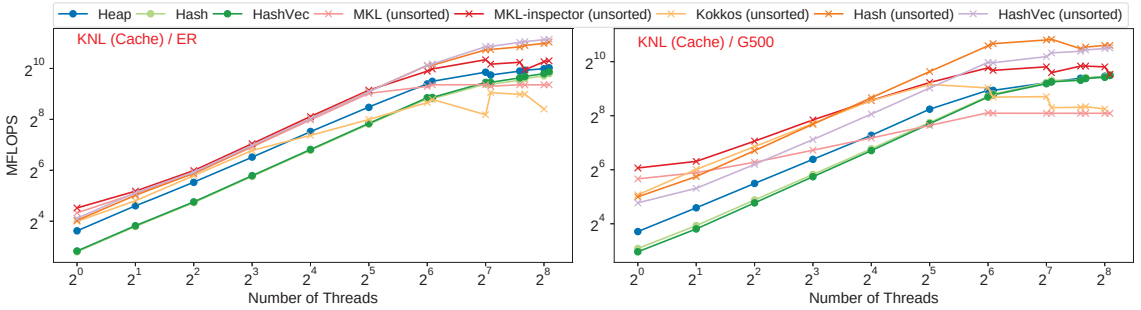


Figure 6.10: Strong scaling with thread count on KNL with ER (left) and G500 inputs (right). Data used is of scale 16 with edge factor 16

Sensitivity to Compression Ratio on Real Matrices

We evaluate SpGEMM performance on 26 real matrices listed in Table 6.2 on KNL. Figure 6.11 shows the result with sorted output and unsorted output respectively in ascending order of compression ratio (= flop / number of non-zero elements of output). The compression ratio is getting larger when the density of the matrix is large. The compression ratio can be the proxy of memory access locality and we can easily predict the performance boost of keeping the output sorted by using the compression ratio of target SpGEMM operation and the performance model proposed in Section 6.4.2. Lines in the graph are linear fitting for each kernel. Firstly, we discuss the result with sorted matrices. The performance of Heap is stable regardless of compression ratio while MKL gets better performance with higher compression ratio. The matrices about graph processing with low compression ratio cause load imbalance and performance degradation on MKL. In contrast, Hash outperforms MKL on most of matrices, and shows high performance independent from compression ratio. For unsorted matrices, we add KokkosKernels to the evaluation, but it underperforms other kernels in this test. The performance of Hash SpGEMM is best for the matrices with low compression ratio and underperforms on high compression ratio matrices. MKL-inspector shows significant improvement especially for the matrices with high compression ratio.

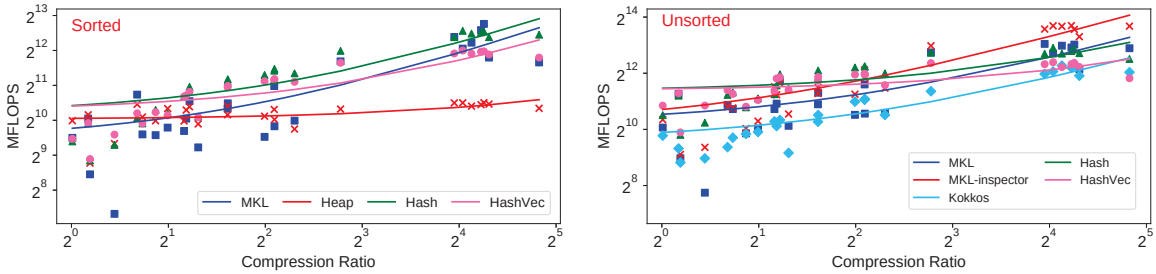


Figure 6.11: Scaling with compression ratio of SuiteSparse matrices on KNL. The algorithms that operate on sorted matrices (both input & output) are on the left and those that operate on unsorted matrices are on the right.

We are also showing L2 cache miss ratio of Heap, Hash and HashVector algorithms on same matrix data in Figure 6.12. We use likwid [79] to profile the L2 cache behavior, and evaluate only SpGEMM part of each algorithm. The cache miss ratio of Hash or HashVector decreases with increased compression ratio both in sorted and unsorted cases. This trend is same as the performance trend showed in Figure 6.11. When the compression ratio is high, the entries in same hash-table is frequently accessed. Since our Hash and HashVector algorithms limit the size of hash-table by the number of flop, the cache hit ratio is likely to increase when the hash-table is repeatedly accessed. As a result, Hash and

HashVector can reduce the ratio of L2 cache miss and achieve better performance on high compression ratio matrices. On the other hand, Heap algorithm shows low L2 cache miss ratios even on matrices with low compression ratio. Therefore, the ratio of L2 cache misses does not change significantly with compression ratio. This is why the performance of Heap algorithm is stable regardless of compression ratio as seen in Figure 6.11.

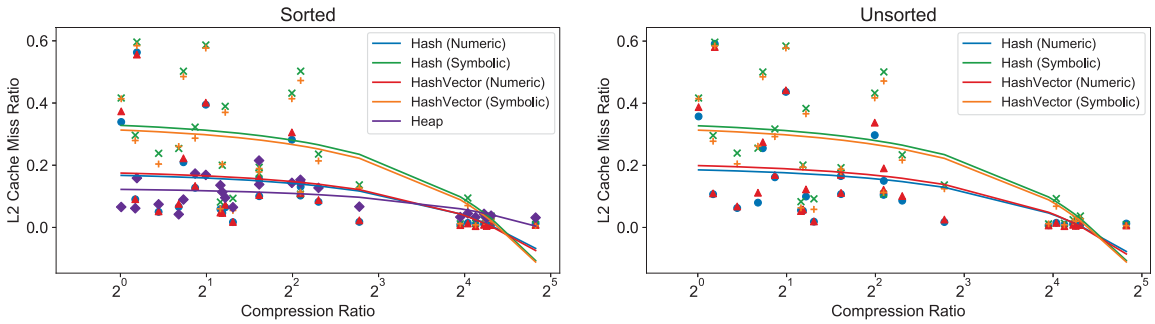


Figure 6.12: L2 cache miss ratio scaling with compression ratio of SuiteSparse matrices on KNL

Comparing sorted and unsorted versions of algorithms that provide the flexibility of choosing either option, we see consistent performance boost of keeping the output sorted. In Hash SpGEMM, for example, sorting requires computation complexity equal to the second term of the equation (6.2) in section 6.4.2. This implies that the unsorted version of SpGEMM operation brings larger benefit on the matrix with lower compression ratio. Such performance boost with unsorted SpGEMM can be seen in other methods, and in particular, the harmonic mean of the speedups achieved operating on unsorted data over all real matrices we have studied from the SuiteSparse collection on KNL is 1.58x for MKL, 1.63x for Hash, and 1.68x for HashVector.

Profile of Relative Performance of SpGEMM Algorithms

We compare the relative performance of different SpGEMM algorithms with performance profile plots [80]. To profile the relative performance of algorithms, the best performing algorithm for each problem is identified and assigned a relative score of 1. Other algorithms are scored relative to the best performing algorithm, with a higher value denoting inferior performance for that particular problem. For example, if algorithm A and B solve the same problem in 1 and 3 seconds, their relative performance scores will be 1 and 3, respectively. Figure 6.13 shows the profiles of relative performance of different SpGEMM algorithms for all 26 matrices from Table 6.2. Hash is clearly the best performer for sorted matrices as it outperforms all other algorithms for 70% matrices and its runtime is always within 1.6x of the best algorithm. Hash is followed by HashVector, MKL and Heap

algorithms in decreasing order of overall performance. For unsorted matrices, Hash, HashVector and MKL-inspector all perform equally well for most matrices (each of them performs the best for about 40% matrices). They are followed by MKL and KokkosKernels, with the latter being the worst performer for unsorted matrices.

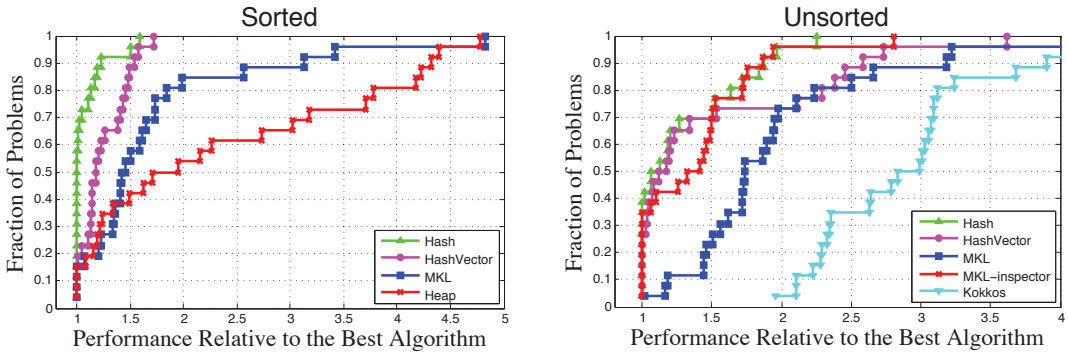


Figure 6.13: Performance profiles of SuiteSparse matrices on KNL using sorted (left) and unsorted (right) algorithms.

Reduction Ratio of Hash Collisions with HashVector

Our HashVector algorithm is designed to reduce the hash collisions by utilizing wide vector registers implemented in Intel KNL. Figure 6.14 shows the ratio of hash collisions to the flop in SpGEMM operation on synthetic matrices generated by R-MAT and SuiteSparse matrices, respectively. HashVector can largely reduce the ratio of hash collisions from Hash on ER matrices compared to G500 matrices. This gap of reduced hash collisions between ER and G500 matrices exposes the difference of best algorithm between ER and G500 matrices. HashVector achieves better performance on ER matrices, while Hash is superior to HashVector on G500 matrices. Since the flop of SpGEMM with G500 matrix is larger than that of ER matrix, the performance degradation due to hash collisions is relatively small. On SuiteSparse matrices, HashVector algorithm causes very few hash collisions. However, Hash outperforms HashVector on most of matrices as Figure 6.11 shows. This is because the hash collisions in Hash algorithm are not so many and reducing hash collisions does not contributed the performance improvement. The increase of the instructions with HashVector dominates the performance of SpGEMM. As a result, HashVector with large reduction of hash collisions is best for ER matrices, and Hash outperforms HashVector on G500 matrices and SuiteSparse matrices.

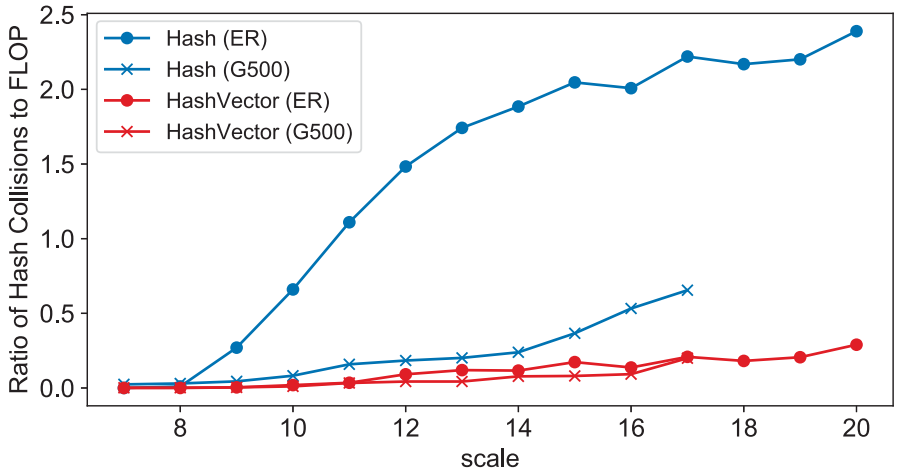


Figure 6.14: Reduction ratio of hash collisions scaling with input size with edge factor 16 on KNL

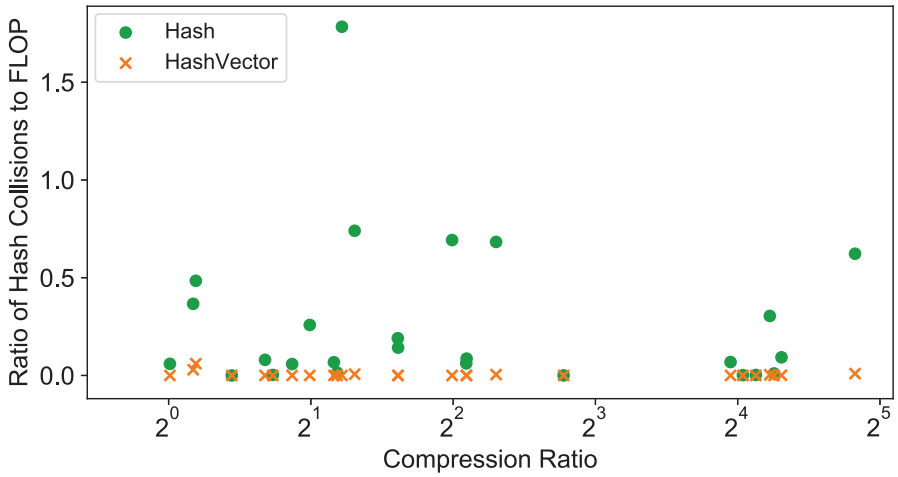


Figure 6.15: Reduction ratio of hash collisions of SuiteSparse matrices on KNL scaling with compression ratio.

6.6.2 Square x Tall-skinny matrix

Many graph processing algorithms perform multiple breadth-first searches (BFSs) in parallel, an example being Betweenness Centrality on unweighted graphs. In linear algebraic terms, this corresponds to multiplying a square sparse matrix with a tall-skinny one. The left-hand-side matrix

represents the graph and the right-hand-side matrix represent the stack of frontiers, each column representing one BFS frontier. In the memory-efficient implementation of the Markov clustering algorithm [58], a matrix is multiplied with a subset of its column, representing another use case of multiplying a square matrix with a tall-skinny matrix. In our evaluations, we generate the tall-skinny matrix by randomly selecting columns from the graph itself. Figure 6.16 shows the result of SpGEMM between square and tall-skinny matrices. A square matrix is generated with scale as 18, 19 or 20. The scale of short side of tall-skinny matrix is set as 10, 12, 14 or 16, and a long side is same as the square matrix. The non-zero pattern of generated matrix is G500 with edge factor=16. The result of square x tall-skinny follows that of A^2 (upper right in Figure 6.9). Both for sorted and unsorted cases, Hash or HashVec is the best performer.

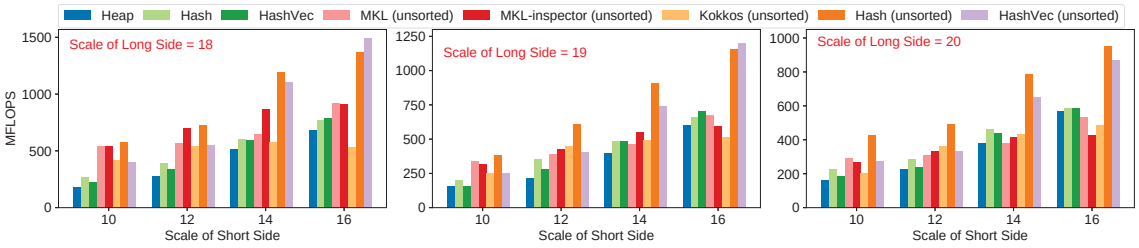


Figure 6.16: SpGEMM between square and tall-skinny matrices on KNL (scales 18, 19, and 20)

6.6.3 Triangle counting

We also evaluate the performance of SpGEMM used in triangle counting [61]. The original input is the adjacency matrix of an undirected graph. For optimal performance in triangle counting, we reorder rows with increasing number of nonzeros. The algorithm then splits the reordered matrix A as $A = L + U$, where L is a lower triangular matrix and U is an upper triangular matrix. We evaluate the SpGEMM performance of the next step, where $L \cdot U$ is computed to generate all wedges. After preprocessing the input matrix, we compute SpGEMM between the lower triangular matrix L and the upper triangular matrix U. Figure 6.17 shows the result with sorted output respectively in ascending order of compression ratio on KNL. Lines in the graph are linear fitting for each kernel. Basically, the result shows similar performance trend to that of A^2. Hash and HashVector generally overwhelm MKL for any compression ratio. Since the triangle counting preprocesses the input matrix before SpGEMM operation, the characteristic of input matrix such as block structure is lost. As the result, the compression ratio tends to decrease compared to the case of A^2, and sorting output takes larger percentage of execution time. While the performance of MKL and Hash for triangle counting degrade compared to the case of computing A^2, the performance of Heap is consistent regardless of compression ratio. Heap performs the best for inputs with low compression ratios.

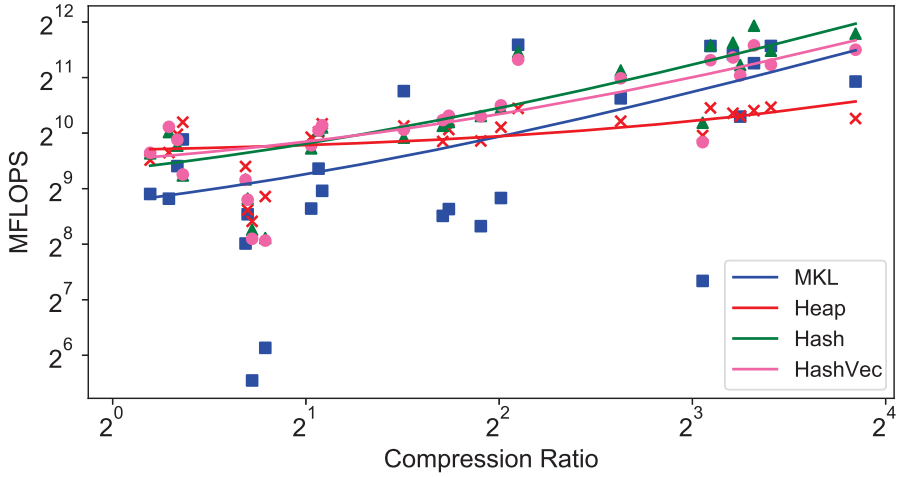


Figure 6.17: Performance of SpGEMM between L and U triangular matrices when used to count triangles on KNL

6.6.4 Empirical Recipe for SpGEMM on KNL

From our evaluation results, best algorithm is different by target use case and inputs. We summarize the recipe to choose best algorithm for SpGEMM on KNL in Table 6.4. The recipe for real data is based on compression ratio, which effects the dominant code. For low compression ratios, especially $L \cdot U$ where output is sparser, Heap shines. In the other cases, Hash and MKL-inspector dominate the high-compression ratio scenarios because Hash has better locality of memory access to hash-table on high compression ratio matrices and can keep low cache miss ratio. On synthetic data, our hash-table-based implementations dominate others for almost cases, and Heap works well for sparser matrices or uniform data. We note that A^2 for uniform input matrices shows low compression ratio. Also, the performance gap between Hash and HashVector comes from how many hash collisions occur in each algorithm. This empirical recipe was already predicted by our analysis in Section 6.4.2.

Table 6.4: Summary of best SpGEMM algorithms on KNL

(a) Real data specified by compression ratio (CR)

		High CR (> 2)	Low CR (≤ 2)
A x A	Sorted	Hash	Hash
	Unsorted	MKL-inspector	Hash
L x U	Sorted	Hash	Heap

(b) Synthetic data specified by sparsity (edge factor, EF) and non-zero pattern

		Sparse (EF ≤ 8)		Dense (EF > 8)	
		Uniform	Skewed	Uniform	Skewed
A x A	Sorted	Heap	Heap	Heap	Hash
	Unsorted	HashVec	HashVec	HashVec	Hash
TallSkinny	Sorted	-	Hash	-	HashVec
	Unsorted	-	Hash	-	Hash

6.7 Impact on the HipMCL clustering software

HipMCL [58] is a high-performance distributed-memory implementation of the Markov cluster algorithm, which itself is commonly used for clustering protein similarity networks. The algorithm performs concurrent random walks from all vertices, which is implemented by squaring the sparse similarity matrix. In the context of protein similarity networks, compression ratios are often high in the first few iterations but they drop rapidly as the algorithm starts to converge. An overwhelming majority of the time is spent during those first few iterations. The current implementation of HipMCL uses the Heap SpGEMM implementation within Combinatorial BLAS [81]. We expect that hash algorithm can provide a significant performance boost because most of the time is spent in iterations with high compression ratios.

The results, shown in Table 6.5, support our hypothesis. The Hash algorithm is up to 10X faster than the Heap algorithm for the SpGEMM time alone. The Hash SpGEMM algorithm accelerates the whole HipMCL pipeline by up to a factor of 2.6X. These impressive performance improvements have significant positive implications because HipMCL is often used in really large datasets that take several hours to cluster on O(1000) of nodes.

In addition to testing the Hash algorithm, we also implemented a hybrid algorithm that chooses Hash or Heap SpGEMM implementation depending on the compression ratio. Instead of making a single coarse decision per iteration, our Hybrid algorithm estimates the compression ratios for each column and chooses a different implementation per column. Specifically, if the compression ratio of

the column is larger than 2, then it uses the Hash algorithm. Otherwise, it uses the Heap algorithm. This threshold of 2 is motivated by the experimental data shown earlier in Figure 6.11. The hybrid algorithm performs only marginally better than the Hash algorithm. This has two reasons. First, an overwhelming majority of the time is spent in high compression ratio computations, thus there is limited benefit that can be gained by choosing a different algorithm in the other regime. Second, Hash algorithm is only slightly worse than the Heap algorithm for low compression ratio scenarios.

However, we believe that this Hybrid approach might be more valuable in other iterative applications that thread a finer line between low and high compression ratio computations. One potential candidate is the multi-source betweenness centrality implementation [57, 82] that uses SpGEMM as its computational workhorse.

Table 6.5: Execution time of HipMCL application on single KNL node using Heap, Hash and Hybrid algorithms [sec]

Dataset	Target	Heap	Hash	Hybrid
Viruses	SpGEMM	20.38	5.14	5.13
	Total	62.59	47.87	47.77
Archaea	SpGEMM	7700.50	717.07	716.42
	Total	11535.00	4550.22	4539.07
Eukarya	SpGEMM	18448.10	1941.93	1964.34
	Total	26717.00	10241.60	10284.80

6.8 Conclusions

We studied the performance of computing the multiplication of two sparse matrices on Intel architectures. This primitive, known as SpGEMM, has recently gained attention in the GPU community, but there has been relatively less work on other accelerators. We have tried to fill that gap by evaluating publicly accessible implementations, including those in proprietary libraries. From architecture point of view, we develop the optimized Heap and Hash SpGEMM algorithms for Intel architectures. Performance evaluation shows that our optimized SpGEMM algorithms largely outperform Intel MKL and Kokkos-kernel.

Our work provides multiple recipes. One is for the implementers of new algorithms on highly-threaded x86 architectures. We have found that the impact of memory allocation and deallocation to be significant enough to warrant optimization as without them SpGEMM performance does not scale well with increasing number of threads. We have also uncovered the impact of MCDRAM for the SpGEMM primitives. When the matrices are sparser than a threshold (≈ 4 nonzeros on average per row), the impact of MCDRAM is minimal because in that regime the computation becomes close to

latency bound. On the other hand, MCDRAM shines as matrices get denser because then SpGEMM becomes primarily bandwidth bound and can take advantage of the higher bandwidth available on MCDRAM. The second recipe is for the users. Our results show that different codes dominate on different inputs. We clarify which SpGEMM algorithm works well by building performance model and showing detailed performance results and profiling data. For example, MKL is a perfectly reasonable option for small matrices with uniform nonzero distributions. However, our heap and hash-table-based implementations dominate others for larger matrices. Similarly, the compression ratio also affects the dominant code. Based on the recipe, the hash-table-based algorithm is applied to HipMCL application and provides significant performance boost. Our results also highlight the benefits of leaving matrices (both inputs and output) unsorted whenever possible as the performance savings are significant for all codes that allow both options. Finally, this optimization strategy for acquiring these two recipes is beneficial for optimization of SpGEMM on future architectures.

Chapter 7

Batched Sparse Matrix Matrix Multiplication

7.1 Introduction

Recently, Deep Learning is getting much attention for its high accuracy in image recognition and other machine learning topics. In image recognition, Convolutional Neural Networks (CNNs), which consist of convolution and pooling layers, effectively expose the features of images, and show significant accuracy. On the other hand, Graph Convolutional Networks (GCNs) have been proposed [4] and can deal with a general graph structure, not only for a regular grid structure such as image. The GCNs show high accuracy in bioassay to predict the characteristics of chemical compounds or protein by treating the structures of substances as graph [83–87]. Furthermore, a knowledge graph is also applied to GCNs [88].

As well as CNNs, GCNs require enormous computational operations, and GPUs with high compute capability are crucial component for the GCNs applications. The structure of input graph is expressed as an adjacency matrix or adjacency list. As the connectivity between nodes among the graph is usually sparse, the adjacency matrix is also sparse. In order to achieve high throughput in training and inference of GCNs applications, the operations on sparse matrix, especially sparse-dense matrix multiplication (SpMM), need to be accelerated. However, the dataset used in GCNs often includes very small graphs, where the number of nodes is less than one hundred. SpMM operation on such small matrix hardly exploits massive parallelism of GPU. Furthermore, it is not clear how to efficiently process SpMM between small matrices since no research in high performance computing or other fields focuses on small sparse matrix. The researchers and developers are forced to process the training or inference of GCNs with low throughput SpMM on small matrices. As a result, the operations on sparse

matrix, especially SpMM, become the bottlenecks in GCNs applications.

We propose Batched SpMM, which increases the parallelism and attains whole computing power of GPU. Firstly, we devise new efficient SpMM algorithm, named Sub-Warp-Assigned (SWA) SpMM, for the data structure such as CSR and SparseTensor in TensorFlow, and integrate cache blocking optimization for utilizing shared memory. Next, we show how Batched SpMM assigns computing resources. The Batched SpMM appropriately applies cache blocking optimization and assigns threads and shared memory to SpMM operations based on the matrix sizes in the batch. We also mention how to efficiently apply our batched approaches to GCNs application, improving the performance of matrix multiplication and addition besides SpMM. The Batched SpMM launches single CUDA kernel and executes tens or hundreds of SpMM operations, corresponding to mini batch size, in parallel. The Batched SpMM enhances the parallelism and throughput of SpMM operations, and also reduces the overhead of CUDA kernel launches.

We have preliminary evaluation of the Batched SpMM on TSUBAME3.0 implementing NVIDIA Tesla P100 GPU. Our batched approaches achieve significant speedups of up to 9.27x compared to non-batched approaches, and attain improvement of up to 6.09x speedup for larger dense input matrix. We also evaluate the performance of Batched GEMM of cuBLAS by treating input sparse matrix as dense matrix, and our Batched SpMM show superior performance to the Batched GEMM. This preliminary evaluation expose the effectiveness of our Batched SpMM to specific batch size and matrix size. The Batched SpMM is integrated to the GCNs application, and accelerates the training and inference of the application. The Batched SpMM attains the speedups of up to 1.59x and 1.37x in training and inference, respectively.

7.2 Background

7.2.1 Graph Convolution

A graph convolution is a general convolutional operation, according to a graph structure, while the conventional convolutional operation assumes regular grid structure. A graph convolutional network consists of multiple graph convolution layers. Graph structures and input features are given as input data, and the convolutional operation is executed along with its graph structure. More specifically, the features of adjacent nodes to the target node is convoluted with filters. This operation is formulated as below when a graph is $G = \{V, E\}$, feature vector of the node $v \in V$ is x_v and a set of filter is expressed as a weight matrix, W.

$$y_i = \sum_{j \in V} a_{ij} x_j^T W \tag{7.1}$$

The adjacency matrix is set as $a_{uu} = 1$, and if an edge from u to v exists, $a_{vu} = 1$, otherwise 0. Executing the operations for each node among the graph is regarded as the multiplication between the

adjacency matrix A, feature vectors and weight matrix as below.

$$Y = AXW \tag{7.2}$$

The adjacency matrix representing a graph structure usually shows sparse property, while the feature vectors and weight matrix are expressed as dense matrix.

7.2.2 Sparse Matrix Format

Sparse matrix is usually compressed by removing zero elements and holding only non-zero elements, which are necessary for computation. Sparse matrix format is designed to reduce both operations and memory usage by compression, and various sparse matrix formats have been proposed. Coordinated (COO) and Compressed Sparse Row (CSR) are widely used. Figure 7.1 shows an example of each sparse matrix format. The COO format has the tuple of value, row index and column index of each non-zero element in the matrix. The CSR accumulates the non-zero elements with same row indices, and then manages the non-zero elements by holding row pointer (rpt), which indicates the beginning point of each row. The CSR can reduce memory usage compared to the COO format. In TensorFlow [89], one of the sophisticated deep learning frameworks, sparse matrix (tensor) is treated as SparseTensor class object. As Figure 7.1 shows, the data structure of SparseTensor is similar to COO. In SparseTensor, the indices of each non-zero elements are stored as the array of the pairs of the row and column indices.

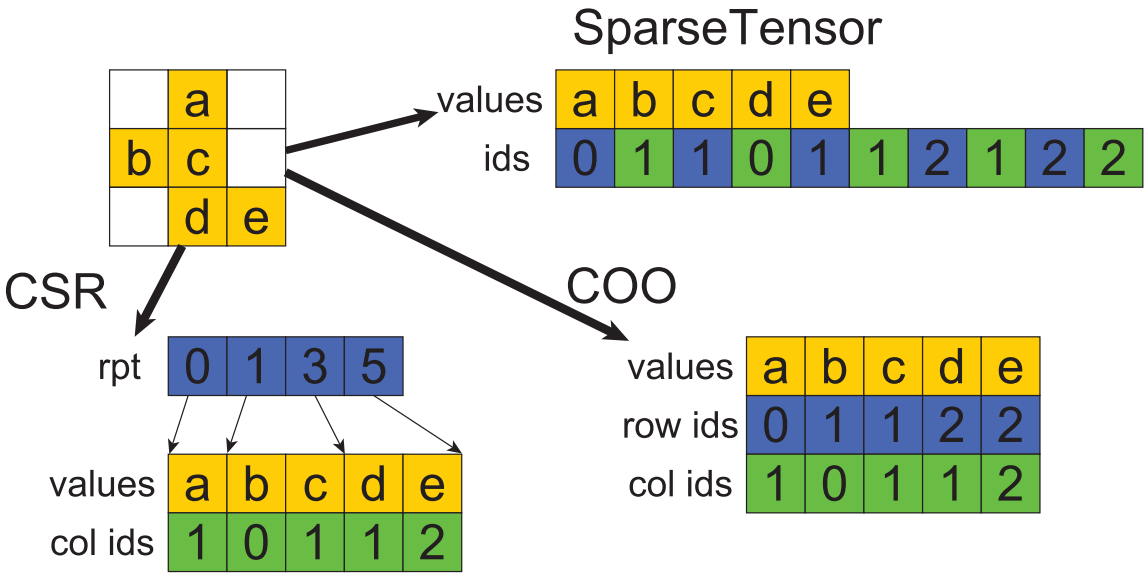


Figure 7.1: Example of sparse matrix formats

7.2.3 Sparse Matrix Multiplication

Let A, B be input matrices, sparse matrix multiplication (SpMM) computes $C = AB$ when A is sparse matrix and B is dense matrix. In the rest of this paper, the number of non-zero elements, row size and column size of a matrix X is written in nnz_X, m_X, n_X, respectively. Figure 7.2 shows the pseudo code of SparseTensorDenseMatMul routine, which computes SpMM in TensorFlow. The multiplications for each non-zero element are processed one after another. In CUDA code designed for GPU, two "**for**" loops are parallelized by $nnz_A * n_B$ threads, and each thread computes a multiplication independently. The product is added to output matrix by using atomic operation since the memory access to output matrix and the "add" operation by each thread have a chance of race condition. When the negative influence of atomic operations is enough negligible, the work assigned to each thread is about equal and the SparseTensorDenseMatMul on GPU can achieve good load balance. However, this approach has poor locality of memory access in spite of matrix-matrix multiplication. Also, the performance is limited by the atomic operations on the global memory of GPU. To make matters worse on small matrices, the approach with $nnz_A * n_B$ threads hardly exploits enormous parallelism of GPU.

SparseTensorDenseMatMul(C, A, B)

```
1  // set matrix C to O
2  for i ← 0 to nnz_A
3    do for j ← 0 to n_B
4         do rid ← ids_A[i * 2]
5            cid ← ids_A[i * 2 + 1]
6            val ← values_A[i]
7            C[rid][j] ← C[rid][j] + val * B[cid][j]
```

Figure 7.2: Pseudo code of the multiplication kernel between sparse tensor and dense matrix provided in TensorFlow. A is stored as SparseTensor data structure.

7.3 Related work

7.3.1 SpMM for GPU

Vázquez and *et al.* proposed new approach of SpMM for GPU with ELLR-T format, which has favorable property of memory access on GPU. While the approach of SpMM with ELLR-T achieves higher performance, the format conversion from COO or CSR format causes performance degradation when the calculation on each matrix is executed only once.

Hong and *et al.* proposed new sparse matrix format and algorithm for SpMM named Row-Segmented-SpMM (RS-SpMM) [90]. The sparse matrix is divided into two groups. One is dense non-zero area and the other is rest of non-zero elements. Each part is processed by it own kernel. This approach can improve the locality of memory access and reduce the memory access to global memory. Also, in order to identify "dense" area, the performance model is developed and brings enough high performance even compared to best case. The format conversion including parameter setting and preprocessing for DCSR format [91] requires the cost equivalent to several times of SpMM calculations.

Yang and *et al.* proposed the state-of-the-art SpMM algorithm for GPU with CSR format [92]. The authors proposed two kernels, and the heuristic selects between the kernels by input matrix with a high accuracy. One kernel for SpMM adopts merge-based approach [93] designed for improving the load-balance in SpMV, and the other kernel is Row-splitting, performing coalesced memory access. It is not clear that the proposed approach is effective for smaller matrices since the approach is evaluated only on large matrices.

7.3.2 Batched BLAS

The calculation on a dense matrix divided into small sub matrices [94, 95] or sparse matrix with block structure [96–98] is dealt with the operations on small dense matrices. The computing kernel on small matrices suffers from the overhead of repeated kernel launches, especially on GPU, and the high parallelism and memory bandwidth of GPU are hardly exploited, thus the total performance of applications is decreased. To overcome these issues, a new basic routine called Batched BLAS has been proposed handling multiple data and operations in a single kernel [99–101]. Even if the data sizes in the batch are different to each, Batched BLAS keeps good load-balance. The approach for GEMM (multiplication between dense matrices) on small matrices is also proposed [102], and a lot of GEMM operations between small matrices can be executed with high throughput. Recently, Batched SpMV for small matrices has been proposed [103]. Our batched approach is very similar concept to the Batched SpMV, but the paper about Batched SpMV includes many assumptions (eg. same non-zero pattern among matrices in batch), and the Batched SpMV is highly specialized for the application. Thus, the batched approach is very simple by adding z-axis parallelism to thread grid in CUDA, and the load imbalance among a batch is not taken into account.

7.3.3 GCNs Application

The GCNs applications are actively developed in bioinformatics or chemoinformatics areas as a practical use case of machine learning approach. Deepchem, an open source library, is publicly available, providing the deep-learning approaches to wide range areas such as drug design and

materials chemistry [104]. DeepChem manages a graph structure as adjacency lists. The convolutional operations accumulate the feature vectors from adjacent nodes of the target node by exploring the adjacency lists. The convolutional operations in DeepChem hardly exploit high parallelism of GPU.

A library for deep learning in chemistry area named Chainer Chemistry is also publicly available [105]. Chainer Chemistry enables to predict the characteristics of chemical compounds by applying a molecular structure to the deep learning approaches. The data with graph structure is applied to GCN as input data. Chainer Chemistry, however, manages an adjacency matrix as dense matrix while the matrix shows sparse property. This policy causes many redundant zero-related calculations. Furthermore, it is hard to keep the data on memory or cache when the graph becomes large.

7.4 Proposal

The GCNs application executes convolutional operations in accordance with graph structure by computing SpMM when the graph structure is expressed as sparse matrix. Learning phase of GCNs requires many SpMM executions since the GCNs need to process a lot of data through the graph convolution layers. However, SpMM on very small matrices such that the dimension is tens or hundreds hardly utilizes the high computing capability of GPU. Therefore, the overhead of CUDA kernel launch for each SpMM operation appears to dominate the total GCNs performance. We propose Batched SpMM, which is high-throughput batched approach for processing many SpMM operations especially on small sparse matrices. Firstly, we devise new efficient SpMM algorithm, named Sub-Warp-Assigned (SWA) SpMM, for the data structures such as CSR and SparseTensor in TensorFlow. Next, we show the cache blocking optimization to the SWA SpMM for utilizing fast shared memory. Finally, we demonstrate the batch strategy for the cache blocking and the assignment of GPU computing resources such as threads and shared memory. We also mention how to efficiently integrate our batched approaches to GCNs application. In our supposed scenario of GCNs application, the format conversion for all sparse matrices causes expensive overhead. Thus, we focus on developing algorithms for simple sparse matrix formats such as COO and CSR. Our target application is implemented with TensorFlow, and stores sparse matrix data as SparseTensor data structure. We assume that the non-zero elements are not sorted by row or column indices in SparseTensor data structure. The column size of dense input matrix in SpMM operation is the size of model in the application. Each column size of dense input in the batch is same.

7.4.1 Sub-Warp-Assigned SpMM

The SparseTensorDenseMatMul in TensorFlow launches $nnz_A * n_B$ threads, and one thread computes one multiplication-and-addition. Thus, n_B threads is assigned to each non-zero element. However, the actual CUDA implementation requires many operations for each thread to find which non-zero elements or column is assigned, reducing the efficiency of parallelism. Furthermore, when n_B becomes larger, especially overcomes 32, more threads access to same non-zero element. This increases memory access demand and instruction counts. That is why the SparseTensorDenseMatMul results in lower performance. We propose Sub-Warp-Assigned (SWA) SpMM, which assigns up to 32 threads (1 warp) to each non-zero element or row. The $subWarp$ is set based on the column-size of dense input matrix, i.e. n_B.

$$subWarp = \begin{cases} 32 & (n_B > 16) \\ \min 2^p \ s.t. \ n_B \leq 2^p & (n_B \leq 16) \end{cases}$$

The number of assigned threads to each non-zero element is set as the power of two. This enables to compute division and modulo operations by executing low-cost bit operations. Furthermore, by setting the limit of assigning threads up to 32 threads to each non-zero element, the SWA SpMM reduces redundant memory pressure and instruction counts. We firstly show the algorithm of SWA SpMM for SparseTensor data structure, and then show that for CSR format. The algorithm for SparseTensor is easily applied to COO format.

Figure 7.3 shows the pseudo CUDA code of SWA SpMM algorithm for SparseTensor data structure. The SWA SpMM algorithm assigns $subWarp$ of threads to each non-zero element. The memory access to the dense input matrix and output matrix by threads among same $subWarp$ is coalesced. As well as SparseTensorDenseMatMul, the SWA SpMM algorithm is load balanced since the work is parallelized by the number of non-zero elements. Since another $subWarp$ assigned to different non-zero element has a chance of accessing same entry of output matrix, add operation is atomically executed.

SWA_SpMM_ST($C, A, B, subWarp$)

1 // A is stored as SparseTensor data structure
2 // set matrix C to O
3 $i \leftarrow threadid$
4 $nzid \leftarrow i\,/subWarp$
5 $rid \leftarrow ids_A[nzid * 2]$
6 $cid \leftarrow ids_A[nzid * 2 + 1]$
7 $val \leftarrow values_A[nzid]$
8 **for** $j \leftarrow (i\,\%subWarp)$ **to** n_B **by** subWarp
9 **do Atomic**$(C[rid][j] \leftarrow C[rid][j] + val * B[cid][j])$

Figure 7.3: Pseudo CUDA code of Sub-Warp-Assigned SpMM for SparseTensor data structure

Figure 7.4 shows the pseudo CUDA code of SWA SpMM algorithm for CSR format. By utilizing the characteristic of managing non-zero elements by row, the SWA SpMM for CSR assigns $subWarp$ to each row. As well as SWA SpMM for SparseTensor, the threads among same $subWarp$ access to same non-zero element, and the memory access to dense input and output matrix is coalesced. The significant difference from the SWA SpMM for SparseTensor is that the algorithm for CSR causes no data race on accessing output matrix. The product can be added without atomic operation.

SWA_SpMM_CSR($C, A, B, subWarp$)

1 // A is stored as CSR format
2 // set matrix C to O
3 $i \leftarrow threadid$
4 $rid \leftarrow i\,/subWarp$
5 **for** $nzid \leftarrow rpt_A[rid]$ **to** $rpt_A[rid + 1]$
6 **do** $cid \leftarrow colids_A[nzid]$
7 $val \leftarrow values_A[nzid]$
8 **for** $j \leftarrow (i\,\%subWarp)$ **to** n_B **by** subWarp
9 **do** $C[rid][j] \leftarrow C[rid][j] + val * B[cid][j]$

Figure 7.4: Pseudo CUDA code of Sub-Warp-Assigned SpMM for CSR format

7.4.2 Efficient use of shared memory and cache blocking

Utilizing shared memory and improving the locality of memory access highly contribute to the performance of matrix multiplication on GPU. Shared memory is implemented on each SM, and provides fast memory access. Also, hardware support improves the performance of atomic operations on shared memory. In our approach for both SparseTensor data structure and CSR format, output matrix is temporally placed on shared memory.

For small matrix stored as COO format or SparseTensor data structure, our approach utilized shared memory as Figure 7.5-(a). Utilizing shared memory for output matrix reduces the overhead of CUDA kernel launch for initializing output matrix. As Figure 7.3 shows, the output matrix needs to be initialized with zero before actual SpMM operation starts. The overhead of another CUDA kernel launch for initialization is not negligible, especially for SpMM on small matrices. If n_B or sparse matrix is large, whole output matrix can not be placed on shared memory of single streaming multiprocessor (SM). In this case, cache blocking optimization divides the output matrix along the column, as Figure 7.5-(b) shows. The cache blocking optimization enables SWA SpMM to utilize fast shared memory and achieve high performance even for larger input matrices. Not only for output matrix, the locality of memory access to input dense matrix is also improved.

While each $subWarp$ in SWA SpMM for SparseTensor specifies only which column of output matrix is assigned without index information of non-zero element, the SWA SpMM for CSR can easily specify which row and column of output matrix is assigned. Thus, shared memory does not need to keep whole output matrix. A $subWarp$ only needs shared memory with the size of n_B. If $TB/subWarp * n_B$, TB is thread block size, exceeds the capacity of shared memory, the cache blocking optimization is applied as Figure 7.5-(d) shows. As well as COO format, the output matrix and input dense matrix are divided along the column.

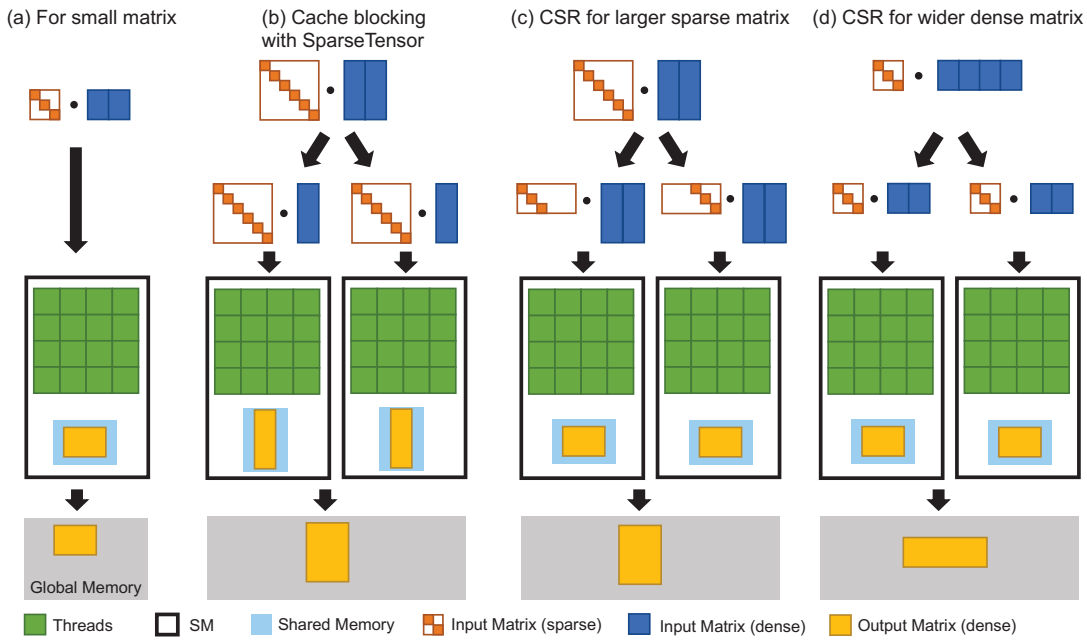


Figure 7.5: Utilization of shared memory and cache blocking optimization for SpMM

7.4.3 Batched Algorithm for SpMM

Our Batched SpMM manages tens or hundreds of SpMM operations by single CUDA kernel launch. Based on the sizes of input sparse and dense matrices, the Batched SpMM algorithm decides whether the cache blocking is applied and how many threads assigned to whole SpMM operations.

The Batched SpMM for SparseTensor decides whether cache blocking optimization is applied by the maximum size of output matrix in batch, i.e. $\max m_A * n_B$. There are three cases by the sizes of input matrices.

1. Whole output matrix is small enough to be placed on shared memory (Figure 7.5-(a))
2. Sub output matrix divided by cache blocking can be placed on shared memory (Figure 7.5-(b))
3. A graph (sparse matrix) is too large to use shared memory even with cache blocking optimization

However, when we assume that the size of shared memory to each SpMM operation in single precision is 32KB, only the input sparse matrices with $m_A > 8192$ require the case 3). In this case, the SpMM operations do not need to be batched, and it is better option to execute single SpMM operation

by optimized kernel for large input sparse matrix. We focus on only the cases 1) and 2) although we support the case 3) as the implementation without using shared memory. The Batched SpMM algorithm applies cache blocking optimization to all SpMM operations in the batch if an output matrix which cannot be placed on shared memory exists. The Batched SpMM assigns one thread block to each SpMM operation for whole matrix or sub matrix. For example, we consider that the Batched SpMM executes one hundred of SpMM operations in single precision and the capacity of shared memory assigned to each thread block is 32KB. If any output matrix in the batch can be placed on shared memory without cache blocking, that is $m_A * n_B * \text{sizeof(float)} \leq 32 * 1024$, the Batched SpMM launches one hundred thread blocks and each thread block computes one SpMM operation. If one SpMM operation in the batch requires cache blocking and the output matrix is divided into two sub matrices, two hundred thread blocks are launched and each thread block is assigned to one SpMM operation on sub matrix.

The required number of threads for each SpMM operation in the Batched SpMM for CSR format is simply $subWarp * m_A$. As Figure 7.5-(c) shows, larger input matrix increases the number of thread blocks launched. The Batched SpMM for CSR format applies cache blocking optimization only when n_B is large as Figure 7.5-(d) shows. If the column size of dense input, n_B, is small enough, the Batched SpMM launches $\max m_A * subWarp * batch$ threads as a whole. If one SpMM operation in the batch needs cache blocking optimization and the output matrix is divided into p sub matrices, the Batched SpMM launches $\max m_A * subWarp * batch * p$ threads as a whole. Although the Batched SpMM for CSR launches more threads than necessary if the batch includes various sizes of input sparse matrices, these redundant threads do not become a problem since the threads terminates immediately.

7.4.4 Batched SpMM for GCNs Application

Our target GCNs application named "ChemGCN" constructs a neural network with multiple graph convolution layers. The application is written with TensorFlow, but our batched approach does not limit the framework or implementation. In our implementation with TensorFlow, we focus on applying the Batched SpMM for SparseTensor, but the Batched SpMM for CSR format can also be utilized in another framework such as Chainer, which supports CSR format. Figure 7.6 shows the flow of the graph convolution layer in ChemGCN. The graph convolution layer executes convolutional operation for each input data. Firstly, the feature vectors on each vertex within the input graph is multiplied by the weight matrix for each channel (MatMul). The bias is added to the multiplication result (Add), and the result is convoluted along with graph structure (SpMM). Finally, the output of SpMM for each channel is accumulated. This GraphConvolution layer requires $batchsize * channel$ times of kernel launches for MatMul, Add and SpMM. Since input matrices are small, the computing kernels for these three operations hardly exploit high parallelism of GPU. The overhead of CUDA kernel launches become critical performance issue. The implementation insufficient for the performance is simply due

to no availability of batched approach for SpMM.

```
GraphConvolution(Y, A, X, W, bias)
1  for b ← 0 to batchsize
2    do for ch ← 0 to channel
3         do U ← MatMul(X[b], W[ch])
4            B ← Add(bias[ch], U)
5            C[ch] ← SpMM(A[b][ch], B)
6     Y[b] ← ElementWiseAdd(C)
```

Figure 7.6: Pseudo code of graph convolution layer in GCN application without batched optimization

Figure 7.7 shows the algorithm of graph convolution layer with the Batched SpMM. To attain more parallelism, the operations within mini batch are batched. The Batched SpMM applied to the graph convolution layer enables MatMul and Add to be also executed by single kernel, respectively. Due to the limitation of the framework, the dimensions of input tensors are changed. In Figure 7.7, the input dense tensor is reshaped from $\mathbb{R}^{m_X \times n_X \times batchsize}$ to $\mathbb{R}^{(m_X * batchsize) \times n_X}$. However, this operation just changes the meta data of input data, and brings only a little overhead. Generating the list of adjacency matrices for executing Batched SpMM requires only accumulating the pointers to the objects. In this way, the operations on the graph convolution layer can be batched with little overhead. The CUDA kernel launches for computing convolution are largely reduced from $O(channel * batchsize)$ to $O(channel)$. The Batched SpMM is also applied to backward propagation.

```
GraphConvolutionBatched(Y, A, X, W, bias)
1  for ch ← 0 to channel
2    do Xr ← Reshape(X, (m_X * batchsize, n_X))
3       U ← MatMul(Xr, W[ch])
4       B ← Add(bias[ch], U)
5       A_list ← [A[0][ch], ..., A[batchsize − 1][ch]]
6       C[ch] ← BatchedSpMM(A_list, B)
7  Y ← ElementWiseAdd(C)
```

Figure 7.7: Pseudo code of graph convolution layer in GCN application with batched optimization

7.5 Performance Evaluation

We evaluated the performance and effectiveness of the Batched SpMM. First we conducted the preliminary evaluation on randomly generated matrices to see the performance difference of SpMM between the non-batched and batched approaches. Next we evaluated the performance of ChemGCN application with the Batched SpMM.

We used TSUBAME3.0 at Tokyo Institute of Technology, which has two sockets of Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz and four cards of NVIDIA Tesla P100-SMX2 GPU per computing node. In our evaluation, we used only one GPU. The Tesla P100 GPU consists of 56 SMs, which are 3584 of CUDA cores, and has 16GB HBM2 main memory with 732GB/sec peak memory bandwidth. The shared memory with 64KB is implemented on each SM, and 4MB L2 cache is shared among all threads. The OS was SUSE Linux Enterprise Server 12 (x86_64), and the version of CUDA was 9.0.176. Our target GCNs application is implemented with TensorFlow. The version of TensorFlow was 1.8.0, and cuDNN was ver.7.0. All evaluation in this paper was in single precision.

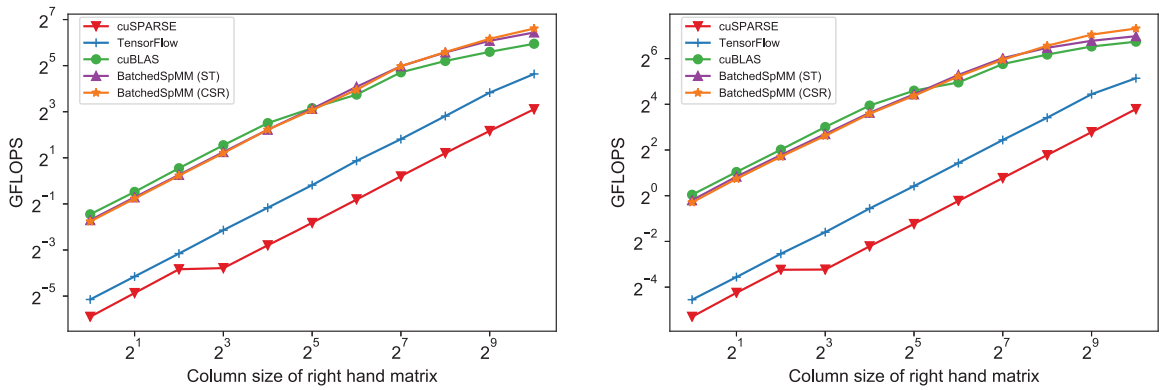
7.5.1 Preliminary Evaluation Results

We evaluated the performance of batched approaches on randomly generated sparse matrix data. We used SpMM functions from cuSPARSE library with CSR format. We evaluated both "csrmm()" and "csrmm2()", and show the best one as "cuSPARSE". We implemented SpMM function with SparseTensor-like format following SparseTensorDenseMatMul as the non-batched approach. Compared to the non-batched approaches, we evaluated Batched SpMM for CSR and SparseTensor-like formats. In addition to them, we evaluated the performance of "gemmBatched()" from cuBLAS library by treating the input sparse matrix as dense matrix. This Batched GEMM function is optimized for processing many GEMM operations by single kernel. The Batched GEMM can exploit GPU's computing power while most of operations are zero-related. Although we just show the throughput of Batched GEMM as evaluation result, holding sparse matrix as dense format requires much memory usage. When we consider the practical application scenario, much more memory requirement by dense format makes it hard to keep the data on GPU memory, causing more memory transfer between CPU and GPU. Since our target is graph data, the randomly generated sparse matrices are square. The row size (dim) and nnz/row are parameterized in generating matrix, and the non-zero pattern is different from each other. The column size of input dense matrix (n_B) is also parameterized. The performance number is showed with FLOPS, calculated by $2 * nnz_A * n_B / exe\_time$. Since the concern is actual execution time, the "gemmBatched()" also follows this performance metric although many zero-related operations are executed. The Batched SpMM and Batched GEMM requires the pointer to each matrix data to be placed on device memory, while it is stored on host memory in the non-batched cases. Our evaluation for batched approaches

includes memory transfer of pointer arrays from host to device.

Figure 7.8 shows the performance of non-batched and batched approaches for SpMM. The parameter setting for the input sparse matrices in the evaluation is based on dataset and configurations of GCNs application showed in later part of this section. The evaluation result in Figure 7.8 works as a proxy for estimating the impact of Batched SpMM applied to the GCN application. In both cases in Figure 7.8, the Batched SpMM shows best performance on fat dense matrices. The non-batched approaches, which sequentially process SpMM, cannot achieve high performance due to the overhead of repeated CUDA kernel launches and poor parallelism of GPU. On the other hand, all of the batched approaches acquire the large performance gain. Compared to the implementation of SpMM following TensorFlow, the Batched SpMM achieves significant speedups of up to 9.27x at $n_B = 64$ in Figure 7.8-(a) and 6.09x at $n_B = 512$ in Figure 7.8-(b). To dive into more detailed analysis, we evaluated the *sm_efficiency* by using nvprof, which is one of the profiler tools for NVIDIA GPU. The *sm_efficiency* is the percentage of active streaming multiprocessors (SMs). While the *sm_efficiency* with non-batched implementation of SpMM following TensorFlow is 35.51% at $n_B = 512$ in Figure 7.8-(b), the Batched SpMM for SparseTensor and for CSR show 89.07% and 87.87%, respectively. This result simply shows that the Batched SpMM exploits GPU's computing resources.

The Batched GEMM routine of cuBLAS also shows high throughput although it includes many zero-related operations. This is because the target sparse matrices are small and do not include so much zero-related operations. It should be noted, however, that treating as sparse matrix and executing as SpMM is natural, and the Batched SpMM overcomes cuBLAS. The performance improvement of Batched SpMM from cuBLAS is 1.26x at $n_B = 64$ in Figure 7.8-(a) and 1.43x at $n_B = 512$ in Figure 7.8-(b). In the cases with smaller n_B, the Batched GEMM of cuBLAS shows superior performance to our Batched SpMM. This is because the actual execution time on GPU is too short and the overhead for more memory transfer from CPU to GPU with Batched SpMM compared to Batched GEMM becomes dominant.

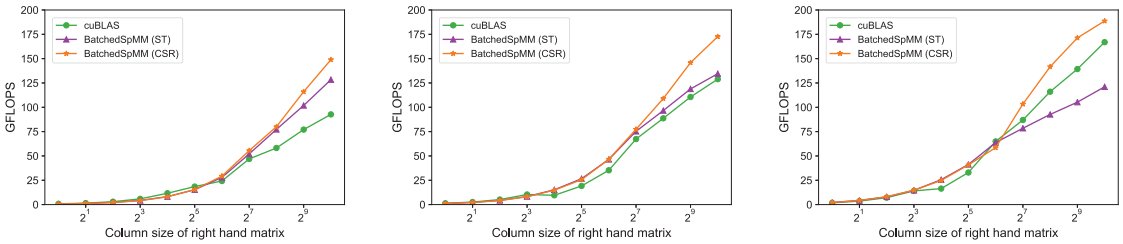


(a) $batchsize = 50, dim = 50, nnz/row = 2$

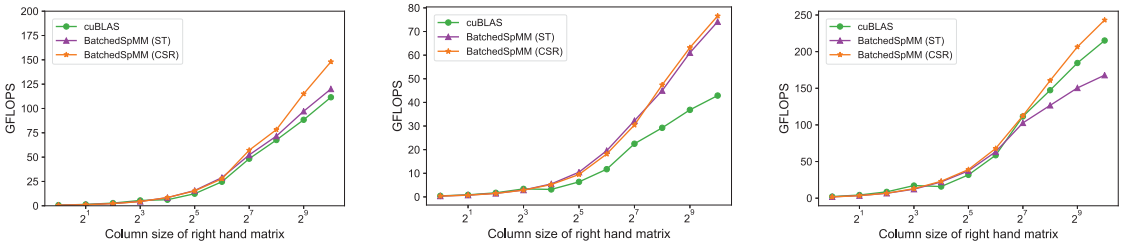
(b) $batchsize = 100, dim = 50, nnz/row = 3$

Figure 7.8: Performance of SpMM and Batched GEMM on randomly generated dataset following the one used in target GCNs application. BatchedSpMM (ST) represents the result of Batched SpMM for Sparse Tensor.

Next, we show the evaluation results of three batched approaches with changing the parameters of input matrices and batch size. The Figure 7.9-(b) and (d) show the performance difference between *batchsize*. The results indicate that more batch size brings more throughput to all batched approaches. The batched approaches with $batchsize = 50$ fail in using all SMs on GPU. While small batch size hardly exploits the computing power of GPU, larger batch size, $batchsize = 100$, ensures enough parallelism. The first row of Figure 7.9 (i.e. (a), (b) and (c)) shows the result of $dim = 32, 64$ and 128, respectively. By increasing the size of input sparse matrix, the Batched SpMM for CSR and cuBLAS are getting better performance numbers. This performance boost simply comes from the improvement of parallelism. The Batched SpMM for CSR format launches more threads in proportion to the row size of input sparse matrix. Thus, a larger input sparse matrix can improve the occupancy of GPU. While the cuBLAS also increases the parallelism on larger input sparse matrix, the performance improvement is relatively smaller than Batched SpMM for CSR since the sparsity is also increased. The Batched SpMM for SparseTensor shows only slight performance change. This is because the number of launching threads does not increase along with the row size of input sparse matrix. The Figure 7.9-(e) and (f) show the evaluation results of $nnz/row = 1$ and 5, respectively. Obviously, the Batched SpMM works efficiently on sparser matrices, and the cuBLAS appears to show better performance on denser matrices. For low nnz/row matrices, the Batched SpMM both for SparseTensor and for CSR are superior to cuBLAS. On the other hand, the performance improvement by Batched SpMM for SparseTensor is limited on denser input sparse matrix due to more race condition by atomic operations. The Batched SpMM for CSR is atomic-free algorithm, and keeps best performer on denser input sparse matrices.



(a) $batchsize = 100, dim = 32, nnz/row = 3$ (b) $batchsize = 100, dim = 64, nnz/row = 3$ (c) $batchsize = 100, dim = 128, nnz/row = 3$



(d) $batchsize = 50, dim = 64, nnz/row = 3$ (e) $batchsize = 100, dim = 64, nnz/row = 1$ (f) $batchsize = 100, dim = 64, nnz/row = 5$

Figure 7.9: Performance of Batched approaches of SpMM on randomly generated dataset

Finally, we consider the batch includes various sizes or densities of input sparse matrices. Figure 7.10 shows the result with mixed sizes and densities, $batchsize = 100, dim = [32, 256], nnz/row = [1, 5]$. We excluded cuBLAS from this evaluation since the kernel only processes GEMM operations with same matrix sizes. The Batched SpMM achieves significant improvements compared to the non-batched approaches even though the matrices among batch have different shapes to each. At $n_B = 1024$, our Batched SpMM achieves up to 3.29x speedup from the non-batched approaches.

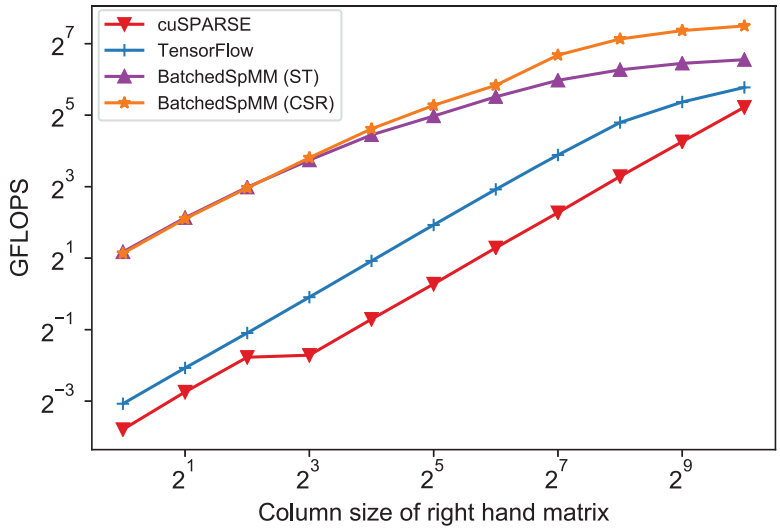


Figure 7.10: Performance of SpMM on randomly generated dataset ($batchsize = 100$, dim and nnz/row are mixed.)

7.5.2 Evaluation on GCNs Application

We evaluated the impact of the Batched SpMM in ChemGCN application implemented with TensorFlow. A graph convolutional network consists of several graph convolution layers. After each graph convolution layer, batch normalization is executed. Table 7.1 shows the dataset and the configurations for the application. We used Tox21 [106] and Reaction100 datasets, which predicts the reaction from its own graph structure of compound structure by selecting 100 kinds of typical reactions from Reaxys database [107].

Table 7.1: Dataset and configurations in the evaluation of ChemGCN application

	#Matrices	Max dim	Epoch	Batch size
Tox21	7,862	50	50	50
Reaction100	75,477	50	20	100

The #Matrices in Table 7.1 is the number of pairs of adjacency matrix and feature matrix. The model is trained by K-fold cross validation, and k=5 in our evaluation. The trained model is used for the evaluation of inference. The evaluation result of inference is the execution time for inferring all data of dataset. In the evaluation of inference, the batch size is set as 200 to increase the throughput

since the batch size does not affect the accuracy of inference. The evaluation results of both training and inference are the mean of 5 executions. We used the Batched SpMM for SparseTensor data structure as the batched approach. It should be noted that our batched optimization does not change the configuration or hyper parameters from non-batched version, and thus no effects on the accuracy in training. The architecture for Tox21 includes two graph convolution layers. The column size of each weight matrix is 64. For Reaction100, three graph convolutional layers are stacked, and the column size of each weight matrix is 512.

Table 7.2 and 7.3 show the performance of training and inference in ChemGCN application, respectively. The training of Tox21 dataset on CPU finishes in less execution time compared to that on GPU with non-batched approach. This is because Tox21 dataset is small enough that all data such as adjacency matrices and feature matrices can be placed on last-level cache of CPU. The non-batched approach does not exploit the computing power of GPU due to the lack of parallelism, and the overhead of repeated CUDA kernel launches degrades the performance. In contrast, the batched approach on GPU significantly improves the performance of GCNs application. Especially on Reaction100 dataset, which has more data and adopts larger batch size, the benefit of batched approach is prominent and the speedup of training compared to the non-batched approaches achieves 1.59x. We can maximize the benefit of the batched approach in the inference by setting larger batch size. The batched approach achieves up to 1.39x speedup of inference throughput.

Table 7.2: Training time of ChemGCN [sec]

	CPU	GPU		**Speedup**
	Non-Batched	Non-Batched	Batched	
Tox21	854.51	918.03	723.80	**1.18x**
Reaction100	16223.98	3029.13	1905.32	**1.59x**

Table 7.3: Inference time of ChemGCN [sec]

	CPU	GPU		**Speedup**
	Non-Batched	Non-Batched	Batched	
Tox21	2.71	2.56	1.97	**1.30x**
Reaction100	44.66	22.42	16.32	**1.37x**

Next, we analyzed the performance of each kernel by using Timeline, which is a performance profiling tool in TensorFlow. Figure 7.11 shows the visualization of each kernel execution in a graph convolutional layer for one mini batch on Tox21 dataset with the non-batched or batched approaches. Although the bars are illustrated with same length, their execution time are completely different.

Table 7.4 shows the "actual" execution time of each operation. As Figure 7.11 shows, the non-batched approach requires $batchsize * 3 = 150$ times of CUDA kernel launches while the batched approach requires only three times. The batched approach can largely reduce the overhead of CUDA kernel launches. The speedup of SpMM with Batched SpMM is about 10x. This performance improvement corresponds to the number showed in Figure 7.8-(a). Also, the batched approach significantly reduces the execution time of MatMul and Add as well as SpMM. This implies that the non-batched approach does not exploit the parallelism of GPU. We concluded that the batched approach can significantly increase the occupancy of GPU not only in SpMM operation but also in other kernels on small matrices.

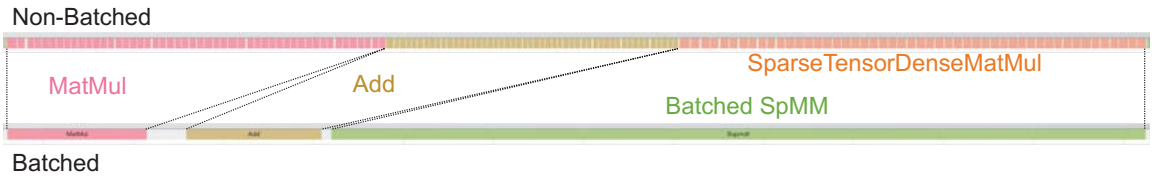


Figure 7.11: Visualization of the execution of ChemGCN with Timeline

Table 7.4: Execution time of each kernel in Figure 7.11 [μsec]

	Non-Batched	Batched
MatMul	1,571	31
Add	1,316	23
SpMM	1,981	190

7.6 Conclusion

In the application of machine learning approaches to various fields including chemistry and biology, GCNs show high accuracy. However, there are yet less work of GCNs in terms of throughput, and the current approach for GCNs, which require many operations on small sparse and dense matrices, cannot exploit the computing power of GPU. The scenario requiring small sparse matrix operations, especially SpMM, does not yet exist, and thus no research or engineering effort has been put into accelerating for such kind of operations. In order to accelerate GCNs applications, we propose the Batched SpMM, which efficiently handles tens or hundreds of SpMM operations together by single kernel. To exploit GPU architecture more, we devise new SpMM algorithm, Sub-Warp-Assigned SpMM, especially for small sparse matrices with cache blocking optimization to dense matrices. We evaluate the performance of our Batched SpMM on randomly generated matrices. The Batched SpMM achieves up to 9.27x speedup compared to the non-batched approaches, and 1.43x compared to Batched GEMM of cuBLAS.

Furthermore, the batched optimization improves the performance of GCNs application with 1.59x speedup for training and 1.37x speedup for inference.

Chapter 8

Conclusion

Many-core processors such as GPUs are becoming main-stream in various IT platforms with the growth of computing power and the scaling of supercomputer. Many-core processors accelerate many scientific and engineering applications. However, operations on sparse matrices for many-core processors are not sophisticated well, while many work is done for modern multi-core CPUs. Furthermore, the emerging area about Big Data and thriving Machine Learning area require other kinds of optimization strategies for sparse matrix kernels on many-core processors. Thus, rich compute power of many-core processors is not exploited, and there is still tremendous room of improvement in applications.

This thesis tackles two main issues with optimizing sparse matrix kernels on many-core processors. The first issue deals with memory access. The sparse matrix kernels usually have indirect memory accesses due to the compression with sparse matrix formats. This random memory accesses cause frequent cache misses. This is more critical on many-core processors, which have smaller cache capacity, compared to on multi-core CPUs. Also, sparse matrix kernels require much amount of bytes accessed since sparse matrix formats hold the indices data. Sparse matrix kernels often waste the compute power of many-core processors since the performance is limited by memory bandwidth. The second issue is concerned with load balancing. Since the pattern or number of non-zero elements in each row or column is different, non-sophisticated task assignment causes load imbalance. This load imbalance issue wastes the massive parallelism of many-core architectures. In addition, hierarchical load-balancing issue emerges when processing many kernels for sparse matrices in parallel.

To overcomes the memory access and load balancing issues of sparse matrix operations, we propose optimization strategies of sparse matrix kernels for many-core processors. Firstly, we propose the NUS and AMB formats for improving the performance of SpMV. By dividing the matrix along the column and processing each sub matrix one by one, the locality of memory access to input vector is largely improved. Our approach also reduces the amount of bytes accessed by compressing the indices

data of non-zero elements. Performance evaluation shows that proposed approaches, especially AMB format, achieve significant speedups of up to 1.4x compared to existing work. Secondly, we accelerate the performance of SpGEMM. We propose hash table-based SpGEMM algorithm with utilizing fast shared memory access with perfect load balancing. Our approach largely improves the performance of SpGEMM with up to over 10x speedup. The proposed optimization strategy for SpGEMM brings large performance improvement to Intel Xeon Phi. We built the guide for selecting the best algorithm for specific input and scenario based on empirical analysis and performance model. Thirdly, we propose a new computing routine, "Batched SpMM", which is designed mainly for GCN emerging in the machine learning field. Executing SpMM between small matrices often required in GCN application hardly exploits the parallelism, resulting in the bottleneck. The batched approaches applied to GCN application handle many SpMM operations by a single kernel, and the speedup in training is up to 1.59x.

This thesis provides several contributions to performance optimizations of sparse matrix kernels across a wide-area of computer science. This thesis implies that software approaches for various operations in high performance computing areas, not limited to sparse matrix operations, can exploit the compute power of many-core processors, and thus, especially data intensive applications attain abundance performance improvement.

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[2] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

[3] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*, 2016.

[4] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[5] Arnon Rungsawang and Bundit Manaskasemsak. Fast pagerank computation on a GPU cluster. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2012)*, pp. 450–456. IEEE, 2012.

[6] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers*, pp. 111–125. Springer, 2010.

[7] Francisco Vázquez, José-Jesús Fernández, and Ester M Garzón. A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience*, Vol. 23, No. 8, pp. 815–826, 2011.

[8] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 273–282. ACM, 2013.

[9] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. A multi-level optimization method for stencil computation on the domain that is bigger than memory capacity of gpu. In *IEEE*

27th International, Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) 2013, pp. 1080–1087. IEEE, 2013.

[10] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[11] Rajesh Nishtala, Richard W Vuduc, James W Demmel, and Katherine A Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, Vol. 18, No. 3, pp. 297–311, 2007.

[12] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, Vol. 35, No. 3, pp. 178–194, 2009.

[13] Benjamin C Lee, Richard W Vuduc, James W Demmel, and Katherine A Yelick. Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *ICPP 2004: International Conference on Parallel Processing 2004*, pp. 169–176. IEEE, 2004.

[14] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Perfomance models for blocked sparse matrix-vector multiplication kernels. In *ICPP'09: International Conference on Parallel Processing 2009.*, pp. 356–364. IEEE, 2009.

[15] Alexander Monakov and Arutyun Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp. 289–297. Springer, 2009.

[16] Wai Teng Tang, Wen Jun Tan, Rajarshi Ray, Yi Wen Wong, Weiguang Chen, Shyh-hao Kuo, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 26. ACM, 2013.

[17] Erik Saule, Kamer Kaya, and Umit V Catalyurek. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. *arXiv preprint arXiv:1302.1078*, 2013.

[18] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies. Technical report, IBM, 2008.

[19] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. Fast conjugate gradients with multiple GPUs. In *Computational Science–ICCS 2009*, pp. 893–903. Springer, 2009.

[20] Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. *Computer Science-Research and Development*, Vol. 25, No. 1-2, pp. 83–91, 2010.

[21] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A unified sparse matrix data format for modern processors with wide SIMD units. *CoRR*, Vol. abs/1307.6209, , 2013.

[22] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp. 18:1–18:11, New York, NY, USA, 2009. ACM.

[23] Satoshi Nakazato, Satoru Tagaya, Norihito Nakagomi, Takayuki Watai, and Akihiro Sawamura. Hardware Technology of the SX-9 (1). *NEC TECHNICAL JOURNAL, Special Issue: Supercomputer SX-9*, Vol. 3, No. 4, 2008.

[24] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, Vol. 38, No. 1, p. 1, 2011.

[25] Intel. Intel Math Kernel Library.

[26] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, Vol. 18, No. 1, pp. 135–158, 2004.

[27] Weizhi Xu, Hao Zhang, Shuai Jiao, Da Wang, Fenglong Song, and Zhiyong Liu. Optimizing sparse matrix vector multiplication using cache blocking method on fermi gpu. In *13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD) 2012*, pp. 231–235. IEEE, 2012.

[28] NVIDIA Corporation. cuSPARSE Library.

[29] S. Yan et al. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, 2014.

[30] Hoang-Vu Dang and Bertil Schmidt. CUDA-enabled Sparse Matrix–Vector Multiplication on GPUs using atomic operations. *Parallel Computing*, Vol. 39, No. 11, pp. 737 – 750, 2013.

[31] J.L. Greathouse and M. Daga. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 769–780, Nov 2014.

[32] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 781–792, Nov 2014.

[33] M. Daga and J. L. Greathouse. Structural agnostic spmv: Adapting csr-adaptive for irregular matrices. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pp. 64–74, Dec 2015.

[34] Y. Liu and B. Schmidt. Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pp. 82–89, July 2015.

[35] Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2, 1994.

[36] John R. Rice and Ronald F. Boisvert. *Solving Elliptic Problems Using ELLPACK*. Springer-Verlag New York, Inc., 1984.

[37] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pp. 273–282, New York, NY, USA, 2014. ACM.

[38] Yusuke. Nagasaka, Akira. Nukada, and Satochi. Matsuoka. Cache-aware sparse matrix formats for kepler gpu. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pp. 281–288, Dec 2014.

[39] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. Reducing vector i/o for faster gpu sparse matrix-vector multiplication. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pp. 1043–1052, May 2015.

[40] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing*, Vol. 36, No. 5, pp. C401–C423, 2014.

[41] Matteo Maggioni and Tanya Berger-Wolf. Coadell: Adaptivity and compression for improving sparse matrix-vector multiplication on gpus. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pp. 933–940. IEEE, 2014.

[42] Nathan Bell, Steven Dalton, and Luke N Olson. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM Journal on Scientific Computing*, Vol. 34, No. 4, pp. C123–C152, 2012.

[43] Stijn Van Dongen. Graph Clustering Via a Discrete Uncoupling Process. *SIAM Journal on Matrix Analysis and Applications*, Vol. 30, No. 1, pp. 121–141, 2008.

[44] Aydın Buluç and John R Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, 2011.

[45] Peter D. Sulatycke and Kanad Ghose. Caching-efficient multithreaded fast multiplication of sparse matrices. In *12th International Parallel Processing Symposium*, pp. 117–123. IEEE Computer Society, 1998.

[46] Kiran Matam, Siva Rama Krishna Bharadwaj Indarapu, and Kishore Kothapalli. Sparse Matrix-Matrix Multiplication on Modern Architectures. In *2012 19th International Conference on High Performance Computing (HiPC)*, pp. 1–10. IEEE, 2012.

[47] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Perfomance Models for Blocked Sparse Matrix-Vector Multiplication kernels. In *ICPP'09: International Conference on Parallel Processing*, pp. 356–364. IEEE, 2009.

[48] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaSpMV: Yet Another SpMV Framework on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pp. 107–118. ACM, 2014.

[49] Weifeng Liu and Brian Vinter. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 370–381, May 2014.

[50] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, Vol. 41, No. 4, p. 25, 2015.

[51] NVIDIA. NVIDIA CUDA Sparse Matrix Library (cuSPARSE).

[52] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations Ver.0.5.1, 2014.

[53] Julien Demouth. Sparse Matrix-Matrix Multiplication on the GPU. In *GPU Technology Conference*, 2012.

[54] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. *SIAM Journal on Scientific Computing*, Vol. 37, No. 1, pp. C54–C71, 2015.

[55] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication. In *Proceedings of the 2016 International Conference on Supercomputing*, Vol. 1 of *ICS '16*, pp. 36:1–36:12, New York, NY, USA, 2016. ACM.

[56] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. High performance graph algorithms from parallel sparse matrices. In *PARA*, pp. 260–269, 2007.

[57] Aydın Buluç and John R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *IJHPCA*, Vol. 25, No. 4, pp. 496–509, 2011.

[58] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. Hipmcl: a high-performance parallel implementation of the markov clustering algorithm for large-scale networks. *Nucleic acids research*, 2018.

[59] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, Vol. 76, No. 3, p. 036106, 2007.

[60] Viral B. Shah. *An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity*. PhD thesis, University of California, Santa Barbara, June 2007.

[61] Ariful Azad, Aydın Buluç, and John Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *IPDPSW*, 2015.

[62] Sandeep R Agrawal, Christopher M Dee, and Alvin R Lebeck. Exploiting accelerators for efficient high dimensional similarity search. In *PPoPP*. ACM, 2016.

[63] Guoming He, Haijun Feng, Cuiping Li, and Hong Chen. Parallel simrank computation on large graphs with iterative aggregation. In *SIGKDD*. ACM, 2010.

[64] Grey Ballard, Christopher Siefert, and Jonathan Hu. Reducing communication costs for sparse matrix multiplication within algebraic multigrid. *SIAM Journal on Scientific Computing*, Vol. 38, No. 3, pp. C203–C231, 2016.

[65] Johannes Sebastian Mueller-Roemer, Christian Altenhofen, and André Stork. Ternary sparse matrix representation for volumetric mesh subdivision and processing on GPUs. *Computer Graphics forum*, Vol. 36, No. 5, 2017.

[66] Nicolas Bock, Matt Challacombe, and Laxmikant V Kalé. Solvers for o(n) electronic structure in the strong scaling limit. *SIAM Journal on Scientific Computing*, Vol. 38, No. 1, pp. C1–C21, 2016.

[67] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM TOMS*, Vol. 4, No. 3, pp. 250–269, 1978.

[68] Ariful Azad, Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, Vol. 38, No. 6, pp. C624–C651, 2016.

[69] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *ICPP*, pp. 101–110. IEEE, 2017.

[70] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, Vol. 13, No. 1, pp. 333–356, 1992.

[71] Peter D Sulatycke and Kanad Ghose. Caching-efficient multithreaded fast multiplication of sparse matrices. In *IPPS/SPDP*. IEEE, 1998.

[72] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *IPDPSW*, pp. 693–702. IEEE, 2017.

[73] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *ISC*, pp. 48–57. Springer, 2015.

[74] Timothy A Davis. *Direct methods for sparse linear systems*. SIAM, 2006.

[75] Timothy A Davis. private communication.

[76] Karl Rupp, Philippe Tillet, Florian Rudolf, Josef Weinbub, Andreas Morhammer, Tibor Grasser, Ansgar Jüngel, and Siegfried Selberherr. Viennacl—linear algebra library for multi-and many-core architectures. *SIAM Journal on Scientific Computing*, Vol. 38, No. 5, pp. S412–S439, 2016.

[77] Kenneth A Ross. Efficient hash probes on modern processors. In *ICDE*, pp. 1297–1301. IEEE, 2007.

[78] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pp. 442–446. SIAM, 2004.

[79] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[80] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, Vol. 91, No. 2, pp. 201–213, 2002.

[81] Aydın Buluç and John R Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, Vol. 25, No. 4, pp. 496–509, 2011.

[82] Aydın Buluç, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. Design of the GraphBLAS API for C. In *IEEE Workshop on Graph Algorithm Building Blocks, IPDPSW*, 2017.

[83] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pp. 2224–2232, 2015.

[84] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[85] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, Vol. 30, No. 8, pp. 595–608, 2016.

[86] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

[87] Felix A Faber, Luke Hutchison, Bing Huang, Justin Gilmer, Samuel S Schoenholz, George E Dahl, Oriol Vinyals, Steven Kearnes, Patrick F Riley, and O Anatole von Lilienfeld. Machine learning prediction errors better than dft accuracy. *arXiv preprint arXiv:1702.05532*, 2017.

[88] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pp. 593–607. Springer, 2018.

[89] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. http://tensorflow.org/, 2015.

[90] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 66–79. ACM, 2018.

[91] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.*, pp. 1–11. IEEE, 2008.

[92] Carl Yang, Aydin Buluc, and John D Owens. Design principles for sparse matrix multiplication on the gpu. *arXiv preprint*, No. arXiv:1803.08601, 2018.

[93] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *SC '16 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 58. IEEE Press, 2016.

[94] Tingxing Dong, Azzam Haidar, Piotr Luszczek, James Austin Harris, Stanimire Tomov, and Jack Dongarra. Lu factorization of small matrices: Accelerating batched dgetrf on the gpu. In *HPCC/CSS/ICESS*, pp. 157–160, 2014.

[95] Azzam Haidar, Tingxing Tim Dong, Stanimire Tomov, Piotr Luszczek, and Jack Dongarra. A framework for batched and gpu-resident factorization algorithms applied to block householder transformations. In *International Conference on High Performance Computing*, pp. 31–47. Springer, 2015.

[96] Ran Zheng, Wei Wang, Hai Jin, Song Wu, Yong Chen, and Han Jiang. Gpu-based multifrontal optimizing method in sparse cholesky factorization. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 90–97. IEEE, 2015.

[97] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. Automating wavefront parallelization for sparse matrix computations. In *SC '16 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 41. IEEE Press, 2016.

[98] Xue Li, Fangxing Li, and Joshua M Clark. Exploration of multifrontal method with gpu in power flow computation. In *2013 IEEE Power and Energy Society General Meeting (PES)*, pp. 1–5. IEEE, 2013.

[99] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance, design, and autotuning of batched gemm for gpus. In *International Conference on High Performance Computing*, pp. 21–38. Springer, 2016.

[100] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *The International Journal of High Performance Computing Applications*, Vol. 24, No. 4, pp. 511–515, 2010.

[101] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Batched matrix computations on hardware accelerators based on gpus. *The International Journal of High Performance Computing Applications*, Vol. 29, No. 2, pp. 193–208, 2015.

[102] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joël Falcou, and Jack Dongarra. High-performance matrix-matrix multiplications of very small matrices. In *European Conference on Parallel Processing*, pp. 659–671. Springer, 2016.

[103] Hartwig Anzt, Gary Collins, Jack Dongarra, Goran Flegar, and Enrique S Quintana-Ortí. Flexible batched sparse matrix-vector product on gpus. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, p. 3. ACM, 2017.

[104] Deepchem. https://deepchem.io/.

[105] pfnet research. Chainer chemistry. https://github.com/pfnet-research/chainer-chemistry.

[106] National Center for Advancing Translational Sciences. Tox21 data challenge 2014. https://tripod.nih.gov/tox21/challenge/about.jsp.

[107] Reaxys. https://www.reaxys.com/.