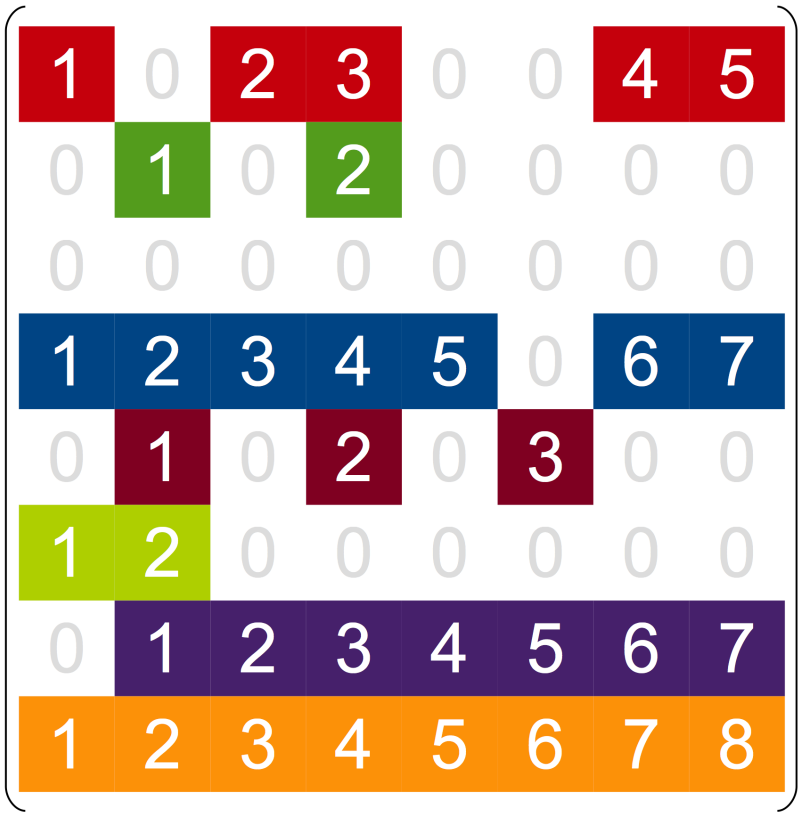
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PhD Thesis

Weifeng Liu

Parallel and Scalable Sparse Basic Linear Algebra Subprograms



Academic advisor: Brian Vinter

Submitted: 26/10/2015

To Huamin and Renheng

for enduring my absence while working on the PhD and the thesis

To my parents and parents-in-law

for always helping us to get through hard times

Acknowledgments

I am immensely grateful to my supervisor, Professor Brian Vinter, for his support, feedback and advice throughout my PhD years. This thesis would not have been possible without his encouragement and guidance. I would also like to thank Professor Anders Logg for giving me the opportunity to work with him at the Department of Mathematical Sciences of Chalmers University of Technology and University of Gothenburg.

I would like to thank my co-workers in the eScience Center who supported me in every respect during the completion of this thesis: James Avery, Jonas Bardino, Troels Blum, Klaus Birkelund Jensen, Mads Ruben Burgdorff Kristensen, Simon Andreas Frimann Lund, Martin Rehr, Kenneth Skovhede, and Yan Wang.

Special thanks goes to my PhD buddy James Avery, Huamin Ren (Aalborg University), and Wenliang Wang (Bank of China) for their insightful feedback on my manuscripts before being peer-reviewed.

I would like to thank Jianbin Fang (Delft University of Technology and National University of Defense Technology), Joseph L. Greathouse (AMD), Shuai Che (AMD), Ruipeng Li (University of Minnesota) and Anders Logg for their insights on my papers before being published. I thank Jianbin Fang, Klaus Birkelund Jensen, Hans Henrik Happe (University of Copenhagen) and Rune Kildetoft (University of Copenhagen) for access to the Intel Xeon and Xeon Phi machines. I thank Bo Shen (Inspur Group Co., Ltd.) for helpful discussion about Xeon Phi programming. I also thank Arash Ashari (The Ohio State University), Intel MKL team, Joseph L. Greathouse, Mayank Daga (AMD) and Felix Gremse (RWTH Aachen University) for sharing source code, libraries or implementation details of their SpMV and SpGEMM algorithms with me.

I would also like to thank our anonymous reviewers of conferences *SPAA '13*, *PPoPP '13*, *IPDPS '14*, *PPoPP '15* and *ICS '15*, of workshops *GPGPU-7* and *PMAA '14*, and of journals *Journal of Parallel and Distributed Computing* and *Parallel Computing* for their invaluable suggestions and comments on my manuscripts.

This research project was supported by the Danish National Advanced Technology Foundation (grant number J.nr. 004-2011-3).

Abstract

Sparse basic linear algebra subprograms (BLAS) are fundamental building blocks for numerous scientific computations and graph applications. Compared with Dense BLAS, parallelization of Sparse BLAS routines entails extra challenges due to the irregularity of sparse data structures. This thesis proposes new fundamental algorithms and data structures that accelerate Sparse BLAS routines on modern massively parallel processors: (1) a new heap data structure named *ad*-heap, for faster heap operations on heterogeneous processors, (2) a new sparse matrix representation named CSR5, for faster sparse matrix-vector multiplication (SpMV) on homogeneous processors such as CPUs, GPUs and Xeon Phi, (3) a new CSR-based SpMV algorithm for a variety of tightly coupled CPU-GPU heterogeneous processors, and (4) a new framework and associated algorithms for sparse matrix-matrix multiplication (SpGEMM) on GPUs and heterogeneous processors.

The thesis compares the proposed methods with state-of-the-art approaches on six homogeneous and five heterogeneous processors from Intel, AMD and nVidia. Using in total 38 sparse matrices as a benchmark suite, the experimental results show that the proposed methods obtain significant performance improvement over the best existing algorithms.

Resumé

Sparse lineær algebra biblioteket kaldet BLAS, er en grundlæggende byggesten i mange videnskabelige- og graf-applikationer. Sammenlignet med almindelige BLAS operationer, er paralleliseringen af sparse BLAS specielt udfordrende, grundet den manglende struktur i matricernes data. Denne afhandling præsenterer grundlæggende nye datastrukturer og algoritmer til hurtigere sparse BLAS afvikling på moderne, massivt parallelle, processorer: (1) en ny hob baseret datastruktur, kaldet ad-heap, for hurtigere hob operationer på heterogene processorer. (2) en ny repræsentation for sparse data, kaldet CSR5, der tillader hurtigere matrix-vektor multiplikation (SpMV) på homogene processorer som CPU'er, GPGPU'er og Xeon Phi, (3) en ny CSR baseret SpMV algoritme for en række, tæt forbundne, CPU-GPGPU heterogene-processorer, og (4) et nyt værktøj for sparse matrix-matrix multiplikationer (SpGEMM) on GPGPU's og heterogene processorer.

Afhandlingen sammenligner de præsenterede løsninger med state-of-the-art løsninger på seks homogene og fem heterogene processorer fra Intel, AMD, og nVidia. På baggrund af 38 sparse matricer som benchmarks, ses at de nye metoder der præsenteres i denne afhandling kan give signifikante forbedringer over de kendte løsninger.

Contents

1. Introduction

In the past few decades, basic linear algebra subprograms (BLAS) [24, 60, 61] have been gaining attention in many fields of scientific computing and simulation. When the inputs of BLAS routines are large and sparse, exploiting their sparsity can significantly reduce runtime and space complexity. In this case, a set of new routines have been designed and called Sparse BLAS [68, 70, 71, 64]. Because a large amount of real-world applications can obtain benefits from the exploitation of sparsity, designing parallel and scalable data structures and algorithms for Sparse BLAS became an important research area in the era of massively parallel processing.

This thesis presents the author's parallel and scalable data structures and algorithms for Sparse BLAS on modern multicore, manycore and heterogeneous processors. Specifically, four main contributions have been made: (1) a new heap data structure named *ad*-heap, for faster heap operations on heterogeneous processors, (2) a new sparse matrix representation named CSR5, for faster sparse matrix-vector multiplication (SpMV) on homogeneous processors such as CPUs, GPUs and Xeon Phi, (3) a new CSR-based SpMV algorithm for a variety of tightly coupled CPU-GPU heterogeneous processors, and (4) a new framework and associated algorithms for sparse matrix-matrix multiplication (SpGEMM) on GPUs and heterogeneous processors.

1.1 Organization of The Thesis

The first chapter, i.e., this chapter, gives an overview and introduces main contributions of the author of the thesis.

Part I, composed of three chapters, describes background of Sparse BLAS and parallelism in architectures and algorithms. Chapter 2 describes basic concepts about sparsity, where sparse data come from, how sparse data are decomposed, and what the main challenges of parallel processing of sparse data. Chapter 3 introduces four architectures of modern microprocessors: multicore CPUs, manycore GPUs, manycore coprocessors and CPUs, and tightly-coupled CPU-GPU heterogeneous processors. Chapter 4 describes useful data structure, i.e., *ad*-heap, and primitives, such as reduction, sorting network, prefix-sum scan, segmented sum, merging and k-selection, for parallel implementation of Sparse BLAS operations.

Part II, composed of two chapters, introduces representation of sparse matrices for modern computer systems. Chapter 5 introduces the basic and newly proposed storage formats for sparse matrices. Chapter 6 describes the CSR5 storage format, which is one of the main contributions of the author's PhD work.

Part III, composed of three chapters, gives the author's algorithms for Sparse BLAS routines on modern homogeneous and heterogeneous processors. Chapter 7 focuses on several methods for adding two sparse vectors, which is the main building block for the more complex SpGEMM operation. Chapter 8 describes the author's two SpMV algorithms for sparse matrices in the CSR format and in the CSR5 format. Chapter 9 describes the author's approach for SpGEMM, the most complex Sparse BLAS operation.

The last chapter concludes the thesis and suggests future work.

Appendix A lists detailed information of 38 sparse matrices used as benchmark suites in this thesis. Appendix B gives the used experimental platforms.

1.2 Author's Publications

In the period of the author's PhD research, he published the following five technical papers in the field of parallel computing as the leading author. This thesis is compiled from the following five technical papers:

1. Weifeng Liu, Brian Vinter. "Ad-heap: An Efficient Heap Data Structure for Asymmetric Multicore Processors". 7th Workshop on General Purpose Processing Using GPUs (held with ASPLOS '14) (GPGPU-7), 2014, pp. 54–63. DOI = http://doi.acm.org/10.1145/2576779.2576786. ISBN 978-1-4503-2766-4. (Reference [117])

2. Weifeng Liu, Brian Vinter. "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data". 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS '14), 2014, pp. 370–381. DOI = http://dx.doi.org/10.1109/IPDPS.2014.47. ISBN 978-1-4799-3800-1. (Reference [118])

3. Weifeng Liu, Brian Vinter. "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication". 29th ACM International Conference on Supercomputing (ICS '15), 2015, pp. 339–350. DOI = http://dx.doi.org/10.1145/2751205.2751209. ISBN 978-1-4503-3559-1. (Reference [120])

4. Weifeng Liu, Brian Vinter. "A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors". Journal of Parallel and Distributed Computing (JPDC). Volume 85, November 2015. pp. 47–61. DOI = http://dx.doi.org/10.1016/j.jpdc.2015.06.010. ISSN 0743-7315. (Reference [119])

5. Weifeng Liu, Brian Vinter. "Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors". Parallel Computing (PARCO). Volume 49, November 2015. pp. 179–193. DOI = http://dx.doi.org/10.1016/j.parco.2015.04.004. ISSN 0167-8191. (Reference [121])

Because the characteristic of this monograph, content of the above five published papers are distributed to Chapters 3, 4, 6, 7, 8 and 9. Specifically, Chapter 3 includes

contributions from papers 1, 4 and 5; Chapter 4 includes contributions from papers 1, 2 and 4; Chapter 6 includes contributions from paper 3; Chapter 7 includes contributions from papers 2 and 4; Chapter 8 includes contributions from papers 3 and 5; Chapter 9 includes contributions from papers 2 and 4.

Additionally, the author co-authored a technical paper in sparse representation for dictionary learning in machine vision:

6. Huamin Ren, Weifeng Liu, Søren Ingvor Olsen, Sergio Escalera, Thomas B. Moeslund. "Unsupervised Behavior-Specific Dictionary Learning for Abnormal Event Detection". 26th British Machine Vision Conference (BMVC '15), 2015. pp. 28.1–28.13. ISBN 1-901725-53-7. (Reference [152])

Part I

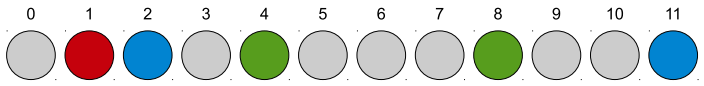
Foundations

2. Sparsity and Sparse BLAS

2.1 What Is Sparsity?

2.1.1 A Simple Example

Given a group of ordered small balls in colors gray, red, green and blue (shown in Figure 2.1(a)), we can create a one-dimensional array (shown in Figure 2.1(b)) to record their colors. In a computer system, the array can be accessed by integer indices, which are implicitly given. This thesis always uses 0-based indexing style (i.e., C style) but not 1-based (i.e., Fortran style). For example, it is easy to locate the 5th entry of the array at index 4, and to find the 5th ball is green.



(a) A group of ordered balls.

gray	red	blue	gray	green	gray	gray	gray	green	gray	gray	blue

(b) An one-dimensional array containing colors of the balls.

Figure 2.1: A group of ordered balls of different colors and their representation.

Now assume that the gray balls are unimportant in a certain scenario and we want to save memory space of the computer system, so that only red, green and blue balls need to be stored. The expected storage form is the array shown in Figure 2.2. However, if low-level details of modern computer storage systems and complex data structures such as linked list are not considered here, typical computer memory are organized in linear order thus cannot "label" unused entries in a hollow data structure. In other words, by using the single array, all "gray" entries still occupy memory space to guarantee correct indexing.

To avoid wasting memory space for the gray balls, a one-dimensional integer array can be introduced to store indices of the red, green and blue balls. We can see that the indices are implicitly given in the first example, but in the second, they are explicitly

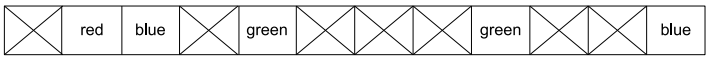


Figure 2.2: The expected storage fashion when gray balls are not important.

stored. Therefore, all "gray" entries can be removed from the original color array, and we still know the originally positions of the other entries. Figure 2.3 gives the new storage form composed of two arrays rather than one. We can search for index 4 (as input) in the index array to find the 5th ball is at the position 2 (as output). In other words, we now know that position 2 of the index array stores index 4. Afterwards, it is easy to find the position 2 (as new input) of the color array is green (as new output). Similarly, when we search index 6 in the index array and find it is not in the array, we can say that the 7th ball is gray.

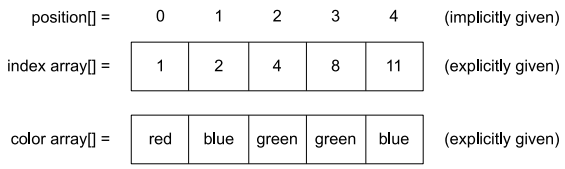


Figure 2.3: Red, green and blue entries and their indices are stored in two arrays.

By using the above representation, the input data are actually "compressed". Given an input array composed of n entries and each entry occupies α storage units, the array needs αn storage units in total. Assume it contains p "important" entries and $n-p$ "unimportant" entries, the space cost of storing the array in the compressed fashion is $(\alpha+\beta)p$, where β is the storage cost of the index of an entry. Because α and β are constant factors, the compressed storage format is much more space-efficient than the ordinary format when n is very large and $p \ll n$. Furthermore, computational cost can also be saved if only "important" entries are involved in a procedure.

We say that a dataset is **sparse** if many entries of the set are "unimportant", and the dataset can be represented in a compressed style to avoid space and time cost of processing its "unimportant" entries. In contrast, we say that a dataset is **dense** or **full** if all entries are seen as "important" and stored explicitly.

2.1.2 Sparse Matrices

The above example of small balls can be seen as dealing with a sparse vector. It is actually a simple matter to extend its concept to matrices and tensors of arbitrary ranks[1] [14, 10, 11, 113]. Given a matrix A of dimension $m \times n$, we say that A is **sparse**

[1]Because this thesis concentrates on Sparse BLAS, we leave sparse tensor computations as a future work.

if it contains many zeros and is stored in a compressed fashion, or is **dense** of **full** if all entries of the matrix are explicitly stored [65].

In matrix computations, zeros are seen as "unimportant", since a zero is an identity element (or a neutral element) for addition and an absorbing element for multiplication. That is to say, it leaves other entries unchanged when added a zero with them, and obtains zero itself when multiplied any entries with a zero.

To demonstrate why zeros are not important in matrix computations, we use an example that multiplies a dense matrix A of size 6×3

$$A = \begin{bmatrix} a & g & m \\ b & h & n \\ c & i & o \\ d & j & p \\ e & k & q \\ f & l & r \end{bmatrix} = \begin{bmatrix} a_{*,0} & a_{*,1} & a_{*,2} \end{bmatrix}$$

composed of three column vectors

$$a_{*,0} = \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \end{bmatrix}, a_{*,1} = \begin{bmatrix} g \\ h \\ i \\ j \\ k \\ l \end{bmatrix}, a_{*,2} = \begin{bmatrix} m \\ n \\ o \\ p \\ q \\ r \end{bmatrix}$$

with a sparse vector x of size 3

$$x = \begin{bmatrix} 2, 0, 0 \end{bmatrix}^T.$$

We obtain a resulting vector y of size 6

$$\begin{aligned} y &= Ax \\ &= x_0 a_{*,0} + x_1 a_{*,1} + x_2 a_{*,2} \\ &= 2a_{*,0} + 0a_{*,1} + 0a_{*,2} \\ &= 2 \begin{bmatrix} a, b, c, d, e, f \end{bmatrix}^T + 0 \begin{bmatrix} g, h, i, j, k, l \end{bmatrix}^T + 0 \begin{bmatrix} m, n, o, p, q, r \end{bmatrix}^T \\ &= \begin{bmatrix} 2a, 2b, 2c, 2d, 2e, 2f \end{bmatrix}^T + \begin{bmatrix} 0, 0, 0, 0, 0, 0 \end{bmatrix}^T + \begin{bmatrix} 0, 0, 0, 0, 0, 0 \end{bmatrix}^T \\ &= \begin{bmatrix} 2a, 2b, 2c, 2d, 2e, 2f \end{bmatrix}^T. \end{aligned}$$

We can see that the 2nd and the 3rd entries of x are zeros thus do not contribute to the final resulting vector y (recall multiplication of zero and any entry gives zero and addition of zero and any entry gives the entry itself). In other words, if it is known in advance that only the 1st entry of x is nonzero, a simpler and faster computation

$$\begin{aligned} y &= Ax \\ &= x_0 a_{*,0} \\ &= 2a_{*,0} \\ &= 2 \begin{bmatrix} a, b, c, d, e, f \end{bmatrix}^T \\ &= \begin{bmatrix} 2a, 2b, 2c, 2d, 2e, 2f \end{bmatrix}^T \end{aligned}$$

is enough to give the expected result.

In real-world applications, it is very common to see that a sparse matrix contains a large amount of zeros and very small proportion of nonzero entries. Appendix A lists the benchmark suite of this thesis, which are selected from the University of Florida Sparse Matrix Collection [56] that contains over 2700 sparse matrices from a variety of real-world applications. We can calculate the ratio of the number of nonzeros to the number of entries in the full matrix, and can see that the matrix *Protein* has 0.33% nonzero entry ratio, which is the densest of the 38 benchmark matrices. The lowest ratio is merely 0.00018%, from the matrix *Circuit5M*. Therefore, it is important to exploit the zeros of a sparse matrix to save memory space and corresponding computations.

2.2 Where Are Sparse Matrices From?

Many real-world applications generate sparse matrices. This section lists three typical scenarios from finite element methods, social networks and sparse representation of signals.

2.2.1 Finite Element Methods

The finite element method (FEM) [124] is used for approximating a partial differential equation over a large domain by connecting many simple element equations over many small subdomains (called finite elements). Figure 2.4 shows a simple 4-vertex mesh composed of two elements.

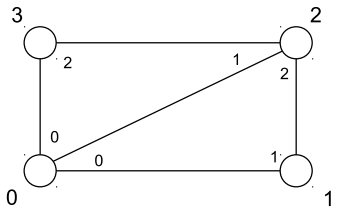


Figure 2.4: A simple mesh with 2 elements and 4 vertices. The numbers outside the mesh are indices of the four vertices. The numbers inside the mesh are local indices of vertices of each element.

Because each element has 3 vertices, we say that the local degrees of freedom (DoF) per element is 3. In this case, each element generates a local dense matrix of size 3×3, and contributes entries of the local dense matrix to a global sparse matrix of 4×4 (because there are 4 vertices, i.e., total number of DoFs in the mesh). The procedure is called finite element assembly [2].

Assume that the local matrix of the first element is

$$E_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \end{array},$$

where the three entries in the column vector $[0, 1, 2]^T$ to the left of the matrix will be row indices in the global matrix, and the three entries in the row vector $[0, 1, 2]$ on the top of the matrix will be column indices in the global matrix. Thus each entry in the matrix knows where the position to write its value is. For example, value g will be stored to location (0, 2) in the global matrix. After adding all entries, we obtain a sparse global matrix

$$A = \begin{bmatrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

Similarly, we can form a local matrix for the second element:

$$E_1 = \begin{array}{c} \\ 0 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 2 & 3 \end{array} \\ \begin{bmatrix} j & m & p \\ k & n & q \\ l & o & r \end{bmatrix} \end{array},$$

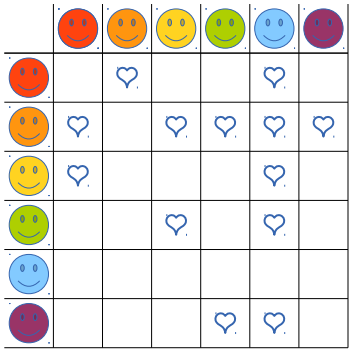
where the three entries in the column vector $[0, 2, 3]^T$ to the left of the matrix will be row indices in the global matrix, and the three entries in the row vector $[0, 2, 3]$ on the top of the matrix will be column indices in the global matrix. By adding all entries of this local matrix to the obtained global sparse matrix, we update the 4x4 sparse matrix to

$$A = \begin{bmatrix} a & d & g+m & p \\ b & e & h & 0 \\ c+k & f & i+n & q \\ l & 0 & o & r \end{bmatrix}.$$

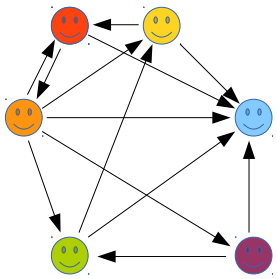
The sparse matrix can have much more zeros thus more sparse when the domain is divided into more elements. Appendix A collects some matrices generated from FEM applications.

2.2.2 Social Networks

A social networks is composed of a set of social actors (e.g., individuals or organizations) and a set of dyadic ties (i.e., groups of two social actors) between them. If each actor is seen as a vertex of a graph and each dyad is seen as an edge connecting two vertices (i.e., social actors), a social network can be represented by a graph. Figure 2.5(a) shows a small and simple social network of 6 individuals and the dyadic ties, and each individual labels his/her preference in a list as a row. We can see that the 1st individual (red) likes the 2nd and the 5th individuals (orange and blue, respectively). The 2nd individual like all the others in the network, and the 5th



(a) Social network including 6 individuals.



(b) Graph representation of the social network. An arrow means "gives like to".

Figure 2.5: A small and simple social network and its graph representation.

individual does not like anybody but is liked by all the others. This social network can be represented by the graph shown in Figure 2.5(b).

Actually, we can directly extract a sparse matrix

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

from Figure 2.5(a) and recognize the equivalence between the sparse matrix A and the graph in Figure 2.5(b). When the number of social actors in a network increases, the sparse matrix obtained from it is more and more sparse. The reason is that generally each individuals friend circle is only a very small proportion of the whole network.

2.2.3 Sparse Representation of Signals

In a vector space, a set of linearly independent vectors is called a basis, if every vector in the space is a linear combination of this set. The dimensions of the space is the number of basis vectors, and is the same for every basis. For instance, a six-dimensional vector space's basis always contains 6 vectors. An example of such a basis is the canonical basis:

$$v_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, v_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, v_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, v_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, v_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, v_5 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Any given vector, for example vector

$$x = \begin{bmatrix}2, 2, 3, 3, 5, 5\end{bmatrix}^T$$

in this space is a linear combination of the basis vectors. Assume that the basis vectors are columns of a dense matrix

$$B = \begin{bmatrix}v0, v1, v2, v3, v4, v5\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 0 & 1\end{bmatrix}.$$

The vector x can be represented by multiplying B with a dense vector

$$a = \begin{bmatrix}2, 2, 3, 3, 5, 5\end{bmatrix}^T.$$

In other words,

$$x = \begin{bmatrix}2\\2\\3\\3\\5\\5\end{bmatrix} = Ba = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}2\\2\\3\\3\\5\\5\end{bmatrix}.$$

If we want to construct the vector x with a sparse vector rather than the dense vector a, we can introduce another dense matrix

$$D = \begin{bmatrix}v0, v1, v2, v3, v4, v5, v6, v7, v8\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1\end{bmatrix},$$

that contains a set of linear dependent vectors (e.g., $v6$ is a linear combination of the first 6 column vectors $v0$ – $v5$) as its columns. To obtain x, we actually can multiply D with a sparse vector

$$a' = \begin{bmatrix}0, 0, 0, 0, 0, 0, 2, 3, 5\end{bmatrix}^T.$$

That is

$$x = \begin{bmatrix}2\\2\\3\\3\\5\\5\end{bmatrix} = Da' = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}0\\0\\0\\0\\0\\2\\3\\5\end{bmatrix}.$$

We can call dense vector x a signal, and can see that even though D required larger memory space than B, sparse vector a' occupies less space than a. When we need to calculate or to store a large amount of signals, using sparse vectors like a' is more space-efficient, when D keeps unchanged for all signals. In this case, a large amount of sparse vectors construct a sparse matrix A. By multiplying D with A, a group of signals are reconstructed. This technique is widely-used in applications such as signal processing and audio/image analysis [153, 152]. The above matrix D is called a dictionary (or overcomplete dictionary), and the sparse vector is called sparse approximation or sparse representation of a signal.

2.3 What Are Sparse BLAS?

Basic Linear Algebra Subprograms, or BLAS for short, are a set of building blocks for a wide range of software. The definitions of interfaces in BLAS help keeping software portable and maintainable [61, 60, 24]. Vector and matrix are two basic elements in BLAS. According to possible combinations among them, BLAS can be grouped into three levels: (1) vector operations, (2) matrix-vector operations, and (3) matrix-matrix operations.

Consider sparsity of an input vector or matrix, Sparse BLAS [64, 68, 70, 71] are defined by using similar interfaces of BLAS for dense inputs. Sparse BLAS have more functions than Dense BLAS since they consider both dense and sparse data as inputs and outputs. Sparse BLAS are in general more complex because of indirect memory accesses and keeping output as sparse whenever possible. This section introduces main operations of Sparse BLAS. Later on, the main challenges of implementing fast and scalable Sparse BLAS will be discussed in Section 2.5.

In this section, subscripts s and d are used for indicating whether a vector or a matrix is sparse or dense.

2.3.1 Level 1: Sparse Vector Operations

Level 1 Sparse BLAS include dot product and adding of one vector to another. The output of a dot product operation is a scalar, regardless the input vectors are sparse or dense. The addition of two vectors generates another vector. If any one of the inputs is dense, the resulting vector is dense. If both inputs are sparse, the output is sparse as well.

Dense Vector – Sparse Vector Dot Product

The operation is performed by

$$dot \leftarrow x_d^T y_s,$$

or

$$dot \leftarrow x_s^T y_d,$$

where one of the input vectors x and y is sparse, the other is dense, and the output dot is a scalar.

This operation is basically a *gather* operation. All nonzero entries of the sparse vector are multiplied with entries of the same indices in the dense vector. Then a sum of the results of the multiplication is the output of the dot product operation.

Sparse Vector – Sparse Vector Dot Product

The operation is performed by

$$dot \leftarrow x_s^T y_s,$$

where input vectors x and y are sparse, and the output dot is a scalar.

In this operation, a nonzero entry in an input sparse vector does not contribute to the dot product result if its index is not found in the nonzero entry list of the other input sparse vector. So dot product of two sparse vectors can be converted to a *set intersection* problem. That is, only the nonzero entries having the same index in both inputs are multiplied. Then the sum of the separate results of the multiplication is the output of dot product.

Dense Vector – Sparse Vector Addition

The operation is performed by

$$z_d \leftarrow x_d + y_s,$$

or

$$z_d \leftarrow x_s + y_d,$$

where one of the input vectors x and y is sparse, the other is dense, and the output z is a dense vector.

Because the indices of the sparse vector is known, the addition is actually a *scatter* operation. In other words, the nonzero entries of the sparse vector are added with entries of the same indices in the dense vector, the results are written to the corresponding positions of the output vector. All the other entries at the indices not found in the sparse vector will directly copy the corresponding values from the dense input.

Sparse Vector – Sparse Vector Addition

The operation is performed by

$$z_s \leftarrow x_s + y_s,$$

where input and output vectors x, y and z are all sparse.

This operation is more complex than the above operations, since all of the three vectors are sparse. From the point view of set operations, nonzero entries in the *intersection* of the two inputs are added and stored to the output vector. The nonzero entries in the *symmetric difference* (i.e., the *union* of two *relative complements* of one input in the other) of x and y are directly saved to z. Chapter 7 describes the developed methods for this operation.

2.3.2 Level 2: Sparse Matrix-Vector Operations

Level 2 Sparse BLAS include sparse matrix and vector operations. When we consider sparsity both from the matrix and from the vector, Level 2 Sparse BLAS contain three possible operations: (1) multiplication of dense matrix and sparse vector, (2) multiplication of sparse matrix and dense vector, and (3) multiplication of sparse matrix and sparse vector. All the three operations have their own application scenarios.

Dense Matrix – Sparse Vector Multiplication

The operation is performed by

$$y_d \leftarrow A_d x_s,$$

where the input A is a dense matrix, the input x is a sparse vector, and the output y is a dense vector.

Two approaches can be used for this operation. Multiplying column vectors in the dense matrix with corresponding nonzero entries of the sparse vector, and adding the resulting vectors together to construct the dense resulting vector. One example of this method is shown in Section 2.2.3. Another method is to compute dot products of each row of A (as a dense vector) and x by using the *dense vector – sparse vector dot product* operation in the Level 1 Sparse BLAS, and to store the scalar outputs as entries of the resulting vector y.

Sparse Matrix – Dense Vector Multiplication

The operation is performed by

$$y_d \leftarrow A_s x_d,$$

where the input A is a sparse matrix, and the input vector x and the output vector y are all dense.

This operation can also be conducted by multiple *dense vector – sparse vector dot product* operations in the Level 1 Sparse BLAS. Then the generated dot products are entries of y. Because many real-world applications (as shown in Section 2.2) generate sparse matrices, this operation has probably been the mostly studied Sparse BLAS routine over the past few decades. This thesis proposes two fast algorithms for this operation described in Chapter 8. In the rest of this thesis, this operation is called SpMV for short.

Sparse Matrix – Sparse Vector Multiplication

The operation is performed by

$$y_s \leftarrow A_s x_s,$$

where the input matrix A and vector x and the output vector y are all sparse.

The *sparse vector – sparse vector dot product* in Level 1 Sparse BLAS can be used as a building block for this operation. Similarly to the above mentioned two Level 2 operations, outputs of dot products construct the resulting vector y. If A is organized by columns, an approach similar to the first method of *dense matrix – sparse vector Multiplication* can be used: multiplying sparse column vectors of A with corresponding

nonzero entries of x and adding the resulting sparse vectors together to obtain the sparse vector y. In this case, A *sparse vector – sparse vector addition* operation is also utilized.

2.3.3 Level 3: Sparse Matrix-Matrix Operations

There are three combinations of matrix-matrix multiplication: (1) multiplication of dense matrix and sparse matrix, (2) multiplication of sparse matrix and dense matrix, and (3) multiplication of sparse matrix and sparse matrix. Besides the three multiplication operations, two matrices, sparse or dense, can be added up, if they have the same dimensions.

Sparse Matrix Transposition

The operation is performed by

$$B_s \leftarrow A_s^T,$$

where the input matrix A and the output matrix B are both sparse.

This operation transpose a sparse matrix to another sparse matrix. All nonzero entries of the input are relocated to new positions in the output, if nonzero entries of the same row/column are required to be organized contiguously.

Dense Matrix – Sparse Matrix Multiplication

The operation is performed by

$$C_d \leftarrow A_d B_s,$$

where the input matrices A and B are dense and sparse, respectively. The resulting matrix C is dense.

The *dense vector – sparse vector dot product* in Level 1 Sparse BLAS can be used for calculating each entry in the resulting matrix. However, to achieve higher performance, this operation can also be seen as multiplication of the dense matrix A and multiple sparse column vectors of B. Thus the *dense matrix – sparse vector multiplication* in Level 2 Sparse BLAS can be used as a building block. In this case, each resulting vector of the Level 2 operation is a column of C.

Sparse Matrix – Dense Matrix Multiplication

The operation is performed by

$$C_d \leftarrow A_s B_d,$$

where the input matrices A and B are sparse and dense, respectively. The resulting matrix C is dense.

This operation looks similar to the previous one, since they can both use *dense vector – sparse vector dot product* in Level 1 Sparse BLAS as a building block. This operation can also be seen as multiplying a sparse matrix A with multiple dense column vectors of B, thus *sparse matrix – dense vector multiplication* can be used. But a better method may be to multiply each nonzero entry of a sparse row in A with corresponding dense row of B, and to add all resulting dense vectors together as a dense row of C.

Sparse Matrix – Sparse Matrix Multiplication

The operation is performed by

$$C_s \leftarrow A_s B_s,$$

where the all matrices A, B and C are sparse.

This is the most complex operation among the Level 1, 2 and 3 Sparse BLAS since the two inputs and the output are all sparse. Even though *sparse vector – sparse vector dot product* in Level 1 Sparse BLAS can be used for calculating each entry in C, its sparsity will be broken since it can be explicitly stored zero. To keep C sparse, a method similar to computing *sparse matrix – dense matrix multiplication* (i.e., exploiting rows of B) can be used. In this case, corresponding rows of B are scaled by nonzero entries of a sparse row of A, and are added up by the operation *sparse vector – sparse vector addition* in Level 1 Sparse BLAS. This thesis proposes a new algorithm for SpGEMM described in Chapter 9. In this thesis, the operation is called SpGEMM for short.

Dense Matrix – Sparse Matrix Addition

The operation is performed by

$$C_d \leftarrow A_d + B_s,$$

or

$$C_d \leftarrow A_s + B_d,$$

where one of the two input matrices A and B is dense and the other is sparse, and the resulting matrix C is dense.

This operation is similar to the *dense vector – sparse vector addition* operation in Level 1 Sparse BLAS, except multiple rows/columns are considered.

Sparse Matrix – Sparse Matrix Addition

The operation is performed by

$$C_s \leftarrow A_s + B_s,$$

where the all matrices A, B and C are sparse.

This operation can use *sparse vector – sparse vector addition* as a building block for each row or column.

2.4 Where Does Parallelism Come From?

Because sparse matrices from real-world applications can be very large, parallel sparse matrix algorithms have obtained a lot of attention. To expose parallelism, a fork-join model is used: a sparse matrix need to be divided into multiple submatrices, and each of them is assigned to one compute unit in a computer system. This stage is called “fork”. After all processes complete, the separate subresults of submatrices are required to be merged for one single output. This stage is called “join”.

2.4.1 Fork: From Matrix to Submatrix

There are three main methods that can decompos a sparse matrix into multiple submatrices. The first method is by row/column. We can partition a 6×6 sparse matrix

$$A = \begin{bmatrix} a & d & g & 0 & 0 & 0 \\ b & e & h & 0 & 0 & 0 \\ c & f & i & 0 & 0 & 0 \\ 0 & 0 & 0 & j & 0 & 0 \\ 0 & 0 & 0 & 0 & k & m \\ 0 & 0 & 0 & 0 & l & n \end{bmatrix}$$

into a sum of three submatrices

$$A_0 = \begin{bmatrix} a & d & g & 0 & 0 & 0 \\ b & e & h & 0 & 0 & 0 \\ c & f & i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, A_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, A_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k & m \\ 0 & 0 & 0 & 0 & l & n \end{bmatrix},$$

where

$$A = A_0 + A_1 + A_2.$$

The second method is by sparsity structure. We can divide the above sparse matrix A into a diagonal matrix A'_0 and another matrix A'_1 composed of the remaining entries. Then we have

$$A'_0 = \begin{bmatrix} a & 0 & 0 & 0 & 0 & 0 \\ 0 & e & 0 & 0 & 0 & 0 \\ 0 & 0 & i & 0 & 0 & 0 \\ 0 & 0 & 0 & j & 0 & 0 \\ 0 & 0 & 0 & 0 & k & 0 \\ 0 & 0 & 0 & 0 & 0 & m \end{bmatrix}, A'_1 = \begin{bmatrix} 0 & d & g & 0 & 0 & 0 \\ b & 0 & h & 0 & 0 & 0 \\ c & f & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & m \\ 0 & 0 & 0 & 0 & l & 0 \end{bmatrix},$$

where

$$A = A'_0 + A'_1.$$

The third method is by the number of nonzero entries. We can evenly divide A, which contains 14 nonzero entries, into two submatrices, A''_0 and A''_1, that contain 7 nonzero entries each. Then we have

$$A''_0 = \begin{bmatrix} a & d & g & 0 & 0 & 0 \\ b & e & h & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, A''_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & f & i & 0 & 0 & 0 \\ 0 & 0 & 0 & j & 0 & 0 \\ 0 & 0 & 0 & 0 & k & m \\ 0 & 0 & 0 & 0 & l & n \end{bmatrix},$$

where

$$A = A''_0 + A''_1.$$

In practice, selecting decomposition method depends on the target Sparse BLAS algorithm and architecture of the used computer system. Because of the compressed storage methods, the above decomposition methods in general do not bring obvious extra memory footprint regardless which method is selected.

2.4.2 Join: From Subresult to Result

Using the first method as an example, if we want to compute one of the level 2 BLAS routines that multiply A with a dense vector

$$x = \begin{bmatrix}1, 2, 3, 4, 5, 6\end{bmatrix}^T.$$

We can compute

$$y_0 = A_0 x = \begin{bmatrix} a & d & g & 0 & 0 & 0 \\ b & e & h & 0 & 0 & 0 \\ c & f & i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1a + 2d + 3g \\ 1b + 2e + 3h \\ 1c + 2f + 3i \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

$$y_1 = A_1 x = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4j \\ 0 \\ 0 \end{bmatrix},$$

$$y_2 = A_2 x = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & k & m \\ 0 & 0 & 0 & 0 & l & n \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 5k + 6m \\ 5l + 6n \end{bmatrix}.$$

We can see that y_0, y_1 and y_2 can be computed in parallel. Thereafter, by summing

them up, we obtain

$$
\begin{aligned}
y &= y_0 + y_1 + y_2 \\
&= \begin{bmatrix} 1a + 2d + 3g \\ 1b + 2e + 3h \\ 1c + 2f + 3i \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4j \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 5k + 6m \\ 5l + 6n \end{bmatrix} \\
&= \begin{bmatrix} 1a + 2d + 3g \\ 1b + 2e + 3h \\ 1c + 2f + 3i \\ 4j \\ 5k + 6m \\ 5l + 6n \end{bmatrix}.
\end{aligned}
$$

2.5 Challenges of Parallel and Scalable Sparse BLAS

2.5.1 Indirect Addressing

Because of the compressed storage fashion, nonzero entries of a sparse vector or matrix have to be accessed by indirect addresses stored in its index array. On one hand, indirect addressing leads to random accesses that bring more memory transactions and lower cache hit rate. On the other, the addresses are only known at runtime but not at compile time when software prefetching may be added to improve performance. Thus compared to sequential memory accesses in Dense BLAS, indirect addressing in general significantly decreases performance of Sparse BLAS.

Some hardware features can improve performance of indirect addressing. For example, modern CPUs are equipped with relatively large private and shared caches. As a result, cache hit rate is increased because more possibly usable data are stored in on-chip memory for latency hiding. In other words, each random memory transaction requires on average less time because of reduced long latency accessing on off-chip memory. Additionally, modern GPUs concurrently run a large amount (e.g., tens of thousands) of light-weight threads on high bandwidth memory to hide latency. Moreover, Improving sparse matrix storage structures, e.g., slicing and grouping column entries [5, 8, 54, 135], can greatly reduce negative affect of indirect addressing by caching more recently usable data.

Therefore, the main target of this thesis is utilizing high throughput of manycore processors such as GPUs and Xeon Phi for Sparse BLAS. All main contributions on data structures and algorithms of this thesis have been designed for them.

2.5.2 Selection of Basic Primitives

When the output is required to be sparse, some computations must be performed on indirect addresses. For example, *sparse vector – sparse vector dot product* in Level 1 Sparse BLAS executes a *set intersection* operation on the index arrays of two sparse

input vectors. This is obviously slower than a scatter or gather operation when one operand is dense, and can be much slower than trivial dot product of two dense vectors.

Therefore, selection of the best building blocks plays a crucial role in the implementation of Sparse BLAS. Many basic primitives such as sorting, searching, insertion, merging, reduction and prefix-sum scan all have their own application scenarios when using different hardware platforms. Chapter 4 of this thesis describes useful parallel primitives such as sorting network, evaluates multiple parallel merge algorithms, and develops new parallel primitives such as fast segmented sum. We further show that the contributions of Chapter 4 greatly improve performance of Sparse BLAS described in Chapters 7 – 9.

2.5.3 Data Decomposition, Load Balancing and Scalability

As listed in Section 2.4, multiple data decomposition methods can be used for generating submatrices for parallel processing on multiple compute units. However, which decomposition method is the best depends on required operation and concrete hardware device. Load balancing is actually the most important criterion of data decomposition, and decides scalability of a data structure or algorithm.

To achieve load balancing, three factors are required to be considered: (1) is data composition based on a sparse matrix's rows/columns, or sparsity structures, or nonzero entries? (2) is data composition conducted in input space or output space? and (3) which merging approach is the best if a decomposition method is decided?

Because various massively parallel chips are the main platform of this thesis, we offers multiple contributions. For example, Chapter 8 demonstrates that partitioning a sparse matrix into 2D tiles of the same size provides the best scalability for SpMV operation, and Chapter 9 shows that data composition conducted in output space delivers the best performance for SpGEMM.

2.5.4 Sparse Output of Unknown Size

Several Sparse BLAS operations, such as addition of two sparse vectors/matrices and multiplication of two sparse matrices, generate sparse output. The number of nonzero entries and their distribution are actually not known in advance. Because sparse matrices are normally very large, it is infeasible to store resulting nonzero entries to a dense matrix then to convert it to sparse.

Precomputing an upper bound is one method to approximate the number of nonzero entries of the output. For example, addition of a sparse vector v_0 including nnz_0 nonzero entries and another sparse vector v_1 containing nnz_1 nonzero entries will generate a sparse vector including no more than $nnz_0 + nnz_1$ nonzero entries. Thus preallocating a memory block of size $nnz_0 + nnz_1$ is a safe choice. However, this method may waste memory space when the intersection of the index sets is large. Another method is to preallocate a sparse resulting vector including only nnz_0 nonzero entries then insert nnz_1 nonzero entries from $v1$. But this method may need memory reallocation, which is not supported on some accelerator platforms such as GPUs.

In Chapter 9 of this thesis, we propose a hybrid memory management method that pre-allocates upper bound size for short rows and a fixed size for long rows, and re-allocates memory for the long rows on-demand. The experimental results show that this hybrid method can significantly save memory space (e.g., by a factor of 20) for some test problems. Further, on tightly coupled CPU-GPU processors, using re-allocatable memory bring extra performance gain.

3. Parallelism in Architectures

3.1 Overview

The performance of Sparse BLAS operations highly depend on the used hardware. In the past decade, a variety of chip multiprocessors (CMPs) have replaced single-core processors as the main targets of shared memory Sparse BLAS algorithm design. This Chapter introduces four mainstream CMPs: multicore CPU, manycore GPU, many-core coprocessor and CPU, and emerging tightly coupled CPU-GPU heterogeneous processor.

3.2 Multicore CPU

Central processing units (CPUs) are designed as the central control units that perform basic compute and control operations of computer systems. Because the CPU is responsible for processing and responding, its single-thread performance is highly emphasized. Inside one single CPU core, various parallelism oriented techniques, such as multiple functional units, pipelining, superscalar, out-of-order execution, branch prediction, simultaneous multithreading and deep cache hierarchies, have been designed. Because of the so-called power wall, clock frequency of a single GPU core cannot be further increased. As a result, multiple CPU cores of lower clock frequency have to be combined onto a single chip for continued performance improvements. Figure 3.1 shows a quad-core CPU equipped with large private and shared caches.

Because of the low-cost of commercial grade CPUs, they are widely used as main compute units of large-scale supercomputers. However, for the same reason, CPUs cannot achieve a performance balance between general purpose operations such as human-computer interaction and specific purpose applications such as scientific computing. As a result, more special purpose devices are required for massively parallel scientific computing.

3.3 Manycore GPU

Graphics processing units (GPUs), also known as graphics cards having a long history, are originally designed by nVidia in 1999 for photorealistic 3D rendering in computer

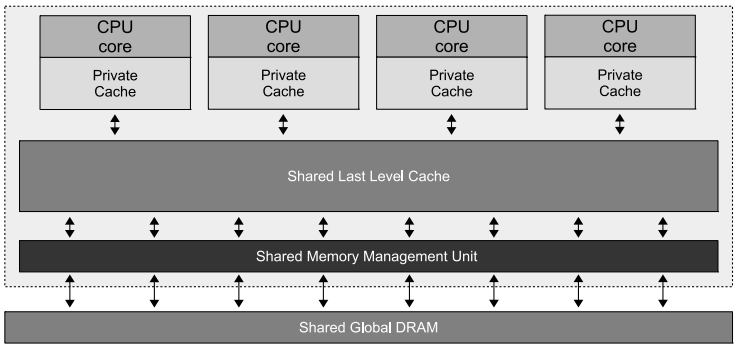


Figure 3.1: A quad-core CPU including multi-level caches.

games. Because basic numerical linear algebra operations play crucial roles in real-time 3D computer graphics, GPUs are designed for this set of operations. Because GPUs offer higher peak performance and bandwidth, numerical linear algebra applications can deliver much higher performance than merely using multicore CPUs. In the year 2007, the birth of nVidia's CUDA programming model made GPU programming much easier for researchers without background knowledge on programming shading languages such as GLSL[1] and HLSL[2]. To describe non-graphics applications on GPU, a new research direction general-purpose computation on graphics hardware (or GPGPU for short) has been naturally established [100, 134]. Figure 3.2 shows a GPU composed of four cores, private scratchpad memory, and private and shared caches.

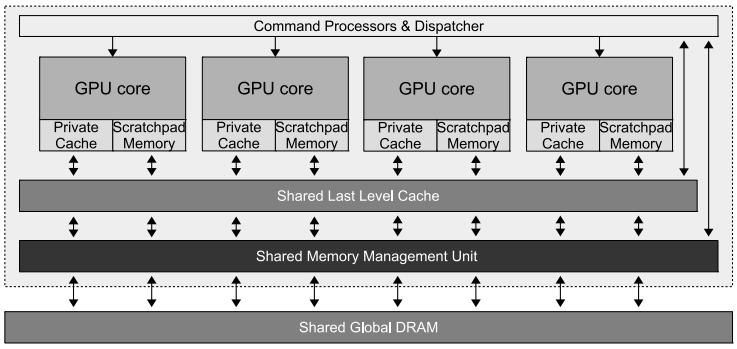


Figure 3.2: A GPU composed of four cores, scratchpad memory, and caches.

Because CUDA and OpenCL are both widely used in GPU programming and they actually deliver comparable performance [74], our Sparse BLAS algorithms

[1]The OpenGL Shading Language from the OpenGL Architecture Review Board (ARB).
[2]The High Level Shading Language from the Microsoft Corp..

support both of them. We use CUDA implementation on nVidia GPUs and OpenCL implementation on AMD GPUs.

For simplicity, we define the following unified terminologies: (1) *thread* denotes *thread* in CUDA and *work item* in OpenCL, (2) *thread bunch* denotes *warp* in nVidia GPU and *wavefront* in AMD GPU, (3) *thread group* denotes *thread block* or *cooperative thread array (CTA)* in CUDA and *work group* in OpenCL, (4) *core* denotes *streaming multiprocessor (SMX)* or *Maxwell streaming multiprocessor (SMM)* in nVidia GPU and *compute unit* in AMD GPU, and (5) *scratchpad memory* denotes *shared memory* in CUDA and *local memory* in OpenCL.

Because GPU cores can execute massively parallel lightweight threads on SIMD units for higher aggregate throughput, applications with good data-level parallelism can be significantly accelerated by GPUs [42, 43, 41]. As such, some modern supercomputers have already used GPUs as their accelerators. However, utilizing the power of GPUs requires rewriting programs in CUDA, OpenCL or other GPU-oriented programming models. This brings non-trivial software engineering overhead for applications with a lot of legacy code [37, 165]. Furthermore, because GPUs have their own memory, data have to be transferred back and forth between CPU-controlled system memory and GPU-controlled device memory. This data transfer may offset gained performance improvements from GPUs. Also, programming data transfer makes software more complicated and less easy to maintain [180]. Figure 3.3 shows a loosely coupled CPU-GPU heterogeneous system. We can see that the CPU and the GPU are connected by a PCIe interface, which delivers much lower bandwidth than the GPU device memory.

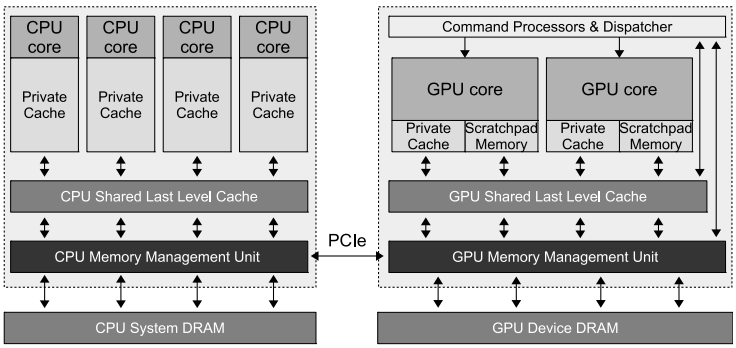


Figure 3.3: A loosely coupled CPU-GPU heterogeneous system with PCIe interface.

3.4 Manycore Coprocessor and CPU

Inspired by the success of GPU computing, Intel designed manycore coprocessors and CPUs for aggregate throughput as well. Like GPUs, manycore coprocessors and CPUs contain many relatively small cores in one chip. However, each small core can run an operating system and is strengthened by wider SIMD units. Even

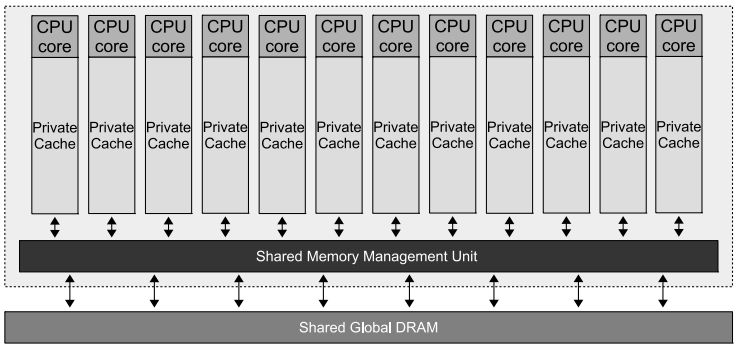


Figure 3.4: A manycore CPU including 12 small cores and their private caches.

though the small cores are not as powerful as their counterparts in multicore CPUs, a manycore coprocessor or CPU can include at least tens of such small cores. Thus the computational power is also impressive [73], compared with GPUs. The first generations of such chips were released in a coprocessor form, which is similar to an accelerator but has more flexible memory management. The later generations can be used independently thus have no fundamental difference compared to classic multicore CPUs. Figure 3.4 plots a manycore coprocessor/CPU composed of 12 small cores and their private caches.

One obvious advantage of manycore coprocessors and CPUs is the programmability. Ideally, unchanged legacy code for classic CPUs can run smoothly on those chips. As a result, a large mount of applications are well prepared. Moreover, the data transfer overhead in a CPU-GPU system can be completely avoided when using a manycore CPU as the only compute device in one system.

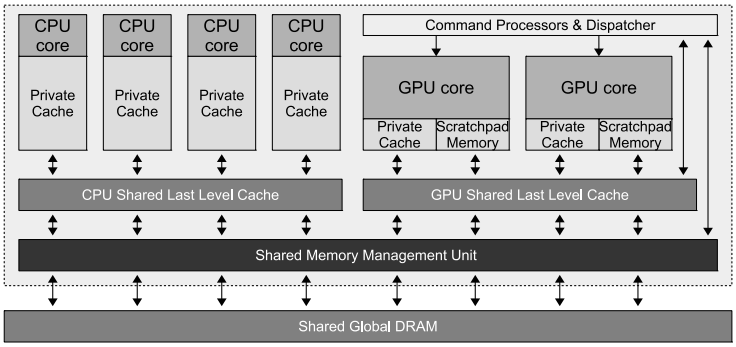
3.5 Tightly Coupled CPU-GPU Heterogeneous Processor

Recently, heterogeneous processors (which are also known as heterogeneous chip multiprocessors or asymmetric multicore processors, hCMPs or AMPs for short) have been designed [105, 96] and implemented [30, 3, 53, 141, 150]. Compared to homogeneous processors, heterogeneous processors can deliver improved overall performance and power efficiency [46], while sufficient heterogeneous parallelisms exist. The main characteristics of heterogeneous processors include unified shared memory and fast communication among different types of cores (e.g., CPU cores and GPU cores). Practically, heterogeneous system architecture (HSA) [90], OpenCL [136] and CUDA [137] have supplied toolkits for programming heterogeneous processors.

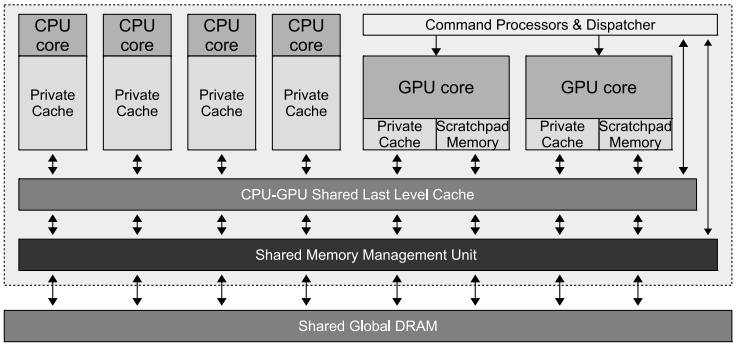
Compared to homogeneous chip multiprocessors such as CPUs and GPUs, the heterogeneous processors are able to combine different types of cores into one chip. Thus heterogeneous processors offer more flexibility in architecture design space. The Cell Broadband Engine [185, 167, 168] can be seen as an early form of heterogeneous processor. Currently, because of mature CPU and GPU architectures, pro-

gramming environments and applications, CPU-GPU integrated heterogeneous processor with multiple instruction set architectures (ISAs) is the most widely adopted choice. Representatives of this model include AMD Accelerated Processing Units (APUs) [30, 3, 159], Intel multi-CPU and GPU system-on-a-chip (SoC) devices [53], nVidia Echelon heterogeneous GPU architecture [96], and many mobile processors (e.g., nVidia Tegra [141] and Qualcomm Snapdragon [150]).

Figure 3.5 shows two block diagrams of heterogeneous processors used as one of the experimental testbeds in this thesis. In general, a heterogeneous processor consists of four major parts: (1) a group of CPU cores with hardware-controlled caches, (2) a group of GPU cores with shared command processors, software-controlled scratchpad memory and hardware-controlled caches, (3) shared memory management unit, and (4) shared global dynamic random-access memory (DRAM). The last level cache of the two types of cores can be separate as shown in Figure 3.5(a) or shared as shown in Figure 3.5(b).



(a) Heterogeneous processor with separate last level cache



(b) Heterogeneous processor with CPU-GPU shared last level cache

Figure 3.5: Block diagrams of two representative heterogeneous processors.

The CPU cores have higher single-thread performance due to out-of-order execution, branch prediction and large amounts of caches. The GPU cores execute massively parallel lightweight threads on SIMD units for higher aggregate through-

put. The two types of compute units have completely different ISAs and separate cache sub-systems.

Compared to loosely-coupled CPU-GPU heterogeneous systems shown in Figure 3.3, the two types of cores in a heterogeneous processor share one single unified address space instead of using separate address spaces (i.e., system memory space and GPU device memory space). One obvious benefit is avoiding data transfer through connection interfaces (e.g., PCIe link), which is one of the most well known bottlenecks of coprocessor/accelerator computing [82]. Additionally, GPU cores can access more memory by paging memory to and from disk. Further, the consistent pageable shared virtual memory can be fully or partially coherent, meaning that much more efficient CPU-GPU interactions are possible due to eliminated heavyweight synchronization (i.e., flushing and GPU cache invalidation). Currently, programming on the unified address space and low-overhead kernel launch are supported by HSA [90], OpenCL [136] and CUDA [137].

To leverage the heterogeneous processors, existing research has concentrated on various coarse-grained methods that exploit task, data and pipeline parallelism in the heterogeneous processors. However, it is still an open question whether or not the new features of the emerging heterogeneous processors can expose fine-grained parallelism in fundamental data structure and algorithm design. Also, whether new designs can outperform their conventional counterparts plus the coarse-grained parallelization is a further question. A lot of prior work has concentrated on exploiting coarse-grained parallelism or one-side computation in the heterogeneous processors. The current literature can be classified into four groups: (1) eliminating data transfer, (2) decomposing tasks and data, (3) pipelining, and (4) prefetching data.

Eliminating data transfer over a PCIe bus is one of the most distinct advantages brought by the heterogeneous processors, thus its influence on performance and energy consumption has been relatively well studied. Research [49, 181, 133] reported that various benchmarks can obtain performance improvements from the AMD APUs because of reduced data movement cost. Besides the performance benefits, research [169, 138] demonstrated that non-negligible power savings can be achieved by running programs on the APUs rather than the discrete GPUs because of shorter data path and the elimination of the PCIe bus and controller. Further, Daga and Nutter [48] showed that using the much larger system memory makes searches on very large B+ tree possible.

Decomposing tasks and data is also widely studied in heterogeneous system research. Research [106, 174] proposed scheduling approaches that map workloads onto the most appropriate core types in the single-ISA heterogeneous processors. In recent years, as GPU computing is becoming more and more important, scheduling on multi-ISA heterogeneous environments has been a hot topic. StarPU [9], Qilin [125], Glinda [163] and HDSS [22] are representatives that can simultaneously execute suitable compute programs for different data portions on CPUs and GPUs.

Pipelining is another widely used approach that divides a program into multiple stages and executes them on most suitable compute units in parallel. Heterogeneous environments further enable pipeline parallelism to minimize serial bottleneck in Amdahl's Law [88, 148, 59, 133]. Chen et al. [44] pipelined *map* and *reduce* stages on different compute units. Additionally, pipelining scheme can also expose wider

design dimensions. Wang et al. [181] used CPU for relieving GPU workload after each previous iteration finished, thus overall execution time was largely reduced. He et al. [87] exposed data parallelism in pipeline parallelism by using both CPU and GPU for every high-level data parallel stage.

Prefetching data can be considered with heterogeneity as well. Once GPU and CPU share one cache block, the idle integrated GPU compute units can be leveraged as prefetchers for improving single thread performance of the CPU [186, 187], and vice versa [189]. Further, Arora et al. [6] argued that stride-based prefetchers are likely to become significantly less relevant on the CPU if a GPU is integrated.

4. Parallelism in Data Structures and Algorithms

4.1 Overview

Fundamental data structures and algorithms play the key roles in implementing any computer programs. Data structures are approaches to store information. Algorithms are methods for solving problems. To accelerate Sparse BLAS on modern computer architectures, effectively utilizing parallel-friendly data structures and scalable algorithms is crucial. This chapter introduces some parallel building blocks used in the implementation of Sparse BLAS.

4.2 Contributions

This chapter makes the following contributions:

- A new method that uses offset information for converting a complex segmented sum operation to a simple inclusive all-scan operation.
- A performance comparison of five recently developed merge methods on three nVidia and AMD GPUs. To the best of our knowledge, no literature has reported performance of merging short sequences of size less than 2^{12}, which fully use on-chip scratchpad memory.
- A new heap data structure named *ad*-heap is proposed for faster fundamental heap operations on heterogeneous processors. To the best of our knowledge, the *ad*-heap is the first fundamental data structure that efficiently leveraged the two different types of cores in the emerging heterogeneous processors through fine-grained frequent interactions between the CPUs and the GPUs.

4.3 Simple Data-Level Parallelism

4.3.1 Vector Addition

Data-level parallelism (or data parallelism) distributes data across different compute units and independently runs the same instructions on them. A typical pattern of

data-level parallelism is single instruction, multiple data (SIMD), which is ubiquitous in modern microprocessors. The most easily understood data-level parallelism algorithm may be the addition of two dense vectors. The resulting vector of this operation is dense as well. Suppose that the two inputs are a and b, both of size n, and the output is vector c of the same size, the parallel procedure is shown in Algorithm 1. Ideally, this operation can be completed within one time unit, if a computer system contains at least n compute units and n memory paths. In contrast, a sequential method requires n time units, even though the used computer system has exactly the same configuration, which is shown in Algorithm 2.

Algorithm 1 Addition of two dense vectors in parallel.

```
1: function VECTOR_ADDITION_PARALLEL(*in1, *in2, *out, len)
2:     for i = 0 to len − 1 in parallel do
3:         out[i] ← in1[i] + in2[i]
4:     end for
5: end function
```

From the point view of pseudocode algorithm representation, this thesis uses keyword **in parallel** right after a **for** loop for exhibiting a routine can be executed in parallel. For example, the sequential version of Algorithm 1 is shown in Algorithm 2.

Algorithm 2 Addition of two dense vectors in serial.

```
1: function VECTOR_ADDITION_SERIAL(*in1, *in2, *out, len)
2:     for i = 0 to len − 1 do
3:         out[i] ← in1[i] + in2[i]
4:     end for
5: end function
```

4.3.2 Reduction

Sometimes it is required to calculate a scalar from an array. The scalar can be the maximum/minimum value or the sum of all of the entries of the array. Reduction operation is commonly used to obtain the scalar. Algorithm 3 gives pseudocode of a serial reduction-sum. This method loops through each entry in the array and sum all entries up to obtain the scalar.

Algorithm 3 Serial reduction-sum.

```
1: function REDUCTION_SUM_SERIAL(*in, len)
2:     sum ← 0
3:     for i = 0 to len − 1 do
4:         sum ← sum+in[i]
5:     end for
6:     return sum
7: end function
```

Suppose that the length of the input array is n, serial reduction has a work complexity $O(n)$. In modern microprocessors, data in memory can be efficiently streamed to a compute unit by prefetching thus has high cache hit rate. But when a system has more compute units, they cannot be automatically used for higher performance. Therefore, parallel reduction has been designed through a balanced tree structure. The height (i.e., level or depth) of a balanced binary tree is $log_2 n + 1$, if it has n nodes (i.e., leaves). So an array of size n to be reduced can be integrated with a balanced binary tree of depth $log_2 n + 1$. Figure 4.1 shows an example of executing reduction-sum on an array of size 8 with a balanced tree of level 4. We can see that addition operations between every two levels of the tree can run in parallel. For example, the four additions between the two levels on the top can run independently in parallel.

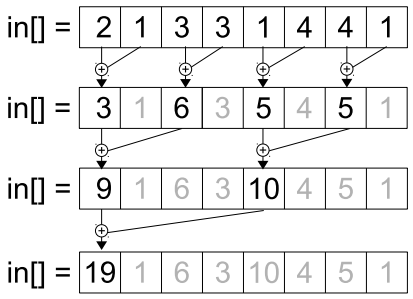


Figure 4.1: An parallel reduction-sum on an array of size 8.

A parallel implementation of the reduction-sum operation is shown in Algorithm 4. The parallel version needs $n - 1$ additions and thus still has runtime complexity $O(n)$. However, the operation can be completed in $O(\log_2 n)$ time, if at least $n/2$ compute units and enough fast memory are available.

Algorithm 4 Parallel reduction-sum.

```
1: function REDUCTION_SUM_PARALLEL(*in, len)
2:     for d = 0 to log2 len − 1 do
3:         for k = 0 to len − 1 by 2^(d+1) in parallel do
4:             in[k + 2^d − 1] ← in[k + 2^d − 1] + in[k + 2^(d+1) − 1]
5:         end for
6:     end for
7:     sum ←in[0]
8:     return sum
9: end function
```

4.4 Scan (Prefix Sum)

Scan, which is also known as prefix sum, is a powerful building block in parallel computing. For different purposes, scan algorithms can be categorized as inclusive scan and exclusive scan. Given an array of size n

$$a = \left[a_0, a_1, ..., a_{n-1}\right]$$

and an associative operator $\oplus$ as inputs of a scan operation, the output of the inclusive scan is another array of the same size

$$a_{in} = \left[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-1})\right],$$

and the exclusive scan generates an array of the same size

$$a_{ex} = \left[0, a_0, ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})\right].$$

We see that

$$a_{ex} = a_{in} - a.$$

For example, setting the $\oplus$ operator to arithmetic $+$ and assuming the input is $[3, 6, 7, 8, 9, 12]$, the outputs will be $[3, 9, 16, 24, 33, 45]$ and $[0, 3, 9, 16, 24, 33]$ through the inclusive scan and the exclusive scan, respectively.

Scan can convert seemingly serial computations to parallel operations. Unstructured sparse matrices have rows/columns of irregular sizes, thus can be a good scenario for using parallel scan operation. The following subsection gives an example.

4.4.1 An Example Case: Eliminating Unused Entries

Suppose that we want to eliminate all even columns of a sparse matrix

$$A = \begin{bmatrix} 0 & c & f & j & 0 & 0 \\ a & 0 & g & k & n & o \\ b & 0 & 0 & 0 & m & 0 \\ 0 & d & h & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & p \\ 0 & e & i & l & n & q \end{bmatrix},$$

and to obatin

$$A_{odd} = \begin{bmatrix} 0 & f & 0 \\ a & g & n \\ b & 0 & m \\ 0 & h & 0 \\ 0 & 0 & 0 \\ 0 & i & n \end{bmatrix}$$

only containing three odd columns of A. Because the two sparse matrices are required to be stored in compressed style, parallelizing this partitioning operation is not trivial.

Using the 2nd row as an example, it is stored as two arrays

$$\begin{aligned} col\_idx &= \left[0, 2, 3, 4, 5\right], \\ val &= \left[a, g, k, n, o\right]. \end{aligned}$$

Obviously, removing its even columns to obtain

$$\begin{aligned} col\_idx_{odd} &= \left[0, 2, 4\right], \\ value_{odd} &= \left[a, g, n\right], \end{aligned}$$

can be easily achieved by sequential iterating all entries and saving the ones with odd indices. However, parallel processing is expected to have higher performance. This can be achievable by using the scan primitive.

First, an intermediate array is introduced to label each nonzero entry. A nonzero entry is labeled as 1 if it is an index for a odd column, otherwise 0 is used. So the 2nd row with this auxiliary array is stored in

$$\begin{aligned} col\_idx &= \left[0, 2, 3, 4, 5\right], \\ val &= \left[a, g, k, n, o\right], \\ aux\_array &= \left[1, 1, 0, 1, 0\right]. \end{aligned}$$

Then an exclusive scan is executed on the auxiliary array to obtain

$$aux\_array_{ex} = \left[0, 1, 2, 2, 3\right],$$

which stores the target positions of the odd columns. Now it is easy to save the odd columns of the whole 2nd row of A by Algorithm 5 in parallel. This method can be easily extended to the whole matrix.

Algorithm 5 Eliminating unused nonzero entries in parallel.

1: **for** $i = 0$ **to** 4 **in parallel do**
2: **if** `aux_array[`i`]` $= 1$ **then**
3: `col_idx`$_{odd}$`[aux_array`$_{ex}$`[`i`]]` $\leftarrow$ `col_idx[`i`]`
4: `val`$_{odd}$`[aux_array`$_{ex}$`[`i`]]` $\leftarrow$ `val[`i`]`
5: **end if**
6: **end for**

4.4.2 All-Scan

A scan operation is called all-scan. If it runs on the whole input array, as opposed to part of it. Algorithm 6 shows a serial out-of-place inclusive scan. Similar to the serial reduction described above, all-scan iterates through all entries of the array and lets each entry of the output array to store the sum of all entries except ones on its right in the input array. Sometimes, the scan is required to be in-place for less memory allocation. Algorithm 7 gives an in-place version of the serial all-scan. Similarly,

Algorithm 6 Serial inclusive all-scan (out-of-place).

```
1: function INCLUSIVE_ALL-SCAN_SERIAL(*in, len, *out)
2:     sum ← in[0]
3:     out[0] ← sum
4:     for i = 1 to len − 1 do
5:         sum ← sum+in[i]
6:         out[i] ← sum
7:     end for
8: end function
```

Algorithm 7 Serial inclusive all-scan (in-place).

```
1: function EXCLUSIVE_ALL-SCAN_SERIAL(*in, len)
2:     oldval ← in[0]
3:     for i = 1 to len − 1 do
4:         newval ← in[i]
5:         in[i] ← oldval+ in[i − 1]
6:         oldval ← newval
7:     end for
8: end function
```

Algorithm 8 Serial exclusive all-scan (out-of-place).

```
1: function EXCLUSIVE_ALL-SCAN_SERIAL(*in, len, *out)
2:     sum ← 0
3:     out[0] ← sum
4:     for i = 1 to len − 1 do
5:         sum ← sum+in[i − 1]
6:         out[i] ← sum
7:     end for
8: end function
```

Algorithm 9 Serial exclusive all-scan (in-place).

```
1: function EXCLUSIVE_ALL-SCAN_SERIAL(*in, len)
2:     oldval ← in[0]
3:     in[0] ← 0
4:     for i = 1 to len − 1 do
5:         newval ← in[i]
6:         in[i] ← oldval+ in[i − 1]
7:         oldval ← newval
8:     end for
9: end function
```

serial out-of-place and in-place implementations of exclusive all-scan are given by Algorithms 8 and 9, respectively.

Parallel scan algorithm has been given by Blelloch [25, 26, 27]. The method divides all-scan into two phases: up-sweep and down-sweep, each is organized by a balanced tree described above. Figure 4.2 gives an example conducting all-scan on an array of size 8. We can see that the up-sweep phase basically executes a parallel reduction operation, and the down-sweep phase transmits zeros and adds previous sums to subsequent entries. Algorithm 10 shows pseudocode of this operation. Its runtime complexity is $O(n)$ but can be completed in $O(\log_2 n)$ time with enough parallel resources.

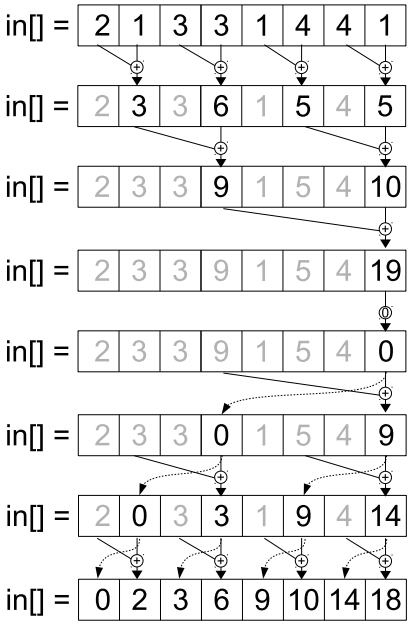


Figure 4.2: An example of the parallel scan.

Recently, the in-place inclusive and exclusive all-scan operations have already been defined by a set of *Workgroup Functions* of OpenCL 2.0 [136]. Using vendors' implementation may bring high efficiency and programming productivity.

4.4.3 Segmented Scan

Segmented scan is a more general case of the all-scan operation. It divides an array into multiple segments and performs a all-scan operation for each segment. So an all-scan can be seen as a segmented scan with only one segment. To label starting and ending points, a segment has its first entry flagged as `TRUE` and the other entries flagged as `FALSE`. Other methods, such as labeling the last entry as `FALSE` and all the others as `TRUE` [188], are also usable.

A parallel implementation of in-place exclusive segmented scan is shown in Algorithm 11. This method is very similar to the above algorithm for the all-scan operation, except flags are taken into account. In each step, flags are used for deciding

Algorithm 10 Parallel exclusive all-scan (in-place).

```
1:  function EXCLUSIVE_ALL-SCAN_PARALLEL(*in, len)
2:      //up-sweep
3:      for d = 0 to log2 len − 1 do
4:          for k = 0 to len − 1 by 2^(d+1) in parallel do
5:              in[k + 2^(d+1) − 1] ← in[k + 2^d − 1] + in[k + 2^(d+1) − 1]
6:          end for
7:      end for
8:      in[len − 1] ← 0
9:      //down-sweep
10:     for d = log2 len − 1 to 0 do
11:         for k = 0 to len − 1 by 2^(d+1) in parallel do
12:             t ← in[k + 2^d − 1]
13:             in[k + 2^d − 1] ← in[k + 2^(d+1) − 1]
14:             in[k + 2^(d+1) − 1] ← t+in[k + 2^(d+1) − 1]
15:         end for
16:     end for
17: end function
```

whether an addition is required. Also, if any one of the inputs of the addition is performed, the target position is flagged as `TRUE` in the up-sweep phase, or modified to `FALSE`, if the entry is involved in any steps of the down-sweep phase.

4.4.4 Segmented Sum Using Inclusive Scan

Segmented sum can be seen as a simplified alternative of segmented scan. It performs a reduction-sum operation for the entries in each segment of an array and collects the sums. Algorithm 12 lists a serial segmented sum algorithm. Parallel segmented sum algorithm can be implemented by segmented scan methods [28, 40, 161, 63]. However, as shown in Algorithm 11, using segmented scan for segmented sum may be complex and not efficient enough. Thus we prepare `seg_offset` as an auxiliary array to facilitate implementation of segmented sum by the way of the all-scan, which can be much more efficient than segmented scan.

Algorithm 13 shows the fast segmented sum using `seg_offset` and an inclusive all-scan. Figure 4.3 gives an example. The principle of this operation is that the all-scan is essentially an increment operation. Once a segment knows the distance (i.e., offset) between its head and its tail, its partial sum can be deduced from its all-scan results. Therefore, the more complex parallel segmented sum operation in [28, 40, 161, 63] can be replaced by a faster all-scan operation (line 4) and a few arithmetic operations (lines 5–7). Also note that the `seg_offset` array can be an alternative to the `flag` array.

Algorithm 11 Parallel exclusive segmented scan (in-place).

```
1: function EXCLUSIVE_SEGMENTED_SCAN(*in, len, *flag_original)
2:     MALLOC(*flag, len)
3:     MEMCPY(*flag, *flag_original, len)
4:     for d = 0 to log2 len − 1 do
5:         for k = 0 to len − 1 by 2^(d+1) in parallel do
6:             if flag[k + 2^(d+1) − 1] = FALSE then
7:                 in[k + 2^(d+1) − 1] ← in[k + 2^d − 1] + in[k + 2^(d+1) − 1]
8:             end if
9:             flag[k + 2^(d+1) − 1] ← flag[k + 2^d − 1] ∨ flag[k + 2^(d+1) − 1]
10:        end for
11:    end for
12:    in[len − 1] ← 0
13:    flag[len − 1] ← FALSE
14:    for d = log2 len − 1 to 0 do
15:        for k = 0 to len − 1 by 2^(d+1) in parallel do
16:            t ← in[k + 2^d − 1]
17:            in[k + 2^d − 1] ← in[k + 2^(d+1) − 1]
18:            if flag_original[k + 2^d] = TRUE then
19:                in[k + 2^(d+1) − 1] ← 0
20:            else if flag[k + 2^d − 1] = TRUE then
21:                in[k + 2^(d+1) − 1] ← t
22:            else
23:                in[k + 2^(d+1) − 1] ← t+in[k + 2^(d+1) − 1]
24:            end if
25:            flag[k + 2^d − 1] ← FALSE
26:        end for
27:    end for
28: end function
```

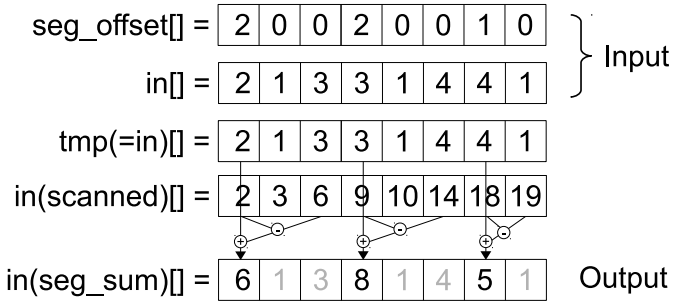


Figure 4.3: An example of the fast segmented sum.

Algorithm 12 Serial segmented sum operation.

```
1: function SEGMENTED_SUM(*in, len, *flag)
2:     for i = 0 to len − 1 do
3:         if flag[i] = TRUE then
4:             j ← i + 1
5:             while flag[j] = FALSE && j < len do
6:                 in[i] ← in[i] + in[j]
7:                 j ← j + 1
8:             end while
9:         else
10:            in[i] ← 0
11:        end if
12:    end for
13: end function
```

Algorithm 13 Fast segmented sum using `seg_offset`.

```
1: function FAST_SEGMENTED_SUM(*in, len, *seg_offset)
2:     MALLOC(*tmp, len)
3:     MEMCPY(*tmp, *in)
4:     INCLUSIVE_ALL-SCAN_PARALLEL(*in, len)
5:     for i = 0 to len − 1 in parallel do
6:         in[i] ← in[i+seg_offset[i]] − in[i] + tmp[i]
7:     end for
8:     FREE(*tmp)
9: end function
```

4.5 Sorting and Merging

Sorting is a fundamental algorithm in computer science and can find its broad use in Sparse BLAS. Recall the first example of this thesis and Figure 2.3, searching the 5th small ball takes longer time if the index array is not sorted. If so, a fast binary search can be utilized to obtain the position of index 4 in $O(\log n)$ as opposed to $O(n)$ time.

In the past decades, many sorting algorithms have been proposed. This thesis mainly focuses on parallel-friendly sorting methods such as sorting network approaches and the recently developed merge path method.

4.5.1 An Example Case: Permutation

Permutation can be used for rearranging a matrix of expected order of rows/-columns [69]. We sometimes want to exploit the number of nonzero entries in rows,

and to permute a sparse matrix

$$A = \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} \begin{bmatrix} 0 & c & f & j & 0 & 0 \\ a & 0 & g & k & n & o \\ b & 0 & 0 & 0 & m & 0 \\ 0 & d & h & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & p \\ 0 & e & i & l & n & q \end{bmatrix}$$

to another form

$$A' = \begin{matrix} 1 \\ 5 \\ 0 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{bmatrix} a & 0 & g & k & n & o \\ 0 & e & i & l & n & q \\ 0 & c & f & j & 0 & 0 \\ b & 0 & 0 & 0 & m & 0 \\ 0 & d & h & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & p \end{bmatrix},$$

in which the longer rows appear before shorter ones, where the column vector to the left of the matrix contains indices of the rows of the matrix. This permutation operation can be implemented by sorting an array of key-value pairs: the key is the number of nonzero entries of each row, and the value is the index of each row of the matrix. In this example, the array of key-value pair is

$$a\langle \texttt{key}, \texttt{value} \rangle = \left[\langle 3, 0 \rangle, \langle 5, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 1, 4 \rangle, \langle 5, 5 \rangle \right].$$

After the array is sorted by its key in a descending order, it becomes

$$a\langle \texttt{key}, \texttt{value} \rangle = \left[\langle 5, 1 \rangle, \langle 5, 5 \rangle, \langle 3, 0 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 1, 4 \rangle \right].$$

Then the value part of the key-value array is actually the new order of rows of the matrix.

4.5.2 Bitonic Sort

Bitonic sort is a class of sorting network algorithms [17], which connect entries of the array to be sorted by comparators. The algorithm can be divided into multiple stages. Each stage can have multiple steps, and each step changes connecting paths and may reorder the entries of the array. Because connecting wires are independent of each other, all comparators within one step can execute in parallel. Figure 4.4 gives an example of sorting an array of size 8. We can see that there are 4 pairs of sequences in the first stage. After that, the array has 4 pairs of ordered sequence. After the second stage, 2 ordered sequences are generated. Finally, there is only one ordered sequence is left.

Algorithm 14 gives the pseudocode of the bitonic sort. This method requires $\log n$ stages for an input array of size n. The ith stage has i steps. So the number of operations of the bitonic sort is $O(n \log_2^2(n) + \log_2(n))$. It can be completed in $O(\log_2^2(n))$ time with enough parallel resources. The bitonic sort can also be efficiently implemented by using SIMD intrinsics [89].

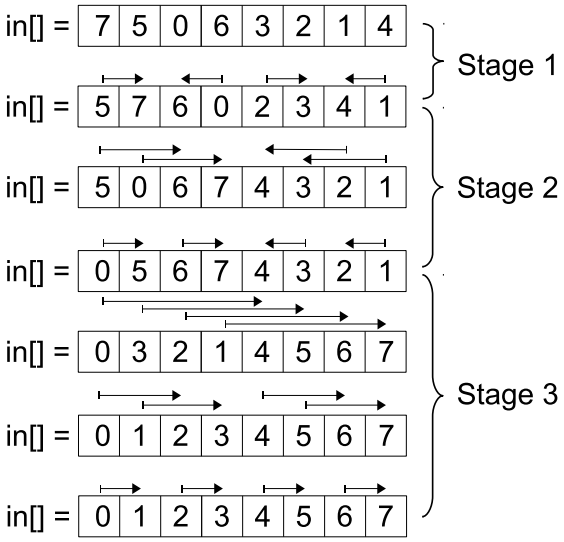


Figure 4.4: An example of bitonic sort. The arrows indicate ascending order operated by comparators.

Algorithm 14 Parallel bitonic sort (in-place).

```
1:  function BITONIC_SORT_PARALLEL(⋆in, len)
2:      for k = 1 to log_2 len do
3:          for j = 2^(k−1) down to 1 do
4:              for i = 0 to len − 1 in parallel do
5:                  ixj ← XOR(i, j)
6:                  if ixj > i then
7:                      if (i ∧ 2^k) = 0 then
8:                          if in[i] > in[ixj] then
9:                              SWAP(in[i], in[ixj])
10:                         end if
11:                     else
12:                         if in[i] < in[ixj] then
13:                             SWAP(in[i], in[ixj])
14:                         end if
15:                     end if
16:                 end if
17:             end for
18:             j ← j/2
19:         end for
20:     end for
21: end function
```

4.5.3 Odd-Even Sort

Another sorting network method is odd-even sort, which is very similar to the above bitonic sort. Its idea is to sort all odd and all even keys separately and then merge them. Figure 4.5 gives an example of sorting an array of size 8. We can see that odd-even sort has $\log_2 n$ stages as well, and the ith stage has i steps. Algorithm 15 gives pseudocode of odd-even sort, which has the same complexity with the bitonic sort.

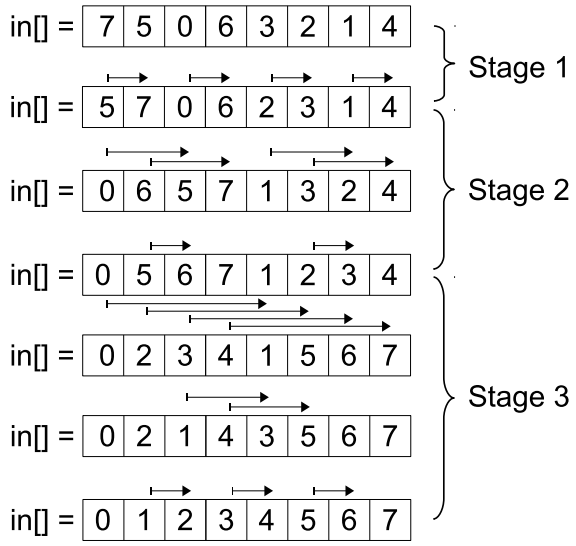


Figure 4.5: An example of odd-even sort. The arrows indicate ascending order operated by comparators.

4.5.4 Ranking Merge

Merge algorithms give better sorting performance, if the input array is composed of multiple already ordered subarrays. Algorithm 16 gives pseudocode of a serial merge that sorts two ordered input arrays and save the result in an output array. This approach constructs 3 pointers for the two input arrays and the output array. Because the two inputs are sorted, the pointers can trivially iterative through all entries of them, compare their values, and select one (e.g., the one less-than-or-equal-to the other one) to save to the output array. To merge two ordered lists, a serial merge algorithm requires n comparison, where n is the size of the resulting array. Merge methods are out-of-place, meaning they need an extra array of size n to save the resulting array.

While multiple compute units are usable, merge algorithm can be parallelized easily by using binary search as a ranking tool. Thus this method is called ranking merge [157]. The basic idea is that each entry of the two input arrays wants to know its position in the output array, thus it computes a rank (i.e., the number of entries

Algorithm 15 Parallel odd-even sort (in-place).

```
1: function ODD-EVEN_SORT_PARALLEL(*in, len)
2:     for k = 1 to log2 len do
3:         for j = 2^(k-1) down to 1 do
4:             for i = 0 to len − 1 in parallel do
5:                 offset ← i ∧ (j − 1)
6:                 ixj ← 2 × i − offset
7:                 if j = 2^(k-1) then
8:                     if in[ixj] > in[ixj + j] then
9:                         SWAP(in[ixj], in[ixj + j])
10:                    end if
11:                else
12:                    if offset ≥ j then
13:                        if in[ixj − j] > in[ixj] then
14:                            SWAP(in[ixj − j], in[ixj])
15:                        end if
16:                    end if
17:                end if
18:            end for
19:            j ← j/2
20:        end for
21:    end for
22: end function
```

Algorithm 16 Serial merge (out-of-place).

```
1: function MERGE_SERIAL(*in1, len1, *in2, len2, *out)
2:     ai ← 0
3:     bi ← 0
4:     for ci = 0 to len1 + len2 − 1 do
5:         if ai < len1 && (bi ≥ len2 || in1[ai] ≤ in2[bi]) then
6:             out[ci] ← in1[ai]
7:             ai ← ai + 1
8:         else
9:             out[ci] ← in2[bi]
10:            bi ← bi + 1
11:        end if
12:    end for
13: end function
```

less-than-or-equal-to it in another input array) and adds the rank to its local index to obtain the final position in the output array. Algorithm 17 gives pseudocode of this method. Because the binary search of each entry is independent of all the other ones, the operation can run in parallel very well. Overall cost of ranking merge is $O(n_1 \log_2 n_2 + n_2 \log_2 n_1)$, where n_1 and n_2 are sizes of the two input arrays, respectively. Similar to its serial version, ranking merge is required to be out-of-place.

Algorithm 17 Parallel ranking merge (out-of-place).

```
1: function RANKING_MERGE_PARALLEL(*in1, len1, *in2, len2, *out)
2:     for i = 0 to len1 in parallel do
3:         idx ← BINARY_SEARCH(*in2, in1[i])
4:         out[idx + i] ← in1[i]
5:     end for
6:     for i = 0 to len2 in parallel do
7:         idx ← BINARY_SEARCH(*in1, in2[i])
8:         out[idx + i] ← in2[i]
9:     end for
10: end function
```

Sorting network algorithms such as bitonic sort and odd-even sort can also be used for merging. In their last stage (e.g., stage 3 in Figures 4.4 and 4.5), the input is already composed of two ordered subarrays of the same size. Then the merge methods can further sort the two subarrays.

4.5.5 Merge Path

Merge path algorithm [81, 142] is a recently designed merge algorithm for load balanced parallel processing. It conceptually construct a rectangular grid of size n_1 by n_2, where n_1 and n_2 are sizes of the two input arrays, respectively. Then multiple conceptual 45-degree lines as anti-diagonals are generated to partition the grid. Those lines are equally spaced, meaning that any single intersection of one line with the grid goes down to another intersection of a neighboring line with the grid through a path of the same length. Figure 4.6 shows an example of the merge path approach. We can see that a grid is established to merge two arrays of sizes 4 and 8, respectively. Then five equally spaced 45-degree anti-diagonals (i.e., lines l_0 to l_4) are generated to intersect with the grid. The paths (colored in red, orange, green and blue) between two intersections from neighboring lines have the same length of 3.

The reason of constructing the grid and the 45-degree anti-diagonals is to evenly partition the two input arrays and let each compute unit to have the same amount of work. Figure 4.6 supposes that there are 4 compute units in the system, thus divide the gird to 4 regions by 5 lines. Then each compute units can deal with 3 entries, thus the 12 entries in total are evenly distributed to the 4 compute units.

The rest problem is to find which 3 entries are assigned to a compute unit. To find those entries, the positions of the three empty circles (i.e., the three connecting points of the red path and the orange path, the orange path and the green path, and the green path and the blue path) are required to be searched by comparing entries around any possible intersections. For example, the intersection connecting the red path and the orange path is selected because entry of value 3 on the x axis is located between the entries of values 2 and 4 on the y axis. Because the two input arrays are sorted, binary search can be utilized to find those intersections.

Once the intersections are known, each compute unit completes a serial merge on two ordered subarrays. For example, the 1st compute units (colored in red) merges a

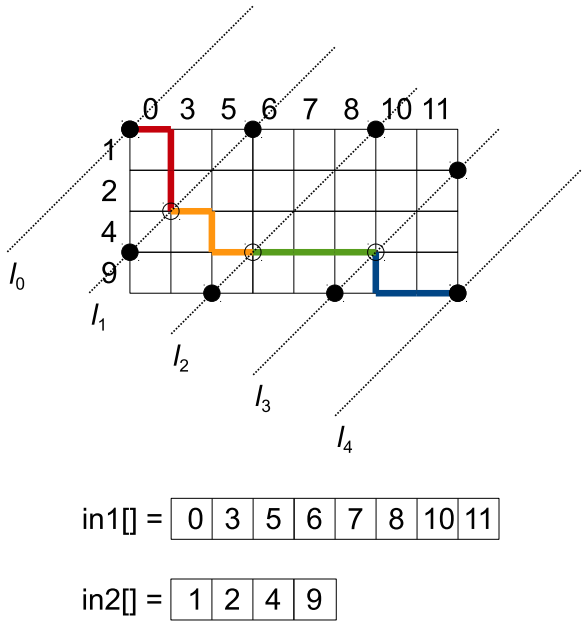


Figure 4.6: An example of merge path.

subarray including entry of value 0 and another subarray including two entries of values 1 and 2, respectively. Then the ordered three entries are stored to the first 3 positions of the resulting array.

Algorithm 18 gives pseudocode of merge path. Lines 7–25 conduct binary search on the 45-degree anti-diagonals to obtain positions of the intersections. Lines 26–33 complete serial merge between those intersections. More detailed description and complexity analysis of the GPU merge path algorithm can be found in [81].

4.5.6 A Comparison

Since merging is an important operation in sparse matrix computations (such as addition of two sparse vectors described in Chapter 7), the most efficient algorithm is required to be selected. Consider on-chip scratchpad memory of GPUs is controlled by the user and may offer performance gain for short inputs and outputs, we consider some recently implemented merge algorithms [157, 81, 99, 146, 145, 91, 55] for GPUs. First, because the main objective of the research [145, 91, 55] is efficiently merging large data in the global memory, they still use basic methods, such as bitonic sort and ranking-based merge, as building blocks for small data in the scratchpad memory. Peters et al. [146] proposed a locality-oriented advanced bitonic sort method that can reduce synchronization overhead by merging data in fast private memory instead of relatively slow shared memory. Therefore we experimentally evaluate 5 GPU merge algorithms: (1) ranking merge [157], (2) merge path [81], (3) basic oddeven

Algorithm 18 Parallel merge path (out-of-place).

```
1: function MERGE_PATH_PARALLEL(*in1, len1, *in2, len2, *out, nt)
2:     MALLOC(*border1, nt + 1)
3:     MALLOC(*border2, nt + 1)
4:     delta ← ⌈(len1 + len2)/nt⌉
5:     border1[nt] ← len1
6:     border2[nt] ← len2
7:     for i = 0 to nt in parallel do
8:         offset ← delta * i
9:         start ← offset > len2 ? offset − len2 : 0
10:        stop ← offset > len1 ? len1 : offset
11:        while stop >= start do
12:            median ← (stop + start/2)
13:            if in1[median] > in2[offset − median − 1] then
14:                if in1[median − 1] < in2[offset − median − 1] then
15:                    break
16:                else
17:                    stop ← median − 1
18:                end if
19:            else
20:                start ← median + 1
21:            end if
22:        end while
23:        border1[i] ← median
24:        border2[i] ← offset − median
25:    end for
26:    for i = 0 to nt − 1 in parallel do
27:        offset ← delta * i
28:        start1 ← border1[i]
29:        l1 ← border1[i + 1]-border1[i]
30:        start2 ← border2[i]
31:        l2 ← border2[i + 1]-border2[i]
32:        MERGE_SERIAL(&in1[start1], l1, &in2[start2], l2, &out[i * offset])
33:    end for
34: end function
```

merge [99], (4) basic bitonic merge [99], and (5) advanced bitonic merge [146]. The implementation of the algorithm (2) is extracted from the Modern GPU library [18]. The implementations of the algorithm (3) and (4) are extracted from the nVidia CUDA SDK. We implement the algorithms (1) and (5). Additionally, another reason why we conduct the evaluation is that none of the above literature presented performance of merging short sequences of size less than 2^{12}, which is the most important length for relatively short rows of our benchmark suite.

Our evaluation results of merging 32-bit keys, 32-bit key-32-bit value pairs and 32-bit key-64-bit value pairs are shown in Figure 4.7. The experimental platforms are

nVidia GeForce GTX Titan Black, nVidia GeForce GTX 980 and AMD Radeon R9 290X (see detailed specifications of the GPUs listed in Table B.2 of Appendix B). Each of the five algorithms merges two short ordered sequences of size l into one ordered output sequence of size $2l$. The sorting network methods in our evaluation only execute the last stage, since both inputs are sorted. To saturate throughput of GPUs, the whole problem size is set to size 2^{25}. For example, 2^{14} thread groups are launched while each of them merges two sub-sequences of size $l = 2^{10}$. We execute each problem set through multiple thread groups of different sizes and record the best performance for the evaluation.

In Figure 4.7, we can see that the GPU merge path algorithm almost always outperforms other methods while sub-sequence size is no less than 2^8. The extra advantages of the merge path method are that it can evenly assign work load to threads and can easily deal with the input sequences of arbitrary sizes.

We can see that the ranking merge is faster than the merge path method in Figure 4.7(f). But this algorithm requires more scratchpad memory and thus cannot scale to longer sequences. The basic bitonic merge and the basic oddeven merge in general do not show better performance and cannot simply deal with data of arbitrary sizes. The advanced bitonic sort method is always the slowest because it loads data from the scratchpad memory to thread private memory (register file or an off-chip memory space) for data locality. However, due to the small or negatively large latency gap between the scratchpad memory and the thread private memory, the load operations actually reduce the overall performance. Thus this method should only be used for migrating global memory access to scratchpad memory access.

We can also see that the AMD Radeon R9 290X GPU is almost always much faster than the two nVidia GPUs in all tests. The reason is that the capacity of the scratchpad memory (2816 kB, 64 kB/core $\times$ 44 cores, in the AMD GPU, 1536 kB, 96 kB/core $\times$ 16 cores, in the nVidia Maxwell-based GTX 980 GPU and 720 kB, 48 kB/core $\times$ 15 cores, in the nVidia Kepler-based GTX Titan Black GPU) heavily influence the performance of merging small sequences. For the same reason, the GTX 980 GPU delivers better overall performance than the GTX Titan Black GPU. On the other hand, even though the AMD GPU has 64 kB scratchpad memory per core, each instance of the kernel program can only use up to 32 kB. Thus the AMD GPU cannot scale to longer sub-sequences (e.g., 2^{12} with 32-bit key-32-bit value pairs) that can be executed by using the nVidia GPUs.

4.6 Ad-Heap and k-Selection

Heap (or priority queue) data structures are heavily used in many algorithms such as *k*-nearest neighbor (*k*NN) search, finding the minimum spanning tree and the shortest path problems. Compared to the most basic binary heap, *d*-heaps [93, 182], in particular implicit *d*-heaps proposed by LaMarca and Ladner [109], have better practical performance on modern processors. However, as throughput-oriented processors (e.g., GPUs) bring higher and higher peak performance and bandwidth, heap data structures did not reap benefit from this trend because their very limited degree of data parallelism cannot saturate wide SIMD units.

In this section, we propose a new heap data structure called *ad*-heap (*a*symmetric

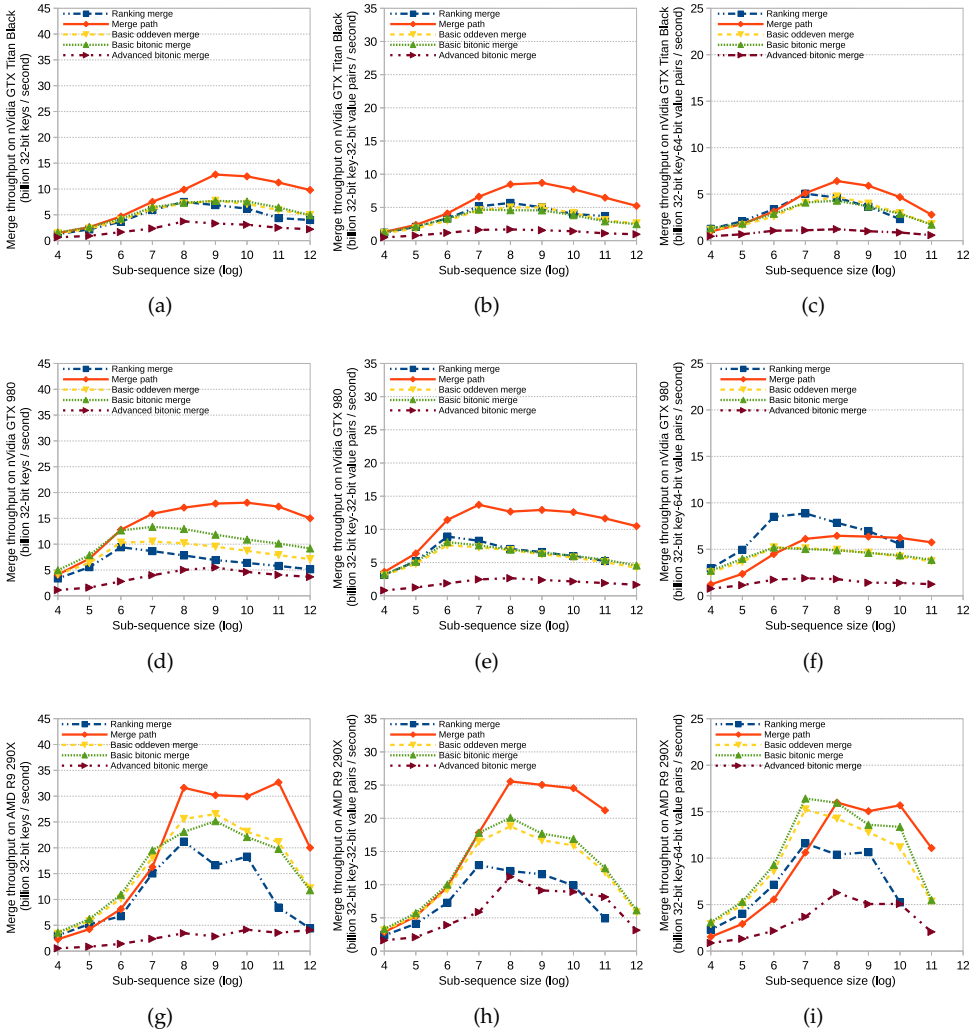


Figure 4.7: Performance comparison of merging 32-bit keys, 32-bit key-32-bit value pairs and 32-bit key-64-bit value pairs through 5 GPU merge algorithms: ranking merge, merge path, basic oddeven merge, basic bitonic merge and advanced bitonic merge on three different GPUs: nVidia GeForce GTX Titan Black, nVidia GeForce GTX 980 and AMD Radeon R9 290X.

d-heap) for heterogeneous processors. The *ad*-heap introduces an implicit bridge structure — a new component that records deferred random memory transactions and makes the two types of cores in a tightly coupled CPU-GPU heterogeneous processor focus on their most efficient memory behaviors. Thus overall bandwidth utilization and instruction throughput can be significantly improved.

We evaluate performance of the *ad*-heap by using a batch *k*-selection algorithm on

two simulated heterogeneous processors composed of real AMD CPU-GPU and Intel CPU and nVidia GPU, respectively. The experimental results show that compared with the optimal scheduling method that executes the fastest *d*-heaps on the standalone CPUs and GPUs in parallel, the *ad*-heap achieves up to 1.5x and 3.6x speedup on the two platforms, respectively.

4.6.1 Implicit d-heaps

Given a heap of size n, where $n \neq 0$, a *d*-heap data structure [93, 182] lets each parent node to have d child nodes, where $d > 2$ normally. To satisify cache-line alignment and reduce cache miss rate, the whole heap can be stored in an implicit space of size $n + d - 1$, where the extra $d - 1$ entries are padded in front of the root node and kept empty [109]. Here we call the padded space "head" of the heap. Figure 4.8 shows an example of the implicit max-*d*-heaps while $n = 12$ and $d = 4$. Note that each group of the child nodes starts from an aligned cache block.

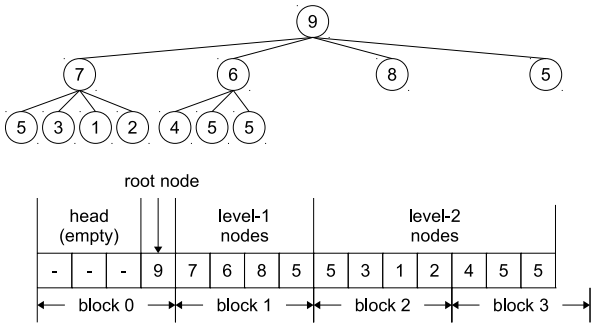


Figure 4.8: The layout of a 4-heap of size 12.

Because of the padded head, each node has to add an $offset = d - 1$ to its index in the implicit array. Given a node of index i, its array index becomes $i + offset$. Its parent node's (if $i \neq 0$) array index is $\lfloor (i - 1)/d \rfloor + offset$. If any, its first child node is located in $di + 1 + offset$ and the last child node is in array index $di + d + offset$.

Given an established non-empty max-*d*-heap, we can execute three typical heap operations:

- *insert* operation adds a new node at the end of the heap, increases the heap size to $n + 1$, and takes $O(\log_d n)$ worst-case time to reconstruct the heap property,
- *delete-max* operation copies the last node to the position of the root node, decreases the heap size to $n-1$, and takes $O(d \log_d n)$ worst-case time to reconstruct the heap property, and
- *update-key* operation updates a node, keeps the heap size unchanged, and takes $O(d \log_d n)$ worst-case time (if the root node is updated) to reconstruct the heap property.

The above heap operations depend on two more basic operations:

- *find-maxchild* operation takes $O(d)$ time to find the maximum child node for a given parent node, and
- *compare-and-swap* operation takes constant time to compare values of a child node and its parent node, then swap their values if the child node is larger.

4.6.2 ad-heap design

Performance Considerations

We first conduct analysis on the degree of parallelism of the *d*-heap operations. We can see that the *insert* operation does not have any data parallelism because the heap property is reconstructed in a bottom-up order and only the unparallelizable *compare-and-swap* operations are required. On the other hand, the *delete-max* operation reconstructs the heap property in a top-down order that does not have any data parallelism either, but executes multiple ($\log_d n$ in the worst case) lower-level parallelizable *find-maxchild* operations. For the *update-key* operation, the position and the new value of the key decides whether the bottom-up or the top-down order is executed in the heap property reconstruction. Therefore, in this paper we mainly consider accelerating the heap property reconstruction in the top-down order. After all, the *insert* operation can be efficiently executed in serial because the heap should be very shallow if the d is large.

Without loss of generality, we focus on an *update-key* operation that updates the root node of a non-empty max-*d*-heap. To reconstruct the heap property in the top-down order, the *update-key* operation alternately executes the *find-maxchild* operations and the *compare-and-swap* operations until the heap property is satisfied or the last changed parent node does not have any child node. Note that the *swap* operation can be simplified because the child node does not need to be updated in the procedure. Actually its value can be kept in thread register and be reused until the final round. Algorithms 19 and 20 show pseudocode of the *update-key* operation and the *find-maxchild* operation, respectively.

Imagine the whole operation is executed on a wide SIMD processor (e.g., GPU), the *find-maxchild* operation can be efficiently accelerated by the SIMD units through a *streaming reduction* scheme within much faster $O(\log d)$ time instead of original $O(d)$ time. And because of wider memory controllers, one group of w continuous SIMD threads (a warp in the nVidia GPUs or a wavefront in the AMD GPUs) can load w aligned continuous entries from the off-chip memory to the on-chip scratchpad memory by one off-chip memory transaction (coalesced memory access). Thus to load d child nodes from the off-chip memory, only d/w memory transactions are required.

A similar idea has been implemented on the CPU vector units. Furtak et al. [75] accelerated *d*-heap *find-maxchild* operations by utilizing x86 SSE instructions. The results showed 15% - 31% execution time reduction, on average, in a mixed benchmark composed of the *delete-max* operations and *insert* operations while $d = 8$ or 16. However, the vector units in the CPU cannot supply as much SIMD processing capability as in the GPU. Further, according to the previous research [6], moving vector operations from the CPU to the integrated GPU can obtain both performance improvement and energy efficiency. Therefore, in this paper we focus on utilizing

Algorithm 19 Update the root node of a non-empty max-d-heap.

```
1:  function UPDATE-KEY(*heap, d, n, newv)
2:      offset ← d − 1                          ▷ offset of the implicit storage
3:      i ← 0                                   ▷ the root node index
4:      v ← newv                                ▷ the root node value
5:      while di + 1 < n do                     ▷ if the first child is existed
6:          ⟨maxi, maxv⟩ ← FIND-MAXCHILD(*heap, d, n, i)
7:          if maxv > v then                    ▷ compare
8:              heap[i + offset] ← maxv         ▷ swap
9:              i ← maxi
10:         else                                ▷ the heap property is satisfied
11:             break
12:         end if
13:     end while
14:     heap[i + offset] ← v
15:     return
16: end function
```

Algorithm 20 Find the maximum child node of a given parent node.

```
1:  function FIND-MAXCHILD(*heap, d, n, i)
2:      offset ← d − 1
3:      starti ← di + 1                         ▷ the first child index
4:      stopi ← MIN(n − 1, di + d)              ▷ the last child index
5:      maxi ← starti
6:      maxv ← heap[maxi + offset]
7:      for i = starti + 1 to stopi do
8:          if heap[i + offset] > maxv then
9:              maxi ← i
10:             maxv ← heap[maxi + offset]
11:         end if
12:     end for
13:     return ⟨maxi, maxv⟩
14: end function
```

GPU-style vector processing but not SSE/AVX instructions.

However, other operations, in particular the *compare-and-swap* operations, cannot obtain benefit from the SIMD units because they only need one single thread, which is far from saturating the SIMD units. And the off-chip memory bandwidth is also wasted because one expensive off-chip memory transaction only stores one entry (lines 8 and 14 in the Algorithm 19). Further, the rest threads are waiting for the single thread's time-consuming off-chip transaction to finish. Even though the single thread store has a chance to trigger a cache write hit, the very limited cache in the throughput-oriented processors can easily be polluted by the massively concurrent threads. Thus the single thread task should always be avoided.

Therefore, to maximize the performance of the d-heap operations, we consider

two design objectives: (1) maximizing throughput of the large amount of the SIMD units for faster *find-maxchild* operations, and (2) minimizing negative impact from the single-thread *compare-and-swap* operations.

ad-heap Data Structure

Because the GPUs are designed for the wide SIMD operations and the CPUs are good at high performance single-thread tasks, the heterogeneous processors have a chance to become ideal platforms for operations with different characteristics of parallelism. This thesis proposes *ad*-heap (*a*symmetric *d*-heap), a new heap data structure that can obtain performance benefits from both of the two types of cores in the heterogeneous processors.

Compared to the *d*-heaps, the *ad*-heap data structure introduces an important new component — an implicit bridge structure. The bridge structure is located in the originally empty head part of the implicit *d*-heap. It consists of one node counter and one sequence of size $2h$, where h is the height of the heap. The sequence stores the index-value pairs of the nodes to be updated in different levels of the heap, thus at most h nodes are required. If the space requirement of the bridge is larger than the original head part of size $d-1$, the head part can be easily extended to $md+d-1$ to guarantee that each group of the child nodes starts from an aligned cache block, where m is a natural number and equal to $\lceil 2(h+1)/d \rceil - 1$. Figure 4.9 shows the layout of the *ad*-heap data structure.

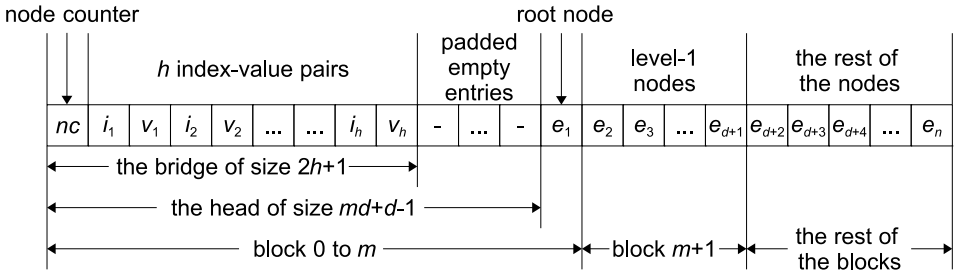


Figure 4.9: The layout of the *ad*-heap data structure.

ad-heap Operations

The corresponding operations of the *ad*-heap data structure are redesigned as well. Again, for simplicity and without loss of generality, we only consider the *update-key* operation described in Algorithm 19.

Before the *update-key* operation starts, the bridge is constructed in the on-chip scratchpad memory of a GPU and the node counter is initialized to zero. Then in each iteration (lines 6–12 of the Algorithm 19), a group of lightweight SIMD threads in the GPU simultaneously execute the *find-maxchild* operation (i.e., in parallel load at most d child nodes to the scratchpad memory and run the *streaming reduction* scheme to find the index and the value of the maximum child node). After each *find-maxchild* and *compare* operation, if a *swap* operation is needed, one of the SIMD

threads adds a new index-value pair (index of the current parent node and value of the maximum child node) to the bridge and updates the node counter. If the current level is not the last level, the new value of the child node can be stored in a register and be reused as the parent node of the next level. Otherwise, the single SIMD thread stores the new indices and values of both of the parent node and the child node to the bridge. Because the on-chip scratchpad memory is normally two orders of magnitude faster than the off-chip memory, the cost of the single-thread operations is negligible. When all iterations are finished, at most $2h + 1$ SIMD threads store the bridge from the on-chip scratchpad memory to the continuous off-chip memory by $\lceil (2h+1)/w \rceil$ off-chip memory transactions. The single program multiple data (SPMD) pseudocode is shown in Algorithm 21. Here we do not give out parallel pseudocode of the *find-maxchild* operation, since it is very similar to reduction operation described in Algorithm 4. After the bridge is dumped, a signal object is transferred to the GPU-CPU queue.

Triggered by the synchronization signal from the queue, one of the CPU cores sequentially loads the entries from the bridge and stores them to the real heap space in linear time. Note that no data transfer, address mapping or explicit coherence maintaining is required due to the unified memory space with cache coherence. And because the entries in the bridge are located in continuous memory space, the CPU cache system can be efficiently utilized. When all entries are updated, the whole *update-key* operation is completed. The pseudocode of the CPU workload in the *update-key* operation is shown in Algorithm 22.

Refer to the command queue in the OpenCL specification and the architected queueing language (AQL) in the HSA design, we list the pseudocode of the *update-key* operation in Algorithm 23.

We can see that although the overall time complexity is not reduced, the two types of compute units more focus on the off-chip memory behaviors that they are good at. We can calculate that the number of the GPU off-chip memory access needs $hd/w+(2h+1)/w$ transactions instead of $h(d/w+1)$ in the *d*-heap. For example, given a 7-level 32-heap and set w to 32, the *d*-heap needs 14 off-chip memory transactions while the *ad*-heap only needs 8. Since the cost of the off-chip memory access dominates execution time, the practical GPU performance can be improved significantly. Further, from the CPU perspective, all read transactions are from the bridge in continuous cache blocks and all write transactions only trigger non-time-critical cache write misses to random positions. Therefore the CPU workload performance can also be expected to be good.

ad-heap Simulator

From the perspective of the programming model, synchronization mechanism among compute units is redefined. Recently, several CPU-GPU fast synchronization approaches [44, 108, 127] have been proposed. In this section, we implement the *ad*-heap operations through the synchronization mechanism designed by the HSA Foundation. According to the current HSA design [108], each compute unit executes its task and sends a signal object of size 64 Byte to a low-latency shared memory queue when it has completed the task. Thus with HSA, CPUs and GPUs can queue tasks to each other and to themselves. Further, the communications can be dispatched in the user

Algorithm 21 The SPMD GPU workload in the *update-key* operation of the *ad*-heap.

```
1: function GPU-WORKLOAD(*heap, d, n, h, newv)
2:     tid ← GET-THREAD-LOCALID()
3:     i ← 0
4:     v ← newv
5:     *bridge ← SCRATCHPAD-MALLOC(2h + 1)
6:     if tid = 0 then
7:         bridge[0] ← 0                                  ▷ initialize the node counter
8:     end if
9:     while di + 1 < n do
10:        ⟨maxi, maxv⟩ ← FIND-MAXCHILD(*heap, d, n, i)
11:        if maxv > v then
12:            if tid = 0 then                            ▷ insert a index-value pair
13:                bridge[2 * bridge[0] + 1] ← i
14:                bridge[2 * bridge[0] + 2] ← maxv
15:                bridge[0] ← bridge[0] + 1
16:            end if
17:            i ← maxi
18:        else
19:            break
20:        end if
21:    end while
22:    if tid = 0 then                                    ▷ insert the last index-value pair
23:        bridge[2 * bridge[0] + 1] ← i
24:        bridge[2 * bridge[0] + 2] ← v
25:        bridge[0] ← bridge[0] + 1
26:    end if
27:    if tid < 2h + 1 then                               ▷ dump the bridge to off-chip
28:        heap[tid] ← bridge[tid]
29:    end if
30:    return
31: end function
```

mode of the operating systems, thus the traditional "GPU kernel launch" method (through the operating system kernel services and the GPU drivers) is avoided and the CPU-GPU communication latency is significantly reduced. Figure 4.10 shows an example of the shared memory queue.

Because the HSA programming tools for the heterogeneous processor hardware described in this section are not currently available yet, we conduct experiments on simulated heterogeneous processor platforms composed of real standalone CPUs and GPUs. The *ad*-heap simulator has two stages:

(1) **Pre-execution stage**. For a given input list and a size d, we first count the number of the *update-key* operations and the numbers of the subsequent *find-maxchild* and *compare-and-swap* operations by pre-executing the work through the d-heap on the CPU. We write N_u, N_f, N_c and N_s to denote the numbers of the *update-*

Algorithm 22 The CPU workload in the *update-key* operation of the *ad*-heap.

```
1: function CPU-WORKLOAD(*heap, d, n, h)
2:     m ← ⌈2(h + 1)/d⌉ − 1
3:     offset ← md + d − 1
4:     *bridge ← *heap
5:     for i = 0 to bridge[0] − 1 do
6:         index ← bridge[2 * i + 1]
7:         value ← bridge[2 * i + 2]
8:         heap[index + offset] ← value
9:     end for
10:    return
11: end function
```

Algorithm 23 The control process of the *update-key* operation.

```
1: function UPDATE-KEY(*heap, d, n, h, newv)
2:     QCtoG ← CREATE-QUEUE()
3:     QGtoC ← CREATE-QUEUE()
4:     Gpkt ← GPU-WORKLOAD(*heap, d, n, h, newv)
5:     Cpkt ← CPU-WORKLOAD(*heap, d, n, h)
6:     QUEUE_DISPATCH_FROM_CPU(QCtoG, Gpkt)
7:     QUEUE_DISPATCH_FROM_GPU(Gpkt, QGtoC, Cpkt)
8:     return
9: end function
```

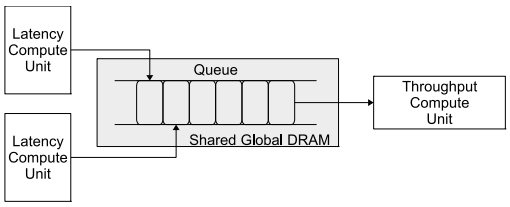


Figure 4.10: A shared memory queue.

key operations, *find-maxchild* operations, *compare* operations and *swap* operations, respectively. Although the N_f and the N_c are numerically equivalent, we use two variables for the sake of clarity.

(2) **Simulation stage**. Then we execute exactly the same amount of work with the *ad*-heap on the CPU and the GPU. The work can be split into three parts:

- The CPU part reads the entries in N_u bridges (back from the GPU) and writes $N_u(N_s + 1)$ values to the corresponding entry indices. This part takes T_{cc} time on the CPU.

- To simulate the CPU-GPU communication mechanism in the HSA design, the CPU part also need to execute signal object sends and receives. We use a lockless

multi-producer single-consumer (MPSC) queue programming tool in the DKit C++ Library [19] (based on multithread components in the Boost C++ Libraries [1]) for simulating the heterogeneous processor queueing system. To meet the HSA standard [90], our packet size is set to 64 Byte with two 4 Byte flags and seven 8 Byte flags. Further, packing and unpacking time is also included. Because each GPU core (and also each GPU) needs to execute multiple thread groups (thread blocks in the CUDA terminology or work groups in the OpenCL terminology) in parallel for memory latency hiding, we use 16 as a factor for the combined thread groups. Therefore, $2N_u/16$ push/pop operation pairs are executed for N_u CPU to GPU communications and the same amount of GPU to CPU communications. We record this time as T_{cq}.

- The GPU part executes N_f *find-maxchild* operations and N_c *compare* operations and writes N_u bridges from the on-chip scratchpad memory to the off-chip global shared memory. This part takes T_{gc} time on the GPU.

After simulation runs, we use overlapped work time on the CPU and the GPU as execution time of the *ad*-heap since the two types of cores are able to work in parallel. Thus the final execution time is the longer one of $T_{cc} + T_{cq}$ and T_{gc}.

Because of the features of the heterogeneous processors, costs of device/host memory copy and GPU kernel launch are not included in our timer. Note that because we use both the CPU and the GPU separately, the simulated heterogeneous processor platform is assumed to have accumulated off-chip memory bandwidths of the both processors. Moreover, we also assume that the GPU supports the device fission function defined in the OpenCL 1.2 specification and cores in the current GPU devices can be used as sub-devices which are more like the GPUs in the HSA design. Thus one CPU core and one GPU core can cooperate to deal with one *ad*-heap. The simulator is programmed in C++ and CUDA/OpenCL.

4.6.3 Performance Evaluation

Testbeds

To benchmark the performance of the *d*-heaps and the *ad*-heap, we use two representative machines: (1) a laptop system with an AMD A6-1450 APU, and (2) a desktop system with an Intel Core i7-3770 CPU and an nVidia GeForce GTX 680 discrete GPU. Their specifications are listed in Tables B.5 and B.6, respectively.

Benchmark and Datasets

We use a heap-based batch *k*-selection algorithm as benchmark of the heap operations. Given a list set consists of a group of unordered sub-lists, the algorithm finds the *k*th smallest entry from each of the sub-lists in parallel. One of its applications is batch *k*NN search in large-scale concurrent queries. In each sub-list, a max-heap of size k is constructed on the first *k* entries and its root node is compared with the rest of the entries in the sub-list. If a new entry is smaller, an *update-key* operation (i.e., the root node update and the heap property reconstruction) is triggered. After traversing all

entries, the root node is the *k*th smallest entry and the heap contains the k smallest entries of the input sub-list.

In our *ad*-heap implementation, we execute heapify function (i.e., the first construction of the heap) on the GPU and the root node comparison operations (i.e., to decide whether an *update-key* operation is required) on the CPU. Besides the execution time described in the *ad*-heap simulator, the execution time of the above two operations are recorded in our timer as well.

According to capacity limitation of the GPU device memory, we set sizes of the list sets to 2^{28} and 2^{25} on the two machines, respectively, data type to 32-bit integer (randomly generated), size of each sub-list to the same length l (from 2^{11} to 2^{21}), and k to $0.1l$.

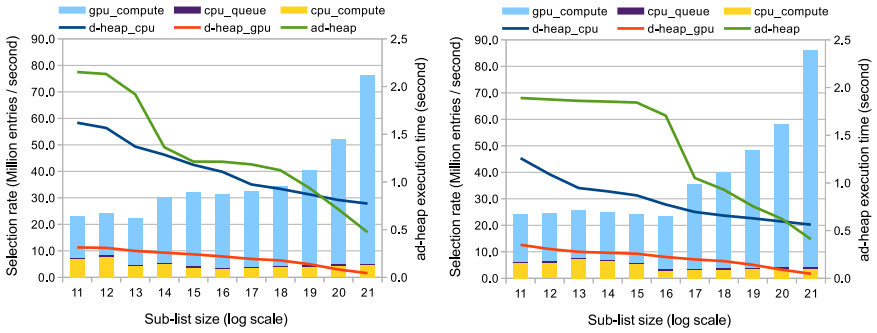
Experimental Results

Primary Y-axis-aligned line graphs in Figures 4.11(a)–(e) and 4.12(a)–(e) show the selection rates of the *d*-heaps (on the CPUs and the GPUs) and the *ad*-heap (on the simulators) over the different sizes of the sub-lists and d values on the machine 1 and the machine 2, respectively. In all tests, all cores of the CPUs are utilized. We can see that for the performance of the *d*-heaps in all groups, the multicore CPUs are almost always faster than the GPUs, even when the larger d values significantly reduce throughputs of the CPUs. Thus, for the conventional *d*-heap data structure, the CPUs are still better choices in the heap-based k-selection problem. For the *ad*-heap, the fastest size d is always 32. On one hand, the smaller d values cannot fully utilize computation and bandwidth resources of the GPUs. On the other hand, the larger d values lead to much more data loading but do not bring the same order of magnitude shallower heaps.

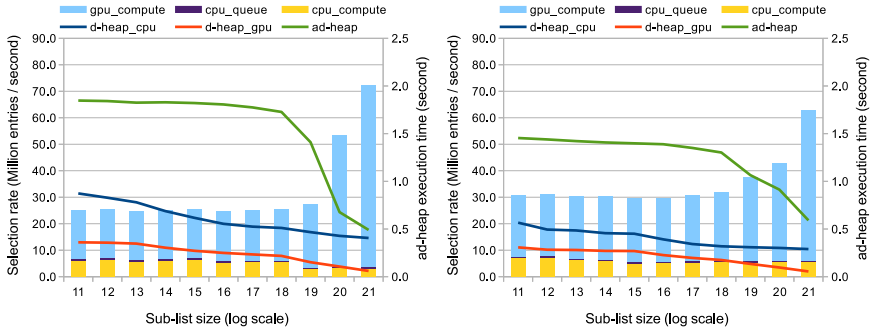
Secondary Y-axis-aligned stacked columns in Figures 4.11(a)–(e) and 4.12(a)–(e) show the execution time of the three parts (CPU compute, CPU queue and GPU compute) of the *ad*-heap simulators. On the machine 1, the execution time of the GPU compute is always longer than the total time of the CPU work, because the raw performance of the integrated GPU is relatively too low to accelerate the *find-maxchild* operations and the memory sub-system in the APU is not completely designed for the GPU memory behaviors. On the machine 2, the ratio of CPU time and GPU time is much more balanced (in particular, while $d = 32$) due to the much stronger discrete GPU.

Figures 4.11(f) and 4.12(f) show aggregated performance numbers include the best results in the former 5 groups and the optimal scheduling method that runs the fastest *d*-heaps on the CPUs and the GPUs in parallel, respectively. In these two sub-figures, we can see that the *ad*-heap obtains up to 1.5x and 3.6x speedup over the optimal scheduling method when the d value is equal to 32 and the sub-list size is equal to 2^{18} and 2^{19}, respectively. Note that the optimal scheduling method is also assumed to utilize accumulated off-chip memory bandwidths of the both processors.

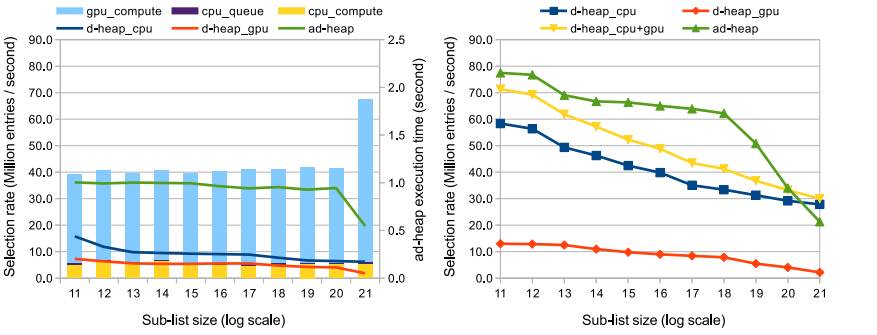
We can see that among all the candidates, only the *ad*-heap maintains relatively good performance stabilities while problem size grows. The performance numbers support our *ad*-heap design that gets benefits from main features of the two types of cores while the CPU *d*-heaps suffer with wider *find-maxchild* operations and the GPU *d*-heaps suffer with more single-thread *compare-and-swap* operations.



(a) $d = 8$ (b) $d = 16$

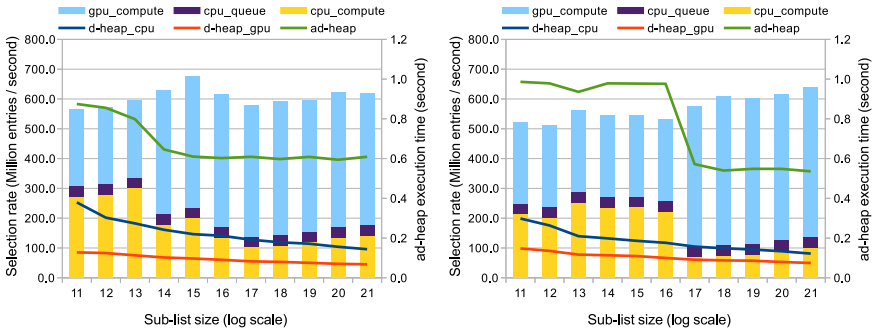


(c) $d = 32$ (d) $d = 64$

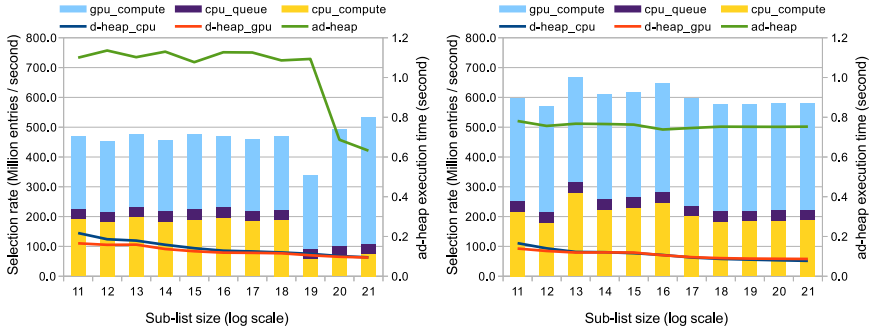


(e) $d = 128$ (f) $aggregated\ results$

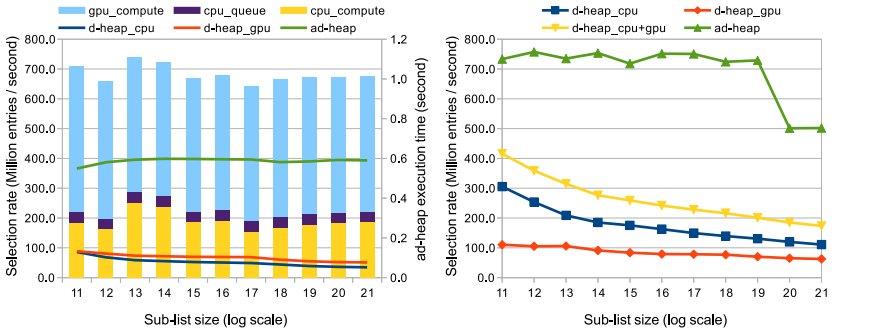
Figure 4.11: Selection rates and *ad*-heap execution time over different sizes of the sub-lists on the machine 1 (i.e., the one with AMD CPU and GPU). The line-shape data series is aligned to the primary Y-axis. The stacked column-shape data series is aligned to the secondary Y-axis.



(a) $d = 8$ (b) $d = 16$



(c) $d = 32$ (d) $d = 64$



(e) $d = 128$ (f) $aggregated\ results$

Figure 4.12: Selection rates and *ad*-heap execution time over different sizes of the sub-lists on the machine 2 (i.e., the one with Intel CPU and nVidia GPU). The line-shape data series is aligned to the primary Y-axis. The stacked column-shape data series is aligned to the secondary Y-axis.

4.6.4 Comparison to Previous Work

This section we proposes *ad*-heap, a new efficient heap data structure for the heterogeneous processors. We conducted empirical studies based on the theoretical

analysis. The experimental results showed that the *ad*-heap can obtain up to 1.5x and 3.6x performance of the optimal scheduling method on two representative machines, respectively. To the best of our knowledge, the *ad*-heap is the first fundamental data structure that efficiently leveraged the two different types of cores in the emerging heterogeneous processors through fine-grained frequent interactions between the CPUs and the GPUs. Further, the performance numbers also showed that redesigning data structure and algorithm is necessary for exposing higher computational power of the heterogeneous processors.

Compared with the prior work, our *ad*-heap not only takes advantage of reduced data movement cost but also utilizes computational power of the both types of cores. As shown in the previous section, we found that 8-heap is the best choice for the CPU and 32-heap is the fastest on the GPU, thus the optimal scheduling method should execute the best *d*-heap operations on both types of cores in parallel. However, our results showed that the *ad*-heap is much faster than the optimal scheduling method. Thus scheduling is not always the best approach, although task or data parallelism is obvious. Actually, in the *ad*-heap, the *find-maxchild* operation can be seen as a parallelizable stage of its higher-level operation *delete-max* or *update-key*. However, the *ad*-heap is different from the previous work because it utilizes advantages of the heterogeneous processors through frequent fine-grained interactions between the CPUs and the GPUs. If the two types of cores shared the last level cache, the *ad*-heap can naturally obtain benefits from heterogeneous prefetching, because the bridge and the nodes to be modified are already loaded to the on-chip cache by the GPUs, prior to writing back by the CPUs.

Because of the legacy CPU and GPU architecture design, in this section we choose focusing on an heterogeneous processor environment with separate last level cache sub-systems, as plotted in Figure 3.5(a). Conducting experiments on a shared last level cache heterogeneous processor like the one in Figure 3.5(b) can be an interesting future work. Additionally, our approach is different from the previous work since we see both GPUs and CPUs as compute units as well but not just prefetchers.

Part II

Sparse Matrix Representation

5. Existing Storage Formats

5.1 Overview

Because different sparse matrices have different sparsity structures, selection of representation (i.e., format) for a certain matrix may affect requirement of memory space and efficiency of algorithms running on it. As a result, research on sparse matrix representation attracts a lot of attention. This chapter introduces some widely used and recently developed sparse matrix representations.

Some representative formats are classified into three groups: (1) four basic formats mostly supported in many mathematical software packages, (2) three hybrid formats mainly developed for irregular matrices, and (3) eight extended formats constructed based on the basic formats.

5.2 Basic Storage Formats

5.2.1 Diagonal (DIA)

A sparse matrix containing nonzero entries only on its diagonals is called a *diagonal matrix*. Hence only entries on the diagonals are required to be saved in the so-called diagonal (DIA) format. For a matrix of size m^2, the primary diagonal has no more than m nonzero entries, and the other diagonals include fewer entries. Thus a diagonal matrix can be stored as arrays of its diagonals. Figure 5.1 shows a simple sparse matrix A_0 of size 6×6 containing 6 nonzero entries on its primary diagonal. So the matrix can be stored as an array `dia` with a label saying the array is diagonal "zero".

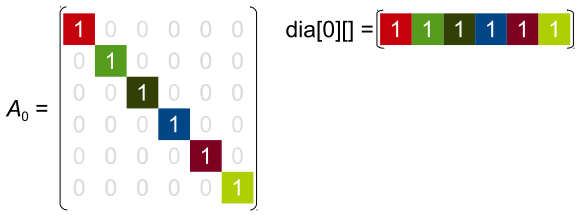


Figure 5.1: A sparse matrix and its DIA format.

Another simple example is shown in Figure 5.2. Sparse matrix A_1 contains 3 diagonals, thus can be saved as a two-dimensional array composed of three rows, and three labels $[-1, 0, 1]$ indicating relative distance to the primary diagonal. Because the diagonals at positions -1 and 1 has only 5 entries, one "empty" fill-in entry is padded for occupying memory space. Note that a gray block with an asterisk in figures of this chapter indicates a padded fill-in. We can see that the space complexity of the DIA format is $O(m)$, while the matrix has a fixed number of nonzero diagonals.

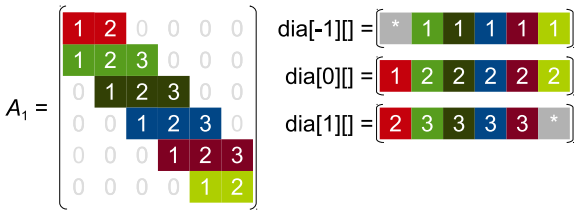


Figure 5.2: A sparse matrix and its DIA format.

However, the DIA format is designed only for sparse matrices dominated by diagonals. If some weak diagonals have very few nonzero entries, the DIA format may waste a large amount of memory space. Figure 5.3 shows an example matrix A_2. The diagonals at positions -2 and 2 only have one nonzero entry, respectively. Thus each of them wastes 5 memory units for padded fill-ins. In general, the DIA format may largely waste memory space for storing more irregular sparse matrices.

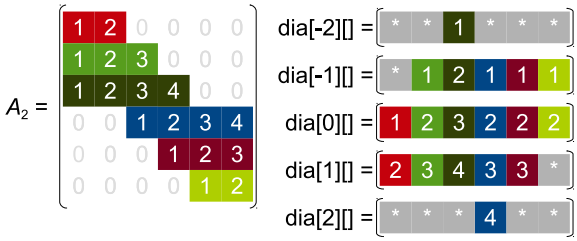


Figure 5.3: A sparse matrix and its DIA format.

5.2.2 ELLPACK (ELL)

Compared to the DIA format, the ELLPACK (or ELL for short) format is a bit more generic. It saves a sparse matrix of size m by n as two two-dimensional arrays, `col_idx` and `val`, of size m by $nnzr_{max}$, where $nnzr_{max}$ is the number of nonzero entries of the longest row of the matrix. The array `col_idx` stores column indices of the nonzero entries organized in rows, and the array `val` saves their values. The space complexity of the ELL format is $O(m\ nnzr_{max})$.

Figure 5.4 plots the ELL format in the row-major order of the matrix A_2. We can see that $nnzr_{max} = 4$ in this case, whereby the size of `col_idx` and `val` is 6×4. Because the 1st, 2nd, 5th and 6th rows have fewer nonzero entries compared to the longest row, some fill-ins are padded to keep arrays `col_idx` and `val` rectangular. Note that the ELL data can also be stored in the column-major order for aligned memory access.

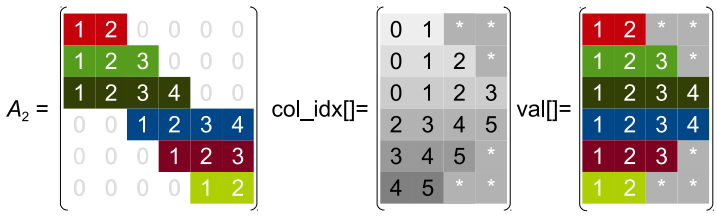


Figure 5.4: A sparse matrix and its ELL format in the row major order.

Unfortunately, for matrices with a relatively long row, the ELL format may need many fill-ins. Figure 5.5 gives an example matrix A_3. Because the 3rd row is much longer than the others, many fill-ins are explicitly stored thus waste memory space.

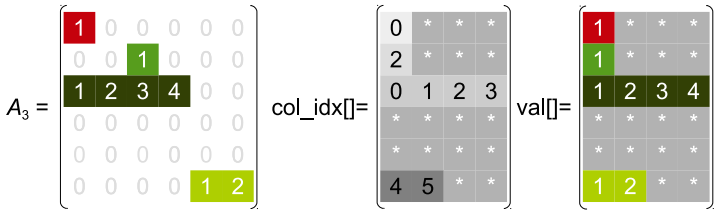


Figure 5.5: A sparse matrix and its ELL format in the row-major order.

5.2.3 Coordinate (COO)

A one-dimensional form of the Coordinate (or COO for short) format has been introduced in the very first example of small colored balls in Section 2.1.1 of the thesis. The term *coordinate* indicates row indices and column indices of the nonzero entries of a sparse matrix. Thus a matrix is stored in the COO format, if each of its entries contains a triple of row index, column index and value. To group all components of triples together, three separate arrays `row_idx`, `col_idx` and `val` of size nnz are constructed, where nnz is the number of nonzero entries of the sparse matrix. The space complexity of the COO format is $O(nnz)$. Unlike the DIA and the ELL formats, the COO format does not require storing any fill-ins. Figure 5.6 shows the COO format of the matrix A_3.

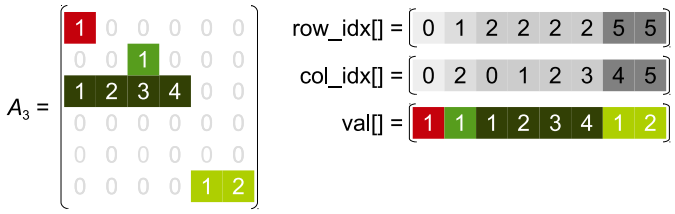


Figure 5.6: A sparse matrix and its COO format.

However, the COO format still stores some redundant row index information, if a row contains more than one nonzero entry. In Figure 5.6, we can see that the `row_idx` array stores four '2's for the nonzero entries in the 3rd row, and two '5's for the 6th row. Thus despite its generality, the COO format may waste memory space in common cases.

5.2.4 Compressed Sparse Row (CSR)

To maximize space efficiency, the Compressed Sparse Row (CSR) format[1] has been designed. For a sparse matrix, the CSR format consists of three arrays: (1) `row_ptr` array of size $m + 1$ saving the starting and ending pointers of the nonzero entries of the rows, (2) `col_idx` array of size nnz stores column indices of the nonzeros, and (3) `val` array of size nnz stores values of the nonzero entries. Hence the overall space complexity of the CSR format is $O(m + nnz)$. While $m < nnz$, the CSR format uses memory space more efficiently than the COO format.

Figure 5.7 shows the CSR representation of the example matrix A_3. We can see that the 3rd entry and the 4th entry of the array `row_ptr` are 2 and 6, respectively. This indicates the 3rd row of the matrix contains 4 nonzero entries.

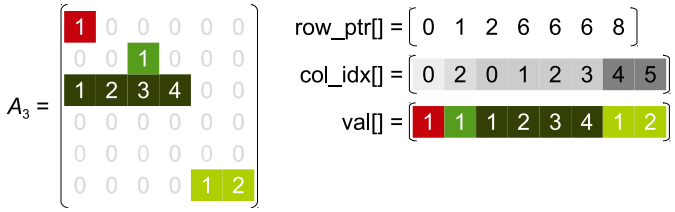


Figure 5.7: A sparse matrix and its CSR format.

Because the CSR format does not store redundant information, it is the most widely used format for sparse matrices. As a result, Sparse BLAS algorithms for matrices in the CSR format are basic functions of a sparse matrix library. Actually,

[1]The CSR format is also known as the Compressed Row Storage (CRS) format [156], and is equal to the Yale format [72] as well.

CSR and COO formats can be further compressed [101, 172, 183]. But description of those methods is beyond the scope of this chapter.

5.3 Hybrid Storage Formats

5.3.1 Hybrid (HYB)

Hybrid formats store a sparse matrix as a set of more than one basic formats. By utilizing hybrid formats, a very irregular sparse matrix can be stored as multiple regular ones, thus may be more friendly to parallel processing. Bell and Garland [21] proposed the first hybrid format called HYB for throughput-oriented processors in particular GPUs. Their method is established on an observation that some very long rows destroy load balancing of massive parallel chips, where each compute unit is responsible for computations on one row of a matrix. So they divide a sparse matrix into two submatrices: one is stored in the ELL format and the other is stored in the COO format, as shown in Figure 5.8. We can imagine that the matrix A_4 uses a large amount of memory if it uses the ELL format for all nonzero entries. But separately storing the nonzero entries except the last three entries of the 3rd row in the ELL format and the three entries in the COO format saves the padded fill-ins in the pure ELL format.

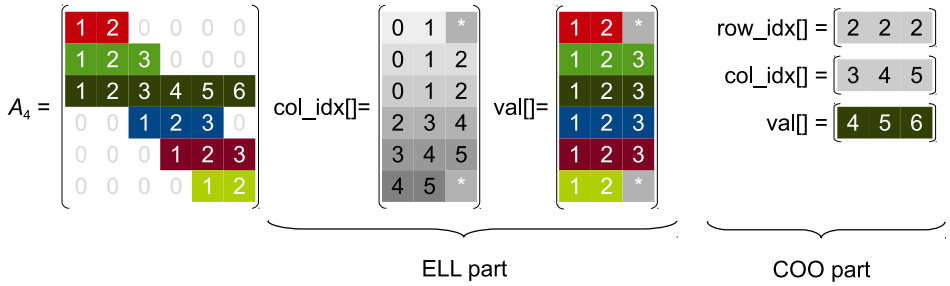


Figure 5.8: A sparse matrix and its HYB format.

5.3.2 Cocktail

Su and Keutzer proposed a more aggressive hybrid format called Cocktail [170]. It constructed a framework that analyzes the sparsity structure of an input matrix and stores it in a combination of 3 categories (i.e., diagonal based formats, flat formats and block based formats) of 9 formats. Figure 5.9 shows that the matrix A_4 can be saved more efficiently if its ELL part is saved in the DIA format.

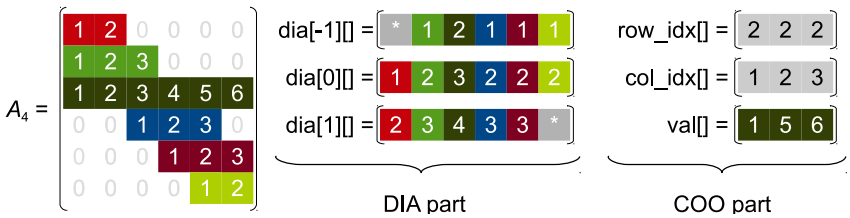


Figure 5.9: A sparse matrix and its Cocktail format.

5.3.3 SMAT

However, given an irregular sparse matrix, selecting the best combination is NP-hard. Li et al. [115] proposed a machine learning based autotuner trained by 2373 matrices in the University of Florida Sparse Matrix Collection [56] for deciding a combination of formats for a new input matrix. Similar to the SMAT, Sedaghati et al. [160] constructed machine learning classifiers for automatic selection of the best format for a given input on a target device.

5.4 Sliced and Blocked Storage Formats

5.4.1 Sliced ELL (SELL)

As a variant of the ELL format, the sliced ELL (SELL) format [135] splits nb rows into a block and stores nonzero entries in the column-major order in each block. Figure 5.10 shows an example matrix A_5. When nb is set to 3, A_5 is divided into 2 blocks containing 8 and 6 nonzero entries, respectively. We can see that the slicing method of the SELL format may bring better cache locality, compared to the ELL format. Thus for some relatively regular sparse matrices such as the one generated by the recent high performance conjugate gradient (HPCG) benchmark [62], SELL can bring good performance [196]. However, both of the ELL and the SELL formats waste memory space if a row is noticeably longer than the others.

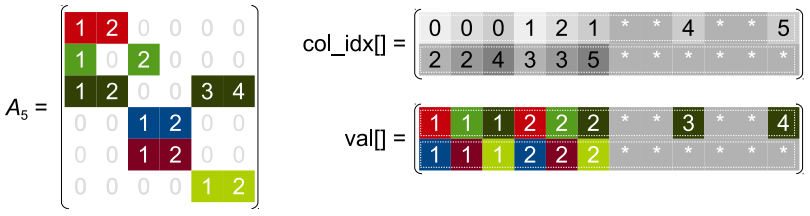


Figure 5.10: A sparse matrix and its SELL format.

5.4.2 Sliced COO (SCOO)

Similar to the slicing method of the SELL format, the COO format can be stored in a sliced fashion as well. Dang and Schmidt designed the sliced COO (SCOO) format [54] and showed its effectiveness. The method also splits the input matrix into blocks of nb rows and stores their entries with triples in the column-major order. Figure 5.11 shows the SCOO format for the matrix A_5. Because the blocks may contain different number of nonzero entries, a new array `slice_ptr` of size $\lceil m/nb \rceil + 1$ is introduced for storing the starting and ending pointers of the nonzero entries of the row blocks.

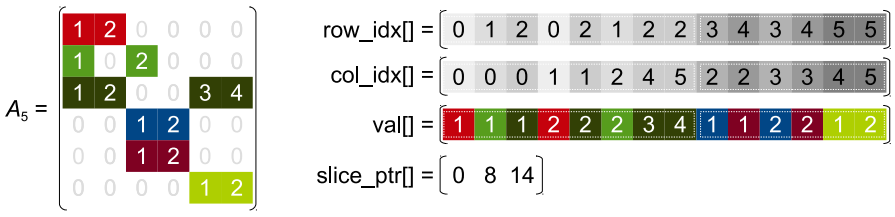


Figure 5.11: A sparse matrix and its SCOO format.

5.4.3 Blocked COO (BCOO)

Because some sparse matrices such as the ones generated from some finite element meshes naturally have dense local block structures, using small dense block as the basic unit can reduce space cost of storing indices [177, 178, 179]. Suppose that a sparse matrix is saved as submatrices of size $nb_1 \times nb_2$, at most $(m \times n)/(nb_1 \times nb_2)$ row/column indices are required. If the indices are stored in the COO style, the so-called blocked COO (BCOO) format [188] is established. Figure 5.12 shows the BCOO representation of the matrix A_5. We can see that the blocks of size 2×2 may contain padded fill-ins to maintain an equally spaced storage, and the arrays `row_idx` and `col_idx` can be much shorter than their counterparts in the standard COO format.

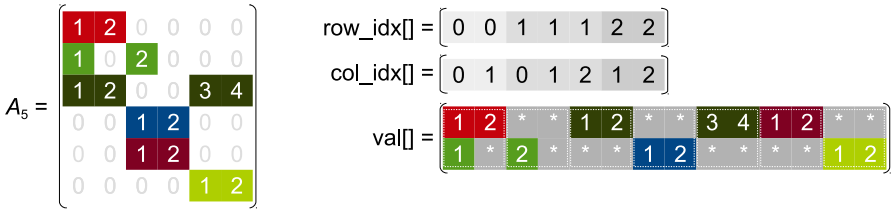


Figure 5.12: A sparse matrix and its BCOO format of 2×2 dense blocks.

5.4.4 Blocked CSR (BCSR)

In analogy to the difference between the COO and the CSR, the indices of the blocks of the BCOO format can be saved in the CSR fashion. Thus the blocked CSR (BCSR) format [45] has been designed. In figure 5.13, we can see that `row_ptr` is used instead of the `row_idx` array for saving duplicated block indices.

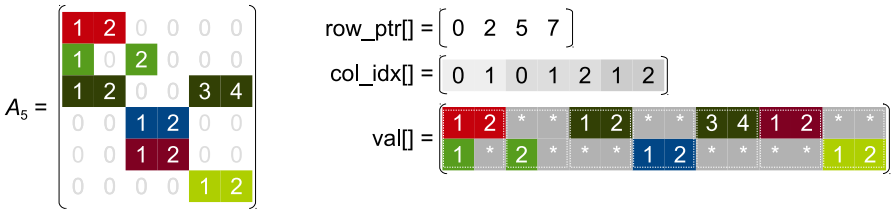


Figure 5.13: A sparse matrix and its BCSR format.

5.4.5 Blocked Compressed COO (BCCOO)

Yan et al. further compressed the BCOO format by introducing a `bit_flag` array in their blocked compressed COO (BCCOO) format [188]. This representation uses one 1-bit `TRUE` or `FALSE` to replace each entry in the `row_idx` array in the BCOO format. The basic idea is that only the ending point for the block units in one row is required to be labelled as `FALSE`, and the others are labelled as `TRUE`. The bit-wise information is actually enough to determine the distribution of row blocks. Assume that an integer row index requires 32 bits, the bit-wised label only needs 1/32 memory space. However, the BCCOO format still explicitly stores padded fill-ins, as the BCOO format does. Figure 5.14 shows an example of the BCCOO format.

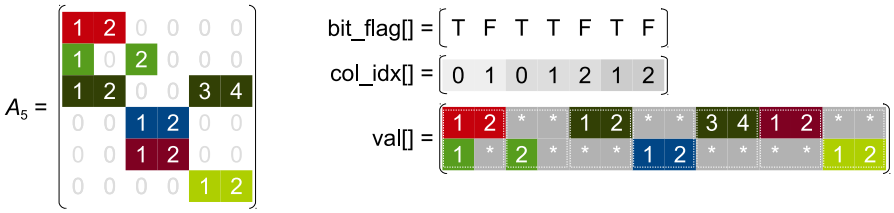


Figure 5.14: A sparse matrix and its BCCOO format.

5.4.6 Blocked Row-Column (BRC)

Motivated by minimizing padded fill-ins in the blocked formats such as the BCOO, the BCSR and the BCCOO, Ashari et al. proposed the blocked row-column (BRC) format [8]. This method first sorts rows of the input matrix in a descending order in

terms of the number of nonzero entries in each row, then aligns the entries to their left as the ELL format does, finally stores entries in dense blocks in the column-major order for better cache locality. Because of the sorting, a new array called `perm` is introduced for storing permutation, i.e., original row positions. Figure 5.15 shows an example. We can see that the 3rd row is longer than the others thus be located to the front.

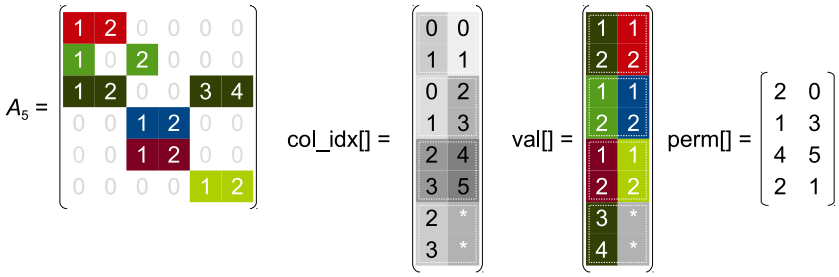


Figure 5.15: A sparse matrix and its BRC format.

5.4.7 Adaptive CSR (ACSR)

Because CSR is the most used format in many mathematical software packages, Ashari et al. proposed a work called adaptive CSR (ACSR) [7] that adds extra information to help balanced computations. Binning rows of a sparse matrix according to their lengths is the main idea of the ACSR format. Figure 5.16 shows an example. The rows of the matrix A_5 is assigned to two bins of size 5 and 1, respectively. The first bin stores rows of size 2, and the second bin contains the 3rd row of length 4. Hence computations on the rows in each bin may consume roughly the same execution time for better load balancing. One obvious advantage of this work is that the CSR data can be kept intact in the ACSR format.

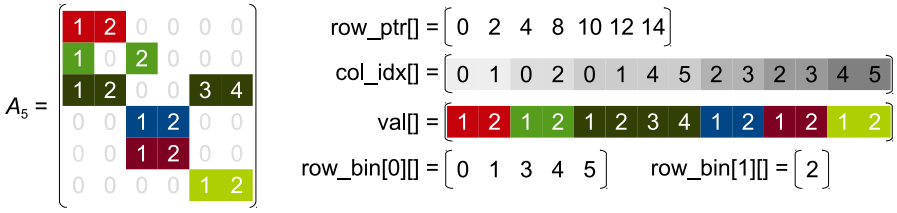


Figure 5.16: A sparse matrix and its ACSR format.

5.4.8 CSR-Adaptive

Greathouse and Daga proposed their CSR-Adaptive format [80, 50] that uses an array `row_block` instead of the binning information of the ACSR format for grouping rows of comparable length. In this format, contiguous rows are organized as row blocks if the sum of their lengths is no longer than the size of usable on-chip memory. The array `row_block` records the starting and ending points of row groups. Figure 5.17 gives an example that assumes each row block does not contain more than 4 nonzero entries. We can see that the 1st row block contains 2 rows, the 2nd one contains 1 row, and so on. If the row blocks have roughly the same number of nonzero entries, good load balancing can be expected on massively parallel devices. As the ACSR format, the CSR data is not changed in the CSR-Adaptive format.

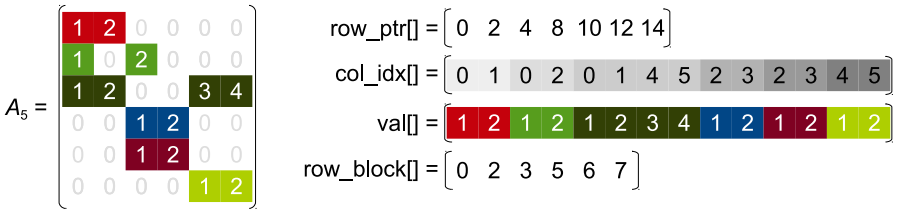


Figure 5.17: A sparse matrix and its CSR-Adaptive format.

6. The CSR5 Storage Format

6.1 Overview

In the previous chapter, we can see that many advanced formats have been proposed for a variety of purposes. However, they may need not cheap format conversion cost, if the input matrix is stored in a basic format such as CSR. The conversion cost is mainly from the expensive structure-dependent parameter tuning of a storage format. For example, some block based formats [33, 35, 45, 177, 178, 188] such as BCCOO require finding a good 2D block size. Moreover, some hybrid formats [21, 170] such as HYB may need completely different partitioning parameters for distinct input matrices.

To avoid the format conversion overhead, the ACSR [7] and the CSR-Adaptive [80, 50] directly use the CSR data. However, they may provide very low performance for irregular matrices due to unavoidable load imbalance. Furthermore, none of them can avoid an overhead from preprocessing, since certain auxiliary data for the basic CSR format have to be generated.

Therefore, to be practical, an efficient format must satisfy two criteria: (1) it should limit format conversion cost by avoiding structure-dependent parameter tuning, and (2) it should support fast sparse matrix computations for both regular and irregular matrices.

6.2 Contributions

To meet these two criteria, this chapter proposes CSR5 (Compressed Sparse Row 5)[1], a new format directly extending the classic CSR format. The CSR5 format leaves one of the three arrays of the CSR format unchanged, stores the other two arrays in an in-place tile-transposed order, and adds two groups of extra auxiliary information. The format conversion from the CSR to the CSR5 merely needs two tuning parameters: one is hardware-dependent and the other is sparsity-dependent (but structure-independent). Because the added two groups of information are usually much shorter than the original three in the CSR format, very limited extra space is required. Furthermore, the CSR5 format is SIMD-friendly and thus can be easily implemented on all mainstream processors with the SIMD units. Because of the

[1] The reason we call the storage format CSR5 is that it has five groups of data, instead of three in the classic CSR.

structure-independence and the SIMD utilization, the CSR5-based sparse matrix algorithm such as SpMV can bring stable high throughput for both regular and irregular matrices.

6.3 The CSR5 format

6.3.1 Basic Data Layout

To achieve near-optimal load balance for matrices with any sparsity structures, we first evenly partition all nonzero entries to multiple 2D tiles of the same size. Thus when executing parallel SpMV operation, a compute core can consume one or more 2D tiles, and each SIMD lane of the core can deal with one column of a tile. Then the main skeleton of the CSR5 format is simply a group of 2D tiles. The CSR5 format has two tuning parameters: ω and σ, where ω is a tile's width and σ is its height. In fact, the CSR5 format *only has* these two tuning parameters.

Further, we need extra information to efficiently compute SpMV. For each tile, we introduce a tile pointer `tile_ptr` and a tile descriptor `tile_desc`. Meanwhile, the three arrays, i.e., row pointer `row_ptr`, column index `col_idx` and value `val`, of the classic CSR format are directly integrated. The only difference is that the `col_idx` data and the `val` data in each complete tile are in-place transposed (i.e., from row-major order to column-major order) for coalesced memory access from contiguous SIMD lanes. If the last entries of the matrix do not fill up a complete 2D tile (i.e., $nnz \bmod (\omega\sigma) \neq 0$), they just remain unchanged and discard their `tile_desc`.

In Figure 6.2, an example matrix A shown in Figure 6.1 of size 8×8 with 34 nonzero entries is stored in the CSR5 format. When $\omega = 4$ and $\sigma = 4$, the matrix is divided into three tiles including two complete tiles of size 16 and one incomplete tile of size 2. The arrays `col_idx` and `val` in the two complete tiles are stored in tile-level column-major order now. Moreover, only the first two tiles have `tile_desc`, since they are complete.

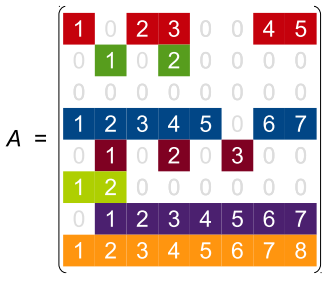


Figure 6.1: A sparse matrix A of size 8×8 including 34 nonzero entries.

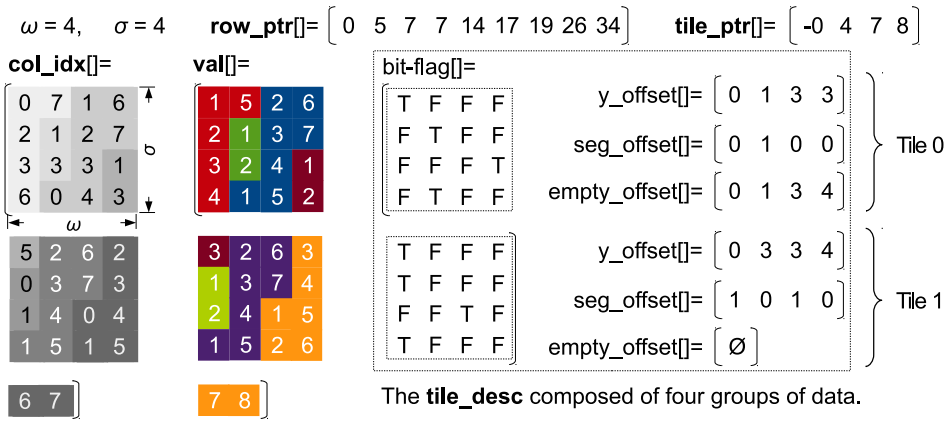


Figure 6.2: The CSR5 storage format of the sparse matrix A. The five groups of information include `row_ptr`, `tile_ptr`, `col_idx`, `val` and `tile_desc`.

6.3.2 Auto-Tuned Parameters ω and σ

Because the computational power of the modern multicore or manycore processors is mainly from the SIMD units, we design an auto-tuning strategy for high SIMD utilization.

First, the tile width ω is set to the size of the SIMD execution unit of the used processor. Then an SIMD unit can consume a 2D tile in σ steps without any explicit synchronization, and the vector registers can be fully utilized. For the double precision SpMV, we always set $\omega = 4$ for CPUs with 256-bit SIMD units, $\omega = 32$ for the nVidia GPUs, $\omega = 64$ for the AMD GPUs, and $\omega = 8$ for Intel Xeon Phi with 512-bit SIMD units. Therefore, ω can be automatically decided once the processor type used is known.

The other parameter σ is decided by a slightly more complex process. For a given processor, we consider its on-chip memory strategy such as cache capacity and prefetching mechanism. If a 2D tile of size $\omega \times \sigma$ can empirically bring better performance than using the other sizes, the σ is simply chosen. We found that the x86 processors fall into this category. For the double precision SpMV on CPUs and Xeon Phi, we always set σ to 16 and 12, respectively.

As for GPUs, the tile height σ further depends on the sparsity of the matrix. Note that the "sparsity" is not equal to "sparsity structure". We define "sparsity" to be the average number of nonzero entries per row (or nnz/row for short). In contrast, "sparsity structure" is much more complex because it includes 2D space layout of all nonzero entries.

On GPUs, we have several performance considerations on mapping the value nnz/row to σ. First, σ should be large enough to expose more thread-level local work and to amortize a basic cost of the segmented sum algorithm. Second, it should not be too large since a larger tile potentially generates more partial sums (i.e., entries to store to y), which bring higher pressure to last level cache write. Moreover, for

the matrices with large nnz/row, σ may need to be small. The reason is that once the whole tile is located inside a matrix row (i.e., only one segment is in the tile), the segmented sum converts to a fast reduction sum.

Therefore, for the nnz/row to σ mapping on GPUs, we define three simple bounds: r, s and t. The first bound r is designed to prevent a too small σ. The second bound s is used for preventing a too large σ. But when nnz/row is further larger than the third bound t, σ is set to a small value u. Then we have

$$\sigma = \begin{cases} r & \text{if} \quad nnz/\text{row} \leq r \\ nnz/\text{row} & \text{if} \quad r < nnz/\text{row} \leq s \\ s & \text{if} \quad s < nnz/\text{row} \leq t \\ u & \text{if} \quad t < nnz/\text{row}. \end{cases}$$

The three bounds, r, s and t, and the value u are hardware-dependent, meaning that for a given processor, they can be fixed for use. For example, to execute double precision SpMV on nVidia Maxwell GPUs and AMD GCN GPUs, we always set $<r, s, t, u> = <4, 32, 256, 4>$ and $<4, 7, 256, 4>$, respectively. As for future processors with new architectures, we can obtain the four values through some simple benchmarks during initialization, and then use them for later runs. So the parameter σ can be decided once the very basic information of a matrix and a low-level hardware are known.

Therefore, we can see that the parameter tuning time becomes negligible because ω and σ are easily obtained. This can save a great deal of preprocessing time.

6.3.3 Tile Pointer Information

The added tile pointer information `tile_ptr` stores the row index of the first matrix row in each tile, indicating the starting position for storing its partial sums to the vector y. By introducing `tile_ptr`, each tile can find its own starting position, allowing tiles to execute in parallel. The size of the `tile_ptr` array is $p + 1$, where $p = \lceil nnz/(\omega\sigma) \rceil$ is the number of tiles in the matrix. For the example in Figure 6.2, the first entry of Tile 1 is located in the $4th$ row of the matrix, and thus 4 is set as its tile pointer. To build the array, we binary search the index of the first nonzero entry of each tile on the `row_ptr` array. Lines 1–4 in Algorithm 24 show this procedure.

Recall that an empty row has exactly the same row pointer information as its first non-empty right neighbor row (see the second row in the matrix A in Figure 5.7). Thus for the non-empty rows with an empty left neighbor, we need a specific process (which is similar to lines 12–16 in Algorithm 30) to store their partial sums to correct positions in y. To recognize whether the specific process is required, we give a hint to the other parts (i.e., tile descriptor data) of the CSR5 format and the CSR5-based SpMV algorithm. Here we set an entry in `tile_ptr` to its negative value, if its corresponding tile includes any empty rows. Lines 5–12 in Algorithm 24 show this operation.

If the first tile has any empty rows, we need to store a -0 (negative zero) for it. To record -0, here we use unsigned 32- or 64-bit integer as data type of the `tile_ptr` array. Therefore, we have 1 bit for explicitly storing the sign and 31 or 63 bits for an index. For example, in our design, tile pointer -0 is represented as a binary style

Algorithm 24 Generating `tile_ptr`.

```
1: for tid = 0 to p in parallel do
2:     bnd ← tid × ω × σ
3:     tile_ptr[tid] ← BINARY_SEARCH(*row_ptr, bnd) −1
4: end for
5: for tid = 0 to p − 1 do
6:     for rid = tile_ptr[tid] to tile_ptr[tid + 1] do
7:         if row_ptr[rid] = row_ptr[rid + 1] then
8:             tile_ptr[tid] ← NEGATIVE(tile_ptr[tid])
9:             break
10:        end if
11:    end for
12: end for
```

'1000 ... 000', and tile pointer 0 is stored as '0000 ... 000'. To the best of our knowledge, the index of 31 or 63 bits is completely compatible to most numerical libraries such as Intel MKL. Moreover, reference implementation of the HPCG benchmark [62] also uses 32-bit signed integer for problem dimension no more than 2^{31} and 64-bit signed integer for problem dimension larger than that. Thus it is safe to save 1 bit as the empty row hint and the other 31 or 63 bits as a 'real' row index.

6.3.4 Tile Descriptor Information

Only having the tile pointer is not enough for a fast SpMV operation. For each tile, we also need four extra hints: (1) `bit_flag` of size $\omega \times \sigma$, which indicates whether an entry is the first nonzero of a matrix row, (2) `y_offset` of size ω used to further let each column know where the starting point to store its local partial sums is, (3) `seg_offset` of size ω used to accelerate the local segmented sum inside a tile, and (4) `empty_offset` of unfixed size (but no longer than $\omega \times \sigma$) constructed to help the partial sums to find correct locations in y if the tile includes any empty rows. The tile descriptor `tile_desc` is defined to denote a combination of the above four groups of data.

Generating `bit_flag` is straightforward. The procedure is very similar to lines 3–5 in Algorithm 30 that describes SpMV based on a segmented sum method. The main difference is that the bit flags are saved in column-major order, which matches the in-place transposed `col_idx` and `val`. Additionally, the first entry of each tile's `bit_flag` is set to `TRUE` for sealing the first segment from the top and letting 2D tiles to be independent from each other.

The array `y_offset` of size ω is used to help the columns in each tile knowing where the starting points to store their partial sums to y are. In other words, each column has one entry in the array `y_offset` as a starting point offset for all segments in the same column. We save a row index offset (i.e., relative row index) for each column in `y_offset`. Thus for the ith column in the tidth tile, by calculating `tile_ptr[`tid`] + y_offset[`i`]`, the column knows where its own starting position in y is. Thus the columns can work in a high degree of parallelism without waiting for a

synchronization. Generating y_offset is simple: each column counts the number of TRUEs in its previous columns' bit_flag array. Consider Tile 1 in Figure 6.2 as an example: because there are 3 TRUEs in the 1st column, the 2nd column's corresponding value in y_offset is 3. In addition, since there are in total 4 TRUEs in the 1st, 2nd and 3rd columns' bit_flag, Tile 1's y_offset[3] = 4. Algorithm 25 lists how to generate y_offset for a single 2D tile in an SIMD-friendly way.

Algorithm 25 Generating y_offset and seg_offset.

```
1: MALLOC(*tmp_bit, ω)
2: MEMSET(*tmp_bit, FALSE)
3: for i = 0 to ω − 1 in parallel do
4:     y_offset[i] ← 0
5:     for j = 0 to σ − 1 do
6:         y_offset[i] ← y_offset[i] + bit_flag[i][j]
7:         tmp_bit[i] ← tmp_bit[i] ∨ bit_flag[i][j]
8:     end for
9:     seg_offset[i] ← 1− tmp_bit[i]
10: end for
11: EXCLUSIVE_ALL-SCAN_SERIAL(*y_offset, ω)          ▷ Algorithm 7
12: SEGMENTED_SUM(*seg_offset, ω, *tmp_bit)          ▷ Algorithm 12
13: FREE(*tmp_bit)
```

The third array seg_offset of size ω is used for accelerating a local segmented sum in the workload of each tile. The local segmented sum is an essential step that synchronizes partial sums in a 2D tile (imagine multiple columns in the tile come from the same matrix row). In the previous segmented sum (or segmented scan) method [28, 40, 161, 63], the local segmented sum is complex and not efficient enough. Thus we use the method described in 4.4.4 introducing a seg_offset array for fast segmented sum.

To generate seg_offset, we let each column search its right neighbor columns and count the number of contiguous columns without any TRUEs in their bit_flag. Using Tile 0 in Figure 6.2 as an example, its 2nd column has one and only one right neighbor column (the 3rd column) without any TRUEs in its bit_flag. Thus the 2nd column's seg_offset value is 1. In contrast, because the other three columns (the 1st, 3rd and 4th) do not have any 'all FALSE' right neighbors, their values in seg_offset is 0. Algorithm 25 shows how to generate seg_offset using an SIMD-friendly method.

The seg_offset array is used for calculating fast segmented sum through an inclusive prefix-sum scan. See Section 4.4.4 for detailes.

The last array empty_offset occurs when and only when a 2D tile includes any empty rows (i.e., its tile pointer is negative). Because an empty row of a matrix has the same row pointer with its rightmost non-empty neighbor row (recall the second row in the matrix A in Figure 1), y_offset will record an incorrect offset for it. We correct for this by storing correct offsets for segments within a tile. Thus the length of empty_offset is the number of segments (i.e., the total number of TRUEs in bit_flag) in a tile. For example, Tile 0 in Figure 6.2 has 4 entries in its empty_−

`offset` since its `bit_flag` includes 4 `TRUE`s. Algorithm 26 lists the pseudocode that generates `empty_offset` for a tile that contains at least one empty row.

Algorithm 26 Generating `empty_offset` for the tidth tile.

```
1: len ← REDUCTION_SUM_PARALLEL(*bit_flag, ω)                ▷ Algorithm 4
2: MALLOC(*empty_offset, len)
3: eid ← 0
4: for i = 0 to ω − 1 do
5:     for j = 0 to σ − 1 do
6:         if bit_flag[i][j] = TRUE then
7:             ptr ← tid × ω × σ + i × σ + j
8:             idx ← BINARY_SEARCH(*row_ptr, ptr) −1
9:             idx ← idx − REMOVE_SIGN(tile_ptr[tid])
10:            empty_offset[eid] ← idx
11:            eid ← eid + 1
12:        end if
13:    end for
14: end for
```

6.3.5 Storage Details

To store the `tile_desc` arrays in a space-efficient way, we find upper bounds to the entries and utilize the bit-field pattern. First, since entries in `y_offset` store offset distances inside a 2D tile, they have an upper bound of $\omega\sigma$. So $\lceil \log_2(\omega\sigma) \rceil$ bits are enough for each entry in `y_offset`. For example, when $\omega = 32$ and $\sigma = 16$, 9 bits are enough for each entry. Second, since `seg_offset` includes offsets less than ω, $\lceil \log_2(\omega) \rceil$ bits are enough for an entry in this array. For example, when $\omega = 32$, 5 bits are enough for each entry. Third, `bit_flag` stores σ 1-bit flags for each column of a 2D tile. When $\sigma = 16$, each column needs 16 bits. So 30 (i.e., $9 + 5 + 16$) bits are enough for each column in the example. Therefore, for a tile, the three arrays can be stored in a compact bit-field composed of ω 32-bit unsigned integers. If the above example matrix has 32-bit integer row index and 64-bit double precision values, only around 2% extra space is required by the three newly added arrays.

The size of `empty_offset` depends on the number of groups of contiguous empty rows, since we only record one offset for the rightmost non-empty row with any number of empty rows as its left neighbors.

6.3.6 The CSR5 for Other Matrix Operations

Since we in-place transposed the CSR arrays `col_idx` and `val`, a conversion from the CSR5 to the CSR is required for doing other sparse matrix operations using the CSR format. This conversion is simply removing `tile_ptr` and `tile_desc` and transposing `col_idx` and `val` back to row-major order. Thus the conversion can be very fast. Further, since the CSR5 is a superset of the CSR, any entry accesses or slight changes can be done directly in the CSR5 format, without any need to convert it to

the CSR format. Additionally, some applications such as finite element methods can directly assemble sparse matrices in the CSR5 format from data sources.

6.4 Comparison to Previous Formats

As shown in the previous chapter, a great deal of work has been published on representing sparse matrices for higher SpMV performance. The **block-based sparse matrix construction** has received most attention [8, 33, 35, 45, 149, 177, 188] because of two main reasons: (1) sparse matrices generated by some real-world problems (e.g., finite element discretization) naturally have the block sub-structures, and (2) off-chip load operations may be decreased by using the block indices instead of the entry indices. However, for many matrices that do not exhibit a natural block structure, trying to extract the block information is time consuming and has limited effects. The BCCOO format is a representative. Its preprocessing stage works in a complex optimizing space consists of up to 10 groups of parameters and up to 442368 tuning possibilities. From our experiments, we can see that although it is much faster than any other formats, the real applicability may be extremely low because of exhaustive parameter tuning. The CSR5 format, in contrast, has only 2 parameters thus requires very low format conversion cost.

On the other hand, the **hybrid formats** [21, 170], such as HYB, have been designed for irregular matrices. However, higher kernel launch overhead and invalidated cache among kernel launches tend to decrease their overall performance. Moreover, it is hard to guarantee that every submatrix can saturate the whole device. In addition, some relatively simple operations such as solving triangular systems become complex while the input matrix is stored in two or more separate parts. The CSR5 format does not require such decomposition for good performance on computing irregular matrices.

Unlike the work of **automatic selection of the best format** from Li et al. [115] and Sedaghati et al. [160] that analyzes sparsity structure and then suggests one or more formats, the CSR5 format described in this work is insensitive to the sparsity structure of the input sparse matrix.

Moreover, to the best of our knowledge, the CSR5 is the only format that supports high throughput **cross-platform** sparse matrix computations on CPUs, nVidia GPUs, AMD GPUs and Xeon Phi at the same time. This advantage may simplify the development of scientific software for processors with massive on-chip parallelism. Chapter 8 will describe CSR5-based SpMV algorithm.

Part III

Sparse BLAS Algorithms

7. Level 1: Sparse Vector Operations

7.1 Overview

In Section 2.3, we can see that many Levels 2 and 3 sparse operations can directly use Level 1 routines as building blocks, thus their performance is crucial for all Sparse BLAS routines. This chapter main focuses on the sparse vector – sparse vector addition operation since it is used as part of the SpGEMM method described in Chapter 9.

7.2 Contributions

In this chapter, a heuristic approach for adding two sparse vectors is developed. Depending on the upper bound length of the sparse output vector, the approach maps the sparse input vectors to a heap method, a bitonic ESC method or a merge method.

7.3 Basic Method

7.3.1 Sparse Accumulator

Gilbert et al. [77] designed a fundamental tool called sparse accumulator, or SPA for short. The SPA conducts sparse operations on a dense array. One of its use scenarios is adding two sparse vectors: the nonzero entries of the 1st sparse vector are assigned to an all-zero dense vector of the same size, then the nonzero entries of the 2nd sparse vector are added to associated positions of the dense vector, finally all zero entries are removed from the dense vector to store nonzero ones only. Algorithm 27 shows this procedure.

7.3.2 Performance Considerations

We can see that the SPA method works in the space of a dense vector, thus may waste storage and compute resources for dealing with zero entries. In particular when the

Algorithm 27 Adding two sparse vectors using SPA.

```
1: function SPARSE_VECTOR_ADDITION_SPA(*in1, *in2, *out, len)
2:     MALLOC(*tmp, len)
3:     MEMSET(*tmp, 0)
4:     nnz_in1 ← in1.nnz
5:     nnz_in2 ← in2.nnz
6:     for i = 0 to nnz_in1 − 1 do
7:         tmp[in1.col_idx[i]] ← in1.val[i]
8:     end for
9:     for i = 0 to nnz_in2 − 1 do
10:        tmp[in2.col_idx[i]] ← tmp[in2.col_idx[i]] + in2.val[i]
11:    end for
12:    nnz_out ← 0
13:    for i = 0 to len do
14:        if tmp[i] ≠ 0 then
15:            out.col_idx[nnz_out] ← i
16:            out.val[nnz_out] ← tmp[i]
17:            nnz_out ← nnz_out + 1
18:        end if
19:    end for
20:    out.nnz ← nnz_out
21:    FREE(*tmp)
22: end function
```

input sparse vectors are short enough and the size of them is enough long.

7.4 CSR-Based Sparse Vector Addition

Here we develop three methods for adding two sparse vectors.

7.4.1 Heap Method

The heap method first creates an empty implicit index-value pair heap (or priority queue) of the upper bound size (i.e., the sum of the number of nonzero entries of the two input vectors) of the output vector. The heap can be located in the on-chip scratchpad memory and collects all candidate nonzero entries from the input vectors. Then the heap executes a heapsort-like operation to generate an ordered sequence located in the tail part of the heap. The difference between this operation and the classic heapsort operation is that the entries in the resulting sequence are duplicate-free while the initial heap includes duplicate entries. In each *delete-max* step in our variant heapsort, the root node and the first entry of the resulting sequence are fused if they share the same index; otherwise the root node is inserted to the head part of the sequence. The method is also distinguished from a heap-based sparse accumulator given by Gilbert et al. [78] by the mechanism of eliminating duplicate entries. Figure 7.1 gives two steps of an example of the heap method. Finally, the

sorted sequence without duplicate indices is generated in the scratchpad memory and saved to the output vector in the global memory. In addition, the numbers of nonzero entries of the output is updated to the sizes of the corresponding resulting sequences. To achieve higher throughput, the *ad*-heap proposed in Section 4.6 can be directly used here.

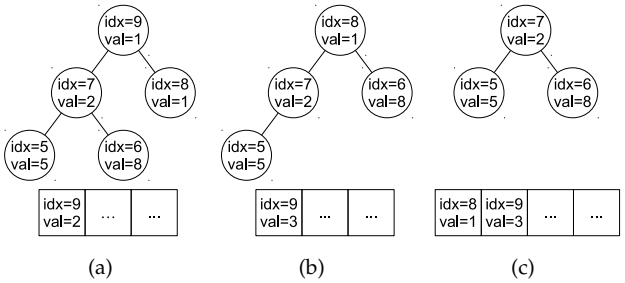


Figure 7.1: Two steps of an example of the heap method. From (a) to (b), the root entry is fused to the first entry in resulting sequence since they share the same index. From (b) to (c), the root entry is inserted to the sequence since they have different indices. After each step, the heap property is reconstructed.

7.4.2 Bitonic ESC Method

The method first collects all candidate nonzero entries to an array in the scratchpad memory, then sorts the array by using basic bitonic sort (see Section 4.5.2) and compresses duplicate indices in the sequence by using a combination of all-scan (see Section 4.4.2) and segmented sum (see Section 4.4.4). Figure 7.2 shows an example of adding two sparse vectors with 4 and 2 nonzero entries, respectively. Finally, a sorted sequence without duplicate indices is generated in the scratchpad memory and saved to the global memory, and the numbers of nonzero entries of the output sparse vector is recorded.

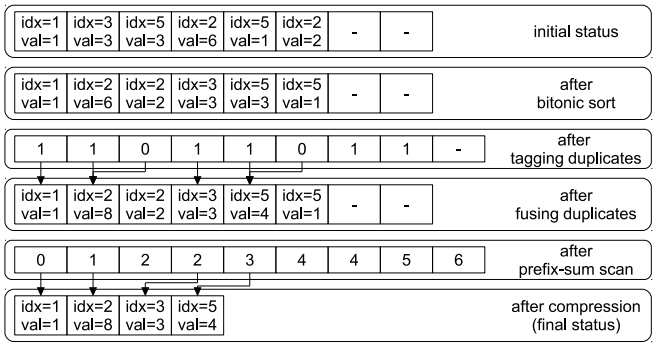


Figure 7.2: An example of the bitonic ESC method.

7.4.3 Merge Method

We can see that nonzero entries of the input vectors and the resulting vector can always be kept ordered and duplicate-free because of the CSR format[1]. Therefore, we can convert the addition operation to a parallel merge operation that merges ordered sequences and the final resulting sequence is ordered and duplicate-free.

Each parallel merge operation can be split into multiple sub-steps: (1) a binary search operation on the resulting sequence for fusing entries with the same indices and tagging them, (2) an all-scan operation on the input sequence for getting continuous positions in the incremental part of the resulting sequence, (3) copying non-duplicate entries from the input sequence to the resulting sequence, and (4) merging the two sequences in one continuous memory space. Figure 7.3 shows an example of this procedure. Finally the resulting sequence is the output sparse vector, and its length is the numbers of nonzero entries in the output.

Because both the binary search and the all-scan take fast logarithmic time for each entry in the input sequence, these operations have relatively good efficiency and performance stability on modern parallel processors. Therefore, a fast merge algorithm is very crucial for the performance of the merge method in our SpGEMM framework. Based on the experimental evaluation described in Section 4.5.6, GPU merge path algorithm is selected from five candidate GPU merge approaches.

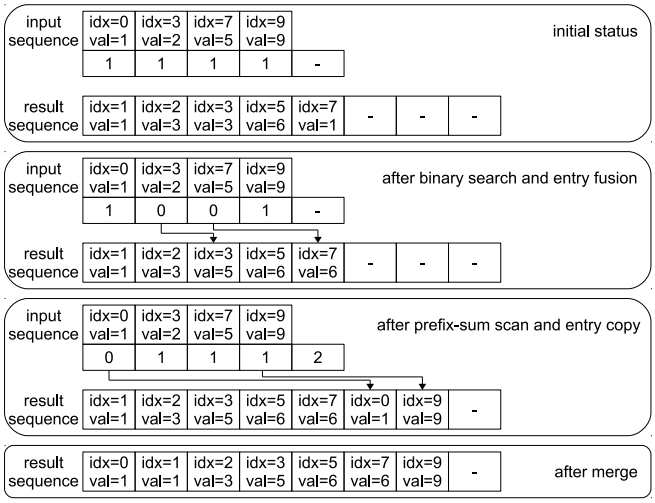


Figure 7.3: An example of the merge method. The original status of the resulting sequence contains the 1st input vector. The input sequence as the 2nd input vector can be stored in the register file. Its mask sequence and the resulting sequence are in the scratchpad memory.

[1]Actually according to the CSR format standard, the column indices in each row do not necessarily have to be sorted. But most implementations choose to do so, thus our method reasonably makes this assumption.

8. Level 2: Sparse Matrix-Vector Operations

8.1 Overview

In the Level 2 Sparse BLAS, multiplication of a sparse matrix and a dense or sparse vector is very useful and challenging. In this chapter, we mainly consider fast algorithms for sparse matrix-vector multiplication (SpMV for short), which is perhaps the most widely-used non-trivial BLAS in computational science and modeling. In the next chapter, a Level 3 BLAS routine SpGEMM will be introduced. This operation can be seen as a more generic case of multiplication of sparse matrix and sparse vector.

The SpMV operation multiplies a sparse matrix A of size $m \times n$ by a dense vector x of size n and gives a dense vector y of size m. The naïve sequential implementation of SpMV can be very simple, and can be easily parallelized by adding a few pragma directives for the compilers [112]. But to accelerate large-scale computation, parallel SpMV is still required to be hand-optimized with specific data storage formats and algorithms [7, 8, 16, 21, 33, 35, 38, 45, 58, 76, 80, 102, 115, 116, 122, 170, 173, 166, 177, 184, 188, 191, 193, 194, 192] (see Chapter 5 for some recently proposed formats for specific hardware architectures such as GPUs and Xeon Phi). The experimental results showed that these formats can provide performance improvement for various SpMV benchmarks.

However, the completely new formats bring several new problems. The first one is backward-compatibility. When the input data are stored in basic formats (e.g., CSR), a format conversion is required for using the new format based SpMV. In practice, fusing a completely new format into well-established toolkits (e.g., PETSc [12]) for scientific software is not a trivial task [132] because of the format conversion. Moreover, Kumbhar [107] pointed out that once an application (in particular a non-linear solver) needs repeated format conversion after a fixed small number of iterations, the new formats may degrade overall performance. Furthermore, Langr and Tvrdík [110] demonstrated that isolated SpMV performance is insufficient to evaluate a new format. Thus more evaluation criteria, such as format conversion cost and memory footprint, must be taken into consideration. Secondly, when the SpMV operation is used with other sparse building blocks (such as preconditioning operations [116] and SpGEMM [118]) that require basic storage formats, using the all-new formats is less feasible.

Therefore, if we can reduce the cost of format conversion and extra memory footprint to a certain level, a new format can offer faster SpMV, compared to the CSR format, in an iterative scenario. Otherwise, accelerating CSR-based SpMV can give the best performance, in particular when the number of iterations is low. In this chapter, we introduce our SpMV algorithms both for the CSR format and for the CSR5 format described in Chapter 6.

8.2 Contributions

Our work described in this chapter particularly focuses on CSR-based SpMV on CPU-GPU heterogeneous processors and CSR5-based SpMV on CPUs, GPUs and Xeon Phi.

The main idea of our CSR-based SpMV algorithm is first speculatively executing SpMV on a heterogeneous processor's GPU cores targeting high throughput computation, and then locally re-arranging resulting vectors by the CPU cores of the same chip for low-latency memory access. To achieve load balanced first step computation and to utilize both CPU and GPU cores, we improved the conventional segmented sum method by generating auxiliary information (e.g., segment descriptor) at runtime and recognizing empty rows on-the-fly. Compared to the row block methods for the CSR-based SpMV, our method delivers load balanced computation to achieve higher throughput. Compared with the classic segmented sum method for the CSR-based SpMV, our approach decreases the overhead of global synchronization and removes pre- and post-processing regarding empty rows.

The CSR-based SpMV work makes the following contributions:

- A fast CSR-based SpMV algorithm that efficiently utilizes different types of cores in emerging CPU-GPU heterogeneous processors.
- An speculative segmented sum algorithm that generates auxiliary information on-the-fly and eliminating costly pre- and post-processing on empty rows.
- On a widely-adopted benchmark suite, the proposed CSR-based SpMV algorithm achieves stable SpMV performance independent of the sparsity structure of input matrix.

On a benchmark suite composed of 20 matrices with diverse sparsity structures, the CSR-based SpMV approach greatly outperforms the row block methods for the CSR-based SpMV running on GPU cores of heterogeneous processors. On an Intel heterogeneous processor, the experimental results show that our method obtains up to 6.90x and on average 2.57x speedup over an OpenCL implementation of the CSR-vector algorithm in CUSP running on its GPU cores. On an AMD heterogeneous processor, our approach delivers up to 16.07x (14.43x) and on average 5.61x (4.47x) speedup over the fastest single (double) precision CSR-based SpMV algorithm from PARALUTION and an OpenCL version of CUSP running on its GPU cores. On an nVidia heterogeneous processor, our approach delivers up to 5.91x (6.20x) and on average 2.69x (2.53x) speedup over the fastest single (double) precision CSR-based SpMV algorithm from cuSPARSE and CUSP running on its GPU cores.

The CSR5 format described in Chapter 6 extends the basic CSR format for using SIMD units efficiently, thus can be an alternative of the CSR format. The CSR5 format merely needs very short format conversion time (a few SpMV operations) and very small extra memory footprint (around 2% of the CSR data). Because the CSR5 format shares data with the CSR format, the CSR-based Sparse BLAS routines can efficiently work with the CSR5 format.

The CSR5-based SpMV work makes the following contributions:

- A CSR5-based SpMV algorithm based on a redesigned low-overhead segmented sum algorithm.
- The work is implemented on four mainstream devices: CPU, nVidia GPU, AMD GPU and Intel Xeon Phi.
- The CSR5-based SpMV is evaluated in both isolated SpMV tests and iteration-based scenarios.

We compare the CSR5 with 11 state-of-the-art formats and algorithms on dual-socket Intel CPUs, an nVidia GPU, an AMD GPU and an Intel Xeon Phi. By using 14 regular and 10 irregular matrices as a benchmark suite, we show that the CSR5 obtains comparable or better performance over the previous work for the regular matrices, and can greatly outperform the prior work for the irregular matrices. As for the 10 irregular matrices, the CSR5 obtains on average speedups 1.18x, 1.29x, 2.73x and 3.93x (up to 3.13x, 2.54x, 5.05x and 10.43x) over the second best work on the four platforms, respectively. Moreover, for iteration-based real-world scenarios, the CSR5 format achieves higher speedups because of the fast format conversion. To the best of our knowledge, this is the first time that a single storage format can outperform state-of-the-art work on all four modern multicore and manycore processors.

8.3 Basic Methods

8.3.1 Row Block Algorithm

In a given sparse matrix, rows are independent from each other. Therefore an SpMV operation can be parallelized on decomposed row blocks. A logical processing unit is responsible for a row block and stores dot product results of the matrix rows with the vector x to corresponding locations in the result y. When the SIMD units of a physical processing unit are available, the SIMD reduction sum operation can be utilized for higher efficiency. These two methods are respectively known as the CSR-scalar and the CSR-vector algorithms, and have been implemented on CPUs [184] and GPUs [21, 170]. Algorithm 28 and Algorithm 29 show parallel CSR-scalar and CSR-vector methods, respectively.

Despite the good parallelism, exploiting the scalability in modern multi-processors is not trivial for the row block methods. The performance problems mainly come from load imbalance for matrices which consist of rows with uneven lengths. Specifically, if one single row of a matrix is significantly longer than the other rows, only a single core can be fully used while the other cores in the same chip may be completely idle.

Algorithm 28 Parallel SpMV using the CSR-scalar method.

```
1: for i = 0 to m − 1 in parallel do
2:     y[i] ← 0
3:     for j = row_ptr[i] to row_ptr[i + 1] − 1 do
4:         y[i] ← y[i] + val[j] × x[col_idx[j]]
5:     end for
6: end for
```

Algorithm 29 Parallel SpMV using the CSR-vector method.

```
1: for i = 0 to m − 1 in parallel do
2:     for j = row_ptr[i] to row_ptr[i + 1] − 1 in parallel do
3:         tmp<vector> ← val[j] × x[col_idx[j]]
4:     end for
5:     len ← row_ptr[i + 1] − row_ptr[i]
6:     y[i] ← REDUCTION_SUM_PARALLEL(tmp<vector>, len)        ▷ Algorithm 4
7: end for
```

Although various strategies, such as data streaming [51, 80], memory coalescing [58], data reordering or reconstruction [16, 84, 149], static or dynamic binning [7, 80], Dynamic Parallelism [7] and and dynamic row distribution [123], have been developed, none of those can fundamentally solve the problem of load imbalance, and thus provide relatively low SpMV performance for the CSR format.

8.3.2 Segmented Sum Algorithm

Blelloch et al. [28] pointed out that the segmented sum may be more attractive for the CSR-based SpMV, since it is SIMD friendly and insensitive to the sparsity structure of the input matrix, thus overcoming the shortcomings of the row block methods. A serial version of segmented sum is shown in Algorithm 12.

In the SpMV operation, the segmented sum treats each matrix row as a segment and calculates a partial sum for the entry-wise products generated in each row. The SpMV operation using the segmented sum methods consists of four steps: (1) generating an auxiliary `bit_flag` array of size nnz from the `row_ptr` array. An entry in `bit_flag` is flagged as `TRUE` if its location matches the first nonzero entry of a row, otherwise it is flagged as `FALSE`, (2) calculating all intermediate entries (i.e., entry-wise products) to an array of size nnz, (3) executing the parallel segmented sum for the array, and (4) collecting all partial sums to the result vector y if a row is not empty. Algorithm 30 lists the pseudocode. Figure 8.2 illustrates an example using the matrix A plotted in Figure 8.1. We can see that once the heaviest workload, i.e., step 3, is parallelized through a fast segmented sum method described in [40, 63, 161], nearly perfect load balance can be expected in all steps of Algorithm 30.

Algorithm 30 Segmented sum method CSR-based SpMV.

```
1: MALLOC(*bit_flag, nnz)
2: MEMSET(*bit_flag, FALSE)
3: for i = 0 to m − 1 in parallel do                          ▷ Step 1
4:     bit_flag[row_ptr[i]] ← TRUE
5: end for
6: MALLOC(*product, nnz)
7: for j = 0 to nnz − 1 in parallel do                        ▷ Step 2
8:     product[j] ← val[j] × x[col_idx[j]]
9: end for
10: SEGMENTED_SUM(*product, nnz, *bit_flag)                   ▷ Step 3, Algorithm 12
11: for k = 0 to m − 1 in parallel do                         ▷ Step 4
12:     if row_ptr[k] = row_ptr[k + 1] then
13:         y[k] ← 0
14:     else
15:         y[k] ← product[row_ptr[k]]
16:     end if
17: end for
18: FREE(*bit_flag)
19: FREE(*product)
```

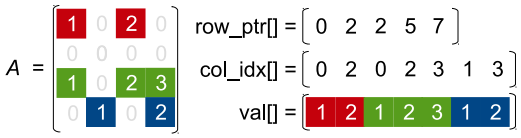


Figure 8.1: A CSR matrix of size 4×4.

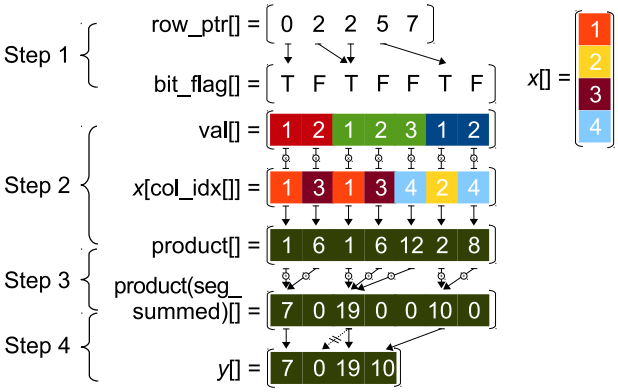


Figure 8.2: CSR-based SpMV using segmented sum.

8.3.3 Performance Considerations

Figure 8.3 gives a performance comparison of the row block method from the cuSPARSE v6.5 library and the segmented sum method from the cuDPP v2.2 [76, 161] library. It shows that the row block method can significantly outperform the segmented sum method, while doing SpMV on relatively regular matrices (see Table A.1 for details of the used benchmark suite). On the other hand, row block method only gives very low performance for irregular matrices.

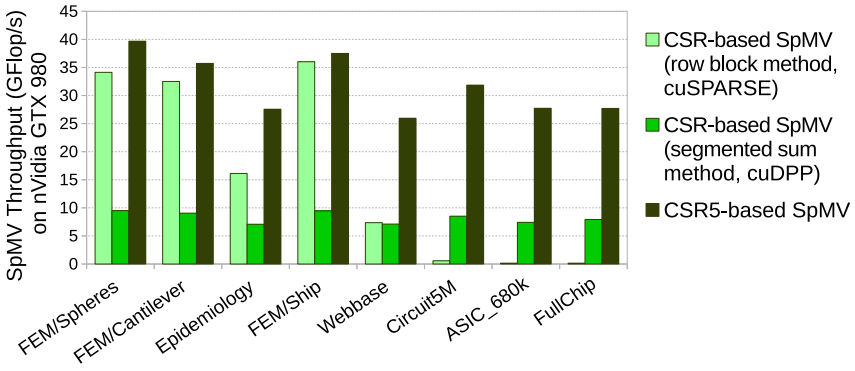


Figure 8.3: Single precision SpMV performance.

Why is this the case? We can see that the step 1 of the segmented sum method is a scatter operation and the step 4 is a gather operation, both from the row space of size m. This prevents the two steps from fusing with the steps 2 and 3 in the nonzero entry space of size nnz. In this case, more global synchronizations and global memory accesses may degrade the overall performance. Previous research [21, 170] has found that the segmented sum may be more suitable for the COO-based SpMV, since the fully stored row index data can convert the steps 1 and 4 to the nonzero entry space: the `bit_flag` array can be generated by comparison of neighbor row indices, and the partial sums in the `product` array can be directly saved to y since their final locations are easily known from the row index array. Further, Yan et al. [188] and Tang et al. [173] reported that some variants of the COO format can also benefit from the segmented sum. However, it is well known that accessing row indices in the COO pattern brings higher off-chip memory pressure, which is just what the CSR format tries to avoid.

In Figure 8.3, we can also see that the CSR5-based SpMV can obtain up to 4x speedup over the CSR-based SpMV using the segmented sum primitive [161]. Its main reason is that the CSR5-based SpMV can utilize both the segmented sum for load balance and the compressed row data for better load/store efficiency. The details will be shown below.

8.4 CSR-Based SpMV

8.4.1 Algorithm Description

Data Decomposition

In the proposed CSR-based SpMV, we first evenly decompose nonzero entries of the input matrix to multiple small tiles for load balanced data parallelism. Here we define a tile as a 2D array of size $W \times T$. The width T is the size of a thread-bunch, which is the minimum SIMD execution unit in a given vector processor. It is also known as wavefront in AMD GPUs or warp in nVidia GPUs. The height W is the workload (i.e., the number of nonzero entries to be processed) of a thread. A tile is a basic work unit in matrix-based segmented sum method [161, 63], which is used as a building block in our SpMV algorithm. Actually, the term "tile" is equivalent to the term "matrix" used in original description of the segmented scan algorithms [161, 63]. Here we use "tile" to avoid confusion between a work unit of matrix shape and a sparse matrix in SpMV.

Since a thread-bunch can be relatively too small (e.g., as low as 8 in current Intel GPUs) to amortize scheduling cost, we combine multiple thread-bunches into one thread-group for possibly higher throughput. We define B to denote the number of thread-bunches in one thread-group. Additionally, we let each thread-bunch compute S contiguous tiles. Thus higher on-chip resource reuse and faster global synchronization are expected.

Therefore, we can calculate that each thread-group deals with $BSWT$ nonzero entries. Thus the whole nonzero entry space of size nnz can be evenly assigned to $\lceil nnz/(BSWT) \rceil$ thread-groups. Figure 8.4 shows an example of the data decomposition. In this example, we set $B = 2$, $S = 2$, $W = 4$, and $T = 2$. Thus each thread-group is responsible for 32 nonzero entries. Then $\lceil nnz/32 \rceil$ thread-groups are dispatched.

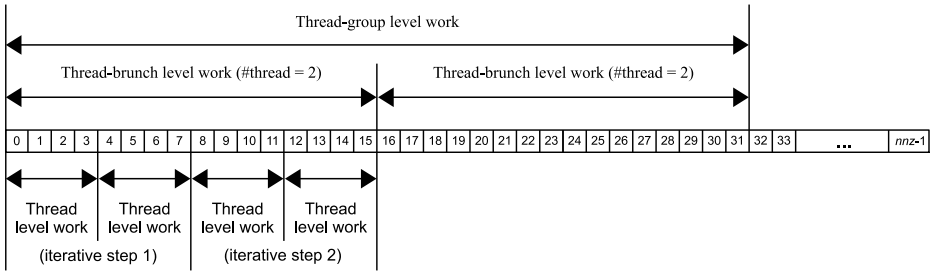


Figure 8.4: Data decomposition on the nonzero entry space. nnz nonzero entries are assigned to multiple thread-groups. In this case, each thread-group consists of 2 thread-bunches (i.e., $B = 2$). The number of threads in each thread-bunch is equal to 2 (i.e., $T = 2$). The workload per thread is 4 (i.e., $W = 4$). The number of iterative steps in each thread-bunch is 2 (i.e., $S = 2$).

Speculative Segmented Sum

Our CSR-based SpMV is based on fundamental segmented sum algorithm, which guarantees load balanced computation in the nonzero entry space. While utilizing segmented sum as a building block in our SpMV algorithm, we have three main performance considerations: (1) the segment descriptor needs to be generated in on-chip memory at runtime to reduce overhead of global memory access, (2) empty rows must be recognized and processed without calling specific pre- and post-processing functions, and (3) taking advantages of both types of cores in a heterogeneous processor. Hence we improve the basic segmented sum method to meet the above performance requirements.

The algorithm framework includes two main stages: (1) speculative execution stage, and (2) checking prediction stage. The first stage speculatively executes SpMV operation and generates a possibly incorrect resulting vector y. Here the term "incorrect" means that the layout of entries in y can be incorrect, but the entries are guaranteed to be numerically identified. Then in the second stage we check whether or not the speculative execution is successful. If the prediction is wrong, a data re-arrangement will be launched for getting a completely correct y.

We first give an example of our algorithm and use it in the following algorithm description. Figure 8.5 plots this example. The input sparse matrix includes 12 rows (2 of them are empty) and 48 nonzero entries. We set B to 1, S to 2, T to 4 and W to 6. This setting means that one thread-group is composed of one thread-bunch of size 4; each thread-bunch runs 2 iteration steps. Before GPU kernel launch, three containers, *synchronizer*, *dirty_counter* and *speculator*, are pre-allocated in DRAM for global synchronization and speculative execution. Algorithm 31 lists pseudocode of the first stage.

The **speculative execution stage** includes the following steps: (1) positioning a range of row indices for nonzero entries in a given tile, (2) calculating segment descriptor based on the range, (3) conducting segmented sum on the tile, (4) saving partial sums to the computed index range in vector y. This stage also has some input-triggered operations such as labeling a tile with empty rows.

Algorithm 31 The SPMD pseudocode of a thread-bunch in speculative execution stage of the CSR-based SpMV

```
1:  function SPECULATIVE_EXECUTION_GPU()
2:      tb ← GET-THREAD-GLOBALID( )/T;
3:      //positioning row indices of tiles in the thread-bunch
4:      for i = 0 to S do
5:          boundary[i] ← tb × S × W × T + i × W × T
6:          tile_offset[i]← BINARY_SEARCH(*row_pointer, boundary[i])
7:      end for
8:      //iterative steps in a thread-bunch
9:      for i = 0 to S − 1 do
10:         start ← tile_offset[i]
11:         stop ← tile_offset[i + 1]
```

```
12:         MEMSET(*seg_descriptor, FALSE)
13:         dirty ← FALSE
14:         //calculating segment descriptor
15:         for j = start to stop − 1 do
16:             if row_pointer[j] ≠ row_pointer[j + 1] then
17:                 descriptor[row_pointer[j] - boundary[i]] ← TRUE
18:             else
19:                 dirty ← TRUE
20:             end if
21:         end for
22:         //collecting element-wise products
23:         for j = 0 to W × T − 1 do
24:             x_value ← x[column_index[boundary[i]+j]]
25:             product[j] ← x_value × value[boundary[i]+j]
26:         end for
27:         //transmitting a value from the previous tile
28:         if descriptor[0] = FALSE then
29:             product[0] ← product[0] + transmitter
30:             descriptor[0] ← TRUE
31:         end if
32:         //segmented sum
33:         SEGMENTED_REDUCTION(*product, *descriptor, *ts, *tc)
34:         //calculating index offset in y
35:         *y_index ← EXCLUSIVE_SCAN(*tc)
36:         //saving partial sums to y
37:         for j = 0 to T − 1 do
38:             for k = 0 to tc[j] − 1 do
39:                 index ← start + y_index[j] + k
40:                 //first segment of the thread-bunch
41:                 if index = tile_offset[0] then
42:                     synchronizer[tb].idx ← index
43:                     synchronizer[tb].val ← product[j × W+ts[j]+k]
44:                 else
45:                     //storing to y directly
46:                     y[index] ← product[j × W+ts[j]+k]
47:                 end if
48:                 if index = stop then
49:                     transmitter ← product[j × W+ts[j]+k]
50:                 end if
51:             end for
52:         end for
53:         //labeling dirty tile
54:         if dirty = TRUE then
55:             pos ← ATOMIC_INCREMENT(*dirty_counter)
56:             speculator[pos] ← ⟨start, stop⟩
57:             transmitter ← 0
58:         end if
59:     end for
60: end function
61:
```

```
62: function SYNCHRONIZATION_CPU()
63:     for i = 0 to ⌈nnz/(S × W × T)⌉ − 1 do
64:         index ← synchronizer[i].idx
65:         value ← synchronizer[i].val
66:         y[index] ← y[index] + value
67:     end for
68: end function
```

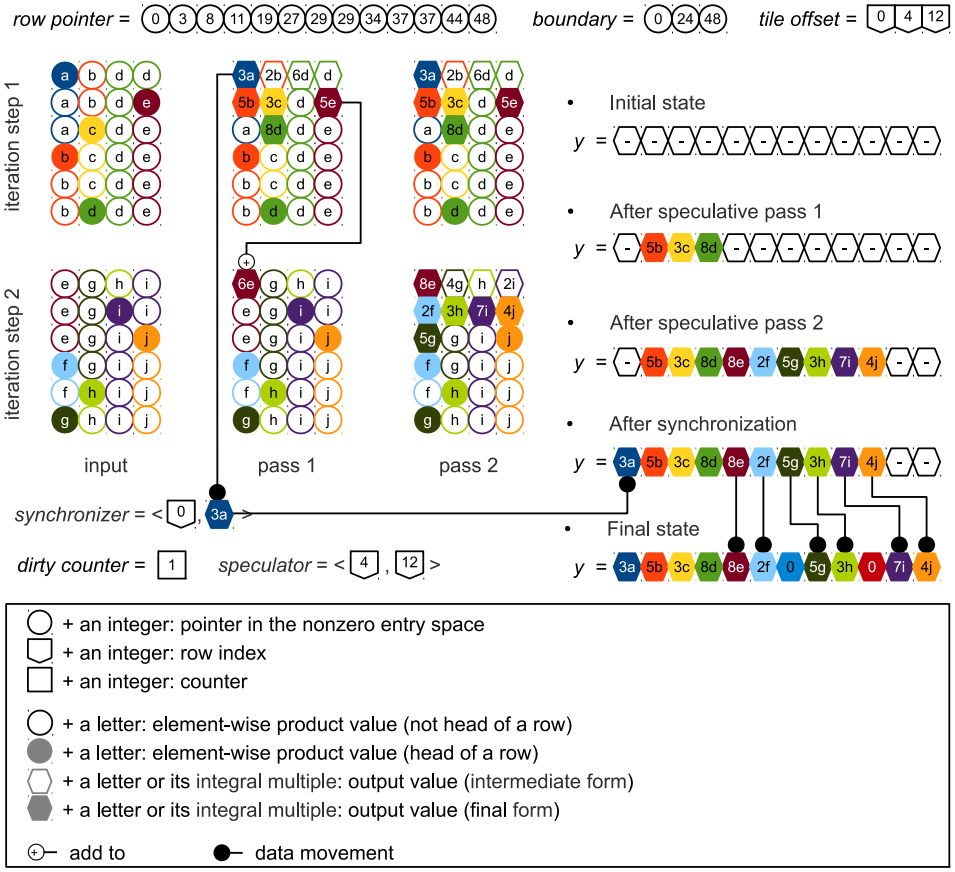


Figure 8.5: An example of our CSR-based SpMV algorithm. The input sparse matrix contains 48 nonzero entries in 12 rows (10 non-empty rows and 2 empty rows). One thread-bunch composed of 4 threads is launched in this 2-iteration process. The arrays *synchronizer* and *speculator* store tuples (shown with angular brackets).

First, each thread-bunch executes binary search of $S + 1$ tile boundaries on the CSR row pointer array. Then we obtain corresponding row indices and store them in a scratchpad array *tile offset* of size $S + 1$. The results of the binary search are starting and ending row indices of the nonzero entries in each tile. Thus each tile knows

the locations to store generated partial sums. Lines 3–7 of Algorithm 31 give a code expression of this step. In our example shown in Figure 8.5, the 2 tiles of size 24 have 3 boundaries {0, 24, 48}. The results of binary search of {0, 24, 48} on the CSR row pointer array are {0, 4, 12}. Note that the binary search needs to return the rightmost match, if multiple slots have the same value.

Then each thread-bunch executes an iteration of S steps. Lines 8–59 of Algorithm 31 give code expression of this step. Each iteration deals with one tile. By calculating offset between the left boundary of a tile and the covered row indices, a local segment descriptor is generated (lines 14–21 in Algorithm 31). For example, the left boundary of the second tile is 24 and its row index range is 4–12. We need to compute offset between 24 and the row pointer {19, 27, 29, 29, 34, 37, 37, 44, 48}. Then we obtain a group of offsets {-5, 3, 5, 5, 10, 13, 13, 20, 24}. After removing duplicate values and overflowed values on the left and the right sides, the effective part {3, 5, 10, 13, 20} in fact implies local segment descriptor for the current tile. We can easily convert it to a binary expression {0, 0, 0, 1, 0, 1, 0, ... , 0, 0, 1, 0, 0, 0} through a scatter operation in on-chip scratchpad memory. Moreover, since each tile is an independent work unit, the first bit of its segment descriptor should be `TRUE`. Thus the final expression becomes {1, 0, 0, 1, 0, 1, 0, ... , 0, 0, 1, 0, 0, 0}. In Figure 8.5, the filled and empty circles are heads (i.e., 1s or `TRUE`s) and body (i.e., 0s or `FALSE`s) of segments, respectively.

While generating the segment descriptor, each thread detects whether or not its right neighbor wants to write to the same slot. If yes (like the duplicate offset information {..., 5, 5, ...} and {..., 13, 13, ...} in the above example), we can make sure that this tile contains at least one empty row, since an empty row is expressed as two contiguous indices of the same value in the CSR row pointer array. Then we mark this tile as "dirty" (line 19 in Algorithm 31). Further, the *dirty counter* array stored in DRAM is incremented by atomic operation, and this tile's offset is recorded in the *speculator* array (lines 53–58 in Algorithm 31). In our example, *dirty counter* is 1 and *speculator* array has a pair of offsets {⟨4, 12⟩} ((shown with angular brackets in Figure 8.5).

Then we calculate and save element-wise products in scratchpad memory, based on its nonzero entries' column indices, values and corresponding values in the vector x. Lines 22–26 of Algorithm 31 show code expression of this step. When finished, we transmit the sum of the last segment to an intermediate space for the next iteration (lines 27–31 in Algorithm 31). In our example, the first tile's last value $5e$ is transmitted to the next tile. Then we execute the matrix-based segmented sum (lines 32–33) on the tile. Because the segmented sum algorithm used here is very similar to the method described in [28], we refer the reader to [28] and several pervious GPU segmented sum algorithms [161, 63] for details. But note that compared to [28], our method makes one difference: we store partial sums in a compact pattern (i.e., values are arranged in order from the first location in the thread work space), but not save them to locations of corresponding segment heads. For this reason, we need to record the starting position and the number of partial sums. Then we can use an ordinary exclusive scan operation (lines 34–35) for obtaining contiguous indices of the partials sums in y. In Figure 8.5, we can see that the partial sums (expressed as filled hexagons) are aggregated in the compact fashion. Note that empty hexagons are intermediate

partial sums, which are already added to the correct position of segment heads.

Finally, we store the partial sums to known locations in the resulting vector. Lines 36–52 of Algorithm 31 show code expression. As an exception, the sum result of the first segment in a thread-bunch is stored to the *synchronizer* array (lines 40–43), since the first row of each thread-bunch may cross multiple thread-bunch. This is a well known issue while conducting basic primitives, such as reduction and prefix-sum scan, using more than one thread-group that cannot communicate with each other. In fact, atomic add operation can be utilized to avoid the global synchronization. But we choose not to use relatively slow global atomic operations and let a CPU core to later on finish the global synchronization. Lines 62–68 of Algorithm 31 show the corresponding code expression. Since the problem size (i.e., $\lceil nnz/(SWT) \rceil$) can be too small to saturate a GPU core, a CPU core is in fact faster for accessing short arrays linearly stored in DRAM. Taking the first tile in Figure 8.5 as an example, its first partial sum is $3a$, which is stored with its global index 0 to the *synchronizer*. After that, the value $3a$ is added to position 0 of y.

When the above steps are complete, the resulting vector is numerically identified, except that some values generated by dirty tiles are not in their correct locations. In Figure 8.5, we can see that after synchronization, vector y is already numerically identified to its final form, but entries $5g$, $3h$, $7i$ and $4j$ generated by the second tile are located in wrong slots.

The **checking prediction stage** first checks value of the *dirty counter* array. If it is zero, the previous prediction is correct and the result of the first stage is the final result; if it is not zero, the predicted entries generated by dirty tiles are scattered to their correct positions in the resulting vector. In this procedure, the CSR row pointer array is required to be read for getting correct row distribution information. Again, we use a CPU core for the irregular linear memory access, which is more suitable for cache sub-systems in CPUs. In our example, entries $5g$, $3h$, $7i$ and $4j$ are moved to their correct positions. Then the SpMV operation is done.

Complexity Analysis

Our CSR-based SpMV algorithm pre-allocates three auxiliary arrays, *synchronizer*, *dirty counter* and *speculator*, in off-chip DRAM. The space complexity of *synchronizer* is $\lceil nnz/(SWT) \rceil$, equivalent to the number of thread-bunches. The size of *dirty counter* is constant 1. The *speculator* array needs a size of $\lceil nnz/(WT) \rceil$, equivalent to the number of tiles. Since W and T are typically set to relatively large values, the auxiliary arrays merely slightly increase overall space requirement.

For each thread-bunch, we executes $S+1$ binary searches in the row pointer array of size $m+1$. Thus $O(\lceil nnz/(SWT) \rceil \times (S+1) \times \log_2(m+1)) = O(nnz \log_2(m)/WT)$ is work complexity of this part. On the whole, generating segment descriptor needs $O(m)$ time. Collecting element-wise products needs $O(nnz)$ time. For each tile, segmented sum needs $O(WT + \log_2(T))$ time. Thus all segmented sum operations need $O(\lceil nnz/(WT) \rceil (WT + \log_2(T))) = O(nnz + nnz \log_2(T)/WT)$ time. Saving entries to y needs $O(m)$ time. Synchronization takes $O(\lceil nnz/(SWT) \rceil) = O(nnz/SWT)$ time. Possible re-arrangement needs $O(m)$ time in the worst case. Thus overall work complexity of our CSR-based SpMV algorithm is $O(m + nnz + nnz(\log_2(m) + \log_2(T))/WT)$.

Implementation Details

Based on the above analysis, we can see that when the input matrix is fixed, the cost of our SpMV algorithm only depends on two parameters: T and W. In our algorithm implementation, T is set to SIMD length of the used processor. Choosing W needs to consider the capacity of on-chip scratchpad memory. The other two parameters B and S are empirically chosen. Table 8.1 shows the selected parameters. Note that double precision is not currently supported in Intel OpenCL implementation for its GPUs.

Table 8.1: The selected parameters

Processor	Intel	AMD		nVidia	
Precision	32-bit single	32-bit single	64-bit double	32-bit single	64-bit double
T	8	64	64	32	32
W	16	16	8	8	4
B	4	2	2	5	5
S	6	2	5	7	7

We implement the first stage of our algorithm in OpenCL for the Intel and AMD platforms (and CUDA for the nVidia platform) for GPU execution and the second stage in standard C language running on the CPU part. Since our algorithm needs CPU and GPU share some arrays, we allocate all arrays in Shared Virtual Memory supported by OpenCL for the best performance. On the nVidia platform, we use Unified Memory in CUDA SDK.

8.4.2 Experimental Results

Experimental Setup

We use three heterogeneous processors, Intel Core i3-5010U, AMD A10-7850K APU and nVidia Tegra K1, for evaluating the CSR-based SpMV algorithms. Tables B.4 and B.5 shows specifications of the three processors. All of them are composed of multiple CPU cores and GPU cores. The platforms from AMD and nVidia are based on the design of Figure 3.5(a); the Intel platform uses the design of Figure 3.5(b).The two types of cores in the Intel heterogeneous processor share a 3 MB last level cache. In contrast, GPU cores in the AMD heterogeneous processor can snoop the L2 cache of size 4 MB on the CPU side. Unlike those, the cache systems of the CPU part and the GPU part in the nVidia Tegra processor are completely separate. Note that currently the Intel GPU can run OpenCL program only on Microsoft Windows operating system. Also note that we use kB, MB and GB to denote 2^{10}, 2^{20} and 2^{30} bytes, respectively; and use GFlop to denote 10^9 flops.

To analyze efficiency of the proposed SpMV algorithm, we also benchmark parallel CSR-based SpMV using some other libraries or methods on CPUs and GPUs.

On CPUs, we execute three CSR-based SpMV approaches: (1) OpenMP-accelerated basic row block method, (2) pOSKI library [36] using OSKI [177] as a building block, and (3) Intel MKL v11.2 Update 2 in Intel Parallel Studio XE 2015 Update 2. The three

approaches are running on all CPU cores of the used heterogeneous processors. For the Intel CPU, we report results from MKL, since it always delivers the best performance and the pOSKI is not supported by the used Microsoft Windows operating system. For the AMD CPU, we report the best results of the three libraries, since none of the three libraries outperforms all the others. For the ARM CPU included in the nVidia Tegra K1 platform, we only report results from OpenMP, since the current pOSKI and Intel MKL implementations do not support the ARM architecture. Moreover, single-threaded naïve implementation on CPU is included in our benchmark as well.

On GPUs, we benchmark variants of the CSR-scalar and the CSR-vector algorithms proposed in [21]. The OpenCL version of the CSR-scalar method is extracted from PARALLUTION v1.0.0 [126] and evaluated on the AMD platform. The OpenCL implementation of the CSR-vector method is extracted from semantically equivalent CUDA code in the CUSP library v0.4.0 and executed on both the Intel and the AMD platforms. On the nVidia platform, we run the CSR-based SpMV from vendor-supplied cuSPARSE v6.0 and CUSP v0.4.0 libraries.

For all tests, we run SpMV 200 times and record averages. The implicit data transfer (i.e., matrices and vectors data copy from their sources to OpenCL Shared Virtual Memory or CUDA Unified Memory) is not included in our evaluation, since SpMV operation is normally one building block of more complex applications. All participating methods conduct general SpMV, meaning that symmetry is not considered although some input matrices are symmetric. The throughput (flops per second) is calculated by

$$\frac{2 \times nnz}{runtime}.$$

The bandwidth (bytes per second) is calculated by

$$\frac{(m + 1 + nnz) \times sizeof(idx\_type) + (nnz + nnz + m) * sizeof(val\_type)}{runtime}.$$

Benchmark Suite

To evaluate our method, we choose 20 unstructured matrices from the benchmark suite. Table 8.2 lists main information of the evaluated sparse matrices. The first 14 matrices of the benchmark suite have been widely used in previous work [21, 170, 188, 8, 120, 184]. The last 6 matrices are chosen as representatives of irregular matrices extracted from graph applications, such as circuit simulation and optimization problems. Appendix A has details of the selected matrices.

The first 10 matrices are relatively regular, due to short distance between the average value and the maximum value of nnz/row. The other matrices are relatively irregular. In this context, 'regular' is used for a sparse matrix including rows of roughly the same size. In contrast, an 'irregular matrix' can have some very long rows and many very short rows. For example, matrices generated from power-law graphs can have a few rows with $O(n)$ nonzero entries and many rows with $O(1)$ nonzero entries.

Id	Name	*Dimensions*	*nnz*	*nnz* per row (min, avg, max)
r1	Dense	2K×2K	4.0M	2K, 2K, 2K
r2	Protein	36K×36K	4.3M	18, 119, 204
r3	FEM/Spheres	83K×83K	6.0M	1, 72, 81
r4	FEM/Cantilever	62K×62K	4.0M	1, 64, 78
r5	Wind Tunnel	218K×218K	11.6M	2, 53, 180
r6	FEM/Harbor	47K×47K	2.4M	4, 50, 145
r7	QCD	49K×49K	1.9M	39, 39, 39
r8	FEM/Ship	141K×141K	7.8M	24, 55, 102
r9	Economics	207K×207K	1.3M	1, 6, 44
r10	Epidemiology	526K×526K	2.1M	2, 3, 4
i1	FEM/Accelerator	121K×121K	2.6M	0, 21, 81
i2	Circuit	171K×171K	959K	1, 5, 353
i3	Webbase	1M×1M	3.1M	1, 3, 4.7K
i4	LP	4K×1.1M	11.3M	1, 2.6K, 56.2K
i5	ASIC_680k	683K×683K	3.9M	1, 6, 395K
i6	boyd2	466K×466K	1.5M	2, 3, 93K
i7	dc2	117K×117K	766K	1, 6, 114K
i8	ins2	309K×309K	2.8M	5, 8, 309K
i9	rajat21	412K×412K	1.9M	1, 4, 119K
i10	transient	179K×179K	962K	1, 5, 60K

Table 8.2: The benchmark suite.

Performance Analysis

Figures 8.6 and 8.7 show throughput of single precision and double precision SpMV of the tested CSR-based approaches, respectively.

In Figure 8.6, we can see that on the Intel heterogeneous processor, our approach obtains up to 6.90x and on average 2.57x speedup over the CSR-vector method running on the used GPU. Although the speedup mainly comes from irregular matrices, our method generally does not obviously lose performance on regular matrices. Further, compared to CPU cores running MKL, both GPU SpMV algorithms are slower. For our algorithm, the main reason is that the integrated GPU implements scratchpad memory in its L3 cache, which has one order of magnitude higher latency compared to fast scratchpad in nVidia or AMD GPUs. Our algorithm in fact heavily uses scratchpad memory for storing and reusing segment descriptor, element-wise products and other shared data by threads. Thus even though the GPU part of the Intel heterogeneous processor has higher single precision theoretical peak performance than its CPU part, the delivered SpMV throughput is lower than expected. For the CSR-vector method, the low performance has another reason: small thread-bunch of size 8 dramatically increases loop overhead [15], which is one of the well known bottlenecks [74] of GPU programming.

In Figures 8.6 and 8.7, we can see that on the AMD heterogeneous processor, our

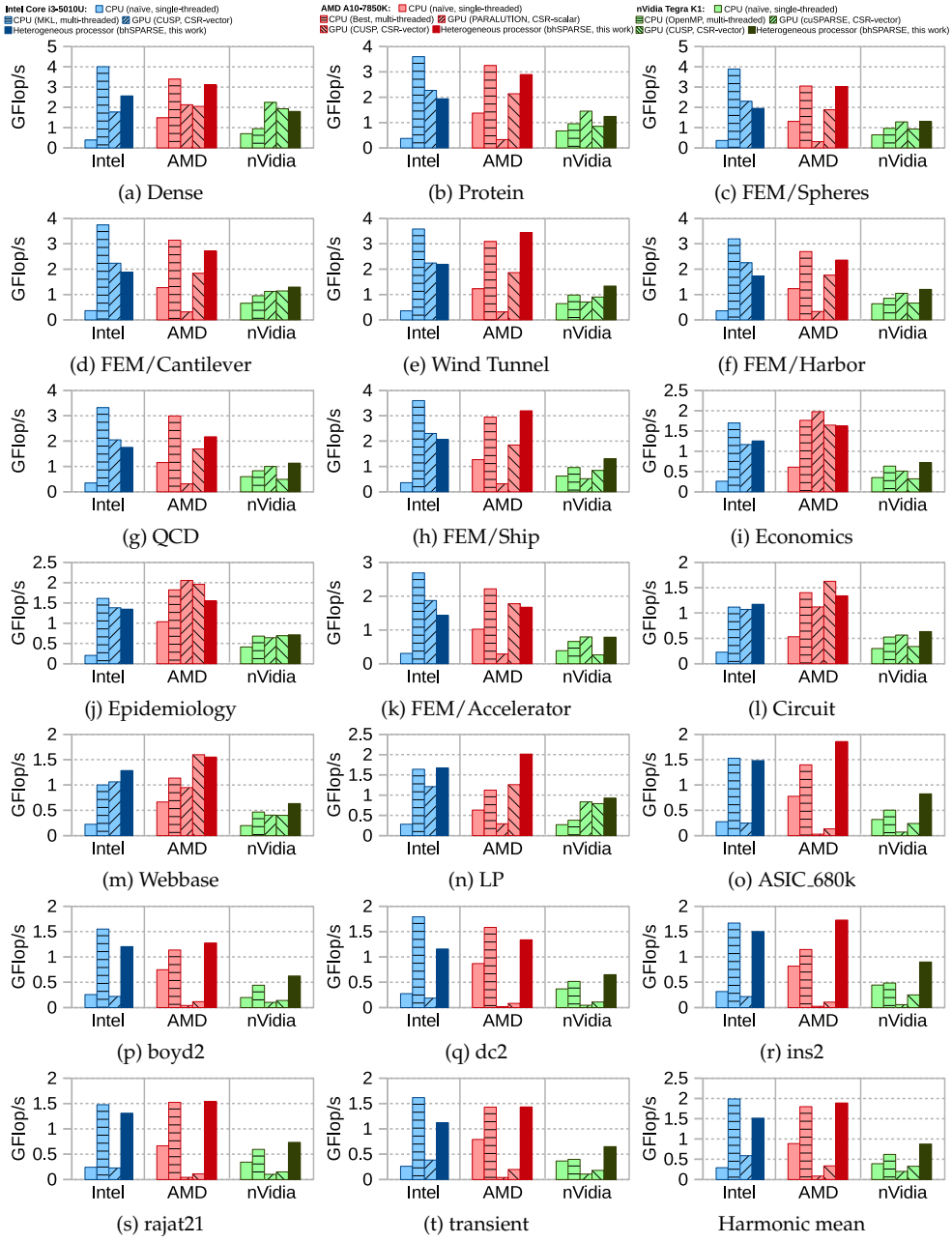


Figure 8.6: Throughput (GFlop/s) of the single precision CSR-based SpMV algorithms running on the three platforms.

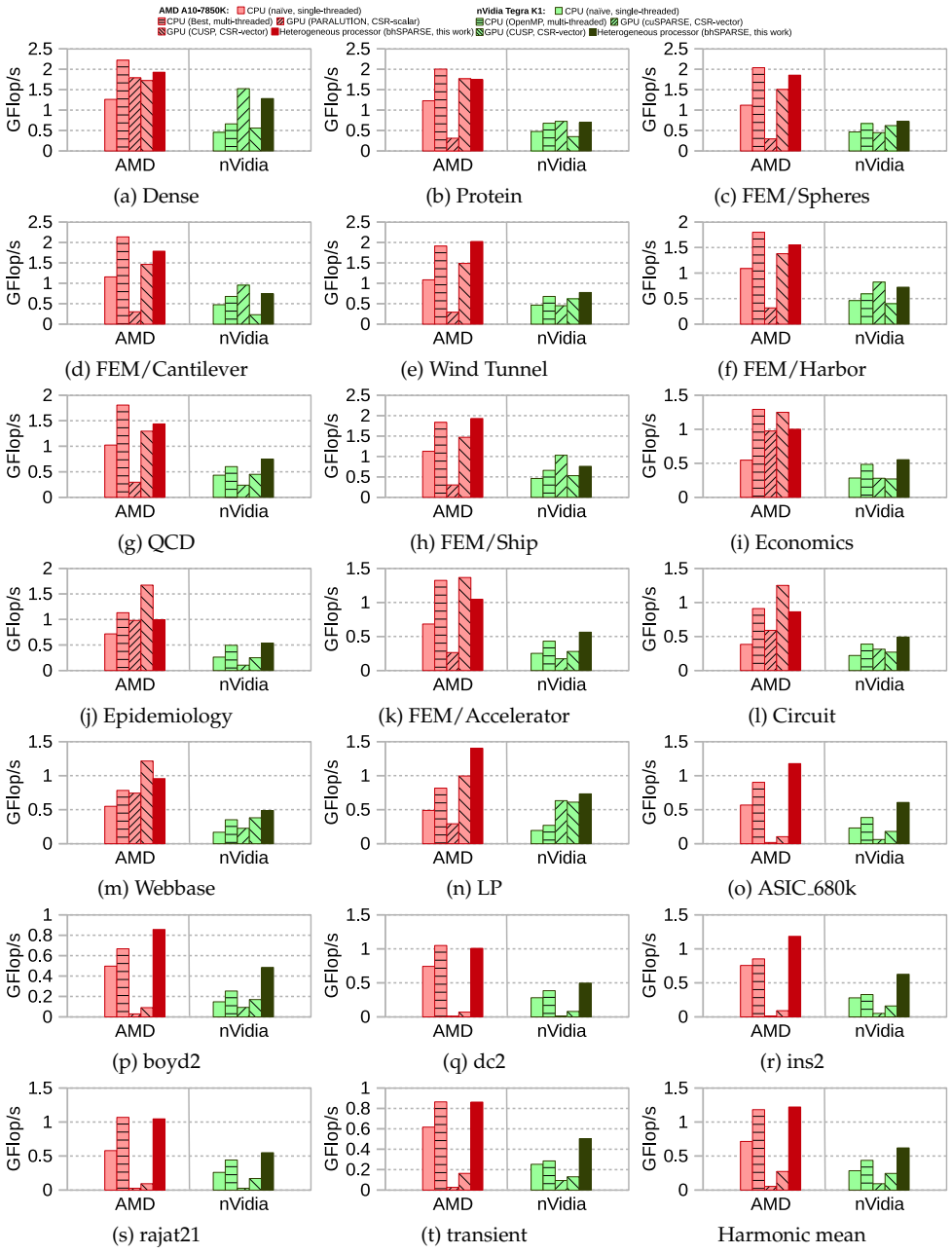


Figure 8.7: Throughput (GFlop/s) of the double precision CSR-based SpMV algorithms running on the AMD and the nVidia platforms.

method delivers up to 71.90x (94.05x) and on average 22.17x (22.88x) speedup over the single (double) precision CSR-scalar method running on the used GPU. Compared to the GPU CSR-vector method, our algorithm achieves up to 16.07x (14.43x) and on average 5.61x (4.47x) speedup. The CSR-scalar and the CSR-vector methods give very low throughput while running the last 6 irregular matrices, because of the problem of load imbalance. Further, we find that the Intel heterogeneous processor's GPU is actually faster than the AMD GPU while running the last 6 matrices. The reason is that the shorter thread-bunch (8 in Intel GPU vs. 64 in AMD GPU) brings a positive influence for saving SIMD idle cost by executing a much shorter vector width for dramatically imbalanced row distribution. On the other hand, for several very regular matrices with short rows, e.g., *Epidemiology*, the CSR-scalar method offers the best performance because of almost perfect load balance and execution of short rows without loop cost. For most regular matrices, our method delivers comparable performance over the best CPU algorithm.

In Figures 8.6 and 8.7, we can see that on the nVidia platform, our method delivers up to 5.91x (6.20x) and on average 2.69x (2.53x) speedup over the single (double) precision SpMV in the CUSP library running on the used GPU. Compared to cuSPARSE, our method has higher speedups. Since the both libraries use CSR-vector algorithm, those speedups are within expectations. Consider the Tegra K1 platform only contains one single GPU core, the problem of load imbalance on this device is not as heavy as on the above AMD platform. As a result, the speedups are not as high as those from the AMD processor. Here our method delivers on average 1.41x (1.42x) speedup over OpenMP-accelerated SpMV on the quad-core ARM CPU, while using single (double) precision benchmark.

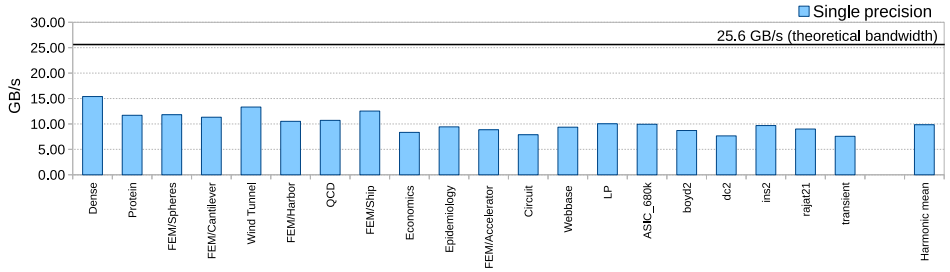
Figure 8.8 shows bandwidth utilization of our algorithm proposed in this section. We can see that the regular matrices can use bandwidth more efficiently compared to the irregular ones. Considering the throughput speedups listed above, our method can obtain higher bandwidth utilization than the other CSR-based SpMV algorithms running on GPUs.

Parameter Selection

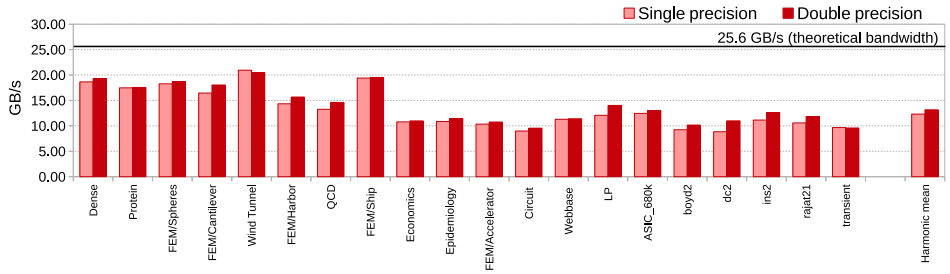
We further conduct experiments to exploit how selected parameters influence overall performance.

Figure 8.9 shows dependency of the overall performance (harmonic means of the 20 benchmarks) on the parameters, while we fix all the parameters except for parameter W (i.e., workload per thread). We can see that in general the overall performance goes up as parameter W increases. This trend matches the algorithm complexity analysis described in Section 3.3. However, when W is larger than a certain value, the overall performance degrades. The reason is that device occupancy may decrease while more on-chip scratchpad memory is allocated for WT work space of each thread-bunch.

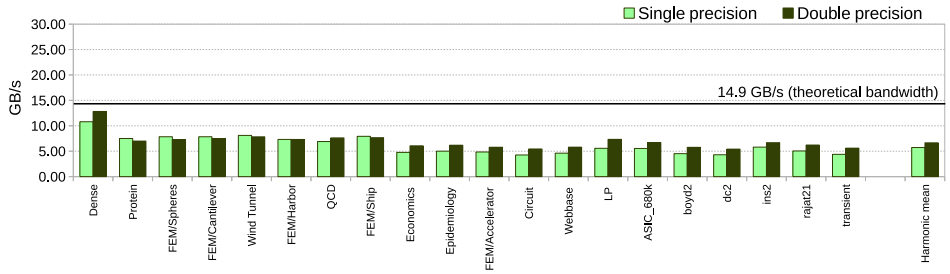
Figure 8.10 shows the trend of the overall performance while we change parameter S (i.e., the number of iterations of each thread-bunch) and fix all the other parameters. We can see that if we assign more work to each thread-bunch, a better performance can be expected. The performance improvement mainly comes from higher on-chip resource reuse.



(a) Single precision SpMV on Intel Core i3-5010U



(b) Single precision and double precision SpMV on AMD A10-7850K



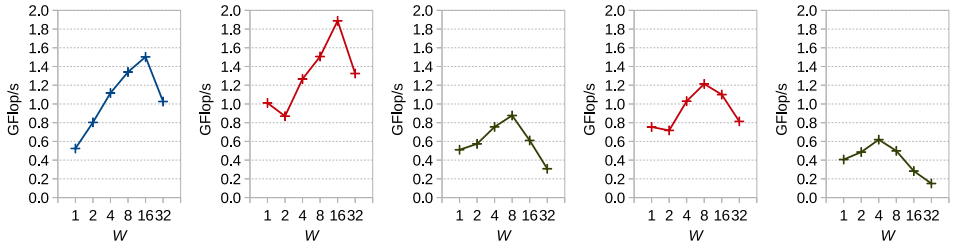
(c) Single precision and double precision SpMV on nVidia Tegra K1

Figure 8.8: Bandwidth utilization (GB/s) of our CSR-based SpMV algorithm running on the three platforms. Theoretical bandwidth from the hardware specifications are marked up using black lines.

8.5 CSR5-Based SpMV

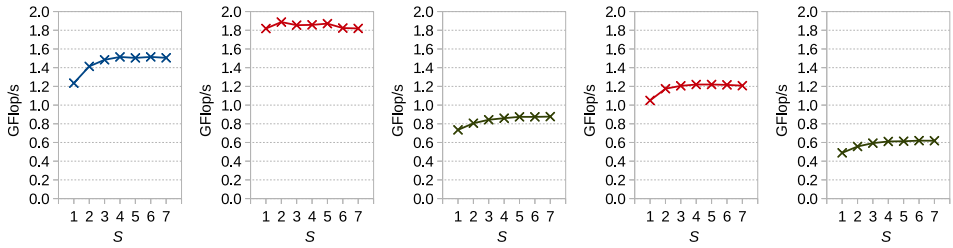
8.5.1 Algorithm Description

Recall the CSR5 format described in Chapter 6, since information (`tile_ptr`, `tile_desc`, `col_idx` and `val`) of its 2D tiles are independent of each other, all tiles can execute concurrently. On GPUs, we assign a thread bunch for each tile. On CPUs and Xeon Phi, we use OpenMP pragma for assigning the tiles to available x86 cores. Furthermore, the columns inside a tile are independent of each other as well. So we assign a thread on GPU cores or an SIMD lane on x86 cores to each column in a tile.



(a) Intel, SP (b) AMD, SP (c) nVidia, SP (d) AMD, DP (e) nVidia, DP

Figure 8.9: Single precision (SP) and double precision (DP) SpMV performance of our algorithm on the three platforms while parameter W changes and all the others fixed to the best observed values (see Table 8.1).



(a) Intel, SP (b) AMD, SP (c) nVidia, SP (d) AMD, DP (e) nVidia, DP

Figure 8.10: Single precision (SP) and double precision (DP) SpMV performance of our algorithm on the three platforms while parameter S changes and all the others fixed to the best observed values(see Table 8.1).

While running the CSR5-based SpMV, each column in a tile can extract information from `bit_flag` and label the segments in its local data to three colors: (1) *red* means a sub-segment unsealed from its top, (2) *green* means a completely sealed segment existed in the middle, and (3) *blue* means a sub-segment unsealed from its bottom. There is an exception that if a column is unsealed both from its top and from its bottom, it is colored to *red*.

Algorithm 32 shows the pseudocode of the CSR5-based SpMV algorithm. Figure 8.11 plots an example of this procedure. We can see that the green segments can directly save their partial sums to y without any synchronization, since the indices can be calculated by using `tile_ptr` and `y_offset`. In contrast, the red and the blue sub-segments have to further add their partial sums together, since they are not complete segments. For example, the sub-segments B_2, R_2 and R_3 in Figure 8.11 have contributions to the same row, thus an addition is required. This addition operation needs the fast segmented sum shown in Algorithm 13 and Figure 4.3. Furthermore, if a tile has any empty rows, the `empty_offset` array is accessed to get correct global indices in y.

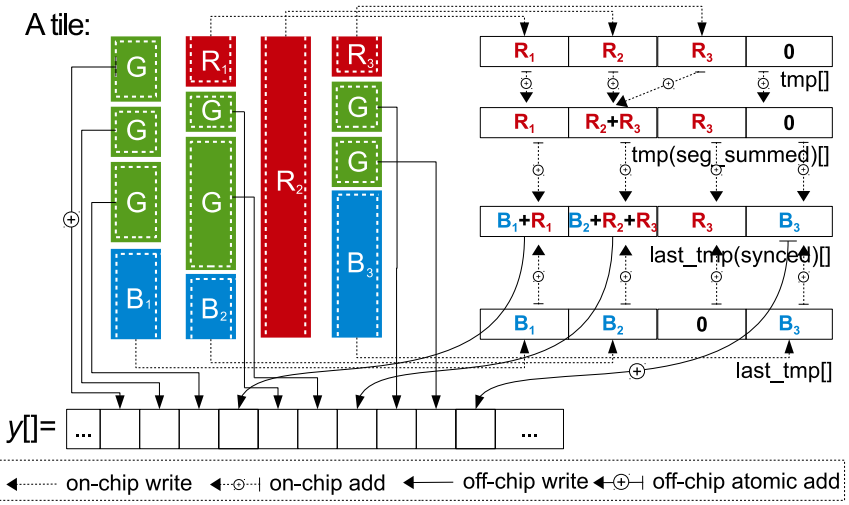


Figure 8.11: The CSR5-based SpMV in a tile. Partial sums of the green segments are directly stored to y. The red and the blue sub-segments require an extra segmented sum before issuing off-chip write.

Consider the synchronization among the tiles, since the same matrix row can be influenced by multiple 2D tiles running concurrently, the first and the last segments of a tile need to store to y by atomic add (or a global auxiliary array used in device-level reduction, scan or segmented scan [63, 161]). In Figure 8.11, the atomic add operations are highlighted by arrow lines with plus signs.

For the last entries not in a complete tile (e.g., the last two nonzero entries of the matrix in Figure 6.2), we execute a conventional CSR-vector method after all of the complete 2D tiles have been consumed. Note that even though the last tile (i.e., the incomplete one) does not have `tile_desc` arrays, it can extract a starting position from `tile_ptr`.

In Algorithm 32, we can see that the main computation (lines 5–21) only contains very basic arithmetic and logic operations that can be easily programmed on all mainstream processors with SIMD units. As the most complex part in our algorithm, the fast segmented sum operation (line 22) only requires a prefix-sum scan, which has been well-studied and can be efficiently implemented by using CUDA, OpenCL or x86 SIMD intrinsics.

8.5.2 Experimental Results

Experimental Setup

We evaluate the CSR5-based SpMV and 11 state-of-the-art formats and algorithms on four mainstream platforms: dual-socket Intel CPUs, an nVidia GPU, an AMD GPU and an Intel Xeon Phi. The platforms and participating approaches are shown in Table 8.3.

Host of the two GPUs is a machine with AMD A10-7850K APU, dual-channel

Algorithm 32 The CSR5-based SpMV for the tidth tile.

```
1: MALLOC(*tmp, ω)
2: MEMSET(*tmp, 0)
3: MALLOC(*last_tmp, ω)
4: /*use empty_offset[y_offset[i]] instead of y_offset[i] for a
   tile with any empty rows*/
5: for i = 0 to ω − 1 in parallel do
6:     sum ← 0
7:     for j = 0 to σ − 1 do
8:         ptr ← tid × ω × σ + j × ω + i
9:         sum ← sum + val[ptr] × x[col_idx[ptr]]
10:        /*check bit_flag[i][j]*/
11:        if /*end of a red sub-segment*/ then
12:            tmp[i − 1] ← sum
13:            sum ← 0
14:        else if /*end of a green segment*/ then
15:            y[tile_ptr[tid] + y_offset[i]] ← sum
16:            y_offset[i] ← y_offset[i] +1
17:            sum ← 0
18:        end if
19:    end for
20:    last_tmp[i] ← sum //end of a blue sub-segment
21: end for
22: FAST_SEGMENTED_SUM(*tmp, ω, *seg_offset)                    ▷ Alg. 13
23: for i = 0 to ω − 1 in parallel do
24:    last_tmp[i] ← last_tmp[i] + tmp[i]
25:    y[tile_ptr[tid] + y_offset[i]] ← last_tmp[i]
26: end for
27: FREE(*tmp)
28: FREE(*last_tmp)
```

DDR3-1600 memory and 64-bit Ubuntu Linux v14.04 installed. Host of the Xeon Phi is a machine with Intel Xeon E5-2680 v2 CPU, quad-channel DDR3-1600 memory and 64-bit Red Hat Enterprise Linux v6.5 installed. Detailed specifications of the used four devices are listed in Tables B.1, B.2 and B.3.

Here we evaluate double precision SpMV. So cuDPP library [76, 161], clSpMV [170] and yaSpMV [188] are not included since they only support single precision floating point as data type. Two recently published methods [103, 173] are not tested since the source code is not available to us yet.

We use OpenCL profiling scheme for timing SpMV on the AMD platform and record wall-clock time on the other three platforms. For all participating formats and algorithms, we evaluate SpMV 10 times (each time contains 1000 runs and records the average) and report the best observed result.

The testbeds	The participating formats and algorithms
Dual-socket Intel Xeon E5-2667 v3	(1) The CSR-based SpMV in Intel MKL 11.2 Update 1. (2) BiCSB v1.2 using CSB [35] with bitmasked register block [33]. (3) pOSKI v1.0.0 [36] using OSKI v1.0.1h [177, 178] kernels. (4) The CSR5-based SpMV implemented by using OpenMP and AVX2 intrinsics.
An nVidia GeForce GTX 980	(1) The best CSR-based SpMV [21] from cuSPARSE v6.5 and CUSP v0.4.0 [51]. (2) The best HYB [21] from the above two libraries. (3) BRC [8] with texture cache enabled. (4) ACSR [7] with texture cache enabled. (5) The CSR5-based SpMV implemented by using CUDA v6.5.
An AMD Radeon R9 290X	(1) The CSR-vector method [21] extracted from CUSP v0.4.0 [51]. (2) The CSR-Adaptive algorithm [80] implemented in ViennaCL v1.6.2 [154]. (3) The CSR5-based SpMV implemented by using OpenCL v1.2.
An Intel Xeon Phi 5110p	(1) The CSR-based SpMV in Intel MKL 11.2 Update 1. (2) The ESB [122] with dynamic scheduling enabled. (3) The CSR5-based SpMV implemented by using OpenMP and MIC-KNC intrinsics.

Table 8.3: The testbeds and participating formats and algorithms.

Benchmark Suite

In Table 8.4, we list 24 sparse matrices as our benchmark suite for all platforms. The first 20 matrices have been widely adopted in previous SpMV research [8, 21, 80, 122, 170, 184, 188]. The other 4 matrices are chosen since they have more diverse sparsity structures. Table A.1 in Appendix A gives more details of the matrices.

To achieve a high degree of differentiation, we categorize the 24 matrices in Table 8.4 into two groups: (1) *regular* group with the upper 14 matrices, (2) *irregular* group with the lower 10 matrices. This classification is mainly based on the minimum, average and maximum lengths of the rows. Matrix *dc2* is a representative of the group of irregular matrices. Its longest single row contains 114K nonzero entries, i.e., 15% nonzero entries of the whole matrix with 117K rows. This sparsity pattern challenges the design of efficient storage format and SpMV algorithm.

Isolated SpMV Performance

Figure 8.12 shows double precision SpMV performance of the 14 regular matrices on the four platforms. We can see that, on average, all participating algorithms deliver comparable performance. *On the CPU platform*, Intel MKL obtains the best performance on average and the other 3 methods behave similar. *On the nVidia GPU*, the CSR5 delivers the highest throughput. The ACSR format is slower than the others, because its binning strategy leads to non-coalesced memory access. *On the AMD GPU*, the CSR5 achieves the best performance. Although the dynamic assigning in

Id	Name	$Dimensions$	nnz	nnz per row (min, avg, max)
r1	Dense	2K×2K	4.0M	2K, 2K, 2K
r2	Protein	36K×36K	4.3M	18, 119, 204
r3	FEM/Spheres	83K×83K	6.0M	1, 72, 81
r4	FEM/Cantilever	62K×62K	4.0M	1, 64, 78
r5	Wind Tunnel	218K×218K	11.6M	2, 53, 180
r6	QCD	49K×49K	1.9M	39, 39, 39
r7	Epidemiology	526K×526K	2.1M	2, 3, 4
r8	FEM/Harbor	47K×47K	2.4M	4, 50, 145
r9	FEM/Ship	141K×141K	7.8M	24, 55, 102
r10	Economics	207K×207K	1.3M	1, 6, 44
r11	FEM/Accelerator	121K×121K	2.6M	0, 21, 81
r12	Circuit	171K×171K	959K	1, 5, 353
r13	Ga41As41H72	268K×268K	18.5M	18, 68, 702
r14	Si41Ge41H72	186K×186K	15.0M	13, 80, 662
i1	Webbase	1M×1M	3.1M	1, 3, 4.7K
i2	LP	4K×1.1M	11.3M	1, 2.6K, 56.2K
i3	Circuit5M	5.6M×5.6M	59.5M	1, 10, 1.29M
i4	eu-2005	863K×863K	19.2M	0, 22, 6.9K
i5	in-2004	1.4M×1.4M	16.9M	0, 12, 7.8K
i6	mip1	66K×66K	10.4M	4, 155, 66.4K
i7	ASIC_680k	683K×683K	3.9M	1, 6, 395K
i8	dc2	117K×117K	766K	1, 6, 114K
i9	FullChip	2.9M×2.9M	26.6M	1, 8, 2.3M
i10	ins2	309K×309K	2.8M	5, 8, 309K

Table 8.4: The benchmark suite.

the CSR-Adaptive method can obtain better scalability than the CSR-vector method, it still cannot achieve near perfect load balance. *On the Xeon Phi*, the CSR5 is slower than Intel MKL and the ESB format. The main reason is that the current generation of Xeon Phi can only issue up to 4 relatively slow threads per core (i.e., up to 4×60 threads in total on the used device), and thus the latency of gathering entries from vector x becomes the main bottleneck. Then reordering or partitioning nonzero entries based on the column index for better cache locality behaves well in the ESB-based SpMV. However, later on we will show that this strategy leads to very high preprocessing cost.

Figure 8.13 shows double precision SpMV performance of the 10 irregular matrices. We can see that the irregularity can dramatically impact SpMV throughput of some approaches. *On the CPU platform*, the row block method based Intel MKL is now slower than the other methods. The CSR5 outperforms the others because of better SIMD efficiency from the AVX2 intrinsics. *On the nVidia GPU*, the CSR5 brings the best performance because of the near perfect load balance. The other two irregularity-oriented formats, HYB and ACSR, behave well but still suffer from imbalanced work

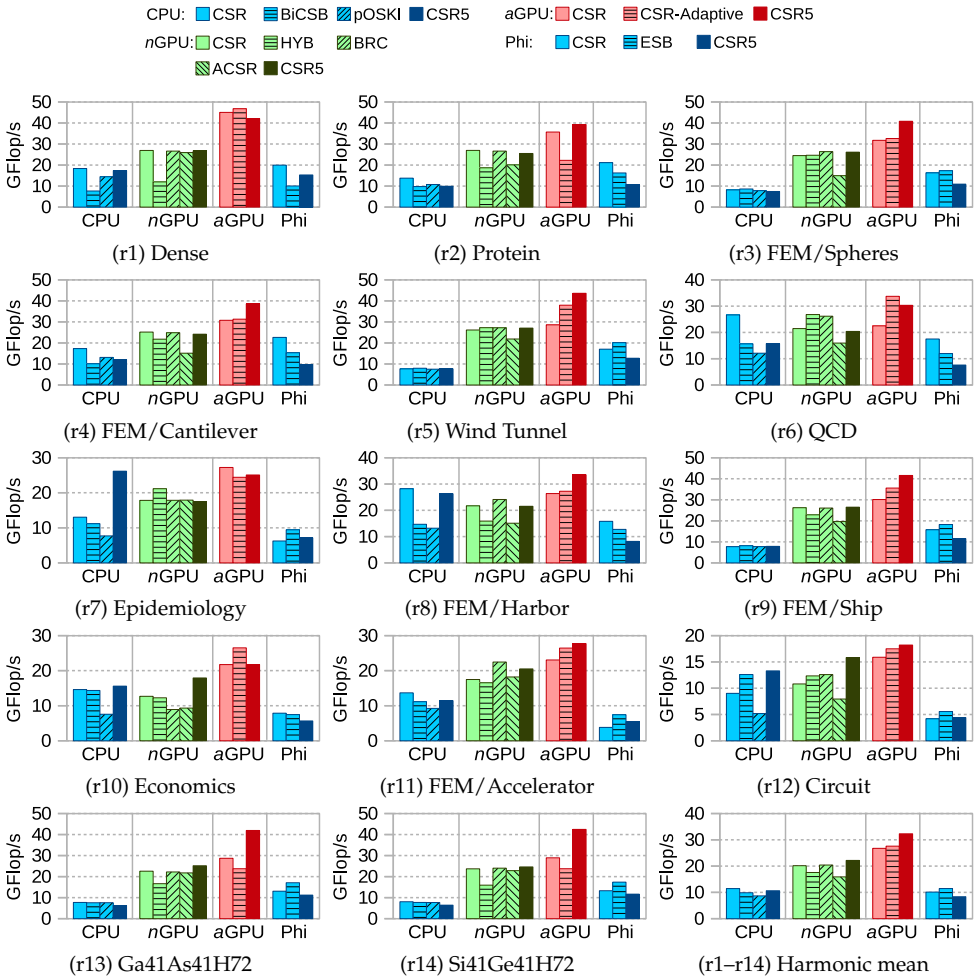


Figure 8.12: The SpMV performance of the 14 regular matrices. (*n*GPU=nVidia GPU, *a*GPU=AMD GPU)

decomposition. Note that the ACSR format is based on Dynamic Parallelism, a technical feature only available on recently released nVidia GPUs. *On the AMD GPU*, the CSR5 greatly outperforms the other two algorithms using the row block methods. Because the minimum work unit of the CSR-Adaptive method is one row, the method delivers degraded performance for matrices with very long rows[1]. *On the Xeon Phi*, the CSR5 can greatly outperform the other two methods in particular when matrices are too irregular to expose cache locality of x by the ESB format. Furthermore, since ESB is designed on top of the ELL format, it cannot obtain the best performance for

[1]Note that we use an implementation of the CSR-Adaptive from the ViennaCL Library. The AMD's version of the CSR-Adaptive may have slightly different performance.

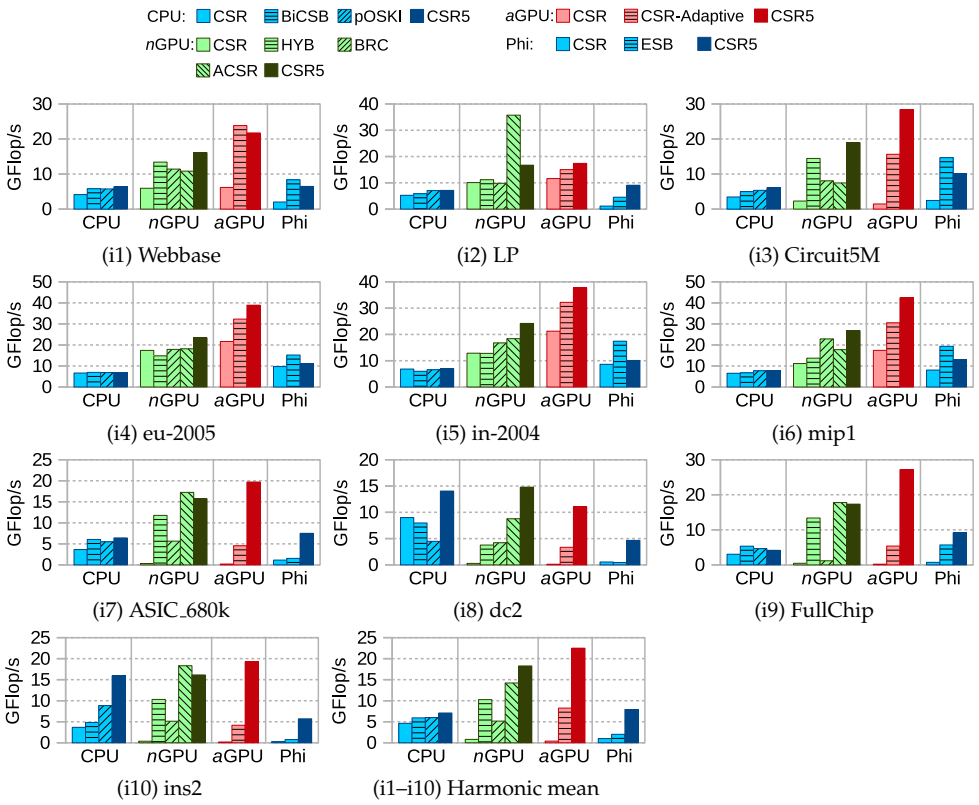


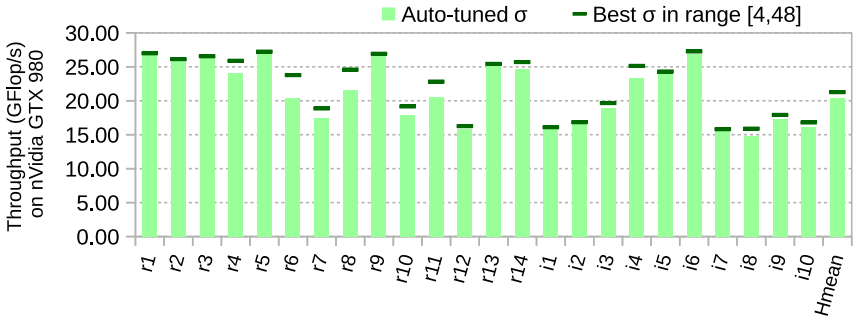
Figure 8.13: The SpMV performance of the 10 irregular matrices. (*n*GPU=nVidia GPU, *a*GPU=AMD GPU)

some irregular matrices.

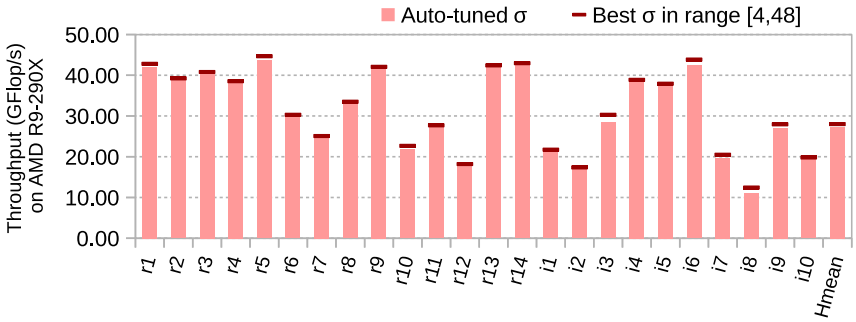
Overall, the CSR5 achieves better performance (on the two GPU devices) or comparable performance (on the two x86 devices) for the 14 regular matrices. For the 10 irregular matrices, compared to pOSKI, ACSR, CSR-Adaptive and ESB as the second best methods, the CSR5 obtains on average speedups 1.18x, 1.29x, 2.73x and 3.93x (up to 3.13x, 2.54x, 5.05x and 10.43x), respectively.

Effects of Auto-Tuning

In section 3.2, we discussed a simple auto-tuning scheme for the parameter σ on GPUs. Figure 8.14 shows its effects (the x axis is the matrix *id*s). We can see that compared to the best performance chosen from a range of $\sigma = 4$ to 48, the auto-tuned σ does not have obvious performance loss. On the nVidia GPU, the performance loss is on average -4.2%. On the AMD GPU, the value is on average -2.5%.



(a) The nVidia GTX 980 GPU.



(b) The AMD R9-290X GPU.

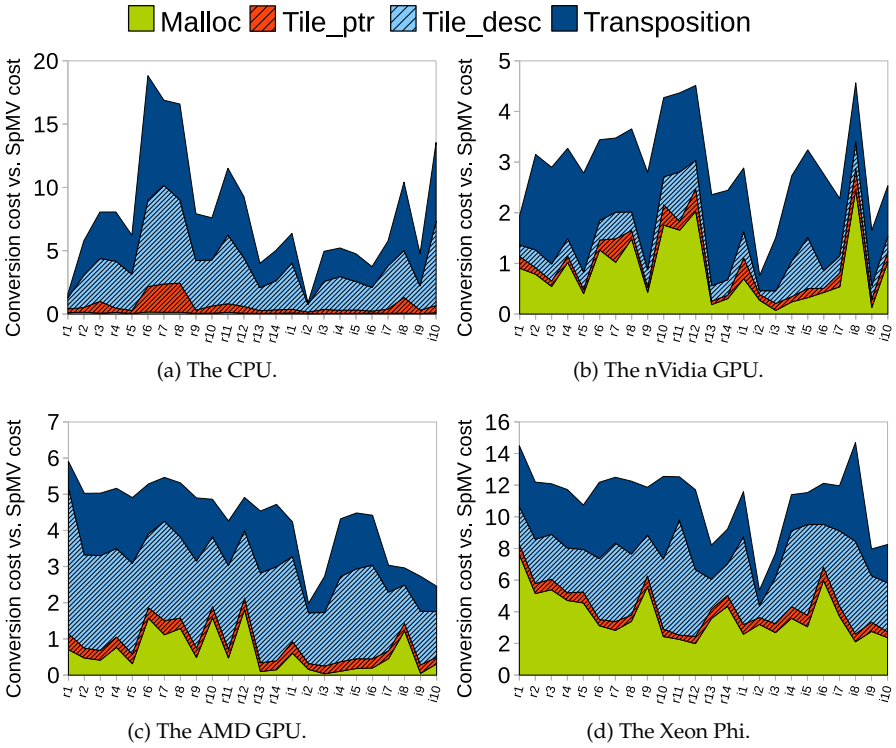
Figure 8.14: Auto-tuning effects on the two GPUs.

Format Conversion Cost

The format conversion from the CSR to the CSR5 includes four steps: (1) memory allocation, (2) generating `tile_ptr`, (3) generating `tile_desc`, and (4) transposition of `col_idx` and `val` arrays. Figure 8.15 shows the cost of the four steps for the 24 matrices (the x axis is the matrix *id*s) on the four used platforms. Cost of one single SpMV operation is used for normalizing format conversion cost on each platform. We can see that the conversion cost can be on average as low as the overhead of a few SpMV operations on the two GPUs. On the two x86 platforms, the conversion time is longer (up to cost of around 10–20 SpMV operations). The reason is that the conversion code is manually SIMDized using CUDA or OpenCL on GPUs, but only auto-parallelized by OpenMP on x86 processors.

Iteration-Based Scenarios

Since both the preprocessing (i.e., format conversion from a basic format) time and the SpMV time are important for real-world applications, we have designed an iteration-based benchmark. This benchmark measures the overall performance of a solver with n iterations. We assume the input matrix is already stored in the CSR format. So the overall cost of using the CSR format for the scenarios is nT^{csr}_{spmv}, where T^{csr}_{spmv} is



(a) The CPU.

(b) The nVidia GPU.

(c) The AMD GPU.

(d) The Xeon Phi.

Figure 8.15: The normalized format conversion cost.

execution time of one CSR-based SpMV operation. For a new format, the overall cost is $T_{pre}^{new} + nT_{spmv}^{new}$, where T_{pre}^{new} is preprocessing time and the T_{spmv}^{new} is one SpMV time using the new format. Thus we can calculate speedup of a new format over the CSR format in the scenarios, through $(nT_{spmv}^{csr})/(T_{pre}^{new} + nT_{spmv}^{new})$.

Tables 8.5 and 8.6 show the new formats' preprocessing cost (i.e., $T_{pre}^{new}/T_{spmv}^{new}$) and their speedups over the CSR format in the iteration-based scenarios when $n = 50$ and $n = 500$. The emboldened font in the tables shows the highest positive speedups on each platform. The compared baseline is the fastest CSR-based SpMV implementation (i.e., Intel MKL, nVidia cuSPARSE/CUSP, CSR-vector from CUSP, and Intel MKL, respectively) on each platform. We can see that because of the very low preprocessing overhead, the CSR5 can further outperform the previous methods when doing 50 iterations and 500 iterations. Although two GPU methods, the ACSR format and the CSR-Adaptive approach, in general have shorter preprocessing time, they suffer from lower SpMV performance and thus cannot obtain the best speedups. On all platforms, the CSR5 always achieves the highest overall speedups. Moreover, the CSR5 is the only format that obtains higher performance than the CSR format when only 50 iterations are required.

Benchmark	The 14 regular matrices				
Metrics	Preprocessing to SpMV ratio	Speedup of #iter.=50		Speedup of #iter.=500	
		avg	best	avg	best
CPU-BiCSB	538.01x	0.06x	0.11x	0.35x	0.60x
CPU-pOSKI	12.30x	0.43x	0.88x	0.57x	0.99x
CPU-CSR5	**6.14x**	0.52x	0.74x	0.59x	0.96x
*n*GPU-HYB	13.73x	0.73x	0.98x	0.92x	1.21x
*n*GPU-BRC	151.21x	0.26x	0.31x	0.80x	0.98x
*n*GPU-ACSR	**1.10x**	0.68x	0.93x	0.72x	1.03x
*n*GPU-CSR5	3.06x	**1.04x**	**1.34x**	**1.10x**	**1.45x**
*a*GPU-CSR-Adaptive	**2.68x**	1.00x	1.33x	1.07x	1.48x
*a*GPU-CSR5	4.99x	**1.04x**	**1.39x**	**1.14x**	**1.51x**
Phi-ESB	922.47x	0.05x	0.15x	0.33x	0.88x
Phi-CSR5	**11.52x**	0.54x	**1.14x**	0.65x	**1.39x**

Table 8.5: Preprocessing cost and its impact on the iteration-based scenarios.

Benchmark	The 10 irregular matrices				
Metrics	Preprocessing to SpMV ratio	Speedup of #iter.=50		Speedup of #iter.=500	
		avg	best	avg	best
CPU-BiCSB	331.77x	0.13x	0.24x	0.60x	1.07x
CPU-pOSKI	10.71x	0.62x	1.66x	0.83x	2.43x
CPU-CSR5	**3.69x**	0.91x	**2.37x**	**1.03x**	**2.93x**
*n*GPU-HYB	28.59x	1.86x	13.61x	2.77x	25.57x
*n*GPU-BRC	51.85x	1.17x	7.60x	2.49x	15.47x
*n*GPU-ACSR	3.04x	5.05x	41.47x	5.41x	51.95x
*n*GPU-CSR5	**1.99x**	**6.43x**	**48.37x**	**6.77x**	**52.31x**
*a*GPU-CSR-Adaptive	**1.16x**	3.02x	27.88x	3.11x	28.22x
*a*GPU-CSR5	3.10x	**5.72x**	**135.32x**	**6.04x**	**141.94x**
Phi-ESB	222.19x	0.27x	1.15x	1.30x	2.96x
Phi-CSR5	**9.45x**	**3.43x**	**18.48x**	**4.10x**	**21.18x**

Table 8.6: Preprocessing cost and its impact on the iteration-based scenarios.

8.6 Comparison to Recently Developed Methods

In recent years, some new formats have been designed for SpMV operation on various processor architectures.

Because of less off-chip memory access and better on-chip memory localization, SpMV using **block-based formats**, such as OSKI [177, 179, 178], pOSKI [36], CSB [35, 33], BELLPACK [45], BCCOO/BCCOO+ [188], BRC [8] and RSB [129], attracted the most attention. However, block-based formats heavily rely on sparsity structure, meaning that the input matrix is required to have a block structure to meet

potential block layout. Therefore, block-based formats are mainly suitable for some matrices generated from scientific computation problems, but may not fit irregular matrices generated from graph applications. Our methods proposed in this chapter is insensitive to the sparsity structure of input matrix, thus a generally better performance is achieved.

A lot of research has focused on improving **row block method** CSR-based SpMV. Williams et al. [184] proposed multiple optimization techniques for SpMV on multicore CPUs and Cell B.E. processor. Nishtala et al. [139] designed a high-level data partitioning method for SpMV to achieve better cache locality on multicore CPUs. Pichel et al. [147] evaluated how reordering techniques influence performance of SpMV on GPUs. Baskaran and Bordawekar [16] improved off-chip and on-chip memory access patterns of SpMV on GPUs. Reguly and Giles [151] improved thread cooperation for better GPU cache utilization. Ashari et al. [7] utilized static reordering and the Dynamic Parallelism scheme offered by nVidia GPUs for fast SpMV operation. Greathouse et al. [80] grouped contiguous rows for better runtime load balancing on GPUs. LightSpMV [123] proposed to dynamically distribute matrix rows over warps in order for more balanced CSR-based SpMV without the requirement of generating auxiliary data structures, and implemented this approach using atomic operations and warp shuffle functions as the fundamental building blocks. However, again, the row block methods cannot achieve good performance for input matrix with dramatically imbalanced row distribution. In contrast, our methods are independent with the sparsity structure of input matrix.

As mentioned, using **segmented sum method** as a building block is potentially a better generic method for the CSR-based SpMV. An early segmented sum method GPU SpMV was introduced by Sengupta et al. [161] and Garland [76] and implemented in the cuDPP library [86]. But the cost of segmented sum and global memory access degrade overall SpMV performance. Zhang [195] improved backward segmented scan for a better cache efficiency and implemented the CSR-based SpMV on multicore CPUs. Recently, nVidia's Modern GPU library [18] implemented an improved reduction method, which has been used as a back-end of cuDPP. However, its performance still suffered by pre- and post-processing empty rows in global memory space. The segmented sum methods have been used in two recently published papers [173, 188] for the SpMV on either GPUs or Xeon Phi. However, both of them need to store the matrix in COO-like formats to utilize the segmented sum. Our CSR-based SpMV methods, in contrast, uses scratchpad memory more efficiently and utilizes the two types of cores in a heterogeneous processor for better workload distribution. Moreover, the CSR5 format saves useful row index information in a compact way, and thus can be more efficient both for the format conversion and for the SpMV operation.

Compared with the CSR5 work designed for cross-platform SpMV on CPUs, GPUs and Xeon Phi, our CSR-based SpMV approach does not need to process any format conversion or generate any auxiliary data for the input CSR matrix. Consider the format conversion from the CSR to the CSR5 merely needs the cost of a few SpMV operations, the CSR5-based SpMV and the CSR-based SpMV can find their own application scenarios, such as solvers with different number of iterations.

9. Level 3: Sparse Matrix-Matrix Operations

9.1 Overview

General matrix-matrix multiplication (GEMM) [114, 23, 128] is one of the most crucial operations in computational science and modeling. The operation multiplies a matrix A of size $m \times k$ with a matrix B of size $k \times n$ and gives a resulting matrix C of size $m \times n$. In many linear solvers and graph problems such as algebraic multigrid method (AMG) [20], breadth first search [79], finding shortest path [39], colored intersection [95] and sub-graphs [175], it is required to exploit sparsity of the two input matrices and the resulting matrix because their dense forms normally need huge storage space and computation cost for the zero entries. Therefore SpGEMM becomes a common building block in these applications.

Compared to CPUs, modern graphics processing units (GPUs) promise much higher peak floating-point performance and memory bandwidth. Thus a lot of research has concentrated on GPU accelerated sparse matrix-dense vector multiplication [21, 120, 121] and sparse matrix-dense matrix multiplication [143, 176] and achieved relatively attractive performance. However, despite the prior achievements on these GPU sparse BLAS routines, massive parallelism in GPUs is still significantly underused for the SpGEMM algorithm, because it has to handle three more challenging problems: (1) the number of nonzero entries in the resulting matrix is unknown in advance, (2) very expensive parallel insert operations at random positions in the resulting matrix dominate the execution time, and (3) load balancing must account for sparse data in both input matrices with diverse sparsity structures.

Previous GPU SpGEMM methods [20, 57, 140, 51, 52, 83] have proposed a few solutions for the above problems and demonstrated relatively good time and space complexity. However, the experimental results showed that they either only work best for fairly regular sparse matrices [57, 140, 83], or bring extra high memory overhead for matrices with some specific sparsity structures [20, 51, 52]. Moreover, in the usual sense, none of these methods can constantly outperform well optimized SpGEMM approach [92] for multicore CPUs.

9.2 Contributions

The work described in this chapter particularly focuses on improving GPU SpGEMM performance for matrices with arbitrary irregular sparsity structures by proposing more efficient methods to solve the above three problems on GPUs and emerging CPU-GPU heterogeneous processors.

This chapter makes the following contributions:

- A 4-stage framework for implementing SpGEMM on manycore platforms including homogeneous GPUs and heterogeneous processors composed of CPU cores, GPU cores and shared virtual memory. This framework effectively organizes memory allocation, load balancing and GPU kernel launches.
- A hybrid method that initially allocates memory of upper bound size for short rows and progressively allocates memory for long rows. The experimental results show that our method saves a large amount of global memory space and efficiently utilizes the very limited on-chip scratchpad memory.
- An efficient parallel insert method for long rows of the resulting matrix by using the fastest merge algorithm available on GPUs. We make an experimental evaluation and choose GPU merge path algorithm from five candidate GPU merge approaches.
- A load balancing oriented heuristic method that assigns rows of the resulting matrix to multiple bins with different subsequent computational methods. Our approach guarantees load balancing in all calculation steps.

Our framework and corresponding algorithms delivers excellent performance in two experimental scenarios: (1) calculating triple matrix Galerkin products (i.e., $P^T AP$) in AMG for 2D and 3D Poisson problems, and (2) computing matrix squaring (i.e., A^2) on a benchmark suite composed of 23 sparse matrices with diverse sparsity structures.

In the context of Galerkin products, our method constantly outperforms the state-of-the-art GPU SpGEMM methods in two vendor supplied libraries cuSPARSE and CUSP. Average speedups of 1.9x (up to 2.6x) and 1.7x (up to 2.7x) are achieved when compared to cuSPARSE and CUSP, respectively.

In the context of matrix squaring, more comparison methods are included. *First*, on two nVidia GPUs (i.e., a GeForce GTX Titan Black and a GeForce GTX 980), compared with cuSPARSE and CUSP, our approach delivers on average 3.1x (up to 9.5x) and 4.6x (up to 9.9x) speedups, respectively. *Second*, compared to a recently developed CUDA-specific SpGEMM method RMerge [83], our method offers on average 2.5x (up to 4.9x) speedup on the nVidia GeForce GTX 980 GPU. *Third*, compared to the SpGEMM method in the latest Intel Math Kernel Library (MKL) on a six-core Xeon E5-2630 CPU and quad-channel system memory, our method gives on average 2.4x (up to 5.2x) and 2.1x (up to 4.2x) speedups on the nVidia GeForce GTX 980 GPU and an AMD Radeon R9 290X GPU, respectively.

Furthermore, our approach can utilize re-allocatable memory controlled by CPU-GPU heterogeneous processors. On an AMD A10-7850K heterogeneous processor,

compared to merely using its GPU cores, our framework delivers on average 1.2x (up to 1.8x) speedup while utilizing re-allocatable shared virtual memory in the system.

9.3 Basic Method

9.3.1 Gustavson's Algorithm

For the sake of generality, the SpGEMM algorithm description starts from discussion of the GEMM and gradually takes sparsity of the matrices A, B and C into consideration. For the matrix A, we write a_{ij} to denote the entry in the ith row and the jth column of A and a_{i*} to denote the vector consisting of the ith row of A. Similarly, the notation a_{*j} denotes the jth column of A. In the GEMM, the ith row of the resulting matrix C can be defined by

$$c_{i*} = (a_{i*} \cdot b_{*1}, a_{i*} \cdot b_{*2}, \ldots, a_{i*} \cdot b_{*p}),$$

where the operation $\cdot$ is the dot product of the two vectors.

We first give consideration to the sparsity of the matrix A. Without loss of generality, we assume that the ith row of A only consists of two nonzero entries in the kth and the lth column, respectively. Thus a_{i*} becomes (a_{ik}, a_{il}). Since all other entries are zeros, we do not record them explicitly and ignore their influence on the dot products in the calculation of the ith row of C. Then we obtain

$$c_{i*} = (a_{ik}b_{k1} + a_{il}b_{l1}, a_{ik}b_{k2} + a_{il}b_{l2}, \ldots, a_{ik}b_{kp} + a_{il}b_{lp}).$$

We can see in this case, only entries in the kth and the lth row of B have contribution to the ith row of C. Then row vector form instead of column vector form is used for the matrix B. So we obtain

$$c_{i*} = a_{ik}b_{k*} + a_{il}b_{l*}.$$

Since the matrix B is sparse as well, again without loss of generality, we assume that the kth row of B has only two nonzero entries in the rth and the tth column, and the lth row of B also has only two nonzero entries in the sth and the tth column. So the two rows are given by $b_{k*} = (b_{kr}, b_{kt})$ and $b_{l*} = (b_{ls}, b_{lt})$. Then

$$c_{i*} = a_{ik}(b_{kr}, b_{kt}) + a_{il}(b_{ls}, b_{lt}).$$

Because the matrix C is also sparse and the ith row of C only has three nonzero entries in the rth, the sth and the tth column, the row can be given by

$$c_{i*} = (c_{ir}, c_{is}, c_{it}),$$

where $c_{ir} = a_{ik}b_{kr}$, $c_{is} = a_{il}b_{ls}$ and $c_{it} = a_{ik}b_{kt} + a_{il}b_{lt}$.

In general there are more nonzero entries per rows of the matrices A, B and C. But from the above derivation we can see that the SpGEMM can be represented by operations on row vectors of the matrices. Therefore, in this work we store all sparse matrices in the CSR format. Actually compressed sparse column (CSC) format is also

widely used for sparse matrices stored in column-major order [77]. The SpGEMM in the CSC format is almost the same as in the CSR format except rows are changed to columns and vice versa.

The above CSR-based SpGEMM algorithm can be performed by pseudocode in Algorithm 33. An early description of this algorithm was given by Gustavson [85].

Algorithm 33 Pseudocode for the SpGEMM.

```
1:  for each a_{i*} in the matrix A do
2:      set c_{i*} to ∅
3:      for each nonzero entry a_{ij} in a_{i*} do
4:          load b_{j*}
5:          for each nonzero entry b_{jk} in b_{j*} do
6:              value ← a_{ij} b_{jk}
7:              if c_{ik} ∉ c_{i*} then
8:                  insert c_{ik} to c_{i*}
9:                  c_{ik} ← value
10:             else
11:                 c_{ik} ← c_{ik} + value
12:             end if
13:         end for
14:     end for
15: end for
```

9.3.2 Performance Considerations

Memory Pre-Allocation For the Resulting Matrix

Compared to SpGEMM, other sparse matrix multiplication operations (e.g., multiplication of sparse matrix and dense matrix [143, 176, 158] and its special case sparse matrix-vector multiplication [21, 120, 121, 184, 33]) pre-allocate a dense resulting matrix or vector. Thus the size of the result of the multiplication is trivially predictable, and the corresponding entries are stored to predictable memory addresses. However, because the number of nonzero entries in the resulting sparse matrix C is unknown in advance, precise memory allocation of the SpGEMM is impossible before real computation. Moreover, physical address of each new entry is unknown either (consider line 7 in Algorithm 33, the position k is only a column index that cannot trivially map to a physical address on memory space).

To solve this problem, the previous SpGEMM algorithms proposed four different solutions: (1) precise method, (2) probabilistic method, (3) upper bound method, and (4) progressive method.

The first method, *precise method*, pre-computes a simplified SpGEMM in the same computational pattern. We can imagine that multiplication of sparse boolean matrices is more efficient than multiplication of sparse floating-point matrices. RMerge algorithm and the SpGEMM methods in cuSPARSE and MKL are representatives of this approach. Even though the pre-computation generates precise size of $nnz(C)$,

this method is relatively expensive since the SpGEMM operation in the same pattern is executed twice.

The second method, *probabilistic method*, estimates an imprecise $nnz(C)$. This group of approaches [4, 47, 144] are based on random sampling and probability analysis on the input matrices. Since they do not guarantee a safe lower bound for the resulting matrix C and extra memory has to be allocated while the estimation fails, they were mostly used for estimating the shortest execution time of multiplication of multiple sparse matrices.

The third method, *upper bound method*, computes an upper bound of the number of nonzero entries in the resulting matrix C and allocates corresponding memory space. Numerically, the upper bound size equals $nnz(\widehat{C})$, or half of $flops$, the number of necessary arithmetic operations. The ESC algorithms use this method for memory pre-allocation. Even though this approach saves cost of the pre-computation in the precise method, it brings another problem that the intermediate matrix $\widehat{C}$ may be too large to fit in the device global memory. Since the SpGEMM algorithm does not take into consideration cancellation that eliminates zero entries generated by arithmetic operations, the resulting matrix is normally larger than the input matrices. Table 9.2 shows that $nnz(\widehat{C})$ is much larger than $nnz(C)$ while squaring some matrices. For example, the sparse matrix *Wind Tunnel* generates 626.1 million nonzero entries (or 7.5 GB memory space for 32-bit index and 64-bit value) for the intermediate matrix $\widehat{C}$ while the real product C (i.e., A^2) only contains 32.8 million nonzero entries. Although the upper bound method can partition the intermediate matrix $\widehat{C}$ into multiple sub-matrices, higher global memory pressure may reduce overall performance.

The last method, *progressive method*, first allocates memory of a proper size, starts sparse matrix computation and re-allocates the buffer if larger space is required. Some CPU sparse matrix libraries use this method. For instance, sparse matrix computation in the Matlab [77] increases the buffer by a ratio of 50% if the current memory space is exhausted.

Since the upper bound method sacrifices space efficiency for the sake of improved performance and the progressive method is good at saving space, we use a hybrid method composed of the both approaches. However, compared to the relatively convenient upper bound method, it is hard to directly implement a progressive method for discrete GPUs. The reason is that although modern GPU devices have the ability of allocating device global memory while kernels are running, they still cannot re-allocate device memory on the fly. We will describe our hybrid method designed for discrete GPUs in the next section.

On the other hand, emerging heterogeneous processors, composed of multiple CPU cores and GPU cores in one chip, supply both flexibility and efficiency. In this architecture, integrated GPU cores can directly use system memory allocated by the CPU part. Then data transfer through connection interfaces such as PCIe link can be avoided to obtain higher performance [82]. This gives our SpGEMM algorithm a chance to let integrated GPUs use re-allocatable system memory for a better overall performance. Later on, we will show the corresponding performance gain by using an AMD APU.

Parallel Insert Operations

As shown in Algorithm 33, for each trivial arithmetic computation (line 6), one much more expensive insert operation (lines 7–11) is required. To the best of our knowledge, none of the previous GPU SpGEMM methods takes into account that the input sequence (line 4) is ordered because of the CSR format. One of our algorithm design objectives is to efficiently utilize this property. Based on experiments by Kim et al. [98], as the SIMD units are getting wider and wider, merge sort methods will outperform hash table methods on the join-merge problem, which is a similar problem in the SpGEMM. Then our problem converts to finding a fast GPU method for merging sorted sequences. Later on we will describe our strategy in detail.

Load Balancing

Because distribution patterns of nonzero entries in both input sparse matrices can be very diverse (consider plots of the matrices in Table 9.2), input space-based data decomposition [57, 171] normally does not bring efficient load balancing. One exception is that computing SpGEMM for huge sparse matrices on large scale distributed memory systems, 2D and 3D decomposition on input space methods demonstrated good load balancing and scalability by utilizing efficient communication strategies [79, 13, 35]. However, in this chapter we mainly consider load balancing for fine-grained parallelism in GPU and CPU-GPU shared memory architectures.

Therefore we use the other group of load balancing methods based on output space decomposition. Dalton et al. [52] presented a method that sorts rows of the intermediate matrix $\widehat{C}$, divides it into 3 sub-matrices that include the rows in different size ranges, and uses differentiated ESC methods for the sub-matrices. We have a similar consideration, but our implementation is completely different. We do not strictly sort rows of the intermediate matrix $\widehat{C}$ but just assign rows to a fixed number of bins through a much faster linear time traverse on CPU. Moreover, we decompose the output space in a more detailed way that guarantees much more efficient load balancing. We will demonstrate that our method is always load balanced in all stages for maximizing resource utilization of GPUs.

9.4 CSR-Based SpGEMM

9.4.1 Framework

Our SpGEMM framework includes four stages: (1) calculating upper bound, (2) binning, (3) computing the resulting matrix, and (4) arranging data. Figure 9.1 plots this framework.

9.4.2 Algorithm Design

The first stage, calculating upper bound, generates the upper bound number of nonzero entries in each row of the resulting matrix C. We create an array U of size m, where m is the number of rows of C, for the upper bound sizes of the rows. We use

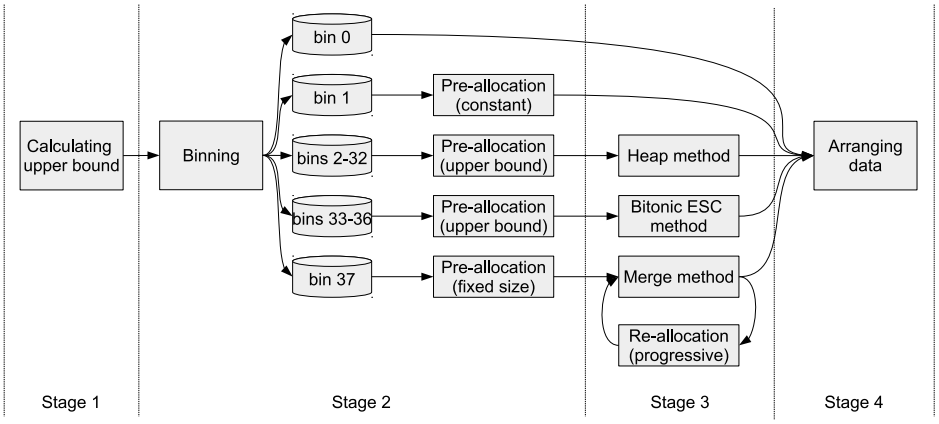


Figure 9.1: The SpGEMM framework composed of four stages.

one GPU thread for computing each entry of the array U. Algorithm 34 describes this procedure.

Algorithm 34 Pseudocode for the first stage on GPUs.

```
1: for each entry u_i in U in parallel do
2:     u_i ← 0
3:     for each nonzero entry a_ij in a_i* do
4:         u_i ← u_i + nnz(b_j*)
5:     end for
6: end for
```

The second stage, binning, deals with load balancing and memory pre-allocation. We first allocate 38 bins and put them into five bin groups. The bins contain the indices of the entries in the array U and present as one array of size m with 38 segments. Then all rows are assigned to corresponding bins according to the number of nonzero entries. Finally, based on the sizes of the bins, we allocate a temporary matrix for nonzero entries in the resulting matrix C.

The first bin group includes one bin that contains the indices of the rows of size 0. The second bin group also only has one bin that contains the indices of the rows of size 1. Because the rows in the first two bins only require trivial operations, they are excluded from subsequent more complex computation on GPUs. Thus a better load balancing can be expected.

The third bin group is composed of 31 bins that contain the indices of the rows of sizes 2–32, respectively. Since the sizes of these rows are no more than the size of a single thread bunch (32 in current nVidia GPUs or 64 in current AMD GPUs) and these rows require non-trivial computation, using one thread bunch or one thread group for each row cannot bring efficient instruction throughput on GPUs. Therefore, we use one thread for each row. Further, because each bin only contains the rows of the same upper bound size, the bins can be executed separately on GPUs with

different kernel programs for efficient load balancing. In other words, 31 GPU kernel programs will be executed for the 31 bins, if not empty.

The fourth bin group consists of 4 bins that contain the indices of the rows located in size ranges 33–64, 65–128, 129–256 and 257–512, respectively. The rows of these sizes are grouped because of three reasons: (1) each of them is large enough to be efficiently executed by a thread group, (2) each of them is small enough for scratchpad memory (48 kB per core in current nVidia Kepler GPUs, 96 kB per core in current nVidia Maxwell GPUs and 64 kB per core in current AMD Graphics Core Next, or GCN, GPUs), and (3) the final sizes of these rows in the resulting matrix C are predictable in a reasonable small range (no less than the lower bound of size 1 and no more than the corresponding upper bound sizes). Even though the rows in each bin do not have exactly the same upper bound size, a good load balancing still can be expected because each row is executed by using one thread group and inter-thread group load balancing is naturally guaranteed by the GPU low-level scheduling sub-systems.

The fifth bin group includes the last bin that contains the indices of the rest of the rows of size larger than 512. These rows have two common features: (1) their sizes can be too large (recall $nnzr(\widehat{C})$ in Table 9.2) to fit in the scratchpad memory, and (2) predicting the final sizes of these rows to a small range (scratchpad memory level) is not possible in advance. Therefore, we execute them in a unified progressive method described later. Again because we use one thread group for each row, load balancing is naturally guaranteed.

Since we do not use precise method for memory pre-allocation, a temporary memory space for the resulting matrix C is required. We design a hybrid method that allocates a CSR format sparse matrix $\widetilde{C}$ of the same size of the resulting matrix C as temporary matrix. We set $nnz(\widetilde{c}_{i*})$ to u_i while the row index i is located in the bin groups 1–4 because compared with modern GPU global memory capacity, the total space requirement of these rows is relatively small. For the rows in the bin group 5, we set $nnz(\widetilde{c}_{i*})$ to a fixed size 256 since normally this is an efficient working size for the scratchpad memory. Therefore, we can see that if all of the indices of the rows are in the bin groups 1–4, our hybrid method converts to the upper bound method, on the other extreme end, our method converts to the progressive method. But generally, we obtain benefits from the both individual methods. The stage 2 is executed on CPU since it only requires a few simple linear time traverses, which are more efficient for the CPU cache sub-systems. The pseudocode is shown in Algorithm 35.

The third stage, computing the resulting matrix, generates $nnz(c_{i*})$ and the final nonzero entries stored in the temporary matrix $\widetilde{C}$.

For the rows in the bin groups 1–2, we simply update the numbers of corresponding nonzero entries. For the rows in the bin groups 3–5, we use three totally different methods: (1) heap method, (2) bitonic ESC method, and (3) merge method, respectively. Note that each bin has a counter (at the host side) that records the number of rows included. So the host can easily decide if a GPU kernel will be issued for a certain bin. In other words, our approach only issue kernels for non-empty bins.

The heap method described in Section 7.4.1 is used for each row in the bin group 3. For the rows in each bin of the bin group 4, a typical ESC algorithm described in Section 7.4.2 is used. For the rows in the bin group 5, our method inserts each input nonzero entry to the corresponding row of the resulting matrix C (lines 7–11 in

Algorithm 35 Pseudocode for the second stage on a CPU core.

for each entry u_i in U **do**
  if $u_i = 0$ **then** ▷ The 1st bin group
    `insert` i `to` bin_0
    $nnz(\widetilde{c}_{i*}) \leftarrow 0$
  else if $u_i = 1$ **then** ▷ The 2nd bin group
    `insert` i `to` bin_1
    $nnz(\widetilde{c}_{i*}) \leftarrow 1$
  else if $u_i \geq 2$ && $u_i \leq 32$ **then** ▷ The 3rd bin group
    `insert` i `to` bin_{u_i}
    $nnz(\widetilde{c}_{i*}) \leftarrow u_i$
  else if $u_i \geq 33$ && $u_i \leq 64$ **then** ▷ The 4th bin group
    `insert` i `to` bin_{33}
    $nnz(\widetilde{c}_{i*}) \leftarrow u_i$
  else if $u_i \geq 65$ && $u_i \leq 128$ **then** ▷ The 4th bin group
    `insert` i `to` bin_{34}
    $nnz(\widetilde{c}_{i*}) \leftarrow u_i$
  else if $u_i \geq 129$ && $u_i \leq 256$ **then** ▷ The 4th bin group
    `insert` i `to` bin_{35}
    $nnz(\widetilde{c}_{i*}) \leftarrow u_i$
  else if $u_i \geq 257$ && $u_i \leq 512$ **then** ▷ The 4th bin group
    `insert` i `to` bin_{36}
    $nnz(\widetilde{c}_{i*}) \leftarrow u_i$
  else if $u_i > 512$ **then** ▷ The 5th bin group
    `insert` i `to` bin_{37}
    $nnz(\widetilde{c}_{i*}) \leftarrow 256$
  end if
end for
$nnz(\widetilde{C}) \leftarrow \sum nnz(\widetilde{c}_{i*})$

Algorithm 33) in parallel. Because the resulting rows in the bin group 5 may require more involved entries, method described in Section 7.4.3 is used.

As we allocate a limited scratchpad memory space for the resulting sequence of the bin group 5, a potential overflow may happen. In this case, we first compare total size of the two sequences (note that the input sequence is in the thread registers, but not in the scratchpad memory yet) with the allocated size of the resulting sequence in the scratchpad memory. If a merge operation is not allowed, our method records current computation position as a checkpoint and dumps the resulting sequence from the scratchpad memory to the global memory. Then the host allocates more global memory (we use 2x each time) and re-launches kernel with a 2x large scratchpad memory setting. The relaunched kernels obtain checkpoint information, and load existing results to the scratchpad memory and continue the computation. The global memory dumping and reloading bring an extra overhead, but actually it does not affect the total execution time too much because of three reasons: (1) the global memory access is almost completely coalesced, (2) the latency could be hidden

by subsequent computation, and (3) this overhead is only a small factor of large computation (short rows normally do not face this problem). For very long rows that exceed the scratchpad memory capacity, our method still allocates a space in the scratchpad memory as a level-1 merge sequence, executes the same merge operations on it and merges the level-1 sequence in the scratchpad memory and the resulting sequence in the global memory only once before the kernel is ready to return.

It is worth noting that the parameters of the binning depends on specifications (e.g., thread bunch size and scratchpad memory capacity) of GPU architectures. In this chapter, we use the abovementioned fixed-size parameters for assigning the rows into the bins since the current nVidia GPUs and AMD GPUs have comparable hardware specifications. However, the strategies in stages 2 and 3 can be easily extended for future GPUs with changed architecture designs.

The fourth stage, arranging data, first sums the numbers of nonzero entries in all rows of the resulting matrix C and allocates its final memory space. Then our method copies existing nonzero entries from the temporary matrix $\widetilde{C}$ to the resulting matrix C. For the rows in the bin group 1, the copy operation is not required. For the rows in the bin group 2, we use one thread for each row. For the rest of the rows in the bin groups 3–5, we use one thread group for each row. After all copy operations, the SpGEMM computation is done.

9.5 Experimental Results

We use four platforms (one CPU and three GPUs) shown in Table 9.1 for evaluating the SpGEMM algorithms. Tables B.1 and B.2 list specifications of the used hardware. The host side of all GPUs is a quad-core 3.7GHz CPU in an AMD A10-7850K APU with 8 GB DDR3-1600 dual-channel system memory and 64-bit Ubuntu Linux 14.04.

The testbeds	The participating formats and algorithms
Intel Xeon E5-2630	(1) Intel MKL v11.0.
nVidia GeForce GTX Titan Black	(1) CUSP v0.4.0 [51]. (2) cuSPARSE v6.5 [140]. (3) RMerge [83]. (4) bhSPARSE (this work).
nVidia GeForce GTX 980	(1) CUSP v0.4.0 [51]. (2) cuSPARSE v6.5 [140]. (3) RMerge [83]. (4) bhSPARSE (this work).
AMD Radeon R9 290X	(1) bhSPARSE (this work).

Table 9.1: The testbeds and participating algorithms.

9.5.1 Galerkin Products

Testing Scenario

Calculating Galerkin products plays an important role in AMG. We use smoothed aggregation preconditioner with Jacobi smoother (described in [20] and implemented in the CUSP library [51]) as a test scenario for evaluating SpGEMM algorithms. In each level of an AMG hierarchy in this context, we multiply three sparse matrices P^T, A and P, where rectangular matrix P^T is a restriction operator, square matrix A is initially the system matrix, and rectangular matrix P is a prolongation operator.

Performance Comparison

Figures 9.2 and 9.3 show execution time of Galerkin products $P^T AP$ in constructing an AMG hierarchy (typically including 3-5 levels) for a smoothed aggregation preconditioner in single precision and double precision, respectively. The input system matrix A is from 2D 5-point, 2D 9-point, 3D 7-point or 3D 27-point Poisson problem, respectively. The two 2D problems have dimensions 1024×1024 and generate system matrices of size 1048576×1048576. The two 3D problems have dimensions $101 \times 101 \times 101$ and generate system matrices of size 1030301×1030301. The SpGEMM approaches in three libraries, CUSP v0.4.0, cuSPARSE v6.5 and bhSPARSE[1], are tested on nVidia GeForce GTX Titan Black and GeForce GTX 980 GPUs. To obtain the best SpGEMM performance, CUSP uses the coordinate (COO) format for its input matrices. The other two libraries use the CSR format. Because the operation multiplies three sparse matrices P^T, A and P, the order of multiplication may influence overall performance. Here we test the two possible orders $(P^T A)P$ and $P^T(AP)$. In our experiments, matrix data transfer time between the host and the device is not included since the SpGEMM is normally one of the building blocks for more complex problem completely running on GPUs.

In Figures 9.2 and 9.3, we can see that our method is constantly faster than SpGEMM algorithms in the other two libraries. When using system matrix from 3D 27-point Poisson problem, bhSPARSE delivers up to 2.6x and up to 2.7x speedups over cuSPARSE and CUSP, respectively. On average, speedups of 1.9x and 1.7x are achieved when compared with the above two libraries, respectively.

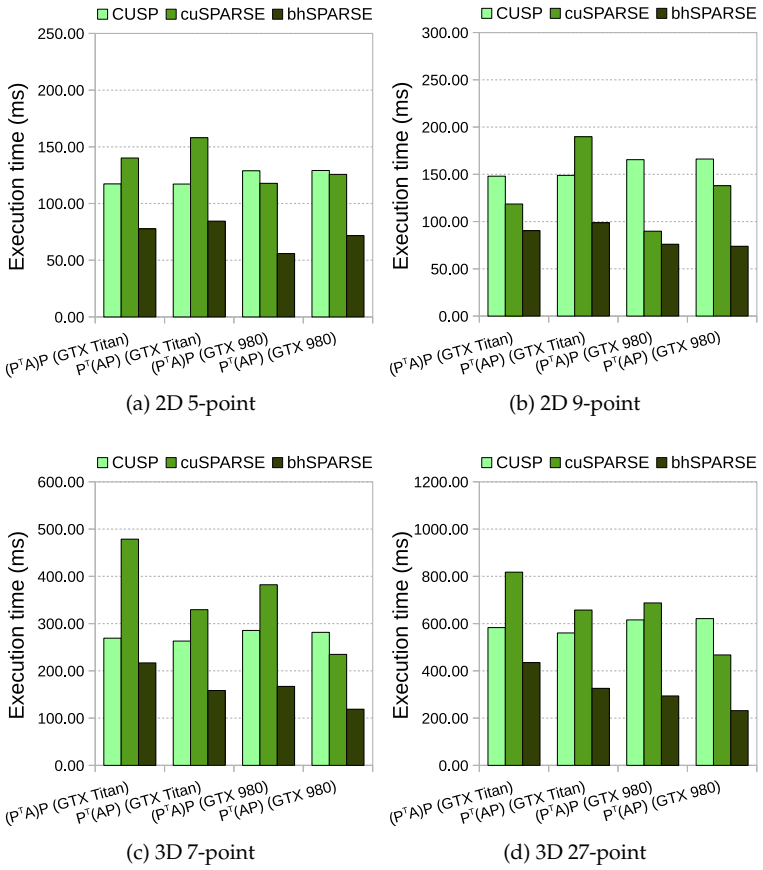
As for the order of multiplication, we can see that our method in general gives better performance while doing $P^T(AP)$, compared to running $(P^T A)P$. In contrast, the order of multiplication does not bring obvious performance difference for CUSP. When cuSPARSE is used, $(P^T A)P$ delivers better throughput for the two 2D problems, but degrades throughput for the two 3D problems.

9.5.2 Matrix Squaring

Benchmark Suite

We also evaluate multiplication of sparse square matrix and itself (i.e., $C = A^2$) to avoid introducing another sparse matrix as a multiplier with different sparsity

[1] We call our library bhSPARSE since this work is under the Project Bohrium [104].

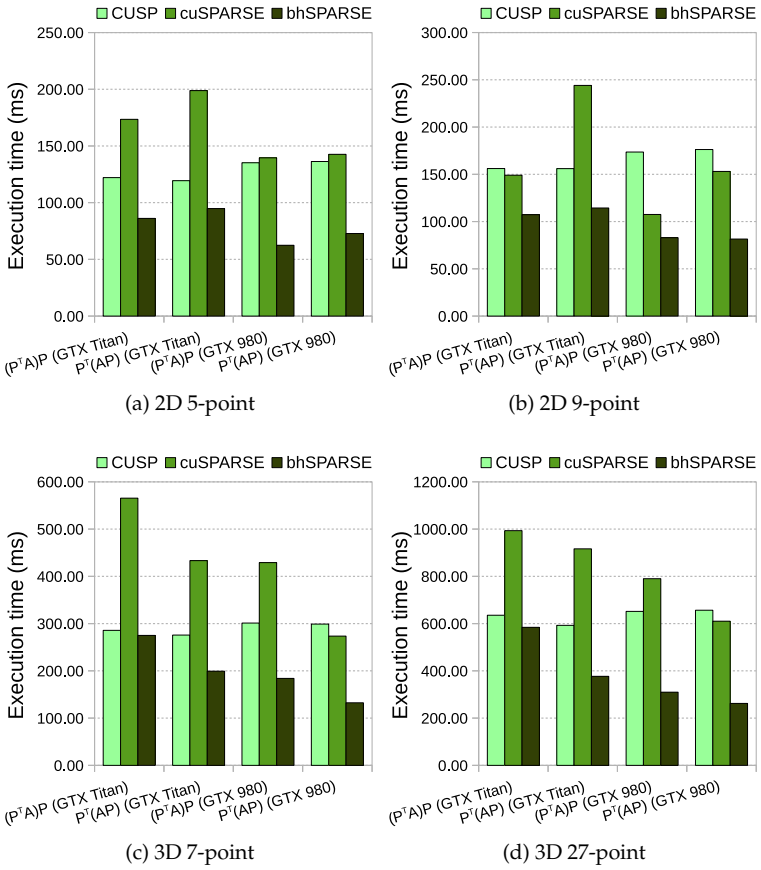


(a) 2D 5-point (b) 2D 9-point

(c) 3D 7-point (d) 3D 27-point

Figure 9.2: Execution time (in milliseconds) comparison of single precision SpGEMM (SpSGEMM) from three libraries CUSP, cuSPARSE and bhSPARSE in the context of smoothed aggregation preconditioner with Jacobi smoother. The system matrices are from four Poisson problems. Both $(P^T A)P$ and $P^T(AP)$ are tested on two nVidia GPUs.

structure. We choose 23 sparse matrices as our benchmark suite. 16 of them were widely used for performance evaluations in previous sparse matrix computation research [120, 121, 57, 52, 83, 92, 158, 184, 34]. The other 7 new matrices are chosen since they bring more diverse irregular sparsity structures that challenge the SpGEMM algorithm design. The variety of sparsity structures are from many application fields, such as finite element methods, macroeconomic model, protein data, circuit simulation, web connectivity and combinational problem. All of the 23 matrices are downloadable from the University of Florida Sparse Matrix Collection [56]. Note that symmetry in the sparse matrices is not used in our SpGEMM algorithm, although some matrices in the benchmark suite are symmetric. Also note that we use the standard CSR format that does not consider symmetric storage pattern.



(a) 2D 5-point

(b) 2D 9-point

(c) 3D 7-point

(d) 3D 27-point

Figure 9.3: Execution time (in milliseconds) comparison of double precision SpGEMM (SpDGEMM) from three libraries CUSP, cuSPARSE and bhSPARSE in the context of smoothed aggregation preconditioner with Jacobi smoother. The system matrices are from four Poisson problems. Both $(P^T A)P$ and $P^T(AP)$ are tested on two nVidia GPUs.

Besides the input matrix A, the work complexities of the different SpGEMM algorithms also depend on the intermediate matrix $\widehat{C}$ and the resulting matrix C. So we list characteristics of the three matrices in Table 9.2. The set of characteristics includes matrix dimension (n), the number of nonzero entries (nnz) and the average number of nonzero entries in rows ($nnzr$). The upper 9 matrices in the table have relatively regular nonzero entry distribution mostly on the diagonal. The other 14 matrices include various irregular sparsity structures.

Table 9.2: Overview of sparse matrices for benchmarking matrix squaring. Here $nnz(\widehat{C})$ is the upper bound size of A^2. Numerically, $nnz(\widehat{C})$ equals to half of $flops$, the number of necessary arithmetic operations while doing SpGEMM. $nnz(C)$ is the number of nonzero entries in the resulting matrix $C = A^2$.

Id	**Name**	n	$nnz(A)$, $nnzr(A)$	$nnz(\widehat{C})$, $nnzr(\widehat{C})$	$nnz(C)$, $nnzr(C)$
r1	FEM/Cantilever	63 K	4 M, 64	269.5 M, 4315	17.4 M, 279
r2	Economics	207 K	1.3 M, 6	7.6 M, 37	6.7 M, 32
r3	Epidemiology	526 K	2.1 M, 4	8.4 M, 16	5.2 M, 10
r4	Filter3D	106 K	2.7 M, 25	86 M, 808	20.2 M, 189
r5	Wind Tunnel	218 K	11.6 M, 53	626.1 M, 2873	32.8 M, 150
r6	FEM/Ship	141 K	7.8 M, 55	450.6 M, 3199	24.1 M, 171
r7	FEM/Harbor	47 K	2.4 M, 51	156.5 M, 3341	7.9 M, 169
r8	Protein	36 K	4.3 M, 119	555.3 M, 15249	19.6 M, 538
r9	FEM/Spheres	83 K	6 M, 72	463.8 M, 5566	26.5 M, 318
i1	2cubes_sphere	102 K	1.6 M, 16	27.5 M, 270	9 M, 88
i2	FEM/Accelerator	121 K	2.6 M, 22	79.9 M, 659	18.7 M, 154
i3	Cage12	130 K	2 M, 16	34.6 M, 266	15.2 M, 117
i4	Hood	221 K	10.8 M, 49	562 M, 2548	34.2 M, 155
i5	M133-b3	200 K	0.8 M, 4	3.2 M, 16	3.2 M, 16
i6	Majorbasis	160 K	1.8 M, 11	19.2 M, 120	8.2 M, 52
i7	Mario002	390 K	2.1 M, 5	12.8 M, 33	6.4 M, 17
i8	Mono_500Hz	169 K	5 M, 30	204 M, 1204	41.4 M, 244
i9	Offshore	260 K	4.2 M, 16	71.3 M, 275	23.4 M, 90
i10	Patents_main	241 K	0.6 M, 2	2.6 M, 11	2.3 M, 9
i11	Poisson3Da	14 K	0.4 M, 26	11.8 M, 871	3 M, 219
i12	QCD	49 K	1.9 M, 39	74.8 M, 1521	10.9 M, 222
i13	Circuit	171 K	1 M, 6	8.7 M, 51	5.2 M, 31
i14	Webbase	1 M	3.1 M, 3	69.5 M, 70	51.1 M, 51

Performance Comparison

The single precision and double precision absolute performance of the SpGEMM algorithms that compute $C = A^2$ are shown in Figures 9.4 and 9.5, respectively. Four GPU methods from CUSP v0.4.0, cuSPARSE v6.5, RMerge [83] and bhSPARSE are evaluated on three GPUs: nVidia GeForce GTX Titan Black, nVidia GeForce GTX 980 and AMD Radeon R9 290X. One CPU method in Intel MKL v11.0 is evaluated on Intel Xeon E5-2630 CPU. The performance of another recent ESC-based GPU SpGEMM work [52] is not included in the comparison because its source code is not available to us yet. The Intel MKL SpGEMM program is multithreaded and utilizes all six cores in the Intel Xeon CPU. For GPU algorithms, again, the host-device data transfer time is not included.

We first compare the performance of the four different GPU SpGEMM algorithms on the nVidia GPUs. We can see that bhSPARSE always outperforms CUSP, cuSPARSE and RMerge on most sparse matrices in the benchmark suite. Compared to the two

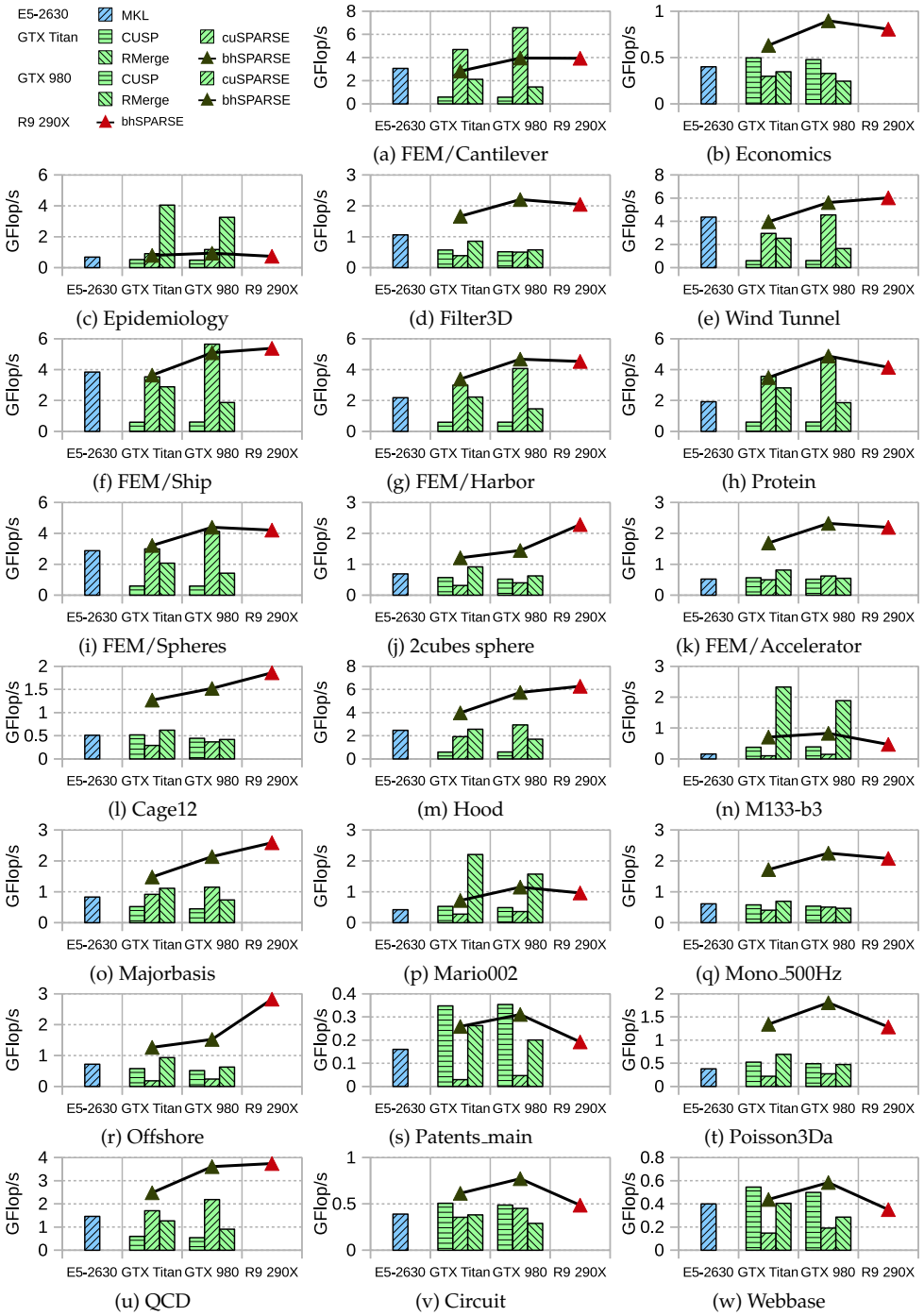


Figure 9.4: Single precision SpGEMM (SpSGEMM) GFlop/s comparison.

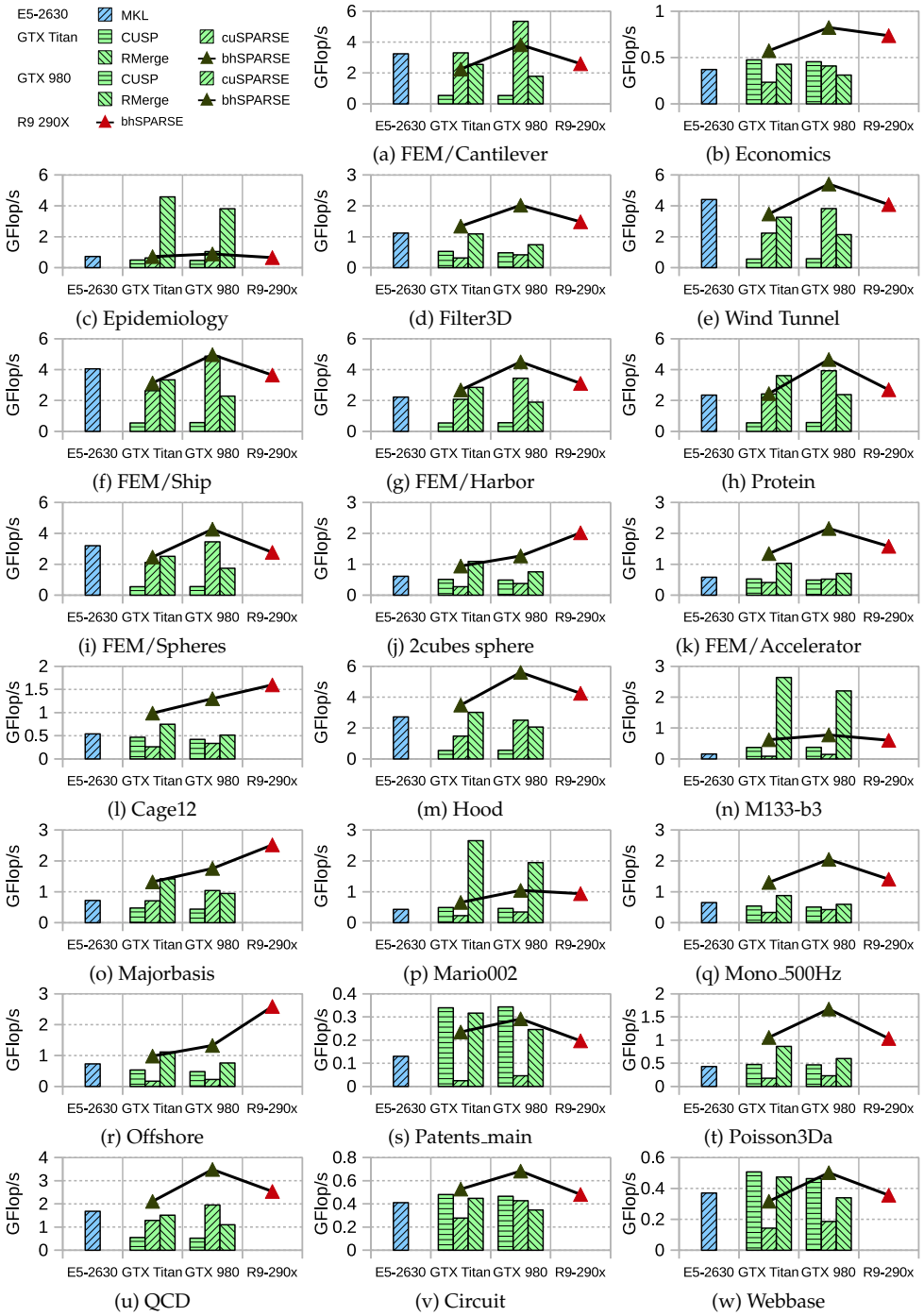


Figure 9.5: Double precision SpGEMM (SpDGEMM) GFlop/s comparison.

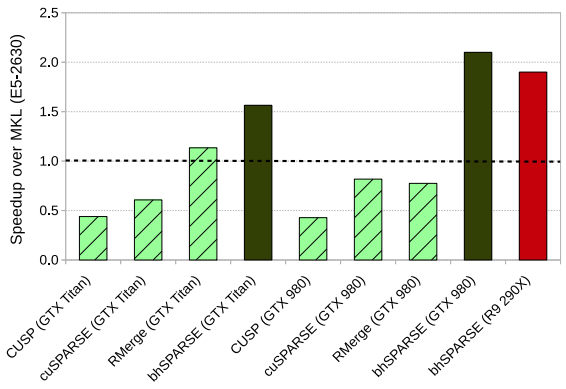
vendor supplied libraries, our method obtains better SpSGEMM and SpDGEMM performance on 21 and 21 matrices out of the whole 23 matrices over CUSP, and on 19 and 21 matrices over cuSPARSE, respectively. Compared to RMerge, another CUDA-specific method, bhSPARSE achieves better SpSGEMM and SpDGEMM performance on 19 and 10 matrices on the GTX Titan Black GPU, and on 19 and 20 matrices on the GTX 980 GPU.

From the perspective of speedup, our method delivers on average 4.6x (up to 9.6x) and 3.1x (up to 8.8x) speedup on SpSGEMM performance over CUSP and cuSPARSE, and on average 4.6x (up to 9.9x) and 3.1x (up to 9.5x) speedup on SpDGEMM performance over them, respectively. Compared to RMerge, our method offers on average 1.4x (up to 2.5x) speedup and 2.8x (up to 4.9x) speedup for SpSGEMM and on average 1.0x (up to 1.5x) and 2.1x (up to 3.4x) speedup for SpDGEMM on the GTX Titan Black GPU and GTX 980 GPU, respectively.

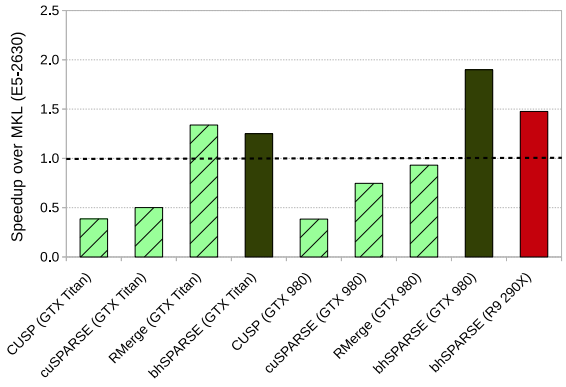
We can see that the cuSPARSE method outperforms our approach when and only when the input matrices are fairly regular (belong to the first 9 matrices in Table 9.2). For all irregular matrices and some regular ones, our bhSPARSE is always more efficient. On the other hand, the absolute performance of the CUSP method is very stable since its execution time almost only depends on the number of necessary arithmetic operations. Therefore this approach is insensitive to sparsity structures. Actually this insensitivity may bring better performance on matrices with some specific sparsity structures. However in most cases, the CUSP method suffers with higher global memory pressure. The RMerge method offers significant speedups over the other methods on three matrices (i.e., *Epidemiology*, *M133-b3* and *Mario002*), which are characterized by short rows. However, for the other matrices, RMerge supplies relatively lower performance due to imbalanced workload and high-overhead global memory operations between iterative steps. Further, we can see that since RMerge mainly relies on computational power of the SIMD units, its performance decreases from GTX Titan Black (2880 CUDA cores running at 889 MHz) to GTX 980 (2048 CUDA cores running at 1126 MHz). In contrast, our method also depends on capacity of scratchpad memory. Thus we can see that bhSPARSE obtains better performance while using GTX 980 (1536 kB scratchpad) over GTX Titan Black (720 kB scratchpad).

Compared to Intel MKL on the Intel CPU, our CUDA-based implementation on the nVidia GPUs obtains better SpSGEMM and SpDGEMM performance on all 23 matrices, and delivers on average 2.5x (up to 5.2x) and 2.2x (up to 4.9x) SpSGEMM and SpDGEMM speedup, respectively. Our OpenCL-based implementation on the AMD GPU in the machine 2 obtains better SpSGEMM and SpDGEMM performance on 23 and 18 matrices, and delivers on average 2.3x (up to 4.2x) and 1.9x (up to 3.8x) SpSGEMM and SpDGEMM speedup, respectively.

The relative performance (harmonic mean) of the SpGEMM algorithms that compute $C = A^2$ is shown in Figure 9.6. We can see that our method in general delivers the best performance on the used testbeds while running the 23 matrices as a benchmark suite. If we set the Intel MKL SpGEMM performance in this scenario as a baseline, our approach is the only GPU SpGEMM that constantly outperforms well optimized CPU method.



(a) Single precision SpGEMM



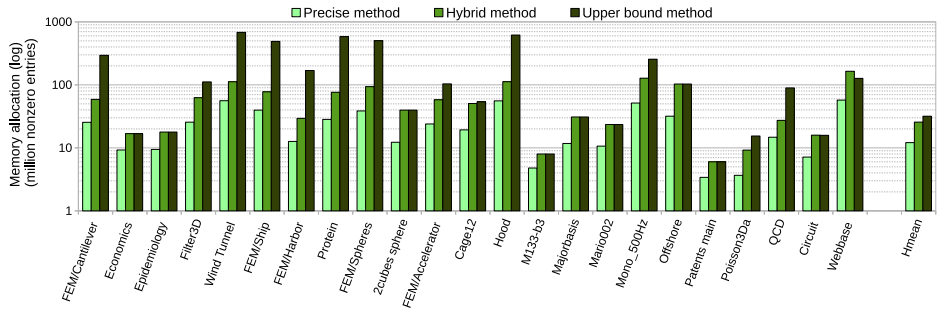
(b) Double precision SpGEMM

Figure 9.6: Average (harmonic mean) relative performance comparison of the 23 matrices, using SpGEMM method in MKL on Intel Xeon E5-2630 as a baseline.

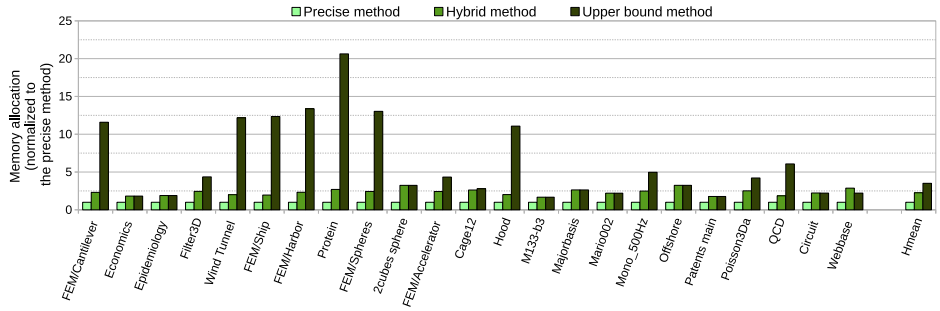
Memory Pre-allocation Comparison

Figure 9.7 shows the comparison of the three memory pre-allocation methods, while benchmarking $C = A^2$. We can see that, for small matrices (e.g., *2cubes_sphere*), our hybrid method shows exactly the same space requirements as the upper bound method does. However, for large matrices, allocated memory sizes through our hybrid method are much closer to the memory sizes allocated by the precise method. Taking the matrix *Protein* as an example, our hybrid method requires 2.7x memory space over the precise method, while the upper bound method needs 20.6x space requirement. One exception is the matrix *Webbase*, our hybrid method actually allocates more memory space than the upper bound method. The reasons are that the reduced rate of the intermediate matrix $\widehat{C}$ to the resulting matrix C is very low (see Table 9.2) and our 2x progression mechanism just allocates memory across the upper bound size. But overall, our hybrid method saves space allocation of the upper bound method and execution time of the precise method without introducing any significant extra

space requirements.



(a) Absolute memory requirement



(b) Relative memory requirement

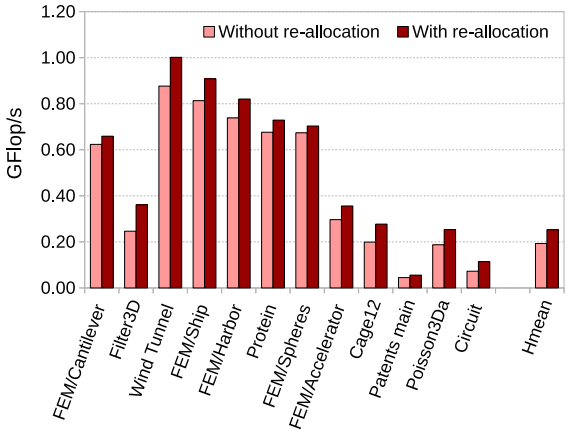
Figure 9.7: Global memory requirement comparison of the precise method, our hybrid method and the upper bound method, when benchmarking $C = A^2$ on the 23 matrices. The memory requirement of the precise method includes the two input matrices and the resulting matrix. The memory requirements of the other two methods also contain additional intermediate matrices. "Hmean" refers to harmonic mean.

9.5.3 Using Re-Allocatable Memory

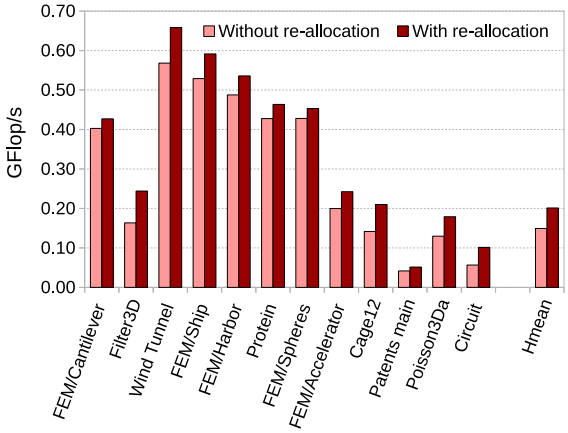
For some matrices with relatively long rows in the bin group 5, our method dumps scratchpad data to global memory, allocates a larger memory block, copies the old data to the newly allocated portion, reloads values and continues processing. We have to do the allocation/copy operation pair and pay the overhead since current GPUs are not able to re-allocate memory (i.e., change the size of the memory block pointed to a certain pointer). However, the emerging heterogeneous processors with shared virtual memory (or unified memory) address space deliver a possibility that lets integrated GPUs use system memory, which is re-allocatable from the CPU side.

We evaluated two memory allocation strategies (i.e., a typical allocation/copy approach and an improved re-allocation approach) of our OpenCL-based SpGEMM algorithm on the GPU part in the AMD A10-7850K APU (see Table B.5 in Appendix B for specifications). Figure 9.8 shows the results. We can see that re-allocatable memory brings on average 1.2x (up to 1.6x) speedup and on average 1.2x (up to 1.8x) speedup

for SpSGEMM and SpDGEMM, respectively. Therefore, our GPU SpGEMM method may deliver further performance improvement on future GPUs with re-allocatable memory, or on emerging heterogeneous processors composed of CPU cores and GPU cores. Moreover, both CPU cores and GPU cores can be utilized for Stage 3 in our framework. We leave this heterogenous workload partitioning (similar to the methods described in [162, 164]) to future work.



(a) Single precision SpGEMM



(b) Double precision SpGEMM

Figure 9.8: $C = A^2$ performance of bhSPARSE running with and without re-allocatable memory on an AMD A10-7850K APU. Note that only executable matrices that require memory re-allocation are included here. "Hmean" refers to harmonic mean.

9.6 Comparison to Recently Developed Methods

A classic CPU SpGEMM algorithm, also known as Matlab algorithm, was proposed by Gilbert et al. [77]. This approach uses a dense vector-based sparse accumulator (or SPA) and takes $O(flops + nnz(B) + n)$ time to complete the SpGEMM, where $flops$ is defined as the number of necessary arithmetic operations on the nonzero entries, $nnz(B)$ is defined as the number of nonzero entries in the matrix B, and n is the number of rows/columns of the input square matrices. Matam et al. [130] developed a similar Matlab algorithm implementation for GPUs. Sulatycke and Ghose [171] proposed a cache hits-oriented algorithm runs in relatively longer time $O(flops + n^2)$. A fast serial SpGEMM algorithm with time complexity $O(nnz^{0.7}n^{1.2} + n^{2+o(1)})$ was developed by Yuster and Zwick [190]. Buluç and Gilbert [32] presented an SpGEMM algorithm with time complexity independent to the size of the input matrices under assumptions that the algorithm is used as a sub-routine of 2D distributed memory SpGEMM and the input matrices are hypersparse ($nnz < n$).

Recent GPU-based SpGEMM algorithms showed better time complexity. The SpGEMM algorithm in the cuSPARSE library [57, 140] utilized GPU hash table for the insert operations (lines 7–11 in Algorithm 33). So time complexity of this approach is $O(flops)$ on average and $O(flops\ \ nnzr(C))$ in the worst case, where $nnzr(C)$ is defined as the average number of nonzero entries in the rows of the matrix C. Because the algorithm allocates one hash table of fixed size for each row of C, the space complexity is $O(nnz(A) + nnz(B) + n + nnz(C))$.

The CUSP library [20, 51] developed an SpGEMM method called expansion, sorting and compression (ESC) that expands all candidate nonzero entries generated by the necessary arithmetic operations (line 6 in Algorithm 33) into an intermediate sparse matrix $\widehat{C}$, sorts the matrix by rows and columns and compresses it into the resulting matrix C by eliminating entries in duplicate positions. By using GPU radix sort algorithm (with linear time complexity while size of the index data type of the matrices is fixed) and prefix-sum scan algorithm (with linear time complexity) as building blocks, time complexity of the ESC algorithm is $O(flops+nnz(\widehat{C})+nnz(\widehat{C}))$. Since $nnz(\widehat{C})$ equals half of $flops$, the ESC algorithm takes the optimal $O(flops)$ time. Dalton et al. [52] improved the ESC algorithm by executing sorting and compression on the rows of $\widehat{C}$, but not on the entire matrix. Therefore fast on-chip memory has a chance to be utilized more efficiently. The improved method sorts the very short rows (of size no more than 32) by using sorting network algorithm (with time complexity $O(nnzr(\widehat{C})\log^2(nnzr(\widehat{C})))$) instead of the radix sort algorithm which is mainly efficient for long lists. So the newer method is more efficient in practice, even though its time complexity is not lower than the original ESC algorithm. Because both of the ESC algorithms allocate an intermediate matrix $\widehat{C}$, they have the same space complexity $O(nnz(A) + nnz(B) + nnz(\widehat{C}) + nnz(C))$.

RMerge algorithm, recently proposed by Gremse et al. [83], iteratively merges rows in the matrix B into the resulting matrix C. Because this approach underutilizes thread interaction and generates one intermediate sparse matrix for each iteration step, it works best for input matrices with evenly distributed short rows. For irregular input matrices, load imbalance and large memory allocation make this method inefficient.

Our experimental results show that the proposed SpGEMM method in general outperforms the pervious SpGEMM methods designed for CPUs and GPUs.

10. Conclusion and Future Work

10.1 Conclusion

This thesis studied some key routines of Sparse BLAS and some fundamental data structures and algorithms as their building blocks.

Chapter 4 proposed *ad*-heap, a new efficient heap data structure for the tightly coupled CPU-GPU heterogeneous processors. Empirical studies were conducted based on the theoretical analysis. The experimental results showed that the *ad*-heap can obtain up to 1.5x and 3.6x performance of the optimal scheduling method on two representative machines, respectively. To the best of our knowledge, the *ad*-heap is the first fundamental data structure that efficiently leveraged the two different types of cores in the emerging heterogeneous processors through fine-grained frequent interactions between the CPUs and the GPUs. Further, the performance numbers also showed that redesigning data structure and algorithm is necessary for exposing higher computational power of the heterogeneous processors.

Chapter 6 proposed the CSR5 format. The format conversion from the CSR to the CSR5 was very fast because of the format's insensitivity to sparsity structure of the input matrix.

Chapter 7 developed three methods for sparse vector addition. Those methods have been used for adding rows of different sizes in the SpGEMM operation described in Chapter 9.

Chapter 8 proposed an efficient method for SpMV on heterogeneous processors using the CSR storage format. On three mainstream platforms from Intel, AMD and nVidia, the method greatly outperforms row block method CSR-based SpMV algorithms running on GPUs. The performance gain mainly comes from the newly developed speculative segmented sum strategy that efficiently utilizes different types of cores in a heterogeneous processor.

Chapter 8 also proposed the CSR5-based cross-platform SpMV algorithm for CPUs, GPUs and Xeon Phi. The CSR5-based SpMV was implemented by a redesigned segmented sum algorithm with higher SIMD utilization compared to the classic methods. The experimental results showed that the CSR5 delivered high throughput both in the isolated SpMV tests and in the iteration-based scenarios.

Chapter 9 demonstrated an efficient SpGEMM framework and corresponding algorithms on GPUs and emerging CPU-GPU heterogeneous processors for solving the three challenging problems in the SpGEMM. In the two experimental scenarios using matrices with diverse sparsity structures as input, the SpGEMM algorithm

delivered excellent absolute and relative performance as well as space efficiency over the previous GPU SpGEMM methods. Moreover, on average, the approach obtained around twice the performance of the start-of-the-art CPU SpGEMM method. Further, the method obtained higher performance on emerging heterogeneous processors with re-allocatable memory.

10.2 Future Work

Besides SpMV and SpGEMM, some other operations of Sparse BLAS such as multiplication of dense matrix and sparse vector or sparse matrix, and multiplication of sparse matrix and sparse vector or dense matrix may be important for some scenarios e.g., machine learning applications. Efficiently implementing them on CPUs/GPUs with more cores and emerging stronger CPU-GPU heterogeneous processors is an interesting future work.

In the SpMV and SpGEMM work designed for heterogeneous processors, we assign different task to the CPU part and the GPU part in one heterogeneous processor. However, the heaviest workload currently only runs on GPU cores, while the CPU cores may be idle. Obviously, it is possible to schedule tasks in the first stage on both CPU cores and GPU cores simultaneously for potentially higher throughput. Using the ideas described in [111, 94, 162, 164] for utilizing both CPU cores and GPU cores in a heterogeneous environment may further improve performance.

Furthermore, designing adaptive algorithms and sparse data structures for Sparse BLAS in graph applications is another promising direction [97]. Whether graph algorithms require a new set of primitives or can directly use current Sparse BLAS is still an open question [31, 131], thus need studies both in theory and in practice.

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] M. S. Alnaes, A. Logg, K-A. Mardal, O. Skavhaug, and H. P. Langtangen. Unified Framework for Finite Element Assembly. *Int. J. Comput. Sci. Eng.*, 4(4):231–244, 2009.

[3] AMD. *White Paper: Compute Cores*, jan 2014.

[4] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better Size Estimation for Sparse Matrix Products. *Algorithmica*, pages 741–757, 2014.

[5] 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, 2015 IEEE International*, IPDPS '15, pages 1043–1052, 2015.

[6] M. Arora, S. Nath, S. Mazumdar, S.B. Baden, and D.M. Tullsen. Redefining the role of the cpu in the era of cpu-gpu integration. *Micro, IEEE*, 32(6):4–16, 2012.

[7] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 781–792, 2014.

[8] 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, pages 273–282, 2014.

[9] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, feb 2011.

[10] Brett W. Bader and Tamara G. Kolda. Algorithm 862: MATLAB Tensor Classes for Fast Algorithm Prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, 2006.

[11] Brett W. Bader and Tamara G. Kolda. Efficient MATLAB Computations with Sparse and Factored Tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[12] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, and Hong Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[13] Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication Optimal Parallel Multiplication of Sparse Random Matrices. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 222–231, 2013.

[14] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. Efficient and Scalable Computations with Sparse Tensors. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–6, 2012.

[15] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 225–234, 2008.

[16] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing Sparse Matrix-Vector Multiplication on GPUs. Technical Report RC24704, IBM, 2008.

[17] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, 1968.

[18] Sean Baxter. *Modern GPU*. nVidia, 2013.

[19] Bob Beaty. *DKit: C++ Library of Atomic and Lockless Data Structures*, 2012.

[20] N. Bell, S. Dalton, and L. Olson. Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012.

[21] 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, pages 18:1–18:11, 2009.

[22] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, jan 2013.

[23] P. Bjorstad, F. Manne, T. Sorevik, and M. Vajtersic. Efficient Matrix Multiplication on SIMD Computers. *SIAM Journal on Matrix Analysis and Applications*, 13(1):386–401, 1992.

[24] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002.

[25] G. E. Blelloch. Scans As Primitive Parallel Operations. *IEEE Trans. Comput.*, 38(11):1526–1538, 1989.

[26] G. E. Blelloch. Prefix Sums and Their Applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, nov 1990.

[27] G. E. Blelloch. *Vector Models for Data-parallel Computing*. MIT Press, Cambridge, MA, USA, 1990.

[28] G. E. Blelloch, Michael A Heroux, and Marco Zagha. Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Technical Report CMU-CS-93-173, Carnegie Mellon University, Aug 1993.

[29] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix Market: A Web Resource for Test Matrix Collections. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, pages 125–137, 1997.

[30] Alexander Branover, Denis Foley, and Maurice Steinman. Amd fusion apu: Llano. *IEEE Micro*, 32(2):28–37, 2012.

[31] A. Buluç and John R Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, 2011.

[32] A. Buluç and J.R. Gilbert. On the Representation and Multiplication of Hypersparse Matrices. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11, 2008.

[33] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 721–733, 2011.

[34] Aydin Buluç and John R. Gilbert. Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 503–510, 2008.

[35] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 233–244, 2009.

[36] Jong-Ho Byun, Richard Lin, James W. Demmel, and Katherine A. Yelick. *pOSKI: Parallel Optimized Sparse Kernel Interface Library User's Guide*. University of California, Berkeley, 1.0 edition, Apr 2012.

[37] Bradford L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[38] Bradford L. Chamberlain and Lawrence Snyder. Array Language Support for Parallel Sparse Computation. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 133–145, 2001.

[39] Timothy M. Chan. More Algorithms for All-pairs Shortest Paths in Weighted Graphs. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 590–598, 2007.

[40] Siddhartha Chatterjee, G.E. Blelloch, and Marco Zagha. Scan Primitives for Vector Computers. In *Supercomputing '90., Proceedings of*, pages 666–675, 1990.

[41] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, 2009.

[42] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A Performance Study of General-purpose Applications on Graphics Processors Using CUDA. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[43] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11, 2010.

[44] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 25:1–25:11, 2012.

[45] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-Driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, 2010.

[46] E.S. Chung, P.A. Milder, J.C. Hoe, and Ken Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 225–236, 2010.

[47] Edith Cohen. On Optimizing Multiplications of Sparse Matrices. In WilliamH. Cunningham, S.Thomas McCormick, and Maurice Queyranne, editors, *Integer Programming and Combinatorial Optimization*, volume 1084 of *Lecture Notes in Computer Science*, pages 219–233. Springer Berlin Heidelberg, 1996.

[48] M. Daga and M. Nutter. Exploiting coarse-grained parallelism in b+ tree searches on an apu. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 240–247, 2012.

[49] Mayank Daga, Ashwin M. Aji, and Wu-chun Feng. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, SAAHPC '11, pages 141–149, 2011.

[50] Mayank Daga and Joseph L. Greathouse. Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices. In *High Performance Computing (HiPC), 2015 22nd International Conference on*, HiPC '15, 2015.

[51] Steven Dalton and Nathan Bell. CUSP : A C++ Templated Sparse Matrix Library.

[52] Steven Dalton, Luke Olsen, and Nathan Bell. Optimizing Sparse Matrix-Matrix Multiplication for the GPU. *ACM Transactions on Mathematical Software*, 41(4), 2015.

[53] Satish Damaraju, Varghese George, Sanjeev Jahagirdar, Tanveer Khondker, R. Milstrey, Sanjib Sarkar, Scott Siers, I. Stolero, and Arun Subbiah. A 22nm ia multi-cpu and gpu system-on-chip. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 56–57, 2012.

[54] Hoang-Vu Dang and Bertil Schmidt. The sliced coo format for sparse matrix-vector multiplication on cuda-enabled gpus. *Procedia Computer Science*, 9(0):57 – 66, 2012.

[55] A. Davidson, D. Tarjan, M. Garland, and J.D. Owens. Efficient Parallel Merge Sort for Fixed and Variable Length Keys. In *Innovative Parallel Computing (InPar), 2012*, pages 1–9, 2012.

[56] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, dec 2011.

[57] Julien Demouth. Sparse Matrix-Matrix Multiplication on the GPU. Technical report, NVIDIA, 2012.

[58] Yangdong (Steve) Deng, Bo David Wang, and Shuai Mu. Taming Irregular EDA Applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 539–546, 2009.

[59] Mrinal Deo and Sean Keely. Parallel suffix array and least common prefix for the gpu. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 197–206, 2013.

[60] J. J. Dongarra, Jermey Du Cruz, Sven Hammerling, and I. S. Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Softw.*, 16(1):18–28, 1990.

[61] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[62] J. J. Dongarra and Michael A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, 2013.

[63] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast Scan Algorithms on Graphics Processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 205–213, 2008.

[64] Iain S. Duff. A Survey of Sparse Matrix Research. *Proceedings of the IEEE*, 65(4):500–535, 1977.

[65] Iain S Duff, Albert M Erisman, and John K Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., 1986.

[66] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Sparse Matrix Test Problems. *ACM Trans. Math. Softw.*, 15(1):1–14, 1989.

[67] Iain S. Duff, Roger G. Grimes, and John G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report RAL-TR-97-031, Rutherford Appleton Laboratory, UK, 1997.

[68] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *ACM Trans. Math. Softw.*, 28(2):239–267, 2002.

[69] Iain S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Anal. Appl.*, 22(4):973–996, 2000.

[70] Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: A user-level interface. *ACM Trans. Math. Softw.*, 23(3):379–401, 1997.

[71] Iain S. Duff and Christof Vömel. Algorithm 818: A Reference Model Implementation of the Sparse BLAS in Fortran 95. *ACM Trans. Math. Softw.*, 28(2):268–283, 2002.

[72] SC Eisenstat, MC Gursky, MH Schultz, and AH Shermané. Yale Sparse Matrix Package II. The Nonsymetric Codes. Technical report, Yale University, 1982.

[73] Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. Test-driving Intel Xeon Phi. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 137–148, 2014.

[74] Jianbin Fang, A.L. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225, 2011.

[75] Timothy Furtak, José Nelson Amaral, and Robert Niewiadomski. Using simd registers and instructions to enable instruction-level parallelism in sorting algorithms. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 348–357, 2007.

[76] Michael Garland. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 2–6, 2008.

[77] J. Gilbert, C. Moler, and R. Schreiber. Sparse Matrices in MATLAB: Design and Implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[78] John R. Gilbert, William W. Pugh, and Tatiana Shpeisman. Ordered Sparse Accumulator and its Use in Efficient Sparse Matrix Computation. United States Patent US 5983230 A, nov 1999.

[79] JohnR. Gilbert, Steve Reinhardt, and ViralB. Shah. High-Performance Graph Algorithms from Parallel Sparse Matrices. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 260–269. Springer Berlin Heidelberg, 2007.

[80] Joseph L. Greathouse and Mayank Daga. Efficient Sparse Matrix-Vector Multiplication on GPUs using the CSR Storage Format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 769–780, 2014.

[81] Oded Green, Robert McColl, and David A. Bader. GPU Merge Path: A GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 331–340, 2012.

[82] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144, 2011.

[83] Felix Gremse, Andreas Höfter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. *SIAM Journal on Scientific Computing*, 37(1):C54–C71, 2015.

[84] Dahai Guo and William Gropp. Adaptive Thread Distributions for SpMV on a GPU. In *Proceedings of the Extreme Scaling Workshop*, pages 2:1–2:5, 2012.

[85] Fred G. Gustavson. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, sep 1978.

[86] Mark Harris, John D. Owens, and Shubho Sengupta. *CUDPP Documentation*. nVidia, 2.2 edition, Aug 2014.

[87] Jiong He, Mian Lu, and Bingsheng He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, aug 2013.

[88] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[89] Kaixi Hou, Hao Wang, and Wu-chun Feng. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, pages 383–392, 2015.

[90] HSA Foundation. *HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG)*, 0.95 edition, May 2013.

[91] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 189–198, 2007.

[92] Intel. Intel Math Kernel Library.

[93] Donald B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3):53 – 57, 1975.

[94] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 151–162, 2014.

[95] Haim Kaplan, Micha Sharir, and Elad Verbin. Colored Intersection Searching via Sparse Rectangular Matrix Multiplication. In *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, SCG '06, pages 52–60, 2006.

[96] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *Micro, IEEE*, 31(5):7–17, 2011.

[97] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[98] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proc. VLDB Endow.*, 2(2):1378–1389, aug 2009.

[99] Peter Kipfer and Rüdiger Westermann. Improved GPU Sorting. In Matt Pharr, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 46, pages 733–746. Addison-Wesley, mar 2005.

[100] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors (Second Edition)*. Morgan Kaufmann, second edition edition, 2013.

[101] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 87–96, 2008.

[102] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Exploiting Compression Opportunities to Improve SpMxV Performance on Shared Memory Systems. *ACM Trans. Archit. Code Optim.*, 7(3):16:1–16:31, 2010.

[103] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiply on modern processors with wide simd units. *SIAM Journal on Scientific Computing*, 2014.

[104] Mads R. B. Kristensen, Simon A. F. Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: A Virtual Machine Approach to Portable Parallelism. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 312–321, 2014.

[105] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38(11):32–38, nov 2005.

[106] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 64–, 2004.

[107] Pramod Kumbhar. Performance of PETSc GPU Implementation with Sparse Matrix Storage Schemes. Master's thesis, The University of Edinburgh, Aug 2011.

[108] George Kyriazis. Heterogeneous system architecture: A technical review. Technical report, AMD, aug 2013.

[109] Anthony LaMarca and Richard Ladner. The influence of caches on the performance of heaps. *J. Exp. Algorithmics*, 1, jan 1996.

[110] D. Langr and P. Tvrdík. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Transactions on Parallel and Distributed Systems*, [in press], 2015.

[111] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent CPU-GPU Collaboration for Data-parallel Kernels on Heterogeneous Systems. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, 2013.

[112] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 101–110, 2009.

[113] Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. An Input-adaptive and In-place Approach to Dense Tensor-times-matrix Multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 76:1–76:12, 2015.

[114] Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. An Optimized Large-scale Hybrid DGEMM Design for CPUs and ATI GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 377–386, 2012.

[115] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 117–126, 2013.

[116] Ruipeng Li and Yousef Saad. GPU-Accelerated Preconditioned Iterative Linear Solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[117] Weifeng Liu and Brian Vinter. Ad-heap: An efficient heap data structure for asymmetric multicore processors. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 54:54–54:63, 2014.

[118] Weifeng Liu and Brian Vinter. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 370–381, 2014.

[119] Weifeng Liu and Brian Vinter. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. *Journal of Parallel and Distributed Computing*, 85:47 – 61, 2015.

[120] Weifeng Liu and Brian Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, pages 339–350, 2015.

[121] Weifeng Liu and Brian Vinter. Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. *Parallel Computing*, pages –, 2015.

[122] 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*, ICS '13, pages 273–282, 2013.

[123] Yongchao 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*, 2015.

[124] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated solution of differential equations by the finite element method: The FEniCS book*, volume 84. Springer Science & Business Media, 2012.

[125] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, 2009.

[126] Dimitar Lukarski and Nico Trost. PARALUTION - User Manual. Technical Report 1.0.0, PARALUTION Labs UG (haftungsbeschränkt) Co. KG, Feb 2015.

[127] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 354–365, 2013.

[128] Fredrik Manne. *Load Balancing in Parallel Sparse Matrix Computations*. PhD thesis, Department of Informatics, University of Bergen, Norway, 1993.

[129] Michele Martone. Efficient Multithreaded Untransposed, Transposed or Symmetric Sparse Matrix-Vector Multiplication with the Recursive Sparse Blocks Format. *Parallel Computing*, 40(7):251 – 270, 2014.

[130] K. Matam, S.R.K.B. Indarapu, and K. Kothapalli. Sparse Matrix-Matrix Multiplication on Modern Architectures. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–10, 2012.

[131] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. Standards for graph algorithm primitives. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–2, 2013.

[132] Victor Minden, Barry Smith, and MatthewG. Knepley. Preliminary Implementation of PETSc Using GPUs. In David A. Yuen, Long Wang, Xuebin Chi, Lennart Johnsson, Wei Ge, and Yaolin Shi, editors, *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, pages 131–140. Springer Berlin Heidelberg, 2013.

[133] Perhaad Mistry, Yash Ukidave, Dana Schaa, and David Kaeli. Valar: A benchmark suite to study the dynamic behavior of heterogeneous systems. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 54–65, 2013.

[134] Sparsh Mittal and Jeffrey S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, 2015.

[135] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin Heidelberg, 2010.

[136] Aaftab Munshi. *The OpenCL Specification*. Khronos OpenCL Working Group, 2.0 edition, Mar 2014.

[137] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. Unified Memory in CUDA 6: A Brief Overview and Related Data Access/Transfer Issues. Technical Report TR-2014-09, University of Wisconsin–Madison, Jun 2014.

[138] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. Power efficiency evaluation of block ciphers on gpu-integrated multicore processor. In Yang Xiang, Ivan Stojmenovic, BernadyO. Apduhan, Guojun Wang, Koji Nakano, and Albert Zomaya, editors, *Algorithms and Architectures for Parallel Processing*, volume 7439 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin Heidelberg, 2012.

[139] Rajesh Nishtala, RichardW. Vuduc, JamesW. Demmel, and KatherineA. Yelick. When Cache Blocking of Sparse Matrix Vector Multiply Works and Why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, 2007.

[140] NVIDIA. NVIDIA cuSPARSE library.

[141] nVidia. *NVIDIA Tegra K1 A New Era in Mobile Computing*, 1.1 edition, Jan 2014.

[142] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge Path - Parallel Merging Made Simple. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1611–1618, 2012.

[143] Gloria Ortega, Francisco Vázquez, Inmaculada García, and Ester M. Garzón. FastSpMM: An Efficient Library for Sparse Matrix Matrix Product on GPUs. *The Computer Journal*, 57(7):968–979, 2014.

[144] Rasmus Pagh and Morten Stöckel. The Input/Output Complexity of Sparse Matrix Multiplication. In AndreasS. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014*, Lecture Notes in Computer Science, pages 750–761. Springer Berlin Heidelberg, 2014.

[145] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. A Novel Sorting Algorithm for Many-Core Architectures Based on Adaptive Bitonic Sort. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 227–237, 2012.

[146] Hagen Peters and Ole Schulz-Hildebrandt. Comparison-Based In-Place Sorting with CUDA. In Wen-Mei Hwu, editor, *GPU Computing Gems Jade Edition*, chapter 8, pages 89–96. Morgan Kaufmann, Oct 2011.

[147] Juan C. Pichel, Francisco F. Rivera, Marcos Fernández, and Aurelio Rodríguez. Optimization of Sparse Matrix-Vector Multiplication Using Reordering Techniques on GPUs. *Microprocessors and Microsystems*, 36(2):65 – 77, 2012.

[148] J.A. Pienaar, S. Chakradhar, and A. Raghunathan. Automatic generation of software pipelines for heterogeneous parallel systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12, 2012.

[149] A. Pinar and M.T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, 1999.

[150] Qualcomm. *Qualcomm Snapdragon 800 Product Brief*, Aug 2013.

[151] I. Reguly and M. Giles. Efficient Sparse Matrix-Vector Multiplication on Cache-Based GPUs. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, May 2012.

[152] Huamin Ren, Weifeng Liu, Søren Ingvor Olsen, Sergio Escalera, and Thomas B. Moeslund. Unsupervised Behavior-Specific Dictionary Learning for Abnormal Event Detection. In Mark W. Jones Xianghua Xie and Gary K. L. Tam, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 28.1–28.13. BMVA Press, 2015.

[153] Huamin Ren and T.B. Moeslund. Abnormal Event Detection Using Local Sparse Representation. In *Advanced Video and Signal Based Surveillance (AVSS), 2014 11th IEEE International Conference on*, pages 125–130, 2014.

[154] Karl Rupp, Florian Rudolf, and Josef Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *GPUScA*, pages 51–56, 2010.

[155] Yousef Saad. Sparskit : A basic tool kit for sparse matrix computations. Technical Report RIACS-90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[156] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[157] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009.

[158] Erik Saule, Kamer Kaya, and ÜmitV. Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *Parallel Processing and Applied Mathematics*, Lecture Notes in Computer Science, pages 559–570. Springer Berlin Heidelberg, 2014.

[159] M.J. Schulte, M. Ignatowski, G.H. Loh, B.M. Beckmann, W.C. Brantley, S. Gurumurthi, N. Jayasena, I. Paul, S.K. Reinhardt, and G. Rodgers. Achieving Exascale Capabilities through Heterogeneous Computing. *Micro, IEEE*, 35(4):26–36, 2015.

[160] Naser Sedaghati, Te Mu, Louis-Noël Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, 2015.

[161] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, 2007.

[162] Jie Shen, Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. An Application-Centric Evaluation of OpenCL on Multi-Core CPUs. *Parallel Computing*, 39(12):834–850, 2013.

[163] Jie Shen, Ana Lucia Varbanescu, Henk Sips, Michael Arntzen, and Dick G. Simons. Glinda: A framework for accelerating imbalanced applications on heterogeneous platforms. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 14:1–14:10, 2013.

[164] Jie Shen, Ana Lucia Varbanescu, Peng Zou, Yutong Lu, and Henk Sips. Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 241–250, 2014.

[165] A. Sidelnik, S. Maleki, B.L. Chamberlain, M.J. Garzaran, and D. Padua. Performance Portability with the Chapel Language. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 582–594, 2012.

[166] I. Simecek, D. Langr, and P. Tvrdik. Space-efficient Sparse Matrix Storage Formats for Massively Parallel Systems. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 54–60, 2012.

[167] Kenneth Skovhede, Morten N. Larsen, and Brian Vinter. Extending Distributed Shared Memory for the Cell Broadband Engine to a Channel Model. In *Applied Parallel and Scientific Computing - 10th International Conference, PARA 2010*, pages 108–118, 2010.

[168] Kenneth Skovhede, Morten N. Larsen, and Brian Vinter. Programming the CELL-BE using CSP. In *33th Communicating Process Architectures Conference, CPA 2011,*, pages 55–70, 2011.

[169] Kyle L. Spafford, Jeremy S. Meredith, Seyong Lee, Dong Li, Philip C. Roth, and Jeffrey S. Vetter. The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 103–112, 2012.

[170] Bor-Yiing Su and Kurt Keutzer. clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 353–364, 2012.

[171] P.D. Sulatycke and K. Ghose. Caching-Efficient Multithreaded Fast Multiplication of Sparse Matrices. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing 1998*, pages 117–123, 1998.

[172] 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*, SC '13, pages 26:1–26:12, 2013.

[173] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huynh, Xibai Li, and Rick Siow Mong Goh. Optimizing and Auto-Tuning Scale-Free Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 136–145, 2015.

[174] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 213–224, 2012.

[175] Virginia Vassilevska, Ryan Williams, and Raphael Yuster. Finding Heaviest H-subgraphs in Real Weighted Graphs, with Applications. *ACM Trans. Algorithms*, 6(3):44:1–44:23, jul 2010.

[176] F. Vazquez, G. Ortega, J.J. Fernandez, I. Garcia, and E.M. Garzon. Fast Sparse Matrix Matrix Product Based on ELLR-T and GPU Computing. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 669–674, 2012.

[177] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. *Journal of Physics: Conference Series*, 16(1):521–530, 2005.

[178] Richard Wilson Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, Dec 2003.

[179] RichardW. Vuduc and Hyun-Jin Moon. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In *High Performance Computing and Communications*, volume 3726 of *Lecture Notes in Computer Science*, pages 807–816. Springer Berlin Heidelberg, 2005.

[180] Hao Wang, S. Potluri, D. Bureddy, C. Rosales, and D.K. Panda. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *Parallel and Distributed Systems, IEEE Transactions on*, 25(10):2595–2605, 2014.

[181] Jin Wang, Norman Rubin, Haicheng Wu, and Sudhakar Yalamanchili. Accelerating simulation of agent-based models on heterogeneous architectures.

In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 108–119, 2013.

[182] Mark Allen Weiss. *Data Structures and Algorithm Analysis*. Addison-Wesley, second edition, 1995.

[183] Jeremiah Willcock and Andrew Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 307–316, 2006.

[184] 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*, 35(3):178–194, mar 2009.

[185] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The Potential of the Cell Processor for Scientific Computing. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 9–20, 2006.

[186] Dong Hyuk Woo, Joshua B. Fryman, Allan D. Knies, and Hsien-Hsin S. Lee. Chameleon: Virtualizing idle acceleration cores of a heterogeneous multicore processor for caching and prefetching. *ACM Trans. Archit. Code Optim.*, 7(1):3:1–3:35, may 2010.

[187] Dong Hyuk Woo and Hsien-Hsin S. Lee. Compass: A programmable data prefetcher using idle gpu shaders. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 297–310, 2010.

[188] 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, pages 107–118, 2014.

[189] Yi Yang, Ping Xiang, Mike Mantor, and Huiyang Zhou. Cpu-assisted gpgpu on fused cpu-gpu architectures. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, pages 1–12, 2012.

[190] Raphael Yuster and Uri Zwick. Fast Sparse Matrix Multiplication. *ACM Trans. Algorithms*, 1(1):2–13, jul 2005.

[191] A. N. Yzelman. *Fast Sparse Matrix-Vector Multiplication by Partitioning and Reordering*. PhD thesis, Utrecht University, Utrecht, the Netherlands, 2011.

[192] A. N. Yzelman and Rob H. Bisseling. Two-Dimensional Cache-Oblivious Sparse Matrix-Vector Multiplication. *Parallel Computing*, 37(12):806–819, 2011.

[193] A. N. Yzelman and D. Roose. High-Level Strategies for Parallel Shared-Memory Sparse Matrix-Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):116–125, 2014.

[194] A. N. Yzelman, D. Roose, and K. Meerbergen. Sparse matrix-vector multiplication: parallelization and vectorization. In J. Reinders and J. Jeffers, editors, *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, chapter 27, page 20. Elsevier, 2014.

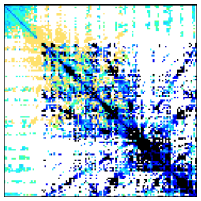
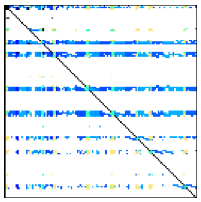
[195] Nan Zhang. A Novel Parallel Scan for Multicore Processors and Its Application in Sparse Matrix-Vector Multiplication. *Parallel and Distributed Systems, IEEE Transactions on*, 23(3):397–404, Mar 2012.

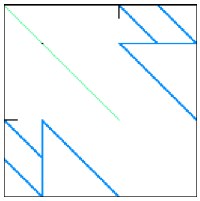
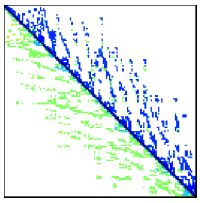
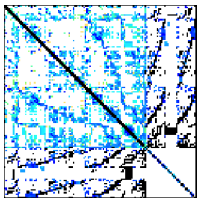
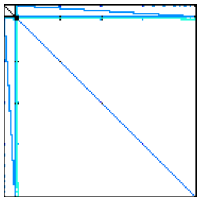
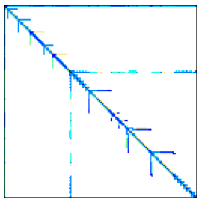
[196] Xianyi Zhang, Chao Yang, Fangfang Liu, Yiqun Liu, and Yutong Lu. Optimizing and Scaling HPCG on Tianhe-2: Early Experience. In *Algorithms and Architectures for Parallel Processing*, volume 8630 of *Lecture Notes in Computer Science*, pages 28–41. Springer International Publishing, 2014.

A. Benchmark Suite

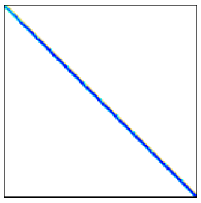
To maintain a relatively fair performance comparison, Duff et al. [66] advocated building publicly accessible sparse matrix collections. In the past several decades, a few collections (e.g., the SPARSKIT Collection [155], the Rutherford-Boeing Sparse Matrix Collection [67], the Matrix Market Collection [29]and the University of Florida Sparse Matrix Collection [56]) have been established and widely used in evaluating sparse matrix algorithms. The latest University of Florida Sparse Matrix Collection [56] is a superset of the former collections and includes over 2700 sparse matrices now. Therefore, all matrices (except a dense matrix named *Dense*) used in this thesis are downloaded from this collection. Details of matrices used in this thesis are listed in the alphabetical order in Table A.1.

Table A.1: Overview of evaluated sparse matrices

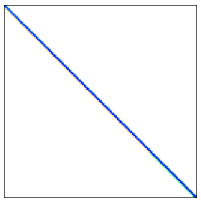
Plot	Information
	**Name**: 2cubes_sphere **#rows**: 101,492 **#columns**: 101,492 **#nonzeros**: 1,647,264 **#nonzeros per row (min; avg; max)**: 5; 16; 31 **Author**: E. Um **Kind**: Electromagnetics problem **Description**: FEM, electromagnetics, 2 cubes in a sphere.
	**Name**: ASIC_680k **#rows**: 682,862 **#columns**: 682,862 **#nonzeros**: 3,871,773 **#nonzeros per row (min; avg; max)**: 1; 6; 395,259 **Author**: R. Hoekstra **Kind**: Circuit simulation problem **Description**: Sandia, Xyce circuit simulation matrix (stripped).

	**Name**: Boyd2 **#rows**: 466,316 **#columns**: 466,316 **#nonzeros**: 1,500,397 **#nonzeros per row (min; avg; max)**: 2; 3; 93,262 **Author**: N. Gould **Kind**: Optimization problem **Description**: KKT matrix - convex QP (CUTEr).
	**Name**: Cage12 **#rows**: 130,228 **#columns**: 130,228 **#nonzeros**: 2,032,536 **#nonzeros per row (min; avg; max)**: 5; 15; 33 **Author**: A. van Heukelum **Kind**: Directed weighted graph **Description**: DNA electrophoresis, 12 monomers in polymer.
	**Name**: Circuit **#rows**: 170,998 **#columns**: 170,998 **#nonzeros**: 958,936 **#nonzeros per row (min; avg; max)**: 1; 5; 353 **Author**: S. Hamm **Kind**: Circuit simulation problem **Description**: Motorola circuit simulation.
	**Name**: Circuit5M **#rows**: 5,558,326 **#columns**: 5,558,326 **#nonzeros**: 59,524,291 **#nonzeros per row (min; avg; max)**: 1; 10; 1,290,501 **Author**: K. Gullapalli **Kind**: Circuit simulation problem **Description**: Large circuit from Freescale Semiconductor.
	**Name**: Dc2 **#rows**: 116,835 **#columns**: 116,835 **#nonzeros**: 766,396 **#nonzeros per row (min; avg; max)**: 1; 6; 114,190 **Author**: T. Lehner **Kind**: Subsequent circuit simulation problem **Description**: IBM EDA circuit simulation matrix.

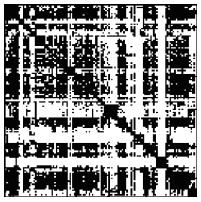
Name: Dense
#rows: 2,000
#columns: 2,000
#nonzeros: 4,000,000
#nonzeros per row (min; avg; max): 2,000; 2,000; 2,000
Author: Unknown
Kind: Dense
Description: Dense matrix in sparse format.



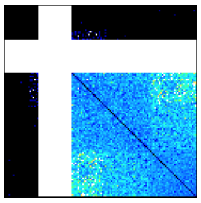
Name: Economics
#rows: 206,500
#columns: 206,500
#nonzeros: 1,273,389
#nonzeros per row (min; avg; max): 1; 6; 44
Author: Unknown
Kind: Economic problem
Description: Macroeconomic model.



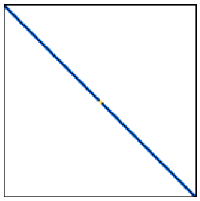
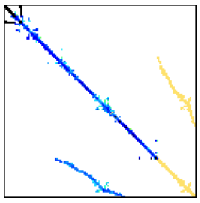
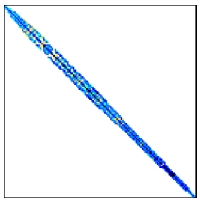
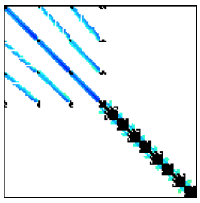
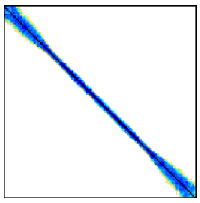
Name: Epidemiology
#rows: 525,825
#columns: 525,825
#nonzeros: 2,100,225
#nonzeros per row (min; avg; max): 2; 3; 4
Author: Unknown
Kind: 2D/3D problem
Description: 2D Markov model of epidemic.

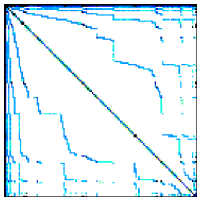


Name: Eu-2005
#rows: 862,664
#columns: 862,664
#nonzeros: 19,235,140
#nonzeros per row (min; avg; max): 0; 22; 6,985
Author: Universita degli Studi di Milano
Kind: Directed graph
Description: Small web crawl of .eu domain.

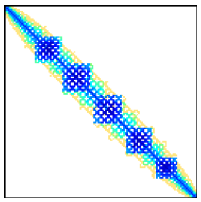


Name: FEM/Accelerator
#rows: 121,192
#columns: 121,192
#nonzeros: 2,624,331
#nonzeros per row (min; avg; max): 0; 21; 81
Author: Unknown
Kind: 2D/3D problem
Description: FEM/Accelerator: Accelerator cavity design.

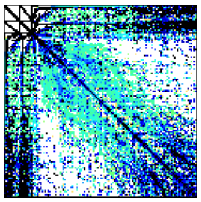
	**Name**: FEM/Cantilever **#rows**: 62,451 **#columns**: 62,451 **#nonzeros**: 4,007,383 **#nonzeros per row (min; avg; max)**: 1; 64; 78 **Author**: Unknown **Kind**: 2D/3D problem **Description**: FEM/Cantilever.
	**Name**: FEM/Harbor **#rows**: 46,835 **#columns**: 46,835 **#nonzeros**: 2,329,092 **#nonzeros per row (min; avg; max)**: 4; 50; 145 **Author**: S. Bova **Kind**: Computational fluid dynamics problem **Description**: 3D CFD model, Charleston harbor.
	**Name**: FEM/Ship **#rows**: 140,874 **#columns**: 140,874 **#nonzeros**: 7,813,404 **#nonzeros per row (min; avg; max)**: 24; 55; 102 **Author**: C. Damhaug **Kind**: Structural problem **Description**: DNV-Ex 4 : Ship section/detail from production run-1999-01-17.
	**Name**: FEM/Spheres **#rows**: 83,334 **#columns**: 83,334 **#nonzeros**: 6,010,480 **#nonzeros per row (min; avg; max)**: 1; 72; 81 **Author**: Unknown **Kind**: 2D/3D problem **Description**: FEM/Spheres: FEM concentric spheres.
	**Name**: Filter3D **#rows**: 106,437 **#columns**: 106,437 **#nonzeros**: 2,707,179 **#nonzeros per row (min; avg; max)**: 8; 25; 112 **Author**: D. Hohlfield, T. Bechtold, H. Zappe **Kind**: Model reduction problem **Description**: Oberwolfach: tunable optical filter.



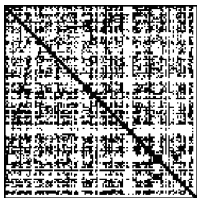
Name: FullChip
#rows: 2,987,012
#columns: 2,987,012
#nonzeros: 26,621,983
#nonzeros per row (min; avg; max): 1; 8; 2,312,481
Author: K. Gullapalli
Kind: Circuit simulation problem
Description: Circuit simulation from Freescale Semiconductor.



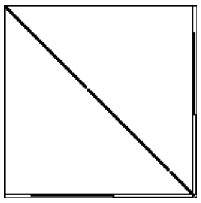
Name: Ga41As41H72
#rows: 268,096
#columns: 268,096
#nonzeros: 18,488,476
#nonzeros per row (min; avg; max): 18; 68; 702
Author: Y. Zhou, Y. Saad, M. Tiago, J. Chelikowsky
Kind: Theoretical/quantum chemistry problem
Description: Real-space pseudopotential method.



Name: Hood
#rows: 220,542
#columns: 220,542
#nonzeros: 10,768,436
#nonzeros per row (min; avg; max): 21; 48; 77
Author: J. Weiher
Kind: Structural problem
Description: INDEED Test Matrix (DC-mh).

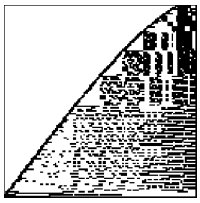


Name: In-2004
#rows: 1,382,908
#columns: 1,382,908
#nonzeros: 16,917,053
#nonzeros per row (min; avg; max): 0; 12; 7,753
Author: Universita degli Studi di Milano
Kind: Directed graph
Description: Small web crawl of .in domain.

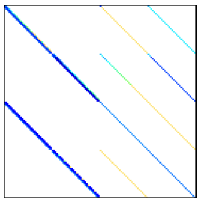


Name: Ins2
#rows: 309,412
#columns: 309,412
#nonzeros: 2,751,484
#nonzeros per row (min; avg; max): 5; 8; 309,412
Author: A. Andrianov
Kind: Optimization problem
Description: ins2 matrix from SAS Institute Inc.

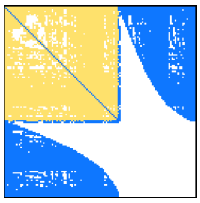
Name: LP
#rows: 4,284
#columns: 1,096,894
#nonzeros: 11,284,032
#nonzeros per row (min; avg; max): 1; 2,633; 56,181
Author: P. Nobili
Kind: Linear programming problem
Description: Italian railways (H. Mittelmann test set).



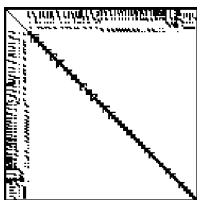
Name: M133-b3
#rows: 200,200
#columns: 200,200
#nonzeros: 800,800
#nonzeros per row (min; avg; max): 4; 4; 4
Author: V. Welker
Kind: Combinatorial problem
Description: Simplicial complexes from Homology.



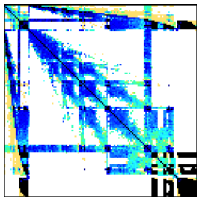
Name: Majorbasis
#rows: 160,000
#columns: 160,000
#nonzeros: 1,750,416
#nonzeros per row (min; avg; max): 6; 10; 11
Author: Q. Li and M. Ferris
Kind: Optimization problem
Description: MCP; mixed complementarity optimization problem; similar to QLi/crashbasis.



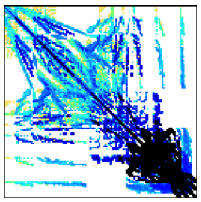
Name: Mario002
#rows: 389,874
#columns: 389,874
#nonzeros: 2,097,566
#nonzeros per row (min; avg; max): 2; 5; 7
Author: N. Gould, Y. Hu, J. Scott
Kind: Duplicate 2D/3D problem
Description: Larger matrix from Mario For which MA47 analysis is slow



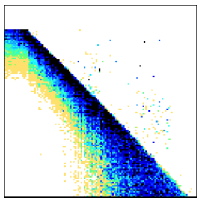
Name: Mip1
#rows: 66,463
#columns: 66,463
#nonzeros: 10,352,819
#nonzeros per row (min; avg; max): 4; 155; 66,395
Author: A. Andrianov
Kind: T. Davis
Description: mip1 matrix from SAS Institute Inc.



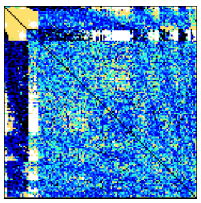
Name: Mono_500Hz
#rows: 169,410
#columns: 169,410
#nonzeros: 5,033,796
#nonzeros per row (min; avg; max): 10; 29; 719
Author: M. Gontier
Kind: Acoustics problem
Description: 3D vibro-acoustic problem, aircraft engine nacelle.



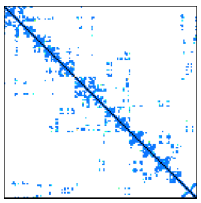
Name: Offshore
#rows: 259,789
#columns: 259,789
#nonzeros: 4,242,673
#nonzeros per row (min; avg; max): 5; 16; 31
Author: E. Um
Kind: Electromagnetics problem
Description: 3D FEM, transient electric field diffusion.



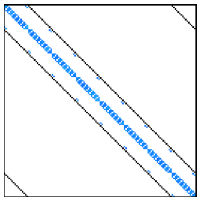
Name: Patents_main
#rows: 240,547
#columns: 240,547
#nonzeros: 560,943
#nonzeros per row (min; avg; max): 0; 2; 206
Author: B. Hall, A. Jaffe, M. Tratjenberg
Kind: Directed weighted graph
Description: Pajek network: main NBER US Patent Citations, 1963-1999, cites 1975-1999



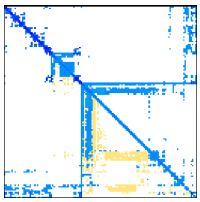
Name: Poisson3Da
#rows: 13,514
#columns: 13,514
#nonzeros: 352,762
#nonzeros per row (min; avg; max): 6; 26; 110
Author: COMSOL
Kind: Computational fluid dynamics problem
Description: Comsol, Inc. www.femlab.com : 3D Poisson problem



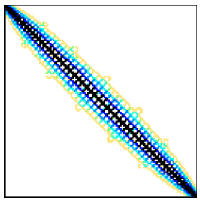
Name: Protein
#rows: 36,417
#columns: 36,417
#nonzeros: 4,344,765
#nonzeros per row (min; avg; max): 18; 119; 204
Author: S. G. Sarafianos et al
Kind: Weighted undirected graph
Description: Protein: protein data bank 1HYS.



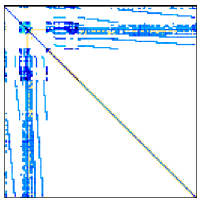
Name: QCD
#rows: 49,152
#columns: 49,152
#nonzeros: 1,916,928
#nonzeros per row (min; avg; max): 39; 39; 39
Author: B. Medeke
Kind: Theoretical/quantum chemistry problem
Description: Quantum chromodynamics conf5.4-00l8x8-2000



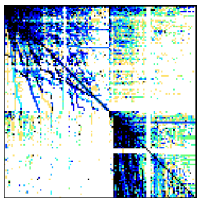
Name: Rajat21
#rows: 411,676
#columns: 411,676
#nonzeros: 1,876,011
#nonzeros per row (min; avg; max): 1; 4; 118,689
Author: Rajat
Kind: Circuit simulation problem
Description: Rajat/rajat21 circuit simulation matrix.



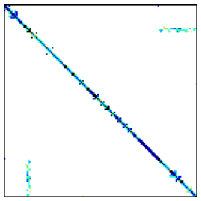
Name: Si41Ge41H72
#rows: 185,639
#columns: 185,639
#nonzeros: 15,011,265
#nonzeros per row (min; avg; max): 13; 80; 662
Author: Y. Zhou, Y. Saad, M. Tiago, J. Chelikowsky
Kind: Theoretical/quantum chemistry problem
Description: Real-space pseudopotential method.



Name: Transient
#rows: 178,866
#columns: 178,866
#nonzeros: 961,368
#nonzeros per row (min; avg; max): 1; 5; 60,423
Author: K. Gullapalli
Kind: Circuit simulation problem
Description: Small circuit from Freescale Semiconductor.



Name: Webbase
#rows: 1,000,005
#columns: 1,000,005
#nonzeros: 3,105,536
#nonzeros per row (min; avg; max): 1; 3; 4,700
Author: unknown
Kind: Weighted directed graph
Description: Web connectivity matrix.



Name: Wind Tunnel
#rows: 217,918
#columns: 217,918
#nonzeros: 11,524,432
#nonzeros per row (min; avg; max): 2; 53; 180
Author: R. Grimes
Kind: Structural problem
Description: Pressurized wind tunnel.

B. Testbeds

A variety of platforms are used for evaluating the proposed sparse matrix algorithms in this thesis. The CPUs, GPUs, Xeon Phi and tightly coupled CPU-GPU heterogeneous processors are listed separately in this Appendix.

B.1 CPUs

The used two CPUs are listed in Table B.1.

Table B.1: Two CPUs used for benchmarking.

Vendor	Intel	Intel
Family	Xeon CPU	Xeon CPU
Device	E5-2630	E5-2667 v3
Codename	Sandy Bridge	Haswell
#Cores	6	8
#SIMD units	6×2×256-bit wide	8×2×256-bit wide
Clock	2.3 GHz	3.2 GHz
SP flop/cycle	96	256
SP Peak	220.8 GFlop/s	819.2 GFlop/s
DP flop/cycle	48	128
DP Peak	110.4 GFlop/s	409.6 GFlop/s
L1 data cache	6×32 kB	8×32 kB
L2 cache	6×256 kB	8×256 kB
L3 cache	15 MB	20 MB
Memory	32 GB DDR3-1333 (4 channels)	32 GB DDR4-2133 (4 channels)
Bandwidth	42.6 GB/s	68.3 GB/s
ECC	on	on
Hyper-Threading	on	on
OS (64-bit)	Ubuntu 12.04	RedHat Enterprise Linux v6.5
Compiler	Intel C/C++ v14.0	Intel C/C++ v15.0.1

B.2 GPUs

The used two nVidia GPUs and one AMD GPU are listed in Table B.2.

Table B.2: Three GPUs used for benchmarking.

Vendor	nVidia	nVidia	AMD
Family	GeForce GPU	GeForce GPU	Radeon GPU
Device	GTX Titan Black	GTX 980	R9 290X
Codename	Kepler GK110	Maxwell GM204	GCN Hawaii
#Cores	15	16	44
#SIMD units	2880 CUDA cores	2048 CUDA cores	2816 Radeon cores
Clock	889 MHz	1126 MHz	1050 MHz
SP flop/cycle	5760	4096	5632
SP Peak	5120.6 GFlop/s	4612.1 GFlop/s	5913.6 GFlop/s
DP flop/cycle	1920	128	704
DP Peak	1706.9 GFlop/s	144.1 GFlop/s	739.2 GFlop/s
L1 data cache	15×16 kB	16×24 kB	44×16 kB
Scratchpad	15×48 kB	16×96 kB	44×64 kB
L2 cache	1.5 MB	2 MB	1 MB
Memory	6 GB GDDR5	4 GB GDDR5	4 GB GDDR5
Bandwidth	336 GB/s	224 GB/s	345.6 GB/s
OS (64-bit)	Ubuntu 14.04	Ubuntu 14.04	Ubuntu 14.04
Device driver	v344.16	v344.16	v14.41
Compiler	g++ v4.9, nvcc v6.5.19	g++ v4.9, nvcc v6.5.19	g++ v4.9, OpenCL v1.2

B.3 Xeon Phi

The used Intel Xeon Phi is listed in Table B.3.

Table B.3: Xeon Phi used for benchmarking.

Vendor	Intel
Family	Xeon Phi
Device	5110p
Codename	Knights Corner
#Cores	60
#SIMD units	60×2×512-bit wide
Clock	1.05 GHz
SP flop/cycle	1920
SP Peak	2016 GFlop/s
DP flop/cycle	960
DP Peak	1008 GFlop/s
L1 data cache	60×32 kB
L2 cache	60×512 kB
Memory	8 GB GDDR5
Bandwidth	320 GB/s
ECC	on
OS (64-bit)	RedHat Enterprise Linux v6.5
Device driver	v3.4-1
μOS	v2.6.38.8
Compiler	Intel C/C++ v15.0.1

B.4 Heterogeneous Processors

The used heterogeneous processors from Intel, nVidia and AMD are listed in Tables B.4 and B.5. One simulated heterogeneous processor is listed in Table B.6.

Table B.4: Intel and nVidia heterogeneous processors used for benchmarking.

Processor	Intel Core i3-5010U		nVidia Tegra K1	
Core type	x86 CPU	GPU	ARM CPU	GPU
Codename	Broadwell	HD 5500	Cortex A15	Kepler
Cores @ clock (GHz)	2 @ 2.1	3 @ 0.9	4 @ 2.3	1 @ 0.85
SP flops/cycle	2×32	3×128	4×8	1×384
SP peak (GFlop/s)	134.4	345.6	73.6	327.2
DP flops/cycle	2×16	3×32	2×2	1×16
DP peak (GFlop/s)	67.2	86.4	18.4	13.6
L1 data cache	4×32 kB	3×4 kB	4×32 kB	1×16 kB
Scratchpad	N/A	3×64 kB	N/A	1×48 kB
L2 cache	4×256 kB	3×24 kB	2 MB	128 kB
L3 cache	N/A	384 kB	N/A	N/A
Last level cache	3 MB		N/A	
DRAM	Dual-channel DDR3-1600		Single-channel DDR3L-1866	
DRAM capacity	8 GB		2 GB	
DRAM bandwidth	25.6 GB/s		14.9 GB/s	
OS (64-bit)	Microsoft Windows 7		Ubuntu Linux 14.04	
GPU driver	v15.36		r19.2	
Compiler	Intel C/C++ 15.0.2		gcc 4.8.2, nvcc 6.0.1	
Toolkit version	OpenCL 2.0		CUDA 6.0	

Table B.5: AMD heterogeneous processors used for benchmarking.

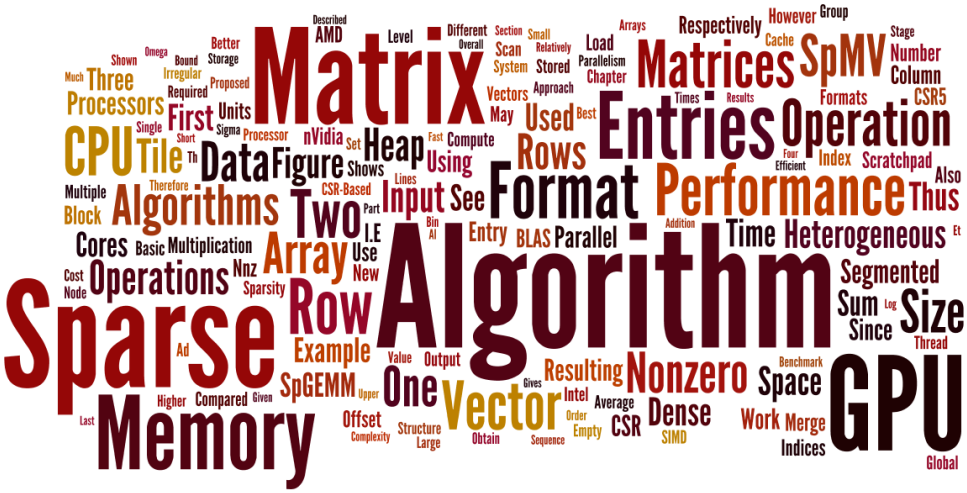
Processor	AMD A6-1450 APU		AMD A10-7850K APU	
Core type	x86 CPU	GPU	x86 CPU	GPU
Codename	Jaguar	GCN	Steamroller	GCN
Cores @ clock (GHz)	4 @ 1.0	2 @ 0.4	4 @ 3.7	8 @ 0.72
SP flops/cycle	4×8	2×128	4×8	8×128
SP peak (GFlop/s)	32	102.4	118.4	737.3
DP flops/cycle	4×3	2×8	4×4	8×8
DP peak (GFlop/s)	12	6.4	59.2	46.1
L1 data cache	4×32 kB	2×16 kB	4×16 kB	8×16 kB
Scratchpad	N/A	2×64 kB	N/A	2×64 kB
L2 cache	2 MB	Unreleased	2×2 MB	Unreleased
DRAM	Single-channel DDR3L-1066		Dual-channel DDR3-1600	
DRAM capacity	4 GB		8 GB	
DRAM bandwidth	8.5 GB/s		25.6 GB/s	
OS (64-bit)	Ubuntu Linux 12.04		Ubuntu Linux 14.04	
GPU driver	13.11 Beta		14.501	
Compiler	gcc 4.6.3		gcc 4.8.2	
Toolkit version	OpenCL 1.2		OpenCL 2.0	

Table B.6: A simulated heterogeneous processor used for benchmarking.

Processor	Simulated Processor	
Core type	x86 CPU	GPU
Product	Intel Core i7-3770	nVidia GeForce GTX 680
Codename	Ivy Bridge	Kepler
Cores @ clock (GHz)	4 @ 3.4	8 @ 1.006
SP flops/cycle	4×16	8×384
SP peak (GFlop/s)	217.6	3090.4
DP flops/cycle	4×8	8×16
DP peak (GFlop/s)	108.8	128.8
L1 data cache	4×32 kB	8×16 kB
Scratchpad	N/A	8×48 kB
L2 cache	4×256 kB	512 kB
L3 cache	8 MB	N/A
DRAM	Dual-channel DDR3-1600	GDDR5
DRAM capacity	32 GB	2 GB
DRAM bandwidth	25.6 GB/s	192.2 GB/s
OS (64-bit)	Ubuntu Linux 12.04	
GPU driver	v304.116	
Compiler	gcc 4.6.3, nvcc 5.0	
Toolkit version	CUDA 5.0	

C. Word Cloud of The Thesis

This appendix includes a "word cloud" generated by `http://www.wordle.net/` by using the text of the thesis as input. It is easy to see the most frequently used words in this thesis.



D. Short Biography

Weifeng Liu is currently a Ph.D. candidate at Niels Bohr Institute, Faculty of Science, University of Copenhagen, Denmark. He is working in the eScience Center under advisor Professor Brian Vinter. Before he moved to Copenhagen, he worked as a senior researcher in high performance computing technology at SINOPEC Exploration & Production Research Institute for about six years. He received his B.E. degree and M.E. degree in computer science, both from China University of Petroleum, Beijing, in 2002 and 2006, respectively. He is a member of the ACM, the IEEE, the CCF and the SIAM.

His research interests include numerical linear algebra and parallel computing, particularly in designing algorithms for sparse matrix computations on throughput-oriented processors. His algorithms run on a variety of many-core devices (e.g., nVidia GPUs, AMD GPUs, Intel GPUs and Intel Xeon Phi) and CPU-GPU heterogeneous processors (e.g., nVidia Tegra, AMD Kaveri and Intel Broadwell).