Optimized GPU algorithms for sparse data problems

Pham, Nguyen Quang Anh

2017

Pham, N. Q. A. (2017). Optimized GPU algorithms for sparse data problems. Doctoral thesis, Nanyang Technological University, Singapore.

<http://hdl.handle.net/10356/72409>

<https://doi.org/10.32657/10356/72409>

NANYANG TECHNOLOGICAL UNIVERSITY

Optimized GPU Algorithms for Sparse Data Problems

Ph.D Thesis

By

Pham Nguyen Quang Anh

Supervisor: Prof. Rui Fan

Co-Supervisor: Prof. Wen Yonggang

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of Doctor of Philosophy

August, 2016

Acknowledgments

I would like to express my gratitude to many people and organization for their support and contribution:

Asst. Prof. Fan Rui, my supervisor, for his invaluable guidance, encouragement, and sharing of his knowledge. He taught me how to do research, to think critically, and to present my work properly. Through discussions, he inspired me of many valuable questions to my research and of my passion to discover solutions. It has been a great honor and pleasure to study and work under his mentorship.

Asst. Prof. Wen Yonggang, my co-supervisor, for his technical advices to my research.

My brother, **P. N. Tuan Anh**, and my Ph.D. friends, for sharing their research experience.

Ng-Goh Siew Lai and other lab technicians for their technical support at the Parallel and Distributed Systems Lab (PDSL).

Last but not least, I thank my family for their unyielding support and encouragement, without which the work would not have been possible.

Contents

Acknowledgments	i
List of Figures	v
List of Tables	vi
Summary	vii
1 Introduction	1
1.1 Dense and sparse problems	1
1.2 GPU computing	2
1.2.1 Factors affecting GPU performance	2
1.3 Contributions	4
1.3.1 Sparse matrix-vector multiplication	4
1.3.2 Sparse matrix-matrix multiplication	5
1.3.3 Graph coloring	6
1.4 Summary and thesis organization	8
2 Literature Review	9
2.1 An Overview of GPUs	9
2.2 Basic Matrix Formats on GPUs	13
2.2.1 Coordinate Format (COO)	13
2.2.2 Compressed Sparse Row (CSR)	14
2.2.3 ELLPACK	15
2.2.4 Hybrid Format (HYB)	15
2.3 Sparse matrix-vector multiplication	16
2.4 Sparse matrix-matrix multiplication	17
2.5 Graph coloring	19
3 Sparse Matrix-Vector Multiplication	22
3.1 Cost of Vector Accesses	24
3.2 Main Techniques	25

3.2.1	Superrows	25
3.2.2	Partial sums	27
3.3	GPU Implementation	27
3.3.1	Data structures	28
3.3.2	Storing the data structures	30
3.3.3	Performing SpMV	31
3.3.4	Performing transformations on the vector	34
3.4	Experiments	36
3.4.1	Datasets and Configuration	36
3.4.2	Throughput comparison	37
3.4.3	Bandwidth comparison	38
3.5	Summary	42
4	Sparse Matrix-Matrix Multiplication	44
4.0.1	A basic SpGEMM algorithm and key performance issues	45
4.1	Main Techniques	49
4.1.1	Counting intermediate products and estimating number of nonze- ros in C	52
4.1.2	Matrix partitioning	54
4.1.3	Generating work lists	55
4.1.4	Summing intermediate products	61
4.1.5	Sorting the output	65
4.2	Experimental Results	66
4.2.1	Throughput comparison	66
4.2.2	Effect of partition size	69
4.2.3	Running time breakdown	70
4.3	Summary	71
5	Graph coloring	72
5.1	Coloring Algorithms	74
5.1.1	Counting-based Jones-Plassmann algorithm	74
5.1.2	Conflict coloring algorithm	78
5.2	Efficient Implementation of CJP and CC	80
5.2.1	Coloring in phases	81
5.2.2	Load balancing	81
5.2.3	Finding unused colors in CJP	82

5.2.4	Finding unused colors in CC	84
5.2.5	Enqueueing nodes	86
5.3	Experimental results	87
5.3.1	Throughput and number of colors	91
5.3.2	Work and efficiency	94
5.4	Summary	97
6	Conclusions and Future Works	98
6.1	Conclusions	98
6.2	Future Works	99
6.2.1	Load balancing	100
6.2.2	Multi-GPU	101
6.2.3	Relation between Matrix Structure and Performance	101
6.2.4	Other Sparse problems	102
	References	103
	Author's Publications	112

List of Figures

2.1	Comparison between CPU and GPUs architecture	10
2.2	GPUs thread and memory hierarchy	11
3.1	Preprocessing matrix M for <i>superrows</i> and <i>partial sums</i>	25
3.2	Main steps of SpMV	32
3.3	Throughput comparison for CUSP, CUSP*, yaSpMV and yaSpMV* . .	38
3.4	Number of transactions in CUSP and CUSP* on low data volume matrices	38
3.5	Number of transactions in yaSpMV and yaSpMV* on low data volume matrices	39
3.6	Number of transactions in CUSP and CUSP* on high data volume matrices	41
3.7	Number of transactions in yaSpMV and yaSpMV* on high data volume matrices	41
4.1	Throughput comparison for CUSP, CUSPARSE, BHSPARSE and our BalancedHash algorithm.	64
4.2	Relative performance of heuristically chosen partition size P versus best partition size.	69
4.3	Break down of running time for each step of BalancedHash.	70
5.1	Illustration of CJP	75
5.2	Illustration of CC	78
5.3	Running times of different algorithm normalized to CC's time	92
5.4	Number of colors used by different algorithms normalized to <i>Smallest- last</i> 's color usage	92
5.5	Number of enqueued nodes (y -axis) in each round (x -axis) for <i>webbase- 1M</i> , <i>patents_main</i> and <i>epidemiology</i>	93
5.6	Size of <i>max-color</i> relative to its final value, as a function of the percent of nodes colored in the first round for <i>cant</i> , <i>harbor</i> and <i>webbase-1M</i> . . .	96

List of Tables

3.1	Total and number of transactions to multiplicand vector x in CUSP and yaSpMV.	24
3.2	Sparse matrices used in the experiments	35
3.3	yaSpMV vector texture cache hit rate	40
4.1	A collection of real life sparse matrices	48
5.1	List of graphs used in experiments	88
5.2	Average running time (in milliseconds) and number of colors used for different algorithms in small graphs.	89
5.3	Average running time (in milliseconds) and number of colors used for different algorithms in big graphs.	90

Summary

Many important problems in science and engineering today deal with sparse data. Examples of sparse data include sparse matrices, where the number of nonzero values is much smaller than the total number possible and where nonzeros are located in scattered instead of regular positions, and graphs in which the average degree is low and the edge set has an irregular structure. This type of data frequently arises from real-world sources and is used to model physical, biological or social phenomena. Many sparse datasets are large, and require parallel computing to process efficiently. But sparsity leads to a number of performance challenges. Irregularities in the size and distribution of data leads to load imbalance between threads and scattered memory accesses that strain memory systems optimized for block based accesses.

Graphics processing units (GPUs) have been successfully used in recent years to solve many big data problems. GPUs can execute thousands of threads simultaneously, and have much higher throughput and memory bandwidth than CPUs. Nevertheless, the GPU architecture is more suited to processing dense, regular datasets. Sparse data problems such as sparse matrix-vector and matrix-matrix multiplication, breadth first search, shortest paths and other graph algorithms achieve much less speedup on GPUs than their dense data counterparts. The problems listed earlier of load imbalance, high memory latency and bandwidth saturation are compounded on the GPUs due to its massively multithreading and SIMD execution model.

In this thesis, we study three fundamental sparse data problems, sparse matrix-vector multiplication (SpMV), sparse matrix-matrix multiplication (SpGEMM) and graph coloring. Each problem is used as a primitive in a number of higher level applications, and thus accelerating these problems leads to a broad range of improvements for other problems. For these problems, we present GPU algorithms based on novel techniques and which offer best in class performance. Our algorithms are based on analyzing the key performance issues and bottlenecks for each problem, and use both heuristical and theoretically motivated techniques to overcome these limitations. While some of the techniques are problem specific, others can be generalized to deal with issues common to many GPU based sparse data computations.

Our SpMV algorithm is based on compacting a sparse matrix to increase its density and regularity of data access, and also making use of the GPU’s fast shared memory to increase the efficiency of repeated SpMV computations. The algorithm reduces I/O for vector accesses by 37% on average, and improves performance up to 35% compared to the previously fastest GPU SpMV algorithm. Our SpGEMM algorithm efficiently enumerates all the work done during a computation to achieve perfect load balancing, and also uses a randomized algorithm to nearly optimally partition the matrix into pieces small enough to be operated on using the fast but limited amount of shared memory. It is up to $2.5\times$ faster than the state of the art GPU SpGEMM algorithm on the most difficult, unstructured matrices. Finally, we present two coloring algorithms optimized respectively for coloring quality and speed. The first algorithm uses a simple counter mechanism to greatly improve overall work efficiency, while the second algorithm achieves both high parallelism and relatively high efficiency by randomly coloring the graph based on estimates of its chromatic number. Compared to existing GPU coloring algorithms, our first algorithm uses $1.1 - 4.3\times$ fewer colors on average, while the second algorithm uses slightly more colors but runs $2.7 - 4.3\times$ faster than other algorithms. The techniques we introduced are the basis for our ongoing work on GPU sparse matrix and graph algorithms, as we seek to bridge the gap between the performance of sparse and dense data algorithms on GPUs.

Chapter 1

Introduction

Recent years have witnessed an explosion of data coming from many sources: the World Wide Web, scientific and medical research, social networks, online transactions, smartphones, sensors etc. For example, Google processes 100 billion search queries each month, and Facebook has 1.7 billion users posting millions of photos, videos and other content daily. The sheer mass of data leads to an ever increasing demand for high performance data processing methods.

1.1 Dense and sparse problems

We can divide data into two broad categories based on its structure and access pattern. In the first category, the data is densely distributed with a well defined structure, and is frequently represented as a dense matrix. Since data items are arranged close to each other, they can be accessed sequentially or in chunks. The computations performed on this data are also very regular, i.e. the instruction streams and amounts of work for different threads are similar. We say these problems are *dense*. Examples of dense problems include dense linear algebra, image processing, Fourier transforms, etc.

The second category of problems are *sparse*, and operate on sparse data structures such as sparse matrices and graphs. Sparse problems are very common in real-world applications, since physical or virtual entities can be represented as nodes and relationships between the entities represented as edges. Since the number of relationships is typically small compared to number of entities, the resulting matrices and graphs are sparse. In sparse problems, data is often located in scattered, random positions, making data accesses highly irregular. In addition, computations are not uniform among

threads, as they may run different instructions or have different workloads. Consider for example the problem of sparse matrix-vector multiplication (SpMV). There are on average only a small number of nonzero values in each row, and these are located in random column. A typical application might assign a thread to process each row. The thread multiplies the nonzeros in its row with the corresponding vector values and sums the products to obtain the vector value for its row. While it is possible to store the matrix so that nonzero values are accessed sequentially, reading the vector entries requires random memory accesses dictated by the structure of the matrix, and this often leads to high memory latency or large amounts of wasted bandwidth. In addition, the amount of work for different threads may vary significantly due to large variations in the number of nonzeros in each row. Other examples of sparse problems include sparse matrix-matrix multiplication [1] [2], breadth first search [3] [4], single source shortest path (SSSP) [5] [6], and many graph or matrix based algorithms.

1.2 GPU computing

Recently, graphic processing units (GPUs) have emerged as a powerful and popular tool for processing large volumes of data. The advantage of GPUs over traditional CPUs comes from its ability to run thousand of threads concurrently, and the fact that GPUs provide much higher total throughput and memory bandwidth than CPUs. Since its inception about a decade ago, general purpose computing on GPUs has played an important role in dramatically accelerating many applications and systems, for example in supercomputing [7] [8], deep learning and artificial intelligence [9] [10], large scale simulations [11] [12], etc.

1.2.1 Factors affecting GPU performance

Due to its architectural design, GPUs are well suited to solving dense problems. However, a number of difficulties are encountered when applying GPUs to sparse problems. We now give an overview of some basic factors affecting GPU performance, especially in the context of sparse data problems. A detailed description of the GPU architecture will be presented in Section 2.1.

A typical GPU is divided into a number of processing blocks called *streaming multiprocessors (SMs)*. Oversimplifying the situation somewhat for ease of exposition, each SM can be considered as a SIMD (single instruction multiple data) processor, typically capable of processing 16 or 32 threads simultaneously. When threads in the SM all run the same instruction, the execution finishes in one step. But if threads need to execute different instructions, their executions are serialized so that only the threads running one instruction are processed in each time step, while the other threads remain idle. For example, consider a binary tree application. At each time point, threads which have arrived at a node may follow different paths depending on the values they are processing. The execution of the threads moving left, right or remaining at the node need to be serialized over three time steps.

Idling also occurs when there is workload imbalance among the threads in a *warp*, a grouping of threads assigned to an SM. If the warp has 32 threads, then in the worst case only one thread still has remaining work, and only $\frac{1}{32}$ fraction of the SM's computing capability is used. Consider for example a parallel reduction operation [13] which sums a sequence of values by combining them in pairs. In step i of the process, a thread k adds the values at positions k and $k + 2^i$. Thus, after each step, half of the active threads will become inactive, so that in the final step of the reduction only one thread in the warp is active. Other types of imbalance include those between different warps, though these have less performance impact because the GPU hardware can quickly switch between warps to occupy the physical GPU cores as long as a sufficient number of warps remain active.

Another important factor in GPU performance is the pattern of memory accesses. GPU memory is divided into *segments* containing multiple (typically 32) data items, similar to cache lines in CPUs. Each time GPU threads request data, the entire segment containing the data is returned. Thus, since an entire warp of 32 threads can all request data at the same time, in the worst case 32 segments of data, each segment containing the single piece of data requested by a thread but additionally also the 31 unrequested items from the segment, are returned for each step of execution of a warp. These *uncoalesced memory accesses* quickly saturate the GPU's

memory bandwidth. Additionally, the data transfer from different segments are serialized, leading to thread idling. To achieve high performance, threads should make *coalesced memory accesses*, where different threads from a warp request consecutive data items from the same segment. In this case, all the data is delivered in parallel in a single bandwidth-conserving memory transaction. To illustrate the memory coalescing issue in practice, consider the GPU merge path algorithm [14] for merging two sorted arrays. There are two main steps. First, threads in parallel partition the input arrays into equal size blocks. Then, each thread performs a linear merge in its partition. The first step is done using binary searches, and during this process threads read random memory locations, resulting in scattered, uncoalesced memory accesses. Another example comes from implementing a global memory hash table, as is done in for example Demouth *et al*'s [15] sparse matrix-matrix multiplication algorithm. While hash tables allow more efficient search and insert operations than for example sorting, they inherently require accessing random memory locations, leading to inefficient noncoalesced memory accesses.

1.3 Contributions

As we have shown, sparse data problems, despite their prominence and importance, pose a number of challenges for GPUs. In this thesis, we propose novel techniques for addressing these challenges. We focus on three fundamental sparse data problems, sparse matrix-vector multiplication (SpMV), sparse matrix-matrix multiplication (SpGEMM) and graph coloring. These problems are used as primitives in a number of higher level applications and thus have wide ranging impact. Nevertheless, their performance on GPUs is currently constrained due to their sparse nature. The three algorithms we introduce all achieve best in class performance. Below, we give an overview of our contributions and techniques.

1.3.1 Sparse matrix-vector multiplication

The SpMV problem consists of multiplying a sparse matrix, with on average $o(n)$ nonzeros in each of its n rows, by a dense vector. It is a fundamental operation

in linear algebra, numerical algorithms and also has applications in combinatorial algorithms. SpMV has been studied extensively on the GPU [16–20]. As we stated earlier, the main issues for SpMV are high memory bandwidth consumption and load imbalance. Most previous works sought to address the memory bandwidth issue by developing improved matrix storage formats to minimize the amount of matrix data transfer [18, 19]. However, these techniques do not address the high bandwidth consumption from accessing the vector, which we found accounts for over half the total memory traffic in experiments on real world sparse matrices.

To overcome this issue, we propose a technique called *superrows* to in effect increase the density of the matrix, which results in more tightly packed and coalesced memory accesses. Superrows also serve as a form of caching, to reduce memory traffic from accessing the same vector entry multiple times. Another observation we make is that many applications using SpMV, for example the conjugate gradient method or the PageRank computation, perform SpMV on the same matrix repeatedly. We propose a technique to store certain compressed values used in consecutive iterations in fast but small blocks of GPU memory called *shared memory*, which reduces traffic to the larger but slower main memory. Lastly, a novel feature of our techniques is that they are complementary to existing techniques based on optimized matrix formats. Thus, our techniques can be used on top of existing methods to further increase their performance. We applied this method to two state of the art GPU SpMV algorithms, CUSP and yaSpMV [17, 20]. On experiments over a wide range of real world sparse matrices, we reduced vector I/O by on average 37% and 36%, resp., and increased overall performance by 20 – 77% and 10 – 35%, resp.

1.3.2 Sparse matrix-matrix multiplication

SpGEMM requires computing the product of two sparse matrices. It is also an important primitive operation, and is used in higher level applications including algebraic multigrid methods [21], breadth first search [22], and shortest paths algorithms [23], among others. There are a number of algorithms for performing SpGEMM on GPUs [2, 15, 21]. However, the challenges caused by sparsity are so severe for

SpGEMM that, until the recent breakthrough BHSPARSE algorithm [1], no GPU SpGEMM algorithm could consistently beat an optimized CPU one.

SpGEMM is usually performed in a row by row manner. The main steps are to multiply values from the two input matrices to produce a set of *intermediate products*, and then combine intermediate products into the nonzero values of the output matrix. The three main challenges are the severe load imbalance that comes from generating different numbers of intermediate products for different rows, the amount of work needed to combine the large number of intermediate products, which often exceed the final number of nonzeros by two orders of magnitude, and the noncoalesced memory accesses used in the combining process.

In our work, we propose a novel method to efficiently enumerate all the intermediate products that arise during an SpGEMM computation. These are stored in a compact way as *work lists*, which are later divided evenly among the threads, resulting in perfect load balancing in the most computationally intensive part of the SpGEMM computation. To combine the large number of intermediate products efficiently, we use a hash table, as opposed to existing methods which are based on work-inefficient sorting. As we noted earlier, storing hash tables in global memory results in poor performance due to uncoalesced memory accesses. However, we propose a way to partition the matrix into small enough parts so that we can store hash tables in the much smaller but faster GPU shared memory. A perfect partitioning would require knowing the number of nonzero values in the final product matrix, a task which is as difficult as the SpGEMM computation itself. However, we propose an efficient, theoretically sound randomized algorithm which experimentally produces nearly perfect estimates of the number of nonzeros. Combining these techniques, we show that our *Balanced Hash* SpGEMM method outperforms the state of the art BHSPARSE algorithm by $1.5\times$ on average, and up to $2.5\times$ on most irregular and hard to handle matrices.

1.3.3 Graph coloring

The graph coloring problem is to assign each node of a graph a color, such that neighboring nodes receive different colors. The goal is to use the minimum number

of colors possible. While the problem is NP-hard, a number of effective heuristics exist in practice. Graph coloring has been studied extensively on CPUs [24–27], and due to a number of applications of graph coloring in linear algebra operations such as incomplete LU-decomposition, has also attracted recent interest from GPU computing researchers [28–30].

Parallel graph coloring algorithms mainly follow two approaches. The first approach is based on the Luby-Jones-Plassman PRAM coloring algorithm, and involves iteratively coloring maximal independent sets of nodes over different rounds. However, this method sometimes results in insufficient parallelism, especially on GPUs capable of processing tens of thousands of concurrent threads. A second, more scalable approach tries to color the nodes optimistically, allowing neighbors to temporarily receive the same color, before detecting these conflicts and correcting them.

We present two algorithms, called *counting Jones-Plassmann (CJP)* and *conflict coloring (CC)*, which significantly improve on both the performance and coloring quality of existing algorithms. CJP, like the traditional Jones-Plassmann method, colors maximal independent sets (MIS). But it uses a lightweight method that associates a counter with each node to quickly identify the MIS in each round. We also show a fast GPU method to color nodes using the minimum allowed color, which reduces our overall color use. Compared to existing GPU algorithms, CJP is on average $1.5 - 2\times$ faster, and at the same time uses $1.1 - 4.3\times$ fewer colors on average.

Our second algorithm CC further improves on the speed of CJP, while using slightly more colors. It is more suited to applications where coloring speed is most important, for example in applications where colorings need to be computed repeatedly. CC allows nodes to temporarily receive the same color before correcting the conflicts. However, it uses a novel technique to reduce the number of conflicts, so that on average each node is only processed 1.3 times. While current conflict coloring based methods choose the minimum available color for each node to minimize color use, thus resulting in large numbers of nodes selecting the same colors, we propose an efficient way to color nodes with a random color, greatly reducing the probability of conflict. To do this while still using a small number of colors, we propose a

fast, parallel way of estimating the chromatic number, i.e. the minimum number of colors needed to color the graph, which then provides nodes the range from which they choose their random colors. CC is on average $2.7 - 4.3\times$ faster than existing algorithms, while using between $3.5\times$ fewer colors to $1.1\times$ more colors.

1.4 Summary and thesis organization

In this thesis, we studied three key sparse data problems. We proposed a number of techniques to address the limitations of GPU algorithms for these problems. The techniques exploit the unique combinatorial properties of each problem. At the same time, they may be adapted to more general settings and thereby help address general performance issues for GPUs on sparse data problems. They also provide a basis for our ongoing and future work on efficient GPU computing for challenging irregular problems.

The remainder of this thesis is organized as follows. In Chapter 2, we present background on the GPU architecture and programming model, and also discuss previous research on the problems we study. Chapter 3 describes our work on the SpMV problem. We analyze in depth the key performance problems, discuss our new techniques and contrast them with existing techniques. Chapter 4 describes our work on SpGEMM. We discuss the main current approaches and the problems they face, then introduce several new techniques to address these issues. We also present extensive experimental results. Chapter 5 describes our GPU algorithms for graph coloring. We discuss both existing CPU and GPU coloring techniques, then present our CJP and CC algorithms optimized for reducing color use and improving coloring speed, resp. Lastly, in Chapter 6, we present conclusions for the thesis and discuss some future research directions.

Chapter 2

Literature Review

In this chapter, we first discuss basic formats to store sparse matrix or sparse graph on GPUs. We then present and analyze some related works on 3 problems: SpMV, SpGEMM and Graph coloring. We also highlight the novelty of our contributions compared to existing methods.

2.1 An Overview of GPUs

In recent years, Graphics Processing Units (GPUs) has emerged as a promising computing platform in High Performance Computing. GPUs outperform CPU both in computing throughput and memory bandwidth. The appearance of NVIDIA's CUDA, which is a programming and runtime platform, allows scientists and developers to write programs solving different computing tasks. GPUs are significantly faster than CPU in processing dense regular data such as dense matrix-vector or matrix-matrix multiplication, LU factorization, etc. However, when processing sparse irregular data, the achieved performance on GPUs is substantially lower than the theoretical peak throughput of the device. This is due to the GPUs architecture and programming model, which we will discuss in this section.

A GPU consists of multiple Streaming Processors (SMs), each of which contains many processing cores. The total number of cores in GPUs is from hundreds to few thousands. Although each core operates at lower frequencies than CPU cores, the massive amount of processing units in GPUs enables us to execute a lot more data in parallel and, thus, obtain a much higher number of computing operations at a

time (throughput). Figure 2.1 shows the differences in architecture between GPU and CPU.

Figure 2.1: Comparison between CPU and GPU's architecture

Based on this architecture, NVIDIA introduced the programming platform CUDA. In CUDA, programs running on devices are called *kernels* and launched by the host program in CPU. A kernel consists of thousands of lightweight threads. The threads are grouped into *thread blocks*, each of which executes entirely on a SM. Threads within a block can be synchronized with each other using `--syncthreads()` instruction explicitly. At any time, GPUs can run a certain number of thread blocks so that the resources (amount of threads, shared memory, register) required by them do not exceed physical limit of GPUs. When a thread block terminates, the *scheduler* will start another block in no particular order. Threads in a block are organized into groups of 32 consecutive threads called *warps*. Threads within a warp are synchronized implicitly and they always run the same instruction at a same time. If a warp's threads need to run different instructions (like in branching **if..then..else** statement), the executions are serialized. This is called *divergence*, and should be avoided to achieve high performance. At every cycle, the scheduler will select an eligible warp to execute, so it is critical to have eligible warps at any time to keep GPUs busy. One way to do this is having a high *occupancy*, i.e. concurrently running a large number of threads on each SM.

An important concept in GPU's architecture is memory hierarchy.

Figure 2.2: GPUs thread and memory hierarchy

- (i) **Global memory:** The main memory of GPUs is the DRAM or *global memory*. Before starting a kernel, GPUs transfer data into global memory. Global memory can be accessed by any SM and any thread. This is one of the ways threads in different blocks communicate to each other. It is typically 1-8 GB in size, and has latency of 300-600 cycles. Global memory is organized into segments of consecutive bytes, each of which is typically 128 bytes or 32 4-byte integers in size. When threads in a warp execute a memory instruction we have a global memory request. If the requested addresses are on a single segment, this segment will be transferred from global memory and a memory *transaction* is issued. We say this request is *coalesced*. If threads access data from multiple segments, the request is called *uncoalesced*, and the data from all the segments are transferred serially from global memory. This results in multiple transactions, greater latency and higher memory bandwidth consumption up to $32\times$. For efficiency, it is crucial to make global memory accesses as coalesced as possible. The high latency of global memory makes threads idle waiting for requested data. To hide this la-

tency, a common approach is launching many threads (high occupancy) so that there are always eligible threads for GPU's cores to execute.

- (ii) **Shared memory:** Each SM has an on-chip memory called *shared memory*. Shared memory has a latency of around 20 cycles, and there are fewer penalties to access data randomly compared to global memory. Each thread block occupies a part of shared memory when running on a SM. Threads within a thread block can access any data in that block's shared memory. The contents of shared memory are controlled programmatically, so it can act as a user defined cache for frequently accessed data.
- (iii) **L1 cache:** Beside shared memory, *L1 cache* is another on-chip memory. Actually, shared memory and L1 cache are the same physical memory component, which is 64 KB in size. Before launching a kernel, we can declare how much shared memory a thread block will use. If thread blocks do not use all on-chip memory at a time when running on a SM, the remaining amount of memory will go to L1 cache. In contrast to shared memory, we cannot access L1 cache explicitly from our program.
- (iv) **Registers:** Registers hold private data of each thread and cannot be accessed by other threads. The total number of 32-bit registers per SM is 65K. In recent CUDA versions, threads in a warp can exchange data in their registers using intrinsics such as shuffle instructions and it is faster than sharing data via shared memory.

The number of threads per block, amount of shared memory per thread block and number of registers per thread will determine how many thread blocks and threads can run in a SM at any time or the occupancy of a CUDA kernel launch.

Beside above types of memory, GPUs also have Constant memory, Texture memory and Local memory. Constant and Texture memory have the same accessing scope as Global memory, but they are read-only and often used in special applications. Local memory has the same scope as Register but performs slower.

Figure 2.2 depicts the GPU thread and memory hierarchy.

Because of this architecture and SIMD programming model, GPUs are especially suitable to problems with regular computation and data access pattern. These problems do not create execute divergence or uncoalesced memory access and they can keep GPUs active most of the time. Thus, the performance of these problems is bounded by how fast GPUs device can compute. Lee et al. [31] report that in compute-bound algorithms such as Single precision floating General Matrix Multiply (SGEMM), Fast Fourier Transform (FFT) or Bilateral filter (Bilat) GPUs can achieve up to 66% peak flops, which is highest throughput we can get when using shared memory. This also brings to $2.8\times$ to $4\times$ speedup over CPU on these algorithms.

2.2 Basic Matrix Formats on GPUs

As we mostly work on sparse matrices and graphs in our research, we will review some storage formats for sparse matrices and graphs in this section. Since a graph is represented by a connectivity matrix, these data structures are used for both graph and matrix.

Sparse matrix formats for GPU computation were first introduced in the work of SpMV by Bell and Garland [17]. In their paper, they introduced a number of sparse matrix formats used in SpMV; however, some formats are useful in other problems. Many later researches have proposed other formats extended from those of Bell and Garland. To demonstrate the matrix layout, we use the following matrix:

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

2.2.1 Coordinate Format (COO)

The coordinate format is a simple storage scheme in which we do not store zero entries and only nonzeros are represented. Given a sparse matrix M , let $nnz(M)$ be the number of nonzeros values in M . The COO format for M consists of three arrays,

row_M , col_M and val_M , each of size $nnz(M)$. The arrays are sorted in order of the row then column indices. For $0 \leq i \leq nnz(M) - 1$, $row_M[i]$ and $col_M[i]$ are the row and column indices of the i 'th nonzero value in this ordering, and $val_M[i]$ is its value. So we have COO representation for our example matrix as follows:

$$\begin{aligned} row_M &= [0 \ 0 \ 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3] \\ col_M &= [0 \ 1 \ 1 \ 2 \ 0 \ 2 \ 3 \ 1 \ 3] \\ val_M &= [a \ b \ c \ d \ e \ f \ g \ h \ i] \end{aligned}$$

2.2.2 Compressed Sparse Row (CSR)

The CSR format is similar to COO, but avoids repeating values in the row array. In particular, CSR retains the col_M and val_M arrays from the COO representation for M , but replaces row_M with an array ptr_M of size $n + 1$, where n is the number of rows in M . As with COO, the arrays are sorted in order of row then column indices. For $0 \leq j \leq n - 1$, $ptr_M[j]$ gives the index in col_M and val_M of the first nonzero element in the j 'th row of M . Also, $ptr_M[n] = nnz(M)$ gives the number of nonzeros in M . Again using our example matrix, we have:

$$\begin{aligned} ptr_M &= [0 \ 2 \ 4 \ 7 \ 10] \\ col_M &= [0 \ 1 \ 1 \ 2 \ 0 \ 2 \ 3 \ 1 \ 3] \\ val_M &= [a \ b \ c \ d \ e \ f \ g \ h \ i] \end{aligned}$$

The CSR format requires about 1/3 less storage than the COO format for the same matrix. In addition, it enables easily accessing a row of the matrix: the data for row i lies between indices $ptr[i]$ and $ptr[i + 1] - 1$ of the col and val arrays. The advantage of the COO format is that it enables quickly determining the row in which a nonzero element lies.

The COO and CSR are considered general purpose formats since all matrices are naturally stored in these representations. Moreover, many type of algorithms can operate on matrices in these formats. In two storage schemes, CSR is widely used because it allows programs to process an entire row which is a common computing pattern of many algorithms.

The following sparse matrix formats were represented in [17]; however, they are suitable to SpMV problem only.

2.2.3 ELLPACK

Given an $m \times n$ matrix, the ELLPACK format stores it in an $m \times k$ matrix, where k is the maximum number of nonzeros in any row. Rows with fewer than k nonzeros are padded with extra zeros. The column indices and matrix values are stored explicitly, while the row indices are implicit. For example, M is represented as:

$$indices_M = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}, val_M = \begin{bmatrix} a & b & * \\ c & d & * \\ e & f & g \\ h & i & * \end{bmatrix}$$

The elements in these 2 matrices are left-shifted and stored in column-wise order in ELLPACK format. Hence, the memory layout of these matrices in memory will be:

$$indices_M = [0 \ 1 \ 0 \ 1 \ 1 \ 2 \ 2 \ 3 \ * \ * \ 3 \ *] \\ val_M = [a \ c \ e \ h \ b \ d \ f \ i \ * \ * \ g \ *]$$

When working with matrix in ELLPACK format, each thread processes a row so that threads access data in $indices_M$ and val_M in a coalesced way.

2.2.4 Hybrid Format (HYB)

The ELLPACK format is well-suited to vector architectures; however, its efficiency significantly degrades when the number of nonzeros per row varies. Meanwhile, the storage and computation efficiency of the COO format is independent of the variance of row lengths. Therefore, a hybrid ELLPACK/COO format called HYB was proposed from the idea of storing majority matrix entries in ELLPACK format and the remaining nonzeros in COO format. The number of columns in the ELLPACK format, k , have to be decided based on a certain rule. In the [17], k is determined such that the number of rows with at least k nonzeros is less than $1/3$ of the total rows.

Based on the fundamental work of Bell and Garland, many sparse matrix formats have been proposed such as Slide ELLPACK [32], ELLPACK-R [33], Blocked

ELLPACK [18], BCCOO/BCCOO+ [20], etc. Most of these storage schemes were presented by studies on SpMV; we will discuss them in the next section.

2.3 Sparse matrix-vector multiplication

A number of researchers have studied SpMV on GPUs in recent years. Bolz et al. [16] conducted some of the first research in this area. Bell and Garland [17] proposed GPU kernels for several sparse matrix formats including COO, ELLPACK, HYB, CSR-scalar, CSR-vector, DIA and PKT. Their work forms the core of Nvidia’s SpMV library CUSPARSE [34] and the open-source library CUSP [35]. Choi [18] later extended ELLPACK to the Blocked ELLPACK (BELLPACK) format. BELLPACK does not perform well on matrices with high variance in row lengths, such as web matrices. To deal with these matrices, Yang et al. [36] organized the matrix in tiles, within which rows are sorted by length and stored in either ELLPACK or CSR format. Tang et al. [19] introduced a compressed bit-representation of Nvidia’s HYB format called BRO-HYB. They also proposed a GPU-friendly decompression algorithm for their format. Sun et al. [37] presented a CRSD storage scheme for diagonal matrices. To improve the memory efficiency of ELLPACK, Monakov et al. [32] divided the matrix into slices containing adjacent rows and store each slice in ELLPACK format. In this Sliced-ELLPACK format, the number of padding zeros is bounded by the difference between the shortest and the longest row in each slice instead of the whole matrix. Vázquez et al.’s [33] ELLPACK-R format also improved ELLPACK by using an additional array to indicate the number of nonzeros in each row. Su et al. [38] introduced a composite storage scheme called COCKTAIL which uses different formats in different parts of the matrix. They also proposed a framework that analyzes a matrix and returns a suitable representation. Similarly, Grewe et al. [39] built a compiler to automatically generate and tune code for SpMV kernels on GPUs. Liu et al. [40] introduced a method to analyze the caching effect in SpMV when considering both texture and L2 cache. Yan et al. [20] proposed the state of the art yaSpMV algorithm, which uses the blocked BCCOO/BCCOO+ formats to reduce the size of

the row and column index arrays, and reduces the size of the row array using bit flags. yaSpMV utilizes its own segmented scan that is much more efficient than the original COO format segmented scan. Lastly, yaSpMV uses a sophisticated optimizer to determine the best storage and execution parameters.

The papers above mainly focus on decreasing the amount of I/O and computation to access the matrix. One approach that has been used to decrease I/O to the vector is to store part of the vector in the GPU’s texture cache [17, 20, 36]. However, as the texture cache line size is 32 bytes, this may result in unused vector values being loaded into cache and wasted cache space. Another difficulty with this approach is controlling the number of threads sharing the cache, because an excessive number of threads causes cache thrashing and premature eviction of useful values. In our algorithm, we control the content of the shared memory and the threads using it, and ensure that the stored values get maximally reused before loading new values. There has also been work on caching the vector in the GPU’s L1 cache [41], but this can also result in unused cached values.

Another approach for improving vector I/O is to permute the rows and columns of the matrix to minimize its bandwidth and make it more clustered. That is, we seek a permutation to transform the matrix closer to a block diagonal one. A number of heuristics exist, such as Reverse Cuthill-McKee, Dulmage-Mendelsohn decomposition and nested dissection. However, finding the optimal permutation is NP-complete. Our superrows technique can be seen as an efficient way of increasing the amount of clustering in a matrix. Moreover, it is optimized for the GPU’s memory architecture, whereas other bandwidth reduction heuristics need to be adapted to run efficiently on GPUs. Finally, there has been work on tiling the matrix and vector in order to improve vector locality [36]. But synchronizing the tiles requires launching a new GPU kernel for each tile, which causes overhead and was found to be ineffective.

2.4 Sparse matrix-matrix multiplication

Demouth [15] proposed an SpGEMM algorithm which is used in Nvidia’s CUSPARSE library [34]. The method uses hash tables in both the shared memory and global mem-

ory for managing intermediate products. The algorithm first tries to insert entries into the shared memory hash table, and if this fails it switches to global memory. It retries until the entry is inserted into one of the two tables. This technique can lead to high divergence in warp executions because some threads may attempt more insertions than others. In addition, since the shared memory is small, many intermediate products are inserted into the global memory hash table, leading to large amounts of uncoalesced accesses.

The CUSP library [35] uses an SpGEMM method called expansion, sorting and compression (ESC) [2]. In the first step, the algorithm generates (or expands) all intermediate products and stores them in an array C^* . Next, it sorts entries in C^* by row and column using radix sort. Finally, it sums (or compresses) all adjacent entries with the same row and column indices to compute matrix C 's nonzeros. As explained earlier, this method suffers from high sorting cost. Dalton *et al* [2] introduced an improvement to ESC by performing the sorting and compression on each row of C^* using fast shared memory. However, this method still ends up sorting all the intermediate products. It also assigns either a thread, warp or block to process each row, and therefore may not be load balanced.

Liu and Vinter [1] proposed BHSPARSE, which is to our knowledge currently the fastest GPU SpGEMM algorithm. Like [2], [1] also processes C row by row. The algorithm needs to allocate sufficient global memory to store C . However, since the size of C is not known *a priori*, the algorithm launches a new kernel if the current allocation is too small. This process may need to be repeated, and so on large matrices this can lead to multiple kernel launches and high cost to copy data from the old C array to the new one. To combine intermediate products, [1] uses different sorting algorithms depending on the number of intermediate products in each row. Some of these algorithms, such as bitonic sort, are not work efficient, while others, such as heap sort, cause thread divergence. Finally, since [1] either sorts or merges all the intermediate products, it incurs a high cost on matrices with large compression ratios.

2.5 Graph coloring

Graph coloring is a classic problem in computer science and has been studied extensively in both the theoretical and applied domains. Coloring a graph with the minimum number of colors is NP-hard [42], but a number of heuristics exist to find good colorings quickly. The *first fit* greedy method visits the graph in an arbitrary order and colors each node using the smallest color not used by its neighbors. The *largest degree first* heuristic [24] visits nodes in order of nonincreasing degree and has been shown to reduce the number of colors used. Other effective node orderings include *smallest degree last* [25], *incidence degree* [26] and *saturation degree* [27]. Gebremedhin *et al* presented efficient implementations of these heuristics and others in the CPU-based graph coloring library ColPack [43]. Hasenplaugh *et al* [44] recently introduced several orderings for parallel graph coloring with provable performance guarantees and implemented them on multicore CPUs.

One of the first works on parallel graph coloring was by Luby *et al* [45], and used a round based scheme which colored a maximal independent set of nodes in each round. A random number is assigned to each node, and each independent set is formed by choosing all the uncolored nodes whose values are larger than those of their uncolored neighbors. Jones and Plassmann [46] extended this algorithm to the asynchronous setting. Our CJP algorithm is based on the Jones-Plassmann technique optimized for GPUs.

Gebremedhin and Manne [47] proposed an optimistic algorithm for coloring graphs in parallel on shared memory computers. Their technique consists of three phases. Each processor is responsible for a set of nodes, and in the first phase the processor colors its own nodes, possibly resulting in conflicting colorings with other processors. In phase 2 processors detect conflicts in parallel, and finally in phase 3 the conflicting nodes are recolored sequentially. Çatalyürek *et al* [48] extended this approach by adding the conflicting nodes to a list and recursively coloring them in later rounds. Rokos *et al* [49] further improved the performance of [48] by reducing the amount of synchronization.

Our CC algorithm is similar to the methods in [47] and [48] in that we also allow nodes to have conflicting colors and later recolor them. However, a key difference is that we choose a random unused color within some range for each node, whereas [47] and [48] choose the minimum allowed color. Using a random color gives each node a larger range of colors to select from, which reduces the number of conflicts and the amount of redundant work. The difference is especially important on GPUs. Indeed, Rokos *et al* [49] state that their implementation of [48]’s algorithm on GPUs failed to terminate due to an infinite loop caused by choosing the minimum color. The difficulty in using a random color is choosing the range to select the color from. In CC we propose a method to dynamically adjust the color range so that it quickly attains a size large enough for avoiding conflicts while still minimizing color use.

In recent years GPUs have emerged as an important platform for high performance and general purpose computing. Grosset *et al* adapted [47]’s algorithm to the GPU but perform some computations on the CPU. Cohen and Castonguay [29] introduced an algorithm based on the Luby-Jones-Plassmann (LJP) technique, and which is implemented as the `csrcolor` function in Nvidia’s popular `cuSPARSE` library. They introduced an optimization which assigns multiple random values to each node, which improves coloring speed at the expense of using substantially more colors. Recently Naumov *et al* [28] applied [29]’s algorithm to perform efficient incomplete LU factorization on the GPU, improving performance up to 8 times compared to the CPU. Che *et al* [50] proposed a hybrid coloring method for the GPU which improves load balancing. In early stages of the algorithm they color high degree nodes as in the largest degree first heuristic, while in later stages they use the LJP technique. Deveci *et al* [30] recently introduced two high quality coloring algorithms based on Gebremedhin and Manne’s optimistic technique [47] for the GPU and Xeon Phi processors. Their vertex-based algorithm processes an entire node using a thread while the edge-based algorithm processes one edge per thread. The edge-based technique generally uses fewer colors and is also faster. However, each edge may be processed multiple times and so the total amount of work done by the algorithm may be high.

For more details about the vast literature on graph coloring we refer the reader to the survey by Gebremedhin *et al* [51]. In addition to basic graph coloring, the survey also discusses problems such as distance- k , star and hypergraph coloring, and also discusses applications for these algorithms and additional theory. Other works on bounds for the chromatic number can be found in [52], [53], [54] and [55], among many others.

Chapter 3

Sparse Matrix-Vector Multiplication

Sparse matrix-vector multiplication forms the basis for a wide range of algorithms in a number of scientific domains. Recent years have witnessed extensive research on implementing SpMV in graphics processing units. The high throughput and bandwidth provided by these devices have led to GPU SpMV implementations that run substantially faster than CPU counterparts. SpMV is inherently memory bound, and one of the key factors to achieving high performance is minimizing the amount of memory I/O. This I/O can be divided into two main parts, accesses to the sparse matrix M and accesses to the multiplicand vector x .

Much of the past work on SpMV in GPUs has focused on reducing the accesses to M [17], [18], [20], [19] (as discussed in Section 2.3). While these techniques are highly effective at reducing matrix I/O, they do not significantly decrease the I/O for accessing vector values. The pattern of vector accesses is determined by the structure of the matrix, and SpMV generally makes a large number of scattered accesses. This type of I/O is costly on a GPU, whose memory system is optimized for regular accesses. We conducted experiments on a large number of matrices, and observed that on average vector accesses accounted for over 54% of the main memory I/O in CUSP and over 68% of the I/O in yaSpMV. Therefore reducing vector I/O is crucial for improving GPU SpMV performance.

In this Chapter, we introduce several techniques to significantly decrease the amount of vector I/O. Our first technique is called *superrows*. It combines a number

of sparse rows in the matrix into a denser superrow with better locality properties, and also reduces the redundant I/O from reading the same vector value in different rows. Superrows are stored in GPU on-chip shared memory, and can be seen as a way of “perfectly” caching vector values so that they are loaded and evicted at exactly the times needed. Another technique we introduce is called *partial sums*. SpMV is often performed iteratively, with the output vector from one iteration becoming, sometimes after certain transformations, the input for the next iteration. We propose a method to store part of the output vector in shared memory in order to compute certain partial sums. These values are used in the next iteration of SpMV to reduce the amount of vector data read.

Superrows and partial sums are in themselves effective at reducing vector and overall I/O. But moreover, because they focus on reducing vector I/O, they are “compatible” with existing GPU SpMV algorithms, which are largely focused on reducing matrix I/O. That is, we can take an existing algorithm and modify it to perform vector accesses using our techniques, while retaining the efficient matrix accesses of the original algorithm to reduce both matrix and vector I/O. Furthermore, the modification require only small changes to the code of the original algorithm. We demonstrated this approach on two open-source GPU SpMV algorithms CUSP and yaSpMV. CUSP was chosen for its popularity, and yaSpMV was chosen because it is currently the fastest GPU SpMV algorithm. The modified algorithms CUSP* and yaSpMV* retain almost all the code of the original algorithms, plus our vector optimization code.

Both CUSP* and yaSpMV* perform significantly better than the original algorithms on a variety of sparse matrices. On average, CUSP* reduces the amount of vector I/O and total I/O by 37% and 31% resp. compared to CUSP. It performs especially well on certain web matrices featuring a certain amount of clustering; here CUSP* decreased vector I/O up by 70%. CUSP* also improved the overall throughput by 20-77% compared to CUSP. Compared to yaSpMV, yaSpMV* on average reduced vector and total I/O by 36% and 31%, resp., and up to 45% for some web matrices. yaSpMV* improved total throughput by 10-35%.

Table 3.1: Total and number of transactions to multiplicand vector x in CUSP and yaSpMV.

	CUSP		yaSpMV	
	Total tx	Tx to x	Total tx	Tx to x
Webbase	927,548	468,759	118,158	74,737
Circuit	341,511	224,015	40,124	29,136
Economics	601,676	433,090	53,355	38,702
FEM/Accelerator	1,183,887	901,368	102,251	84,591
FEM/Cantilever	800,015	496,891	39,374	24,887
FEM/Harbor	617,603	355,030	28,740	19,211
FEM/Ship	1,483,739	877,679	78,736	53,541
FEM/Spheres	1,170,494	743,274	68,636	48,346
Protein	1,053,356	696,470	31,550	16,764
QCD	267,136	144,256	28,186	18,818
Wind Tunnel	1,574,111	809,537	139,890	95,100
Circuit5M	13,835,521	5,868,219	1,514,926	1,060,454
Ga41As41H72	2,924,753	1,579,635	880,367	761,049
mip1	1,339,250	369,438	122,844	86,053
Si41Ge41H72	2,523,578	1,345,239	704,308	612,281
uk1m	11,547,802	5,714,790	551,505	368,089
uk100k	925,706	493,292	45,260	29,829
cnr-2000	1,118,408	504,895	76,223	45,327
in-2004	5,335,993	2,175,132	282,816	164,379
eu-2005	7,394,855	4,504,867	531,825	396,165

3.1 Cost of Vector Accesses

The matrix storage formats described in Chapter 2 allow efficient coalesced accesses to the matrix. However, because of the scattered distribution of nonzeros in most sparse matrices, accesses to the vector are frequently uncoalesced, causing excessive I/O which saturates the main memory bandwidth. Table 3.1 shows the total number of memory transactions and the number of memory transactions for accessing the vector x by CUSP and yaSpMV on a variety of sparse matrices taken from [20] and the LAW project [56–58]. These numbers are obtained by using NVIDIA profiler

(nvprof). Two things can be seen. The first is that yaSpMV performs far fewer total memory transactions than CUSP, which partly accounts for yaSpMV's much higher performance. The second is that a large majority, 54% for CUSP and 68% for yaSpMV, of the memory transactions are for accessing the vector, as opposed to the matrix. This clearly shows the importance of optimizing vector I/O.

3.2 Main Techniques

In this section, we describe two techniques, superrows and partial sums, to achieve more coalesced vector accesses. We show how to use the techniques to implement SpMV on a GPU in the next section. In the following, x and y refer to the vectors in the computation $y = Mx$, where M is a sparse matrix.

Figure 3.1: Preprocessing M : (a) Extract dense rows for column groups and partial sums matrices. (b) Remaining entries of M used to form row groups. (c) The superrow for row group 0. (d) The partition matrix for row group 0. (e) Combining partition matrix with partial sums to form augmented partition matrix 0.

3.2.1 Superrows

Since the nonzeros in many sparse matrices are scattered, if we process each row of the matrix individually, each memory transaction is unlikely to return more than

a few vector values used in the row. Also, since multiple rows of the matrix may contain nonzeros in the same coordinates, accessing the matrix row by row results in redundant vector accesses on the repeated coordinates. The way we address both problems is to combine multiple rows of the matrix into one superrow. In particular, given a set of rows $r_1, \dots, r_k$, let S_i be the set of indices of the nonzero values in r_i , for $1 \leq i \leq k$. Then the *superrow* formed from $r_1, \dots, r_k$ is an index set $S = \bigcup_{1 \leq i \leq k} S_i$. The superrow indices are more densely distributed than the indices in each r_i , and also remove any repeat elements in the r_i 's. Hence, if we access x using superrow S instead of row by row using $r_1, \dots, r_k$, we obtain significantly more coalesced memory requests. We will store the x values corresponding to a superrow in shared memory so they can be accessed efficiently later.

As an example, consider row group 0 in part (b) of Figure 3.1, which consists of the first 4 rows of matrix M' ¹. The indices of the nonzeros for these rows are $S_1 = \{1, 6\}$, $S_2 = \{7\}$, $S_3 = \{4, 7\}$, and $S_4 = \{2\}$. Assume the segment size is 3, so that 3 consecutive values in x can be returned in one memory transaction. Then accessing elements in x for multiplication with nonzeros in partition 1 row by row requires 2, 1, 2, and 1 transactions, resp., or a total of 6 transactions. On the other hand, the superrow formed by these rows is $\{1, 2, 4, 6, 7\}$. Accessing the nonzeros in the superrow requires only 3 transactions, both due to the higher density of the superrow and the elimination of the redundant access to 7.

Superrows can be seen as a form of caching. However, as the structure of the matrix is fixed and known ahead of time, we can control when to load the vector values for each superrow to achieve maximal reuse of the values. This corresponds to caching in a “perfect” way, so that vector values are loaded and evicted at precisely the times they are needed. Indeed, as we show in Section 3.4, our techniques achieve substantial improvements in vector I/O even in systems which cache vector values.

¹ M' is a submatrix of M whose construction is described in Section 3.3.1.1. The details of the construction are not important for this example.

3.2.2 Partial sums

Many applications use SpMV iteratively, where the result vector from one iteration becomes, sometimes after certain transformations, the input vector in the next iteration. In existing GPU implementations, this is done by first writing the (transformed) result of the current iteration, say y , to GPU main memory, then reading back y on the next iteration. Instead of performing these redundant reads and writes, our *partial sums* technique stores a portion of y in shared memory (limited by the size of shared memory), performs some of the computations needed in the next iteration of SpMV using the stored part of y , then writes these partial sums to main memory. This reduces the amount of vector data read in the next iteration.

Part (a) of Figure 3.1 shows an example of partial sums. Matrix K is a submatrix of M ². We break K into two groups of 4 columns each. Let y be a 8×1 column vector that is the result of the previous iteration of SpMV, and assume we can store 4 values of y in shared memory. If we currently have the first 4 values of y in shared memory, we can multiply these with the first group of columns in K , producing partial sum values PS_1 and PS_4 as shown in the figure. Similarly, if we have the last 4 values of y in shared memory, we can multiply them with the second column group to get partial sums PS_2, PS_3 and PS_5 . Now, we can write values $PS_1, \dots, PS_5$ to main memory, then read back and sum them on the next iteration to get the next product. Since the number of partial sum values is much less than the number nonzero values in K , this can significantly reduce the overall number of vector accesses each iteration.

3.3 GPU Implementation

In this section, we describe how to use our vector optimization techniques in an SpMV algorithm. While the techniques can be used by themselves in a stand-alone algorithm, they work better when integrated with an existing SpMV algorithm optimized for matrix I/O. In this section, we discuss the procedure for integrating with yaSpMV [20].

² K consists of the high density portions of M , and its construction is described in Section 3.3.1.2. The details of the construction are not important for this example.

We call the integrated algorithm yaSpMV*. The procedure for integrating with other algorithms such as CUSP is similar, though we will point out some of the differences as they arise.

Given a matrix, we first describe how to preprocess it to produce certain data structures used for partial sums and superrows. Then we describe how to perform SpMV using these data structures. Lastly, we describe how to perform transformations on the vector after each iteration of SpMV when using our vector optimizations, which is necessary in some applications.

3.3.1 Data structures

Let M be an $n \times n$ matrix. The preprocessing for M involves first splitting it into two matrices, a dense submatrix on which partial sums is most effective, and the remaining values on which we apply the superrows technique. The main steps are illustrated in Figure 3.1.

3.3.1.1 Superrows

A superrow is formed from a set of rows in M , and contains the indices of the nonzero entries in those rows. Each superrow has size at most s , where s depends on the size of the shared memory. In our implementation $s = 4000$. We only apply superrows to the part of the matrix not processed using partial sums. Thus, consider the matrix M' formed from M by removing all the values occurring in any of the column groups $CG_0, \dots, CG_{n/c} - 1$. Assume first that no row in M' contains more than s nonzero values; we describe how to deal with such long rows later. We form the superrows by going through the rows of M' one by one. For each row, we add the indices of the nonzero values in the row to the current superrow, as long as the superrow's size will not exceed s . If the size exceeds s , we form a new superrow and add the row's indices to it. Let $SR_0, \dots, SR_{t-1}$ be the set of superrows we form. Note that each superrow corresponds to a submatrix, i.e. a consecutive set of rows of M' . We say a row from this submatrix is *contained* in the superrow.

If a row of M' contains more than s values, we mark the row and do not use superrows when computing its dot product. That is, we read the vector values used in the row from main memory instead of shared memory when processing the row. In practice, s is several thousand, so there are few such long rows in most sparse matrices.

Next, we describe the partition matrices. Consider a superrow SR of size $s' \leq s$, and let R be a $r \times n$ submatrix of M' corresponding to SR . Then we form a $r \times s'$ *partition matrix* corresponding to SR , by removing all the columns from R that contain only zeros. This is possible because SR has size s' , meaning that among all the rows in R there are exactly s' columns containing nonzero values. The procedure for forming partition matrices is shown in parts (c)-(d) of Figure 3.1.

3.3.1.2 Partial sums

To use partial sums, we store a certain number, say c , of values from the vector x in shared memory. This allows us to multiply by c corresponding columns from matrix M . Thus, our first step is to divide M into groups of c columns, denoted $C_0, \dots, C_{n/c-1}$. The size of c is limited by the amount of the shared memory. In our implementation $c = 1000$. Each C_i is an $n \times c$ matrix containing columns ci to $c(i+1) - 1$ of M . Next, we want to select a submatrix of C_i consisting of dense rows. Using partial sums incurs overhead from reading and writing the partial sums values, so we only use it on dense rows which would require a large number τ of transactions to process without partial sums; in our implementation $\tau = 4$. We form an $n \times c$ *column group* matrix CG_i from C_i , by taking exactly the rows of C_i that require at least τ transactions to read; all the other rows of CG_i contain only zero values. To compute the number of transactions needed to read a row of C_i , we partition the row into blocks of 32 consecutive values corresponding to the data read by each transaction, and count the number of nonempty blocks. The construction of column groups is shown in part (a) of Figure 3.1. For visual clarity, we included all rows with at least 3 nonzeros (as opposed to requiring 4 transactions) in the column groups.

In addition to the column groups, we also form two *partial sums* matrices PS and PS' . Both matrices are $n \times \frac{n}{c}$, and have one column corresponding to each of the $\frac{n}{c}$ column groups. Entry $PS(i, j)$ contains the sum formed by adding up the values from the i 'th row of CG_j times the corresponding values from x , i.e.

$$PS(i, j) = \sum_{k=0}^{c-1} CG_j(i, k)x_{cj+k}.$$

PS' holds the partial sums from the current iteration of SpMV, and PS holds the sums from the previous iteration.

3.3.2 Storing the data structures

After the preprocessing step we have a set of column groups, partial sums matrices, superrows and partition matrices. We store the superrows as arrays in GPU global memory. For the matrices, since we are combining our algorithm with yaSpMV, we store them in yaSpMV's BCCOO/BCCOO+ format in global memory. This is done by running yaSpMV's optimizer on the matrices, which produces the most efficient storage format for each matrix. We call the outputs *compressed* matrices.

We can simplify the code for our algorithm somewhat by breaking up the partial sums matrix PS into $c \times n/c$ size submatrices and storing the i 'th such submatrix with the i 'th partition matrix P_i , appending it to the right of P_i as shown in part (e) of Figure 3.1. We call this an *augmented* partition matrix. As described in the next section, this allows us to combine the output from the superrows and partial sums portions of the algorithm using one SpMV-like operation. The augmented partition matrix is compressed using yaSpMV's optimizer producing a file F . We later need to be able to write to arbitrary entries in PS , and so we have to be able to locate these entries in F . To do this, we can perform a preprocessing step where we append a $c \times (n/c)$ size dummy matrix containing specially marked entries to P_i , compress the matrix, then scan it to find the marked locations, giving us the location of each PS entry. We store these locations in an array ps_addr in global memory.

Procedure 1 *SpMV* SpMV code for thread tid in block bid .

```
1:  $i \leftarrow bid \times c$ ;  $j \leftarrow (bid + 1) \times c$ 
2: VectorPartitionMultiply( $i, j$ )
3: ComputePartialSum( $Y$ )
4:
5: for ( $t \leftarrow tid$ ;  $t < c$ ;  $t \leftarrow t + block\_size$ ) do
6:    $y[i + t] = Y[t]$ 
7: end for
```

Procedure 2 *VectorPartitionMultiply* Multiply partition matrix by x using super-rows.

```
1:  $SR \leftarrow$  first superrow containing row  $i$ 
2:  $P' \leftarrow$  augmented partition matrix corresponding to  $SR$ 
3: repeat
4:    $t \leftarrow tid$ 
5:   while  $t < length(SR)$  do
6:      $index \leftarrow SR[t]$ 
7:      $b[t] \leftarrow x[index]$ 
8:      $t \leftarrow t + block\_size$ 
9:   end while
10:   $\_syncthreads()$ 
11:
12:   $[i', j'] \leftarrow$  range of rows in  $P'$  modulo  $c$ 
13:   $Y[i', j'] \leftarrow SpMV\_kernel\_sh(P', b)$ 
14:   $SR \leftarrow$  next superrow
15:   $\_syncthreads()$ 
16: until all  $c$  values of  $Y$  are computed
17:  $\_syncthreads()$ 
```

Procedure 3 *ComputePartialSum* Compute partial sums for the next iteration.

```
1:  $CG \leftarrow Column\_Group[bid]$ 
2: SpMV_kernel_sh2( $CG, Y, ps\_addr$ )
3:  $\_syncthreads()$ 
```

3.3.3 Performing SpMV

We now describe how to perform SpMV using the data structures above. Pseudocode for the implementation is given in the listings 1, 2 and 3 above. Given a block of threads with block ID bid , we use these threads to compute coordinates $c \cdot bid$ to $c \cdot (bid + 1) - 1$ of the result vector $y = Mx$. For brevity, let $i = c \cdot bid$ and $j = c \cdot (bid + 1) - 1$. The thread block uses a shared memory array b of size s to store values from x , and a shared memory array Y of size c to store coordinates i to j of y . The thread block performs the following steps.

Figure 3.2: Main steps of SpMV (example for superrow 0): (a) Load values of x to shared memory using superrow 0, multiply with partition matrix 0 and add partial sums to produce result vector y . (b) Multiply column group 0 by Y to produce partial sums for next iteration.

- (i) The rows from i to j may correspond to several superrows. We perform the following for each such superrow. Suppose a superrow SR corresponds to rows i' to j' of M' , for $i \leq i' < j' \leq j$. We use the indices in SR to load values of x from global memory into b (lines 5 to 10 of Procedure 2). SR has a corresponding partition matrix P . To compute the values of y in coordinate i' to j' , we need to multiply P by the vector represented by b , then add in the corresponding partial sums values. This is illustrated in part (a) of Figure 3.2.

We perform both steps together by calling the function $SpMV_kernel_sh(P', b, Y)$ in line 13 of Procedure 2. Here, P' is the augmented partition matrix corresponding to SR , consisting of P appended with partial sums values. $SpMV_kernel_sh$

is based on yaSpMV's SpMV function. In particular, it reads in P' from global memory and parses it based on the BCCOO/BCCOO+ format, and performs dot products using yaSpMV's improved segmented scan. However, *SpMV_kernel_sh* has been modified to read in vector values from the shared memory array b instead of global memory, and to write the output to the shared memory array Y instead of global memory. Additionally, *SpMV_kernel_sh* adds in the values from the partial sums portion of P' . The partial sums values in P' can be identified as those that lie in column s' or greater of P' , where s' is the size the superrow SR . For example, in part (a) of Figure 3.2, the superrow has size 5, and the partial sums lie in columns 5 and higher in P' .

Making the above modifications required going through yaSpMV's SpMV function and changing all the references to vectors stored in global memory to shared memory vectors, and also adding two lines of code to add in the partial sums values. In total, we modified 17 lines of code in yaSpMV. The resulting algorithm yaSpMV* combines our vector optimizations with yaSpMV's matrix and segmented scan optimizations. When integrating our algorithm with CUSP, we made similar modifications, changing 20 lines of code in CUSP.

At the end of the first step, we have the values of y in coordinates i to j stored in Y .

- (ii) Next, we compute the partial sum values for the next iteration. To do this, we need to multiply CG_{bid} , which contains the nonzero values in columns i to j of M that are used to calculate the partial sums, with array Y and write the output to the partial sums portion of the appropriate augmented partition matrix.

We do this by calling the function *SpMV_kernel_sh2*(CG_{bid}, Y, ps_addr). Like *SpMV_kernel_sh*, *SpMV_kernel_sh2* is derived from yaSpMV's SpMV function. It reads CG_{bid} from global memory in BCCOO/BCCOO+ format and performs yaSpMV's improved segmented scan. However, it has been modified

to read vector values from the shared memory array Y . Additionally, it writes the output to global memory, at the addresses indicated in ps_addr . Recall from Section 3.3.2 that ps_addr contains the addresses of the partial sums in the augmented partition matrices. *SpMV_kernel_sh2* largely changes the same lines of code as *SpMV_kernel_sh*, plus one additional line of code changed to redirect the output to the ps_addr addresses.

At the end of this step, the partial sums used in the next iteration are stored in the augmented partition matrices. This step is illustrated in part (b) of Figure 3.2.

Finally, in lines 5-7 of Procedure 1, the values for coordinates i to j of y are flushed from shared memory to main memory.

3.3.4 Performing transformations on the vector

In many applications that use SpMV iteratively, we may perform other operations on the vector besides matrix-vector multiplication, such as rescaling the vector by a factor, taking its norm or checking it for convergence. For example, the PageRank computation largely consists of applying the following formula iteratively until convergence:

$$\pi^{(k+1)} = cM^T\pi^{(k)} + (1 - c)\pi^{(0)}$$

Here, $\pi^{(k)}$ is the PageRank vector in iteration k , M is the sparse transition matrix for a set of webpages, c is a damping factor and $\pi^{(0)}$ is the initial vector. After each iteration of SpMV, the result vector $M^T\pi^{(k)}$ needs to be scaled by c and added to another vector.

Our vector optimizations can accommodate such transformations with no additional I/O. If the transformation on the result vector y of the current iteration is known before the iteration is complete, for example if we multiply y by a constant, then the transformation can be done as the coordinates of y are being computed, i.e. during step (a) in Figure 3.2. Otherwise, if the transformation is only known after all of y has been computed, for example if we scale y by its norm, then we complete the

Table 3.2: Sparse matrices used in the experiments

Name	Size	Nonzeros
webbase-1M	$1\text{M} \times 1\text{M}$	3,105,536
Circuit	$171\text{K} \times 171\text{K}$	958,936
dense2	$2\text{K} \times 2\text{K}$	4,000,000
Economics	$207\text{K} \times 207\text{K}$	1,273,389
Epidemiology	$526\text{K} \times 526\text{K}$	2,100,225
FEM/Accelerator	$121\text{K} \times 121\text{K}$	2,620,000
FEM/Cantilever	$62\text{K} \times 62\text{K}$	4,007,383
FEM/Harbor	$47\text{K} \times 47\text{K}$	2,374,001
FEM/Ship	$141\text{K} \times 141\text{K}$	7,813,404
FEM/Spheres	$83\text{K} \times 83\text{K}$	6,010,480
Protein	$36\text{K} \times 36\text{K}$	4,344,765
QCD	$49\text{K} \times 49\text{K}$	1,916,928
Wind Tunnel	$218\text{K} \times 218\text{K}$	11,634,424
Circuit5M	$5.56\text{M} \times 5.56\text{M}$	59,524,291
Ga41As41H72	$268\text{K} \times 268\text{K}$	18,488,476
mip1	$66\text{K} \times 66\text{K}$	10,352,819
Si41Ge41H72	$168\text{K} \times 168\text{K}$	15,011,265
uk-1m	$1\text{M} \times 1\text{M}$	41,247,159
uk-100k	$100\text{K} \times 100\text{K}$	3,050,615
cnr-2000	$325\text{K} \times 325\text{K}$	3,216,152
in-2004	$1.38\text{M} \times 1.38\text{M}$	16,917,053
eu-2005	$862\text{K} \times 862\text{K}$	19,235,140

computation of y for this iteration, compute its norm, then on the next iteration of SpMV, scale the input vector and partial sum values by the norm as they are being read in from global memory, i.e. during step (a) in Figure 3.2 of the next iteration.

3.4 Experiments

3.4.1 Datasets and Configuration

In this section we evaluate the effectiveness of our vector optimizations, both in reducing the amount of vector and total I/O, and also in improving SpMV performance. We ran experiments on a large variety of matrices taken from [20] and the LAW project [56–58]. The matrices and their properties are summarized in Table 3.2. The sizes of the matrices were limited by the amount of global memory on our GPU. We combined our techniques with two existing SpMV algorithms, CUSP (v0.4) and yaSpMV, producing the modified algorithms CUSP* and yaSpMV*.

As described in Section 3.3.1, instead of storing the matrix M as a whole, we preprocess it into a number of smaller matrices. In CUSP*, we store each such matrix in the HYB format, a combination of the ELL and COO formats, using the methodology suggested in [17], namely we take the largest m for which at least one third of the rows have m nonzeros, and use m columns for the ELL portion of the matrix, and store the remaining nonzeros in COO format. For yaSpMV*, we ran yaSpMV’s optimization algorithm to determine the format of each matrix.

The time measurements for performing SpMV given in this section do not include the preprocessing time. One reason for this is that we performed the processing in Matlab using its sparse matrix library. We believe the preprocessor can run an order of magnitude faster by implementing it in C/C++. Another reason is that CUSP and yaSpMV also perform their own preprocessing, which may take longer than our preprocessing. For example, yaSpMV took about 60 seconds preprocess the *uk-1m* matrix on the machine we used, whereas our Matlab preprocessor took 10 seconds on the matrix. Lastly, we can largely amortize the preprocessing time since it is performed only once for each matrix, while applications such as conjugate gradient may perform thousands of SpMV’s on the same matrix.

In each of our experiments on CUSP* or yaSpMV*, we performed 5 consecutive iterations of SpMV and report the average result. We did this because the first iteration of CUSP* or yaSpMV* may be somewhat slower than the subsequent iterations,

since it does not benefit from the bandwidth reductions from using partial sums. All the iterations after the first one have approximately the same cost.

The experiments were performed on an Nvidia Tesla K20Xm GPU containing 2688 cores in 14 SMs. The peak single precision performance of the device is 3935 GFlop/s and the memory bandwidth is 250 GB/s. Each SM has 48 KB of shared memory. We wanted to maximize the amount of shared memory allocated to each thread block, while still ensuring high occupancy (number of active threads) on each SM. We ran two thread blocks with 1024 threads per block on each SM, which gives 24 KB of shared memory per thread block. The version of CUDA is 7.0.

3.4.2 Throughput comparison

Figure 3.3 shows the performance of CUSP* and yaSpMV* compared to CUSP and yaSpMV. yaSpMV is currently the fastest GPU SpMV algorithm, and yaSpMV* improves yaSpMV's performance on almost all the matrices we tested. The largest improvements are in *Webbase* and *FEM/Harbor*, at 35% and 32%, resp. yaSpMV* also showed improvement on matrices where yaSpMV's performance was already high, such as in *FEM/Ship*, *FEM/Spheres*, *Protein* and *Wind Tunnel*. The only matrices on which yaSpMV* performed worse than yaSpMV are *circuit5M* and *mip1*. These matrices both contain a number of long rows with many scattered nonzeros. Our superrows technique is ineffective on these types of matrices, since it works best when it can combine a large number of sparse rows. On average, yaSpMV* improved over yaSpMV by 12%.

CUSP* also improved over CUSP in most matrices. The largest increases were in *Protein*, *uk1m* and *uk100k*, at 65%, 64% and 78%, resp. These matrices all contained a certain amount of clustering, where certain sets of nodes were more likely to be connected to each other. This structure is advantageous for both superrows and partial sums, because the former can group the clustered nodes into a superrow, and the latter can represent the sum of a large number of clustered coordinates as one partial sum. The average improvement of CUSP* for all the matrices was 14%. There were several matrices on which CUSP* performed worse than CUSP, for example *Webbase*, *dense2*

Figure 3.3: Throughput comparison for CUSP, CUSP*, yaSpMV and yaSpMV*

and *Epidemiology*. The main reason for this was low thread occupancy, i.e. there were not enough threads with work to do. This is because our techniques break a matrix into smaller column group and partition matrices. The column groups in particular have very few nonzero rows after our preprocessing steps, and since CUSP’s assigns one thread per row to process the HYB format, most of the threads were idle. This problem did not arise with yaSpMV* because yaSpMV’s BCCOO format is better at balancing work across the threads.

3.4.3 Bandwidth comparison

Figure 3.4: Number of transactions in CUSP and CUSP* on low data volume matrices

We now look at the reduction in I/O from using our techniques. As we focus on reducing vector I/O, we first need a way to distinguish vector I/O from matrix I/O. We use Nvidia’s *nvprof* profiling tool to count the number of uncached read transactions (which missed in both the L1 and L2 cache) and write transactions to global

Figure 3.5: Number of transactions in yaSpMV and yaSpMV* on low data volume matrices

memory. For each matrix, we first use *nvprof* to count the total number of transactions an algorithm (e.g. yaSpMV or yaSpMV*) incurs; call this number *total_tx*. Then we modify the algorithm so that it does not perform any transactions for vector accesses, by simply removing the parts of the algorithm that read from the input x vector or write to the output y vector; call the number of transactions in this case *matrix_tx*. Then we calculated the number of transactions used to access the vector as $total_tx - matrix_tx$. Note that this number may not be exactly accurate, as some accesses to the matrix or vector may be cached and not result in main memory transactions. However, as this error occurs when measuring the vector transactions performed by any algorithm, then when comparing the number of vector transactions of (say) yaSpMV and yaSpMV*, the errors largely cancel out. Hence, these measurements serve as a good estimate of the I/O savings our techniques achieve.

The number of total transactions, vector transactions and matrix transactions performed by CUSP and CUSP*, and by yaSpMV and yaSpMV* are shown in figures 3.4, 3.5, 3.6 and 3.7. Note that for each pair of algorithms we divided their transaction measurements into two graphs, consisting of matrices on which a small or large number of transactions were performed. The figures show that in all matrices vector I/O accounts for a majority of the overall global memory transactions. In CUSP, on average 54% of the transactions are vector accesses. This number is even higher for yaSpMV, 68%, since matrix I/O in yaSpMV is more efficient.

Table 3.3: yaSpMV vector texture cache hit rate

Name	Hit rate
webbase-1M	67.7%
Circuit	48.1%
Economics	56.2%
Epidemiology	69.9%
FEM/Accelerator	54.5%
FEM/Cantilever	18.5%
FEM/Harbor	23.1%
FEM/Ship	13.8%
FEM/Spheres	2.2%
Protein	19.6%
QCD	9.0%
Wind Tunnel	18.5%
Circuit5M	54.5%
Ga41As41H72	52.1%
mip1	75.2%
Si41Ge41H72	49.5%
uk-1m	59.4%
uk-100k	46.0%
cnr-2000	82.7%
in-2004	74.0%
eu-2005	71.1%

Our techniques reduced the number of total and vector transactions for all the matrices in both CUSP and yaSpMV. In the best case, CUSP* reduced the number of vector I/O in *uk1m* and *uk100k* by 70% and 60%, resp. These are both web matrices, and as noted earlier, the nonzeros are distributed in clusters which benefits both our superrow and partial sum techniques. CUSP* also reduced the vector I/O for another web matrix *webbase* by 46%. Nevertheless, CUSP*'s throughput was worse on *webbase* because of low thread occupancy, as stated earlier. On average, CUSP* reduced vector I/O by 37%.

yaSpMV* also reduced vector I/O on average by 36%, even though yaSpMV

Figure 3.6: Number of transactions in CUSP and CUSP* on high data volume matrices

Figure 3.7: Number of transactions in yaSpMV and yaSpMV* on high data volume matrices

uses the texture cache to cache vector values³ and tries to maximize data locality in the BCCOO format. yaSpMV* achieves the highest vector savings on *eu-2005* and *FEM/Accelerator* (45%) and *FEM/Harbor* (44%). Table 3.3 shows yaSpMV’s cache hit rate for vector accesses. This is quite high in certain matrices, for example *cnr-2000* (83%), *mip1* (75%) or *in-2004* (74%). Yet even on these matrices our optimizations reduced vector I/O by 34%, 28% and 41%, resp. For *eu-2005*, yaSpMV*

³CUSP also uses the texture cache for vector values, though it is less effective.

reduced vector I/O by 45%, even though yaSpMV had a 71% hit rate on this matrix. Hence, our methods are significantly more effective than caching at reducing vector I/O.

Lastly, even though our focus was on vector I/O, our methods also helped to reduce the number of matrix transactions. One reason for this is that by decreasing the amount of vector data transferred from global memory, we freed up more space in the L1 and L2 cache to hold matrix data, thereby reducing the amount of matrix I/O.

At the time our SpMV research was conducted, yaSpMV was the newest and most state-of-the-art SpMV framework. Since then, a number of studies have introduced new SpMV methods, some of which presented new matrix formats [59], [60], [61], [62] while others devised improved techniques based on basic formats [40], [63], [64]. However, these works did not compare their performance directly with yaSpMV. Based on the reported GFLOPs, it seems that yaSpMV’s throughput is about 2 times higher than the other libraries. Since our algorithm is faster than yaSpMV, we infer that our algorithm is likely faster than the above works. Additionally, our improvements are of a different nature than the previously proposed techniques, since we focus on reducing vector transactions instead of optimizing the matrix data structure. In this thesis, we showed that our vector optimizations are complementary to matrix optimizations, i.e. they can be added on top of matrix optimizations to further improve performance. In the studies above, [60] has a similar approach as ours in that they reconstruct the COO format to make vector accesses more coalesced. In addition, they also improve the parallel reduction of COO format by performing it in fast shared memory. Overall, [60] demonstrated a high cache hit rate and high performance. However, it was not clear whether the performance improvement was due to better I/O behavior or faster reduction.

3.5 Summary

In this Chapter, we studied methods to improve SpMV performance on GPUs by reducing vector I/O. We introduced two novel techniques, superrows, which forms

a dense row from sparse ones, and partial sums, which computes sums using values in shared memory to reduce I/O in the next SpMV iteration. We also described how to integrate our techniques with CUSP and yaSpMV. The modified algorithms have significantly reduced vector and total I/O and consistently better throughput. In the future, we would like to study how we can use our techniques generically to improve vector I/O for any GPU SpMV algorithm. We would also like to study the overheads incurred using our optimizations and ways to reduce them. We would like to investigate other ways of using shared memory to reduce I/O. Another problem we want to study is how to apply our method in multi-GPU environment. Finally, we would like to understand how the the performance and I/O of GPU SpMV is affected by the structure of the matrix.

Chapter 4

Sparse Matrix-Matrix Multiplication

Given sparse matrices A and B , the SpGEMM problem is to compute the product $C = AB$. SpGEMM is a key component in a number of important algorithms, including algebraic multigrid methods [21], breadth first search [22], and algorithms for shortest paths [23], colored intersection [65] and subgraph detection [66]. Efficient algorithms for SpGEMM are therefore of fundamental importance.

Graphics processing units (GPUs) have been highly successful in accelerating a number of linear algebra operations. For example, they offer more than an order of magnitude speedup on structured problems such as dense matrix-matrix multiplication. Recently there has been considerable interest in applying GPUs to unstructured problems such as SpGEMM. But the speedups on these problems have thus far been limited. The main reason is that GPUs are optimized for balanced, regular control flow and memory accesses. SpGEMM however, often produces very imbalanced workloads, because the amount of work in different portions of the output matrix depends on the patterns of nonzeros in the input matrices. These patterns also lead to the production of a large number of temporary intermediate products, which are summed to produce a much smaller number of nonzeros in the output. Summing the intermediate products is typically done by sorting [1], or combining them using hashing [15]. Since the number of intermediate products is often much larger than the number of nonzeros in the output, sorting suffers from high overhead. The hashing method, meanwhile, is often performed in the large but slow GPU global memory. This results in high numbers of uncoalesced memory accesses, which are expensive on the GPU.

In this Chapter, we propose several novel methods to address the above problems. We first show how to achieve perfect load balancing when generating and combining the intermediate products, which forms the most expensive stage of the SpGEMM computation. This is done using one of two low cost methods to generate work lists containing the memory locations accessed during the intermediate products computation. To deal with the large number of intermediate products, we use hashing to combine them, reducing them to a much smaller number of nonzero output values. Our hash tables are implemented in on-chip shared memory, resulting in much better memory performance. However, since the amount of shared memory is very small compared to global memory, only a small portion of the intermediate products can be inserted into the hash table at a time. A novel aspect of our algorithm is to control the number of inserted intermediate products to fully utilize the hash tables while not overflowing them. To do this, we give a lightweight method to estimate the number of nonzeros in the output matrix. Combining these techniques, our algorithm achieves significantly higher performance than existing GPU SpGEMM algorithms. Compared to the SpGEMM algorithms in the CUSP [35] and CUSPARSE [34] libraries, as well as the state-of-the-art BHSPARSE algorithm [1], our algorithm is $5.6\times$, $2.4\times$ and $1.5\times$ faster on average, respectively. The algorithm performs especially well on highly irregular matrices, with large variations in the the number of nonzeros per row and producing large numbers of intermediate products. Here, we achieve up to $11.8\times$, $9.5\times$ and $2.5\times$ the performance of the other algorithms, respectively.

4.0.1 A basic SpGEMM algorithm and key performance issues

We now describe a basic template for performing SpGEMM used by most GPU SpGEMM algorithms, including ours. The algorithms differ in how they implement specific parts of the template, as we describe later. Suppose we are given two sparse matrices A and B and want to compute $C = AB$. Since A and B are typically represented in CSR or COO format, we can efficiently iterate only through a row of the

Procedure 1 Pseudocode for SpGEMM

```

1:  $C_{ij} \leftarrow \perp$  for all  $i, j$ 
2: for all  $A_{i*}$  in matrix  $A$  do
3:   for all nonzero  $A_{ij}$  in  $A_{i*}$  do
4:     for all nonzero  $B_{jk}$  in  $B_{j*}$  do
5:        $v_{ijk} \leftarrow A_{ij}B_{jk}$ 
6:       if  $C_{ik} = \perp$  then
7:          $C_{ik} \leftarrow v_{ijk}$ 
8:       else
9:          $C_{ik} \leftarrow C_{ik} + v_{ijk}$ 
10:      end if
11:    end for
12:  end for
13: end for
    
```

matrix, not a column. Therefore, we perform a row based SpGEMM as follows. Normally, we want to store B in column-wise CSC format for easier access B 's nonzeros. However, the cost of converting matrix from CSR to CSC is quite high and can be as much as 50% of SpGEMM time [67]. There is also storage overhead for converting matrix to CSC, especially for big matrices. That is why in our research we keep both matrices in CSR format.

Let M_{ij} represent the element in the i 'th row and j 'th column of a matrix M . Also, let M_{i*} represent the i 'th row of M , and M_{*j} represent the j 'th column. We first initialize all values in the output matrix C to $\perp$. Then for each row i of A , we iterate through the nonzero elements in the row. If an element lies in column j , we proceed to iterate through row j of matrix B . Given a nonzero element in B_{j*} , say in column k , we compute an *intermediate product* $v_{ijk} = A_{ij}B_{jk}$. Notice that v_{ijk} contributes to the value of C_{ik} in the output C . We say that v_{ijk} *belongs* to C_{ij} . We also say v_{ijk} lies *in* rows A_{i*} and C_{i*} . We will check whether C_{ik} is still initialized to $\perp$. If so, we set $C_{ik} \leftarrow v_{ijk}$; otherwise we set $C_{ik} \leftarrow C_{ik} + v_{ijk}$. Notice that after iterating through a row A_{i*} , the algorithm produces all the values in row C_{i*} of the output. This algorithm is shown in Algorithm 1. We assume all the for loops iterate in order of increasing indices.

4.0.1.1 Load balancing

Algorithm 1 gives a basic skeleton for SpGEMM. However, an important issue when implementing the algorithm on a GPU is how to assign threads to process the intermediate products. Most algorithms assign a group of threads, consisting either of a single thread, a warp or a block of threads, to produce all the intermediate products in one row of A . However, there can be large variations in the number of intermediate products in different rows, and so assigning such a coarse grouping of threads to each row can result in significant load imbalance and idling among the threads, leading to poor performance. As an example, consider matrices A and B in (4.2). The intermediate products in each row of A are as follows:

Row 1: $v_{0,0,1} = aw, v_{0,2,0} = bt, v_{0,2,1} = by, v_{0,2,2} = bz$

Row 2: $v_{1,1,1} = cx$

Row 3: $v_{2,0,1} = dw, v_{2,1,1} = ex$

Here, there is a factor of 4 difference between the amount of work in the first and second rows. We measured the variation in work for a number of real life sparse matrices from the University of Florida sparse matrix collection [68]. The **Imb.** column in Table 4.1 shows the imbalance in the number of intermediate products per row in terms of the coefficient of variation (COV). That is, letting σ and μ denote the standard deviation and mean, respectively, of the number of intermediate products in different rows of A , the COV equals $\frac{\sigma}{\mu}$. The COV can be as large as 680%, indicating very high work imbalance in some matrices.

4.0.1.2 Combining intermediate products

Another important performance issue is how to sum together all the intermediate products v_{ijk} belonging to the same nonzero value C_{ik} . For example, for A and B in (4.2), we have $C = AB$ as shown in (4.1). Here, we need to sum $v_{0,0,1}$ and $v_{0,2,1}$ to get $C_{0,1}$, and sum $v_{2,0,1}$ and $v_{2,1,1}$ to get $C_{2,1}$.

Table 4.1: A collection of real life sparse matrices. Each matrix has the same number of rows and columns. The second and third columns give the number of rows and nonzero values in each matrix A . The $\mathbf{nnz}(C)$ column gives the number of nonzeros in the matrix $C = A^2$. The $\mathbf{np}(C)$ column gives the number of intermediate products produced when computing C . The **Comp.** column shows the ratio of the number of intermediate products to nonzero values for C . The **Imb.** column shows the coefficient of variation of the number of intermediate products in each row of A .

Name	Rows	$\mathbf{nnz}(A)$	$\mathbf{nnz}(C)$	$\mathbf{np}(C)$	Comp.	Imb.
cant	63K	4M	17.4M	269.5M	15.5	29%
economics	207K	1.3M	6.7M	7.6M	1.1	69%
epidemiology	526K	2.1M	5.2M	8.4M	1.6	3%
accelerator	121K	2.6M	18.7M	79.9M	4.3	65%
QCD	49K	1.9M	3M	11.8M	6.9	0%
harbor	47K	2.4M	7.9M	156.5M	19.8	90%
scircuit	171K	1M	5.2M	8.7M	1.7	136%
webbase-1M	1M	3.1M	51.1M	69.5M	1.4	680%
spheres	83K	6M	26.5M	463.8M	17.5	31%
ship	141K	7.8M	24.1M	450.6M	18.7	17%
pwtk	218K	11.6M	32.8M	626.1M	19.1	13%
protein	36K	4.3M	19.6M	555.3M	28.1	37%
2cubes_sphere	101K	1.6M	9M	27.5M	3.1	18%
cage12	130K	2M	15.2M	34.6M	2.3	56%
hood	221K	10.8M	34.2M	562M	16.4	29%
m133-b3	200K	0.8M	3.2M	3.2M	1.0	6%
majorbasis	160K	1.8M	8.2M	19.2M	2.3	6%
mario002	390K	2.1M	6.4M	12.8M	2.0	30%
offshore	260K	4.2M	23.4M	71.3M	3.1	18%
patents_main	241K	0.6M	2.3M	2.6M	1.1	195%
poisson3Da	14K	0.4M	3M	11.8M	4.0	53%

$$C = \begin{bmatrix} bt & aw + by & bz \\ 0 & cx & 0 \\ 0 & dw + ex & 0 \end{bmatrix} \quad (4.1)$$

One way to determine which intermediate products to sum is to first generate all the intermediate products v_{ijk} , then sort them lexicographically by i and k . This places all the v_{ijk} with the same i and k values consecutively, after which they can

be summed to give C_{ik} . The problem with this method is that the number of intermediate products is often much larger than the number of nonzero values in C . For example, column 4 in Table 4.1 shows the number of nonzero values in the matrix $C = A^2$, denoted $nnz(C)$, for different matrices A , and column 5 shows the number of intermediate products in C , denoted $np(C)$. Column 6 gives the *compression ratio*, i.e. the quotient of these two quantities. The compression ratio can be as high as 28, indicating that the sorting based method needs to sort 28 times as data as the final output, leading to substantially reduced performance.

Another way to combine the v_{ijk} values is to hash them based on the key (i, k) . This places all intermediate products belonging to the same C value in the same hash table location, as long as we take care to deal with hash collisions. The problem with this method is that if we want to store all the nonzeros of C in the hash table, then the only storage large enough is the global memory of the GPU. However, accessing the hash table then requires reading and writing random global memory locations, resulting in huge numbers of uncoalesced memory accesses and low performance. Thus, in order for hashing to be effective, we must avoid storing the hash table in global memory.

4.1 Main Techniques

In this section, we present our SpGEMM algorithm and introduce several new techniques to address the problems described in Section 4.0.1. We first give an overview of the techniques and describe how they fit together, before discussing them in more detail in the following subsections.

Balanced work lists We first describe how to achieve perfect load balancing when processing the intermediate products. The basic idea is to generate all the indices in A and B that are accessed when computing the intermediate products, and to store the indices as lists in global memory. Subsequently, the lists are read back by the threads and evenly divided among themselves, allowing them to generate the intermediate products and insert them into hash tables in a completely balanced way. Generating the work lists incurs a relatively small cost, and is more than made up for by the substantial speedup obtained in the expensive intermediate products processing step.

As an example, consider the matrices given in (4.2). The intermediate products, in the order computed by Algorithm 1 are aw, bt, by, bz, cx, dw and ex . Since a and w are the first elements in the CSR representations of A and B , the indices accessed for the first intermediate product are 0 and 0. Similarly, the indices for the second intermediate product bt are 1 and 2. Altogether, the list of indices for computing all 7 intermediate products are 0, 1, 1, 1, 2, 3, 4 for A , and 0, 2, 3, 4, 1, 0, 1 for B . We will give two methods for efficiently generating work lists in Section 4.1.3.

$$A = \begin{bmatrix} a & 0 & b \\ 0 & c & 0 \\ d & e & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & w & 0 \\ 0 & x & 0 \\ t & y & z \end{bmatrix} \quad (4.2)$$

Shared memory hash tables In order to add together the intermediate products belonging to the same C value, we hash each v_{ijk} using the key (i, k) to a location in a hash table. Hash table is stored in shared memory so that intermediate products produced by different threads can be accumulated into final nonzeros. All v_{ijk} hashing to the same location and having key (i, k) are summed. Instead of using one large global memory hash table to hold all the C values, we create a number of small shared memory hash tables, one on each SM. Shared memory is much faster than global memory and allows more flexible access patterns, and hence our hash tables perform much better than global memory ones.

Since a GPU has a few dozen SMs and the shared memory on each SM is only a few dozen kilobytes, the hash tables can only hold a small fraction of C 's values at a time. To process all the values, we divide matrix A into groups of consecutive rows called *partitions*. Recall that in Algorithm 1, iterating through one row of A in line 3 produces one row in the output C . Thus, by inserting all the intermediate products from a partition of A into a hash table, we generate all the nonzero C values in the corresponding rows. After this, we flush the contents of the hash table to global memory, clear out the hash table, and repeat the previous process with the next partition of A . After processing all the partitions, all of C 's values will be stored in global memory. The C values will generally not be in CSR or COO order at this point, since they have been randomly permuted by the hash functions. Thus, we do

a final sort on the C indices to put the values in order. Note that we only need to sort $nnz(C)$ values, instead of all the intermediate products.

Size of partitions To achieve high performance, it is important to choose the right size for the partitions. We want to make partitions large in order to maximize the amount of work available to a thread block and reduce thread idling. However, if a partition is too large, it will generate too many intermediate products and the associated C values will not fit into the hash table. In that case, the extra products need to be stored in a spillover queue to insert into a later hash table, leading to wasted work.

To determine how many intermediate products can fit into a hash table, let H be the table size, and let λ be the compression ratio of C . We describe later how to compute λ . By definition, on average λ intermediate products are summed to produce each nonzero value of C . Therefore, we expect heuristically that inserting λH intermediate products into a hash table will produce H distinct nonzero C values, which is exactly the number that fits into the table. We describe in Section 4.1.2 how we can size the partitions so they produce approximately λH intermediate products.

Our heuristic is based on several assumptions, for example, that the number of intermediate products in different rows of A is roughly the same, and that hash collisions are infrequent, so we can store approximately H different C values in each hash table. While these assumptions do not always hold, we found that the heuristic still worked very well in practice. We show in Section 4.2.2 that it almost always selects a nearly optimal partition size, and that doing this leads to a large performance improvement over using suboptimal partition sizes. Furthermore, we show in Section 4.2.3 that implementing the heuristic incurs relatively low overhead.

Estimating compression ratio To choose the partition size we need to know the compression ratio, which in turn requires knowing the total number of intermediate products, $np(C)$, and the number of nonzeros in C , $nnz(C)$. We can compute $np(C)$ quickly using parallel reduction type operations. However, it is difficult to compute $nnz(C)$ exactly and efficiently. Instead, we use a sampling method based on techniques

from [69] to quickly estimate $nnz(C)$. This is sufficiently accurate to enable a nearly optimal selection of the partition size.

Outline of complete algorithm Putting the previous parts together, our algorithm consists of the following main steps. We indicate in parentheses the section in which each step is described in more detail.

- (i) Compute the number of intermediate products produced by each row of A and in all of A . Also estimate the number of nonzeros in C . (Section 4.1.1)
- (ii) Use the previous estimate to divide A into partitions containing λH intermediate products each. Assign one partition to each thread block. (Section 4.1.2)
- (iii) Each thread block generates work lists of A and B indices to produce the intermediate products in the partition. The work lists are written to global memory. (Section 4.1.3)
- (iv) Each thread block uses the work lists for a partition to generate intermediate products and sum them based on their hashed locations. Intermediate products causing hash collisions are added to a queue to be inserted into the hash table during the next round. After each round, the hash table contents, consisting of indices (i, k) and the corresponding C_{ik} values are written to global memory, and the hash table is cleared. (Section 4.1.4)
- (v) After all the partitions have been processed, the contents of all the hash tables stored in global memory are sorted lexicographically by the indices i and k . The result is the output in COO format of $C = AB$. (Section 4.1.5)

4.1.1 Counting intermediate products and estimating number of nonzeros in C

4.1.1.1 Counting intermediate products

We want to count the total number intermediate products generated by all rows in A (or equivalently, C), as well as the number of intermediate products generated by each row of A . The former is used to estimate the compression ratio, and the latter is used to partition A .

To compute these values, we iterate through A in the order given in Algorithm 1. Each nonzero value A_{ij} needs to be multiplied by every nonzero value in B_{j*} , the j 'th row of B . Recall that $ptr_B[j]$ gives the index of the first nonzero in B_{j*} in B 's CSR representation. Thus, each A_{ij} generates $ptr_B[j+1] - ptr_B[j]$ intermediate products. As we iterate through A , we write these counts to global memory. After processing each row of A , we perform a reduction operation on the counts in global memory to compute the number of intermediate products for the row. We write all the row counts to global memory in an array *count*. Finally, after processing all the rows, we do a reduction on *count* to get the total number of intermediate products.

4.1.1.2 Estimating nonzeros

To estimate the number of nonzeros in the output matrix C , we use techniques proposed in [69]. We first form a submatrix A' of A by taking a random p fraction of the rows of A , for a $p \in (0, 1)$ defined later. Similarly, we form a submatrix B' of B by taking a random p fraction of B 's columns. We then estimate the number of nonzero values in $A'B'$. To do this, imagine forming all the intermediate products in $A'B'$. For each product v_{ijk} , hash the indices (i, k) to a random value in $[0, 1]$. Let v be the k 'th largest value formed this way, for some k defined later. Then as shown in [69], we can estimate the number of nonzeros in $A'B'$ as $\frac{k}{v}$, and also estimate the number of nonzeros in $C = AB$ as $\frac{k}{vp^2}$, with certain bounds on the error probability.

We now describe some details of the implementation. We obtained good results, both in terms of efficiency and accuracy, by choosing $k = 100$ and $p = \frac{1}{10}$. Also, instead of choosing random submatrices, we retained every tenth row and column of A and B , respectively. We assigned one warp of threads to process each row of A' . Instead of keeping track of the k 'th smallest value among all the hashed intermediate products, we first have each warp keep track of the k smallest hash values among its own intermediate products. To do this, each warp maintains a sorted array F in shared memory with its k smallest hash values, and it sets a value *threshold* to be the largest value in F . Since inserting values into F is expensive, the warp also maintains a stack acting as a filter to prevent unnecessary accesses to F . Values are only inserted

into the stack if they are less than *threshold*. Stack manipulations are done in a data parallel manner using methods similar to those in [70]. Whenever the stack has at least 32 values in it, the warp will remove the top 32 items, sort them using bitonic sort, then insert them into F and update *threshold*. The insertion is done using the merge path algorithm [14], a fast GPU algorithm for merging sorted lists. Accessing F in this way allows us to do bulk insertions, which minimizes thread divergence and is more efficient than doing many individual insertions.

After each warp has computed its k smallest hash values, we copy the values back to the CPU host where they are sequentially merged to get the globally k 'th smallest value. We use this to estimate $nnz(C)$ and compute the compression ratio. We also use the estimate to allocate space on the GPU to store the output C .

4.1.2 Matrix partitioning

To break A into partitions with at most λH intermediate products each, we iterate through the *count* array computed in Section 4.1.1. *count*[i] gives the number of intermediate products in the i 'th row of A . We maintain a running sum of how many intermediate products are in the current partition, and start a new partition whenever the value exceeds λH . Suppose A has m rows. Then the output of this process is a sequence of numbers $n_0, n_1, \dots, n_k$ representing k partitions, with $n_0 = 0$ and $n_k = m$. The i 'th partition lies between rows n_{i-1} to $n_i - 1$. The partitioning step takes $O(m)$ sequential time. Since m is negligible compared to other problem parameters such as $np(C)$ or $nnz(C)$, we perform this step on the CPU.

Having determined the partitions of A , we also convert the partitions to COO format and store them in GPU global memory. These COO-formatted partitions are used in Section 4.1.4. During an execution, each thread block works on one partition at a time, and stores the COO format of that partition in global memory. After the thread block is done with the partition, it can discard it and reuse the memory for the COO format of the next partition. This minimizes the space overhead of our algorithm, as we discuss in Section 4.2.1.3.

4.1.3 Generating work lists

We will describe two algorithms for generating work lists. The first algorithm is based on parallel prefix sum, and is itself load balanced. The second algorithm assigns different groups of threads to process each row, and its load may be imbalanced depending on the structure of the matrix. However, we found that in practice the second algorithm performs a smaller number of memory accesses and has better performance, and so we use the second algorithm in our implementation. In both algorithms, we assume a thread block is provided with a partition of A which starts at row n_1 and ends immediately before row n_2 .

We first describe at a high level how the work lists are formed. Following Algorithm 1, we iterate through the nonzero values in A row by row. For each nonzero A_{ij} , we multiply it by all the nonzero values in the j 'th row of B . To do this using the CSR representations of A and B , we iterate through arrays col_A and val_A . The t 'th nonzero A value $val_A[t]$ lies in $col_A[t]$ of A . It should be multiplied by all the nonzero values in row $col_A[t]$ of B , which are located between indices $ptr_B[col_A[t]]$ and $ptr_B[col_A[t] + 1] - 1$ of val_B . Thus, the indices of A and B accessed to generate these intermediate products are $(t, ptr_B[col_A[t]]), \dots, (t, ptr_B[col_A[t] + 1] - 1)$, and these are added to our work list. As an example, consider matrices A and B from (4.2). The CSR representations of the matrices are

$$\begin{array}{lll} ptr_A = [0, 2, 3, 5] & col_A = [0, 2, 1, 0, 1] & val_A = [a, b, c, d, e] \\ ptr_B = [0, 1, 2, 5] & col_B = [1, 1, 0, 1, 2] & val_B = [w, x, t, y, z] \end{array}$$

For $t = 0$, value $val_A[0]$ is multiplied by the B values between indices $ptr_B[col_A[0]] = 0$ to $ptr_B[col_A[0] + 1] - 1 = 0$. This generates an item $(0, 0)$ in the work list. For $t = 1$, $val_A[1]$ is multiplied by B values between indices $ptr_B[col_A[1]] = 2$ to $ptr_B[col_A[1] + 1] - 1 = 4$, generating $(1, 2)$, $(1, 3)$ and $(1, 4)$ in the work list. Proceeding this way, the entire sequence of indices in the work list is

$$(0, 0), (1, 2), (1, 3), (1, 4), (2, 1), (3, 0), (4, 1).$$

Procedure 2 Procedure for a thread block to generate work lists $work_A$ and $work_B$.

```

1:  $s_1 \leftarrow ptr_A(n_2) - ptr_A(n_1)$ 
2: parfor  $i \leftarrow 0, s_1 - 1$  do
3:    $X[i] \leftarrow ptr_B[col_A[i] + 1] - ptr_B[col_A[i]]$ 
4:    $Y[i] \leftarrow ptr_B[col_A[i]]$ 
5: end parfor
6:  $S \leftarrow \text{PREFIXSUM}(X)$ 
7:  $s_2 \leftarrow S[s_1 - 1]$ 
8: parfor  $i \leftarrow 0, s_2 - 1$  do
9:    $\mathcal{A}[i] \leftarrow 0$ 
10:   $\mathcal{B}[i] \leftarrow 1$ 
11: end parfor
12: parfor  $i \leftarrow 0, s_1 - 2$  do
13:   $\mathcal{A}[S[i]] \leftarrow 1$ 
14:   $\mathcal{B}[S[i]] \leftarrow Y[i + 1] - Y[i] + 1 - X[i]$ 
15: end parfor
16:  $\mathcal{A}[0] = 0$ 
17:  $\mathcal{B}[0] \leftarrow Y[0]$ 
18:  $work_A \leftarrow \text{PREFIXSUM}(\mathcal{A})$ 
19:  $work_B \leftarrow \text{PREFIXSUM}(\mathcal{B})$ 

```

It is more convenient to store the A and B indices separately, and so we separate the tuples above into two lists, $[0, 1, 1, 1, 2, 3, 4]$ for A and $[0, 2, 3, 4, 1, 0, 1]$ for B . Next, we describe two methods to efficiently produce these lists.

4.1.3.1 Work lists using prefix sums

The first algorithm is given in Algorithm 2. We write the code in terms of parallel for operations for ease of exposition, but it is straightforward to translate it to efficient CUDA code. The arrays $X, Y, \mathcal{A}$ and $\mathcal{B}$ are stored in global memory. The code uses parallel prefix sum operations, which have very efficient GPU implementations [71].

Array X gives the number of B indices corresponding to each A index. For example, using A and B above, index 0 in A has one corresponding index 0 in B , so $X[0] = 1$. Index 1 in A has three corresponding indices, 2, 3 and 4 in B , and so $X[1] = 3$. In total, we have $X = [1, 3, 1, 1, 1]$. Array Y gives the starting B index corresponding to each A index. For the above example, we have $Y[0] = 0$ because the B indices corresponding to the first A index 0 start at 0, $Y[1] = 2$ because the B indices for the second (unique) A index 1 start at 2, etc. Altogether, $Y = [0, 2, 1, 0, 1]$. Since the partition we are working on lies between rows n_1 and n_2 of A , s_1 is the number of

unique A indices occurring in A 's work list. s_2 is the total number of (not necessarily unique) B indices. This equals the number of intermediate products in the partition, and thus also equals the length of the work lists for A and B for this partition. For the example above, we have $S = [1, 4, 5, 6, 7]$ and $s_2 = 7$. Arrays $\mathcal{A}$ and $\mathcal{B}$ have size s_2 and are used to construct A and B 's work lists. In the example, we can compute after line 17 of Algorithm 2 that $\mathcal{A} = [0, 1, 0, 0, 1, 1, 1]$ and $\mathcal{B} = [0, 2, 1, 1, -3, -1, 1]$. After the prefix sums in lines 18 and 19, we produce $work_A = [0, 1, 1, 1, 2, 3, 4]$ and $work_B = [0, 2, 3, 4, 1, 0, 1]$ as the work lists for A and B , as desired.

Theorem 4.1 *The k th intermediate product is the product between the nonzero at index $work_A[k]$ of A and the nonzero at index $work_B[k]$ of B : $val_A[work_A[k]] \times val_B[work_B[k]]$.*

Proof:

Consider first $work_A$. As described above, X and Y give the number of B indices and the starting B index, resp., corresponding to each A index. In other words, $X[i]$ represents number of intermediate products produced by $A[i]$. Since S is the prefix sum of X , we have $S[i] = \sum_{k=0}^i X[k]$ is the number of intermediate products produced by $A[0..i]$. This also means $A[i+1]$ processes intermediate products starting from index $S[i]$. Thus, the k 'th intermediate product is produced by i 'th nonzero of A s.t.:

$$i = \min\{t \mid k < S[t]\} \quad (4.3)$$

To find the A index corresponding to k 'th intermediate product, in Procedure 2 we use array $\mathcal{A}$. After initializing $\mathcal{A}$ (line 9 and 13), we have:

$$\mathcal{A}[t] = \begin{cases} 1 & \text{if } \exists i : t = S[i] \\ 0 & \text{otherwise} \end{cases}$$

Then, $work_A$ is the prefix sum of $\mathcal{A}$. We have $work_A[k] = \sum_{t=0}^k \mathcal{A}[t] = \sum_{t:S[t] \leq k} 1 = (\text{number of } S[t] \text{ values not larger than } k)$. Thus, we have that, $i = work_A[k]$ satisfies (4.3).

Next, consider $work_B$. After initializing $\mathcal{B}$ (line 10 and 14), we have:

$$\mathcal{B}[t] = \begin{cases} Y[i+1] - Y[i] + 1 - X[i] & \text{if } \exists i : t = S[i] \\ 1 & \text{otherwise} \end{cases}$$

Then, we compute the prefix sum of $\mathcal{B}$; $work_B = \text{PREFIXSUM}(\mathcal{B})$. Consider the k 'th value of $work_B$:

$$\begin{aligned} work_B[k] &= \sum_{i=0}^{S[i] \leq k} (Y[i+1] - Y[i] + 1 - X[i]) + \sum_{i=1}^{(i \leq k) \wedge (\forall t: i \neq S[t])} 1 \\ &= \sum_{i=0}^{S[i] \leq k} (Y[i+1] - Y[i]) + \sum_{i=1}^k 1 - \sum_{i=0}^{S[i] \leq k} X[i] \\ &= (Y[l] - Y[0]) + k - S[l-1] \\ &= Y[l] + k - S[l-1] \end{aligned}$$

Here, we set $l = \min\{t \mid k < S[t]\}$.

According to Eq. (4.3), l is the index of the A value, $A[l]$, that produces k 'th intermediate product. The corresponding B index is $Y[l] + k - S[l-1]$ since the B indexes to which $A[l]$ should be multiplied start at $Y[l]$, and we should advance $k - S[l-1]$ indices from $Y[l]$, since $S[l-1]$ intermediate values have been already produced using $A[0], \dots, A[l-1]$. In other words, $Y[l] + k - S[l-1]$ gives the B index for the k th intermediate product. $\square$

Since the prefix sum and reduce operations do linear work, Algorithm 2 does a total of $O(s_2)$ operations. All the operations can be done in a load balanced way using the GPU implementations in [71]. Also, most of the memory accesses in Algorithm 2 are coalesced, except for the $O(s_1)$ accesses in the first and third parallel for loops. Since s_1 is typically much less than s_2 , the impact of these noncoalesced accesses is usually small. Lastly, we note that we can also generate the work lists using a similar though conceptually somewhat simpler way using segmented scans [72]. However, we found that in practice, the algorithm above using normal scans (i.e. prefix sums) performs somewhat better.

Procedure 3 Procedure for W warps to generate work lists for the m intermediate products in row r .

```
1:  $i_A \leftarrow ptr_A[r]$ 
2:  $lane \leftarrow \text{threadId} \pmod{32}$ 
3:  $w = \text{warpId} \pmod{W}$ 
4:  $lo = w \lceil \frac{m}{W} \rceil$ ;  $hi = lo + \lceil \frac{m}{W} \rceil$ 
5:  $seg \leftarrow \text{GETINDEX}(lo + lane, S)$ 
6:  $prev \leftarrow S[seg]$ ;  $next \leftarrow S[seg + 1]$ 
7: for  $i \leftarrow lo + lane, hi$  stride 32 do
8:   while  $next \leq i$  do
9:      $seg \leftarrow seg + 1$ 
10:     $prev \leftarrow next$ 
11:     $next \leftarrow S[seg + 1]$ 
12:   end while
13:    $i_B \leftarrow ptr_B[col_A[i_A + seg]]$ 
14:    $work_A[i] \leftarrow i_A + seg$ 
15:    $work_B[i] \leftarrow i_B + i - prev$ 
16:   if  $lane = 31$  then
17:     broadcast  $seg, prev, next$  to warp
18:   end if
19: end for
```

4.1.3.2 Work lists using search

We now describe a second method to generate work lists, which we found performs somewhat better in practice. The pseudocode is given in Algorithm 3. We assign W warps from a thread block to process each row r of a partition. We will describe how W is chosen later. Recall that in Section 4.1.1.1 we computed the number of intermediate products per row. Suppose row r has m intermediate products. Then warp w among the W warps will generate work lists for the lo 'th to the hi 'th intermediate products, for lo and hi defined in line 4. The thread in lane $lane$ of warp w is responsible for generating the A and B indices of every 32'nd product in this range, starting from the $lo + lane$ 'th product as in line 7.

We first define an array S for row r . Conceptually, this is the same as the S array in Algorithm 2. Let the nonzero values in row r of A lie in columns $c_1, \dots, c_k$. Then when we generate intermediate products according to Algorithm 1, the i 'th nonzero value is multiplied by all the nonzeros in row c_i of B ; there are $ptr_B[c_i + 1] - ptr_B[c_i]$

such nonzeros. We define

$$S[i] = \sum_{j=1}^i ptr_B[c_i + 1] - ptr_B[c_i].$$

Also, let $S[0] = 0$. Thus, $S[i]$ represents the total number of intermediate products generated by the nonzeros in row r , up to and including the i 'th nonzero. S can be efficiently computed using parallel prefix sum; we omit the code from Algorithm 3.

Now, we describe how a thread finds the A and B indices for an intermediate product it is responsible for. For example, using the A and B from (4.2), if a thread is processing the third intermediate product by in row 0 of A , the indices are 1 and 3 for A and B , respectively. We first describe how to determine the A index. A thread in lane $lane$ of w begins by processing the $lo + lane$ 'th intermediate product. In line 5 of the code, we call the `GETINDEX` function, which uses binary search to find the largest seg such that $S[seg] \leq lo + lane$. Then, since the intermediate product lies in row r of A , its A index is $ptr_A[r] + seg$.

For example, using A and B from (4.2), suppose $r = 0$ and $lo + lane = 2$. Thus, we are looking for the A index of the third intermediate product by in row 0 of A . The S array for the row is $[0, 1, 4]$. So since $S[1] \leq lo + lane \leq S[2]$, we have $seg = 1$. Then we get that the A index for value b is $ptr_A[0] + 1 = 1$, as desired.

Having computed the A index for the $lo + lane$ 'th intermediate product in r , we now show how to compute the A index for the i 'th product, for any i . This is done using a linear search in lines 8 to 12 of the code. In particular, since each thread processes every 32'nd intermediate product, let $i' = i - 32$ be the previous product processed by the thread, and suppose seg satisfies $S[seg] \leq i' < S[seg + 1]$. We want to update seg so that it satisfies $S[seg] \leq i < S[seg + 1]$. To do this, we set $prev = S[seg]$ and $next = S[seg + 1]$ as in line 6, so that $prev \leq i' < next$. Then as long as $i > next$, we increment seg and update $prev$ and $next$ as in lines 10 and 11. When the while loop on line 8 terminates, we have the desired new value for seg . Then the A index for the i 'th intermediate product is $i_A + seg$.

Next, we describe how to compute the B index for the i 'th intermediate product. Suppose the product is ab , for $a \in A$ and $b \in B$. Since the A index for a is $i_A + seg$,

then b lies in row $col_A[i_A + seg]$ of B , which starts at index $i_B = ptr_B[col_A[i_A + seg]]$ of B , as shown in line 13 of the code. Furthermore, there are $prev$ intermediate products in row r with A indices less than $i_A + seg$, and so ab is the $i - prev$ 'th intermediate product with A index $i_A + seg$. Thus, the B index for b is $i_B + i - prev$. In lines 14 and 15, the A and B indices for the i 'th intermediate product are stored at index i in the work lists $work_A$ and $work_B$.

Next, we explain the purpose of lines 16 and 17. Here, the last thread in the warp uses CUDA's intrinsic `__shfl` function to broadcast the value of $seg, prev$ and $next$ to all the other threads in the warp. This is done because the values of $seg, prev$ and $next$ for any thread in the warp on the next iteration of the main for loop are at least as large as $seg, prev$ and $next$ for the lane 31 thread in the current iteration. Thus, by broadcasting $seg, prev$ and $next$, we may reduce the amount of linear search threads do in the next iteration to determine these values.

Lastly, we describe how we choose W , the number of warps to assign to each row of the partition. We set $W = \lceil \frac{U}{R} \rceil$, where U is the number of warps in the thread block and R is the number of rows in the partition. If $R \geq U$, then we assign one warp per row in a round robin fashion. In addition, if a row has a very few intermediate products, we only want to assign a part of a warp to process the row to avoid thread idling. In particular, if m is the number of intermediate products in the row and i satisfies $2^{i-1} \leq m < 2^i$, then we assign 2^{i-1} threads from a warp to process the row. This is done using the virtual warp techniques from [73]. If we assign $w < 32$ threads to process a row, we will change the value 32 in lines 2 and 7 of Algorithm 3 to w , and change the value 31 in line 16 to $w - 1$. We note that the `__shfl` function allows multiple threads within a warp to each broadcast to their own virtual warp.

4.1.4 Summing intermediate products

We now describe how intermediate products are summed together to produce C values using shared memory hash tables. Each thread block sums the intermediate products from one partition of A . The input to the thread block are the work lists $work_A$ and $work_B$ for the partition computed in the previous step. The threads in the block

Procedure 4 Procedure for a thread to insert intermediate product v_{ijk} into the hash table.

```

1:  $h \leftarrow \text{HASH}(i, k) \pmod{H}$ 
2:  $k_1 \leftarrow \text{COMPACT}(i, k)$ 
3:  $k_2 \leftarrow \text{keys}[h]$ 
4: if ( $k_1 = k_2$ ) then
5:    $\text{atomicAdd}(\text{vals}[h], v_{ijk})$ 
6: else if ( $k_2 \neq \perp$ ) then
7:   add  $v_{ijk}$  to spillover queue
8: else
9:    $k_0 \leftarrow \text{atomicCAS}(\text{keys}[h], \perp, k_1)$ 
10:  if ( $k_0 = \perp$ )  $\vee$  ( $k_0 = k_1$ ) then
11:     $\text{atomicAdd}(\text{vals}[h], v_{ijk})$ 
12:  else
13:    add  $v_{ijk}$  to spillover queue
14:  end if
15: end if

```

divide the lists in round robin fashion among themselves. That is, if there are T threads in the block, then thread 0 will process items $0, T, 2T, \dots$ in the lists, thread 1 processes items $1, T + 1, 2T + 1, \dots$, etc. In this way, each thread gets (nearly) the same number of items, and the work of processing the intermediate products is almost completely balanced.

The thread block maintains a hash table in shared memory, implemented as two arrays *keys* and *vals*, each of size H . We assume all values in *keys* are initialized to $\perp$, and all values in *vals* are initialized to 0. We chose $H = 2000$, so that each hash table uses 16 KB of shared memory. Each SM has 48 KB of shared memory and runs two thread blocks. Thus, we use 32 KB of shared memory on each SM for the hash tables, and set aside 16 KB for other tasks.

We now discuss the processing of a particular item from the work lists. This step requires us to use the COO representation of the partition of A being processed, which was generated in Section 4.1.2. Now, suppose a thread is processing the t 'th item from the work lists. Let $i_A = \text{work}_A[t]$ and $i_B = \text{work}_B[t]$; these are indices into A and B 's *row*, *col* and *val* arrays. Let $v_A = \text{val}_A[i_A]$ and $v_B = \text{val}_B[i_B]$ be the A and B values corresponding to the t 'th work item. Using A 's COO representation, we can determine that v_A lies in $i = \text{row}_A[i_A]$ and column $j = \text{col}_A[i_A]$ of A^1 . Also, v_B lies

¹Note that if A were only stored in CSR format, we can not directly determine the row index of

in row j and column $k = \text{col}_B[i_B]$ of B . The intermediate product $v_A v_B$ therefore belongs to value C_{ik} of C .

The pseudocode for inserting v_{ijk} into the hash table is given in Algorithm 4. We first hash the tuple (i, k) to a location h in the table. Also, we compact the indices i and k , normally represented as two 32-bit numbers, into a single 32-bit value k_1 using the function COMPACT. This is to speed up the sorting step in Section 4.1.5, and we explain how it is done in Section 4.1.4.1. Now, let k_2 be the 32-bit key currently stored in location h of the *keys* array. There are three possibilities. If $k_1 = k_2$, this means some thread has earlier inserted an intermediate product $v_{ij'k}$, for $j' \neq j$, and so C_{ik} already has a non-null value. In this case, we add v_{ijk} to the current value of C_{ik} , stored in *vals*[h]. We use CUDA's intrinsic atomic add function to avoid race conditions. If $k_1 \neq k_2$ and $k_2 \neq \perp$, then this means that some thread has processed an intermediate product $v_{i'j'k'}$ for $i' \neq i$ or $k' \neq k$, and (i', k') also hashes to location h . In this case, we have a hash collision, and v_{ijk} cannot be inserted into the hash table. In this case we add v_{ijk} to a spillover queue. We describe how the queue is dealt with in the next paragraph. In the third case, $k_2 = \perp$. This means that v_{ijk} is the first intermediate product belonging to C_{ik} to be inserted. We set C_{ik} to v_{ijk} , using the atomic CAS and add operations to avoid race conditions. After the thread block has finished trying to insert all the items from the work lists, it will flush the contents of the hash table to global memory, by writing out all tuples $(\text{keys}[i], \text{vals}[i])$ for which $\text{keys}[i] \neq \perp$. This can be done efficiently using a standard parallel compaction algorithm. Then the hash table is cleared, and can be reused.

Next, the thread block will process all the uninserted items in the spillover queue. It treats these items in the same way as items from the work lists. That is, it applies Algorithm 4 to each item to insert them into the hash table. This may again lead to uninserted items, which are added to another spillover queue². After all the items in the first queue have been processed, the non-null entries in the hash table are flushed to global memory, the hash table is cleared, and the thread block processes the new

the i_A 'th nonzero in A .

²The storage space for different queues can be recycled.

(a) Matrices with high throughput

(b) Matrices with low throughput

Figure 4.1: Throughput comparison for CUSP, CUSPARSE, BHSPARSE and our BalancedHash algorithm.

spillover queue. It keeps doing this until all items from the original work lists have been inserted into the hash table. In practice, we found that we could usually insert all the items after just two passes (i.e. items only spilled over once). In addition, the second pass is faster as it needs to insert fewer items.

4.1.4.1 Optimizations

We now describe several optimizations we used to further speed up the processing of the intermediate products. First, we used data prefetching. As a thread iterates through the work list, it loads the A and B data it will use on the *next* iteration. This decreases the amount of time a thread spends waiting for data, and improved

performance by 20-40%. Next, we managed the spillover queue in a novel way. Instead of performing queue operations using warp-sized prefix sums, as done in for example [71], we used a variable *pred* for each thread indicating whether its last hash table insertion was successful. We then used CUDA’s intrinsic `__ballot` and `__popc` functions and bitmasking to detect whether any thread failed its insertion, and if so, to compute the queue location to store the spilled item.

Finally, we compacted the i and k indices for an intermediate product v_{ijk} to 32 bits, by observing that all the matrices in Table 4.1 had less than one million rows. This allowed us to represent the column index k using 20 bits. Also, we modified the row index to refer to a row within a partition instead of the entire matrix. This required only 12 bits for i , and thus 32 bits overall for i and k ; i occupies the most significant bits. In order to deal with matrices with more than 1 million columns, we can either continue to pack i and k into 32 bits but reduce the number of bits for i , at the expense of possibly using smaller partitions, or simply use 32 bits each for i and k . Using the former method, we can process matrices with up to 8 million rows, allowing partitions of 512 rows, with minimal performance degradation. The latter method allows processing arbitrary size matrices, but doubles the sorting time in the following sorting step.

4.1.5 Sorting the output

Lastly, we sort a block of hash table data stored in global memory after Section 4.1.4 to produce the corresponding partition of matrix C in COO format, then convert this to CSR format. The block consists of a *keys* and a *vals* array containing indices and values for C arranged in a random order due to hashing, and we want to sort them into COO order.

Each entry in *keys* contains the row and column indices of an element of C packed into 32 bits, with the row index as the most significant bits. We sort *keys* and *vals* in increasing order of *keys* values, which places the C elements in nondecreasing order of rows and then columns. We use radix sort [74]. Since the row and column indices are packed into 32 bits, radix sort runs twice as fast as if we used separate 32-bit values

for each index. After a block is sorted, we expand each *keys* entry into separate 32-bit row and column indices. Then we add r to each row index, where r is the first row in the partition corresponding to the block. This produces the C values for the partition stored in COO format. We then convert this to CSR format for our final output. We also note that since different hash table blocks correspond to disjoint partitions, each block can be sorted independently by a thread block without any synchronization between the thread blocks.

4.2 Experimental Results

In this section, we evaluate the performance of our algorithm and examine the effectiveness of our optimizations. We tested our algorithm using 21 matrices from the University of Florida sparse matrix collection [68]. For each matrix A in the test set, we computed $C = A^2$. The matrices and their properties are listed in Table 4.1. The experiments were performed on an Nvidia Tesla K20Xm GPU containing 2688 cores in 14 SMs. The peak single precision performance of the device is 3935 GFlops/s and the memory bandwidth is 250 GB/s. Each SM has 48 KB of shared memory. The version of CUDA is 7.0.

4.2.1 Throughput comparison

We first compare our algorithm’s throughput with those of CUSP (v0.4) [35], CUSPARSE (in CUDA 7.0) [34] and BHSPARSE [1]. We define the throughput for computing A^2 as two times the number of intermediate products in A^2 divided by the running time. The numerator is approximately equal to the number of arithmetic operations in the SpGEMM computation, since each intermediate product requires one multiplication and one addition operation.

The results of the comparison are shown in Figure 3.3. Our algorithm is labeled “BalancedHash”. The figure shows that we perform significantly better than CUSP and CUSPARSE. CUSP has very steady performance, but its throughput is low because it sorts all the intermediate products. CUSPARSE is faster than CUSP on large matrices but often slower on small ones. BalancedHash is $5.6\times$ faster than CUSP on

average and $11.8\times$ faster in the best case. It is also $2.4\times$ faster than CUSPARSE on average, and $9.5\times$ faster in the best case.

BalancedHash is also substantially faster than BHSPARSE, which is currently the fastest GPU SpGEMM algorithm. Our algorithm is $2.5\times$ faster than BHSPARSE on the *patents_main* matrix. Also, on the matrices where BHSPARSE achieves the highest performance, including *ship*, *putk*, *protein* and *hood*, our algorithm performs $1.36\times$, $1.56\times$, $1.92\times$ and $1.19\times$ faster, respectively. On average, BalancedHash is $1.5\times$ faster than BHSPARSE.

The only matrices on which BalancedHash is slower than BHSPARSE are *m133-b3* (slower by 15.3%) and *majorbasis* (slower by 12.2%). These matrices have low compression ratios and work imbalance. For example, *m133-b3* has a compression ratio of 1.0 and a work imbalance of 6%. Thus, they represent nearly the worst case input for our algorithm. We note that we can improve performance on such matrices using the following method. We first quickly estimate the work imbalance of a matrix using the number of intermediate products per row. Then, for matrices that are balanced, we can omit the worklist generation stage in Section 4.1.3 and proceed directly to the hashing and summation stage in Section 4.1.4. The performance breakdowns for *m133-b3* and *majorbasis* in Section 4.2.3 show that omitting worklist generation reduces the total running time by 15-20%, and should cause minimal slowdown in the summation stage because the matrices are already balanced.

4.2.1.1 Load balancing

One of the main goals of our algorithm was to improve load balancing. We now look at the effectiveness of our load balancing techniques. We focus on matrices with large imbalances, as shown in Table 4.1. Recall that the imbalance of a matrix is defined as the coefficient of variation (COV) of the number of intermediate products per row. The most imbalanced matrix in our dataset was *webbase-1M*, with a COV of 680%. It represents a web matrix, and the number of nonzeros per row follows a power law distribution. Two other matrices with large imbalances are *scircuit* and *patents_main*, with COVs of 136% and 195%, respectively. On these matrices, BalancedHash was

$2.2\times$, $1.8\times$ and $1.48\times$ faster, respectively, than BHSPARSE. In addition, since all three matrices had low compression ratios, we can infer that the speedups achieved came largely as a result of improved load balancing.

We also measured load balancing directly by looking at the level of multiprocessor activity, i.e. the percent of time when at least one warp was active on each SM, using Nvidia’s CUDA profiler. When processing intermediate products, as in Section 4.1.4, the activity level was on average 99.2%, and the warp execution efficiency averaged 86.3%. During the other stages of the algorithm, the multiprocessor activity was typically between 70-90%.

4.2.1.2 Work efficiency

Another goal of our algorithm was to efficiently deal with matrices with high compression ratios, i.e. number of intermediate products divided by number of nonzeros. From Table 4.1, we see that matrices *protein*, *pwtk*, *spheres* and *cant* have among the highest compression ratios, at 28.3, 19.1, 17.5 and 15.5, respectively. On these matrices, BalancedHash performed $1.92\times$, $1.56\times$, $1.75\times$ and $1.78\times$ faster, respectively, than BHSPARSE. Furthermore, none of the matrices are highly imbalanced, indicating that the performance improvements are mostly due to the work and memory efficiency of BalancedHash’s shared memory hash tables.

4.2.1.3 Space and time overhead

Since our algorithm requires storing worklists and converting to and from COO format, we measured the space and time costs for these operations. Each thread block processes one partition of A at a time, and only needs to store the worklists and COO formats of A and C for that partition in global memory. When the thread blocks moves to the next partition, it can reuse the space for the previous partition. This limits the space overhead of our algorithm to on average only 1.96% of the space needed to store the input and output matrices (in CSR format). The highest space overhead was 24.1% on the matrix *protein*.

Figure 4.2: Relative performance of heuristically chosen partition size P versus best partition size.

The overall time overhead for converting between CSR and COO formats was on average 0.7% of the total running time. The highest overhead was 9.1% for the matrix *epidemiology*.

4.2.2 Effect of partition size

Next, we look at the effect of using different partition sizes on the performance of our algorithm. In Figure 4.2, the horizontal axis P gives the *partition size*, i.e. the number of intermediate products in a partition, and the vertical axis represents the normalized throughput of BalancedHash using different partition sizes, compared to its throughput using the optimal partition size. The highlighted point on each curve indicates the partition size chosen by the heuristic we described in Section 3.2, and the performance BalancedHash achieved using that partition size compared to its performance using the best possible partition size. For example, the point on the yellow line indicates that BalancedHash used a partition size of 12,000, and that this achieved 99% of the performance using the best partition size, which is about 10,500. We tested the effects of partition size for all the matrices in Table 4.1, though for visual clarity we only show the results for *economics*, *cant*, *ship* and *QCD*.

The first thing we see from Figure 4.2 is that the heuristic we use almost always chooses a nearly optimal partition size. On average, we achieved 98.7% of the per-

Figure 4.3: Break down of running time for each step of BalancedHash.

formance using the optimal partition size. BalancedHash’s worst performance was on *ship*, where it achieved 94.3% of the performance of the optimal. Another observation from the figure is that partition size has a large impact on BalancedHash’s performance. For example, looking at the plot for *economics*, we see that choosing an excessively large partition can reduce performance by over 55%. Finally, the partition size affects the number of intermediate products we insert into the hash tables and hence the hash collision rate. We measured this, and found that on average each intermediate product was hashed 1.3 times before it was successfully inserted. The highest collision rate occurred on the matrix *cage12*, where each item was hashed about 1.6 times.

4.2.3 Running time breakdown

Lastly, we look at the amount of time used by the different stages of BalancedHash. We also show that the load balancing and partition size optimizations we perform do not incur substantial overhead. Figure 4.3 gives the proportion of the total running time consumed by each stage. Stage 1 corresponds to the counting and estimation steps in Section 4.1.1, stage 2 corresponds to the generation of work lists in Section 4.1.3, stage 3 is the time to process intermediate products described in Section 4.1.4,

and stage 4 is the sorting step in Section 4.1.5. Note that we do not include the time to partition the A matrix as this was negligible.

We see that stages 1 and 2, which allow us to optimize the partition size and load balance, respectively, take only 12% and 19% of the overall running time on average. Stage 3 is the main computation step for SpGEMM, and takes on average 54% of the running time. Lastly, the sorting steps takes 15% of the total time on average. Since stages 1 and 2 allow a substantial reduction in the running time of the other stages, we see that their costs are more than compensated for by the performance improvements they bring.

4.3 Summary

In this Chapter, we proposed several novel techniques for improving load balancing and work efficiency in SpGEMM on GPUs. Our BalancedHash algorithm performs $1.5\times$ faster than the current fastest GPU SpGEMM algorithm BHSPARSE, and up to $2.5\times$ faster on highly unstructured matrices. In the future, we would like to further study the relationship between the structure of sparse matrices and the complexity of SpGEMM, in order both to improve the throughput of our algorithm and to make its performance more uniform on different matrices. In addition, we are interested in performing SpGEMM on larger matrices and on multiple GPUs. Finally, we would like to integrate our SpMM algorithm into a general GPU sparse matrix library, as well as study its use in real-world applications.

Chapter 5

Graph coloring

In the graph coloring problem, we assign colors to the nodes of a graph so that neighboring nodes have different colors, while trying to minimize the number of colors used. In practice, nodes often represent tasks and edges conflicts between tasks. Coloring the graph partitions the tasks into a small number of non-conflicting classes, each of which can be performed in parallel. Graph coloring has a number of applications in computer science and engineering, including processor and network scheduling [75] [76] [77], sparse linear algebra [78] [79] [80], matrix preconditioning [81] [82], optimization [83], community detection [84] [85] and computational geometry [86] [87].

Recent years have seen large growth in the size of data, and applications now frequently process graphs with millions of nodes or more. To color these large graphs efficiently, a number of parallel graph coloring algorithms have been developed for shared and distributed memory systems [46–49]. More recently, coloring algorithms have also been proposed for graphics processing units (GPUs). These include the algorithm by Cohen and Castonguay [29], which is implemented as the *csrcolor* function in Nvidia’s cuSPARSE library [28], and Deveci *et al*’s [30] recent *vertex-based* and *edge-based* coloring algorithms, which we refer to as VEB. GPUs are an attractive platform for coloring, both due to their high throughput and also because of their wide use in linear algebra and optimization, two of the main application areas for graph coloring. In this Chapter, we present two new GPU coloring algorithms, *counting-based Jones-Plassmann (CJP)* and *conflict coloring (CC)*. CJP is optimized for coloring quality, and is better for applications where minimizing color usage is

most important. CC is optimized for speed, and is more suitable for applications where coloring takes a large proportion of the overall running time, or where colorings need to be computed repeatedly.

CJP is based on the classic Jones-Plassmann parallel coloring algorithm [46], which operates in rounds and colors a maximal independent set of nodes in each round. We propose a work-efficient implementation of the algorithm, by associating each node with a counter value to quickly identify the round in which the node should be colored. This allows CJP to examine each edge in the graph only a constant number of times during an execution. To reduce color use, CJP colors each node using the minimum available color. We show how to perform this operation efficiently on the GPU. In contrast, [28] colors nodes using a multiple of the round number plus an offset, resulting in much higher color use.

CC further improves on the speed of CJP, at the expense of using somewhat more colors. Whereas CJP always ensures nodes have a valid coloring at any stage of execution, CC allows neighboring nodes to temporarily have the same color. These coloring conflicts are later detected and some of the conflicting nodes are recolored in the next round. While this results in somewhat more work than CJP, it also allows greater parallelism and better utilization of the GPU’s massive multithreading. In contrast, the parallelism in CJP is limited by the size of the independent set in each round. In CC, nodes are assigned random colors, up to a current estimate of the chromatic number (i.e. the minimum number of colors needed to color the graph). We show how nodes can independently update the estimate to quickly converge on an accurate value. Choosing a random color instead of the minimum one results in fewer conflicts and faster termination compared to other speculative coloring algorithms [47], [48], [49].

We compared CJP and CC to several other coloring algorithms, including the GPU-based algorithms `csrcolor` and `VEB`, and the CPU-based sequential coloring algorithm in `ColPack` [43]. Sequential algorithms generally use fewer colors than parallel ones, since they avoid certain race conditions and can effectively implement several color-reducing node ordering heuristics. Thus, the comparison to `ColPack` provides a

baseline to assess CJP and CC's coloring quality. We performed experiments using a variety of graphs from the University of Florida sparse matrix collection [68]. We found that CJP was on average $1.5\times$, $2\times$ and $1.9\times$ faster than csrcolor, VEB and ColPack resp., while CC was on average $4.3\times$, $3.3\times$ and $2.7\times$ faster, resp. In terms of coloring quality, ColPack almost always provided the best results. CJP and CC were both much better than csrcolor, and CJP was somewhat better than VEB while CC was somewhat worse. For nonbipartite graphs, CJP used on average $0.23\times$, $0.9\times$ and $1.3\times$ as many colors as csrcolor, VEB and ColPack resp., while CC used $0.28\times$, $1.1\times$ and $1.5\times$ as many colors resp. Lastly, we examined CJP and CC in terms of execution efficiency. We found that CC colors more nodes each round, and generally terminated in 5 to 7 rounds while CJP took 20 to 35 rounds. In addition, CC's conflict reduction mechanism was quite successful, and each node was processed on average only 1.3 times in CC.

5.1 Coloring Algorithms

In this section we present the CJP and CC coloring algorithms. We give the main ideas for the algorithms in Sections 5.1.1 and 5.1.2, then discuss how to efficiently implement them on GPUs in the next section.

5.1.1 Counting-based Jones-Plassmann algorithm

Our first algorithm *CJP* is an efficient implementation of the Jones-Plassmann technique. CJP associates a counter with each node to quickly determine the round in which a node should be colored. This allows the algorithm to be work efficient, and examine each edge only a constant number of times during an execution. Consider an undirected input graph $G = (V, E)$ with n nodes and m edges. The CJP algorithm performs the following steps.

- (i) Initially every node v is assigned a random value $val(v)$. Then v counts the number of neighboring nodes with higher values and stores the result in $count(v)$, i.e. $count(v) = |\{w \mid ((v, w) \in E) \wedge (val(w) > val(v))\}|$. Assume for simplicity

Figure 5.1: Illustration of CJP. The value in each node is its hash value. **(a)** Two local maxima are colored using the first color. **(b)** Three additional nodes are colored using the second color. Note that these nodes are neighbors of nodes colored in step (a) and they become local maxima in this step. **(c)** Nodes 2 and 3 are colored. While node 2 is colored by a new color, node 3 is colored by the first color, which is the smallest available color for it. **(d)** Node 1 is colored. As its neighbors are colored by first and third color, it uses the second color.

all nodes have different values; if ties occur they can be broken by node ID. In general, $count(v)$ represents the number of v 's neighbors that have higher values and that are still uncolored.

Our GPU algorithm computes $count(v)$ in parallel by assigning a virtual warp of W threads (described in 5.2.2) to v , as described in Section 5.2.2. Each thread processes every W 'th neighbor of v in round robin fashion, counting how many node values are higher than v 's among the nodes it processes. Then the per-thread counts are summed using reduction to obtain $count(v)$.

- (ii) After a node v computes $count(v)$, it enqueues itself into a queue Q_1 if $count(v) = 0$. Q_1 is stored in global memory, and contains the first set of nodes to be colored. We insert nodes in parallel into Q_1 using the prefix sum technique in [70].
- (iii) We then begin the main loop for the algorithm, which occurs in a series of rounds. Suppose we are currently in round $i \geq 1$. We will assign colors to the nodes in queue Q_i , which contains all the uncolored nodes that have higher values than all their uncolored neighbors. In addition, we will construct Q_{i+1} , the nodes to be colored in the next round.

Each node v in Q_i first finds the minimum color not used by any of its neighbors and sets its own $color(v)$ to this value. We describe an efficient GPU method to find the color in Section 5.2.3.

Next, v iterates through all its neighbors and decrements $count(w)$ for any neighbor w with $count(w) > 0$. This is done because $val(v) > val(w)$, and so after coloring v node w will have one less uncolored neighbor with higher value. After the decrement, v checks if $count(w) = 0$. If so, then w 's value is now higher than all those of its uncolored neighbors, so v enqueues w into Q_{i+1} . We describe how to efficiently enqueue w in Section 5.2.5.

- (iv) Round i ends after all nodes in Q_i have been processed in step 3. Then if Q_{i+1} is not empty, we proceed to round $i + 1$ and repeat step 3. Otherwise, Q_{i+1} is empty and all the nodes in G have been colored.

After finishing round i , Q_i is no longer needed. We can conserve memory by reusing Q_i 's space to store queue Q_{i+2} , so that at all times we only need space for two queues.

The steps of CJP are illustrated in Figure 5.1.

5.1.1.1 Analysis of the CJP algorithm

We now show that CJP computes a correct coloring. We also show it is work efficient, doing $O(m)$ amount of work, where m is the number of edges. To show correctness, we first prove that at any point after step 2 of the algorithm, the *count* value of any node is at least as large as the number of uncolored neighbors of the node that have higher values.

Lemma 5.1 *Consider an execution of the CJP algorithm and a point in time t after step 2 in the execution. For any node v , $count(v) \geq |\{w \mid ((v, w) \in E) \wedge (val(w) > val(v)) \wedge (w \text{ is uncolored at } t)\}|$.*

Proof: The proof is by induction. The lemma is true immediately after finishing step 2 of the execution. Assume it holds at time $t - 1$ (i.e. after $t - 1$ operations have been performed in the execution). Since a neighbor of v can never go from colored to uncolored, the quantity on the right hand side of the inequality in the lemma never

increases. So it suffices to check the lemma still holds if the operation at time t decreases $\text{count}(v)$.

Since $\text{count}(v)$ decreases, v is currently uncolored. In addition, there must be a neighbor w of v in the queue at time $t - 1$ that decremented $\text{count}(v)$. Since w is in the queue, it has $\text{count}(w) = 0$. Then by the inductive hypothesis, w does not have any uncolored neighbors with higher values at time $t - 1$. So since v is an uncolored neighbor of w , v 's value is less than w 's. Before decrementing $\text{count}[v]$, w colored itself, which decreased the number of uncolored neighbors of v with higher values. Thus, after w decrements $\text{count}(v)$, the lemma still holds.

Theorem 5.1 *The CJP algorithm produces a valid coloring.*

Proof: We show that at any time when two neighboring nodes v and w are colored, $\text{color}(v) \neq \text{color}(w)$. Suppose first that v and w are colored in different rounds, and WLOG v was colored first. Then when w chooses its color, it will read $\text{color}(v)$ and choose a different color. So since nodes never change their colors, we have $\text{color}(v) \neq \text{color}(w)$ at all future times. Next, suppose v and w are colored in the same round. Then they are both in the queue at some time t , and $\text{count}(v) = \text{count}(w) = 0$. Also, v and w are both uncolored at time t . Assume WLOG that $\text{val}(w) > \text{val}(v)$. We now have a contradiction, because Lemma 5.1 implies $\text{count}(v) \geq 1$, since w is an uncolored neighbor of v with higher value. Thus, v and w cannot be colored in the same round, and so the theorem holds. Next, we show the algorithm is work efficient.

Theorem 5.2 *Given a input graph with m edges, CJP does $O(m)$ amount of work.*

Proof: We first show how to associate every piece of work done in the algorithm with an edge in the graph. The first type of work in the algorithm is to decrease $\text{count}(v)$ for some node v . When this happens, we can associate it with an edge (v, w) , where w is a neighbor of v that was in the queue and caused the decrement. Also, for each edge (v, w) , this type of work occurs at most once, since w is only in the queue for one round. The second type of work in the algorithm is for a node v to find a minimum unused color among its neighbors. This involves examining every

Figure 5.2: Illustration of CC. The value in each node is its value. (a) Assume we start from two initial colors. Each node is assigned to one of these two colors randomly. These assignments result in conflicts. (b) Resolve conflicts: A node loses its color if it has any neighbor that has the same color and higher degree. (c) Uncolored nodes are assigned random colors unused by neighbors. We still have a conflict after this step. (d) Previous steps are repeated until there are no conflicts.

neighbor w of v , and so we can associate the examination of w with edge (v, w) . This work also only occurs once, since it happens when v is colored, which only happens once. The last type of work in the algorithm is to enqueue nodes. When a node v is enqueued, we can associate this with a neighbor w whose decrement of $\text{count}(v)$ caused v to be enqueued. The work only occurs once, since v is only enqueued once. Combining the above, we have that each piece of work in the algorithm is associated with some edge. Also, each edge is only associated with a constant number of pieces of work. Thus, the theorem follows.

5.1.2 Conflict coloring algorithm

We now describe a second algorithm called *conflict coloring*, or *CC*. Like CJP, the CC algorithm also runs in rounds. However, whereas CJP maintains a valid coloring at all points in an execution, CC allows neighboring nodes to temporarily have the same color during a round. At the end of a round CC performs a conflict detection procedure that restores a valid coloring.

CC is significantly faster than CJP, with a key reason being that it has more parallelism. The amount of work in each round of CJP is determined by the size of the maximal independent set formed by nodes with $\text{count} = 0$. For some graphs this is much less than the tens of thousands of concurrent threads needed to fully occupy a GPU. CC on the other hand tries to color all uncolored nodes each round,

resulting in more work. CC however uses slightly more colors than CJP on average. This is because nodes in CC choose a random color within some range, as opposed to choosing the minimum allowed color as in CJP. We discuss the performance and coloring quality of CC and CJP in more detail in Section 5.3.2.

The CC algorithm maintains a global variable *max-color* giving the maximum color any node can use at the current time. *max-color* represents an estimate of the graph's chromatic number. We initialize *max-color* to 4, though this choice is somewhat arbitrary. During the course of an execution threads can increase *max-color* when they find it is insufficient to color certain nodes; however, *max-color* never decreases. The algorithm also maintains a queue Q of *active* nodes which still need to be colored. Initially Q contains all the nodes. Nodes not in Q are *inactive*, and these all have a color, which will never change. Lastly, each node v has a random value $val(v)$ used for tie-breaking, as we describe later. We now describe the main steps of the algorithm (these steps are also illustrated in Figure 5.2).

- (i) At the beginning of each round every node in Q chooses a random color from 1 to *max-color* that is unused by its neighbors, at the time that it reads its neighbors' colors. Note that due to race conditions two nodes may choose the same color. We intentionally allow this situation in order to maximize parallelism. We describe an efficient GPU method for finding a random unused color in Section 5.2.4.

If v finds that all the colors from 1 to *max-color* are used by its neighbors, then it atomically increments *max-color*, and selects the new value as its color. Note that in a round *max-color* can increase multiple times. As we show in Section 5.3.2.2, *max-color* quickly grows large enough so that nodes picking random unused colors will rarely conflict with each other, despite the race condition noted above.

- (ii) After all active nodes in the round have selected some color, we iterate through the active nodes again. This time, each node checks whether it has any active

neighbor with the same color. If a node v has an active neighbor w with the same color, we will recolor either v or w .

To do this, we first compare v and w 's degrees. If one node has a smaller degree, it loses its color and is added to next round's queue to be recolored. We recolor the lower degree node in order to emulate the largest-degree-first node ordering, which often improves coloring quality. If v and w have the same degree, then the node with lower *val* loses its color and is enqueued for the next round. We describe an efficient GPU enqueue method in Section 5.2.5.

After all nodes have checked their neighbors for conflicts, we go back to step 1 and begin the next round, processing the nodes enqueued in the current round.

If no nodes were enqueued, then all nodes have a color and the algorithm ends.

Since nodes always check for coloring conflicts after all of them have selected a color, the CC algorithm will run until no conflicts remain and will produce a valid coloring. The algorithm may not work efficiently, as a node may be enqueued several times. However, as we discuss in Section 5.3.2, in practice the amount of redundant work is low, and each node was processed an average of 1.3 times.

In CC, we update *max-color* on the fly to estimate the graph's chromatic number. We considered other estimation methods, for example, running CJP on a small random subgraph and using its number of colors to bootstrap CC. However, we found these approaches gave inconsistent coloring quality, and in addition were slower than our current method.

5.2 Efficient Implementation of CJP and CC

In this section, we give details about how to efficiently implement CJP and CC on GPUs. The main task is for nodes to quickly find a minimum or random unused color among its neighbors. This is done by constructing bitmasks to represent the colors used by each node's neighbors. The bitmasks are stored in registers, and due to the limited number of registers this approach can only handle 256 neighbor colors at a time. In Section 5.2.1 we first describe how to break the coloring process into

phases of 256 colors each. We then describe virtual warps and some load balancing techniques in Section 5.2.2. We show the color selection algorithms in Sections 5.2.3 and 5.2.4. Finally, in Section 5.2.5 we describe how to enqueue nodes in different rounds of CJP or CC.

5.2.1 Coloring in phases

Threads in CJP and CC need to find the colors used by each uncolored node's neighbors. To do this each thread uses four 64-bit integer registers to form a bitmask; the value 4 was selected to balance between the number of registers available on each SM and the number of concurrent threads we can run. If a thread is assigned to process a node v 's neighbors, then the i 'th bit in the j 'th integer of the thread's bitmask, for $1 \leq i \leq 64$ and $0 \leq j \leq 3$, is set to 1 only if some neighbor of v is using color $i + 64j$. In this way, we can represent up to 256 colors used by the node's neighbors.

To color graphs using more than 256 colors, we split an execution of CJP or CC into multiple phases. Each phase consists of multiple rounds, as in Sections 5.1.1 and 5.1.2, with the restriction that in the k 'th phase ($k \geq 0$), we try to color the remaining nodes using only colors $256k + 1$ to $256(k + 1)$. Thus, in phase k if bit i in the j 'th integer for a thread is set, it indicates some neighbor of the node the thread is processing has color $i + 64j + 256k$. If at any point during a phase we detect that a node's neighbors are using all the colors in $[256k + 1, 256(k + 1)]$, then the node cannot be colored in the current phase, and so we add it to a queue to color during the next phase. Note that the phase queues are different from the queues used to store uncolored nodes in different rounds of CJP or CC. A phase ends when either all the nodes have been colored, or all remaining nodes have been added to the next phase's queue.

5.2.2 Load balancing

We describe two ways CJP and CC perform load balancing and minimize thread idling. While the GPU hardware provides relatively effective load balancing between

different thread blocks and warps, algorithms need to manually ensure that threads within a warp do similar amounts of work. This is especially important when coloring graphs with widely varying node degrees [70]. Our first method for load balancing uses the technique of Hong *et al* [73], which divides a physical warp of 32 threads into multiple *virtual warps* with W threads each. For each graph, we choose W to be the largest power of two that is less than the average degree of the graph, up to a maximum of $W = 32$. Threads determine their virtual warp ID based on their thread ID and W , and access data according to the virtual warp ID.

Using virtual warps minimizes thread idling when processing low degree nodes because multiple nodes can be processed simultaneously by one physical warp. However, it does not eliminate idling when there are high degree nodes. For example, if there are two virtual warps in one physical warp, and the first virtual warp processes a low degree node while the second processes a high degree node, then the first virtual warp may idle waiting for the second to finish. To address this problem, we also order nodes so that consecutive nodes have similar degrees. In particular, we sort the nodes by degree, and assign virtual warps to the nodes in nondecreasing order of degree. We do the assignment using pointer indirection, and do not change the actual layout of the graph in memory. Sorting also helps emulate the largest degree first heuristic¹ which can improve coloring quality. However, the sorting step incurs some overhead, which we found prohibitive on small graphs. Thus, we only apply the sorting based load balancing technique on graphs with more than 1M nodes.

5.2.3 Finding unused colors in CJP

We now describe how a node v finds the minimum color not used by its neighbors in the CJP algorithm. Consider a virtual warp of W threads assigned to process a node v ; in the rest of this section we simply say warp instead of virtual warp. The threads first perform the function `GETNEIGHBORCOLORS( $v, k$ )`, where $k \geq 0$ is the current phase. This returns an array *used* of four 64-bit integers representing the

¹Since our algorithms are parallel, sorting does not guarantee processing nodes in order of non-decreasing degree.

colors of the current phase used by v 's neighbors, as described in Section 5.2.1. Next, the first thread in the warp performs $\text{MINUNUSEDCOLOR}(used)$ to find the minimum unused color in $used$. The other threads in the warp are idle during this step; since the step is quite fast, this does not hurt performance. If MINUNUSEDCOLOR returns $\perp$, it indicates all colors in the current phase are used by v 's neighbors, and the first thread enqueues v for the next phase. The pseudocode for GETNEIGHBORCOLORS and MINUNUSEDCOLOR are shown in Figure 5.

Procedure 5

```

1: function GETNEIGHBORCOLORS( $v, k$ )
2:    $lane \leftarrow \text{threadIdx.x} \pmod{W}$ 
3:   for  $i \leftarrow 0, 3$  do  $u[i] \leftarrow 0$  end for
4:   for  $i \leftarrow ptr[v] + lane, ptr[v + 1]$  stride  $W$  do
5:      $w \leftarrow col[i]; \quad c_w \leftarrow color[w]$ 
6:     if  $c_w > 256k$  then
7:        $j \leftarrow (c_w - 1) / 64; \quad r \leftarrow (c_w - 1) \pmod{64}$ 
8:        $u[j] \leftarrow u[j] \text{ OR } (1 \ll (r - 1))$ 
9:     end if
10:  end for
11:  for  $i \leftarrow 0, 3$  do
12:     $used[i] \leftarrow \text{REDUCEOR}(u[i])$ 
13:  end for
14:  return  $used$ 
15: end function
16:
17: function MINUNUSEDCOLOR( $used$ )
18:  for  $i \leftarrow 0, 3$  do
19:     $free \leftarrow \neg used[i]$ 
20:    if  $free \neq 0$  then
21:       $r \leftarrow \_ffsll(free)$ 
22:      return  $64i + r$ 
23:    end if
24:  end for
25:  return  $\perp$ 
26: end function

```

We now describe $\text{GETNEIGHBORCOLORS}(v, k)$ in more detail. A thread in the warp assigned to v first computes its $lane$ in the warp. This thread will process every W 'th neighbor of v , starting from the $lane$ 'th. In line 3, the thread initializes an array u of four 64-bit integers. This array is stored as registers of the thread, and is a bitflag representing the colors used by v 's neighbors, among those neighbors processed by the

thread. We assume the graph is represented in CSR format. Then v 's neighbors span the indices $ptr[v]$ to $ptr[v + 1] - 1$ in the graph's column array col , and v 's $lane$ 'th neighbor has index $ptr[v] + lane$ in col . In line 5, the thread finds a neighbor w of v , and reads from the global memory array $color$ to find w 's color c_w . If $c_w > 256k$, it indicates w has been assigned a color used in the current phase. This is recorded by setting the r 'th bit in the appropriate $u[j]$ to 1, as in lines 7 and 8. After the thread has done this for every W 'th neighbor of v , it combines its bitflags with those from the other threads in the warp in lines 11-13. In particular, for each i from 0 to 3, every thread calls the function `REDUCEOR( $u[i]$ )`, which computes the bitwise OR of the W values of $u[i]$, one from each thread, using a fast reduction operation. After this, the four values in $used$ show all the used colors for the current phase among v 's neighbors.

Next, we describe the function `MINUNUSEDCOLOR`. It takes as input the node v and the four 64-bit values in $used$. For every i from 0 to 3, it computes $free$ as the negation of $used[i]$. If $free \neq 0$, then there is at least one set bit in $free$, indicating an unused color in $used[i]$. This color is extracted using CUDA's `__ffsll` intrinsic function, which returns the least significant set bit in its input, and `MINUNUSEDCOLOR` returns $64i + r$ as the smallest unused color. If $free = 0$ for $0 \leq i \leq 3$, this indicates all 256 colors of the current phase are used by v 's neighbors, and the function returns $\perp$. Then the calling function will enqueue v to be colored in the next phase.

5.2.4 Finding unused colors in CC

We next describe how a node v in the CC algorithm finds a random color not used by its neighbors. As in CJP, v is processed by a virtual warp of W threads. Also as in CJP, the threads first call `GETNEIGHBORCOLORS( $v$ )` to find the colors used by v 's neighbors, getting back an array $used$. Then the first thread in the warp calls `RANDOMUNUSEDCOLOR( $used, max-color$ )`. Let the return value be c . If $c \leq max-color$, then c is a random color unused by v 's neighbors, and we assign c to v . Otherwise, $c = max-color + 1$, indicating that v 's neighbors are using all the colors

for the current phase up to $max\text{-}color$. In this case, if $c \leq 256(k + 1)$, where k is the current phase, we increment the number of allowed colors by letting the *lane 0* thread perform `atomicMax(max-color, c)`. If $c > 256(k + 1)$, then $max\text{-}color$ has reached the largest possible value for the current phase, and the thread enqueues v to color in the next phase.

Procedure 6

```

1: function RANDOMUNUSEDCOLOR(used, max-color)
2:   for  $i \leftarrow 0, 3$  do
3:      $u \leftarrow u + \text{\_\_popc}(\textit{used}[i])$ 
4:   end for
5:   if  $u = max\text{-}color$  return  $max\text{-}color + 1$  end if
6:    $r \leftarrow \text{RANDOM}(max\text{-}color - u)$ 
7:   for  $i \leftarrow 0, 3$  do
8:      $free \leftarrow \neg \textit{used}[i]$ 
9:      $s \leftarrow \text{\_\_popc}(free)$ 
10:    if  $s < r$  then
11:       $r \leftarrow r - s$ ; continue
12:    end if
13:    while true do
14:       $c \leftarrow \text{\_\_ffsll}(free)$ 
15:       $r \leftarrow r - 1$ ;
16:      if  $r = 0$  then return  $64i + c$  end if
17:       $free \leftarrow free \text{ AND } \neg(1 \ll (c - 1))$ 
18:    end while
19:  end for
20: end function

```

The pseudocode for `RANDOMUNUSEDCOLOR` is shown in Figure 6. In lines 2-4, it first counts the total number of used colors u in $used$ using the `\_\_popc` intrinsic function, which returns the number of set bits in its input. If $u = max\text{-}color$, then all the currently allowed colors are used by i 's neighbors, and the function returns $max\text{-}color + 1$. Otherwise, the function proceeds to look for the r 'th unset bit in $used$, for a random r computed in line 6. It does this by setting $free$ to the unused colors in $used[i]$, for i from 0 to 3. Then it counts off r colors from $free$ in lines 7 to 19 and returns the last such color. Specifically, lines 9-12 are used to find the register containing the r 's unset bit. Then in line 14, we set c to be the least significant set bit in $free$ for this register, and proceed to decrement r . If $r = 0$, we return $64i + c$. If $r > 0$, then we unset the c 'th bit of $free$ in line 17, so that on the next iteration of

the while loop we find the next set bit after the c 'th.

5.2.5 Enqueueing nodes

We now describe how nodes are enqueued in CJP and CC. The pseudocode for the enqueue procedure for CJP is shown in Figure 7. Recall that the nodes in a CPJ queue decrement the *count* values of all their neighbors, then enqueue any neighbor with *count* = 0 into the next round's queue. Consider some node v , processed using a virtual warp of W threads. Each thread in the warp first computes its *lane*, and then computes v 's pseudorandom value val by hashing v . The thread then iterates through every W 'th neighbor of v , starting from the *lane*'th. For each neighbor w , it hashes w to compute its value val_w . If $val > val_w$, it decrements $count[w]$, stored in global memory, using `atomicSub`. This returns the value c_w of $count[w]$ before the decrement. If $c_w = 1$, then we now have $count[w] = 0$, and w should be enqueued. This is indicated by setting the thread's variable c to 1.

Next, when all the threads in the warp arrive at line 10, they perform the `--ballot( $c$ )` intrinsic function, which returns an integer s whose *lane*'th bit is set if and only if the *lane*'th thread in the warp has $c = 1$. If $s = 0$, then no thread from the warp wants to enqueue a node, and we continue to the next iteration of the main for loop. Otherwise, in lines 12 and 13 each thread computes its *rank*, the number of threads in the warp with smaller thread IDs that want to enqueue a node. Note that *rank* for the lane $W - 1$ thread in the warp equals the total number of threads that want to enqueue a node into next round's queue. Then, the lane $W - 1$ thread increases the *head* pointer for the queue by its *rank* using `atomicAdd`. Since at most one thread from each warp does this atomic operation, the number of atomicity collisions should be low. The `atomicAdd` returns the old value of *head*, which the $W - 1$ 'st thread broadcasts to the other threads in the warp using the `--shfl` intrinsic function in line 17. Now, each thread in the warp with $c = 1$ can enqueue its node into position $old-head + rank - 1$ of next round's queue.

Procedure 7

```
1: procedure CJPENQUEUENEIGHBORS( $v$ )
2:    $lane \leftarrow \text{threadIdx.x} \pmod{W}$ 
3:    $val \leftarrow \text{HASH}(v)$ 
4:   for  $j \leftarrow ptr[i] + lane, ptr[i + 1]$  stride  $W$  do
5:      $w \leftarrow col[j]$ ;  $val_w \leftarrow \text{HASH}(w)$ ;  $c \leftarrow 0$ 
6:     if  $val > val_w$  then
7:        $c_w \leftarrow \text{atomicSub}(count[w], 1)$ 
8:       if  $c_w = 1$  then  $c \leftarrow 1$  end if
9:     end if
10:     $s \leftarrow \_\_ballot(c)$ 
11:    if  $s = 0$  then continue end if
12:     $s \leftarrow s \text{ AND } ((1 \ll lane) - 1)$ 
13:     $rank \leftarrow \_\_popc(s)$ 
14:    if  $lane = W - 1$  then
15:       $old\_head \leftarrow \text{atomicAdd}(head, rank)$ 
16:    end if
17:     $old\_head \leftarrow \_\_shfl(old\_head, W - 1, W)$ 
18:    if  $c = 1$  then
19:       $queue[old\_head + rank - 1] \leftarrow w$ 
20:    end if
21:  end for
22: end procedure
```

The enqueue procedure for CC is similar. The main difference is in lines 5 to 9, where nodes look for neighbors with conflicting colors instead of those with $count = 0$, and enqueue nodes with lower degrees or smaller hash values. The rest of the procedure is the same.

5.3 Experimental results

Table 5.1: List of graphs used in experiments

Name	# Nodes	# Edges	Avg degree	Max degree
cant	63K	2M	64	78
economics	207K	0.7M	12	56
epidemiology	526K	1.1M	7	7
accelerator	121K	1.3M	22	81
QCD	49K	0.9M	42	42
harbor	47K	1.2M	51	145
scircuit	171K	0.5M	6	353
webbase-1M	1M	2.5M	5	28,687
spheres	83K	3M	72	81
ship	141K	3.9M	55	68
pwtk	218K	5.8M	53	180
protein	36K	2.2M	119	204
2cubes_sphere	101K	0.8M	16	31
cage12	130K	1M	16	33
hood	221K	5.4M	49	77
m133-b3	200K	0.4M	8	8
majorbasis	160K	0.9M	15	19
mario002	390K	1.1M	5	7
offshore	260K	2.1M	16	31
patents_main	241K	0.3M	5	212
poisson3Da	14K	0.2M	26	4,491
circuit5M	5.6M	29.8M	11	1,290,501
audikw_1	0.9M	38.8M	82	345
Bump_2911	2.9M	63.8M	44	195
rgg_n_2_24_s0	8.4M	265.1M	16	40
europe_osm	50.9M	54M	2	13
kron_g500-logn21	2.1M	91M	87	213,905
soc-LiveJournal1	4.8M	43.1M	18	20,334
hollywood-2009	1.1M	57M	100	11,468
indochina-2004	7.4M	152.2M	41	25,6425
nlpkkt200	16.2M	224.1M	28	28
nlpkkt240	28M	387.3M	28	28

Table 5.2: Average running time (in milliseconds) and number of colors used for different algorithms in small graphs.

Graph	csrcolor		VEB		ColPack						CJP		CC	
	time	colors	time	colors	Largest-first		Smallest-last		Incidence-degree		time	colors	time	colors
					time	colors	time	colors	time	colors				
cant	10.9	112.0	17.1	36.1	9.1	27	11.6	23	11.4	25	4.0	31.0	3.5	40.6
economics	6.9	64.0	7.6	11.6	8.8	7	10.4	7	13.5	7	4.4	10.0	3.4	13.7
epidemiology	4.3	32.0	12.5	7.4	8.5	4	15.5	3	24.6	3	6.0	7.0	4.6	7.0
accelerator	5.7	79.0	9.7	22.5	9.9	20	13.5	18	14.7	18	5.7	20.0	3.4	23.5
QCD	5.1	80.0	7.7	15.0	4.8	2	6.0	2	5.6	2	3.9	8.0	2.1	17.5
harbor	10.5	139.0	19.6	55.1	7.8	53	8.1	49	7.9	48	9.1	51.0	2.6	58.0
scircuit	3.7	48.0	7.0	10.3	5.7	8	7.4	8	6.9	7	3.5	10.0	1.6	11.0
webbase-1M	110.2	208.0	34.4	63.1	21.0	62	21.2	63	22.0	63	98.8	62.0	17.2	67.0
spheres	15.7	128.0	24.7	46.1	11.8	30	16.7	33	18.3	33	8.5	42.0	4.6	49.5
ship	16.3	109.0	33.8	43.1	20.2	54	23.3	38	28.4	37	10.0	38.0	6.0	46.9
pwtk	22.0	96.0	51.5	39.2	25.8	42	28.3	35	32.8	40	10.0	37.0	9.1	43.5
protein	19.8	189.0	28.3	84.9	12.4	78	11.2	64	11.9	66	12.4	75.0	3.9	85.3
2cubes_-sphere	3.5	48.0	5.9	13.5	9.3	12	12.4	10	12.4	10	3.5	12.0	2.2	14.3
cage12	4.0	64.0	7.0	15.4	8.7	13	13.7	12	14.1	13	4.8	14.0	2.7	16.2
hood	20.4	96.0	39.2	36.2	24.5	42	33.1	35	41.7	35	11.4	35.0	8.1	41.2
m133-b3	3.7	48.0	6.2	8.7	4.3	6	14.1	7	13.7	7	3.2	7.0	2.2	9.0
majorbasis	3.7	48.0	7.0	12.3	5.1	6	4.9	7	5.6	9	8.0	10.0	3.0	13.8
mario002	3.4	32.0	5.8	7.4	9.2	5	14.5	4	14.3	5	4.0	6.0	3.3	7.0
offshore	6.5	48.0	14.8	14.6	25.3	11	34.0	10	36.3	11	5.9	13.0	5.4	15.7
patents_-main	5.6	64.0	5.7	15.5	12.2	10	14.3	9	14.8	9	6.3	13.0	2.2	14.6
poisson3Da	2.8	64.0	4.8	20.2	1.5	16	2.4	15	2.4	15	4.2	18.0	0.9	20.9

Table 5.3: Average running time (in milliseconds) and number of colors used for different algorithms in big graphs.

Graph	csrcolor		VEB		ColPack						CJP		CC	
	time	colors	time	colors	Largest-first		Smallest-last		Incidence-degree		time	colors	time	colors
					time	colors	time	colors	time	colors				
circuit5M	8,540.1	341.0	380.2	7.8	117.2	6	120.1	5	126.8	5	1.991.8	9.0	237.4	13.9
audikw_1	216.1	160.0	222.5	50.2	224.9	51	261.9	44	291.2	44	55.0	46.0	54.2	69.3
Bump_2911	203.8	96.0	332.1	32.2	350.3	31	449.2	25	518.1	28	94.2	29.0	115.3	37.0
rgg_n.2- 24_s0	⊥	⊥	790.1	26.7	2,812.8	21	4,045.1	21	3,510.3	22	421.4	24.0	719.1	26.5
europe_osm	⊥	⊥	148.3	6.1	1,096.8	5	1,773.9	4	1,433.1	4	328.0	5.0	270.1	6.0
kron_g500- logn21	14,069.6	3,465.0	1,185.4	644.4	1,201.0	490	1,219.0	484	1,308.8	487	7,428.3	604.0	721.6	816.3
soc- LiveJournal1	610.8	557.0	279.8	332.6	1,012.7	323	1,069.8	321	1,218.7	325	299.0	331.0	217.0	339.7
hollywood- 2009	1,779.5	2,317.0	665.8	2,209.0	499.1	2,209	453.5	2,209	490.6	2,209	2,930.5	2,209.0	252.5	2,209.0
indochina- 2004	8,437.4	7,030.0	6,300.7	6,849.8	1,009.5	6,848	991.6	6,848	836.1	6,848	⊥	⊥	1,305.4	6,851.2
nlpkkt200	502.1	77.0	⊥	⊥	819.0	2	1,041.4	2	1,853.0	2	413.0	11.0	457.2	12.0
nlpkkt240	⊥	⊥	⊥	⊥	1,408.3	2	1,792.3	2	3,424.6	2	725.8	11.0	786.8	13.0

In this section, we evaluate the performance of our algorithms. We ran the algorithms on 32 matrices from the University of Florida sparse matrix collection [68], listed in Table 5.1. Since our coloring is on undirected graphs, we transformed any nonsymmetric matrix A to a symmetric one by computing $A + A^\top$. The graphs in the top part of the table are smaller, with up to 5.8M edges, while the graphs in the bottom part have up to 387M edges. Our GPU is an Nvidia Tesla K20Xm with 2688 cores in 14 SMs, peak single precision performance of 3935 GFlops/s and 250GB/s of memory bandwidth. Each SM has 48 KB of shared memory and 32K 64-bit registers. The version of CUDA is 7.0. The CPU experiments were done on an 8-core Intel Xeon E5-2670 2.60GHz with 128GB of RAM running Linux CentOS 6.4.

We compared our algorithms with `csrcolor` from Nvidia’s `cuSPARSE` (in CUDA 7.0) sparse matrix library [34], Deveci *et al*’s algorithms from [30], and Gebremedhin *et al*’s CPU-based algorithm in ColPack. Deveci *et al* proposed two algorithms called *vertex-based* and *edge-based*. For each graph, we show the running time from the faster of the two algorithms as well as the coloring quality of the faster algorithm. We label the result VEB, since either the vertex or edge based algorithm may be faster depending on the graph. The ColPack algorithm is single-threaded. It implements a number of node ordering heuristics, which we label *Largest first*, *Smallest last* and *Incidence degree* in our results. Since many of the algorithms are randomized, the reported running time and color usage for each graph are the average from 10 runs.

5.3.1 Throughput and number of colors

We first compare the running times of CJP and CC with the other algorithms. The results are shown in Tables 5.2 and 5.3. We also show the results graphically in Figure 5.3. However, since the running times on different graphs vary widely, we normalized the running times of all the algorithms to that of CC, which was typically the fastest algorithm. Note that on certain graphs some of the algorithms crashed. These are indicated by a $\perp$ in Table 5.3 or by missing bars in Figure 5.3.

CJP and CC are on average $1.5\times$ and $4.3\times$ faster than `csrcolor`, resp. Both CJP and CC showed the highest speedup on the graph *circuit5M*, where they were

(a)

(b)

Figure 5.3: Running times of different algorithm normalized to CC's time

(a)

(b)

 Figure 5.4: Number of colors used by different algorithms normalized to *Smallest-last*'s color usage

$4.3\times$ and $35\times$ faster, resp. Compared to VEB, CJP and CC were $2\times$ and $3.3\times$ faster on average, resp. CJP had the largest improvement on *ptwk*, where it was at $5.2\times$ faster, and CC performed best on *harbor*, where it was $7.6\times$ faster. Among the three

Figure 5.5: Number of enqueued nodes (y -axis) in each round (x -axis) for *webbase-1M*, *patents_main* and *epidemiology*.

heuristics we tested in ColPack, Largest-first was the fastest. CJP and CC were $1.9\times$ and $2.7\times$ faster, resp. than Largest-first on average, and up to $6.7\times$ and $5.5\times$ faster for some graphs.

Next, we compare the algorithms in terms of the number of colors they used. The results are shown in Tables 5.2 and 5.3 and Figure 5.4. In the two figures, the results have been normalized to the number of colors used by ColPack’s Smallest-last ordering, since this usually led to the best coloring. `csrcolor` gave the worst results, often using many times more colors than Smallest-last, and so for visual clarity we do not show it in the figures. On average CJP used $4.3\times$ fewer colors than `csrcolor` and CC used $3.5\times$ less. The number of colors used by VEB was intermediate between CJP and CC. On average VEB used 10% more colors than CJP and 10% fewer colors than CC. ColPack provided the best coloring quality, with the Smallest-last and Incidence-degree orderings typically the most effective. On average, CJP and CC used $1.5\times$ and $2\times$ more colors than Smallest-last, resp. However, these results are somewhat skewed by the presence of three bipartite graphs, *QCD*, *nlpkkt200* and *nlpkkt240*. Smallest-last colored these graphs optimally, whereas CJP and CC used $4 - 6.5\times$ more colors. If these outliers are removed, then CJP and CC used $1.3\times$ and $1.5\times$ more colors than Smallest-last, resp. Meanwhile, since bipartite graphs can be quickly colored using breadth-first search, we can add a heuristic to CJP and CC to quickly deal with this special case.

In graph *indochina-2004*, ColPack performs best in terms of both runtime and number of colors. However, ColPack uses only about 3 colors fewer than our CC method, which is insignificant since the graph needs high number of colors (6,848 in

ColPack). Compared to other GPU library VEB using smallest-first coloring technique, CC chooses random color for each node, which tends to produce more colors; though, CC spends only 2 more colors than VEB on average. Note that, the running time of CC is about 5 times smaller than that of VEB. *indochina-2004* is a web graph, in which the row lengths follow power-law distribution. The high variance in row length is not suitable to parallel algorithm since it is difficult to balance work among threads. ColPack does not suffer from this issue as it is a serial library. Compared to fastest ColPack algorithm, our CC method is 60% slower. However, CC is 5-6.5 times faster than other parallel libraries (`csrcolor` and VEB), which shows that CC is better at dealing with imbalance than them.

As shown in Table 5.3, `csrcolor` crashes in *rgg-n-2-24-s0*, *nlpkkt240* and so does VEB in *nlpkkt200* and *nlpkkt240* datasets. Since we don't know the detailed internal implementation of other libraries, we cannot explain the exact reason why they crash. Our guess is they got memory overflow issue since they often crash on big graphs.

Our CJP method did not crash but could not finish execution on Indochina-2004 dataset. Our analysis shows that when processing this graph, after a certain number of iterations, the queue size becomes very small and there are only few nodes in the queue. This reduces the parallelism of CJP method and takes it a lot of iterations and a very long time to finish coloring. We are not sure the exact reason for this behavior, but it may be due to the special structure of Indochina-2004.

5.3.2 Work and efficiency

In this section, we analyze CJP and CC in terms of their parallelism and work efficiency. We also look at how quickly the value of *max-color* converges in CC.

5.3.2.1 Parallelism and work

The amount of work in each round of CJP and CC depends on the number of nodes in the round's queue. Figure 5.5 shows the queue size in different rounds of CJP and CC on three graphs, *epidemiology*, *webbase-1M* and *patents-main*. These graphs were

chosen because they exhibit large variations in the relative performance of CJP and CC.

CJP perform far more rounds than CC on all the graphs, finishing in 20, 175 and 45 rounds on *epidemiology*, *webbase-1M* and *patents_main*, resp., versus 5, 13 and 4 rounds for CC. Since each round is performed in a separate kernel, this created a large amount of overhead for CJP. CC was able to finish in a small number of rounds because its queue size decreased very quickly, dropping by a factor of 23, 108 and 27 after the first round on *epidemiology*, *webbase-1M* and *patents_main*, resp. We describe why this happened when we discuss the rate of increase in *max-color* in the next section.

In addition to running for a large number of rounds, CJP also had relatively small queue sizes for a large portion of its execution in *webbase-1M* and *patents_main*. The queues had less than 4000 nodes for about 3/4 of the rounds in both cases. Such a small queue size may not provide enough work to fully occupy the GPU. As a result, CC performed substantially better than CJP on these graphs, being about $2.9\times$ and $5.7\times$ faster on *patents_main* and *webbase-1M*, resp. On the other hand, CJP was able to maintain relatively large queues of over 20K nodes in *epidemiology*. Here, CJP's performance was much closer to CC's, being only $1.3\times$ slower.

The CC algorithm may process a node multiple times due to coloring conflicts and thus perform redundant work. To measure the amount of redundancy on a graph, we summed up the sizes of the queues in all the rounds of CC's execution on the graph and divided this by the number of nodes in the graph. The average redundancy across all the test graphs was 1.3. This shows that conflicts are not very frequent, and that CC is almost as work efficient as CJP.

5.3.2.2 Rate of increase of *max-color*

Recall that in CC, nodes that cannot find an unused color increment *max-color*. In this section we look at how quickly *max-color* increases on several graphs. A faster increase leads to fewer coloring conflicts and less redundant work. At the same time, we want to keep *max-color* small enough to avoid wasting colors. Figure 5.6 shows the

Figure 5.6: Size of $max\text{-color}$ relative to its final value, as a function of the percent of nodes colored in the first round for *cant*, *harbor* and *webbase-1M*.

size of $max\text{-color}$ compared its final value at the end of the execution, as a function of the percent of nodes in the first round queue that have been colored, for the graphs *cant*, *harbor* and *webbase-1M*. These matrices were chosen because they show different patterns in $max\text{-color}$'s increase. Also, we focus on $max\text{-color}$'s change in the first round because it does not increase much in subsequent rounds.

In *cant*, $max\text{-color}$ increases quite quickly, and reaches 80% of its final value after processing about 1/3 of the nodes. For *harbor*, the growth is much slower, and it only reaches 80% after processing 3/4 of the nodes. The reason is, *harbor* being a small graph, we did not sort its nodes by degree, and so many low degree nodes occurred near the start of the execution. These nodes are able to color themselves using only a few colors, and so $max\text{-color}$ increased relatively slowly. In *webbase-1M*, we again did not sort the nodes by degree. The degrees followed a power law distribution, where most nodes have low degrees and a few nodes have much higher degrees. Thus as the nodes are processed we initially encounter mostly low degree nodes which can be colored using only a few colors. It is only after about half the nodes have been processed that we encounter some high degree nodes which require a large value of $max\text{-color}$, and so it is at this point that $max\text{-color}$ increased sharply to the its final value. After this large increase, all the nodes in the graph could more easily find unused colors, which improves CC's performance. On this graph CC was $6.4\times$ faster than *csrcolor*.

5.4 Summary

In this Chapter, we presented two algorithms for efficient graph coloring on the GPU. The CJP algorithm is a work efficient implementation of the Jones-Plassmann algorithm, and the CC algorithm optimistically colors nodes while adjusting the allowed color range dynamically. Both algorithms find high quality colorings while being substantially faster than state-of-the-art algorithms.

In the future, we hope to incorporate our coloring algorithms into GPU linear algebra packages to study the effects of coloring speed and quality on end-to-end linear algebra performance. We also want to further improve the performance of our algorithms through better load balancing, especially in scale-free networks with highly skewed degree distributions. Finally, we want to implement additional node ordering heuristics such as smallest-degree-last to improve coloring quality, and examine the tradeoff in coloring speed versus quality.

Chapter 6

Conclusions and Future Works

In this chapter, we recap the chapters to highlight main contributions of this thesis. Then, we discuss some future research directions in our area.

6.1 Conclusions

The main contents of this thesis can be outlined as follows:

- **Chapter 1:** We provide background knowledge about GPUs and explain why it is hard to get high throughput on irregular data using GPUs. We also provide overview of 3 irregular graph problems that we study in this thesis: *Sparse matrix-vector multiplication*, *Sparse matrix-matrix multiplication* and *Graph coloring*.
- **Chapter 2:** We discuss existing researches on sparse graph problems on GPUs, including: different sparse graph storage formats on GPUs, previous works on 3 problems.
- **Chapter 3:** In this chapter, we describe our study on GPU SpMV problem. First, we present our discover that accesses on vector's data are the performance bottleneck for SpMV on GPUs. Then, we illustrate our 2 proposed methods to improve the vector access. The first technique is based on the idea that aggregates many scattered accesses for each row to a single dense access by a *super-row*. The second method optimizes the vector accesses between 2 consecutive rounds in iterative SpMV. We also show that our 2 approaches are independent

of matrix format used by SpMV by applying them on top of other well-known SpMV libraries. Finally, we evaluate these 2 techniques via experiments and demonstrate that they help improve the SpMV throughput.

- **Chapter 4:** This chapter focuses on the SpGEMM problem. We discuss 2 main issues in SpGEMM on GPUs that existing studies have not resolved efficiently: load imbalance and workload efficiency. To overcome load imbalance problem, we design 2 techniques that divide work among threads evenly. These techniques are lightweight so their overhead is minor; whereas, they help us to get perfect balancing in the main SpGEMM steps. Then, we propose using hash table on fast shared memory to process data as opposed to using high-latency global memory like existing approaches. Using hash table is also more work efficient than previous merging/sorting method used by previous studies. We also show how to divide the data so that we can process them on each SM's shared memory. The experiments illustrates the efficiency of our improvement especially on matrices with high load imbalance and work overhead problem.
- **Chapter 5:** We describe main issue of parallel Graph coloring method, i.e. high color conflicts when many nodes are colored concurrently. From this observation, we devise 2 methods to perform Graph coloring on GPUs. The first technique extends Luby's approach to work on GPUs; this method is conflict-free and very work efficient. The second technique allows temporary color conflicts to obtain better parallelism. Also in the second method, we improve the way color is selected for each node so that the conflict rate is minimized when the number of colors is small. Compared to other GPU Graph coloring libraries, our methods yield better running time and color quality.

6.2 Future Works

In this study, we have improved the performance of different sparse graph algorithms on GPUs. Nevertheless, as shown in experiments, the effective performance of these

algorithms is just only a fraction of maximum theoretical throughput of GPUs. There is still a huge gap between performance on irregular and regular algorithms. This is a challenge and an opportunity to enhance GPU's throughput on sparse scattered data further.

6.2.1 Load balancing

In all 3 graphs algorithms studied in this thesis, we apply different techniques to balance the work among threads when processing data in ill-structured pattern. The most frequently used technique is *virtual warp* [73] since it is easy to implement (as it can be seen a generalization of conventional GPU programming technique using physical warp) and it gives little overhead during runtime. However, this method can only solve the imbalance problems between threads in a warp (up to factor of 32). When there is imbalance between warps and the factor is very high, the virtual warp method fails to fix it. A solution proposed by Liu et al. [4] is dividing the workload into 3 groups and the data in each group is processed by a thread block, a warp and a thread, resp. This technique helps us to deal with cases with huge imbalance, e.g. rows having ten thousands nonzeros. The shortcoming of this approach is we need different kernels for these groups and launching separate kernels may cause overhead and low occupancy. Moreover, when processing data using a thread block, different techniques are needed to synchronize and combine data from threads, which brings more synchronizing and work overhead. The perfect load balancing mechanism used in our SpGEMM study (Chapter 4) is beneficial only when the threads' jobs are independent of each other and their workloads are high, so the balancing overhead will not overwhelm the performance gain achieved by this method.

Load balancing is a major obstacle when processing unstructured data using GPUs due to its SIMD nature. Many techniques have been proposed but none of them can resolve this issue completely. This leaves room for further improvement in terms of both software and hardware in future researches.

6.2.2 Multi-GPU

In our study, we only have considered only single GPU problem with assumption that data can fit in a GPU's memory. Nevertheless, as data has been increased enormously, the desire for multi-GPU solution is also undeniable. When it comes to multi-GPU environment, new issues arise and we need different methods to overcome these issues. The biggest problem for multi-GPU programs is the cost of synchronization and data transfer between GPUs. Take SpMV as an example, we assume that the matrix is so big that we need multiple GPUs to store their data. This means we also have to have multiple copies of vectors x and y in $Mx = y$ to perform the SpMV in each GPU. In the end of SpMV or each round in iterative SpMV, we combine the results y in all GPUs to create the overall y . Although the amount of y data may not be significant, the latency of transferring data between GPU and CPU is high compared to SpMV computation in GPUs. Because of this latency, the speedup of multi-GPU algorithms are not linear with the number of GPUs [36], [4]. With the recent development of NVLINK technology, which allows more than 300 GB/s bandwidth between GPUs, the performance on multi-GPU will be increased significantly and it is interesting to see how our algorithms would perform.

6.2.3 Relation between Matrix Structure and Performance

In this research, we have proposed different heuristics for our sparse matrix problems. Based on these algorithms, we improve the performance over previous methods. However, we still do not understand how the performance varies in different patterns of matrix's data. In other words, we would like to have a quantitative analysis of factors in matrix structure that affect the outcome throughput. This study will help us to have a good modeling of how GPUs work and what is the key element for high performance when processing any type of data. With this computing model, we can predict the behavior of GPUs in various circumstances, e.g. different nonzeros patterns, GPUs with different specifications, etc.; hence, we are able to tune the runtime parameters to suitable values accordingly.

6.2.4 Other Sparse problems

There are number of other important Sparse problems for GPUs such as Breadth First Search (BFS) [70], [88], [89], Single-source and All-pairs Shortest Path [23], [5], [6], [90]. These sparse problems also have similar issues of our studied problems. For example, in BFS, the main performance obstacle is the non-coalesced memory access which happens when we check the visiting status of a node's neighbors. This is similar to accessing vector entries regulated by nonzeros in a row of a sparse matrix in SpMV. It suggests that we can apply our SpMV techniques for BFS problem although it is more difficult since in BFS we often go through the graph only once, the overhead of preprocessing step may be higher for BFS and the access is determined by starting point, which is different for each BFS run.

References

- [1] W. Liu and B. Vinter, “An efficient gpu general sparse matrix-matrix multiplication for irregular data,” in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 370–381.
- [2] S. Dalton, L. Olson, and N. Bell, “Optimizing sparse matrix-matrix multiplication for the gpu,” *ACM Trans. Math. Softw.*, vol. 41, no. 4, pp. 25:1–25:20, Oct. 2015. [Online]. Available: <http://doi.acm.org/10.1145/2699470>
- [3] S. Beamer, K. Asanović, and D. Patterson, “Direction-optimizing breadth-first search,” *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.
- [4] H. Liu and H. H. Huang, “Enterprise: Breadth-first graph traversal on gpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 68.
- [5] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal, “Scalable single source shortest path algorithms for massively parallel systems,” in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 889–901.
- [6] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, “Work-efficient parallel gpu methods for single-source shortest paths,” in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 349–359.
- [7] F. Wang, C.-Q. Yang, Y.-F. Du, J. Chen, H.-Z. Yi, and W.-X. Xu, “Optimizing linpack benchmark on gpu-accelerated petascale supercomputer,” *Journal of Computer Science and Technology*, vol. 26, no. 5, pp. 854–865, 2011.
- [8] X.-J. Yang, X.-K. Liao, K. Lu, Q.-F. Hu, J.-Q. Song, and J.-S. Su, “The tianhe-1a supercomputer: its hardware and software,” *Journal of Computer Science and Technology*, vol. 26, no. 3, pp. 344–351, 2011.

- [9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cudnn: Efficient primitives for deep learning,” *arXiv preprint arXiv:1410.0759*, 2014.
- [10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [11] T. Shimokawabe, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, T. Aoki, A. Nukada, and S. Matsuoka, “Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer,” in *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2011, pp. 1–11.
- [12] C. Xu, X. Deng, L. Zhang, J. Fang, G. Wang, Y. Jiang, W. Cao, Y. Che, Y. Wang, Z. Wang *et al.*, “Collaborating cpu and gpu for large-scale high-order cfd simulations with complex grids on the tianhe-1a supercomputer,” *Journal of Computational Physics*, vol. 278, pp. 275–297, 2014.
- [13] M. Harris *et al.*, “Optimizing parallel reduction in cuda,” *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.
- [14] O. Green, R. McColl, and D. A. Bader, “Gpu merge path: a gpu merging algorithm,” in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 331–340.
- [15] J. Demouth, “Sparse matrix-matrix multiplication on the gpu,” in *Proceedings of the GPU Technology Conference*, 2012.
- [16] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, “Sparse matrix solvers on the gpu: conjugate gradients and multigrid,” in *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3. ACM, 2003, pp. 917–924.
- [17] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 18.
- [18] J. W. Choi, A. Singh, and R. W. Vuduc, “Model-driven autotuning of sparse matrix-vector multiply on gpus,” in *ACM sigplan notices*, vol. 45, no. 5. ACM, 2010, pp. 115–126.

- [19] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S.-h. Kuo, R. S. M. Goh, S. J. Turner, and W.-F. Wong, “Accelerating sparse matrix-vector multiplication on gpus using bit-representation-optimized schemes,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 26.
- [20] S. Yan, C. Li, Y. Zhang, and H. Zhou, “yaspmv: Yet another spmv framework on gpus,” in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 107–118.
- [21] N. Bell, S. Dalton, and L. N. Olson, “Exposing fine-grained parallelism in algebraic multigrid methods,” *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012. [Online]. Available: <http://dx.doi.org/10.1137/110838844>
- [22] J. R. Gilbert, S. Reinhardt, and V. B. Shah, “High-performance graph algorithms from parallel sparse matrices,” in *Applied Parallel Computing. State of the Art in Scientific Computing*, ser. Lecture Notes in Computer Science, B. Kogström, E. Elmroth, J. Dongarra, and J. Waniewski, Eds. Springer Berlin Heidelberg, 2007, vol. 4699, pp. 260–269. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75755-9_32
- [23] T. 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*, ser. STOC ’07. New York, NY, USA: ACM, 2007, pp. 590–598. [Online]. Available: <http://doi.acm.org/10.1145/1250790.1250877>
- [24] D. J. Welsh and M. B. Powell, “An upper bound for the chromatic number of a graph and its application to timetabling problems,” *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.
- [25] D. W. Matula and L. L. Beck, “Smallest-last ordering and clustering and graph coloring algorithms,” *Journal of the ACM (JACM)*, vol. 30, no. 3, pp. 417–427, 1983.
- [26] T. F. Coleman and J. J. Moré, “Estimation of sparse jacobian matrices and graph coloring blems,” *SIAM journal on Numerical Analysis*, vol. 20, no. 1, pp. 187–209, 1983.
- [27] D. Brélaz, “New methods to color the vertices of a graph,” *Communications of the ACM*, vol. 22, no. 4, pp. 251–256, 1979.

- [28] M. Naumov, P. Castonguay, and J. Cohen, “Parallel graph coloring with applications to the incomplete-lu factorization on the gpu,” Nvidia Technical Report NVR-2015-001, Nvidia Corp., Santa Clara, CA, Tech. Rep., 2015.
- [29] J. Cohen and P. Castonguay, “Efficient graph matching and coloring on the gpu,” in *GPU Technology Conference*, 2012, pp. 1–10.
- [30] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, “Parallel graph coloring for manycore architectures,” *IEEE International Conference on Parallel and Distributed Processing (IPDPS)*, May 2016.
- [31] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, “Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu,” in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 451–460.
- [32] A. Monakov, A. Lokhmotov, and A. Avetisyan, “Automatically tuning sparse matrix-vector multiplication for gpu architectures,” in *High Performance Embedded Architectures and Compilers*. Springer, 2010, pp. 111–125.
- [33] F. Vázquez, J.-J. Fernández, and E. M. Garzón, “A new approach for sparse matrix vector product on nvidia gpus,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 8, pp. 815–826, 2011.
- [34] “Nvidia cusparse library,” <https://developer.nvidia.com/cusparse>.
- [35] “Cusp library,” <http://code.google.com/p/cusp-library/>.
- [36] X. Yang, S. Parthasarathy, and P. Sadayappan, “Fast sparse matrix-vector multiplication on gpus: implications for graph mining,” *Proceedings of the VLDB Endowment*, vol. 4, no. 4, pp. 231–242, 2011.
- [37] X. Sun, Y. Zhang, T. Wang, X. Zhang, L. Yuan, and L. Rao, “Optimizing spmv for diagonal sparse matrices on gpu,” in *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011, pp. 492–501.
- [38] B.-Y. Su and K. Keutzer, “clspmv: A cross-platform opencl spmv framework on gpus,” in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 353–364.

- [39] D. Grewe and A. Lokhmotov, “Automatically generating and tuning gpu code for sparse matrix-vector multiplication from a high-level representation,” in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011, p. 12.
- [40] Y. Liu and B. Schmidt, “Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus,” in *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*. IEEE, 2015, pp. 82–89.
- [41] I. Reguly and M. Giles, “Efficient sparse matrix-vector multiplication on cache-based gpus,” in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–12.
- [42] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.
- [43] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, “Colpack: Software for graph coloring and related problems in scientific computing,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 1, p. 1, 2013.
- [44] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, “Ordering heuristics for parallel graph coloring,” in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM, 2014, pp. 166–177.
- [45] M. Luby, “A simple parallel algorithm for the maximal independent set problem,” *SIAM journal on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.
- [46] M. T. Jones and P. E. Plassmann, “A parallel graph coloring heuristic,” *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.
- [47] A. H. Gebremedhin and F. Manne, “Scalable parallel graph coloring algorithms,” *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1131–1146, 2000.
- [48] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, “Graph coloring algorithms for multi-core and massively multithreaded architectures,” *Parallel Computing*, vol. 38, no. 10, pp. 576–594, 2012.
- [49] G. Rokos, G. Gorman, and P. H. Kelly, “A fast and scalable graph coloring algorithm for multi-core and many-core architectures,” in *Euro-Par 2015: Parallel Processing*. Springer, 2015, pp. 414–425.

- [50] S. Che, G. Rodgers, B. Beckmann, and S. Reinhardt, “Graph coloring on the gpu and some techniques to improve load imbalance,” in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 610–617.
- [51] A. H. Gebremedhin, F. Manne, and A. Pothen, “What color is your jacobian? graph coloring for computing derivatives,” *SIAM review*, vol. 47, no. 4, pp. 629–705, 2005.
- [52] D. W. Matula, “A min-max theorem for graphs with application to graph coloring,” in *Siam Review*, vol. 10, no. 4. Siam Publications 3600 Univ City Science Center, Philadelphia, PA 19104-2688, 1968, p. 481.
- [53] G. Szekeres and H. S. Wilf, “An inequality for the chromatic number of a graph,” *Journal of Combinatorial Theory*, vol. 4, no. 1, pp. 1–3, 1968.
- [54] R. Diestel, “Graph theory. 2005,” *Grad. Texts in Math*, vol. 101, 2005.
- [55] T. Bjarne, “Colouring, stable sets and perfect graphs,” *Handbook of combinatorics*, vol. 1, p. 233, 1995.
- [56] P. Boldi, A. Marino, M. Santini, and S. Vigna, “Law project,” <http://law.di.unimi.it/>.
- [57] P. Boldi and S. Vigna, “The WebGraph framework I: Compression techniques,” in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.
- [58] P. Boldi, M. Rosa, M. Santini, and S. Vigna, “Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks,” in *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.
- [59] W. Liu and B. Vinter, “Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 339–350.
- [60] H.-V. Dang and B. Schmidt, “Cuda-enabled sparse matrix-vector multiplication on gpus using atomic operations,” *Parallel Computing*, vol. 39, no. 11, pp. 737–750, 2013.

- [61] M. Maggioni and T. Berger-Wolf, “Optimization techniques for sparse matrix–vector multiplication on gpus,” *Journal of Parallel and Distributed Computing*, vol. 93, pp. 66–86, 2016.
- [62] H. Anzt, S. Tomov, and J. Dongarra, “Implementing a sparse matrix vector product for the sell-c/sell-c- σ formats on nvidia gpus,” *University of Tennessee, Tech. Rep. ut-eecs-14-727*, 2014.
- [63] C. Zheng, S. Gu, T.-X. Gu, B. Yang, and X.-P. Liu, “Biell: A bisection ellpack-based storage format for optimizing spmv on gpus,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2639–2647, 2014.
- [64] D. Merrill and M. Garland, “Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, p. 43.
- [65] H. Kaplan, M. Sharir, and E. Verbin, “Colored intersection searching via sparse rectangular matrix multiplication,” in *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*, ser. SCG ’06. New York, NY, USA: ACM, 2006, pp. 52–60. [Online]. Available: <http://doi.acm.org/10.1145/1137856.1137866>
- [66] V. Vassilevska, R. Williams, and R. Yuster, “Finding heaviest h-subgraphs in real weighted graphs, with applications,” *ACM Trans. Algorithms*, vol. 6, no. 3, pp. 44:1–44:23, Jul. 2010. [Online]. Available: <http://doi.acm.org/10.1145/1798596.1798597>
- [67] H. Wang, W. Liu, K. Hou, and W.-c. Feng, “Parallel transposition of sparse data structures,” in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 33.
- [68] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: <http://doi.acm.org/10.1145/2049662.2049663>
- [69] R. R. Amossen, A. Campagna, and R. Pagh, “Better size estimation for sparse matrix products,” in *Proceedings of the 13th International Conference on Approximation, and 14 the International Conference on Randomization*,

- and Combinatorial Optimization: Algorithms and Techniques*, ser. APPROX/RANDOM'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 406–419. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1886521.1886554>
- [70] D. Merrill, M. Garland, and A. Grimshaw, “Scalable gpu graph traversal,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '12. New York, NY, USA: ACM, 2012, pp. 117–128. [Online]. Available: <http://doi.acm.org/10.1145/2145816.2145832>
- [71] M. Harris, S. Sengupta, and J. D. Owens, “Parallel prefix sum (scan) with cuda,” *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.
- [72] S. Sengupta, M. Harris, and M. Garland, “Efficient parallel scan algorithms for gpus,” *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, no. 1, pp. 1–17, 2008.
- [73] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, “Accelerating cuda graph algorithms at maximum warp,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '11. New York, NY, USA: ACM, 2011, pp. 267–276. [Online]. Available: <http://doi.acm.org/10.1145/1941553.1941590>
- [74] D. Merrill and A. Grimshaw, “High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing,” *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011. [Online]. Available: <http://www.worldscinet.com/ppl/21/2102/S0129626411000187.html>
- [75] A. Gamst and W. Rave, “On frequency assignment in mobile automatic telephone systems,” in *Proc. Globecom*, vol. 82, 1982, pp. 309–315.
- [76] G. J. Chaitin, “Register allocation & spilling via graph coloring,” in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 98–105.
- [77] M. Dániel, “Graph colouring problems and their applications in scheduling,” *Periodica Polytech., Electr. Eng.*, vol. 48, no. 1-2, pp. 11–16, 2004.
- [78] M. T. Jones and P. E. Plassmann, “Scalable iterative solution of sparse linear systems,” *Parallel Computing*, vol. 20, no. 5, pp. 753–773, 1994.

- [79] A. H. Gebremedhin, A. Tarafdar, A. Pothén, and A. Walther, “Efficient computation of sparse hessians using coloring and automatic differentiation,” *INFORMS Journal on Computing*, vol. 21, no. 2, pp. 209–223, 2009.
- [80] M. McCourt, B. Smith, and H. Zhang, “Efficient sparse matrix-matrix products using colorings,” *SIAM Journal on Matrix Analysis and Applications*, 2013.
- [81] D. Hysom and A. Pothén, “A scalable parallel algorithm for incomplete factor preconditioning,” *SIAM Journal on Scientific Computing*, vol. 22, no. 6, pp. 2194–2215, 2001.
- [82] S. Yousef, “Iterative methods for sparse linear systems,” *Society for Industrial and Applied Mathematics, Philadelphia*, 2003.
- [83] M. Chiarandini, T. Stützle *et al.*, “An application of iterated local search to graph coloring problem,” in *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, 2002, pp. 112–125.
- [84] D. Chavarria-Miranda, M. Halappanavar, and A. Kalyanaraman, “Scaling graph community detection on the tilera many-core architecture,” in *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, 2014, pp. 1–11.
- [85] H. Lu, M. Halappanavar, and A. Kalyanaraman, “Parallel heuristics for scalable community detection,” *Parallel Computing*, vol. 47, pp. 19–37, 2015.
- [86] L. A. Freitag, M. T. Jones, and P. E. Plassmann, *The scalability of mesh improvement algorithms*. Springer, 1999.
- [87] R. Nasre, M. Burtscher, and K. Pingali, “Morph algorithms on gpus,” in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 147–156.
- [88] T. Mitsuishi, J. Suzuki, Y. Hayashi, M. Kan, and H. Amano, “Breadth first search on cost-efficient multi-gpu systems,” *ACM SIGARCH Computer Architecture News*, vol. 43, no. 4, pp. 58–63, 2016.
- [89] M. Bisson, M. Bernaschi, and E. Mastrostefano, “Parallel distributed breadth first search on the kepler architecture,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2091–2102, 2016.
- [90] G. J. Katz and J. T. Kider Jr, “All-pairs shortest-paths for large graphs on the gpu,” in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. Eurographics Association, 2008, pp. 47–55.

Author's Publications (Published)

- [1] **Anh, Pham Nguyen Quang**, Rui Fan, and Yonggang Wen, “Reducing vector I/O for faster gpu sparse matrix-vector multiplication.” *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pp. 1043-1052, Hyderabad, India, May. 2015.
- [2] **Anh, Pham Nguyen Quang**, Rui Fan, and Yonggang Wen, “Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication.” *Proceedings of the 2016 International Conference on Supercomputing (ICS). ACM 2016*, pp. 36:1-36:12, Istanbul, Turkey, June. 2016.
- [3] **Anh, Pham Nguyen Quang**, Rui Fan, and Yonggang Wen, “Fast, Effective Graph Coloring on GPUs.” (submitted)