Code Optimization on GPUs

Dissertation

Presented in Partial Fulfillment of the Requirements
for the Degree Doctor of Philosophy
in the Graduate School of The Ohio State University

By

Changwan Hong, B.S., M.S.

Computer Science and Engineering

The Ohio State University

2019

Dissertation Committee:

Prof. P. (Saday) Sadayappan, Advisor

Prof. Atanas Rountev

Prof. Radu Teodorescu

ABSTRACT

Graphic Processing Units (GPUs) have become popular in the last decade due to their high memory bandwidth and powerful computing capacity. Nevertheless, achieving high-performance on GPUs is not trivial. It generally requires significant programming expertise and understanding of details of low-level execution mechanisms in GPUs. This dissertation introduces approaches for optimizing regular and irregular applications. To optimize regular applications, it introduces a novel approach to GPU kernel optimization by identifying and alleviating bottleneck resources. This approach, however, is not effective in irregular applications because of data-dependent branches and memory accesses. Hence, tailored approaches are developed for two popular domains of irregular applications: graph algorithms and sparse matrix primitives.

Performance modeling for GPUs is carried out by abstract kernel emulation along with latency/gap modeling of resources. Sensitivity analysis with respect to resource latency/-gap parameters is used to predict the bottleneck resource for a given kernel's execution. The utility of the bottleneck analysis is demonstrated in two contexts: i) Enhancing the OpenTuner auto-tuner with the new bottleneck-driven optimization strategy. Effectiveness is demonstrated by experimental results on all kernels from the Rodinia suite and GPU tensor contraction kernels from the NWChem computational chemistry suite. ii) Manual code optimization. Two case studies illustrate the use of a bottleneck analysis to iteratively improve the performance of code from state-of-the-art DSL code generators.

However, the above approach is ineffective for irregular applications such as graph algorithms and sparse linear systems. Graph algorithms are used in various applications, and high-level GPU graph processing frameworks are an attractive alternative for achieving both high productivity and high-performance. This dissertation develops an approach to graph processing on GPUs that seeks to overcome some of the performance limitations of existing frameworks. It uses multiple data representations and execution strategies for dense- versus sparse vertex frontiers, dependent on the fraction of active graph vertices. Experimental results demonstrate performance improvement over current state-of-the-art GPU graph processing frameworks for many benchmark programs and data sets.

Sparse matrix primitves such as sparse matrix vector multiplication (SpMV), sparse matrix multi-vector multiplication (SpMM), and Sampled Dense-dense matrix multiplication (SDDMM), are key kernels for scientific computing as well as data science and machine learning. A large number of recent research studies have focused on various GPU implementations of the SpMV kernel. But SpMM and SDDMM kernels have received much less attention. This dissertation presents in-depth analyses to contrast SpMV and SpMM, and develops new sparse-matrix representations and computation approaches suited to achieving high data-movement efficiency and effective GPU parallelization of SpMM. It also introduces a novel tiling approach for high-performance implementations for SpMM and SDDMM with the standard sparse matrix representation – Compressed Sparse Row (CSR). Experimental evaluation demonstrates performance improvement over existing implementations.

In short, this dissertation contributes to enhancing compiler technology to achieve high-performance on GPUs by mainly considering data locality and concurrency.

To my parents and brother, for their love and support

ACKNOWLEDGMENTS

First and foremost, I thank my advisor, Professor P. (Saday) Sadayappan, for his guidance and support. I am fortunate to have had the opportunity to collaborate with him. In 2015, he chose to accept me as his advisee; without his selection, supervision, and help, I might have wasted precious time. His passion, brilliance, and dedication have inspired me. Despite his tight schedule, he spent a lot of time working with me and he was enthusiastic to discuss new ideas and give me constructive feedback. Intensive meetings with him enabled me to acquire good research skills; his useful feedback, insight, and comments guided my research and will be the bedrock for my future research.

Special thanks go to my excellent mentors: Professors Fabrice Rastello and Louis-Noël Pouchet. Our conversations were instructive and exhilarating. Their clever and insightful feedbacks/comments played a pivotal role in publishing two PLDI papers. I will remember fondly the pleasant time we shared in Grenoble, France. They also listened to my personal concerns with interest and gave me constructive feedback.

I gratefully thank Doctors Albert Cohen, Aravind Sukumaran-Rajam, and Sriram Krishnamoorthy. In addition to their valuable help with my research, I learned perspectives for sound research from them.

I also would like to thank my candidacy/dissertation committee members, Professors Gagan Aggarwal, Atanas (Nasko) Rountev, Mircea-Radu Teodorescu, and Bhavik Ramesh Bakshi for their penetrating questions and vaulable feedback.

I would like to express my gratitude to all my labmates at OSU; thanks to them, I had a better time during my doctoral studies.

Last but not least, I thank my family for their unqualified support and love; I really respect my family, for whom I will try to be a person who contributes to the world.

VITA

Aug. 2012	B.S., Departmetnt of Computer Science and Engineering, Seoul National University, Korea.
Aug. 2014 - Present	Ph.D. Student, Departmetnt of Computer Science and Engineering, The Ohio State University, USA.
Jan. 2015 - Present	Graduate Research Assistant, Departmetnt of Computer Science and Engineering, The Ohio State University, USA.
May. 2016 - Jun. 2016	Visiting Scholar, INRIA, France.
May. 2018	M.S., Departmetnt of Computer Science and Engineering, Seoul National University, Korea.

PUBLICATIONS

Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, P. Sadayappan
Adaptive Sparse Tiling for Sparse Matrix Multiplication.
Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Feb. 2019.

Israt Nisa, Aravind Sukumaran Rajam, Sureyya Emre Kurt, Changwan Hong, P. Sadayappan
Sampled Dense Matrix Multiplication for High-performance Machine Learning.
IEEE 25th International Conference on High Performance Computing (HiPC), Dec. 2018.

Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, P. Sadayappan
GPU Code Optimization using Abstract Kernel Emulation and Sensitivity Analysis.
Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Jun. 2018.

Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Sureyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Umit V Catalyurek, Srinivasan Parthasarathy, P. Sadayappan
Efficient Sparse-matrix Multi-vector Product on GPUs.
The 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC), Jun. 2018.

Jinsung Kim, Aravind Sukumaran-Rajam, Changwan Hong, Ajay Panyala, Rohit Kumar Srivastava, Sriram Krishnamoorthy, P. Sadayappan
Optimizing Tensor Contractions in CCSD(T) for Efficient Execution on GPUs.
32nd ACM International Conference on Supercomputing (ICS), Jun. 2018.

Jyothi Vedurada, Arjun Suresh, Aravind Sukumaran-Rajam, Jinsung Kim, Changwan Hong, Ajay Panyala, Sriram Krishnamoorthy, V. Krishna Nandivada, Rohit Kumar Srivastava, P. Sadayappan
TTLG - An Efficient Tensor Transposition Library for GPUs.
32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), May. 2018.

Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, P. Sadayappan
Performance Modeling for GPUs using Abstract Kernel Emulation.
Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Feb. 2018.

Sureyya Emre Kurt, Vineeth Thumma, Changwan Hong, Aravind Sukumaran-Rajam, P. Sadayappan
Characterization of Data Movement Requirements for Sparse Matrix Computations on GPUs.
IEEE 24th International Conference on High Performance Computing (HiPC), Dec. 2017.

Wenlei Bao, Changwan Hong, Sudheer Chunduri, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, P. Sadayappan

Static and Dynamic Frequency Scaling on Multicore CPUs.
ACM Transactions on Architecture and Code Optimization (TACO), Nov. 2017.

Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, P. Sadayappan
MultiGraph: Efficient Graph Processing on GPUs.
The 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Sep. 2017.

Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, Atanas Rountev, P. Sadayappan
Resource Conscious Reuse-driven Tiling for GPUs.
The 25th International Conference on Parallel Architectures and Compilation Techniques (PACT), Sep. 2016.

Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, J Ramanujam, P. Sadayappan
Effective Padding of Multidimensional Arrays to Avoid Cache Conflict Misses.
Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Jun. 2016.

Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, P. Sadayappan
Effective Resource Management for Enhancing Performance of 2D and 3D Stencils on GPUs.
Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit (GPGPU), Mar. 2016.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

High Performance Computing Prof. P. Sadayappan

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

Listing **Page**

CHAPTER 1

Introduction

Graphics Processing Units (GPUs) have been spotlighted due to their higher peak performance (GFLOPs) and peak bandwidth between DRAM memory and processor cores than multi-core CPUs. Hence, GPUs have the potential for higher performance than multi-core CPUs both for compute-intensive codes with high operational intensity (e.g., convolutional neural networks for deep learning) and for memory-bandwidth limited applications with low operational intensity (e.g., community detection in large social networks). However, actually achieving the high potential performance of GPUs is not trivial due to several factors: uncoalesced memory access, insufficient concurrency to tolerate high memory access latency, load imbalance, and thread divergence. There are two categories of GPU Applications: regular application and irregular applications. Regular applications present predictable and regular data access patterns, while irregular applications have input-dependent patterns, unpredictable memory-access patterns, and data-dependent control flow. Hence, for optimizing graph applications, SpMM, and SDDMM, which are popular irregular applications, this dissertation develops a tool that exploits regular access patterns and helps application developers implement high-performance applications on GPUs. On the other hand, to optimize popular kernels in irregular applications, it devises optimization techniques based on the input pattern.

Optimizing regular applications: Since achieving high-performance with GPUs is not simple and generally requires significant programmer expertise, such development of high-performance applications is very time consuming even for GPU experts and is not very feasible for the large number of developers. Therefore, there is considerable interest in developing tools that help application developers implement high-performance applications on GPUs.

An important use of performance modeling is to guide code optimization for automated compiler optimization and for manual optimization. However, sufficiently accurate modeling of performance is extremely challenging, so that optimizing compilers generally use extremely simple performance models. The complexity of developing analytical performance models for GPUs is in large part due to the multitude of asynchronously operating hardware components in a GPU. Although several efforts [56, 116, 10, 155] have been made in developing analytical performance models for GPUs, none of them has been demonstrated to provide good accuracy over a range of application codes. Further, none of the previously presented analytical models has been shown to be useful in model-driven code optimization.

This dissertation takes a radically different approach, Sensitivity Analysis via Abstract Kernel Emulation (SAAKE), to develop a performance modeling tool that is accurate enough to guide compile-time optimization in a large space of possible alternatives. Instead of an analytical modeling approach, we use an abstract emulation of the actual binary code for the kernel (SASS code generated by NVIDIA's NVCC compiler). The abstract emuluation involves simulation of the start and completion of each instruction for a set of warps on one Streaming Multiprocessor (SM), but without actual computation of the results of the instructions. Key resources of the system, such as the global memory subsystem and

shared-memory, are modeled in an abstract manner using two parameters: latency and gap (inverse of throughput). The abstract emulation of a collection of warps on one SM simulates inter-dependences between different hardware resources during kernel execution, so that the predicted kernel execution time is sufficiently accurate to guide in the choice of alternate transformations. Since the abstract emulation only models the execution of instructions from a small set of warps on one SM, the time it takes is extremely low: usually in the order of milliseconds for typical GPU kernels. The effectiveness of the proposed approach is demonstrated via experimental results on two GPU platforms, using the entire Rodinia benchmark suite of CUDA programs.

Optimizing graph applications: Graphs algorithms play a crutial role in many applications. However, implementing efficient graph algorithms on GPUs takes a lot of time for developers of new methods for data/graph analytics. Thus, there is substantial interest in developing domain-specific graph processing frameworks that offer application developers a convenient high-level abstraction for developing their algorithms, and high-performance on GPUs. Several such GPU graph processing frameworks have been developed, including VWC [55], Cusha [64], WS [63], GreenMarl [54], Groute [15], and Gunrock [136]. While these frameworks achieve much higher performances than popular vertex-centric graph processing frameworks like Pregel [87], Giraph [124], GraphLab [80], etc., current GPU graph processing frameworks still have some performance limitations.

This dissertation begins by identifying sources of performance limitations for current GPU graph processing frameworks, and then develops the MultiGraph which addresses those limitations. The approaches we develop can be incorporated into existing GPU graph processing frameworks like Gunrock, Groute, WS, etc. A key theme of this work is that

graphs and graph traversals exhibit significant non-uniformity, and therefore a single data representation or execution strategy is unlikely to be effective across the board. By identifying important use cases and execution scenarios, we develop a GPU graph processing framework that uses multiple internal representations of the graph, as well as differentiated execution strategies for different use cases of graph traversal.

Optimizing SpMM with a novel structure: Multi-vector (SpMM) multiplication (also called Sparse Matrix Dense Matrix multiplication or SpMDM) is a key kernel used in a range of applications. SpMV requires a vector to be multiplied by a sparse matrix. SpMM is a generalization of SpMV, and requires multiple vectors to be multiplied by a sparse matrix. While repeated applications of SpMV can be used to perform SpMM, better data reuse can be achieved by devising sparse-matrix representations and access patterns that are customized for SpMM. In examining the distribution of non-zeros in sparse matrices drawn from a variety of application domains, it is found that they are not uniformly spread over the row/column index space, but that non-uniform clustering of elements occurs.

This dissertation exploits this property to partition the elements into two groups: one containing clustered segments, and the other, the remaining singleton elements. Different processing strategies are used for the two partitions, with higher reuse and lower overheads being achievable for the clustered partition.

Optimizing SpMM and SDDMM with CSR: The novel structure mentioned above is incompatible with existing libraries which are used in many applications. To tackle this issue, we use the standard CSR representation, within which intra-row reordering is performed to enable adaptive tiling. Tiling is a key technique for data locality optimization

and is widely used in high-performance implementations of dense matrix-matrix multiplications. However, the irregular and matrix-dependent data access pattern of sparse matrix multiplication makes it challenging to use tiling to enhance data reuse. This dissertation devises and applies an adaptive tiling strategy to enhance the performance of two primitives: SpMM and SDDMM.

Key Contributions: The contributions of the dissertation are presented in four chapters as follows:

- Chapter 2 is related to some of my publications [49, 51, 111, 130, 65, 110]. It presents an approach to GPU kernel optimization by focusing on identification of bottleneck resources and determining optimization parameters that can alleviate the bottleneck. Performance modeling for GPUs is done by abstract kernel emulation along with latency/gap modeling of resources. Sensitivity analysis with respect to resource latency/gap parameters is used to predict the bottleneck resource for a given kernel's execution.

- Chapter 3 is comprised of my publication [52]. It presents an approach to graph processing on GPUs that seeks to overcome some of the performance limitations of existing frameworks. It uses multiple data representation and execution strategies for dense versus sparse vertex frontiers, dependent on the fraction of active graph vertices. A two-phase edge processing approach trades off extra data movement for improved load balancing across GPU threads, by using a 2D blocked representation for edge data.

- Chapter 4 is composed of my publication [48]. It presents an efficient SpMM computer architecture-aware algorithm for GPUs by exploiting the clustering in sparse

matrices. Our approach tries to get good reuse of the elements from the input vector, the output vector, and the sparse matrix simultaneously.

- Chapter 5 is related to my several publications [48, 70, 96, 53, 47]. It devised an adaptive tiling strategy for two popular primitives, SpMM and SDDMM. We used the standard Compressed Sparse Row (CSR) representation which is compatible with other libraries, within which intra-row reordering is conducted for adaptive tiling. This is in contrast to prior studies that have used non-standard sparse matrix representation to improve performance. Tiling is a key technique for data locality optimization, but effective tiling for sparse matrix computations is challenging because of the matrix-dependent data access pattern.

Overall, this dissertation proposes approaches to optimize various kinds of kernels on GPUs by identifying bottleneck resources or developing novel data structures & computer architecture-aware algorithms.

CHAPTER 2

GPU Code Optimization Using Abstract Kernel Emulation and Sensitivity Analysis

2.1 Introduction

Code transformations for optimization are typically guided by performance models. However, we face two significant when optimizing compilers for GPUs:

i) The complexity of modeling the multiple concurrent interacting hardware components in a GPU makes it extremely challenging to develop sufficiently accurate performance models that can reliably predict which of two alternative code structures will execute faster on a given GPU; and

ii) Even if a sufficiently discriminating performance model is developed, the space of semantically equivalent code structures for most compute-intensive algorithms is extremely large when considering the number of possible ways for mapping the statement instances to threads/thread blocks.

Production compilers therefore use simple and fairly imprecise cost models to guide optimizing transformations, in part because very precise performance models usable in compilers are unavailable, and in part because production compilers cannot afford excessively high compile times in performing an extensive search over a large configuration space. But

application developers are willing to wait for minutes or even hours for the "final" compilation of a production code, if the performance of the resulting compiled code can be significantly improved by a slow but effective optimizing compiler. Production compilers do not address this use case; this is the scenario we target in this chapter.

We present a new approach to GPU code optimization with the following features:

- Code optimization is driven by the identification of bottleneck hardware resources.
- Instead of using analytical performance models, performance modeling of GPU kernel execution is done via fast abstract kernel emulation, along with simplified modeling of two key hardware parameters for resources in the GPU: latency and gap (inverse throughput).
- A novel sensitivity analysis with respect to hardware resource parameters is used to identify resource bottlenecks.
- Experimental evaluation of performance modeling is done using all kernels of the Rodinia benchmark suite.
- Utility in code optimization is demonstrated in two ways: i) automated GPU kernel optimization by coupling a bottleneck-analysis-driven search with auto-tuning; ii) two manual case studies of improving the performance of code generated by domain-specific code generators for tensor contractions [82] and stencil computations [110].

2.2 Overview of Approach

In this section, we use examples to present an overview of the new approach to performance modeling and bottleneck identification for GPU kernel execution. We first illustrate some of the key factors that influence GPU kernel performance and the impact of bottleneck resources.

2.2.1 Bottleneck Resources on GPUs

The performance of a GPU kernel is typically constrained by one of its resources, such as the global memory subsystem, shared memory, or the special function unit. We begin with a simple illustration of the limiting bottleneck resource and how kernel performance optimization can be guided by the knowledge of the limiting bottleneck.

Figure 2.1: Grid reshaping

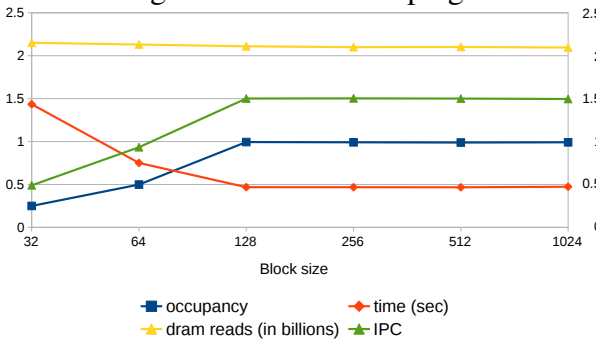
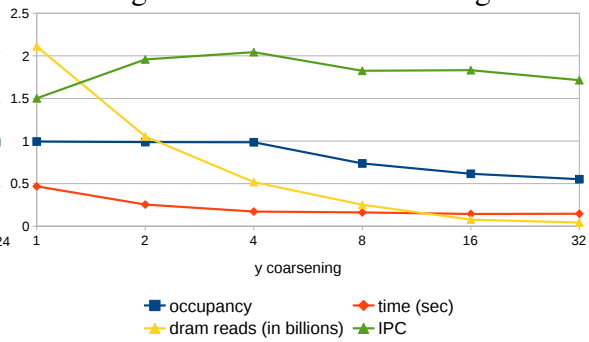


Figure 2.2: Thread coarsening



Listing 2.1: Matrix Multiplication

```
1.  int tx = threadIdx.x, i = blockIdx.x*32 + tx,
        j = blockIdx.y;
2.  __shared__ float cb[32];
3.  float sum = 0.0;
4.  for( int ks = 0; ks < p; ks += 32 ){
5.    cb[tx] = c[ks+tx+pitch_c*j];
6.    for( int k = ks; k < ks+32; ++k )
7.      sum += b[i+pitch_b*k] * cb[k-ks];
8.  }
9.  a[i+pitch_a*j] = sum;
```

Performance under limited concurrency: Fig. 2.1 shows the execution time of a CUDA kernel code for dense matrix-matrix multiplication. The parallel i and j loops of the standard triple-nested loop matrix multiplication code are mapped to threads in the grid, and the innermost loop over k performs a dot product. The CUDA kernel operates on 2048×2048

matrices, using 1D thread blocks, on an NVIDIA K20c GPU. The figure shows occupancy, defined as the ratio of active warps on a Streaming Multiprocessor (SM) to the maximum number of active warps the SM can support. In addition, it reports DRAM accesses, instructions per cycle (IPC), and execution times with varying thread block sizes. We observe that the performance improvements follow the occupancy closely. For a thread block size of 128, an occupancy of one is achieved, delivering the best performance. Since the number of DRAM accesses remains constant for all cases, increasing occupancy helps tolerate the latency of global memory accesses. In this regime (thread block size < 128), the global memory is the resource bottleneck, and performance is limited by memory access latency. As occupancy increases, the code is better able to tolerate global memory latency and performance improves until maximum occupancy is achieved. Further increase in thread block size does not result in any further increase in concurrency, and there is no improvement in performance.

Latency-bound vs. throughput-bound resource: Beyond a thread block size of 128, performance is limited by global memory bandwidth, not by global memory latency. Thus, the same hardware resource can serve as the performance limiting bottleneck in different ways: latency-bound or throughput-bound. When a resource is latency-bound, increasing the number of concurrent requests to that resource can improve performance. When a bottleneck resource is throughput-bound, the only way to improve performance is to reduce the total demand on the resource, as shown below.

Enhancing data reuse by thread coarsening: Thread coarsening [126, 85] is an approach to achieving register tiling for GPU kernel code. Fig. 2.2 shows the performance trends for a thread-coarsened version of the matrix-matrix multiplication code. The total

number of global memory loads gets reduced because the number of thread blocks along y are halved, and each thread performs the same number of global memory load operations. For a thread-coarsening factor of 2, halving the volume of DRAM transactions leads to proportional improvements in the execution time. Execution time and IPC continue to improve until a thread coarsening factor of 4. While thread coarsening increases the per-thread register use, this does not impact the occupancy for a coarsening factor less than four. Beyond a coarsening factor of four is reached, the hardware limit on available registers per SM causes the number of active loaded warps (and thus occupancy) to decrease. At this point, the kernel's performance is once again bound by memory latency. Due to this limitation, further reductions in the volume of DRAM loads do not improve performance.

2.2.2 SAAKE

In this section, we present Sensitivity Analysis via Abstract Kernel Emulation (SAAKE), an approach to identifying the bottleneck resource for a given program version and determining whether the resource is latency- or throughput bound. The objective is to identify the bottleneck resources for a given kernel binary (SASS) code. To this end, we employ sensitivity analysis; i.e., we evaluate the potential performance impact of changing the modeled latency or throughput parameters of a resource. This approach requires performance modeling, motivating our development of a lightweight kernel emulation approach.

Abstract Kernel Emulation: In contrast to analytical performance modeling, we perform an abstract emulation of the actual kernel binary (SASS) code in an emulator. Only a (tiny) fraction of the thread blocks to be executed are emulated, typically taking only a few milliseconds. During abstract kernel emulation, ready instructions from active warps are

scheduled for execution using some warp scheduling strategy, such as Greedy Then Oldest (GTO) [112].

Each primary hardware resource in the GPU is modeled using two parameters: *latency* and *gap*. The *latency* of a resource is the total time for a request to be completed by that resource from start to finish. For pipelined hardware resources, multiple concurrent requests can be in flight, but a new request may not be processable every clock cycle. The minimum number of cycles between the start (or the end) of execution of two successive requests at a resource is its *gap*. Thus, the maximum throughput achievable by a resource is one request per *gap* cycles; i.e., the gap is inversely related to the peak throughput of the resource.

The completion time of an instruction is modeled by adding the latency of the needed resources (e.g., shared-memory, global memory, special functional units, etc.). For each resource, an "earliest admission" time is maintained, which is incremented by its gap when a new request starts execution on it. Dependences between instructions are tracked, so that the earliest schedulable time for an instruction is later than the completion times of all previously scheduled instructions on which it depends.

Bottleneck identification: To detect resource bottlenecks, we use sensitivity analysis with respect to resource model parameters. This is done by performing multiple kernel emulations using modified resource parameters for latency and gap. First kernel execution time is modeled using resource model parameters that correspond to the target GPU, determined via use of microbenchmarks. Next, kernel emulation is repeated with one resource parameter changed, say increase modeled latency of global memory by 10%. Kernel

emulation is then performed by changing the global memory gap parameter by 10%. Similarly, emulations with modified parameters for each of the hardware resources is performed, with one resource parameter modified for each emulation. The hardware resource with the largest relative change in predicted kernel execution time (over the base time) is identified as the bottleneck resource. Further, whether the large change occurred with the change of latency or gap parameter points to whether the resource is latency-bound or bandwidth-bound.

Sometimes, no single resource parameter may stand out as the clear bottleneck, with multiple resource parameters exhibiting comparable and moderately high sensitivity. In this case, it is possible that different portions of a GPU kernel are constrained by different bottlenecks. The sensitivity analysis can then be performed over different portions of the kernel, as illustrated later through a case study on optimizing a tensor contraction kernel.

2.3 Bottleneck Identification via Sensitivity Analysis

In this section, we present the key idea behind the proposed approach to GPU kernel optimization. To illustrate the technique, we restrict our study to a simple analytical performance model for a *single* pipelined hardware resource such as an arithmetic functional unit in one of the SMs of a GPU. More realistic scenarios of complex kernels using multiple asynchronous pipelined units are discussed in the next section.

2.3.1 Resource parameters

A pipelined hardware resource is modeled by two fundamental performance parameters: *latency* and *gap*. *Latency* (denoted L) is defined as the number of cycles an operation

waits before execution, when it depends on the immediately preceding operation.[1] When two operations depend on one another, their starting time must be separated by the latency of the first. The *throughput* is the number of operations a resource can issue (equivalently complete) per cycle. The *gap* (denoted G) is the inverse of the throughput. Because the resource is pipelined, its *gap* is lower than its *latency*.

2.3.2 Modeling performance impact of concurrency

The total *overall concurrency* (denoted C) in a warp's execution is the product of warp-level parallelism (WLP) and instruction-level parallelism (ILP) within each warp. Consider a repetitive loop, where instructions of the current iteration only depend on results computed by the corresponding instruction in the previous iteration. The number of iterations in the loop is denoted by P, the *number of phases*.

For this simple scenario, the total execution time can be modeled as shown in Fig. 2.3. There are two cases:

Latency-limited execution ($L > C \times G$): Fig. 2.3 (left) depicts the execution in this case. In the first phase, successive instructions can only be issued once every G cycles since they all need the same pipelined hardware resource. Since $L > C \times G$, the first instruction of each phase can be issued L cycles apart. The entire execution is completed after $T = L \times P + (C - 1) \times G$ cycles.

Throughput-limited execution ($L \leq C \times G$): Fig. 2.3 (right) illustrates the scheduling of operations in this case. Although the first instruction of the second phase has its input operand ready after L clock cycles (satisfying the dependence on the corresponding instruction from the first phase), the gap constraint means that it cannot be issued until

[1]Typically, latency is defined as the number of cycles for an operation to complete. We use a modified definition to account for architectural features such as pipeline forwarding.

$C \times G$ cycles after the first instruction of phase one. The total completion in this case is $T = L + (C \times P - 1) \times G$ cycles.

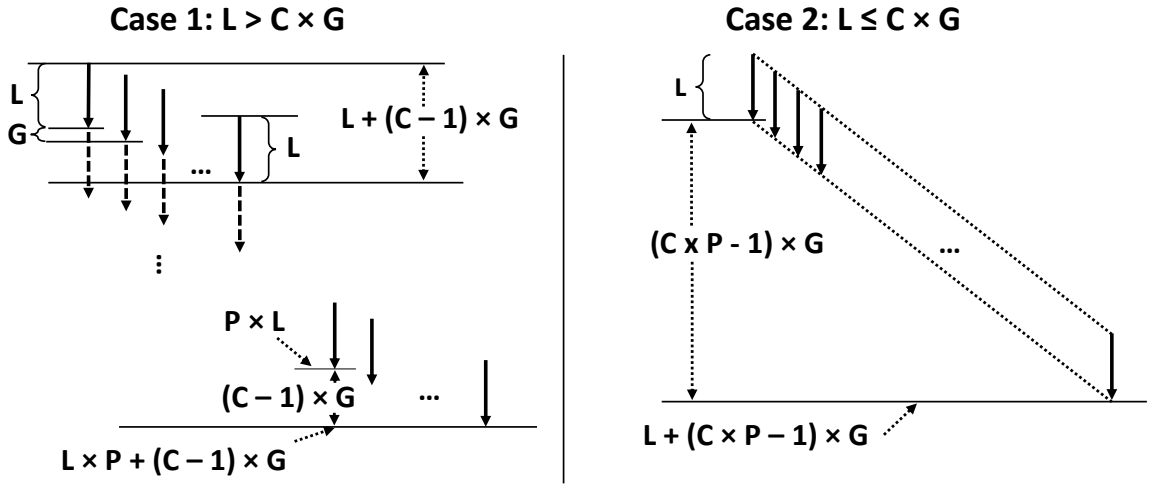


Figure 2.3: Illustration of latency-limited (left) and throughput-limited (right) execution

2.3.3 Sensitivity analysis for bottleneck identification

In this section, we perform sensitivity analysis by observing the fractional increase/decrease of execution time ($\frac{\Delta T}{T}$) for a given fractional increase/decrease of a resource parameter ($\frac{\Delta L}{L}$ for *latency* and $\frac{\Delta G}{G}$ for *throughput*) - equivalently, comparing $\frac{\Delta T}{\Delta L} \times \frac{L}{T}$ (resp. $\frac{\Delta T}{\Delta G} \times \frac{G}{T}$) with zero. We use ΔX notation here to reflect the discrete property of resources. A continuous form of the analytical expressions for our example can be obtained as follows:

	$\frac{\delta T}{\delta L} \times \frac{L}{T}$	$\frac{\delta T}{\delta G} \times \frac{G}{T}$
Latency-Limited	$1/\left(1 + \frac{(C-1)G}{LP}\right)$	$1/\left(1 + \frac{LP}{(C-1)G}\right)$
Throughput-Limited	$1/\left(1 + \frac{(CP-1)G}{L}\right)$	$1/\left(1 + \frac{L}{(CP-1)G}\right)$

Consider the *latency*-limited case. As the larger L is compared to $C \times G$, the sensitivity expressions ($\frac{\delta T}{\delta L} \times \frac{L}{T}$) (based on *latency*) and $\frac{\delta T}{\delta G} \times \frac{G}{T}$ (based on *gap*) become one and zero,

respectively. We can derive symmetric conclusions for the *gap*-limited case, leading to the following rules:

	$\frac{\delta T}{\delta L} \times \frac{L}{T}$	$\frac{\delta T}{\delta G} \times \frac{G}{T}$
Latency-limited	$\gg 0$	≈ 0
Throughput-limited	≈ 0	$\gg 0$

We use the above observations to identify the mode of bottleneck (latency vs. throughput). For more complex kernels and multiple GPU resources, accurate analytical modeling to identify bottleneck resources is extremely challenging. However, sensitivity analysis with respect to each resource is feasible by performing multiple abstract kernel emulations with changed resource parameters, as described in the next section. When analyzing the sensitivity of multiple resources, non-bottleneck resources will show little change in sensitivity with respect to both *latency* and *gap*. The resource that exhibits the largest sensitivity analysis metrics is identified as the bottleneck resource.

2.4 Abstract Kernel Emulation

This section details the approach to abstract kernel emulation. The algorithm uses the *latency* and *gap* parameters for each resource in the target GPU architecture. Due to space constraints, we do not discuss how these parameters are obtained; however, we provide some details in a technical report [50]. Our approach is similar to the one described by Papadopolou [103].

2.4.1 Overview

Fig. 2.4 illustrates the approach to kernel emulation. A maximal number of warps that can concurrently occupy one SM of the GPU is modeled. Each warp is modeled by a current-instruction pointer and an earliest-schedule-time for the current instruction. Each modeled hardware resource is associated with a latency and gap parameter. Each resource

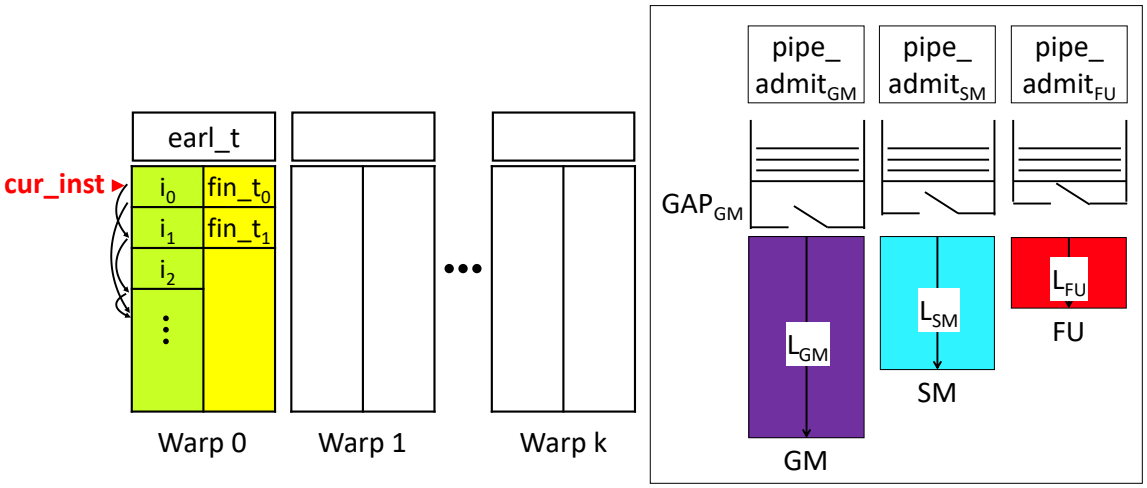


Figure 2.4: Overview of abstract kernel emulation

is essentially treated as having an unbounded input queue for requests, with one waiting request being allowed to enter the resource every *gap* units of time. The state of each hardware resource is maintained by a *pipe_admit* time, which represents the earliest time at which a new request to that resource can enter that resource pipeline. The current instruction of a warp is eligible to be scheduled if all needed operands are ready. When an instruction is scheduled, the *pipe_admit* of the needed hardware resource is the time at which the processing of the operation will begin. Adding the *latency* of the resource determines when that instruction will be completed, which is recorded in the *fin_t* entry associated with that instruction. Within each warp, intra-instruction dependences are tracked. When a new instruction from some warp is scheduled, the availability status of its operands at that time is known, since the finish-times of all preceding instructions have been recorded in the *fin_t* entries of the producer instructions.

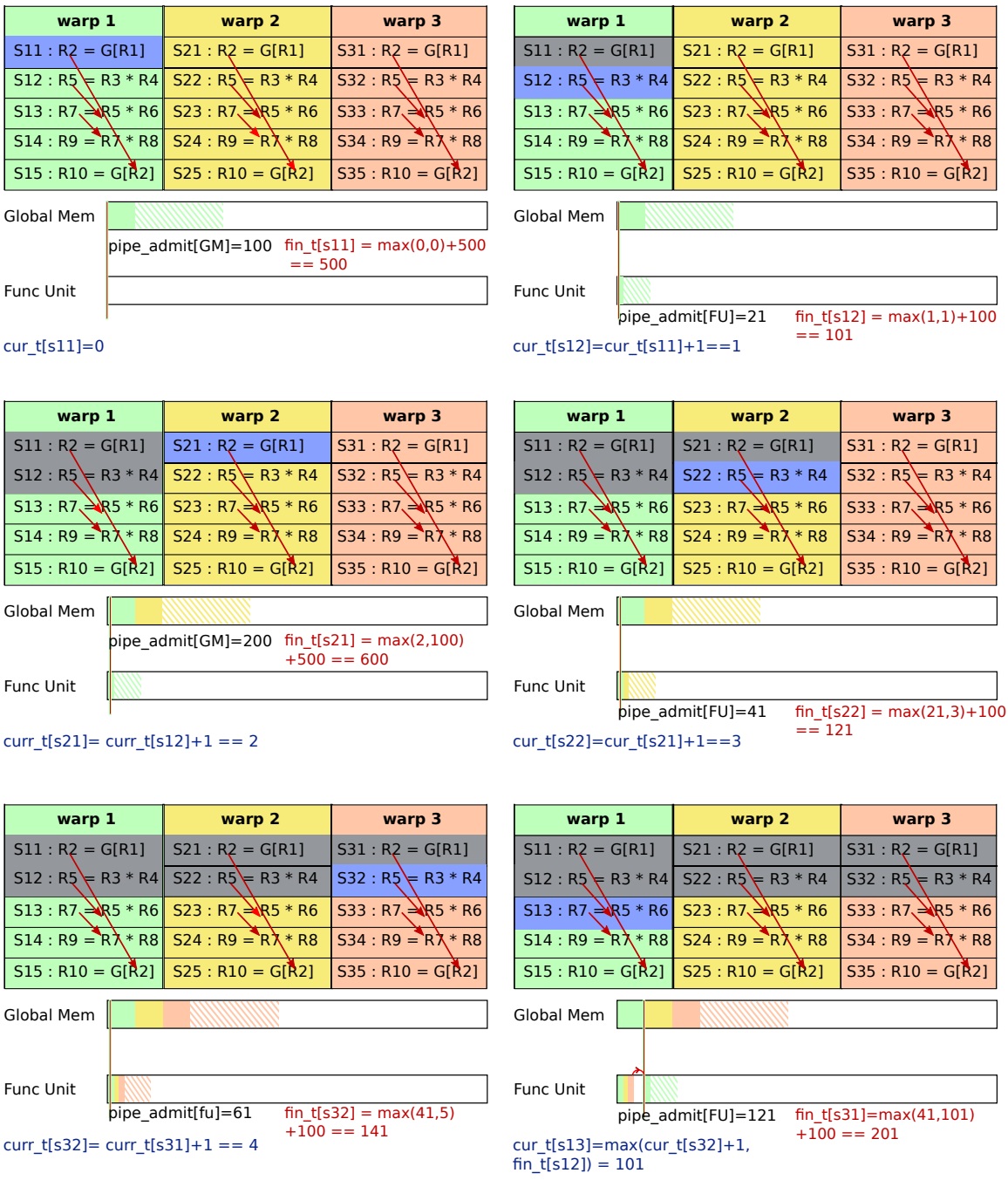


Figure 2.5: Illustration of abstract kernel emulation

2.4.2 Illustrative Example

Consider an SM with the following resource parameters: global memory latency = 500; global memory gap = 100; functional unit latency = 100; functional unit gap = 20. Figure 2.5 shows different stages of abstract kernel emulation with three warps. Initially, statement $S11$ in the first warp reads a value from global memory into a register. Since $S11$ is not dependent on any other instruction and the global memory is ready to accept/admit a request, the statement can be issued immediately. Once $S11$ is issued, global memory requests are blocked for 100 cycles (the *gap* parameter of the global memory resource). $S11$ will require 500 cycles (global memory *latency*) to finish its execution. The next instruction, $S12$, is an add instruction requiring the functional unit. Because $S12$ does not depend on any previous instruction and the functional unit is idle, it can be scheduled at clock cycle 1. $S12$ will require 100 cycles to complete execution. The next instruction of warp 1, $S13$, is dependent on $S12$. Since $S12$ will only be completed at clock cycle 101, $S13$ cannot be scheduled immediately and we move to warp 2. $S21, S22, S31$, and $S32$ follow the same pattern as $S11$ and $S12$. $S32$ is scheduled in clock cycle 4 and completes execution in clock cycle 141. At clock cycle 5, no warp has an instruction that can be scheduled. The next instruction that can be scheduled is $S13$ at clock cycle 101. The clock is advanced to step 101 and $S13$ is selected for scheduling.

The abstract kernel emulation is based on a Greedy-Then-Oldest (GTO) scheduling policy [112]. We use GTO because the actual warp scheduling policy used in NVIDIA GPUs is not publicly known. Alternative scheduling policies can easily be incorporated into the emulator.

2.4.3 Emulation Algorithm

Alg. 1 outlines the abstract emulation algorithm to predict the kernel execution times. Several details, such as handling of conditional statements, L1/L2 cache misses, and uncoalesced global memory access, are not included in this high-level pseudocode; these issues are discussed at the end of this sub-section. Modern GPUs have multiple warp schedulers [140, 139] and each warp scheduler can issue more than one independent instruction from the same warp in a single clock cycle. For simplicity, Alg. 1 models a single warp scheduler that schedules a single instruction per cycle. However, the implementation of the abstract kernel emulator models the actual number of warp schedulers and the number of instructions scheduled per cycle per warp scheduler.

Abstract kernel emulation begins by scheduling the first instruction from warp 0 and proceeds until all instructions from all warps are scheduled. At each step, a warp with a ready instruction is selected. For a warp's current instruction (*cur_inst*), the earliest scheduling time (*earl_t*) is determined as the maximum among the finish times of its predecessors and the current clock (lines 5-7). If the current instruction cannot be scheduled at the current clock, then the warp whose instruction has the smallest earliest_schedule time is sought (line 9) and the clock is advanced (line 10). If the current instruction can be scheduled in the current clock cycle, it is scheduled for execution. A scheduled instruction may not be executed immediately, but might be enqueued if the needed hardware resource is not available. Thus, the actual time at which an instruction begins execution is the maximum of the current clock and the clock cycle at which the corresponding resource r is ready to accept a request (*pipe_admit*). An instruction's finish time (*fin_t*) is the sum of the time at which its execution begins and the latency of the corresponding resource. Once a resource accepts a request, it is unavailable to accept another request for *gap* cycles: The

Algorithm 1: Abstract kernel emulation

input : Number of warps in one block, number of warps in one SM, *latency* and *gap* of each resource
output: Estimate of execution time, utilization of resources

1 ***t*:** current time, ***w*:** current warp, ***r*:** current resource, ***i*:** current instruction
2 **fin_t**[w][i]**:** finish time of instruction i in warp w
3 **cur_inst**[w]**:** current instruction of warp w (initialized to 0)
4 **pipe_admit**[r]**:** earliest time when the next instruction can be admitted to the pipeline of resource r (initialized to $-\infty$)
5 **start**[r]**:** time at which the last instruction that uses r was issued
6 **utilization**[r]**:** total time resource r was active (initialized to 0)
7 **latency**[r]**:** latency of resource r, **gap**[r]**:** gap of resource r
8 $p \rightarrow q$: instruction q is dependent on instruction p
9 **earl_t**[w]**:** earliest schedule time of the current instruction of warp w (initialized to 0)
10 **end**: non executable last instruction of each warp

11 $t \leftarrow 0$
12 $w \leftarrow$ warp 0
13 **while** $\exists x$ ***s.t.*** *cur_inst*[x] $\neq$ *end* **do**
14 $\quad i \leftarrow$ cur_inst[w]
15 $\quad$ **if** $i \neq$ *end* **then**
16 $\quad\quad$ earl_t[w] $\leftarrow \max(\{\text{fin\_t}[w][p] \mid p \rightarrow i\},\ t)$
17 $\quad$ **end**
18 $\quad$ **else** earl_t[w] $\leftarrow \infty$;
19 $\quad$ **if** *earl_t*[w] $> t$ **then**
20 $\quad\quad$ **find** w **s.t.** earl_t[w] = $\min_x$ earl_t[x]
21 $\quad\quad$ $t \leftarrow$ earl_t[w]
22 $\quad\quad$ **continue**
23 $\quad$ **end**
24 $\quad r \leftarrow$ resource(i)
25 $\quad$ fin_t[w][i] $\leftarrow \max(t, \text{pipe\_admit}[r]) + \text{latency}[r]$
26 $\quad$ pipe_admit[r] $\leftarrow \max(\text{pipe\_admit}[r], t) + \text{gap}[r]$
27 $\quad$ utilization[r] $\leftarrow$ utilization[r] + $\min(t - \text{start}[r],\ \text{latency}[r])$
28 $\quad$ start[r] $\leftarrow t$
29 $\quad$ cur_inst[w] $\leftarrow$ cur_inst[w] + 1
30 $\quad t \leftarrow t + 1$
31 **end**
32 **return** ($\max_w$ fin_t[w][cur_inst[w] $-$ 1]), utilization[]

corresponding resource's pipe_admit time is updated by adding *gap* (line 14). On line 15, *start* represents the time at which the previous instruction (the one before i) got issued on

resource r. If *latency* is smaller than the elapsed time in ($t - start$), it means r was idle during the remaining time ($t - start - latency$). The utilization of resource r is updated by adding only the time during which it was active (line 15). Finally, the instruction pointer of the current warp is updated (line 17) and the clock is updated (line 18). This process is repeated until all instructions from all warps are scheduled. After all of the instructions are scheduled, the clock is set to the latest finish time of the last completed instruction among all warps (argument 1 of line 19).

The kernel emulation models the execution of a maximal set M (which is determined based on the resource requirements of each thread block) of thread blocks that can concurrently be executed on an SM. The number of needed "phases" to execute all thread blocks of a kernel is modeled by dividing the total number of thread blocks by the product of M and the number of SMs in the GPU. Thus, the total emulation time is generally a small fraction of the actual execution time of compute-intensive kernels.

2.4.4 Additional Details

We now present some additional details regarding performance modeling via abstract kernel emulation that were omitted in Alg. 1.

Conditional Statements: For if-then-else conditions and loop bounds that are dependent on known parameters, emulated instructions reflect actual executed instructions [137]; for loops with unresolvable loop bounds, representative trip counts are provided to the emulator; for if-then-else conditions dependent on dynamically computed values, all predicated control paths are conservatively emulated in lexical order of the SASS code.

Irregular/uncoalesced data access: This can be identified by static analysis, but was manually identified since a static analysis for it has not been implemented. A full DRAM transaction request is conservatively assumed to occur for each load/store from every thread in a warp for uncoalesced accesses. In contrast, for coalesced accesses only one DRAM transaction is scheduled for the entire warp.

L1/L2 cache: The cache is not explicitly emulated. However, modeling accuracy can be enhanced by providing the emulator an optional cache miss-rate parameter for a region of the code. Although no cache simulation is performed in the emulator, the availability of actual or estimated cache miss ratio is used in the following manner in the emulation. An additional L2 cache resource is added to the abstract emulation. For every emulated global memory access instruction, its execution is modeled by randomly assigning it to either the modeled L2 cache resource (with lower gap and latency parameters) or the global memory resource (with higher gap and latency parameters). The probability of assigning the execution of the load to the modeled L2 resource is the provided cache miss rate. In the next subsection, we present experimental results that demonstrate the improvement in the accuracy of performance prediction from such modeling.

Barrier synchronization: Every warp in a block is stalled at barrier synchronization statements until all instructions in the block finish execution.

Bank conflicts: For cases where indices of shared memory are dependent on known parameters, bank conflict can be emulated. Register bank conflicts can also be emulated as described in the literature [71, 154]. Register bank conflicts are sensitive to register names that can be extracted from the SASS codes. On the Kepler GPU, there are four register banks for each thread [71, 154] and operands can read only one value from each register bank per cycle.

Atomic operations: Similar to the handling of "if" statements and bank conflict, if indices of array atomic operations are dependent only on known parameters, atomic operations can be emulated.

Instruction queue: Instruction queue lengths are obtained using the approach developed by Nervana [95].

2.4.5 Experimental Evaluation of Prediction Accuracy

We next present results from experimental evaluation of the prediction accuracy of the abstract kernel emulation on two GPU systems: an NVIDIA K20c with 13 Kepler SMs, 5GB global memory, 706MHz, 1.25MB L2 cache, and 48KB shared memory, and an NVIDIA Titan X with 28 Pascal SMs, 12GB global memory, 1417MHz, 4MB L2 cache, and 96KB shared memory. We evaluate *all* of the 58 kernels in the Rodinia benchmark suite. These kernels collectively include 369 if-then-else conditions and loop bounds dependent on known parameters, 155 if-then-else conditions dependent on dynamically computed values, and 11 loops with statically unresolvable loop bounds.

Each kernel's binary was extracted and the SASS code [98] was subjected to abstract emulation, using the respective parameters for both targeted GPUs. The benchmarks were also executed to measure actual execution times. For each kernel in each benchmark, the prediction error, $\frac{pred-actual}{actual}$, is shown in Fig. 2.6. SAAKE models the time required for executing a maximally concurrently loadable set of thread blocks on one SM. This time is then scaled based on the actual number of thread blocks and number of SMs on the GPU to predict the entire kernel execution time.

The plots show two sets of data, with and without including L2 cache effect modeling described in subsection 2.4.4. As explained earlier, the L2 cache miss rate parameter can

	Kepler		Pascal	
	w/o cache	w/ cache	w/o cache	w/ cache
(geo) mean	16.9%	11.8%	16.4%	13.7%
median	20.2%	13.8%	21.9%	13.2%

Table 2.1: Geometric mean and median of error(=$\left|\frac{predicted-actual}{actual}\right|$) in Fig. 2.6

be obtained by incorporating a cache miss prediction model, or, as done here, by actual measurement. For both machines, the execution time is predicted quite well for a majority of the kernels, especially when cache modeling is included. We note that gathering cache miss data from hardware counters to use in the emulator is well justified, since it is very quick and the ultimate use of the kernel emulation is not just to predict kernel execution time, but to identify hardware bottlenecks and use that information in code optimization, as presented in the following sections. We are unaware of any automatable GPU performance modeling approach that has demonstrated a comparable level of prediction accuracy across such a wide range of kernels. Aggregated accuracy metrics over the full set of kernels is presented in Table 2.1. The accuracy for many of the benchmarks can be further improved with enhancements to the emulator. Below we elaborate on the reasons for a relatively high error for some benchmarks and how some of those errors can be lowered.

SAAKE conservatively considers all branches of conditional expression to be executed. However, in kernels like BFS, only a few data-dependent branches are executed. The effectiveness of coalescing is another factor that affects performance. For codes with indirect-memory access, the number of actual required DRAM transactions (effectiveness of coalescing) cannot be statically analyzed in general. The serialization effect of atomic operations on data-dependent memory locations cannot be estimated statically. Hence, SAAKE

assumes that the atomic operations are carried out on different memory locations. In huffman/histo_kernel, the instruction "atomicAdd(&temp[buffer[i]], 1);" depends on the data, and the values of buffer[i] are largely 116 or 129. This results in highly significant serialization overhead, which explains the low prediction accuracy. When the kernel has very few instructions, the achieved occupancy is lower than the computed occupancy (obtained using [29]) which affects the prediction accuracy. Further, if the total execution time of a kernel is less than five micro-seconds (e.g., huffman/uniformAdd), then kernel launch overhead dominates the kernel execution time, and this affects prediction accuracy.

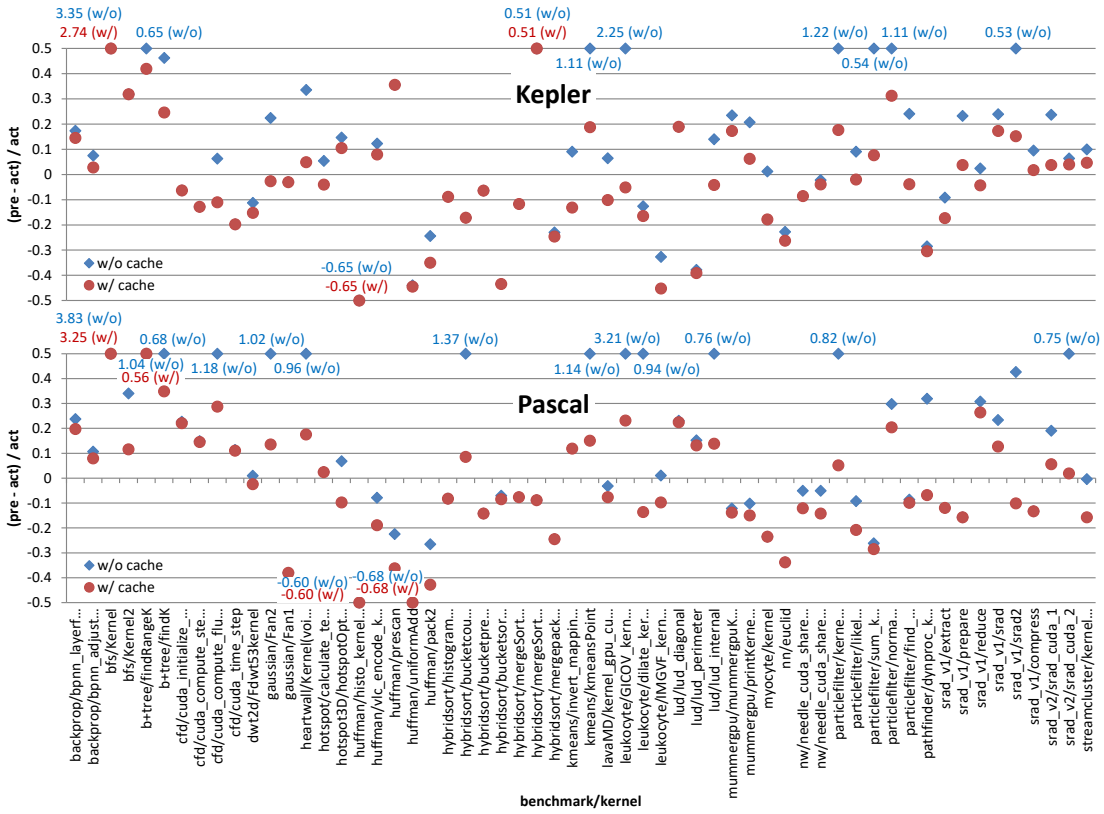


Figure 2.6: Performance modeling accuracy with Rodinia benchmarks (top: Kepler, bottom: Pascal)

2.4.6 Limitations

The evaluation using Rodinia benchmark kernels shows good prediction accuracy for a large number of kernels, but also very high errors for some kernels. Modeling error can be high when data-dependent effects have a significant impact on execution time. This is because the efficiency of the kernel emulation approach derives from the fact that actual simulation of all operations is not performed, but abstracted kernel emulation is performed for only a tiny fraction of all thread blocks of a GPU kernel. The following types of kernels can incur high modeling errors:

- Kernels where a non-trivial fraction of global memory accesses are actually returned from cached data in L2 cache: since the actual execution of instructions is not modeled, the best we can do is to randomly model a global memory access as returning from cache or global memory based on estimated/known cache miss rates. Further, multi-level memory hierarchy is not modeled and can result in high error rates for some kernels.
- Kernels with significant thread divergence: Since conditional expressions in kernel code are not actually simulated, codes with significant data-dependent conditional execution can result in high modeling errors.

Despite the above limitations to prediction accuracy, there are many practical scenarios where the abstract emulation approach is sufficiently accurate. An important use case is that of domain-specific code generators that use tiling and effective use of shared memory. With such codes, data is explicitly copied from global memory to shared memory and reused multiple times from shared memory. Frequently repeated accesses to data in global memory does not occur with such kernels and the prediction accuracy from abstract kernel emulation tends to be high. We demonstrate two such use cases later in this chapter.

2.5 Model-Driven Search and Enhanced Auto-Tuning

In this section, we describe how a search directed by bottleneck-analysis with SAAKE can be coupled with auto-tuning systems like OpenTuner [6] that use ensemble search techniques and allow inclusion of custom search strategies.

2.5.1 Model-driven Search

The essential idea behind SAAKE's model-driven search is to start from an initial configuration and iteratively move along the canonical search-space directions by either reshaping, thread coarsening, block coarsening, or unrolling, until a termination condition is reached. The decision on which direction to move is driven by the most significant bottleneck of the current configuration. Once the direction of a move is decided, the distance of a move (multiplicative factor) is the amount that enables maximum expected alleviation of the bottleneck (e.g., moving from GM latency-bound to SM throughput-bound). This distance is evaluated by extrapolating (linearly) the impact of the coalescing on the resource usage (reuses, concurrency).

The *utilization* of a resourcei and the time *prediction* are computed using the abstract kernel emulation (Algorithm 1). Both the abstract emulation and the computation of the *occupancy* rely on NVCC to compile the current configuration.

Alg 2 and Alg 3 describe the search algorithm used to traverse the configuration space. Alg 2 provides an overview of the algorithm. It uses Function *next_best* to find the best move (Algorithm 3). SASS code for the original baseline CUDA code is first passed to the bottleneck analyzer to predict the execution time (line 6). Based on the resource usage, the bottleneck analyzer also computes the achievable occupancy.

Algorithm 2: Main Search Algorithm

prediction(c): predicted execution time for configuration c
occupancy(c): occupancy of configuration c
***ustrides*:** set of all canonical vectors along which we want the configuration to evolve (multiplicative factors. Set to $(2)^{3d+1}$ with d the grid dimension)
input : Original code
output: Predicted best coarsening configuration

```
function entry()
    curr = init_config
    exec ← prediction(curr)
    occu ← occupancy(curr)
    (best_exec, best_curr) ← (exec, curr)
    (prev_exec, prev_curr) ← (∞, ×)
    while True do
        curr′ ← next_best(curr, ustrides)
        exec′ ← prediction(curr')
        occu′ ← occupancy(curr′)
        if occu′ < occu then
            if best_exec ≥ prev_exec then
                break
            end
            (prev_exec, prev_curr) ← (best_exec, best_curr)
            (best_exec, best_curr) ← (∞, ×)
        end
        if exec′ < best_exec then
            (best_exec, best_curr) ← (exec′, curr′)
        end
        occu ← occu′
    end
    return prev_curr
```

For bottleneck-guided traversal of the configuration space, the profitability of moving along each dimension is assessed. This is done by using unit stride analysis. Unit stride analysis is performed by moving a "unit" distance along each dimension of the configuration space, one at a time, starting from the initial configuration. For each such unit stride configuration, the global memory and shared memory traffic are estimated from abstract kernel emulation and saved in a table.

Algorithm 3: Search Step Algorithm

$0 \leq U_{GM} \leq 1$**:** Global memory utilization factor
$0 \leq U_{SM} \leq 1$**:** Shared memory utilization factor
***conf*:** current configuration
#T(c**):** number of threads in a thread block for configuration c
#TB(c**):** number of thread blocks in an SM for c
maxT: hardware thread limit in an SM
occupancy(c**):** occupancy of c (#T(c) $\times$ #TB(c)/maxT)
***olist*:** list of occupancies
utilization$_r(c)$**:** Utilisation factor of configuration c for resource r
transactions$_r(c)$**:** number of transactions to resource r in c
bottleneck$(c,\ o)$**:** bottleneck of configuration c with occupancy o (from Δ-analysis)
***bott*:** main bottleneck of the current configuration
***ustrides*:** set of all canonical vectors along which configuration may evolve (multiplicative factors)
***s*:** canonical vector of multiplicative factor
K**:** scalar multiplicative factor
#regs: total number of registers per SM
maxregs: maximal number of registers per thread for the chosen occupancy

input : *conf*: Current configuration
ustrides
output: Predicted next best configuration

function ***next_best***(*conf*, *ustrides*)
  bott $\leftarrow$ bottleneck(*conf*, occupancy(*conf*))
  if *bott.type* = *Latency* **then**
    $r \leftarrow$ *bott*.resource
    find $s \in$ *ustrides* **that maximizes** utilization$_r$(*conf* $\otimes$ s)
    $\Delta U_r \leftarrow$ utilization$_r$(*conf* $\otimes$ s) $- U_{GM}$
    find min K **s.t.** $U_{GM} + \Delta U_r(K-1) \geq 1$
    conf $\leftarrow$ *conf* $\otimes$ $K.s$
  end
  else `// throughput-limited`
    olist $\leftarrow \{k \times$ #TB(c)/maxT $\mid\ 1 \leq k <$ #T(c)$\}$
    find biggest *occu* **s.t.** bottleneck(*conf*, *occu*) $\neq$ *bott*
    bott $\leftarrow$ bottleneck(*conf*, *occu*)
    maxregs $\leftarrow \lfloor$*#regs*/(*occu* $\times$ maxT)$\rfloor$
    configs $\leftarrow$ gen_configs(*maxregs*, *occu*)
    $r \leftarrow$ *bott*.resource
    find *conf* $\in$ *configs* **that minimizes** transactions$_r$(*conf*)
  end
  return *conf*

To navigate the configuration space, the search algorithm uses the bottleneck analyzer to predict the bottleneck and moves in a direction that will alleviate the bottleneck. Given the current configuration, Alg 3 is used to predict the next configuration to be chosen.

This step is explained in detail in the following paragraphs. For the chosen configuration, the occupancy and predicted execution time are estimated using the bottleneck analyzer. If the best predicted execution time among configurations evaluated by Alg 2 at the currently tested occupancy is lower than that at the previous occupancy, the best configuration at the current occupancy replaces any previous selection as the current best configuration seen so far and the traversal in that direction is continued. On the other hand, if the best configuration among all evaluated cases at the currently tested occupancy has a higher predicted time than the best recorded one at the previously tested occupancy, the search is terminated and the previously save best configuration is returned.

Given one configuration, Alg 3 describes the search algorithm to select the next configuration. The first step is to determine the current configuration's bottleneck using sensitivity analysis (line 2). If the current configuration is limited by the latency of a resource, concurrency is increased to enhance latency tolerance. This is done by coarsening along a direction that will increase ILP for that resource. If there is data sharing across threads/blocks, coarsening helps reduce the total volume of traffic to the latency-bound resource. However, coarsening may increase the resources required per thread (registers, shared-memory), possibly resulting in a reduction in warp-level parallelism. However, by coarsening, the increased ILP and reduced data traffic may help improve performance. The traversal direction is chosen to be the one with maximum reduction in data traffic for that particular resource. For this, results from unit stride analysis are used. The total stride of the move is chosen as the minimum number that would result in full utilization of the bottleneck resource (line 7).

On the other hand, if the resource is throughput-limited, the total number of transactions to that resource is reduced. This can be done by coarsening, if different threads/blocks share data. Since coarsening may reduce WLP, the amount of occupancy that can be sacrificed without dropping performance is determined. This is done by using abstract kernel emulation for the current configuration, for various occupancies lower than the current occupancy. The occupancy is lowered till the bottleneck changes. Then the amount of additional resources gained by lowering the occupancy is determined. The higher the coarsening factor, the better the reuse of shared data elements. The configuration that has the highest reduction in data volume is selected that can still achieve the minimum required occupancy.

2.5.2 Coupling with Auto-tuning

OpenTuner is a general framework for auto-tuning, which uses an ensemble of techniques to navigate the search space. Initially, a random seed configuration is determined and different search techniques are invoked. The resulting configurations are run by OpenTuner to identify the best one. Each technique is then allotted a time slot proportional to the quality of the result it produces. Mechanisms are provided to communicate through a common database to enable cooperation among different techniques. We coupled SAAKE with OpenTuner. In coupled-mode, SAAKE is invoked just before OpenTuner begins its initial search.

SAAKE starts with the base configuration (i.e., the initial code); the predicted configuration is compiled (not run) to generate an SASS code. The SASS code is then analyzed to determine the bottleneck and this is used to predict the next configuration. This process is repeated until SAAKE finds a configuration which is best according to its model-driven

search. We note that no measured metrics like cache hit rates were provided to SAAKE for these experiments. The final configuration from SAAKE is used as the initial seed for OpenTuner. OpenTuner then proceeds with its ensemble of techniques to find the best configuration. In our experiments, in most cases, the output of SAAKE is either the optimal one or very close to the optimal, which significantly reduces the time required for OpenTuner to find the best configuration.

2.5.3 Experimental Evaluation

We evaluated the effectiveness of the bottleneck-guided optimization approach on the entire set of Rodinia benchmark kernels. We compare three scenarios: i) SAAKE only, ii) OpenTuner only, and iii) SAAKE coupled with Opentuner. Fig. 2.8 presents the results for Kepler and Pascal GPUs. The vertical bars show the achieved speedup over the base Rodinia kernel (scale is linear, shown on the left). Each benchmark has a pair of bars: striped bar for SAAKE-only, and solid bar for SAAKE+OpenTuner. Roughly half of the kernels achieve some speedup by changing the original kernel. In about half of those, SAAKE by itself achieves the maximum possible performance improvement, and gets a good fraction of achievable speedup for most of the others. The connected lines in the two charts show the ratio of time taken by OpenTuner-Only versus the Coupled SAAKE+OpenTuner to find the best configuration. The scale for this data is logarithmic (scale on the right). It may be seen that SAAKE enables significant acceleration for OpenTuner by providing it a very good starting configuration. The results are summarized in Table 2.3.

Figure 2.7 and Table 2.2 present data for the evaluation of SAAKE and OpenTuner for optimizing tensor contraction kernels. Tensor contractions are at the core of many computational chemistry models such as the coupled cluster methods [114]. Due to the

significant fraction of compute time spent in performing tensor contractions, developers of the NWChem [127] computational chemistry suite created a domain-specific code generator [82] to synthesize efficient GPU kernels for tensor contractions. The NWChem suite includes a separate customized GPU kernel for each of 27 tensor contractions for the CCSD(T) method. These customized GPU kernels represent the current state-of-the-art one for this set of contractions [81, 82]. The CUDA source code for these tensor contraction kernels is available in the open-source NWChem software distribution [99].

Three of the CCSD(T) tensor contractions are shown below.

```
sd_t_d1_1: T3(h3,h2,h1,p6,p5,p4) -= t2(h7,p4,p5,h1)*v2(h3,h2,p6,h7)
[...]
sd_t_d1_5: T3(h3,h1,h2,p5,p4,p6) += t2(h7,p4,p5,h1)*v2(h3,h2,p6,h7)
sd_t_d1_6: T3(h1,h3,h2,p5,p4,p6) -= t2(h7,p4,p5,h1)*v2(h3,h2,p6,h7)
[...]
```

Table 2.2 presents performance data for two of the CCSD(T) tensor contractions. For each contraction, and each of the two machines, the best configuration found by SAAKE+OpenTuner is shown, along with the achieved performance for the base version, SAAKE-Only, and SAAKE+OpenTuner. It may be seen that SAAKE-Only achieves a high fraction of the speedup achieved by SAAKE+OpenTuner. The time for performing auto-tuning with/without SAAKE is shown in terms of min/max/average. It can be observed that integrating OpenTuner with SAAKE results in significant reduction in the average tuning time, as well as the maximum time, which can be over an hour (4188 seconds). Fig. 2.7 shows the trajectory over time for OpenTuner versus SAAKE+OpenTuner for one of the benchmarks. It may be seen that for both machines, about a 10x decrease in tuning time is achieved.

Kernel	Mac	Best Config	perf (GFLOP)			Tuning time					
						min		avg		max	
			Base	SAAKE	Best	w/o	w/	w/o	w/	w/o	w/
sd_t_d1_5	k	(16,16,2,3,1,1)	87	157	157	1091	47	1452	50	1702	56
	p	(32,8,2,4,1,1)	178	223	242	279	71	544	164	865	294
sd_t_d1_6	k	(16,8,2,2,1,1)	79	106	129	612	58	981	218	1359	679
	p	(16,8,2,1,1,4)	178	208	226	419	61	2143	150	4188	333

Table 2.2: CCSD(T) tensor contraction: SAAKE and/or OpenTuner

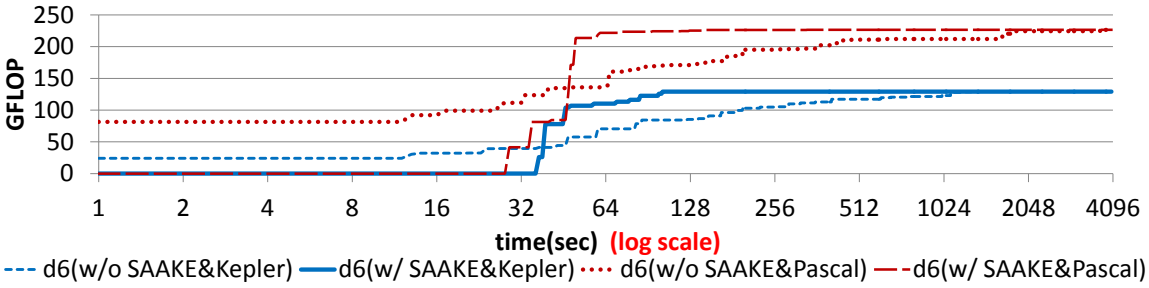


Figure 2.7: OpenTuner auto-tuning: tensor contractions

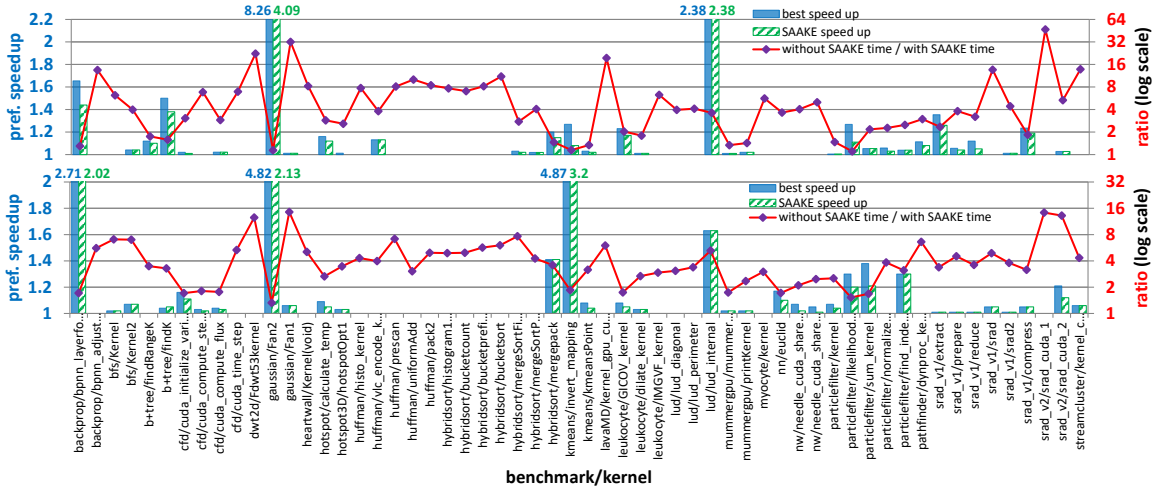


Figure 2.8: OpenTuner auto-tuning: all Rodinia kernels

	Kepler			Pascal		
	Best Speed-up	SAAKE Speed-up	Ratio of time w/o SAAKE vs w/ SAAKE	Best Speed-up	SAAKE Speed-up	Ratio of time w/o SAAKE vs w/ SAAKE
(geo) mean	1.11	1.09	4.16	1.13	1.09	3.72
median	1.01	1.01	3.88	1.02	1.01	3.53

Table 2.3: Geometric mean and median in Fig. 2.8

2.6 Assisting Manual Optimization: Case Studies

As discussed in Sec. 2.5, SAAKE can be used to identify the resource bottleneck(s) for a given GPU kernel. We demonstrate through two detailed case-studies that the bottleneck insights from sensitivity analysis can provide extremely useful information to complement information obtainable through performance tools such as NVPROF and NSIGHT. On the one hand, performance tools like NVPROF and NSIGHT provide very accurate information based on actually measured hardware counter data, while SAAKE bottleneck analysis is based on a simple approximate model of the execution. But on the other hand, SAAKE's sensitivity analysis enables the kind of specific "what-if" exploration with respect to critical resource parameters that is not feasible with the more accurate measurement-based tools.

We present the following illustrative case studies: i) further improving the performance of the tensor contractions discussed in the previous section, and ii) improving the code for stencil computations that was synthesized by a state-of-the-art stencil code generator for GPUs [110].

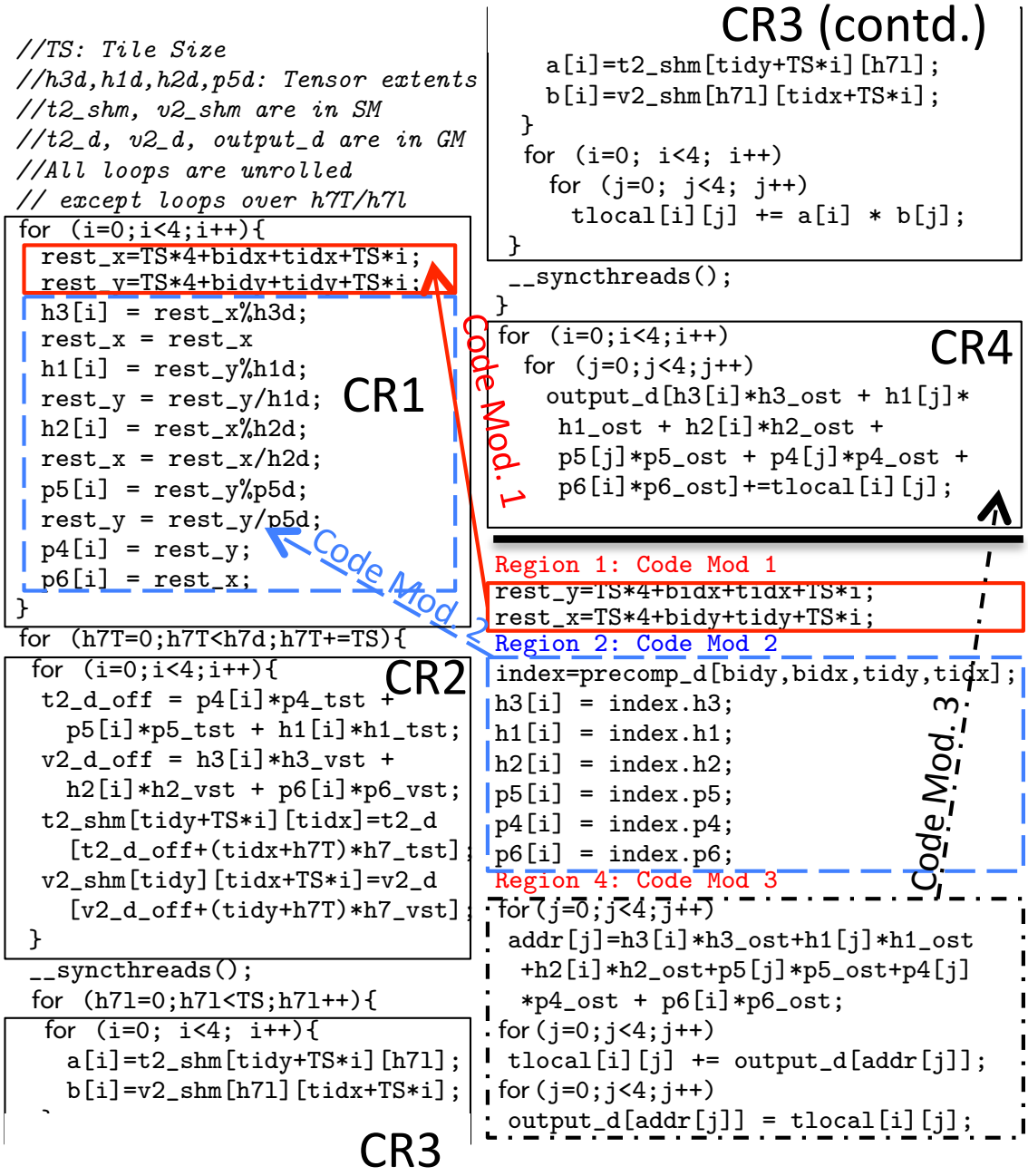


Figure 2.9: SAAKE-based optimization of tensor contraction kernel

Orig. Code (92 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	3%	0%	1%	0%	2%
SFU LAT	16%	16%	0%	0%	0%
GM THR	58%	0%	6%	0%	52%
SM THR	7%	0%	0%	7%	0%

(a)

Code Mod. 1 (123 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	38%	0%	1%	0%	35%
SFU LAT	19%	19%	0%	0%	0%
GM THR	22%	0%	11%	0%	7%
SM THR	8%	0%	0%	8%	0%

(b)

Code Mod. 2 (139 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	41%	0%	2%	0%	40%
SFU LAT	0%	0%	0%	0%	0%
GM THR	31%	0%	14%	0%	13%
SM THR	9%	0%	0%	9%	0%

(c)

Code Mod. 3 (161 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	11%	0%	2%	0%	10%
SFU LAT	0%	0%	0%	0%	0%
GM THR	59%	0%	16%	0%	39%
SM THR	11%	0%	0%	11%	0%

(d)

Table 2.4: SAAKE optimization steps for sd-t-d1-6 on Kepler

Orig. Code (166 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	9%	0%	6%	0%	3%
GM THR	48%	0%	0%	0%	48%
INT THR	6%	6%	0%	0%	0%
DP THR	32%	0%	0%	32%	0%

Code Mod. 1 (190 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	32%	0%	25%	0%	7%
GM THR	13%	0%	0%	0%	13%
INT THR	9%	9%	0%	0%	0%
DP THR	38%	0%	0%	38%	0%

Code Mod. 2 (225 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	21%	0%	13%	0%	8%
GM THR	14%	0%	0%	0%	14%
INT THR	12%	10%	2%	0%	0%
DP THR	46%	0%	0%	46%	0%

Code Mod. 3 (250 GFLOPs)

	total	CR 1	CR 2	CR 3	CR 4
GM LAT	3%	0%	3%	0	1%
GM THR	23%	0%	0%	0	23%
INT THR	14%	11%	3%	0	0%
DP THR	53%	0%	53%	0	0%

Table 2.5: SAAKE optimization steps for sd-t-d1-6 on Pascal

2.6.1 Tensor Contraction Kernel

We present the use of SAAKE in optimizing the sd_t_d1_6 tensor contraction kernel, one of the more complex cases to optimize. The same process can be applied to any of the CCSD(T) kernels. Fig. 2.9 shows the structure of the current code in NWChem's GPU kernel as well as the sequence of changes made during the optimization process. The

SAAKE analysis of the kernel code revealed sensitivity to multiple hardware resources. A hierarchical strategy was used to partition code regions: first run a coarse analysis without partitioning and then refine based on results of the analysis, drilling down into smaller regions that are the focus of sensitivity analysis. The analysis of this kernel, resulted in partitioning the kernel code into four code regions, CR1, CR2, CR3, and CR4, demarcated in Fig. 2.9. The sequence of code modifications (explained later) is also shown in the latter portion of the figure, with arrows connecting the modified code with the original code. Each thread computes a data slice of the result tensor $output\_d$ using needed elements from input tensors, $t2\_d$ and $v2\_d$. Arrays with a "d" suffix denote data in "device" global memory, and those with suffix "shm" are in shared memory. CR1 computes base indices (for $h1, h2, h3, p4, p5, p6$) of the hyper-rectangular data slices of the tensors upon which each thread operates. CR3 is the compute loop for the contraction over the index $h7$. In CR2, slices of the input tensors ($t2\_dandv2\_d$) are moved into shared-memory arrays. CR3 performs the floating-point operations to accumulate the results computed in a local array, *tlocal*, which is placed in registers by the compiler. CR4 writes the final results to global memory ($output\_d$). The i/j loops in the pseudocode are fully unrolled in the actual CUDA kernel code, but are shown as loops for compactness of the pseudocode.

Table 2.4 shows results from the sensitivity analysis of the original kernel code for a subset of resources that exhibited sensitivity: global memory (GM) latency, special function unit (SFU) latency, GM throughput (gap), and shared memory (SM) throughput. The left column of Table 2.4(a) shows the overall sensitivity for these resources, while the remaining four columns present the region-wise sensitivity for CR1, CR2, CR3, and CR4. The highest sensitivity is with respect to GM throughput in CR4 is the primarily affected code region. Index $h3$ is mapped to $threadIdx.x$ to achieve coalesced memory access in

CR2. The fastest varying index (FVI) of the output array is $h1$. Since it is mapped to *threadIdx.y*, global memory accesses for accumulating results are uncoalesced. If $h1$ is mapped to *threadIdx.x* and $h3$ to *threadIdx.y*, coalesced accesses can be achieved in CR4, while uncoalesced memory accesses would now occur in CR2. This swap is shown under *Code Mod. 1* in the pseudocode.

The results of applying SAAKE to the modified code are shown in Table 2.4(b). The performance increased from 92 GFLOPs for the original code to 123 GFLOPs and the sensitivity to global memory throughput (GM-THR) in CR4 is dramatically reduced. There is now sensitivity to GM-THR in CR3, because the transposed mapping now causes uncoalesced reads in CR3. However, it is only 11%, compared to 52% for CR4. We note that the percentage change in time reflected by the sensitivity data is for the total kernel execution time and not just for the time attributable to each local region. We observe a 19% sensitivity to SFU latency. This is due to the modulo and division operations transformed into a sequence of operations including reciprocal ones that are executed by the SFU – this leads to many chains of dependences and, therefore, insufficient concurrency to tolerate the large SFU latency.

Since sensitivity to GM-THR for CR1 is zero (no GM loads/stores occur here), we alleviated the SFU bottleneck by precomputing the indices, storing them in global memory, and reading them from GM, instead of computing them. This is shown in Fig. 2.9 as *Code Mod. 2*.

After this optimization, performance increased to 139 GFLOPs. Table 2.4(c) shows the SAAKE results for this code version. Next, we sought to alleviate the GM-LAT bottleneck in CR4 (40% sensitivity), a consequence of inadequate concurrency to tolerate the long latency of the chained address-computation, read from GM and written to GM in CR4. *Code*

Mod. 3 shows a split code for the same computation, where a set of independent address computations are first performed, followed by a set of GM reads to accumulate results in registers, followed by a set of GM writes. After this code change, performance is increased to 161 GFLOPs and the GM-LAT sensitivity of CR4 is decreased significantly from 40% to 10%. At this point, the kernel is limited by GM-THR on the output. Since each output element is only written once and the GM stores are coalesced, no further optimization is attempted.

Table 2.5 shows the data for applying SAAKE to the same kernel on the Pascal GPU. The sensitivity metrics for the original code are quite different from those seen on Kepler. A significant reason is that this Pascal P102 GPU has very low double-precision performance (peak of 343 GFLOPs). Due to space limitations, we do not provide details on the sequence of code modifications. However, the sensitivity analysis metrics in Table 2.5 demonstrate significant differences.

Table 2.6 compares the performance of the original kernel with the two optimized versions on both GPUs. The Kepler-optimized version achieves the lower performance of 219 GFLOPs on the Pascal, compared to 250 GFLOPs for the Pascal-optimized kernel, but it is better than the original kernel's 166 GFLOPs. However, the Pascal-tuned kernel only achieves 49 GFLOPs on the Kepler GPU, as compared to 92 GFLOPS for the original kernel and 161 GFLOPs for the Kepler-optimized version. The reason is as follows: In the optimized code for the Pascal system, shared memory operations were changed to global memory instructions to utilize the Double Precision unit efficiently. Hence, this code causes more global memory transactions. However, this strategy is very detrimental to the Kepler machine which has a lower memory bandwidth and many more double precision units.

Mode	Kepler GFLOPs	Pascal GFLOPs
Original	92	166
Kepler-Opt	161 (1.75x speedup)	219 (1.32x speedup)
Pascal-Opt	49 (0.53x speedup)	250 (1.51x speedup)

Table 2.6: Performance of optimized kernels on Kepler and Pascal GPUs

	all-15 before	all-15 after	all-16 before	all-16 after	all-17 before	all-17 after
(geo) mean	84.3	134.2	103.9	162.6	82.9	124.3
median	86.3	134.6	106.3	162.2	83.6	123.3

Table 2.7: Geometric mean and median of GFLOPs in Fig. 2.10

Similar SAAKE analysis and optimization was carried out for some other tensor contraction kernels in the set. Based on the gained insights, the tensor contraction code generator that produced the codes in NWChem was modified so that the emitted code structure was like the optimized versions that had been manually generated through SAAKE-based optimization. Fig. 2.10 charts the performance on 18 tensor contraction kernels used in the NWChem CCSD(T) method. Experiments were conducted for three tensor sizes: all tensor extents of 16, all-15, and all-17. We observe that the modified code generator generates kernels that are executed consistently faster than the ones currently used in the NWChem distribution. Results are summarized in Table 2.7.

2.6.2 Optimizing the Hypterm Stencil Computation

We carried out the second exercise on optimizing the Hypterm function from the Exp_CNS benchmark.[2] Due to space limitations, we only present a short summary here; full details are presented in a technical report [50].

[2]https://ccse.lbl.gov/ExaCT/CNS_Nospec.tgz

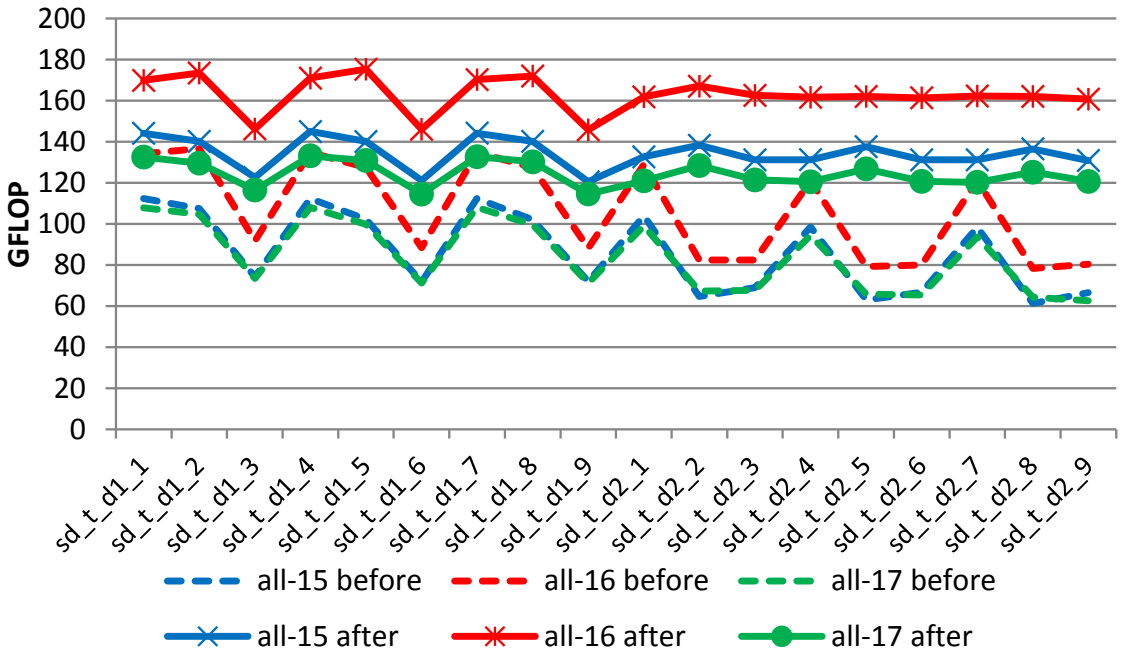


Figure 2.10: Performance of original kernels versus new kernels from modified code generator

The Hypterm function involves a sequence of 15 stencils, S_1, .. S_{15}, presented as a detailed case study by Rawat et al., who developed a domain-specific compiler for optimizing stencil computations on GPUs [110]. Groups of three adjacent stencils make contributions to the same output array and benefit from fusion to reduce data movement. By considering both occupancy and data movements, Rawat et al. [110] presented the following grouping as the final fused configuration: G1={S_1, S_2, S_{13}, S_{14}}, G2={S_3, S_{10}, S_{11}}, G3={S_4, S_5, S_7, S_8}, G4={S_6}, G5={S_9}, and G6={S_{12}, S_{15}}.

Note that in [110], there are seven groups, but S_{12} and S_{15} are fused in the current version. The optimized Hypterm GPU code generated by the code-generator of Rawat et al. [110] was obtained and SAAKE analysis was performed. It revealed that the six kernels were highly optimized in terms of GM and SM data movements. So, the only way to improve performance is by fusing kernels to reduce data movement. A fused kernel

can decrease occupancy but increase ILP and kernels are currently GM/SM throughput limited. We made three fused kernels from the seven kernels, based on the usage of the same inputs/outputs, reducing occupancy to 0.5 (by decreasing occupancy from 1 to 0.5, each thread/thread block can use more resources): G1'=G1,G2, G2'=G3,G4,G5, G3'=G6. Note that we reduce occupancy of the kernel G6 to eliminate register spills which may cause global memory movements. The transformation improved performance from 141 GFLOPs to 161 GFLOPs. SAAKE analysis on the three fused kernels revealed them to still be throughput limited, suggesting further fusion. So, we fused the three kernels to make one fused kernel, decreasing occupancy to 0.25, but improving performance. We also changed the thread block sizes and tile sizes. The final achieved performance was 260 GFLOPs on the Kepler GPU, a significant improvement over the 141 GFLOPs of the version generated by Rawat et al. [110].

2.7 Related Works

GPU Performance Modeling: Several studies have developed analytical approaches for modeling kernel performance on GPUs. The work of Hong et al. [56] represents one of the earliest efforts at modeling GPU performance in terms of the impact of warp level concurrency for computational operations and memory operations on performance. Sim et al. [116] extended the modeling approach of Hong et al. [56] to address more hardware features, and to provide feedback to application developers on four metrics that could guide them in identifying execution bottlenecks and how they may be overcome to improve performance. Baghsorkhi et al. [10] developed an analytical approach to predicting GPU kernel performance by using a more detailed modeling of the assembly-level statements of the kernel binary and inter-statement dependences. Zhang et al. [155] also developed a

quantitative performance analysis model aimed at identifying the primary resource bottleneck for a GPU kernel. Lee et al. [73] proposed a DSL for performance modeling along with an analytical modeling framework, using static analysis of the program to estimate the number of floating point operations and memory accesses. Recently, Xu et al. [146] developed a precise analytical performance model for a cache-less heterogeneous many-core processor SW26010, the current top supercomputer [36]. The performance modeling approach extends that of Hong et al. [56], and its use in static performance tuning of a number of Rodinia benchmarks is demonstrated. Very high accuracy (average error of 5%) and very significant speedup (43x) over auto-tuning were achieved with high tuning quality (6% loss). Zhou et al. recently developed a performance analysis framework for GPUs, with a focus on deep neural networks (DNNs) [158]. Using assumptions on the behavior of GEMM-like computations in DNNs, analytical models for potential resource bottlenecks are presented. These models take the result of ASM code analysis as parameters to describe the input application.

The popular Roofline model [143] is applied in the context of GPU kernel optimization. The GPURoofline [58] system helps non-expert users tune GPU kernels for high-performance. The roofline model is used to first identify the performance bottleneck, which is then relieved through iterative improvement via specific optimization techniques such as Reducing Dynamic Instructions (RDIS) and increasing Instruction-level Parallelism (ILP).

A notable difference between previously reported approaches to GPU performance modeling and the approach developed in this chapter is that, to the best of our knowledge, none of the previous approaches has demonstrated applicability for the *automated* performance modeling of arbitrary GPU kernels. We have demonstrated the use of the developed approach to modeling and code transformation on the entire set of Rodinia benchmarks,

as well as non-trivial tensor-contraction benchmarks used in the NWChem computational chemistry suite.

Compiler Optimization for GPUs: Several research efforts have focused on compiler optimization for GPUs. Many of these efforts have been in the context of affine programs, where precise dependence analysis is feasible. Baskaran et al. [13] developed a C-to-CUDA transformation in the Pluto polyhedral optimizer [18]. The PPCG compiler [131] is another powerful polyhedral compiler for GPU code generation from the C input program. It implements a variety of optimizations such as shared memory promotion. However, its cost models to determine transformation profitability remain very basic.

OpenACC [141] is a directive-based programming model for GPU computing, where a C program annotated with OpenACC directives is automatically transformed by the OpenACC compiler for execution on GPUs. OpenMP [31] also now offers an "Offload" mode that allows user-annotated C programs to be executed on GPUs, in a similar manner to that introduced by OpenACC several years ago. OpenARC [74] is a directive-based compiler that is intended to accept either OpenACC or OpenMP-Offload programs, and generates architecture-specific (including GPU targets) codes.

We did perform several tests with PPCG and OpenACC/OpenMP-Offload for tensor contraction examples, and observed that the achieved performance was considerably lower than that we were able to achieve in this chapter. We believe our tools can be used to improve the performance of codes generated by these compilers, by enabling the selection of better transformations if suitable interfaces are developed. However, that may require significant implementation efforts.

Several efforts have been made to develop domain-specific languages (DSLs) and compilers for GPUs, e.g. [109, 40, 46, 41, 28, 82]. We demonstrated the use of the tools/methodology we developed on two DSL examples in this chapter. We believe that other DSLs for GPUs can benefit from these methods.

For thread coarsening in particular, Unkule et al. [126] developed an approach to analyze GPU codes and perform source-level transformations to obtain GPU kernels with varying thread granularity. Magni et al. [85, 83] developed a thread coarsening tool for OpenCL GPU kernels, in conjunction with their research on developing a machine-learning model for identifying the best thread coarsening factor. It would be interesting to compare the effectiveness of their approach with ours, but we are unable to do so, since their software is not publicly available.

2.8 Summary

Performance optimization requires a clear understanding of the impact of various program transformations on its execution time. In this chapter, we have presented a new approach to capture the key architectural features and their impact on application performance. The usefulness of the abstract kernel emulation approach was demonstrated, and also coupled with the OpenTuner auto-tuning framework.

In addition to the case studies presented, our approach can provide feedback to application developers in helping them identify potentially beneficial transformations. Because the approach only needs *latency* and *gap* parameters for key GPU resources, and not the availability of the actual hardware, it can also help in codesigning algorithms and architectures to maximize performance for a specific application workload on a future system.

CHAPTER 3

MultiGraph: Efficient Graph Processing on GPUs

3.1 Introduction

GPUs offer the potential for higher performance and energy efficiency than multi-core processors. However, actually achieving high-performance with GPUs is not trivial. It generally requires sufficient programmer expertise and understanding of details on low-level execution mechanisms in GPUs, and attention to a number of considerations essential to achieving high-performance. Developing such high-performance GPU algorithms is very time consuming for GPU experts and is not very feasible for the vast number of developers of new data/graph analytics methods.

Therefore, there is considerable interest in developing domain-specific graph processing frameworks that offer application developers a convenient high-level abstraction for developing their algorithms, and also deliver high-performance on GPUs. Several GPU graph processing frameworks have been recently developed, including VWC [55], MapGraph [38], Medusa [157], CuSha [64], WS [63], Frog [115], GreenMarl [54], Falcon [27], Groute [15], and Gunrock [136]. While these frameworks achieve much higher performance than popular vertex-centric graph processing frameworks like Pregel [87], Giraph [124], GraphLab [80] etc., as elaborated in the next section, current GPU graph processing

frameworks still have some performance limitations. In this chapter, we identify sources of performance limitation for current GPU graph processing frameworks and present the MultiGraph[3] system for graph processing on GPUs.

The approaches we present can be incorporated into existing GPU graph processing frameworks like Gunrock, Groute, WS, etc. A key hypothesis that drives this chapter is that graphs and graph traversals exhibit significant non-uniformity; and therefore, a single data representation or execution strategy is unlikely to be effective across the board. By identifying important use cases and execution scenarios, we develop a GPU graph processing framework that uses multiple internal representations of the graph, as well as different execution strategies for different use cases of graph traversal. The key ideas behind the proposed approach are as follows:

- Multiple data representation and execution strategies are used for dense versus sparse vertex frontiers, dependent on the fraction of the graph vertices that are active in a given iteration. Topology-driven algorithms [108] process every graph vertex and edge during each iteration. A different graph representation and a different vertex/edge processing strategy is used for this use case, in contrast to the scenario where only a small fraction of vertices is active in a given iteration.
- A two-phase edge processing approach using a 2D blocked distribution facilitates improved load balancing across GPU threads and improved global memory access efficiency. While 2D partitioning strategies have been used earlier for coarse-grained multi-GPU parallelization [17], we are unaware of its prior use for improving fine-grained parallelization in a GPU.

[3](https://github.com/hochawa/multigraph)

- Different representations of edge data for high-degree vertices versus low-degree vertices enables avoiding expensive global memory atomic operations typically used by GPU graph frameworks.
- Efficient dynamic work distribution is achieved for sparse frontier processing by grouping a vertex's edges into bins of three granularities: thread block, warp, and thread.

We present experimental data comparing performance with the proposed approach and state-of-the-art GPU graph-processing frameworks, using graph datasets that have been used in other recent studies. We show that the new approach can achieve high performance for a range of benchmarks.

3.2 Background and Motivation

Several customized frameworks have been developed for implementation of graph algorithms on GPUs [55, 64, 63, 136, 15]. The developments described in this chapter are motivated by the performance limitations of existing frameworks for graph processing on GPUs. These limitations stem from one or more of the challenges in achieving high-performance on GPUs. In this section, we first provide general background on graph processing frameworks, followed by a discussion of specific sources of overhead with current state-of-the-art GPU graph processing frameworks.

Graph Processing Frameworks: Graph processing frameworks facilitate the convenient development of portable high-performance algorithms that iteratively traverse "active" graph vertices and/or edges, performing some operations to update vertex/edge attributes. For example, let us consider Breadth-First Search (BFS). An attribute is associated with each vertex to designate its level in the BFS. This attribute is initialized to infinity (a number

larger than the number of vertices) for all but the root vertex, whose level is set to zero. At each iteration, an active set of vertices is processed, with the active vertex at the first step being the root vertex. The processing of an active vertex involves traversing each of its outgoing edges to determine the current level of the destination vertices. If a neighbor vertex has a current level of infinity, its level is set to one more than that of the active source vertex. All such vertices that have their level set in the current iteration are placed in the active front for the next iteration. This process is repeated until the levels of all vertices get finalized and the active front for the next iteration is empty.

GPU Performance Challenges: There is parallelism at two levels for each iteration of the BFS execution described above: i) each vertex in the current active front can be processed in parallel, and ii) each outgoing edge of an active vertex can be processed in parallel. When edges are processed, identification of the active vertices for the next front can be done in parallel. However, achieving efficient parallelization on GPUs poses challenges.

- **Load Balancing:** One option for work distribution is to assign each active vertex to a thread; however, different threads may require very different amounts of work, since the degrees of the active vertices can differ significantly. Another option is to assign each outgoing edge of an active vertex to a thread, but this is challenging to do efficiently: each thread may spend more time identifying its edge attributes than in processing it.
- **Concurrent Edge Processing:** Often, processing an edge entails some modification of an attribute at the destination vertex. Since two concurrently processed edges may point to a common destination vertex, atomic operations may need to be used, which can be quite expensive, especially if performed at global memory locations.

- **Formation of Next Front:** The insertion of vertices/edges to form the next active front requires coordination among concurrently executing threads.
- **Uncoalesced Global Memory Data Access:** Access of attributes from the set of destination vertices of a set of active edges will generally require inefficient uncoalesced access, since they will generally be scattered and not contiguously located in global memory.

Analysis of Existing Frameworks: We now contrast different GPU graph processing frameworks in terms of their merits and challenges.

CuSha [64]: The CuSha framework was the first to address the limitation of uncoalesced global memory data access for GPU graph processing by performing updates in shared memory. Instead of the standard CSR data structure, CuSha uses an alternative G-Shard representation. CuSha is a framework for topology-driven algorithms, making it inefficient for data-driven algorithms like BFS. Although results are updated in shared memory, CuSha can suffer from serialization overhead from atomics when processing highly skewed datasets.

WS [63]: Warp Segmentation (WS) provides an edge-distributing load-balancing strategy using binary search to locate the vertex ID corresponding to an edge in CSR format. This kind of load balancing idea, first proposed by Andrew et al. [34], is now widely used in many GPU frameworks such as Gunrock. WS is also a framework for topology-driven algorithms, and therefore is not well suited for data-driven algorithms. In addition, it incurs uncoalesced global memory accesses for vertex values due to use of the CSR representation.

Gunrock [136]: Gunrock uses a data-centric abstraction focused on operations on frontiers (vertex or edge). These primitives help programmers develop new graph algorithms without much effort. Gunrock is currently regarded as one of the state-of-the-art GPU graph processing systems. However, Gunrock incurs overhead due to uncoalesced global memory accesses and global memory atomic operations. When the frontier size is large, Gunrock's performance is limited by these factors.

IrGL [102]: Pai et al. [102] identify three factors that limited GPU performance for processing graph algorithms. To resolve these limitations, they introduce three optimizations: iteration outlining, hierarchical aggregation, and nested parallelism. The IRGL compiler produces CUDA code from an intermediate-level program representation. They demonstrate excellent performance when these limitations are resolved. However, as with Gunrock, uncoalesced global memory accesses and global memory atomic operations can limit performance.

Groute [15]: Contrary to synchronous GPU frameworks, Groute implements a scalable asynchronous model for graph processing. Performance on mesh-like networks is considerably better than synchronous frameworks like Gunrock. However, the issues with uncoalesced global memory accesses and global memory atomics can limit performance.

Addressing Performance Limiters: In this chapter, we develop an approach to GPU graph processing that alleviates some of the performance limiters of current frameworks through use of a dual data representation and different vertex/edge processing strategies that depend on the context. We provide an overview of the approach in the next section.

3.3 Overview

This section provides an overview of the system for iterative graph processing developed in this chapter. A guiding hypothesis behind this chapter is that no single standard graph representation (such as CSR or CSC) offers sufficient flexibility to achieve high processing efficiency for the variety of traversal patterns and graph characteristics encountered in graph applications.

Fig. 3.1 provides a high-level view of the graph processing system. A primary distinction is made between two scenarios: i) all (or most) vertices are processed in each iteration (**dense input frontier**), and ii) only a small subset of vertices is processed in an iteration (**sparse input frontier**) and that active set of vertices is not known until the end of the previous iteration. Different data representations and execution strategies are used for these two cases, as described in detail in Sec. 3.4 (dense-frontier) and Sec. 3.5 (sparse-frontier). For dense-frontiers, the same processing happens at every iteration, and hence the data structures can be set up to optimize execution. In contrast, with sparse-frontiers, the active set can change significantly across iterations, making it more challenging to achieve high-performance. The input graph is first read in and a pre-processing step generates the dual representations of the graph, so that it can be processed in either dense or sparse frontier mode, as appropriate.

The dense-frontier mode can also be used (via "masking") for input frontiers, where only a subset of vertices is active. Although the sparse-frontier mode only traverses edges corresponding to the active vertices in the front (whereas the dense-frontier mode essentially traverses all edges), the dense-frontier mode can achieve higher performance than the sparse-frontier mode when the fraction of active vertices is sufficiently high. The cross-over front-density threshold depends both on the graph and the graph algorithm. Therefore, a

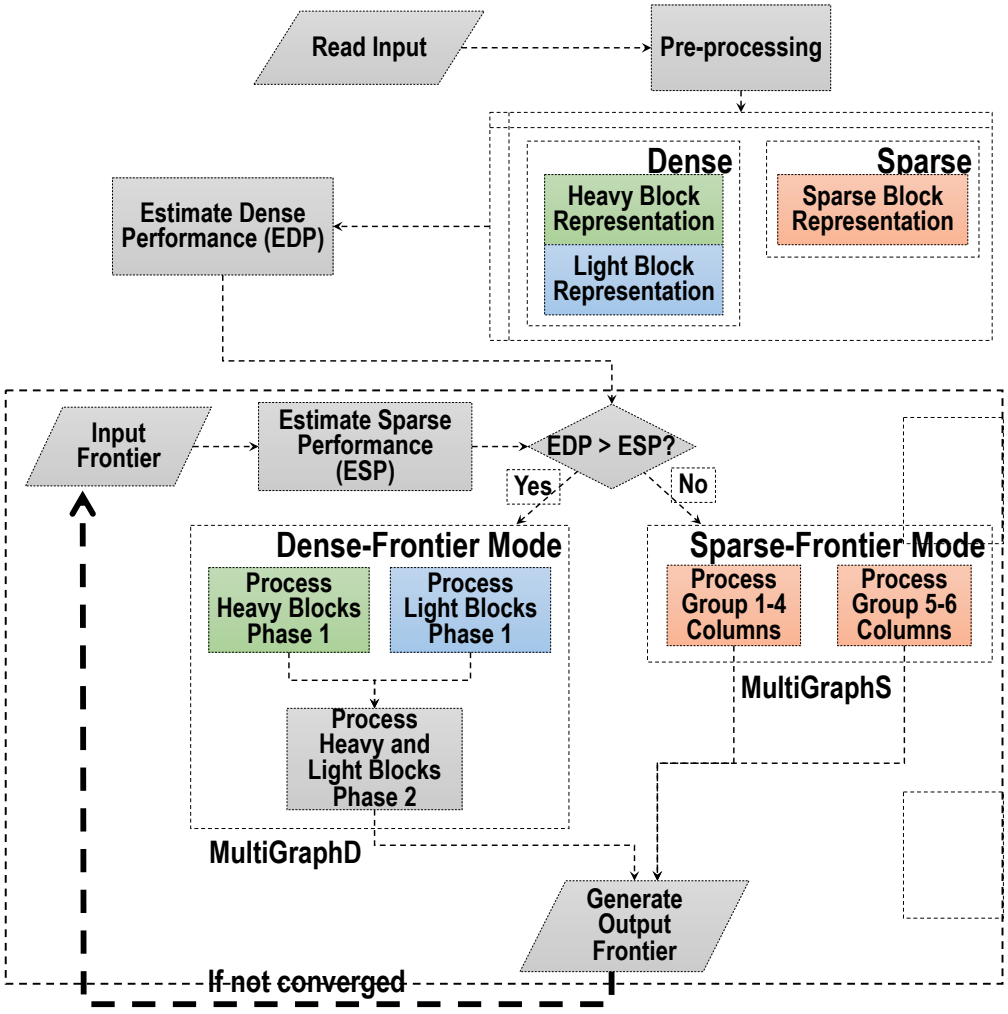


Figure 3.1: Overview of MultiGraph system

sampling-based approach is used to implement a hybrid algorithm (Sec. 3.6) that selects either the dense-frontier or sparse-frontier mode of execution for each iteration of an iterative graph algorithm. The choice is made by comparing estimated performance for sparse-front processing (ESP) with estimated performance for dense-front processing (EDP). EDP is independent of the fraction of active vertices and is determined by execution with a small

fraction of vertices before the first iteration of execution of the full initial front. ESP is estimated for each iteration by sampling a small fraction of the active vertices and processing them in sparse-front mode. Depending on whether EDP or ESP is larger, the dense-frontier mode or sparse-frontier mode is chosen.

In describing the algorithmic details of the developed graph processing framework, we use an edge-matrix abstraction to represent the graph. Each element in this matrix represents an edge in the graph. During the pre-processing step, the vertices of the rows of the edge-matrix are reordered by placing vertices into one of six groups, based on the number of non-zeros in a row (which is the corresponding vertex's 'indegree'), as specified in Table 3.1. Row (vertex) reordering is performed so that vertices in the same group are numbered consecutively. Grouping together vertices of similar indegree facilitates load-balanced work distribution across threads, and different edge processing strategies based on the amount of work per vertex. In sparse-frontier mode, different strategies are used for groups 1-4 versus 5-6. In dense-frontier mode, graph edges are grouped by 2D block-partitioning of the index space, with different representations and processing strategies for heavily populated versus lightly covered blocks. After processing the active vertices using the selected mode, the output frontier is created, and the iterative process repeated, until reaching the iteration termination condition.

3.4 MultiGraphD (Dense Frontier)

This section describes MultiGraphD, the two-phase streaming approach for dense-frontier processing. It is aimed at achieving good load balance and low divergence across threads, high warp occupancy, and efficient coalesced data access to/from global memory. In addition, compared to frameworks like Gunrock, MultiGraphD's two-phase scheme is

Table 3.1: Grouping criteria

Group	Criteria
1	indegree >= 2048
2	1024 <= indegree <2048
3	512 <= indegree <1024
4	256 <= indegree <512
5	128 <= indegree <256
6	indegree <128

designed to reduce the total number of atomic operations on global memory and global memory transactions (as is quantitatively demonstrated using hardware counter measurements when experimental data is presented in Sec. 3.7). Processing is performed in two phases:

- In phase-1, graph edges are streamed in, structured in batches over limited contiguous ranges in the column-index space of the edge matrix, i.e., over a limited range of source vertex IDs for the processed edges. Prior to streaming in a batch of edges, the vertex attributes over that index-range are loaded into shared memory. Multiple contributions for any destination vertex are first locally combined and tuples of (destination_index,contribution) are written out into pre-determined locations in an intermediate stream-buffer in GPU global memory.
- In phase-2, key-value pairs for contributions to destination vertices are streamed in, pre-structured in batches that correspond to limited contiguous ranges in the row-index space of the edge matrix. This enables multiple accumulations to a destination vertex to be performed using cheaper shared-memory atomic operations instead of global memory atomics, as typically used in other graph processing frameworks like

Gunrock. Final accumulated result values are written out to destination vertex attributes in global memory.

3.4.1 Phase-1

In phase-1, both edge- and vertex data are streamed in from global memory in a coalesced manner, and partial contributions are accumulated in registers or shared memory. Different strategies are used for edge processing, depending on the amount of work per edge block. This differential processing is motivated by the fact that different edge blocks can differ greatly in the amount of work to be performed. If the number of edges in an edge block is greater than 18k (an empirically determined threshold that was found to be the best), then it is classified as *heavy* in work, otherwise as *light*.

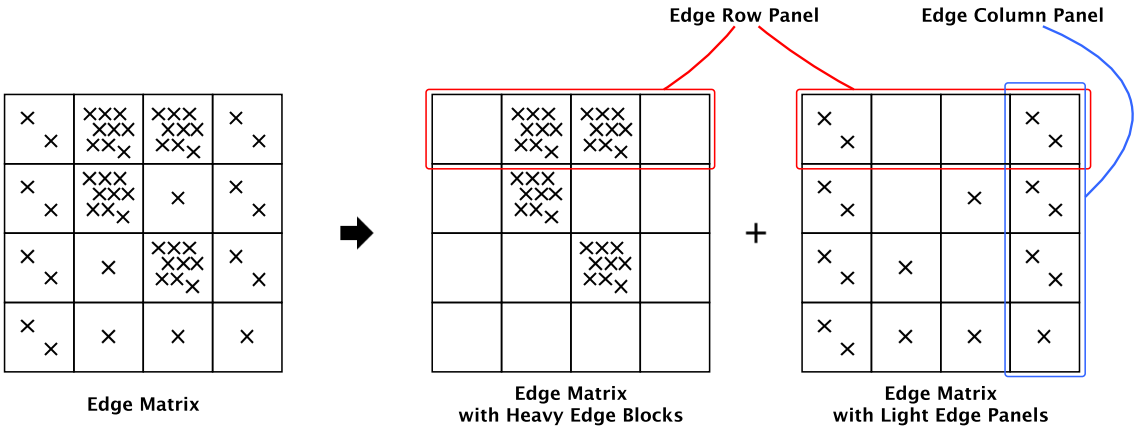


Figure 3.2: MultiGraphD: edge matrix partitioning into heavy and light blocks/panels

As shown in Figure 4.3, the edge-matrix is split into two parts: a part containing heavy edge blocks and another with only light blocks. This split is performed only once, in the pre-processing phase. All of the light edge blocks in a single block column (column panel)

are processed by a single thread block. In contrast, each of heavy edge blocks is processed by a thread block. The destination vertex attributes are kept in global memory and are set to the identity of the associative accumulation operator (zero for addition).

Heavy edge block: The representation used for heavy edge blocks is depicted in Figure 3.3. The conceptual view of a single heavy edge block is represented on the left. The data representation is comprised of two matrices: i) a frontier matrix, holding source/destination vertex IDs (middle); and ii) a matrix with edge weights. The dimensionality and total number of elements in both matrices are the same. The total number of elements is computed as the sum of the number of threads in a thread block, number of edges and number of destinations which have at least one contribution from the current edge block. The number of columns is equal to the number of threads in a thread block. The number of rows is the ceiling of the number of elements divided by the number of threads (ceil(#elements/#threads)). The edge block frontier is filled in column-major order. The first element is the negated value of the destination ID, which receives a contribution from the current edge block. This is followed by placing all of the edges which are connected to the latter destination node. Then the negated value of the next destination node follwed by the corresponding incoming edges is placed and so on. While filling the table, if the row limit is reached, the destination ID is placed at the beginning of the next column, which is followed by edges. For example, destination node 106 in Figure 3.3 has four edges. After placing the first source vertex, the row limit is reached. Hence, the destination ID is repeated again at the beginning of the next column. In this matrix, the negated values represent the destination IDs; and positive values represent source IDs. Corresponding to each source vertex in the edge block frontier matrix, the edge block is populated with edge weights. All other entries in the edge block frontier matrix are marked as invalid. This representation is created during pre-processing.

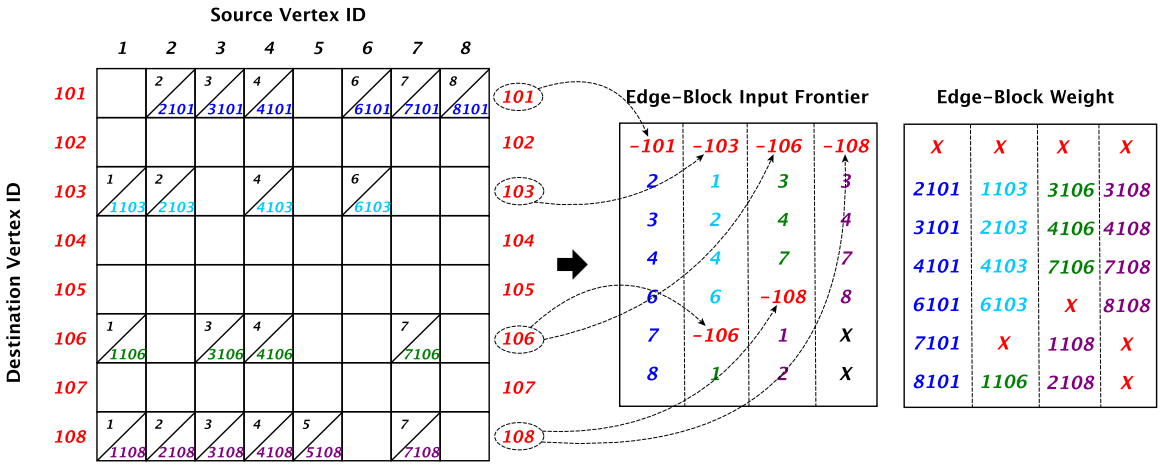


Figure 3.3: MultiGraphD: processing of heavy edge block

The pseudocode for processing of heavy edge blocks is shown in Algorithm 4. All of the threads in a thread block collectively bring a portion of the input frontier corresponding to an edge block to shared memory. All vertices in the front may not be active. The information regarding whether a frontier vertex is active or not is also brought to shared memory.

The work is uniformly distributed across threads. Each thread processes a single column in the edge block frontier matrix using the corresponding element in the edge block weight matrix. The partial contributions are accumulated to a thread local register as long as the destination vertex does not change. If the destination vertex changes, as indicated by a negative entry in the column, the accumulated partial contribution is written to shared memory using an atomic operation. The use of shared-memory atomics is necessary, since two threads could concurrently attempt to update the same destination value (e.g., with destination 106 in Fig. 3.3). After a heavy edge block is processed, accumulated contributions to all destination vertices of the edge block are written out.

Algorithm 4: Algorithm for processing heavy edge blocks in dense frontiers

```
1  kernel MultiGraphD__heavy_edge_block()
2      edge_col = edge_block_list[tb_id].col
3      edge_blk_id = tb_id
       // Bring input frontier to shared memory
4      for (i = edge_col + t_id to edge_col + EDGE_BLK_SIZE − 1 step tb.size()) do
5          sm_frontier_val[i-edge_col] = frontier_val[i]
6          sm_frontier_active[i-edge_col] = frontier_active[i]
7          sm_dest_value[i -edge_col] = init()
8      end
9      __syncthreads
10     start = start_edge_position[edge_blk_id]
11     end = start_edge_position[edge_blk_id + 1]
12     num_rows = floor((end-start + tb.size() -1 - t_id)/tb.size())
13     dest_id = abs(edge_block[edge_blk_id][0][t_id] + 1)
14     reg_val = init()
15     for i = 1 to num_rows − 1 do
16         cur_val = edge_block[edge_blk_id][i][t_id]
17         if cur_val ¡0 then
18             if reg_val != init() then
                   // Update dest val & actv flag
                   atomic_update(&sm_dest_value[dest_id], reg_value)
19                 reg_val = init()
20             end
21             dest_id = abs(cur_val + 1)
22         end
23         else if sm_frontier_active[cur_value] == True then
               // Update register value
24             reg_val = comp_contrib(sm_frontier_val[cur_val] ,
                 edge_block_wt[edge_blk_id][i][t_id], reg_val)
25         end
26     end
27     if reg_val != init() then
28         atomic_update(&sm_dest_value[dest_id], reg_value)
29     end
30     stream_buf_base = loc_dest_buffer[tb_id]
31     for (i = t_id to EDGE_BLK_SIZE − 1 step tb.size()) do
32         stream_buf_base[i] = sm_dest_value[i]
33     end
```

The data representation used for the heavy edge blocks combines both the coalesced data access benefit from the transposed data representation, as used by Liu and Vinter with CSR5 [76], and the load balancing across threads achieved with the merge-based CSR scheme of Merrill and Garland [90].

Each thread has to efficiently find a location in global memory to write its partial contribution. During pre-processing, the maximum space required to save the partial results (by assuming that the output frontier is dense) is computed. The space is allocated in such a way that all of the edge blocks in an edge-row-panel occupy contiguous memory locations.

The thread block size was chosen to be 1024, and edge block size was chosen as 6*1024 in order to fully utilize shared memory, as well as achieve maximum occupancy and minimize the number of edge blocks.

Light edge blocks: The representation used for light edge blocks is shown in Figure 3.4. The top matrix shows the conceptual view of the edge-matrix. Unlike heavy edge blocks, all light edge blocks in an edge column panel are processed by a single thread block. The edges are streamed in from global memory. This is represented by the bottom matrix. The last row contains the input frontier vertex ID and the corresponding edge-weights are shown in the second row. The first row contains an index/pointer to a stream-buffer location where the partial contribution is be written; the actual location is represented by the first row of the middle matrix. The second row of the middle matrix contains the ID of the destination node.

Algorithm 5 describes the processing of the light column panels. Each column panel is processed by a single thread block. All of the threads collectively bring the input frontier values to shared memory. The entire work is then divided cyclically across threads. Each thread first loads a vertex, and if it is active, it directly writes the partial product to global

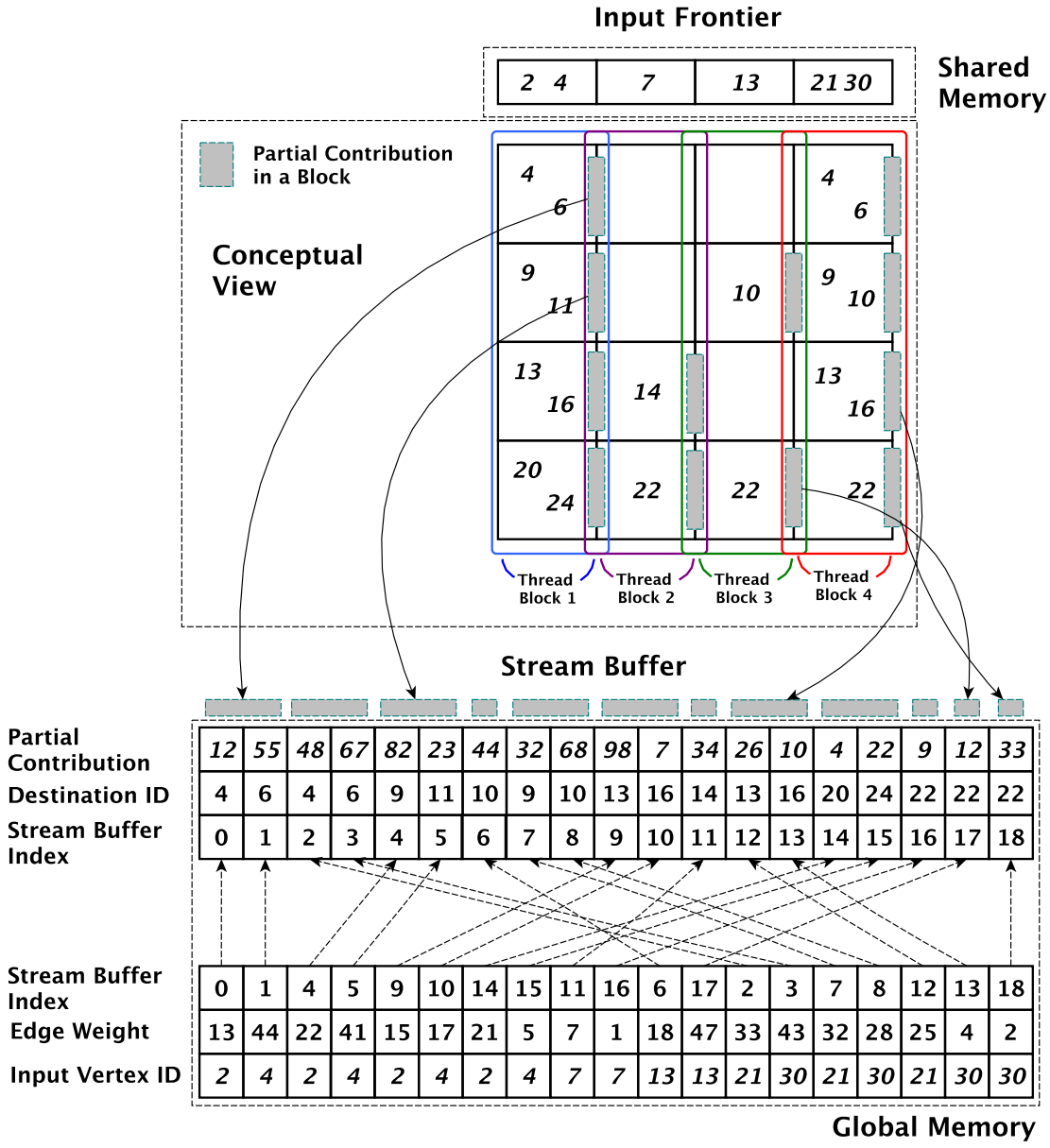


Figure 3.4: MultiGraphD: processing light edge column panel

memory. Since the memory is pre-allocated, atomic operations are not required. However, unlike heavy edge blocks, contributions from the light edge blocks are not locally compressed.

Algorithm 5: Algorithm for processing light edge blocks in dense frontiers

```
1  kernel MultiGraphD__light_edge_block()
2      edge_col = tb_id * edge_block.size()
       // Bring input frontier to shared memory
3      for (i = edge_col + t_id to edge_col + EDGE_BLK_SIZE − 1 step tb.size()) do
4          sm_frontier_val[i-edge_col] = frontier_val[i]
5          sm_frontier_active[i-edge_col] = frontier_active[i]
6      end
7      __syncthreads
8      start = start_edge_position[tb_id]
9      end = start_edge_position[tb_id+1]
       // Stream out contributions to global memory
10     for i = start + t_id to end − 1 step tb.size() do
11         if sm_frontier_active[input_vertex_id[i]] == True then
12             pos_to_write = stream_buf_index[i]
13             stream_buf[pos_to_write] = compute_contrib(sm_frontier_val[
                   input_vertex_id[i]], edge_weight[i])
14         end
15     end
```

This work division strategy between heavy- and light edge blocks helps in achieving good performance with a small number of atomic operations. Since the number of partial contributions from a heavy edge block is high, assigning an entire thread block to it helps in efficient processing. In contrast, since the number of partial contributions from a light edge block is generally low, assigning a thread block to an entire edge column panel helps in attaining better performance.

3.4.2 Phase-2

A single destination vertex can have contributions from multiple edge blocks, i.e., there could be contributions to the same destination vertex from different heavy edge blocks and different light edge blocks. Each such partial product should be combined to produce the

final result. Phase-2 combines the partial results produced in phase-1, to form the final output.

In phase-2, each edge row panel is processed by a thread block. Note that the global memory allocated for storing the partial products from heavy edge blocks and light edge blocks are contiguous in memory. Each thread block loads the corresponding output vertices to shared memory, which helps in reducing the total number of global transactions. Then it loads the corresponding global memory blocks containing the partial results, cyclically distributing work among threads to achieve coalesced global memory access. Updates to output vertices are performed using shared-memory atomic operations.

3.5 MultiGraphS (Sparse Frontier)

When the input vertex frontier represents only a small fraction of the graph's vertices, the data representation and processing approach of MultiGraphD is very inefficient, since essentially every edge must be traversed. For such a scenario, we use a completely different graph representation and edge processing approach. The subsystem for processing sparse frontiers, MultiGraphS, is described in this section. A major challenge to be addressed is that of balanced work distribution. Intra-warp, inter-warp and inter-thread block load balance are key factors that affect performance.

MultiGraphS partitions the vertices of the input frontier into groups of 64 active vertices. Each such group is processed by a single thread block of 256 threads. This work distribution is shown in Figure 3.5.

The outdegrees of frontier vertices can differ significantly. If a warp is assigned to process a frontier node, there could be severe load imbalance, as some warps may have very little work, while others may have a lot of work. Further, for vertices with an outdegree a

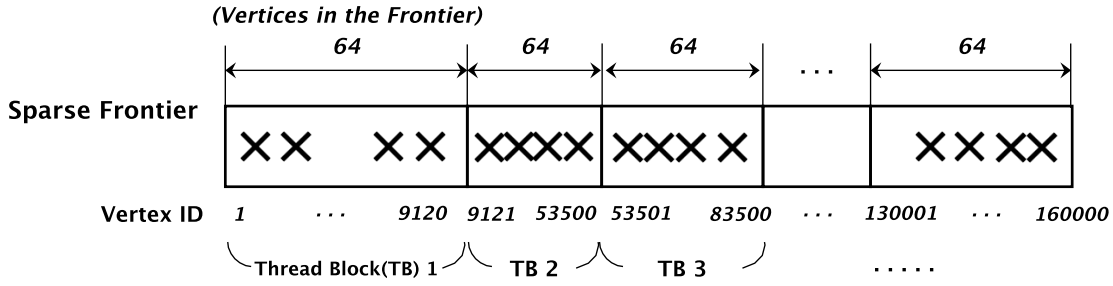


Figure 3.5: MultiGraphS: work distribution

lot less than 32, many threads within a warp will be inactive and cause idleness of hardware resources in the GPU's SIMD functional units. Below, we describe the MultiGraphS approach to load balancing.

The processing of outgoing edges of active frontier vertices is performed in three stages:

- **Stage-1:** involves processing a small number of edges from each vertex, so that the remaining unprocessed edge-count for each vertex is a perfect multiple of 32. If $vertex_k$ has E_k outgoing edges, the last E_k % 32 edges are selected for processing. For this stage, the thread block of 256 threads is partitioned into 64 virtual warps of four threads each, and virtual warp k handles edges from vertex k in the current group.
- **Stage-2:** involves processing more outgoing edges (if any remain) from each vertex, so that the remaining unprocessed edge-count of each vertex is a perfect multiple of 256. For $vertex_k$ with E_k initial edges, of which E_k % 32 got processed in stage-1, (E_k - E_k % 32) % 256 will be selected for processing in stage-2. For this stage, the thread block of 256 threads is organized as eight warps, with each warp processing all stage-2 edges from eight vertices. For this stage, no intra-warp load imbalance or thread

divergence can occur, with the only negative effect of inter-warp load imbalance being the loss of effective warp occupancy, since some active warps may be idle.

- **Stage-3:** processes all remaining edges for all vertices. This is done sequentially across frontier vertices that still have edges to be processed. Since that count is a multiple of 256, perfect load-balancing across all warps of the thread block is achieved.

Fig. 3.6 shows an example illustrating the three stages of processing. Frontier vertex-2 has four outgoing edges and all its edges are selected for processing in stage-1 (4 % 32 = 4). Frontier vertex-500 has 100 outgoing edges, from which four edges are selected for processing in stage-1 (100 % 32 = 4). In stage-1, each vertex is processed by four threads.

The edges that were not processed in stage-1 are passed to stage-2. In Figure 3.6, for frontier vertex-500, all 96 remaining edges are selected for processing in stage-2 (96 % 256 = 96). Similarly, for frontier vertex-80000, 160 edges (2976 % 256 = 160) are selected for processing in stage-2. Each vertex (corresponding to the selected edges) is processed by the threads of a warp using a cyclical work distribution. The remaining set of edges after stage-2 are processed in stage-3, where an entire thread block is assigned to each vertex.

The active frontier is loaded to shared memory and its computation is efficiently calculated using warp aggregation [37]. After processing edges in stage-1, vertices that have fewer than 32 outgoing edges will be fully processed. Hence, they should be removed from the frontier. This is done using warp aggregation. Similarly, after stage-2, all vertices that have fewer than 256 outgoing edges should be removed.

Updates from each edge are accumulated in global memory using atomic operations. An output node with ‘n’ incoming edges will require exactly ‘n’ atomic operations on the same memory location. This implies that nodes with heavy in-edge degree will have limited parallelism, since the atomic operations on the same memory locations will have a

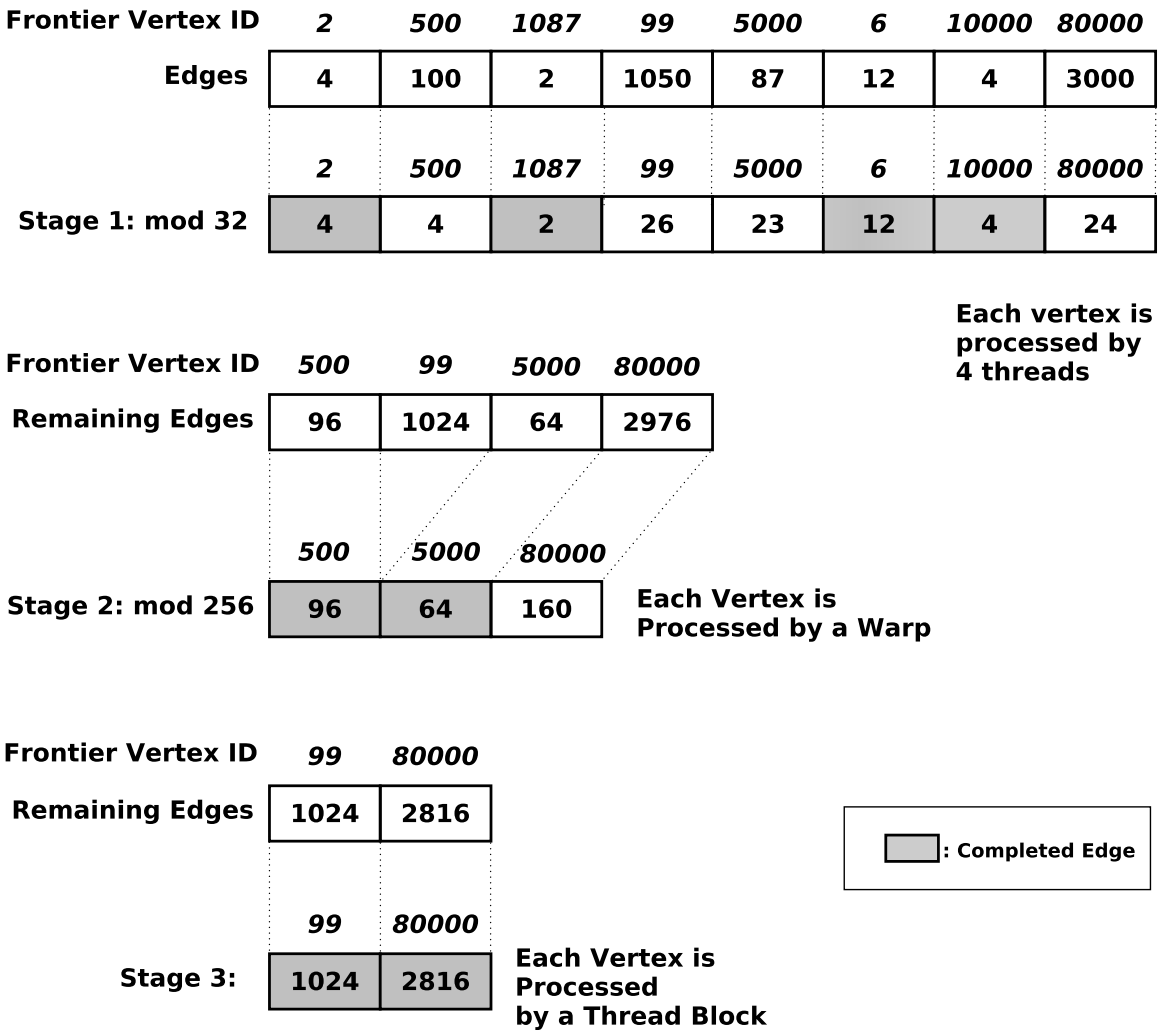


Figure 3.6: MultiGraphS: example illustrating 3-stage work distribution

serializing effect. This will have a lesser impact on performance if the atomic operations were carried out in shared memory instead of global memory. One way to resolve this is to bring a set of output edges in shared memory, process them (using atomics in shared memory), and then stream out the results to global memory. However, using a lot of shared memory may degrade performance, as warp occupancy will be affected. In addition, output nodes with few incoming edges will have minimal benefits when using shared memory for

atomics. Hence, we use shared memory only for output nodes with high indegree. The grouping information computed during pre-processing (mentioned in Section 5.3) is used for this classification. All of the output nodes belonging to group-1 through group-4 are classified as 'high indegree' nodes and the rest are classified as 'low indegree' nodes. The high indegree nodes are partitioned into segments.

Each of these segments is further sub-divided into sub-segments, such that each sub-segment contains 1024 active input frontier vertices. This helps in achieving good inter-block load balance. The segmentation strategy is depicted in Figure 3.7. Each heavy sub-segment is processed by a single thread block. The thread block loads the output vertices to shared memory, accumulates the partial contributions in shared memory using atomics, and writes out the result to global memory.

The entire set of edges for low indegree nodes are treated as a single segment. Each of these segments is then sub-divided into sub-segments with 64 active input frontier vertices. Each sub-segment is processed by a thread block. For efficient access, each segment is represented in CSC format. The CSC representation for each segment is created during pre-processing.

For groups 1-4, the largest possible thread block size (1024) was selected and frontier size was also set to 1024 to maximize available shared memory size per thread block and minimize the number of segments. The key idea behind load balancing of MultiGraphS comes from Merrill et al. [91]. The same thread block size of 256 was chosen as it was empirically verified to outperform other thread block sizes. The frontier size per thread block was chosen to be 64 because each vertex is processed by four threads in stage-1 (256/4 = 64).

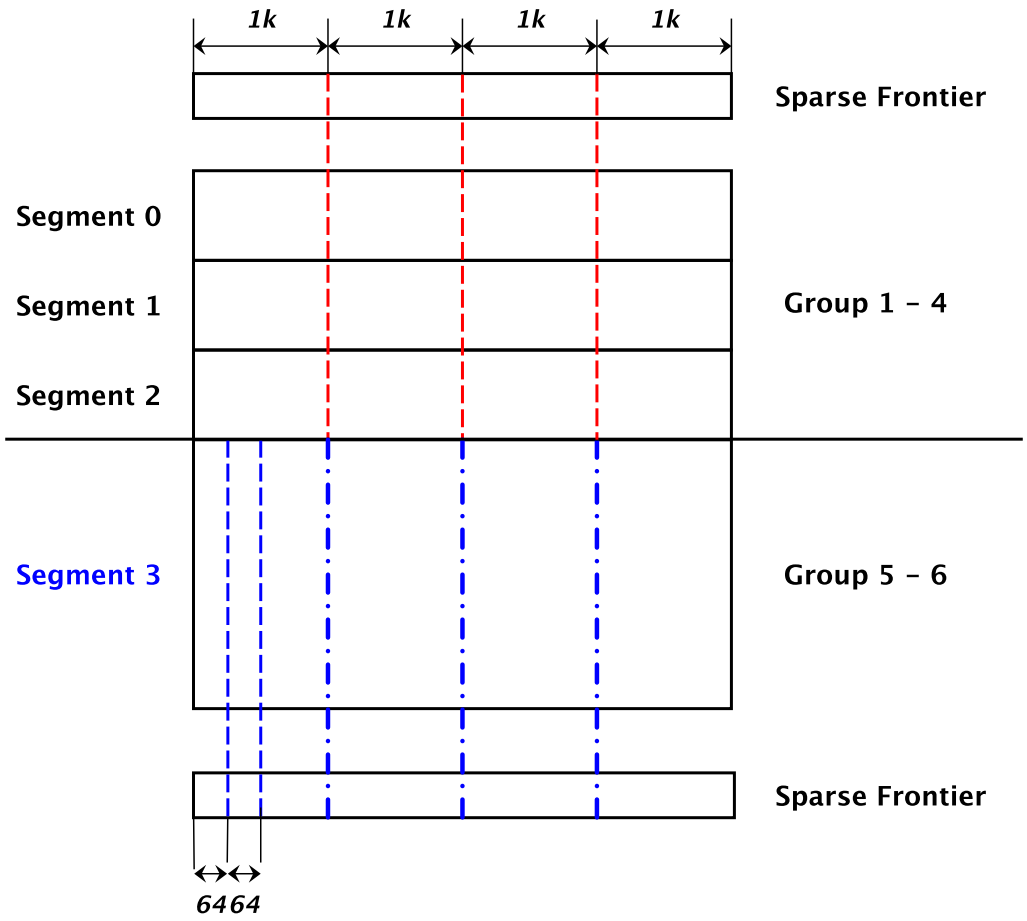


Figure 3.7: MultiGraphS: edge block segmentation

3.6 MultiGraph (Hybrid)

In previous sections, the data structures and algorithms for dense-frontier (MultiGraphD) and sparse-frontier (MultiGraphS) processing were presented. When processing graphs, the sizes of frontiers do not remain constant across iterations. The profile of variation in the size of a frontier depends on the algorithm/benchmark and the dataset. Consider Table

Table 3.2: BFS frontier size variation: soc-orkut

Iter	Frontier size # vertices	# edges	Sparse time	Dense time	Best possible time
1	1	12	0.07	21.81	0.07
2	12	580	0.05	21.81	0.05
3	460	38590	0.13	21.85	0.13
4	25400	3294619	1.40	21.95	1.40
5	728870	73887541	33.93	22.48	22.48
6	2210215	135105894	66.53	22.22	22.22
7	32216	371130	0.26	21.84	0.26
8	32	52	0.07	21.82	0.07
total time			102.45	175.78	46.69

3.2, which shows how the frontier size varies over the iterations of BFS for dataset 'soc-orkut', as well as the execution time taken by MultiGraphD and MultiGraphS. The frontier size initially grows and then decreases. For small frontier sizes, MultiGraphS is much faster, while MultiGraphD can be up to three times as fast for the densest front (at iteration 6). Most datasets exhibit a similar trend, with neither MultiGraphD nor MultiGraphS being consistently better across all iterations of an algorithm like BFS. This motivates the need for a hybrid algorithm that judiciously selects between the MultiGraphD and MultiGraphS at the beginning of each iteration, based on the size of the frontier. If such a choice is achieved with perfect precision and zero overhead, the achievable completion time is shown in the last column of Table 3.2 to be 46.69 ms, compared to 102.45 ms for MultiGraphS and 175.78 ms for MultiGraphD.

The major challenges associated with designing a hybrid algorithm are: i) Performance modeling – for a given benchmark-dataset pair and frontier size, estimating the relative performance of MultiGraphS and MultiGraphD; and ii) Efficient mode-switching – if the

predicted performance is better for a different mode than the current one, efficient switching to the other mode. One approach to estimate performance is to build a linear regression model against the number of active vertices and edges in the current and next frontier. This was attempted, and although it worked reasonably well for some cases, it was not consistently effective across datasets and graph algorithms. Therefore, a dynamic sampling-based approach was developed for a hybrid implementation, called MultiGraph.

Pseudocode for MultiGraph (the hybrid algorithm) is shown in Algorithm 6. At the beginning of each iteration, a random fraction of the current frontier is sampled and the corresponding subgraph processed using MultiGraphS. The performance of MultiGraphD is also estimated using a small sample of vertices. However, unlike MultiGraphS, for a given dataset and algorithm, the performance of MultiGraphD does not vary much as a function of the number of active vertices in the frontier. This can be observed in Table 3.2. Hence, MultiGraphD execution on a sample is only done once, before the first iteration.

In each iteration, MultiGraphS is executed on a small sampled fraction of the frontier. If the current mode is dense (i.e., MultiGraphD), then a small subset of the frontier is generated with the flag "frontier_active". Warp aggregation is used to accumulate sample_frontier. The overhead for generating sample_frontier is low, since only a small fraction of the whole frontier is used. If the current mode is sparse (i.e., MultiGraphS), then a sample can be directly obtained from the whole frontier.

When switching from MultiGraphD to MultiGraphS, the sparse frontier has to be computed. This is done using warp aggregation [37]. Each warp scans the frontier_active flag and collects the list of active vertices. The warp leader then uses a single atomic operation to allocate space for storing the sparse front. Each thread then writes the active vertex (if

Algorithm 6: Algorithm for MultiGraph_Hybrid

```
1  void MultiGraph_Hybrid()
2      EDP = MultiGraphD_sample()
3      mode = init_processing_mode()
4      if mode == sparse then
5          frontier = init_frontier()
6      end
7      else
8          frontier_active = init_frontier_active()
9      end
10     while not_converged do
11         if mode == dense then
               // sample_frontier is just a small part of frontier
12             sample_frontier = generate_sample_frontier(frontier_active)
13         end
14         else
15             sample_frontier = select_part(frontier)
16         end
17         ESP = MultiGraphS_sample(sample_frontier)
18         if EDP ≥ ESP then
19             if mode == sparse then
20                 frontier_active = generate_frontier_active(frontier)
21             end
22             mode = dense
23             frontier_active = MultiGraphD()
24         end
25         else
26             if mode == dense then
27                 frontier = generate_frontier(frontier_active)
28             end
29             mode = sparse
30             frontier = MultiGraphS()
31         end
32     end
```

any) to the allocated space. Switching from MultiGraphS to MultiGraphD is done by first resetting the frontier_active flag, and then marking the flags corresponding to the sparse front as active.

3.7 Experimental Evaluation

Table 3.3: Dataset description

Dataset						
Type	**Name**	**—V—**	**—E—**	**Max Degree**	**Dia.**	**Type**
Scale-free	rmat_s24_e16	16.8M	520.0M	434813	6	s
	rmat_s23_e32	8.4M	506.4M	438471	6	s
	rmat_s22_e64	4.2M	484.0M	428234	6	s
	indochina-04	7.4M	302.0M	256425	26	r
	soc-orkut	3M	212.7M	27466	9	r
	kron_g500-logn21	2.1M	182.1M	213904	6	s
	hollywood-09	1.1M	112.8M	11467	11	r
	soc-Livejournal1	4.8M	85.7M	20333	16	r
Mesh-like	rgg_n_24	16.8M	265.1M	40	2622	r
	road_USA	23.9M	57.7M	40	6809	s
	roadnet-CA	2M	5.5M	12	849	r

s: synthetic; r: real-world

Table 3.4: Machine configuration

Resource	**Details**
Host CPU	Intel core i7-2600 (4 cores, 3.40 GHz, 8MB L3cache), 16GB DDR3-1333)
GPU	Tesla K40c (15 Kepler SMs, 192 cores/MP, 12 GB Global Memory, 745 MHz, 1.5MB L2 cache, ECC off)

In this section, experimental results are presented, comparing MultiGraph with other GPU graph processing frameworks: CuSha, Groute, Gunrock, and Warp-Segmentation

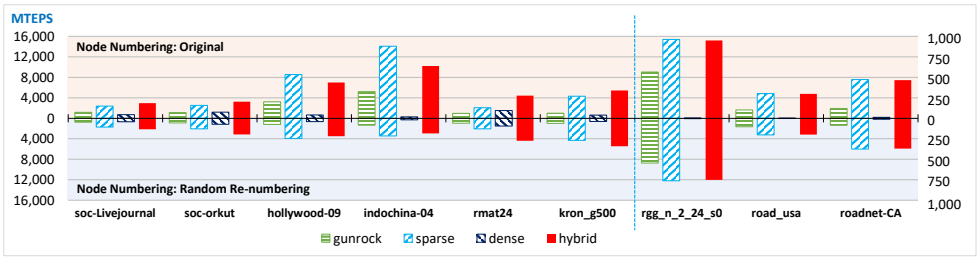


Figure 3.8: Performance: Betweenness Centrality

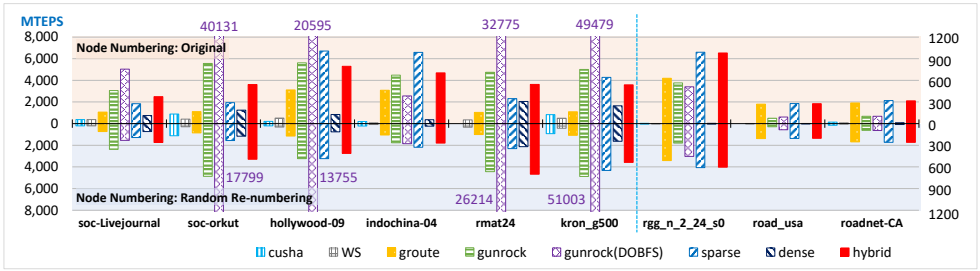


Figure 3.9: Performance: Breadth First Search

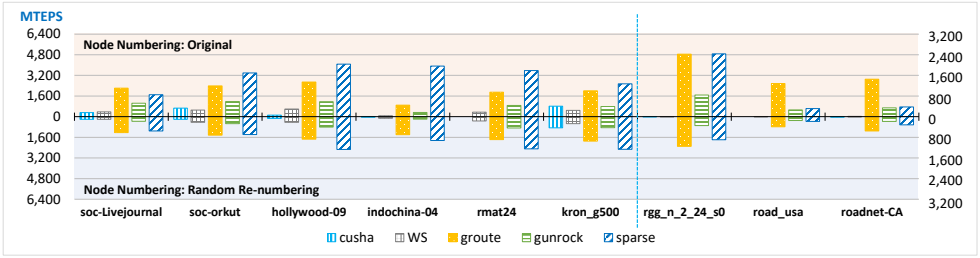


Figure 3.10: Performance: Connected Components

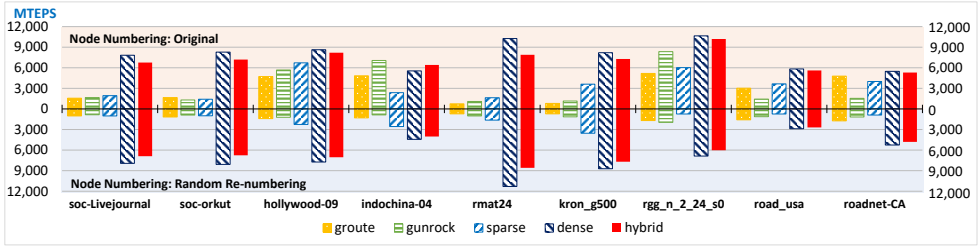


Figure 3.11: Performance: Pagerank (Data-Driven)

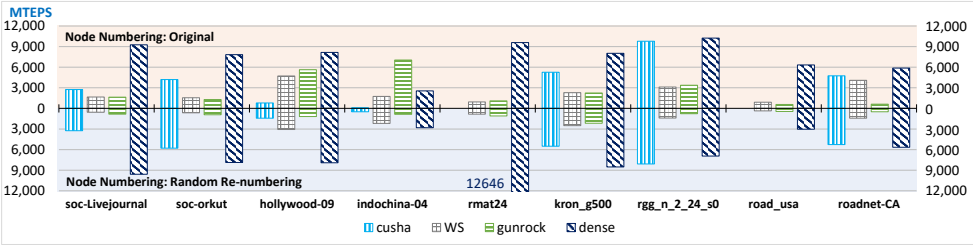


Figure 3.12: Performance: Pagerank (Topology-Driven)

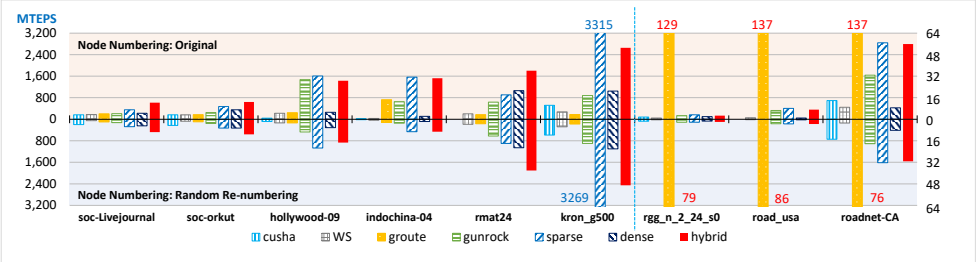


Figure 3.13: Performance: Single Source Shortest Path

(WS). All experiments were performed on an NVIDIA Tesla K40c GPU with 12GB memory, hosted by an Intel Xeon system. Table 3.4 provides details about the system. ECC was turned off for the tests.[4] For all evaluations, NVIDIA's NVCC compiler (version 7.5) was used, with the -O3 flag. We report performance for MultiGraph and other frameworks with all data already in GPU global memory. In reporting achieved performance (throughput), we do not include time for the input of graphs from file, any pre-processing costs for the graphs, or the time for data transfer from CPU to GPU. Later in this section, we separately report pre-processing overhead for each test dataset. All tests were run 10 times and the average was used for the reported results. The Gunrock team actively maintains a website (http://gunrock.github.io) with performance data on a range of graph datasets for

[4]Experiments were also conducted with ECC turned on. There was very little difference in performance, well under 5%.

various graph processing frameworks. The dataset includes eight scale-free datasets and three mesh-like datasets We used the same datasets, downloaded from their website (except for the synthetic datasets RMAT22, RMAT23, and RMAT24, which were generated using the RMAT generator [63]). Characteristics of the datasets are shown in Table 3.3.

3.7.1 Performance comparison with other graph frameworks

The performance achieved with MultiGraph was compared with several available GPU graph processing frameworks, including both topology-driven frameworks (CuSha, WS (Warp Segmentation)) and data-driven frameworks (Groute, and Gunrock version 0.4). The following standard graph algorithms were evaluated:

Breadth-First Search (BFS): For each iteration, the BFS level of all vertices connected to a vertex in the current frontier are updated. We note that Gunrock provides a "Direction-Optimized (DO)-BFS" which is very powerful for scale-free graphs, but we only implemented standard BFS. In order to enable comparison of the various frameworks on the same algorithm, we present results for Gunrock with DOBFS disabled, as well as with DOBFS enabled.

Single-Source Shortest Path (SSSP): For each iteration, relaxation is performed for all vertices connected to any vertex in the active frontier. We did not use any other optimization techniques like priority queues.

Connected Components (CC): We implemented Gunrock's CC algorithm [1].

Pagerank (PR): We developed two variants: i) topology-driven PR – CuSha's topology-driven pagerank algorithm, and ii) data-driven PR – Gunrock's data-driven pagerank algorithm.

Betweenness Centrality (BC): We implemented Gunrock's BC algorithm [1].

In order to optimize performance using Groute for BFS and SSSP, we auto-tuned two parameters: "prio_delta_fused" and "prio_delta_softprio". For Gunrock, we used tuned parameters used by the authors and reported in their publications.

We report performance in pseudo Millions of Traversed Edges Per Second (MTEPS), where the MTEPS metric is derived as follows: For BFS, SSSP, PR, and CC, MTEPS is computed as $|E|$/t, where $|E|$ is the number of edges in the graph and t the execution time in $\mu secs$. For BC, MTEPS is defined as $2\,|E|$/t, since each edge is visited in the forward pass and again in the backward pass. Since the execution times vary significantly across the graphs, this normalized measure is more convenient. Even if different frameworks actually process a different number of edges (an edge may be relaxed multiple times, and the number of times could differ from one implementation to another), the MTEPS measure is inversely proportional to t, and allows a fair comparison of the ratios of completion times of different frameworks. The data structure for CuSha requires more space than the other frameworks. On the test GPU, there was insufficient global memory to hold RMAT22, RMAT23, RMAT24, and roadnet_USA. Hence, performance data is not available for these datasets with CuSha.

For all benchmarks, the relative performance trends for RMAT22 and RMAT23 were found to be very similar to that of RMAT24, with RMAT23 having slightly higher performance, and RMAT22 a bit higher still. So we only display performance for RMAT24. For all benchmarks, we also tested the different frameworks on performance sensitivity to a random renumbering of graph vertices. We observed that many datasets use contiguous numbering for many neighbor vertices. This is very favorable for a CSR representation, where access to needed vertex values to process a set of edges achieves a high degree of coalescing. In the bar charts, the performance with the given dataset order faces upward,

while the corresponding data with randomly renumbered vertices faces downward. A general conclusion is that except for CuSha and MultiGraphD, frameworks are quite sensitive to vertex numbering, often suffering around 30% loss of performance.

Breadth-First Search (BFS): Fig. 3.9 presents performance data in MTEPS for BFS. CuSha and WS use topology-driven execution. For each iteration, all edges and vertices are traversed. Groute and Gunrock use data-driven BFS. For scale-free graphs, 6-26 iterations are required to find all reachable vertices with a bulk synchronous model for the datasets. In contrast, for mesh-like graphs, 555-6626 iterations are needed for BFS. CuSha and WS achieve very low performance due to the data-driven execution. Performance of MultiGraph is generally on a par with or better than the other frameworks, except for Gunrock, which is faster than MultiGraph on four of the nine datasets. With DOBFS enabled, Gunrock is up to an order of magnitude faster on several datasets. Also, for scale-free graphs, Gunrock performs better than Groute. On the other hand, for mesh-like graphs, Groute performs better than Gunrock.

For scale-free graphs, the hybrid algorithm is better than either MultiGraphD or MultiGraphS. For mesh-like graphs, the dense-frontier algorithm MultiGraphD is never activated. For each iteration, the sparse algorithm processes the dataset. When we use the hybrid algorithm, a small mode selection overhead is incurred, so that MultiGraphS performs a little better than the hybrid algorithm.

Single-Source Shortest Path (SSSP): Performance data are shown in Fig. 3.13. The difference between BFS and SSSP performance is due to the usually much larger size of the frontier for SSSP. Several iterations have extremely large frontier size, and these iterations dominate the execution time. For these iterations, MultiGraphD significantly outperforms Gunrock, thereby resulting in high speedup relative to BFS. Groute is an asynchronous

algorithm, but it may suffer from the limitations described earlier. For mesh-like graphs, the dense algorithm is never activated. MultiGraph performance is better than Gunrock in this case, but the performance of Groute is significantly better than MultiGraph and others. Groute uses a soft priority scheduler and fused workers. These can significantly reduce the amount of redundant (useless) work performed in other frameworks.

Connected Components (CC): For scale-free graphs, MultiGraph performance is consistently better than other frameworks (Fig. 3.10). For road_USA and roadnet-CA, Groute is much better than other frameworks. The reason is that Groute uses a different connected components algorithm, which is well matched with the asynchronous GPU framework.

Pagerank (topology-driven PR and data-driven PR): Performance data is shown in Fig. 3.11. The topology-driven PR algorithm and data-driven PR algorithm are used in Gunrock update values of all vertices until convergence. For MultiGraph, the dense algorithm is always chosen, and MultiGraphD generally outperforms frameworks that are mainly targeted toward data-driven algorithms. CuSha uses a novel structure to process graphs. It is also geared toward topology-driven algorithms and performs better than MultiGraph in the case of rgg_n_2_24_s0.

Betweenness Centrality (BC): Performance data is shown in Fig. 3.8. Only Gunrock had an implementation of BC; so we compared MultiGraph only with Gunrock. BC has two stages: forward processing and backward processing. BFS is used for forward processing. The difference between BFS and BC is that in BC, every edge is visited twice (forward processing and backward processing). Hence, Gunrock cannot take advantage of edge skipping for BC as in BFS. For scale-free datasets, the performance mainly depends on a few iterations with a very large frontier. Similar to SSSP, for these iterations, MultiGraph chooses the dense algorithm, which outperforms Gunrock. We note that MultiGraphS also

achieves consistently higher performance than Gunrock. The reasons are efficient work-list management and use of a combination of shared and global memory atomics, instead of all global memory atomics. For mesh-like graphs, as with previous algorithms, MultiGraph always chooses the sparse algorithm, and achieves higher performance than Gunrock.

Tables 3.5 and 3.6 present geometric means of the speedup achieved by MultiGraph over other frameworks for original node numbering and random renumbering, respectively.

Tables 3.7 and 3.8 present data for the pre-processing overheads for MultiGraph – all previously presented performance data excludes any pre-processing time for MultiGraph and any of the other frameworks that require transformation from a standard CSR representation. Table 3.7 presents measured pre-processing time for each benchmark algorithm and each graph dataset, both for the original numbering and the randomized renumbering of vertices for the datasets. The measured time accounts for the conversion of the graph data from a standard COO format to the data-structures required for MultiGraphD and MultiGraphS. This data conversion is performed in the GPU, and the reported time does not include the time for reading in data from files to the host CPU and transfering the data to the GPU global memory. In Table 3.8, aggregated normalized data is presented as a relative fraction of the actual execution time for each of the tested algorithms (geometric mean across the graph datasets). Optimization of pre-processing overheads has not been a focus so far, and it is expected that significant reduction of this overhead is feasible.

3.7.2 Hardware performance metrics

In this section, we present some metrics from hardware counter measurements that validate design decisions for MultiGraph. Due to space limitations, we only show data for one

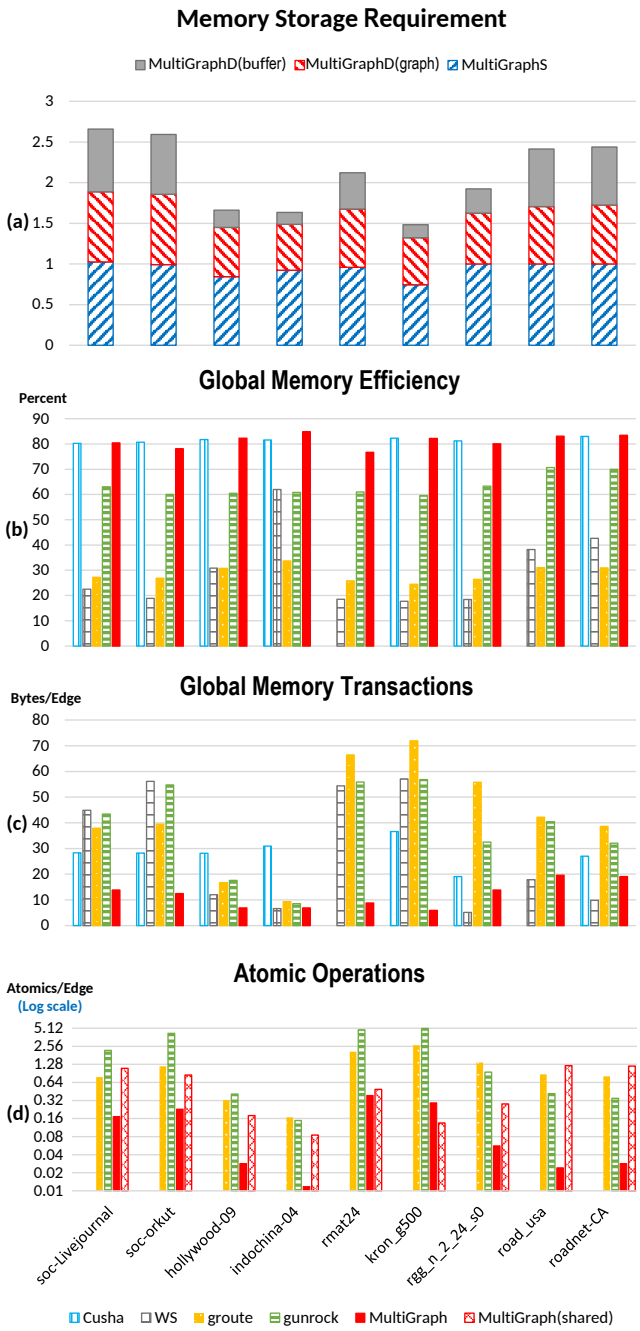


Figure 3.14: Comparison of memory usage and hardware metrics across systems (PR algorithm)

Table 3.5: Speedup: original node numbering

Benchmark	CuSha	WS	Gunrock	Groute
BFS	16.02	27.30	1.20	1.93
SSSP	7.77	7.04	1.86	1.28
CC	29.81	24.82	2.78	1.01
PR(top)	3.71	3.79	4.14	-
PR(dat)	-	-	4.07	3.95
BC	-	-	2.90	-
average	10.83	11.60	2.59	1.77

Table 3.6: Speedup: random re-numbering

Benchmark	CuSha	WS	Gunrock	Groute
BFS	10.96	33.51	1.27	2.05
SSSP	4.05	8.75	2.17	1.53
CC	18.95	22.07	2.81	0.95
PR(top)	2.33	6.13	8.01	-
PR(dat)	-	-	6.96	6.36
BC	-	-	2.97	-
average	6.65	14.11	3.30	2.09

of the evaluated graph algorithms: Pagerank. Trends for other algorithms are also broadly similar. Figure 3.14(a) shows memory storage requirements for the various datasets, normalized to the storage requirements for the standard CSR representation used by Gunrock, Groute, and WS. The total memory requirement is computed as the sum of the memory required for heavy edge block representation, light edge block representation in MultiGraphD, and sparse representation in MultiGraphS. The space overhead for MultiGraph over CSR ranges from 1.5x to 2.5x.

Table 3.7: Preprocessing time: original/renumbered (in ms)

Dataset	BC	BFS/PR	CC	SSSP
soc-Livejournal1	130(200)	128(198)	6.7(6.7)	189(280)
soc-orkut	311(425)	312(419)	13.5(12.7)	473(598)
hollywood-09	191(270)	242(369)	7(6.3)	332(469)
indochina-04	391(557)	453(595)	19.5(16.5)	681(827)
rmat24	1009(1049)	1092(1122)	33.7(26.8)	1655(1681)
kron_g500	675(668)	1077(1065)	9.8(9.9)	1249(1235)
rgg_n_2_24_s0	188(633)	133(588)	15.8(16)	149(828)
road_USA	105(177)	87(144)	8.1(8.1)	111(166)
roadnet-CA	13.2(12.8)	11.8(11.3)	0.7(0.7)	12.5(15.3)

Table 3.8: Normalized pre-processing overhead

Algorithm	Original	Renumbered
BC	1.97	1.97
BFS	3.19	3.40
CC	0.13	0.07
PR(data-driven)	0.55	0.66
PR(topology-driven)	0.15	0.21
SSSP	0.57	0.57

Figure 3.14(b) presents the global memory efficiency for each framework, computed as follows. For each GPU kernel, the number of global memory load/store transactions and global load/store efficiency were collected using NVPROF. The sum of the product of global memory load/store transactions and the corresponding global load/store efficacy was divided by the total number of load/store transactions to obtain an average global memory efficiency for the execution. It can be seen that MultiGraph and CuSha attain consistently high global memory load/store efficiencies of around 80%. WS and Groute exhibit the

lowest global memory efficiency, well under 50% for all datasets. Gunrock achieves efficiencies between 60- and 70%. The main reason for the higher global memory efficiency for CuSha and MultiGraph is the lower incidence of uncoalesced data accesses, due to the alternate data-structures and edge processing strategies used.

Figure 3.14(c) displays statistics on measured global memory transactions for different frameworks across different datasets. The metric is presented normalized to the total number of graph edges traversed (graph edges multiplied by the number of iterations). It was computed as the sum of DRAM loads/stores (obtained from NVPROF) multiplied by 32 (conversion to bytes) and divided by the product of the total number of edges and the number of PR iterations. This metric varies across a range from around eight bytes/edge to 72 bytes/edge across the frameworks and datasets. MultiGraph incurs data movement ranging from 8- to 20 bytes/edge, while Gunrock incurs consistently higher data movement, ranging around 10-55 bytes/edge. Groute also requires higher data movement than MultiGraph, ranging around 10-70 bytes/edge. CuSha requires slightly higher global memory data movement than MultiGraph, and has relatively low variance across datasets, ranging around 20-35 bytes/edge. In contrast, WS has the highest variance in data movement volume across datasets, with volumes in some cases being as low as five bytes/edge (lower than MultiGraph) and as high as 55 bytes/edge.

Figure 3.14(d) presents data on the number of atomic operations, again normalized to the number of processed graph edges. The number of global atomic operations was directly obtained using NVPROF. A significant reason for the improved performance of MultiGraph over other frameworks is the reduction in the number of global memory atomics by use of atomics in shared-memory instead. Since hardware counters are not available on the NVIDIA Kepler for shared-memory atomics, it was estimated as follows. MultiGraphS was

computed as the sum of the number of edges in groups 1 to 4 for the active input vertices. The number of shared memory atomic operations for the MultiGraphD was computed as the sum of i) the number of active destination vertices in each heavy edge block, ii) 1024 * number of heavy edge blocks, and iii) sum of the number of active edges in each light edge block. The total number of global atomic operations for MultiGraph (red solid bars) can be seen to be significantly lower than Gunrock and Groute (we note that the y-axis scale for this chart is logarithmic). For MultiGraph, global atomics have been replaced by shared-memory atomics (red bars with a criss-cross pattern), which incur much lower overheads than global atomics.

3.8 Summary

This chapter has addressed the development of a high-performance approach to graph processing on GPUs. It uses multiple data representation and execution strategies for dense- versus sparse-vertex frontiers, dependent on the fraction of active graph vertices. A two-phase edge processing approach trades off extra data movement for improved load balancing across GPU threads, by using a 2D blocked representation for edge data. Experimental results demonstrate performance improvement over current state-of-the-art GPU graph processing frameworks for many benchmark programs and datasets.

CHAPTER 4

Efficient Sparse-Matrix Multi-Vector Product on GPUs

4.1 Introduction

Sparse Matrix-Vector (SpMV) multiplication and Sparse Matrix Multi-vector (SpMM) multiplication (also called SpMDM – Sparse Matrix Dense Matrix multiplication) are key kernels used in a range of applications. SpMV computes the product of a sparse matrix and a (dense) vector.

Although SpMM may be viewed as a set of independent SpMV operations on different dense vectors, and therefore can be computed by using an SpMV kernel repeatedly, it is very beneficial in terms of achievable performance to implement a separate SpMM kernel, because of the significantly greater data locality that can be exploited by doing so. However, as shown in the next section using *Roofline* [143] performance bounds, the achieved performance with the NVIDIA cuSPARSE library SpMM implementation achieves a significantly lower fraction of roofline limits when compared to the cuSPARSE SpMV implementation.

In this chapter, we seek to answer the following question: *Is it feasible to achieve a significantly higher fraction of the roofline upper-bound for SpMM on GPUs, just as has already been accomplished by various implementations of SpMV [90, 118, 76] on GPUs?*

We undertake a systematic analysis of the SpMM problem and develop a new implementation that is demonstrated to be significantly faster than other currently available GPU implementations of SpMM – in cuSPARSE [2], MAGMA [7], and CUSP [32]. A key observation that drives the new implementation is the fact that non-zeros in sparse matrices drawn from a variety of application domains are not uniformly spread over the row/column index space, but exhibit non-uniform clustering of elements. We exploit this property to partition sparse-matrix elements into two groups: one group containing clustered segments and the other holding the remaining scattered elements. Different processing strategies are used for the two partitions, with much higher data reuse and lower overheads being achievable for the clustered partition.

4.2 Background

Graphics Processing Units (GPUs) are very attractive for sparse matrix computations due to their high memory bandwidth and computing power. However, achieving high performance is nontrivial due to the irregular data access pattern. A number of recent efforts have addressed the development of efficient SpMV for GPUs [118, 90, 76, 149, 69, 144, 30, 16, 9]. SpMM is also a key kernel in sparse computations in computational science and machine-learning/data-science, but very few studies have to date focused on it. [147, 148, 24, 88, 93, 59, 12, 11, 104]. SpMM is a core kernel for the Locally Optimal Block Preconditioned/conv Conjugate Gradient (LOBPCG) method [67, 4, 7], aerodynamic design optimization [107], PageRank algorithm [11], sparse convolutional neural networks (CNNs) [43, 104], image segmentation in videos [122], atmospheric modeling [11], etc.

Roofline [143] performance upper-bounds based on global memory bandwidth on GPUs can be developed for SpMV and SpMM, as follows. Consider single-precision floating-point computation with a square $N \times N$ sparse matrix with nnz non-zero elements. Each element of the sparse matrix in standard CSR representation requires $8 \times nnz + 4 \times (N + 1)$ bytes of storage. Including the storage for the input and output dense vectors, the total data footprint for SpMV is $8 \times nnz + 12 \times N + 4$ bytes. The total floating-point operation count is $2 \times nnz$, i.e., one multiply-add operation for each non-zero element in the sparse matrix. The input vector and sparse matrix reside in GPU global memory before execution of the SpMV kernel, and the resulting vector is stored in global memory after execution. Thus, a minimum volume of $8 \times nnz + 12 \times N + 4$ bytes of data must be moved between global memory and GPU registers across the memory. Dividing this data volume by the peak global memory bandwidth BW_{GM} of the GPU (e.g., 750 Gbytes/sec. for an NVIDIA Pascal P100 GPU) gives the minimum time for the global memory data transfers, and the roofline upper bound performance of $\frac{2 \times nnz \times BW_{GM}}{8 \times nnz + 12 \times N + 4}$. Similarly, for SpMM, the total data footprint is $8 \times nnz + 8 \times K \times N + 4 \times N + 4$ bytes and the total floating-point operation count is $2 \times K \times nnz$, giving a roofline performance upper-bound of $\frac{2 \times K \times nnz \times BW_{GM}}{8 \times nnz + 8 \times K \times N + 4 \times N + 4}$, where K is the number of vectors (the width of dense matrices).

Fig. 4.1 displays achieved SpMV and SpMM performance in GFLOPs by NVIDIA's cuSPARSE library on a Pascal GP100 GPU, for the entire set of 2720 sparse matrices of the SuiteSparse [3, 35] collection (formerly known as the University of Florida Sparse Matrix collection). In these charts, data for the sparse matrices is plotted by sorting the matrices along the X-axis in increasing order of the number of non-zeros (nnz), with one point in the scatter-plot for each matrix. In Fig. 4.1(a), for each sparse matrix, a blue dot represents the achieved performance in GFLOPs, and a corresponding red dot placed vertically above

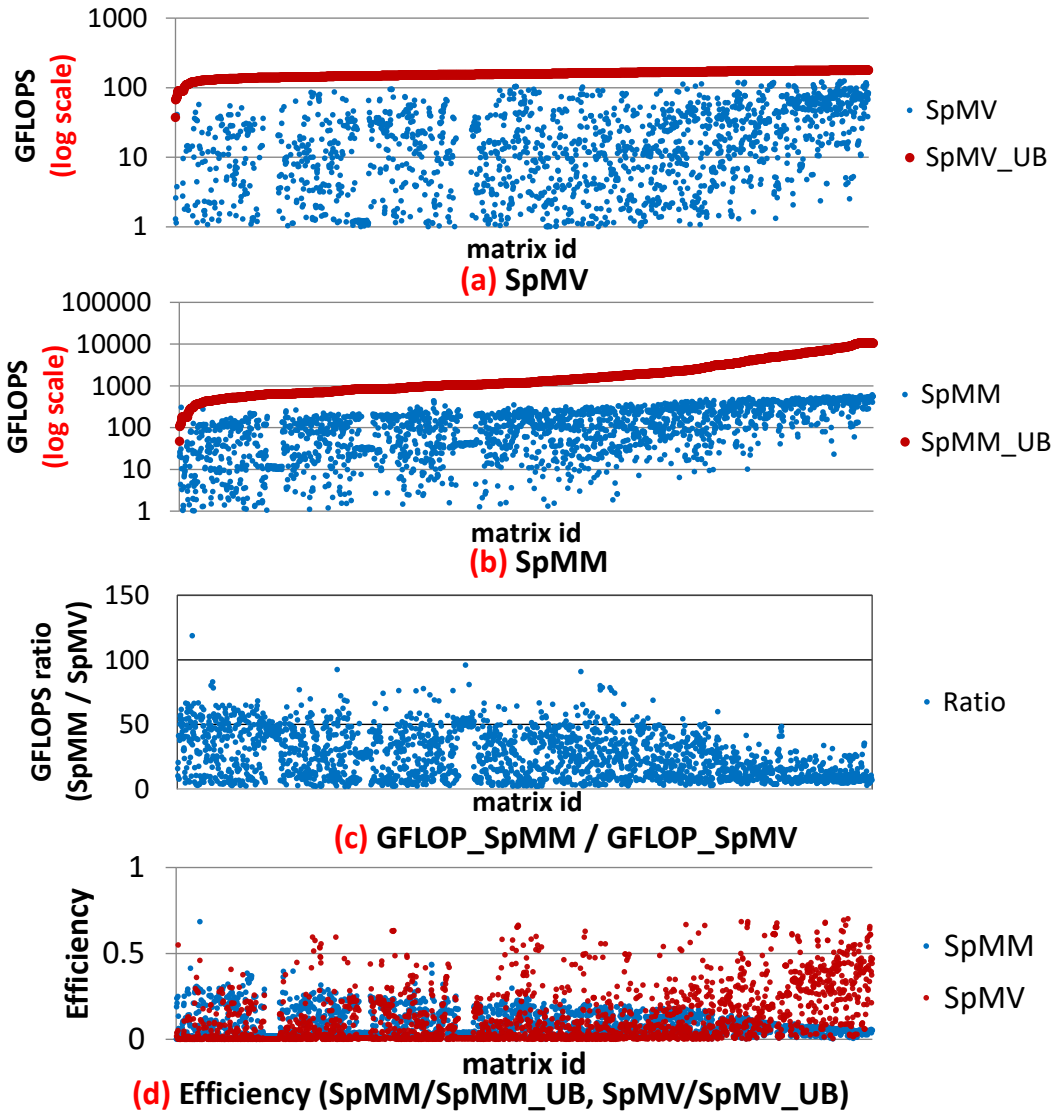


Figure 4.1: cuSPARSE SpMV/SpMM performance and upper-bound: NVIDIA Pascal P100 GPU

it marks the roofline performance upper-bound for that matrix. The performance upper-bound is around 170 GFLOPs (does not vary too much across matrices). cuSPARSE SpMV performance approaches the roofline bound for around 670 matrices.

Fig. 4.1(b) displays achieved performance and roofline upper-bounds for the same matrices with cuSPARSE SpMM. The achieved absolute performance can be seen to be much higher than SpMV, but the gap between actually achieved performance and the roofline upper-bound is also much higher. Fig. 4.1(c/d) presents the same data as in Fig. 4.1(a/b), respectively, but expresses achieved performance as a fraction of the roofline upper-bound (efficiency). It is clear that cuSPARSE SpMM achieves a significantly lower fraction of the roofline bound than SpMV.

Pseudocodes for sequential SpMV and SpMM are shown in Alg. 7 and Alg. 9, respectively. The sparse matrix S is stored in the standard CSR (Compressed Sparse Row) format. The non-zero elements are compacted and stored in S.values[:], with all non-zeros in a row being placed contiguously. S.rowptr[i] points to the first non-zero element from row i in S.values[:]. S.colidx[i] holds the column index of the corresponding non-zero located in S.values[i]. Fig. 4.2(b) shows an example of a sparse matrix stored in the CSR format. The Doubly Compressed Sparse Row (DCSR) format [19] further compresses CSR by only storing non-empty rows. As seen in Fig. 4.2(c), indices of non-empty rows are managed in S.rowidx[:] and S.rowptr[i] points to the first non-zero elements from row S.rowidx[i].

SpMV (Alg. 7) iterates over the rows of S, forming the sparse dot-product of the ith row of S with the input dense vector D to produce the ith element of the output dense vector O.

SpMM (Alg. 9) iterates over the rows of S, forming the sparse dot-product of the ith row of S with the kth dense vector, $D[][k]$, to produce the ith element of the kth output dense vector, $O[i][k]$.

Aktulga et al. [4] describe an SpMM scheme based on the compressed sparse blocks (CSB) format, optimized for both transposed and non-transposed SpMM ($O = SD$ and

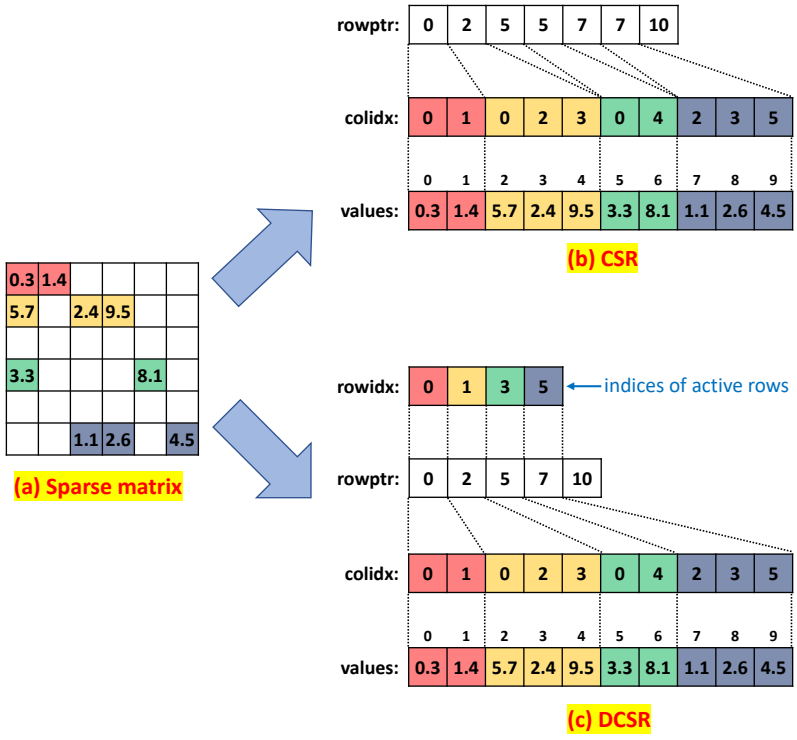


Figure 4.2: CSR and DCSR formats

$O = S^T D$). The elements of the sparse matrix are partitioned into blocks of size $\beta \times \beta$. Each block is represented in coordinate format (COO). For non-transposed SpMM ($O = SD$), threads process row blocks of size $((num\_threads \times \beta) \times N)$. For transposed SpMM ($O = S^T D$), threads process column blocks of size $(N \times (num\_threads \times \beta))$.

Anzt et al. [7] develop an SpMM scheme optimized for GPUs, based on the SELL-P format. The SELL-P format is built by partitioning rows of the sparse matrix into blocks, and then each row block is converted into ELLPACK format, with the rows being padded so that the row length of each block is a multiple of the number of threads. The threads are organized into 3D blocks, where the x-dimension maps to a row within a SELL-P block, the y-dimension maps to columns, and the z-dimension maps to multiple vectors. Since

multiple threads are assigned to process the elements of the same row, reduction operations are required and performed in shared memory.

FastSpMM [101, 133] uses ELLPACK-R [129] to enhance performance by storing the sparse matrix in a regular data structure. However, this strategy may suffer when processing very irregular sparse matrices. cuSPARSE [2] from NVIDIA also supports SpMM. It offers two modes: i) T: Transposed and ii) NT: Non transposed. In BidMach [26], SpMM is implemented as one of the many kernels and internally uses the cuSPARSE format.

Algorithm 7: SpMV: Sparse Matrix-Vector Multiplication.

```
   input : CSR S[M][N], float D[N]
   output: float O[M]
1  for i = 0 to S.rows-1 do
2  |   for j = S.rowptr[i] to S.rowptr[i+1]-1 do
3  |   |   O[i] += S.values[j] * D[S.colidx[j]]
4  |   end
5  end
```

Algorithm 8: SpMM: Sparse Matrix Multi-Vector Multiplication.

```
   input : CSR S[M][N], float D[N][K]
   output: float O[M][K]
1  for i = 0 to S.rows-1 do
2  |   for j = S.rowptr[i] to S.rowptr[i+1]-1 do
3  |   |   for k = 0 to K-1 do
4  |   |   |   O[i][k] += S.values[j] * D[S.colidx[j]][k]
5  |   |   end
6  |   end
7  end
```

4.3 RS-SPMM

In this section, we present the proposed GPU SpMM algorithm, labeled RS-SpMM for *Row-Segmented SpMM*. The name derives from the fact that the sparse matrix is partitioned into two parts, one holding clustered non-zero row-segments, enabling higher data reuse in the GPU and thus lower data movement from global memory than previously developed SpMM approaches.

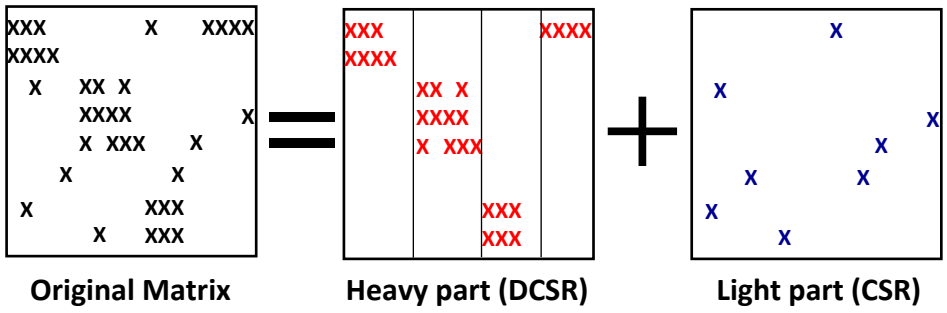


Figure 4.3: Splitting sparse matrix into heavily clustered row-segments and remainder

Fig. 4.3 illustrates the splitting of a sparse matrix into two matrices, one holding non-zeros in heavily clustered row-segments, and the other holding the remaining non-zeros that are randomly scattered over the column-index space in each row. The rationale for this splitting is elaborated below, and is based on the observation that large sparse matrices found in practice generally do not exhibit a fully random distribution of their non-zeros in the row/column space. A sizable fraction of non-zeros tend to be grouped in clusters in the row/column index space.

In SpMM (also known as SpMDM: Sparse Matrix Dense-Matrix product), a sparse matrix is multiplied with a dense matrix to produce a dense matrix. SpMV can be seen as a

special case of SpMM, where the width of the dense input matrix is one. In the remainder of this section, we refer to the input sparse matrix as **S** ($M \times N$), the input dense matrix as **D** ($N \times K$), and the output dense matrix as **O** ($M \times K$).

When compared to SpMV, SpMM has the following properties: i) unlike SpMV, the sparse matrix elements have a reuse factor of K; the reuse of input/output dense matrix elements is similar to the SpMV; ii) unlike SpMV, it is possible to achieve coalesced access of the dense input matrix (due to width K); iii) similarly, it is possible to achieve coalesced access on the output.

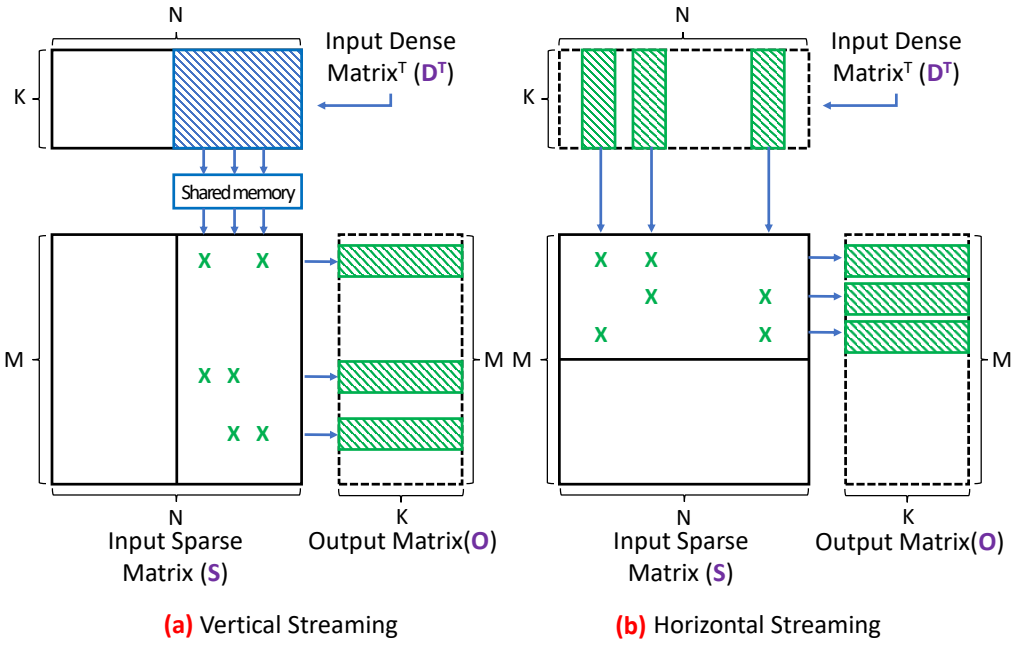


Figure 4.4: Vertical and horizontal streaming

Here, we discuss two options for SpMM: *vertical streaming* and *horizontal streaming* (Fig. 4.4). As with dense matrix multiplication, tiling is also crucial to optimizing performance of SpMM. In general, tiling of all of the three loops $\{i, j, k\}$ is feasible in Alg. 9.

However, the nature of reuse for the three arrays $\{D, S, O\}$ in SpMM is along distinct loops: the i loop for D, the j loop for O, and the k loop for S. Since the data footprint of an array is invariant with respect to iterations of the "reuse" loop index, it means that one of the three arrays will have an invariant data footprint and therefore will achieve complete reuse as the innermost tile is changed in a 3D tiled execution. This maximal reuse is independent of the chosen tile size; therefore, it is best to minimize this tile size, thereby enabling maximization of the other two tile sizes to achieve as much reuse as possible for the other two arrays. This observation leads to what is referred to as *streamed* execution along that innermost tile dimension, which is equivalent to performing only 2D tiling over two out of the three loops in Alg. 9. Of the three arrays in SpMM, S has a reuse factor of K, while D and O have reuse factors corresponding to the average number of non-zeros in S along a column and row, respectively. Typically, K is much larger than the average number of non-zeros in a row/column. Further, achieving reuse along k for each element in S is much easier, since the dense matrix elements in D and O accessed for such reuse are contiguously located in global memory. Therefore, it is best to use i or j as the streaming direction for SpMM. Streaming along i is referred to as *vertical streaming* and streaming along j is called *horizontal streaming*.

For vertical streaming (Fig. 4.4(a)), the input sparse matrix (**S**) is partitioned into column panels. Each column panel is processed by a thread block. A thread block collectively brings the **D** elements corresponding to the column panel into shared-memory. Different warps in a thread block process different rows within the column panel. The threads of a warp are mapped across "K", that is, each thread is responsible for computing the partial result of a single output element for that column panel. All threads in a warp compute the

product of the same non-zero element of the sparse matrix with a distinct input matrix element, and then they move to the next sparse matrix element in the row. Threads accumulate partial results in thread local registers, and at the end of processing of each row-segment in a column panel, the partial contribution is moved from registers to global memory corresponding to the output dense matrix using atomic operations. Even though this scheme achieves coalesced access on input and output (with sufficiently large K), the downside is that it requires atomic operations for accumulating partial results to global memory. For the remainder of this section, we refer to this scheme as the vertical scheme.

In horizontal streaming (Fig. 4.4(b)), the input sparse matrix (**S**) is partitioned into row panels. Each row panel is processed by a thread block. Threads cyclically processes elements along the horizontal dimension of **S**. Each row is processed by a warp; and different threads in the warp are distributed across "K". Each thread accumulates the partial results in thread's local registers, and at the end of each row the threads move the partial contribution from registers to global memory corresponding to the output dense matrix using atomic operations. For the remainder of this section, this scheme is referred to as the horizontal scheme.

In both vertical and horizontal SPMM schemes, the threads in a warp are spread across K. This helps in i) avoiding intra-warp communication which would otherwise be required for reduction of partial products across the threads in the same warp, and ii) not requiring shared memory for holding the intermediate results. If the threads in a warp are distributed across a row of the sparse matrix, the amount of work for different threads can vary widely, leading to intra-warp load imbalance. The downside of distributing the threads in a warp across the columns of the input matrix is that, if the number of columns of the input matrix (K) is not a multiple of 32, the last column slice will not be load balanced. However, in

practice, the last scheme works better, as the load imbalance across the non-zero elements in a row-segment of a column partition is much worse than the load imbalance across the rows of the input dense matrix.

The vertical scheme is not beneficial if the average number of elements in a row of a column panel is low. Note that in the vertical scheme, at the end of processing each row in the column panel, there is an expensive atomic update to global memory. If the number of elements per row-segment is small, the relative overhead of the atomic operations is more prominent. However, if the average number of elements in a row-segment in a column panel is high, then the vertical scheme is beneficial as it gets full reuse of elements in **D**. The disadvantage of the horizontal scheme is that there is no reuse of elements in **D** (except for possibly from cache).

This motivates a dual scheme where the rows with the number of non-zero elements in a column panel larger than a threshold are processed using vertical streaming and the rest of the rows are processed using horizontal streaming. In the remainder of the section, rows whose non-zero count within a column panel is greater than a parametric threshold are called **heavy rows** and the others are called **light rows**.

4.3.1 Data Structure

We use a CSR representation for the light rows and a DCSR representation for the heavy rows. All of the heavy row-segments in each column panel are represented by a DCSR structure. All of the light rows in the entire sparse matrix **S** are represented by a CSR matrix. Fig. 4.5 illustrates the hybrid data representation. For the illustration, the threshold for a row to be classified as heavy is two; column partition width is four. Blocks with yellow background are the heavy rows in column partition zero. Similarly, blocks

with green background are the heavy rows in column partition 1. The blocks with white background represent the light rows. Details of the representation of the heavy and light blocks are shown in Figs. 4.5 (b-3,b-4) and Fig. 4.5 (c-2), respectively.

The number of columns in **S** that are processed by a thread block (W) is chosen such that $W \times k \times sizeof(data\_type) \times no\_of\_active\_tb\_per\_sm$ is equal to the shared-memory capacity. In our experiments, "k" (k columns of input and output are processed by a thread block) is chosen as 64 and no_of_active _tb_per_sm is chosen as two. "k" should be a multiple of 32 for coalesced access (to avoid thread divergence). Choosing no_of_active_tb_per_sm as two helps in achieving maximum occupancy.

4.3.2 Algorithm

The pseudocode for the RS-SpMM scheme for heavy rows (vertical) is shown in Listing 4.1. Columns of **D** of size K are divided into slices of size k. Each column panel is processed by K/k thread blocks. All threads in a thread block collectively bring a slice of the input dense matrix corresponding to the column panel (**D**) to shared-memory (lines 5-7). The columns of **D** that are brought to shared-memory depend on the slice ID along K.

Each warp then processes the rows in the column panel in a cyclical fashion (lines 9-20). Each thread initializes the partial result (which will be held in a thread local register) to zero (line 10). All threads in a warp process the same non-zero element in the sparse matrix. Reading each sparse element one by one will result in uncoalesed access. In order to avoid this, the threads read 32 non-zero elements (lines 15-18) and then use warp shuffle to exchange elements (line 19). In the first iteration, all threads in a warp need index_matrix[start] and value_matrix[start], and these values are stored in the first lane (tid

% WARP_SIZE == 0). Hence, *value* and *index* held by the first lane are broadcast to other threads. At the second iteration, all threads in a warp need index_matrix[start+1] and value_matrix[start+1], which are stored in the second lane. Hence, values of the second lane are broadcast.

Each thread then computes the partial product in the thread local register (line 19). Finally, the accumulated partial results are updated to global memory using atomic operations (lines 21-23). Atomic operations are required, as different thread blocks can write to the same location.

The RS-SpMM algorithm for light rows (horizontal scheme) is shown in Listing 4.2. Each row of the light matrix is processed by a warp. In order to minimize inter-warp load imbalance, we chose the smallest thread block size (64 on Pascal) that maintains full occupancy. For example, on an NVIDIA Pascal GPU, since one block has only two warps, rows 0 and 1 are processed by the first block, row 2 and 3 are processed by the second block, and so on. Each thread initially computes the row and slice along K that it should process. Each thread initializes the partial result (held in a thread's local register) to zero (line 6). Similar to the SpMM heavy scheme, the column indices and values are collectively read by all threads in the warp (lines 10, 11) and broadcast to all threads in a warp (line 13). The partial products are then computed and accumulated in the thread local register (line 13). Finally, the partial result is accumulated to global memory using atomic operations (line 15). Atomic operations are still needed since the thread blocks for the heavy scheme and light scheme are launched concurrently.

Listing 4.1: SPMM pseudocode (heavy row segments)

```
1. row_offset = tb_idx * IN_TILE_ROW_SIZE;
2. slice_offset = tb_idy * IN_TILE_SLICE_SIZE;
3. warp_id = tid/WARP_SIZE;
4. lane_id = tid%WARP_SIZE;
5. for i=warp_id to IN_TILE_ROW_SIZE step tb.size()/WARP_SIZE do
6.    sm_input_value[i][lane_id] =
      input_value[row_offset+i][slice_offset+lane_id];
```

```
7. end
8. __syncthreads;
9. for i=seg_start_num[tb_idx] to seg_start_num[tb_idx+1]-1 step
        tb.size()/WARP_SIZE do
10.  val = 0;
11.  start = start_seg_position[i];
12.  end = start_seg_position[i+1];
13.  for j=start to end-1 do
14.    mod = (j - start)%WARP_SIZE
15.    if mod == 0 then
16.      index_buf = seg_index[j + lane_id];
17.      value_buf = seg_value[j + lane_id];
18.    end
19.    val += sm_input_value[__shfl(index_buf, mod)][lane_id]
             * __shfl(value_buf, mod);
20.  end
21.  row_idx = seg_row_position[i];
22.  //directly accumulate results in global memory
23.  atomicAdd(&dest_value[row_idx][slice_offset+lane_id], val);
24. end
```

Listing 4.2: SPMM pseudocode (light rows)

```
1.  row_offset = (tb_idx*tb.size() + tid) / WARP_SIZE;
2.  slice_offset = tb_idy * IN_TILE_COL_SIZE;
3.  lane_id = tid%WARP_SIZE;
4.  start = csr_row_pointer[row_offset];
5.  end = csr_row_pointer[row_offset+1];
6.  val = 0;
7.  for i=start to end-1 do
8.    mod = (i - start)%WARP_SIZE
9.    if mod == 0 then
10.     index_buf = csr_column_idx[i + lane_id];
11.     value_buf = csr_column_val[i + lane_id];
12.   end
13.   val += input_value[__shfl(index_buf, mod)][lane_id]
            * __shfl(value_buf, mod);
14. end
    //directly accumulate results in global memory
15. atomicAdd(&dest_value[row_offset][slice_offset+lane_id], val);
```

Thread coarsening is used for both the vertical and horizontal schemes to improve performance. Thread coarsening helps to reduce the global memory latency effects and helps in reducing the required number of warp-shuffle operations. Thread coarsening is done along "K". If the thread coarsening factor is one, each thread processes only one element in a slice. If the thread coarsening factor is two, then each thread processes two elements in a slice. In the remainder of this chapter, we refer to the thread coarsening factor for the vertical and horizontal schemes as CF_V and CF_H, respectively. Listing 4.3 shows pseudocode corresponding to thread coarsening factor CF_V of 2. As seen in lines 6 and

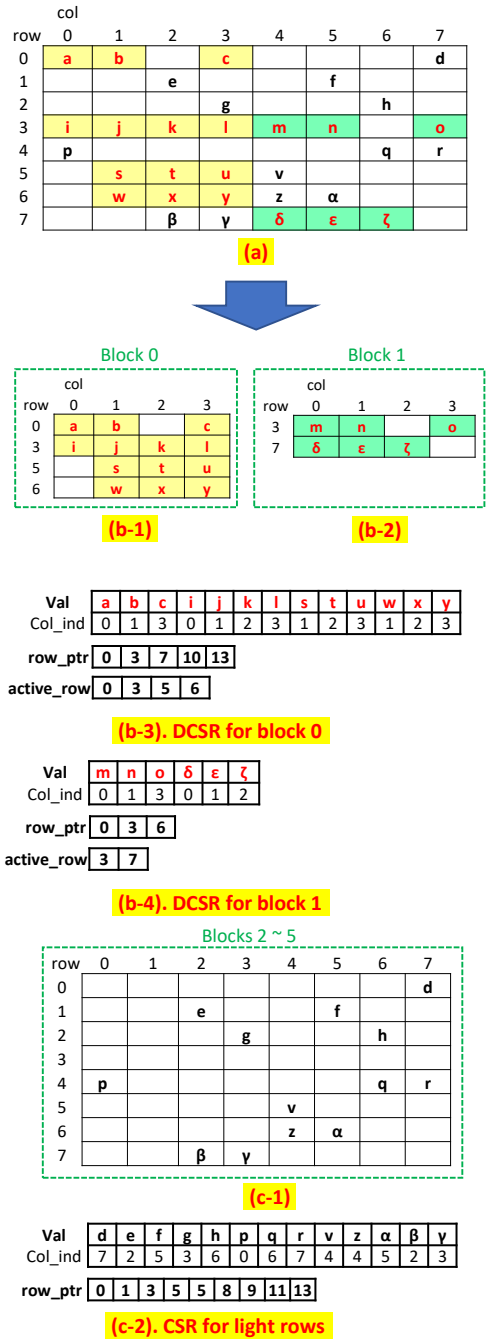


Figure 4.5: SpMM overview

7 in Listing 4.3, since there is no dependence between two global memory load instructions, they can be loaded concurrently, which helps in tolerating memory access latency. The number of warp-shuffle operations (lines 20, 21) processed by a thread block remains the same and thread coarsening reduces the number of thread block launched. Hence, the number of warp-shufle operations is reduced by half.

4.4 Modeling impact of slice-size choice

Listing 4.3: SpMM pseudocode (heavy rows), $CF_V = 2$

```
1. row_offset = tb_idx * IN_TILE_ROW_SIZE;
2. slice_offset = tb_idy * IN_TILE_SLICE_SIZE * 2;
3. warp_id = tid/WARP_SIZE;
4. lane_id = tid%WARP_SIZE;
5. for i=warp_id to IN_TILE_ROW_SIZE step tb.size()/WARP_SIZE do
6.   sm_input_value[i][lane_id] =
     input_value[row_offset+i][slice_offset+lane_id];
7.   sm_input_value[i][lane_id+WARP_SIZE] =
     input_value[row_offset+i][slice_offset+lane_id+WARP_SIZE];
7. end
8. __syncthreads;
9. for i=seg_start_num[tb_idx] to seg_start_num[tb_idx+1]-1 step
        tb.size()/WARP_SIZE do
10.  val1 = 0;
11.  val2 = 0;
12.  start = start_seg_position[i];
13.  end = start_seg_position[i+1];
14.  for j=start to end-1 do
15.    mod = (j - start)%WARP_SIZE
16.    if mod == 0 then
17.      index_buf = seg_index[j + lane_id];
18.      value_buf = seg_value[j + lane_id];
19.    end
20.    shfl_index = __shfl(index_buf, mod);
21.    shfl_value = __shfl(value_buf, mod);
22.    val1 += sm_input_value[shfl_index][lane_id] * shfl_value;
23.    val2 += sm_input_value[shfl_index][lane_id+WARP_SIZE] *
       shfl_value;
24.  end
25.  row_idx = seg_row_position[i];
26.  //directly accumulate results in global memory
27.  atomicAdd(&dest_value[row_idx][slice_offset+lane_id], val);
28.  atomicAdd(&dest_value[row_idx][slice_offset+lane_id+WARP_SIZE],
     val2);
29. end
```

In this section, we describe how we determine the threshold for classifying a row as light or heavy and choosing CF_V and CF_H. These factors were chosen based on a subset of matrices (training set) from SparseSuite [3]. First, we eliminated all matrices having

less than 100,000 non-zeros. Then we sorted all of the matrices in increasing order of the number of non-zeros, including every 20th matrix in the training set. In total, 43 matrices (5%) were selected.

Figure 4.6 presents an overview of the approach.

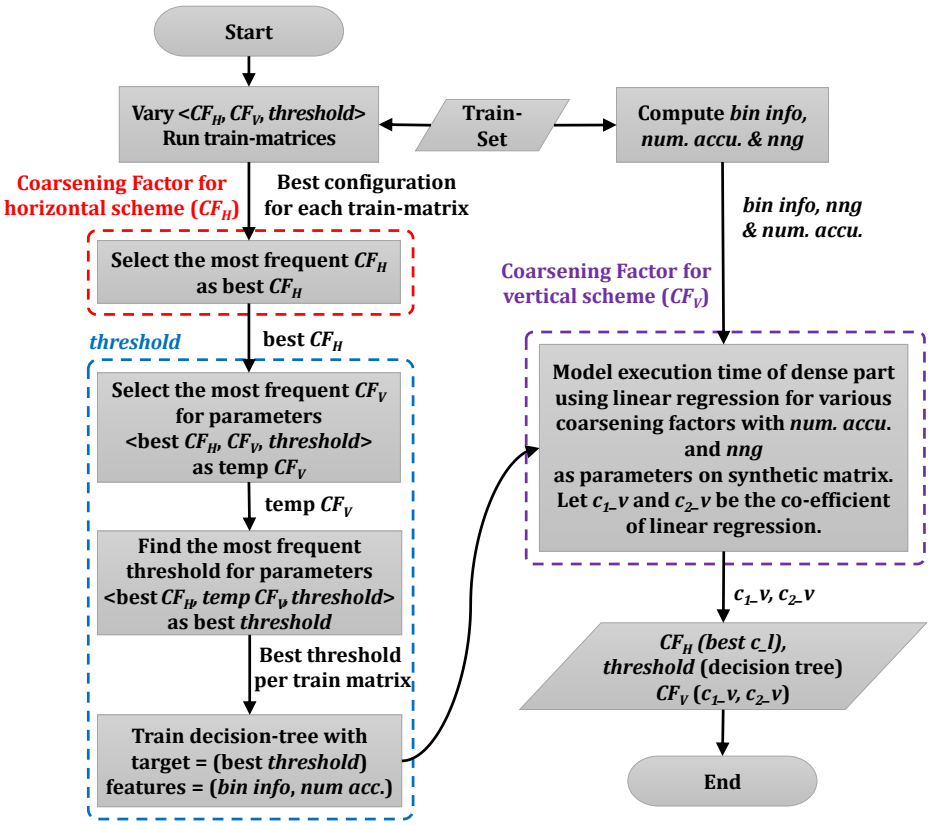


Figure 4.6: Modeling overview

4.4.1 The coarsening factor of the horizontal scheme (CF_H)

For the horizontal scheme, the coarsening factor along K is determined empirically based on the training set. We ran each training matrix with various CF_H and CF_V and

thresholds to identify the best performing configuration per matrix, from which we chose the most frequent CF_H as the final CF_H (c_l). We observed that using a coarsening factor of one (no coarsening) achieves the best performance for most matrices for double precision. For single precision, we observed that a coarsening factor of two achieves the best performance in most cases. Based on NVPROF performance metrics, the difference between the best coarsening factor for single vs. double precision results from the relative cost of the warp shuffle function.

4.4.2 Threshold for classifying rows as light or heavy

The threshold used to classify rows as light or heavy is computed using a decision tree. To train the decision tree, we chose clustering information and number of accumulations as the input features and the best threshold as the target. In order to model clustering, for each matrix, we calculated the distance between every two adjacent elements and assigned each distance to the corresponding bin. The bins are organized as powers of 4 (1, 1-3, 4-15, 16-63, . . ., 262,144-1,048,576).

The best threshold is found in two phases. In the first phase, we fix the $CF_H$and vary CF_V (1, 2, 4) and the threshold (1, 2, 3, . . ., 16). We evaluate the training set over these configurations to determine the best parameters for each matrix. The most frequent vertical coarsening factor among the best parameters is then identified. In the second phase, CF_V and CF_H are fixed on the base of the previous experiments and the corresponding best threshold per matrix is identified.

The decision tree (Weka 3.8 “J48 -C 0.25 -M 2” without any filtering) is then trained with the bin counts (distance between adjacent elements and number of accumulations as

input feature and the best threshold computed in phase 2 as the output feature. In order to determine the threshold of the test set, we used this trained decision tree.

4.4.3 Coarsening factor of the vertical scheme (CF_V)

Increasing the coarsening factor along K for the vertical scheme helps to hide memory latency and reduce shuffle overhead. However, increasing the coarsening factor increases the amount of shared memory required for storing the dense matrix elements, and this decreases the column panel size. When the size of the column panel is decreased, the number of rows which are classified as heavy rows will decrease, and this will adversely affect the performance. In order to determine the coarsening factor, for the vertical scheme, we estimate the execution time for each coarsening factor and then select the coarsening factor which minimizes the execution time. The execution time is estimated as:

$$\text{exec\_time} = C_{1,v} \times acc_{dense} + nnz_d \times C_{2,v} + \text{exec\_time\_sparse},$$

where $C_{,v}$ is the cost when CF_V is v, and acc_{dense} is the number of accumulations in the dense part. Since all of the elements are processed by the heavy scheme, the execution time of light scheme (exec_time_sparse) is zero. $C_{1,v}$ and $C_{2,v}$ are constants which depend on v.

In order to determine $C_{1,v}$ and $C_{2,v}$, we use two synthetic matrices i) 4K X 4K fully dense matrix and ii) 4K X 4K sparse matrix. Both matrices will be processed only by the vertical scheme. The sparse matrix is designed such that for every row in a column panel of size 256, there are threshold numbers of non-zero elements. The exact column ID is chosen at random.

Assuming that Num_Col_Panel = 4K / 256, the execution time for the two synthetic matrices for a coarsening factor of one, can be represented as

$$T_{1,1} = C_{1,1} \times 4K \times Num\_Col\_Panel + C_{2,1} \times 4K \times 4K$$

$$T_{2,1} = C_{1,1} \times 4K \times Num\_Col\_Panel +$$

$$C_{2,1} \times 4K \times Num\_Col\_Panel \times THRESHOLD$$

The execution time of $T_{1,1}$ and $T_{2,1}$ is determined by actually running these synthetic matrices using the vertical scheme (with the coarsening factor as one) and the above equations are solved to find $C_{1,1}$ and $C_{2,1}$. Similarly, the rest of the $C_{*,*}$ values are computed.

In order to find CF_V for a given matrix, the execution time is estimated by applying the above equations using the $C_{*,*}$ values. The coarsening factor is then selected as the one which yields the lowest predicted execution time.

4.4.4 Effectiveness of Model

Fig. 4.7 presents the performance of RS-SpMM for all SparseSuite matrices for single precision with K = 128. We present the performance of RS-SpMM without any modeling, with the best parameters (separately selected for each individual case), and with our modeling. As seen in the figure, our modeling matches or closely follows the performance of the best parameter case.

4.5 Experimental Evaluation

This section details the experimental evaluation of the RS-SpMM scheme. The experiments were performed on an NVIDIA Pascal P100 GPU. In all experiments, ECC was turned off and we used the NVCC compiler optimization flag -O3 with NVCC 8.0 and GCC

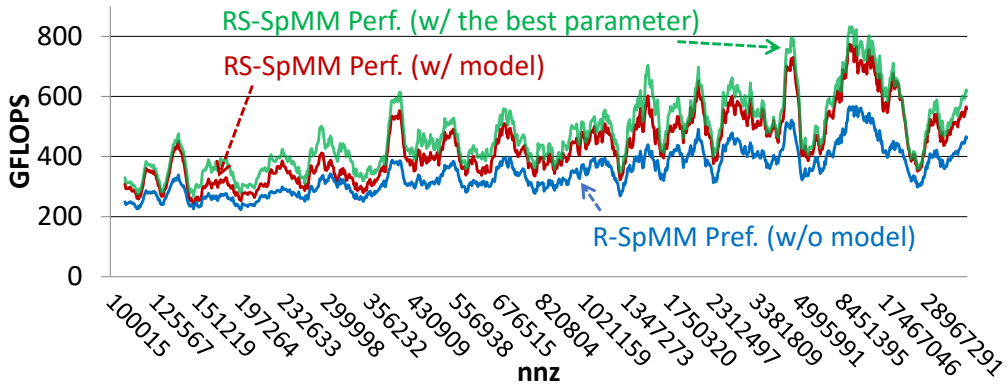


Figure 4.7: Modeling effect (WIDTH=128, single precision)

4.9.2. For all of the schemes, we only include the kernel execution time; preprocessing time and data transfer time from CPU to GPU are not included (we document preprocessing time separately). All tests were run five times and average numbers are reported.

4.5.1 Performance of RS-SPMM

Fig. 4.8 (a) shows the measured performance of RS-SpMM for single-precision computation, with four values of K (the width of the dense matrices O and D): 8, 32, 128, 512. Use-cases for SpMM vary depending on application. For applications in computational science, such as LOBPCG, widths in the tens, and below 100 are typical. For machine learning applications, the width K often corresponds to the number of latent features in models, and values of K around 100 are common, with interest in going higher to several hundreds or thousands. Hence, we evaluate RS-SpMM using four values in the range of 8-512. The performance of RS-SpMM improves as we increase dense matrix width from 8 to 32 and 128, tending to saturate at that point – the performance curves for K = 128 and K = 512 are nearly indistinguishable. For large matrices, performance is around 200

GFLOPs for K = 8, approximately doubling to around 400 GFLOPs for K = 32, and further increasing to almost 800 GFLOPs for several matrices for K = 128/512.

Figs. 4.8 (b-e) show relative performance improvement over NVIDIA's cuSPARSE library implementation of SpMM. cuSPARSE provides two variants: $O = SD$ (denoted cuSPARSE(NT)) and $O = SD^T$ (denoted cusPARSE(T)). As can be seen below, the performance of cuSPARSE for $O = SD^T$ (transposed product) is often much higher than the performance for $O = SD$ (non-transposed product). RS-SpMM also implements a transposed product, discussed later in Sec. 4.6. Since users have a choice in data layout, even if they only need the non-transposed product, because of the higher performance achieved by cuSPARSE SpMM for the transposed product, users have the option of changing the layout of their dense input matrix to the transposed form. Therefore, we present a comparison of non-transposed RS-SpMM with both the transposed and non-transposed cuSPARSE variants.

The speedup of RS-SPMM(NT) vs. cuSPARSE(NT and T) is shown for different values of K; since cuSPARSE(NT) tends to achieve much lower performance than cuSPARSE(T), higher speedup is achieved over it. RS-SpMM speedup is generally in the range between 1 and 2 over cuSPARSE(T), and around 4x over cuSPARSE(NT). The speedup over cuSPARSE tends to be higher for larger K.

Figs. 4.9 (a-e) present the performance of RS-SpMM for double-precision. Overall performance in GFLOPs for double-precision is slightly lower than that for single-precision, but well over 0.5x, compared to single-precision. RS-SpMM (NT) is consistently faster than cuSPARSE(T and NT). In contrast to single precision, the cuSPARSE(T) version is not consistently faster – quite often, the speedup over cuSPARSE(T) is higher than the speedup over cuSPARSE(NT). For sparse matrices exhibiting high variance in row lengths,

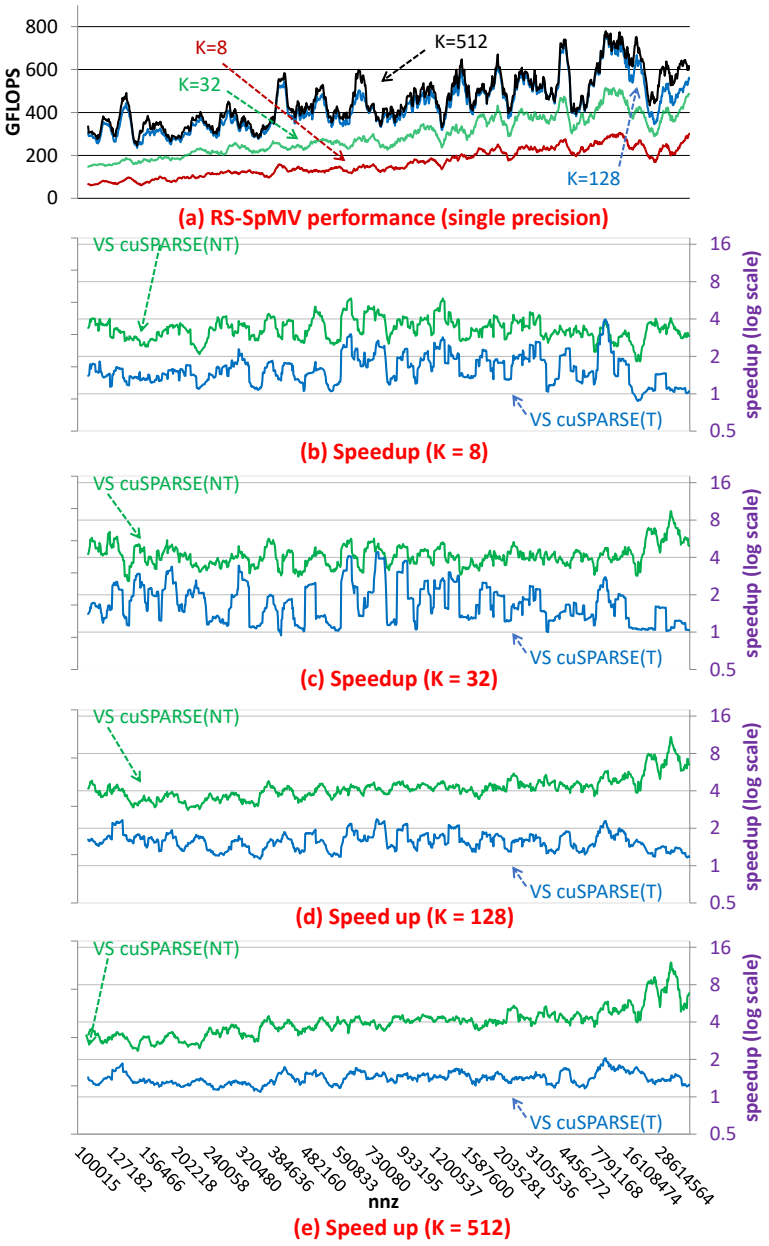


Figure 4.8: Performance comparison: RS-SpMM (NT) vs CuSPARSE (NT and T); single precision

the performance of cuSPARSE(NT) and cuSPARSE(T) degrades considerably because of load imbalances. On the other hand, the performance of RS-SpMM is less degraded, since

rows having a large nonzeros are normally classified as heavy rows and are processed by multiple thread blocks. Hence, there is a fluctuation of achieved speedup with different sparse matrices in Figs. 4.8 and 4.9. We note that in Figs. 4.8 and 4.9, only the kernel execution time is reported; the pre-processing overhead to create the needed representation for RS-SpMM is discussed later.

In Fig. 4.10, we present a performance profile comparing RS-SpMM with cuSPARSE SpMM (both (T) and (NT) variants). In each row, different charts are given for varying K (8, 32, 128, and 512). For each matrix, the best performing version among RS-SpMM and cuSPARSE SpMM variants is used as a normalizer to compute performance loss of the other instances. For each SpMM implementation, the cumulative curve shows the fraction of cases for which the slowdown with respect to the best performer is less than the X-axis value. Thus, a point (x,y) implies that a fraction y of all matrices achieved a slowdown less than x relative to the fastest implementation. Better performing variants have curves that rise rapidly to hit y = 1.0, while relatively poor performance results in a shallow curve that may not get to y = 1.0 even for the largest slowdown value in the range of the plot. The top (red) curve shows that RS-SpMM quite significantly outperforms the cuSPARSE variants. We note that for $K = 8$ and $K = 32$, we also compared RS-SpMM with CUSP [32] SpMM. The NVIDIA CUSP library is an open-source library for linear algebra and graph computations on GPUs. CUSP implements SpMM, but we find its results are only correct for dense matrix widths of 2,4,8,16, and 32 and incorrect for other widths; even for the few correct cases, CUSP-SpMM performance is consistently lower than cuSPARSE(T) performance, as seen in Figs. 4.10 (a,b,e,f).

The first two rows of charts document performance over all SuiteSparse matrices with more than 100K non-zeros for single and double precisions. The third and fourth rows

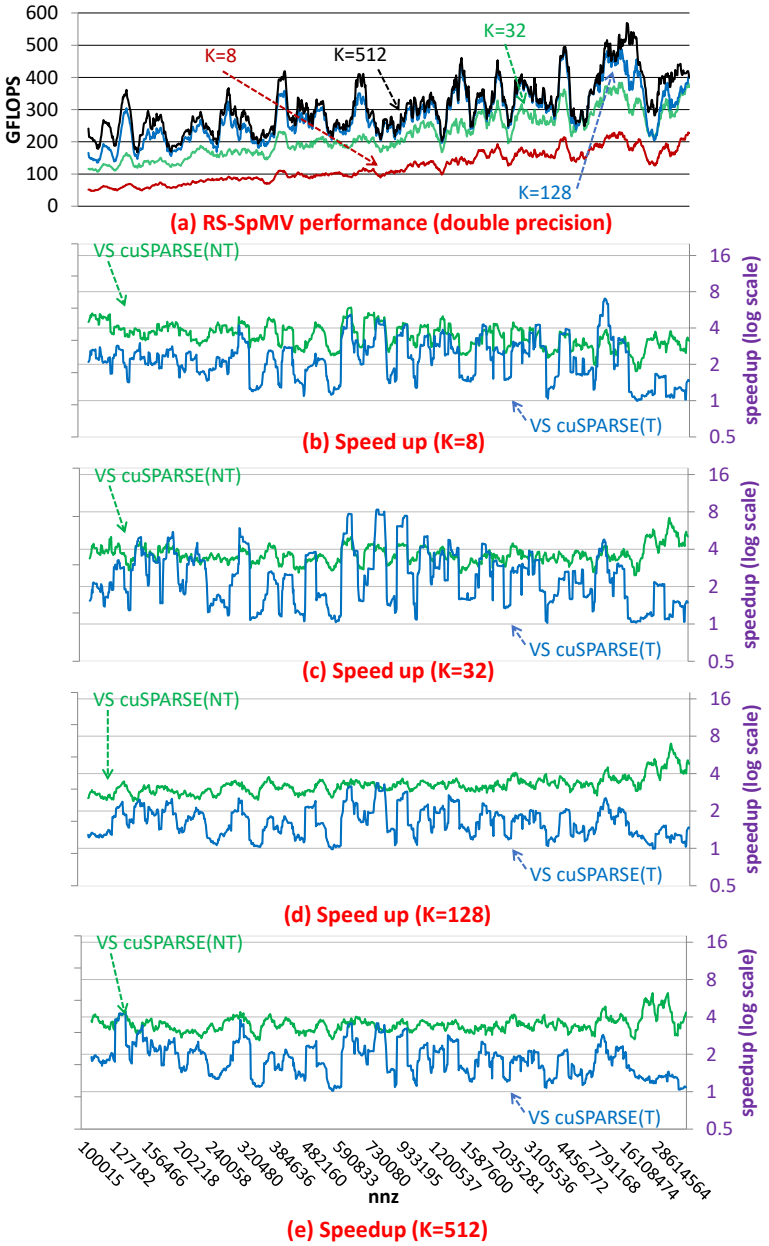


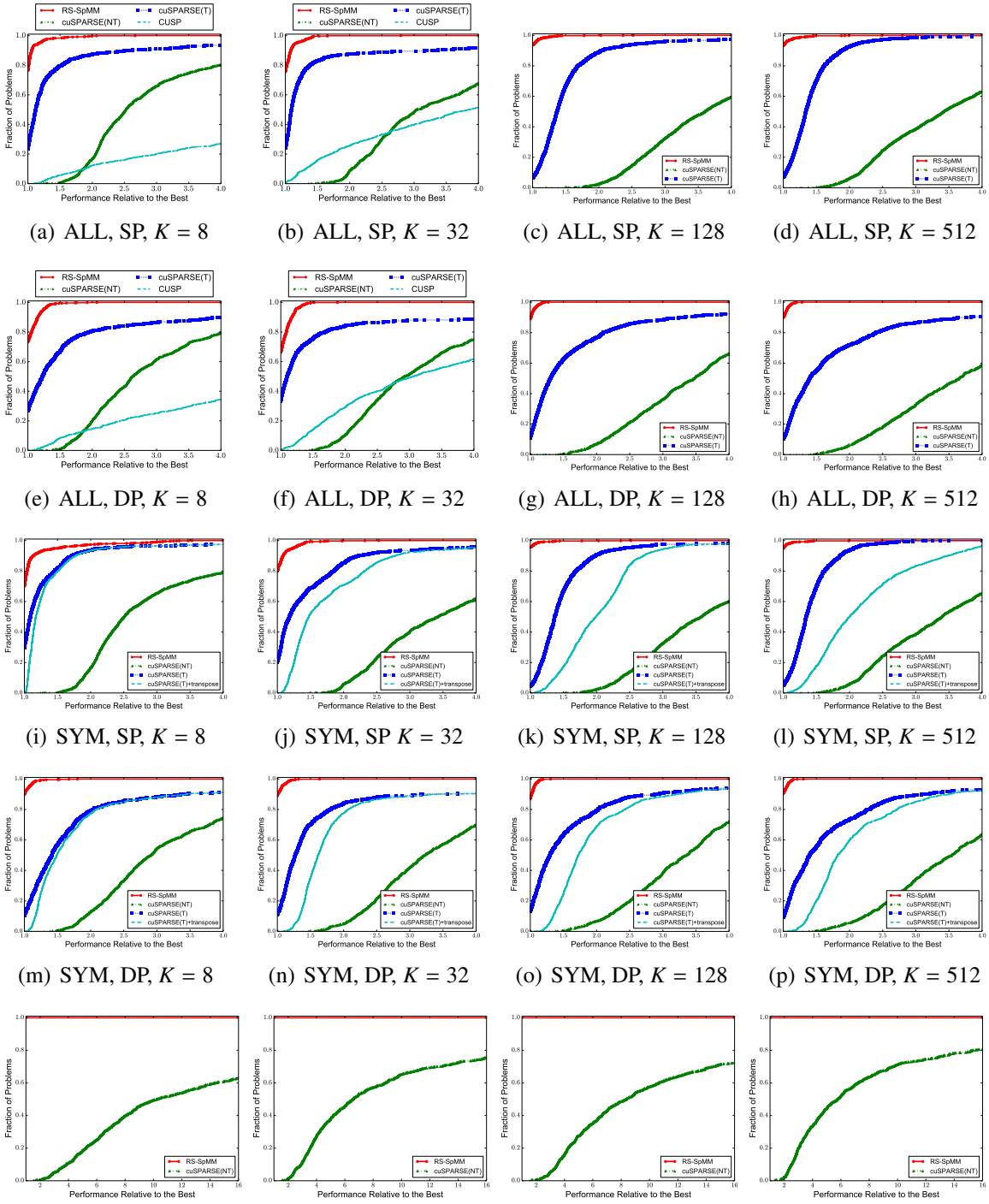
Figure 4.9: Performance comparison: RS-SpMM (NT) vs CuSPARSE (NT and T); double precision

show performance for an iterative execution scenario that is common with applications like block Krylov solvers. Here, the output dense matrix O from a previous iteration is modified

by scaling or other simple point-wise operations and then becomes the input matrix D for the following iteration. With the cuSPARSE(T) form, an explicit transpose will be required. The time for cuBLAS [97] dense matrix transpose is added to cuSPARSE(T) for this scenario. Experimental results are only presented for square matrices from SuiteSparse, since this is only applicable to square matrices. Finally, the last row shows the profile for the $S^T D$ computation and the case where both SD and $S^T D$ are required. In this case, performance of RS-SpMM is considerably higher than that of cuSPARSE.

The MAGMA library [7] has high-performance implementations for dense and sparse linear algebra functions for GPUs. We attempted a performance comparison of RS-SpMM with MAGMA, in addition to cuSPARSE. However, we were unable to successfully process all of the matrices from SuiteSparse with MAGMA; several matrices resulted in MAGMA error messages. However, we were able to run a significant subset of matrices successfully with MAGMA's SpMM. In Fig. 4.11, we present a profile comparison on just the successful subset with MAGMA, for single-precision and K = 128. We can observe in this profile that cuSPARSE(T) is faster than MAGMA and RS-SpMM is faster than cuSPARSE(T).

bhSPARSE [78] and Merge-based CSR [90] provide very high-performance GPU implementations of SpMV, but do not provide an SpMM implementation. SpMM can be implemented as a "loop over SpMV", with an outer loop over the width of the dense matrix. Fig. 4.12 (a-d) show measured performance using such a loop-over-SpMV approach, with bhSPARSE, Merge-based CSR and cuSPARSE-SpMV for single-precision computation, with four values of K: 8, 32, 128, and 512. For K values beyond 8, RS-SpMM is



(a) ALL, SP, $K = 8$ (b) ALL, SP, $K = 32$ (c) ALL, SP, $K = 128$ (d) ALL, SP, $K = 512$

(e) ALL, DP, $K = 8$ (f) ALL, DP, $K = 32$ (g) ALL, DP, $K = 128$ (h) ALL, DP, $K = 512$

(i) SYM, SP, $K = 8$ (j) SYM, SP $K = 32$ (k) SYM, SP, $K = 128$ (l) SYM, SP, $K = 512$

(m) SYM, DP, $K = 8$ (n) SYM, DP, $K = 32$ (o) SYM, DP, $K = 128$ (p) SYM, DP, $K = 512$

(q) $O = S^T \times D$, SP (r) $O = S^T D$ & $O = SD$, SP (s) $O = S^T D$, DP (t) $O = S^T D$ & $O = SD$, DP

Figure 4.10: Performance Profiles. (a)-(h) all matrices (ALL) for single precision (SP) and double precision (DP) with varying K, (i)-(p) symmetric (SYM) matrices for SP and DP with varying K, (q)-(t) $O = S^T D$ & $O = SD$ for SP and DP

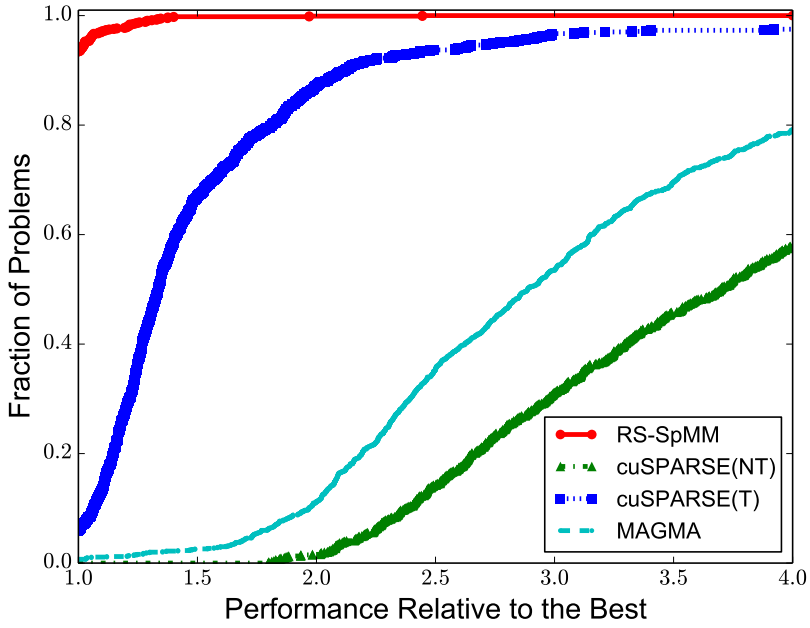
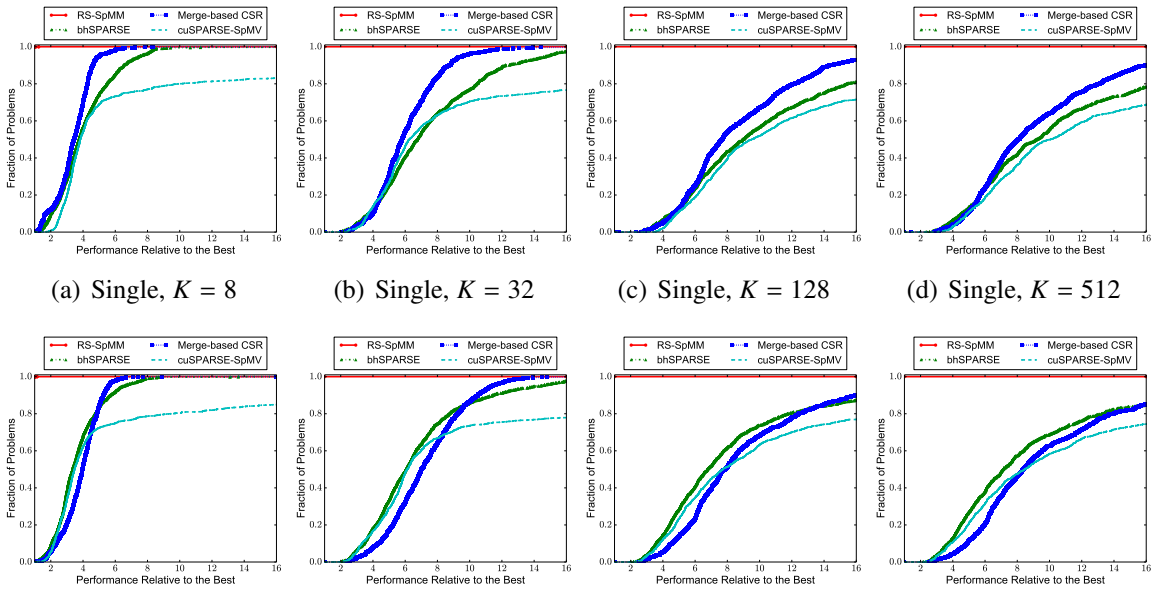


Figure 4.11: Performance of RS-SpMM compared with MAGMA and cuSPARSE; K=128, single precision

significantly faster than loop-over-SpMV (bhSPARSE, Merge-based CSR and CuSPARSE-SpMV) because of significantly higher data reuse achieved by SpMM primitives (the maximum value of X-axis in Fig. 4.12 is 16). Similar trends can be seen for double precision, as shown in Figs. 4.12 (e-h).

4.5.2 Pre-processing overhead

Constructing the data structures required for the RS-SPMM scheme incurs an additional overhead. Fig. 4.13 shows the preprocessing overhead normalized to one iteration of RS-SpMM. Note that typical applications involving SpMM can execute a large number of iterations such as [12, 104]. For example, with sparse convolutional neural networks in inference mode, even though the input dense matrix changes (holding the new data to be



(a) Single, $K = 8$ (b) Single, $K = 32$ (c) Single, $K = 128$ (d) Single, $K = 512$

(e) Double, $K = 8$ (f) Double, $K = 32$ (g) Double, $K = 128$ (h) Double, $K = 512$

Figure 4.12: Performance profiles: RS-SpMM and Loop-over-SpMV; single and double; K=8,32,128,512

processed), the structure of the sparse coefficient matrix remains unchanged. Hence, the preprocessing overhead is relatively insignificant.

Pre-processing is carried out on the GPU – converting from standard CSR structure (column indices within each row assumed sorted) to the structure which splits out heavy row-segments (in DCSR) and the remainder (standard CSR format). For threshold T, heavy row-segments are extracted by scanning over rows and checking if col_ptr[i] and col_ptr[i+T] belong to the same partition in the same row; if so, the elements between index i to i+T get included in a heavy row. Fig. 4.13 shows our pre-processing cost normalized to one SpMM iteration with K = 128.

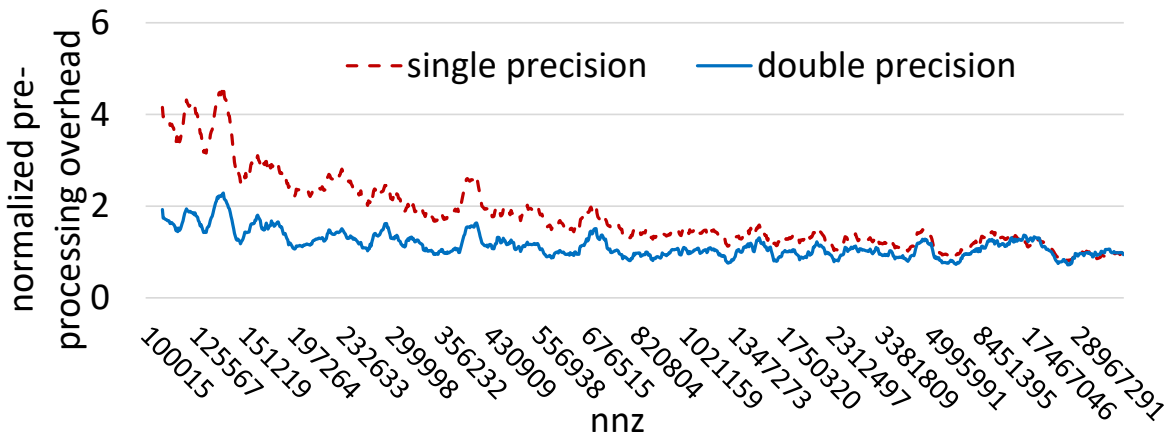


Figure 4.13: Preprocessing overhead

4.6 SpMM for $O = S^T D$

Some applications require both an original and a transposed sparse matrix to be multiplied by the dense matrix ($O = S \times D$ and $O = S^T \times D$) [4, 7]. Explicitly transposing the sparse matrix just for SpMM can be expensive. The RS-SpMM scheme has been adapted to perform $O = S^T \times D$ without changing the data structure.

Pseudocodes for heavy row segments and light row segments are shown in Listings 4.4 and 4.5, respectively. The heavy rows are represented in DCSR format. In order to perform $O = S^T \times D$, we use shared memory corresponding to the columns of the column panel to store the output. The shared memory is initialized to zero (lines 5-7 in Listing 4.4). Each row in a column panel is processed by a warp. The input dense matrix and sparse matrix elements are read from the global memory, and the partial products are computed (lines 17-22). Each such partial product is accumulated in shared memory using atomic operations (line 23). At the end of each column panel, accumulated results in shared memory are updated to global memory using atomic operations (line 28). The light rows

are represented in CSR format. When $O = S^T \times D$, we iterate over the CSR representation. Each thread loads the corresponding element of the input matrix to a thread local register (line 6 in Listing 4.5) and computes partial products (lines 9-14). Each partial product is updated to global memory using atomic operations (line 15).

Listing 4.4: SpMMT pseudocode (heavy row segments)

```
1. row_offset = tb_idx * IN_TILE_ROW_SIZE;
2. slice_offset = tb_idy * IN_TILE_SLICE_SIZE;
3. warp_id = tid/WARP_SIZE;
4. lane_id = tid%WARP_SIZE;
5. for i=warp_id to IN_TILE_ROW_SIZE step tb.size()/WARP_SIZE do
6.   sm_output_value[i][lane_id] = 0;
7. end
8. __syncthreads;
9. for i=seg_start_num[tb_idx] to seg_start_num[tb_idx+1]-1 step
         tb.size()/WARP_SIZE do
10.  val = 0;
11.  start = start_seg_position[i];
12.  end = start_seg_position[i+1];
13.  column_idx = seg_row_position[i];
14.  column_value = input_value[column_idx][lane_id];
15.  for j=start to end-1 do
16.    mod = (j - start)%WARP_SIZE
17.    if mod == 0 then
18.      index_buf = seg_index[j + lane_id];
19.      value_buf = seg_value[j + lane_id];
20.    end
21.    row_offset = __shfl(index_buf, mod);
22.    val = column_value * __shfl(value_buf, mod);
23.    atomicAdd(&sm_output_value[row_offset][slice_offset+lane_id], val);
24.  end
25.  __syncthreads;
26.  row_idx = seg_row_position[i];
27.  for i=warp_id to IN_TILE_ROW_SIZE step tb.size()/WARP_SIZE do
28.    atomicAdd(&output_value[row_offset+i][lane_id],
       sm_out_value[i][lane_id]);
29. end
```

Listing 4.5: SpMMT pseudocode (light row segments)

```
1.  column_offset = (tb_idx*tb.size() + tid) / WARP_SIZE;
2.  slice_offset = tb_idy * IN_TILE_COL_SIZE;
3.  lane_id = tid%WARP_SIZE;
4.  start = csr_row_pointer[column_offset];
5.  end = csr_row_pointer[column_offset+1];
6.  column_value = input_value[column_offset][slice_offset+lane_id];
7.  for i=start to end-1 do
8.    mod = (i - start)%WARP_SIZE
9.    if mod == 0 then
10.     index_buf = csr_column_idx[i + lane_id];
11.     value_buf = csr_column_val[i + lane_id];
12.   end
13.   row_offset = __shfl(index_buf, mod);
14.      val = column_value * __shfl(value_buf, mod);
    //directly accumulate results in global memory
15.   atomicAdd(&dest_value[row_offset][slice_offset+lane_id], val);
16. end
```

Figs. 4.14 and 4.15 compare RS-SpMM with CuSPARSE for $O = S^T D$, for single and double precision, respectively. Two scenarios are evaluated: i) $O = S^T D$ only, and ii) both $O1 = S^T D$ and $O2 = S D$ are required. The performance of RS-SPMM is significantly higher than that of cuSPARSE for both scenarios, for both precisions.

When non-zeros are clustered, the performance of cuSPARSE can be higher than 200 GFLOPS. However, when non-zeros are scattered (which results in low cache utilization), performance drops to 10 GFLOPS. On the other hand, the performance of RS-SpMM is consistently higher than 80 GFLOPS when concurrency is not very low. Hence, the performance of cuSPARSE is over 16x times slower than ours for more than 20% of matrices, as shown in Figs. 4.10 (q) and (s).

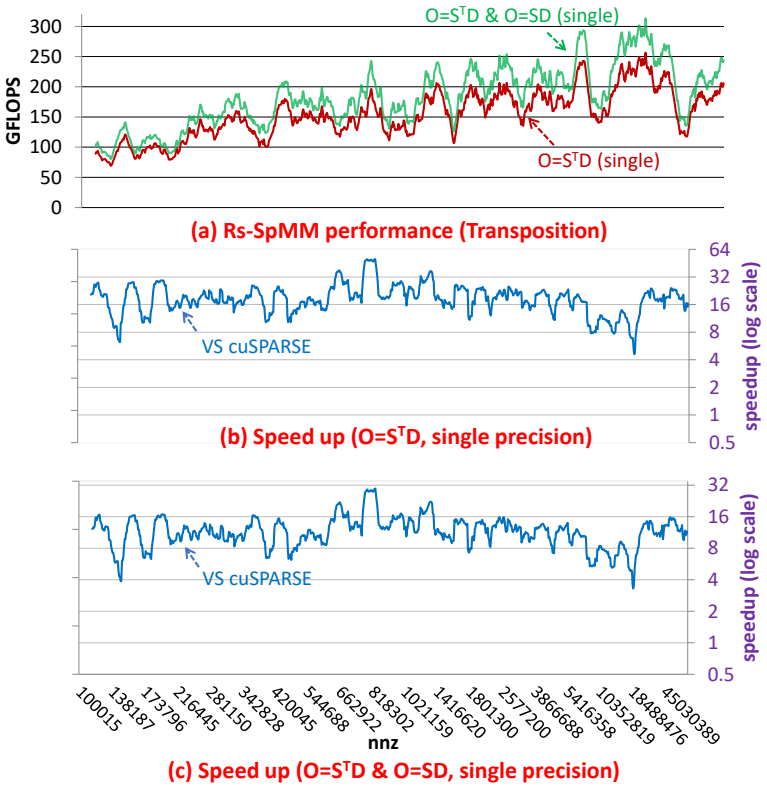


Figure 4.14: $O = S^T D$ performance ($K = 128$, single precision)

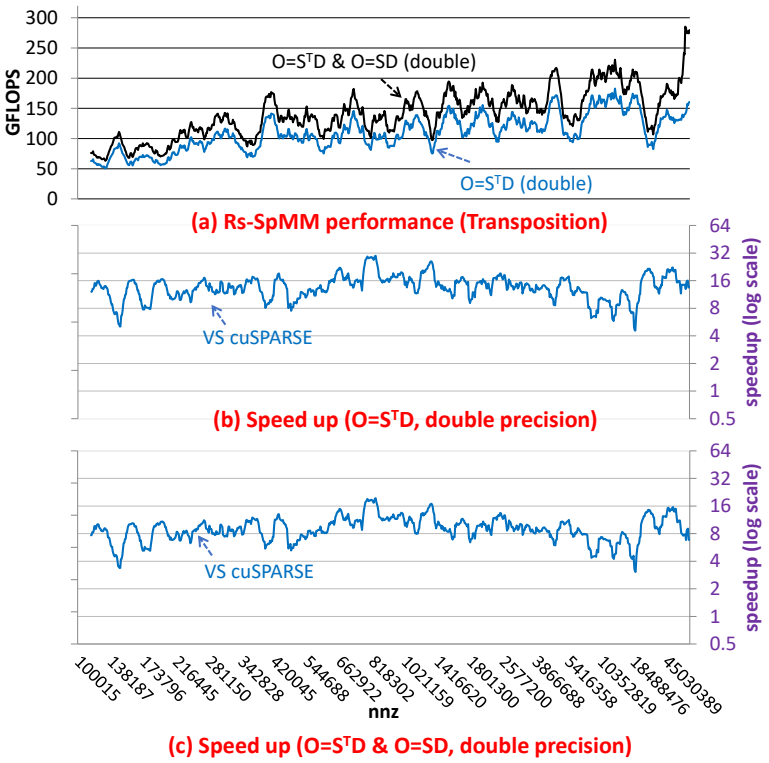


Figure 4.15: $O = S^T D$ performance ($K = 128$, double precision)

4.7 Discussion

Our extensive evaluation of the Florida dataset shows that in most cases, we significantly outperform cuSPARSE. However, the performance with RS-SpMM is still far below the upper bound discussed in Sec. 4.1. In this section, we describe the SpMM bottlenecks and possible improvements. NVIDIA's documentation [140] states that the P100 GPU used in this chapter has 56 Streaming Multiprocessors (SMs) and uses a GPU Clock of 1328 MHz. For each SM, there are 16 LD/ST units. That is, at most 16 LD/ST transactions (64 bytes) can be processed for each clock on one SM. Therefore, bandwidth between LD/ST units for one SM cannot exceed 64 bytes $\times$ 1328 MHz = 84.99 Gbytes/sec.

Since there are 56 SMs, the shared memory bandwidth is at most $84.99 \times 56 = 4759.55$ Gbytes/sec = 1189.89 Gwords/sec. For the vertical streaming scheme, one shared memory load must be associated with two floating point operations, as can be seen in line 19 in Listing 4.1. Therefore, the upper bound with RS-SpMM is $2 \times 1189.89 = 2379.78$ GFLOPs/sec. Across the full Florida dataset, we could achieve up to 1650 GFLOPs (for TSOPF_RS_b2383 dataset), which is 69.3% of this upper bound. For the horizontal scheme, to process the sparse non-clustered non-zeros, one LD/ST operation is associated with two floating point operations, as seen in line 13 in Listing 4.2.

Consider a sparse matrix with M rows, N columns and assume that the width of the input and output dense matrices are K. Assume that the horizontal scheme is used to process SpMM. The minimum volume of data transfer passed through the LD/ST units can be computed as follows. In this scheme, the output matrix **O** gets full reuse. Hence, the minimum number of transactions (in words) for **O** is $M \times K$. Each **D** element is multiplied by all of the elements in the corresponding column of **S**. Assuming that each **D** element gets an average reuse of R from registers, the minimum number of transactions (in words) for **D** is $\frac{nnz \times K}{R}$. Each **S** element has a K-way reuse. Assuming that the coarsening factor is C, and each **S** access gets full reuse across a full warp (of 32 threads), the minimum number of transactions for **S** is $\frac{nnz \times K}{C \times 32}$. Thus, the minimum number of LD/ST transactions for this scheme is $\frac{nnz \times K}{R} + M \times K + \frac{nnz \times K}{C \times 32}$ words. When $M \times R < nnz$ and $R < C \times 32$, the dominant term would be $\frac{nnz \times K}{R}$.

This analysis suggests the following:

i) If a sparse matrix is also loaded from shared memory/cache/global memory, performance can be at most 2379.776/2 = 1189.89 GFLOPs/sec since two load operations are involved in two floating point operations. Thus, to achieve high-performance, warp shuffling (which

does not go through LD/ST units) may be more beneficial for accessing the sparse matrix.

ii) When warp shuffling is used, thread coarsening along the K dimension is not very helpful, since it does not reduce the number of accesses of LD/ST units that much.

iii) Register reuse is a key to achieving better performance, and thread coarsening along the M dimension can be beneficial, since it can reduce LD/ST transactions.

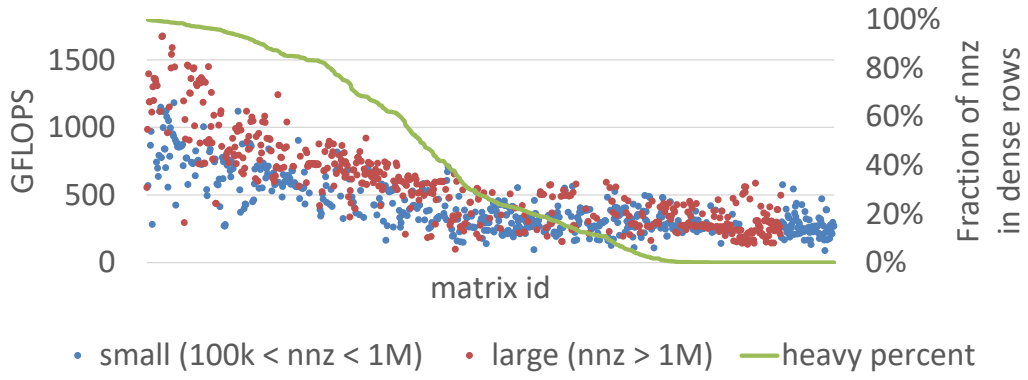


Figure 4.16: RS-SpMM performance: impact of row-segment density

Fig. 4.16 presents the RS-SpMM performance data for the matrices ordered along the X-axis by the fraction of the non-zeros in the heavy row-segments. Matrices on the left side of the graph have higher non-zeros in the heavy row-segments and the matrices on the right have lower non-zeros in heavy row-segments. A clear trend shows that matrices with a larger fraction in the heavy part tend to perform better. This suggests that column reordering schemes that increase row-segment density might be a promising direction to take to further improve performance.

4.8 Conclusion

In this chapter, we develop an efficient SpMM algorithm for GPUs by exploiting the clustering in sparse matrices. Our approach tries to get good reuse of the elements from the input vector, the output vector, and the sparse matrix, simultaneously. Our extensive experimental evaluation section demonstrates the superior performance of our approach when compared to other state-of-the-art SpMM frameworks.

CHAPTER 5

Adaptive Sparse Tiling for Sparse Matrix Multiplication

5.1 Introduction

Tiling is a key technique for effective exploitation of data reuse and is used in all high-performance implementations of dense linear algebra computations, convolutional neural networks, stencil computations, etc. While tiling for such regular computations is well understood and is heavily utilized in high-performance implementations on multicore/many-core CPUs and GPUs, the effective use of tiling for sparse matrix multiplication poses challenges.

In order to motivate the need for data locality optimization via tiling for sparse-matrix dense-matrix multiplication (SpMM),[5] we present some experimental data on an Intel Xeon Phi processor (KNL, Knights Landing) using the *mkl_scsrmm* routine for SpMM in the Intel MKL library. A number of banded matrices ($\{S \mid S[x][y] \neq 0 \iff (0 \leq x < \#rows, max(0, y - b) \leq y < min(\#cols, y + b))\}$, where b is the half band-size) of size $16K \times 16K$ with different band-sizes were used as the sparse matrix argument for the MKL SpMM routine. Fig. 5.1 presents the performance trend (GFLOPs) as the band-size is varied. Performance improves up to a band-size of 1025 and drops beyond that.

[5]We use SpMM to denote the product of a sparse matrix with a dense matrix, to be distinguished from sparse matrix-matrix multiplication (SpGEMM), where two sparse matrices are multiplied.

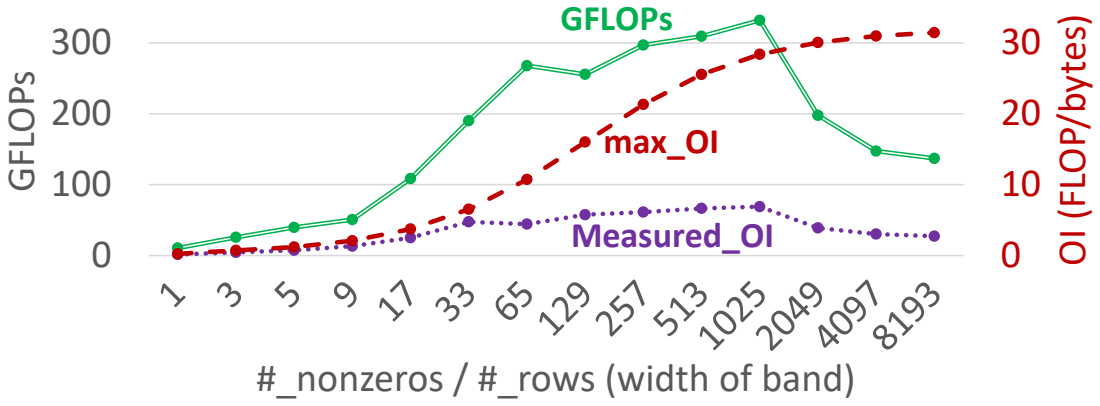


Figure 5.1: OI and GFLOPs with respect to matrices having different bands

The figure also plots the measured operational intensity (OI), the ratio of floating-point operations to the number of bytes of data moved to/from main memory. We can observe that measured OI increases up to a band-size of 1025 and then drops for larger band sizes. Thus, the performance drop is correlated with increased data movement per operation. As we explain in greater detail in Sec. 5.3, this is in contrast to the potential maximum OI, which increases with band-size.

With dense matrix-matrix multiplication, uniform tiling is the norm, where all tiles (except boundary tiles) have the same number of operations and the same data footprint. However, with SpMM, the number of non-zero elements will vary significantly across different uniform-sized tiles due to the very non-uniform distribution of non-zero elements across the 2D index space of a sparse matrix. As we explain in detail later in the paper, whether tiled execution can achieve higher performance than untiled execution for a 2D region of a sparse matrix depends on the sparsity structure in that region. In this chapter, we develop an Adaptive Sparse Tiling (ASpT) approach to tiling two variants of sparse matrix multiplication: Sparse-dense Matrix Multiplication (SpMM) and Sampled Dense

Dense Matrix Multiplication (SDDMM). A key idea is that the average number of non-zeros per "active" row/column segment (i.e., at least one non-zero) within a 2D block plays a significant role in determining whether tiled or untiled execution is preferable for a 2D block. The sparse matrix is partitioned into panels of rows, with the active columns within each row panel being either grouped into 2D tiles for tiled execution, or relegated to untiled execution because its active column density is inadequate. In contrast to prior efforts that have used customized data representations to improve the performance of sparse matrix operations, we achieve the hybrid tiled/untiled execution by using the standard (unordered) Compressed Sparse Row (CSR) representation of sparse matrices: the non-zero elements in column segments that are to be processed in untiled mode are reordered to be contiguously located at the end in the unordered CSR format.

We demonstrate the effectiveness of the proposed model-driven approach to hybrid-tiled execution of sparse matrix computations by developing implementations for SpMM and SDDMM kernels on GPUs and multicore/manycore processors (Intel Xeon, and Intel Xeon Phi KNL). On all platforms, the significant performance improvement is achieved over available state-of-the-art alternatives – Intel's MKL and NVIDIA's cuSPARSE libraries for SpMM, and the MIT TACO compiler and BIDMach for SDDMM.

5.2 Background and Related Works

5.2.1 Standard Sparse Matrix Representation

The CSR representation is one of the most widely used data structures for representing sparse matrices [119, 151]. As shown in Figs. 5.2 (b,c), the CSR structure is composed of three arrays: row_ptr, col_idx, and values. The value of row_ptr[i] contains the index of the first element of row i. values[] holds the actual numerical values of the non-zero

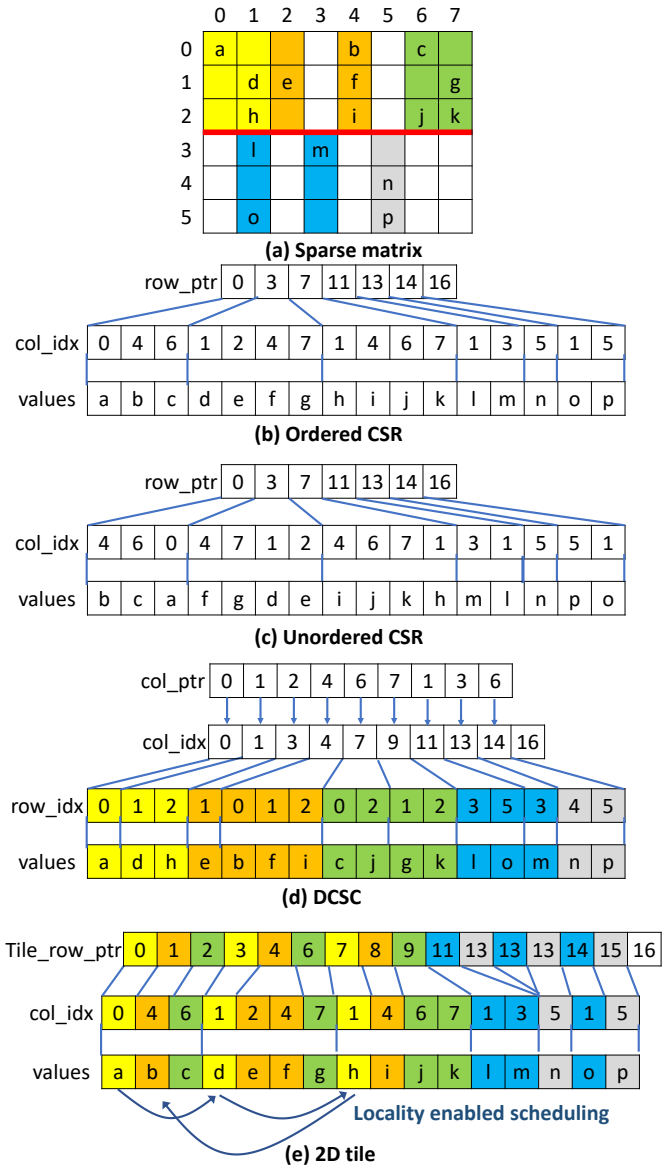


Figure 5.2: Various data representations for a sparse matrix

elements, and col_idx[] holds the corresponding column indices. As shown in Figs. 5.2 (b,c), non-zeros within each row are placed contiguously in col_idx[] and values[]. CSR has two variants, ordered CSR and unordered CSR [62]. In ordered CSR, the column indices within a row are sorted, whereas in unordered CSR the column indices may not be

kept in sorted order. Fig. 5.2 (c) illustrates unordered CSR and Fig. 5.2 (b) represents the corresponding ordered CSR version.

Double-Compressed Sparse Column (or Row) DCSC (DSCR) [20] is an alternate format, used for ultra-sparse matrices where many rows (columns) may be completely empty. Fig. 5.2 (d) shows a DCSC representation, where a sparse matrix is partitioned into row panels. Four arrays are maintained: col_ptr[], col_idx[], row_idx[], and values[]. col_idx[] contains the column index and col_ptr[] points to the first element of the corresponding column segment. row_idx[] and values[] store the row indices and the actual non-zero values, respectively. Fig. 5.2 (d) shows the DCSC representation.

Sparse matrices can also be represented using 2D tiles, as shown in Fig. 5.2 (e). Like DCSC, the sparse matrix is partitioned into a set of row panels. Each row panel is further divided into 2D tiles. tile_row_ptr[] is used to track the start point of each row within a 2D tile. The col_idx[] and values[] hold the column indices and actual non-zero values, respectively.

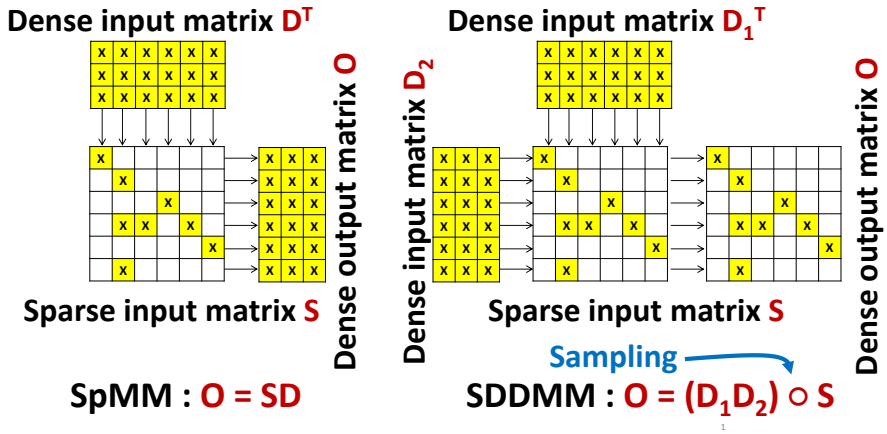


Figure 5.3: Conceptual view of SpMM and SDDMM

Algorithm 9: Sequential SpMM (Sparse Matrix Matrix Multiplication)

```
   input :  CSR S[M][N], float D[N][K]
   output:  float O[M][K]
1  for i = 0 to S.num_rows-1 do
2      for j = S.row_ptr[i] to S.row_ptr[i+1]-1 do
3          for k = 0 to K-1 do
4              O[i][k] += S.values[j] * D[S.col_idx[j]][k];
5          end
6      end
7  end
```

Algorithm 10: Sequential SDDMM (Sampled Dense Dense Matrix Multiplication)

```
   input :  CSR S[M][N], float D1[N][K], float D2[M][K]
   output:  CSR O[M][N]
1  for i = 0 to S.num_rows-1 do
2      for j = S.row_ptr[i] to S.row_ptr[i+1]-1 do
3          for k = 0 to K-1 do
4              O.values[j] += D2[i][K] * D1[S.col_idx[j]][k];
5          end
6          O.values[j] *= S.values[j];
7      end
8  end
```

5.2.2 SpMM and SDDMM

In SpMM, a sparse matrix S is multiplied by a dense matrix D to form a dense output matrix O. Fig. 5.3 (left) shows a conceptual view of SpMM. Alg. 9 shows sequential SpMM using a CSR representation. SpMM is widely used in many applications such as Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) for finding the eigenvalues of a matrix [8], Convolutional Neural Networks (CNNs) [44], and graph centrality calculations [113]. SpMM is also one of the core GraphBLAS primitives [21].

In SDDMM two dense matrices, *D1* and *D2*, are multiplied and the resulting matrix is then scaled by an input sparse matrix *S* (Hadamard product). Fig. 5.3 (right) shows a conceptual view of SDDMM. Alg. 10 shows sequential SDDMM using a CSR representation. The SDDMM primitive can be used for efficient implementation of many applications such as Gamma Poisson (GaP) [125], Sparse Factor Analysis (SFA) [25], and Alternating Least Squares (ALS) [68].

Both SpMM and SDDMM traverse the rows of S (the outer loop). In SpMM, each element in the i-th row (with column index k) of the input sparse matrix *S* is used to scale the k-th row of *D1[k][:]* and the partial products are accumulated to form the i-th row of the output matrix *O[i][:]*. In SDDMM, the dot product of the j-th row of *D1* (i.e., D1[j][:]) and i-th row of *D2* (i.e., D2[i][:]) is computed at non-zero position (i,j) of the sparse matrix S and then scaled with S(i,j) to form O(i,j).

Several recent research efforts have been directed toward the development of efficient Sparse Matrix-Vector Multiplication (SpMV) [39, 77, 90, 79, 119, 92, 106, 117, 121, 120, 123, 128, 152, 145]. However, very few efforts have focused on SpMM and SDDMM.

In typical applications where SpMM or SDDMM are used, these operations are repeated many times using the same sparse matrix (the values may change, but the sparsity structure does not). For instance, SpMM is useful in the Generalized Minimum Residual (GMRES) [12] method, where several hundred iterations are required. This usage pattern allows a one-time light pre-processing of the sparse matrix to enhance performance, and the cost of this pre-processing is amortized across the iterations. As explained later, our approach requires a one-time reordering of sparse matrix elements to enhance data locality and reuse.

5.2.3 Related Works

Efforts to optimize SpMM and SDDMM may be grouped into two categories: using standard representation (CSR) or non-standard customized sparse matrix representations.

Intel's MKL [135] is a widely used library for multi/many-cores. MKL includes optimized kernels for many sparse matrix computations, including SpMM, SpMV and sparse-matrix sparse-matrix multiplication (SpGEMM).

TACO [66] is a recently developed library using compiler techniques to generate kernels for sparse tensor algebra operations, including SpMM and SDDMM. Generated kernels are already optimized, and OpenMP parallel pragma is used for parallelization.

cuSPARSE [2] provided by NVIDIA also supports SpMM. It offers two different modes depending on access patterns of dense matrices (i.e., row- or column-major order).

BIDMach [26] is a library for large-scale machine learning, and includes several efficient kernels for machine learning algorithms such as Non-negative Matrix Factorization (NMF) and Support Vector Machines (SVM). It includes an implementation of SDDMM.

Recently, Yang et al. [151] applied row-splitting [14] and merge-based [90] algorithms to SpMM to efficiently hide global memory latency. Based on the pattern of the sparse matrix, one of two algorithms is applied.

Several efforts have sought to improve SpMM performance by defining new representations for sparse matrices. Variants of ELLPACK have been used to improve performance, e.g., ELLPACK-R in FastSpMM [101], and SELL-P in MAGMA [8].

OSKI [132] uses register blocking to enhance data reuse in registers/L1-cache which improves the SpMV performance. When the non-zero elements are highly clustered, register blocking can reduce the data footprint of the sparse matrix.

Compressed Sparse Blocks (CSB) [5] is another sparse matrix storage format which exploits register blocking. The sparse matrix is partitioned and stored as small rectangular blocks. In CSB, register blocking also reduces the overhead of transposed SpMM ($O = A^T B$). SpMM implementation with CSB data representation has been demonstrated to achieve high-performance when both SpMM and transposed SpMM ($O = A^T B$) are simultaneously required [5]. Register blocking also plays a pivotal role in many sparse matrix formats for both CPUs [22, 23, 142] and GPUs [150].

We recently developed an SpMM implementation for GPUs based on a hybrid sparse matrix format called RS-SpMM that enabled significant performance improvement over alternative SpMM implementations [48]. However, a disadvantage of the approach is that a customized non-standard data structure is used to represent the sparse matrix, making it incompatible with existing code bases and libraries. Applications often use many library functions, which are based on the CSR representation. Iterative applications that make repeated use of SpMM interleaved with other sparse matrix operations may incur a high overhead in a repeated conversion from standard CSR to the non-standard representation [151, 119].

Reordering of sparse matrices has been widely explored in many other contexts. Yzelman et al. [153] show that reordering by recursive hypergraph-based sparse matrix partition can enhance cache locality, and thus performance. Oliker et al. [100] demonstrate that the performance of conjugate gradient (CG) and incomplete factorization (ILU) preconditioning can be improved by several reordering techniques such as METIS graph partitioning [61] that enhance locality.

While the above reordering strategies improve performance, they suffer from significant pre-processing overhead. GOrder [138] and ReCALL [72] try to reduce the preprocessing

overhead using greedy strategy for graph algorithms. The key idea is to number/index the vertices, such that vertices with many common neighbors are assigned indices that are close to each other to improve data locality.

In this chapter, we seek to improve the SpMM/SDDMM performance without the use of any non-standard sparse matrix representations. A significant benefit of using an unordered CSR format rather than an arbitrary new data format is the compatibility with existing code and libraries. Another benefit is the reduced storage space requirement. With non-standard representations, it may be necessary to keep an additional copy of the sparse-matrix in a standard format such as CSR for use with other library functions. This space overhead can be avoided by using a standard representation.

There are two significant differences between the reordering technique used in ASpT and existing works. First, the pre-processing overhead is significantly lower (milliseconds) than reordering schemes like GOrder [138] and ReCALL [72], which take tens of seconds for SuiteSparse datasets with large numbers of non-zeros [35]. Second, the original indexing of vertices is preserved, which eliminates overheads for re-indexing.

5.3 Overview of ASpT

This subsection provides an overview of Adaptive Sparse-matrix Tiling (ASpT), a strategy for tiled execution of sparse matrix multiplication using an unordered Compressed Sparse Row (CSR) representation.

We first elaborate on the observed performance trend shown earlier in Fig. 5.1. Let us consider the execution of the CSR SpMM algorithm (Alg. 9). The outer (i) loop traverses the rows of the sparse matrix S; the middle (j) loop accesses the non-zero elements in row_i of S, and the inner (k) loop updates row_i of the output array O by scaling appropriate rows

of the input dense matrix D by the values of the non-zero elements of S in row_i. The total number of non-zero elements for an $N \times N$ banded matrix with band-size B is approximately NB, and so the total number of floating point operations for the SpMM product with a dense matrix of size $N \times K$ is 2NBK. The total data footprint for the computation (sum of sizes of all arrays) for single-precision (four bytes per word for the two dense matrices; eight bytes per non-zero in the sparse matrix, for column index and value; four bytes per row pointer in CSR) is 4NK + 4NK + 8NB + 4N, or approximately 8N(K+B). The maximum possible operational intensity (OI), corresponding to complete reuse of data elements in cache/registers, is thus $\frac{2NBK}{8N(K+B)} = \frac{1}{\frac{4}{B}+\frac{4}{K}}$. Therefore, as the band-size increases, max_OI also increases. The actually achieved (measured) OI first increases as B increases, but then decreases due to limited L2 cache capacity. The Intel Xeon Phi KNL system has a 1Mbyte L2 cache shared by two cores. The inner (k) loop in Alg. 9 traverses K distinct elements, and the middle (j) loop traverses B iterations, resulting in accessing a total of BK elements of D. With a banded matrix, the set of column indices for adjacent rows almost completely overlaps (except for two elements at the ends of the band), resulting in almost complete reuse of the data from D if sufficient cache capacity is available. For K = 128, setting 4*128*B = 512K gives B = 1024, consistent with the experimental data that shows a drop in performance when B is raised from 1025 to 2049.

Thus we can observe that data reuse for the input dense matrix D may suffer significantly if too many other rows are accessed before a row is again referenced (when the next non-zero in the corresponding column of S is accessed). The extent of achieved reuse of D is thus highly dependent on the sparsity structure of S. In the extreme case, for Alg. 9, no reuse at all may be achieved for D, while full reuse is achieved for S and O. This is

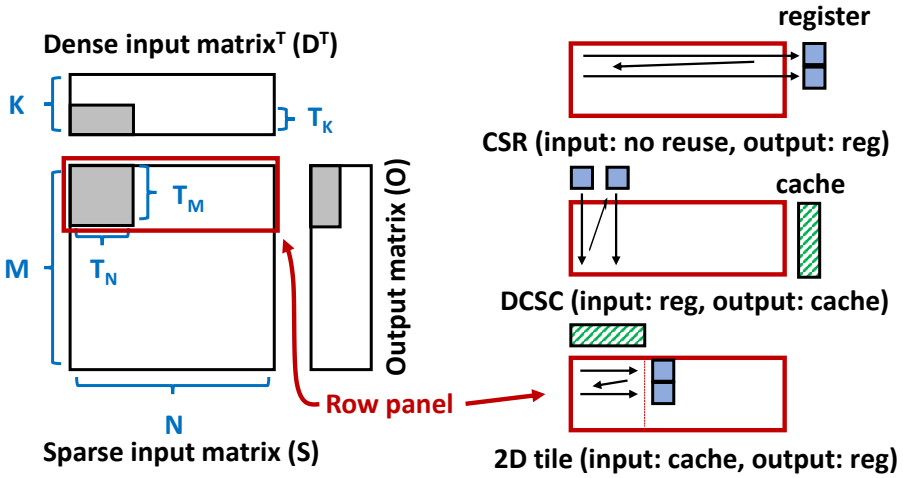


Figure 5.4: Data reuse with three different data representations

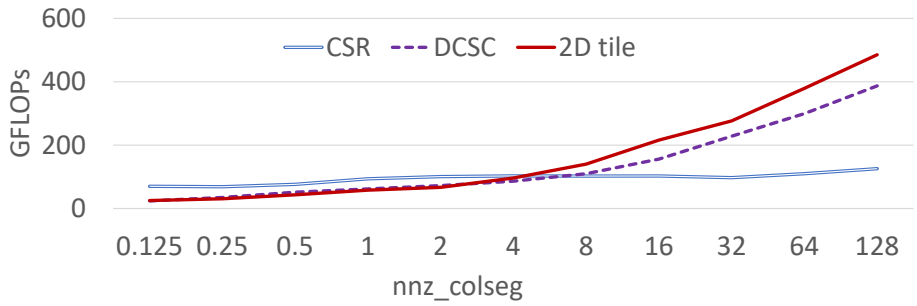


Figure 5.5: Performance of CSR, DCSC, and 2D tiles for different synthetic matrices

illustrated in Fig. 5.4 – the element $O[i][j]$ in Alg. 9 being placed in a register because of its repeated access in the innermost loop.

In order to achieve better reuse for the elements of D, the order of accessing the elements of S must be changed. For a banded sparse matrix, full reuse of D can be achieved by accessing S column-wise. But full column-wise access will result in loss of reuse for O. By performing column-wise access within row panels of S, it becomes feasible to still

achieve full reuse for O in cache, as well as some reuse for D. This corresponds to the use of DCSC data representations, with the access pattern shown in Fig. 5.4. While this DCSC scheme may be superior in terms of minimization of data movement to/from memory, its access pattern may be detrimental to Instruction Level Parallelism (ILP) due to the very small average number of non-zeros within the active columns in a row panel. Further, the number of register loads/stores increases with this scheme, since each operation requires a load-modify-store from/to registers. A third alternative is to use 2D tiling, as shown in Fig. 5.4, where access is row-wise within a tile, allowing better register-level reuse for the accumulated results.

The trade-offs between these three alternatives depend on the sparsity structure of the matrix and are very difficult to model analytically due to the complex interplay between the impact of reducing the volume of data access from memory and the increase in stall cycles due to reduced ILP. Therefore, we used micro-benchmarks based on synthetic random matrices to understand the performance trends for the three alternative schemes shown in Fig. 5.4. Fig. 5.5 shows the performance (single precision) for different synthetic sparse matrices on an Intel Xeon Phi KNL. The non-zeros in the synthetic matrices are randomly distributed with different sparsities, and *nnz_colseg* (the average number of elements in a column segment in DCSC) is computed as $\frac{T_M \times nnz}{M \times N}$, where T_M is the recommended row segment size for DCSC, and nnz is the number of non-zeros in the sparse matrix with M = 128K, N = 4K, K = 128. To fully exploit the L2 cache on KNL, the row panel size is chosen as 512 and 256, for DCSC and 2D tiles, respectively (the row panel size is halved for 2D tiles since the cache is used for both dense input D and dense output O matrices).

With CSR, the non-zero elements of the sparse matrix in a row are accessed one by one and multiplied by the corresponding elements in D. The partial results are accumulated

in registers and written out to memory at the end of each row. The O elements get full reuse in registers. However, the reuse of D elements is highly dependent on the sparsity structure of S and for large sparse matrices the reuse for these elements can be very low. In other words, with the CSR representation, it is difficult to exploit locality for D. This performance impact is shown in Fig. 5.5

Performance can be improved by exploiting the reuse of D elements. When the number of elements in a column segment (*nnz_colseg*) is high, DCSC targets the improvement of the reuse of D. In DCSC (Fig. 5.4), the sparse matrix is partitioned into a set of row panels, each of which has T_M contiguous rows. The size of a row panel (T_M) is chosen such that the corresponding O elements can fit in the L1/L2 cache (or shared memory in case of GPUs), i.e., $T_M \times T_K \leq$ cache size. Each non-empty column-segment of the row panel is processed sequentially. The D elements corresponding to the column are brought into registers and the partial results are accumulated in the cache. Thus, within a row panel, the D elements get full reuse from registers and the O elements get full reuse from the cache. In DCSC, the reuse for D is increased, but the reuse of O elements is from the cache as opposed to registers in CSR (register accumulations are faster). Hence, as shown in Fig. 5.5, the performance with a DCSC representation increases with *nnz_colseg*. When *nnz_colseg* is low, the standard CSR representation outperforms DCSC.

2D tiling can be used to achieve good reuse of D *and* O elements. In 2D tiling (in Fig. 5.4), the sparse matrix is partitioned into a set of row panels which are further subdivided into a set of 2D tiles, such that each tile has T_M rows and T_N columns of the sparse matrix. The elements within a 2D tile are represented in CSR format (other formats can also be used). In this scheme, the D elements are loaded to cache and the partial accumulations in each row of the 2D tile are done in registers (similar to CSR). However, as shown

in Fig. 5.5, when *nnz_colseg* is low, the performance of the 2D tiling scheme is lower than CSR.

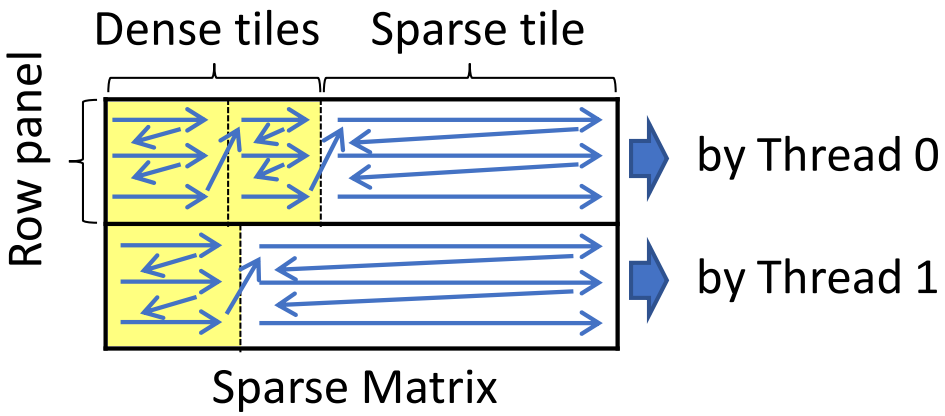


Figure 5.6: SpMM with ASpT on many-cores

The ASpT scheme is based on the observation that when columns in a 2D tile have sufficiently high *nnz_colseg*, 2D tiling achieves the best performance. When *nnz_colseg* is low, row-wise access with the standard CSR algorithm is best. Our empirical evaluation with synthetic benchmarks did not reveal scenarios where DCSC performance is the best. Therefore, we use a combination of row-wise CSR access and 2D-tiled execution. Fig. 5.6 shows the high-level idea behind the Adaptive Sparse Tiling (ASpT) approach that we describe in detail in the next section. The sparse matrix is first divided into row panels, where the row panel size is determined by cache/scratchpad capacity constraints. Within each row panel, column segments are classified as sufficiently dense or not (the threshold is dependent on the target system and is determined from the cross-over point between CSR and 2D tiles performance with the micro-benchmarking using synthetic matrices (e.g., Fig. 5.5). The columns within a row panel are then reordered so that columns over the

threshold are placed in 2D tiles, while all columns below the threshold are placed at the right end of the row panel in a large group targeted for untiled row-wise CSR execution. The horizontal sizing of the 2D tiles is explained in the next section.

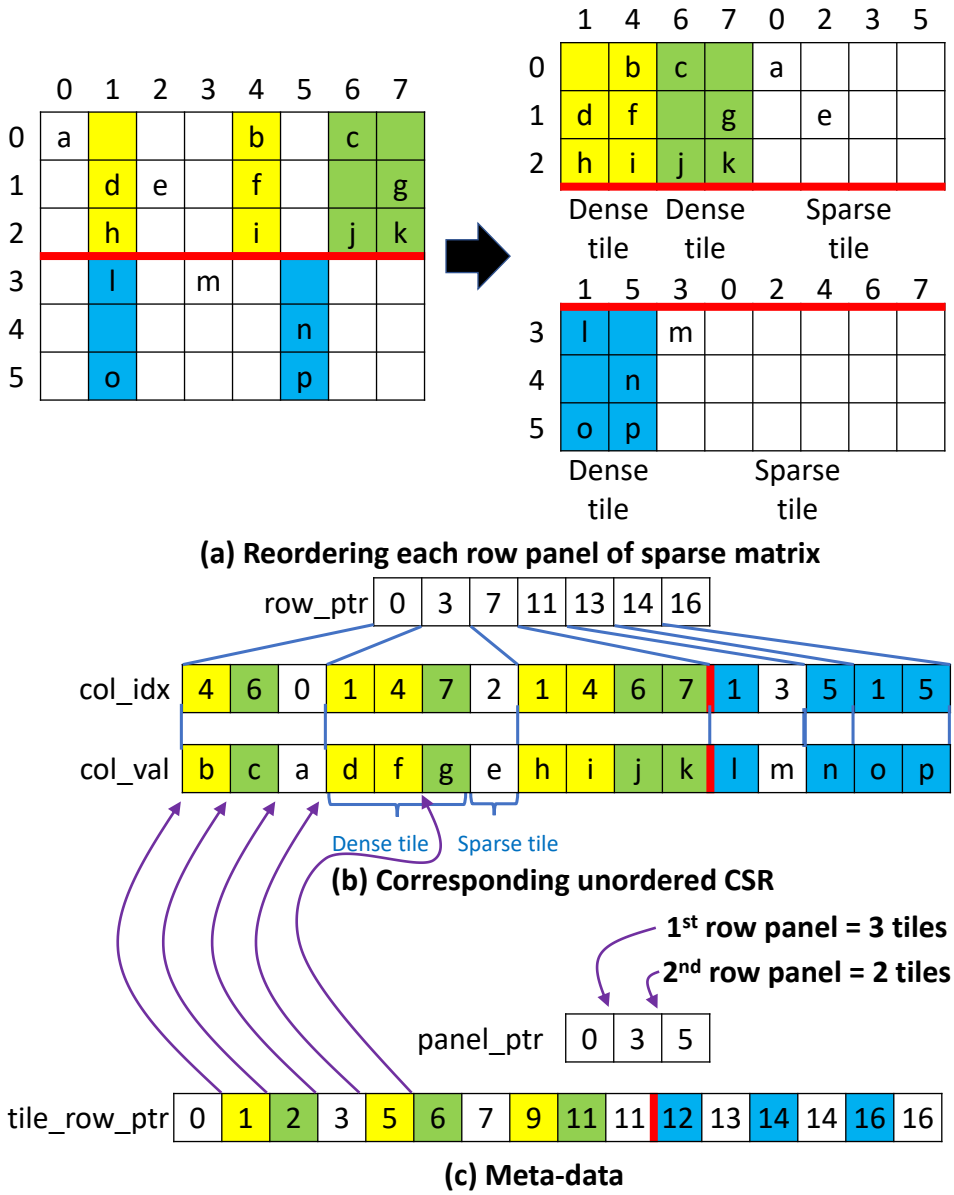


Figure 5.7: Splitting sparse matrix into heavily clustered row-segments and remainder

5.4 SpMM with ASpT

5.4.1 Data Representation

Our SpMM scheme uses the unordered CSR representation with additional metadata, as depicted in Fig. 5.7. Figs. 5.7 (a) and (b) show the conceptual view of the sparse matrix and the corresponding unordered CSR representation, respectively (the corresponding ordered CSR representation is seen in Fig. 5.2 (b)). In Fig. 5.7 (a), the entire matrix is split into two row panels, where each row panel contains a set of contiguous rows. A column segment within a row panel is classified as heavy if it has at least two non-zeros. Each row panel of the sparse matrix is reordered as seen in Fig. 5.7 (b). All the heavy columns in a row panel are placed before the light columns. Each reordered row panel can thus be viewed as two segments, where the first segment contains a set of heavy columns and the second segment consists of light columns. The first segment (heavy) is further subdivided into 2D tiles, while the entire second segment is viewed as a single 2D tile. The width of the 2D tiles in the heavy segments of row panels is selected such that the corresponding elements of D fit in the cache (or shared memory). We note that our approach only performs reordering of non-zeros but **not** re-numbering (i.e., column indexes of the non-zero elements remain unchanged).

Additional metadata in 'tile_row_ptr' keeps track of the start and end pointers of each tile (Fig. 5.7 (c)) within each row. For example, consider the first row of the reordered matrix. The first tile begins at position '0'. Hence, *tile_row_ptr[0]* is '0'. The first element corresponding to the second tile begins at position '1'; hence, *tile_row_ptr[1]* is '1', and so on. The number of 2D tiles in each row panel is encoded in *panel_ptr[i]*. For a given row panel, the number of 2D tiles can be obtained by subtracting *panel_ptr[i]* from *panel_ptr[i+1]*.

5.4.2 SpMM on Multi/many-cores

Listing 5.1: SpMM with ASpT on multi-cores

```
1. #pragma omp parallel
2. for row_panel_id=0 to num_row/panel_size-1 do
3.  num_tiles = panel_ptr[row_panel_id+1]-panel_ptr[row_panel_id];
4.  // if tile_id == num_tile-1, sparse tile is processed. Otherwise dense.
5.  for tile_id=0 to num_tile-1 do
6.   for i=0 to panel_size-1 do
7.    ptr = panel_ptr[row_panel_id]*panel_size + i*num_tile + tile_id;
8.    out_idx = i+row_panel_id*panel_size;
9.    low = tile_row_ptr[ptr]
10.   high = tile_row_ptr[ptr+1];
11.   for j = low to high-1 do
12.    #pragma simd
13.    for k = 0 to K-1 do
14.     // inputs is expected to be in cache in dense tiles
15.     // output is expected to be in register
16.     O[out_idx][k] += col_val[j] * D[col_idx[j]][k];
17.    done
18.   done
19.  done
20. done
21.done
```

Listing 5.1 shows the ApST SpMM algorithm specialized for multi/many-core processors. The row panels of the sparse matrix are distributed among threads (lines 2-21). As mentioned in the previous sub-section, the entire row panel is split into a set of heavy tiles ($tile\_id < num\_tile - 1$) and a single sparse tile ($tile\_id == num\_tile - 1$). Both the heavy tiles and the sparse tile are processed by the same kernel; however, we expect most of the D accesses in heavy tiles to be served by the L2 cache and from memory for the sparse tiles. The tiles within a row panel are processed sequentially (line 5). The elements of each row within a 2D tile are identified by using *tile_row_ptr[0]* and *panel_ptr[i+1]* (lines 7-10). The non-zero elements in each row of the 2D tile are processed sequentially (lines 11-18). In order to increase Instruction Level Parallelism (ILP), vectorization is done along the K dimension. This also helps to achieve good cache line utilization.

5.4.3 SpMM on GPUs

Although we can use the same high-level tiling idea for GPUs, the GPU implementation should take advantage of the GPU architecture in order to achieve high-performance. The main difference between GPUs and multi/many-core processors is that GPUs have many more registers per thread. GPUs also have an explicitly managed scratchpad memory called shared memory. However, the cache capacity per thread in GPUs is quite small ($\frac{cache\ size}{\#\ of\ threads\ on\ SM\ or\ SMs}$).

Utilization of Shared Memory and Registers

Contrary to multi/many-cores, where only a few threads access L1/L2 cache simultaneously, a large number of threads can access GPU caches at the same time. For instance, on the NVIDIA Pascal P100, 2K and 112K threads can simultaneously access the same 24KB L1 and 4MB L2 caches. If each thread accesses unique memory locations, then each thread can only use $\frac{24K}{2K} = 12B/threads$ L1 and $\frac{4MB}{112K} = 37B/threads$ L2 cache. However, GPUs have a large number of registers and shared memory per Streaming Multiprocessor (SM). For example, P100 has 256KB storage capacity in registers and 64KB shared memory per each SM. The shared memory bandwidth on the P100 is higher than that of L1 cache. Therefore, utilizing these resources to improve locality would be beneficial for performance. In GPUs, only the memory locations with statically resolvable access patterns can be placed in registers. Since we process each row sequentially, the access pattern of O can be statically determined, and these elements can be placed in registers. However, the accesses for D depend on the sparsity structure; hence the accesses are kept in shared memory.

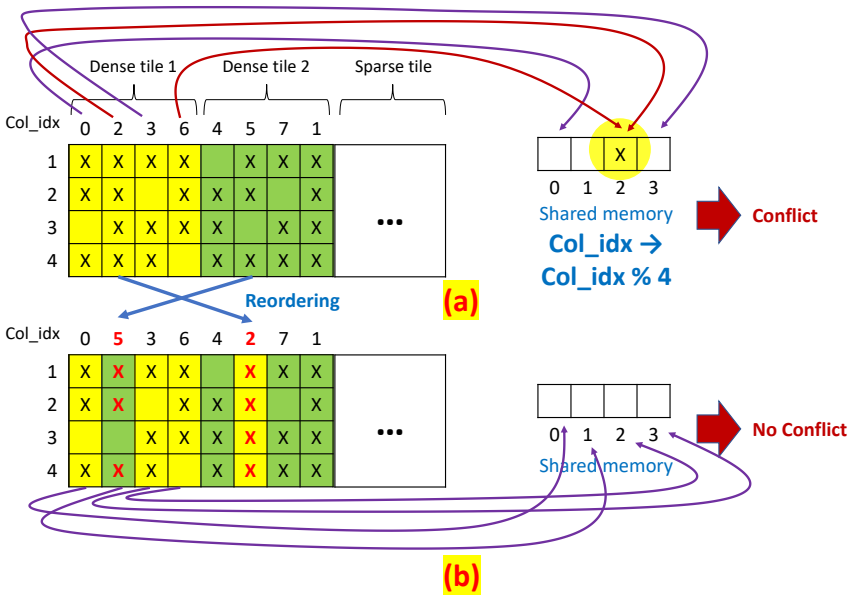


Figure 5.8: Remove index mapping conflicts to shared memory

We next describe the approach to mapping columns of D to shared memory. Consider Fig. 5.8 and assume that column 'i' of D is mapped to column $i\%4$-th column in the shared memory. This would result in mapping the first (col_idx:2) and third columns (col_idx:6) of the yellow tile to the same location in shared memory (Fig. 5.8 (a)), resulting in a conflict. Alternatively, the needed columns of D could be mapped contiguously in shared memory, with an indirection array to indicate the mapping of column indices to shared memory. However, this strategy incurs two major overheads: i) extra space required for the indirection array and ii) overhead due to access of the indirection array. For each non-zero access, the indirection array needs to be accessed to find the element in shared memory, which is inefficient. We addressed this issue by reordering column indices to remove mapping conflicts in each tile. That is, every column index in the tile is mapped to a different location in the shared memory. By doing so, we can directly access the shared memory using a simple

modulo operation. For example, in Fig. 5.8 (b), 'modulo 4' mapping can be used. This strategy may result in partially filled tiles. If a heavy 2D tile does not have enough column segments, they are moved to the sparse segment. The reordering can be easily done once during the pre-processing stage.

Listing 5.2: SpMM with ASpT on GPUs (dense tile)

```
1. row_panel_id = tb_idx;
2. row_offset = tid/WARP_SIZE;
3. slice_base = tb_idy*WARP_SIZE;
4. slice_offset = tid%WARP_SIZE;
5. for tile_id=0 to panel_ptr[row_panel_id+1]-panel_ptr[row_panel_id]-1 do
6.  for i=row_offset to TILE_WIDTH-1 step tb.size()/WARP_SIZE do
7.   map_id = map_list[panel_ptr[row_panel_id]+tile_id][row_offset];
8.   sm_D[map_id%TILE_WIDTH][slice_offset] = D[map_id][slice_base+slice_offset];
9.  done
10. __syncthreads();
11. // processing dense blocks
12. for i=row_offset to panel_size-1 step tb.size()/WARP_SIZE do
13.  ptr = panel_ptr[row_panel_id]*panel_size + i*(panel_ptr[row_panel_id+1]
          -panel_ptr[row_panel_id]) + tile_id;
14.  out_idx = i+row_panel_id*panel_size/WARP_SIZE;
15.  low = tile_row_ptr[ptr]
16.  high = tile_row_ptr[ptr+1];
```

```
17.  buf_0 = 0;
18.  for j=low to high-1 do
19.   buf_0 += col_val[j] * sm_D[col_idx[j]%TILE_WIDTH][slice_offset];
20.  done
21.  O[i+row_panel_id*panel_size][slice_base+slice_offset] += buf_0;
```

```
22.  __syncthreads();
23. done
24.done
```

Listing 5.3: SpMM with ASpT on GPUs (sparse tile)

```
1. row_panel_id = tb_idx;
2. row_offset = tid/WARP_SIZE;
3. slice_id = tb_idy*WARP_SIZE;
4. slice_offset = tid%WARP_SIZE;
5. // processing a sparse block
6. for i=row_offset to panel_size-1 step tb.size()/WARP_SIZE do
7.  ptr = panel_ptr[row_panel_id]*panel_size + (i+1)*(panel_ptr[row_panel_id+1]
         -panel_ptr[row_panel_id])-1;
8.  out_idx = i+row_panel_id*panel_size;
9.  low = tile_row_ptr[ptr];
10. high = tile_row_ptr[ptr+1];
```

```
11. buf_0 = 0;
12. for j=low to high-1 do
13.  buf_0 += col_val[j] * D[col_idx[j]][slice_base+slice_offset];
14. end
15. O[i+row_panel_id*panel_size][slice_base+slice_offset] += buf_0;
```

```
16.end
```

SpMM Algorithm: GPUs

Listings 5.2 and 5.3 show the SpMM-ASpT GPU algorithm for dense and sparse tiles, respectively. The GPU algorithm is similar to that for multi/many-cores. The different threads in a warp are mapped along K to avoid thread divergence and to achieve good load balance. For heavy 2D tiles, the corresponding elements of D are brought to shared memory, whereas for light 2D tiles shared memory is not used. Each 2D tile is processed by a thread block.

For processing both dense and sparse tiles, row panel ID, row offset, and slice index are first computed (lines 1-4 in Listings 5.2 and 5.3). Then, for dense tiles, all of the threads in a thread block collectively bring the corresponding elements of D to shared memory (lines 6-9 in Listing 5.2). The *map_id* (line 7 in Listing 5.2) keeps track of the original column index, and is used to access elements of D.

The rest of the code (lines 12-23 in Listing 5.2 and lines 6-16 in Listing 5.3) is very similar to the multi/many-core algorithm. Different warps process different rows within a row panel, and the threads within a warp are distributed along K. The results are accumulated in registers and written out to global memory at the end of each row (line 21 in Listing 5.2 and line 15 in Listing 5.3).

5.4.4 Parameter Selection

The key parameters that affect performance for ASpT are i) the threshold for the number of non-zeros in a column segment to be classified as heavy and ii) the tile sizes (T_M, T_N, and T_K). These parameters were empirically determined using synthetic matrices described in Sec. 5.3. Fig. 5.5 shows the performance of CSR, DCSC and 2D tiles as a function of column density. The threshold for classifying a column segment as heavy is chosen as the

minimum column density at which 2D tiles outperforms CSR. The rationale is that heavy segments are processed by the 2D tile algorithm, whereas the light segment, even though represented as a single 2D tile, is processed by the CSR algorithm. Thus, if the number of non-zeros in a column segment is less than the crossover point in Fig. 5.5, it is better to process those non-zeros using the CSR algorithm, otherwise the 2D tile algorithm. The tile sizes were also chosen empirically, such that the data footprint of the tile fits in the L2 cache (512 KB per core) for the KNL (i.e., $(T_M + T_N) \times T_K \times sizeof(word) \times \frac{\#\_of\_threads}{\#\_of\_cores}$ = $512K$). For our experiments, the L1 cache was too small to exploit locality (and thus did not produce great benefits). We explored different (T_M, T_N, T_K) subjected to the L2 footprint constraint, and selected the best performing parameters. The best performance was obtained when $T_M = T_N$, $T_K = K$, and $\frac{\#\_of\_threads}{\#\_of\_cores} = 2$.

We followed similar steps for selecting GPU parameters. Since D elements are kept in shared memory, the tile sizes T_K and T_N are constrained by the shared memory capacity. The shared memory size per thread block was selected such that full occupancy was maintained (for P100 we assigned 32KB of shared memory per thread block of size 1024). Since the elements of O are kept in registers, T_K and T_M are constrained by the register capacity. Thread coarsening [86, 84] was also employed to improve performance.

5.5 SDDMM with ASpT

In SDDMM, two dense matrices are multiplied and the resulting matrix is then scaled using an element-wise multiplication (Hadamard product) with a sparse matrix. Since the sparsity structure of the input and output sparse matrices is the same, we can optimize SDDMM by forming dense matrix products only at locations corresponding to non-zero elements in the input sparse matrix, as done by existing implementations [26, 66].

Listing 5.4: Part of SDDMM on multi-cores

```
11. #pragma simd
12. for j = low to high-1 do
13.  #pragma simd reduction
14.  for k = 0 to K-1 do
15.   // D2 is expected to be in cache in dense tiles
16.   // output_col_val is expected to be in register
17.   O[j] += D2[out_idx][k] * D[col_idx[j]][k];
18.  done
19.  O[j] *= col_val[j];
20. done
```

Listing 5.5: Part of SDDMM on GPUs (dense tile)

```
11. buf_D2 = D2[i+row_panel_id*panel_size][slice_base+slice_offset];
12. for j=low to high-1 do
13.  buf_O = buf_D2 * sm_D[col_idx[j]%TILE_WIDTH][slice_offset];
14.  for k=WARP_SIZE/2 downto 1 step k=k/2 do
15.   buf_O += __shfl_down(buf_output, k);
16.  done
17.  if slice_offset == 0 then
18.   O[j] += buf_O * col_val[j];
19.  end
20. done
```

Listing 5.6: Part of SDDMM on GPUs (sparse tile)

```
11. buf_D2 = D2[i+row_panel_id*panel_size][slice_base+slice_offset];
12. for j=low to high-1 do
13.  buf_O = buf_D2 * D[col_idx[j]][slice_base+slice_offset];
14.  for k=WARP_SIZE/2 downto 1 step k=k/2 do
15.   buf_O += __shfl_down(buf_O, k);
16.  done
17.  if slice_offset == 0 then
18.   O[j] += buf_O * col_val[j];
19.  end
20. done
```

SDDMM on multi/many-cores can be implemented by substituting the box (lines 11-18) in Listing 5.1 with Listing 5.4. For SDDMM, the K dimension is not tiled, and the tile sizes T_M and T_N are chosen such that both *D1 and D2* fit in the L2 cache. The *for loop* in line 11 computes the dot product of *D1 and D2*. The inner *for loop* (line 14) corresponding to K is vectorized. Unlike SpMM, SDDMM requires reduction across the K dimension and is implemented by specifying the 'reduction clause' in line 17. Line 19 scales the result by multiplying it with the corresponding element in the input sparse matrix S.

SDDMM on GPUs for dense and sparse tiles can be implemented by replacing the box in Listings 5.2 and 5.3 by Listings 5.5 and 5.6, respectively. The only difference between

Listings 5.5 and 5.6 is in line 13, where elements of *D1* are served by shared-memory in the dense version and global memory in the sparse version. The elements of *D2* corresponding to the row are kept in registers for both dense and sparse tiles (line 11 in Listing 5.5). Since the K dimension is mapped across threads and the corresponding O elements are kept in registers that are private to a thread, we use warp shuffling for reduction (lines 14-16 in Listing 5.5). The accumulated output value is scaled and written back to global memory (lines 17-19 in Listing 5.5).

5.6 Experimental Evaluation

This section details the experimental evaluation of the ASpT-based SpMM and SDDMM on three different architectures:

- NVIDIA P100 GPU (56 Pascal SMs, 16GB global memory with a bandwidth of 732GB/sec, 4MB L2 cache, and 64KB shared memory per each SM)
- Intel Xeon Phi (68 cores at 1.40 GHz, 16GB MCDRAM with a bandwidth of 384GB/sec, 34MB L2 cache)
- Intel Xeon CPU E5-2680 v4 (2 $\times$ 14 cores at 2.40 GHz, 16GB MCDRAM with a bandwidth of 72GB/sec, 35MB L3 cache)

For GPU experiments, the code was compiled using NVCC 9.1 with the -O3 flag and was run with ECC turned off.

For the Intel Xeon Phi, the code was compiled using Intel ICC 18.0.0 with -O3 and -MIC-AVX512 flags. The clustering mode was set to 'All-to-All' and the memory mode was set to 'cache-mode' (to fit big datasets).

For the Intel Xeon CPU, the code was also compiled with Intel ICC 18.0.0 using -O3 flag.

We only include the kernel execution time for all experiments. Preprocessing time and data transfer time from CPU to GPU or disk to RAM are not included. The impact of pre-processing overhead is reported separately. All tests were run five times, and average numbers are reported.

5.6.1 Datasets and Comparison Baseline

For experimental evaluation, we selected 975 matrices from the SuiteSparse collection [35], using all matrices with at least 10K rows, 10K columns, and 100K non-zeros. The matrices in SuiteSparse are from diverse application domains and represent a wide range of sparsity patterns.

For SpMM on KNL, we compared ASpT with Intel MKL [135], CSB [5], and TACO [66], which represent the current state-of-the-art SpMM implementations.

For SpMM on GPUs, we compared ASpT with NVIDIA cuSPARSE. cuSPARSE offers two modes, and we compare against the better performing one. We did not compare ASpT with MAGMA [8] and CUSP [33], as they are consistently outperformed by cuSPARSE (more than 40% on average).

For SDDMM on manycores (KNL), we compared ASpT with TACO which has been shown to significantly outperform Eigen [42] and uBLAS [134]. For SDDMM on GPUs, we compared ASpT with BIDMach [26], which represents the state-of-the-art SDMMM implementation.

We evaluated ASpT with single precision (SP) and double precision (DP), with the number of vectors set to 32 and 128 (K = 32,128). However, only SP was used for SDDMM comparison on GPUs, since BIDMach does not support DP for SDDMM.

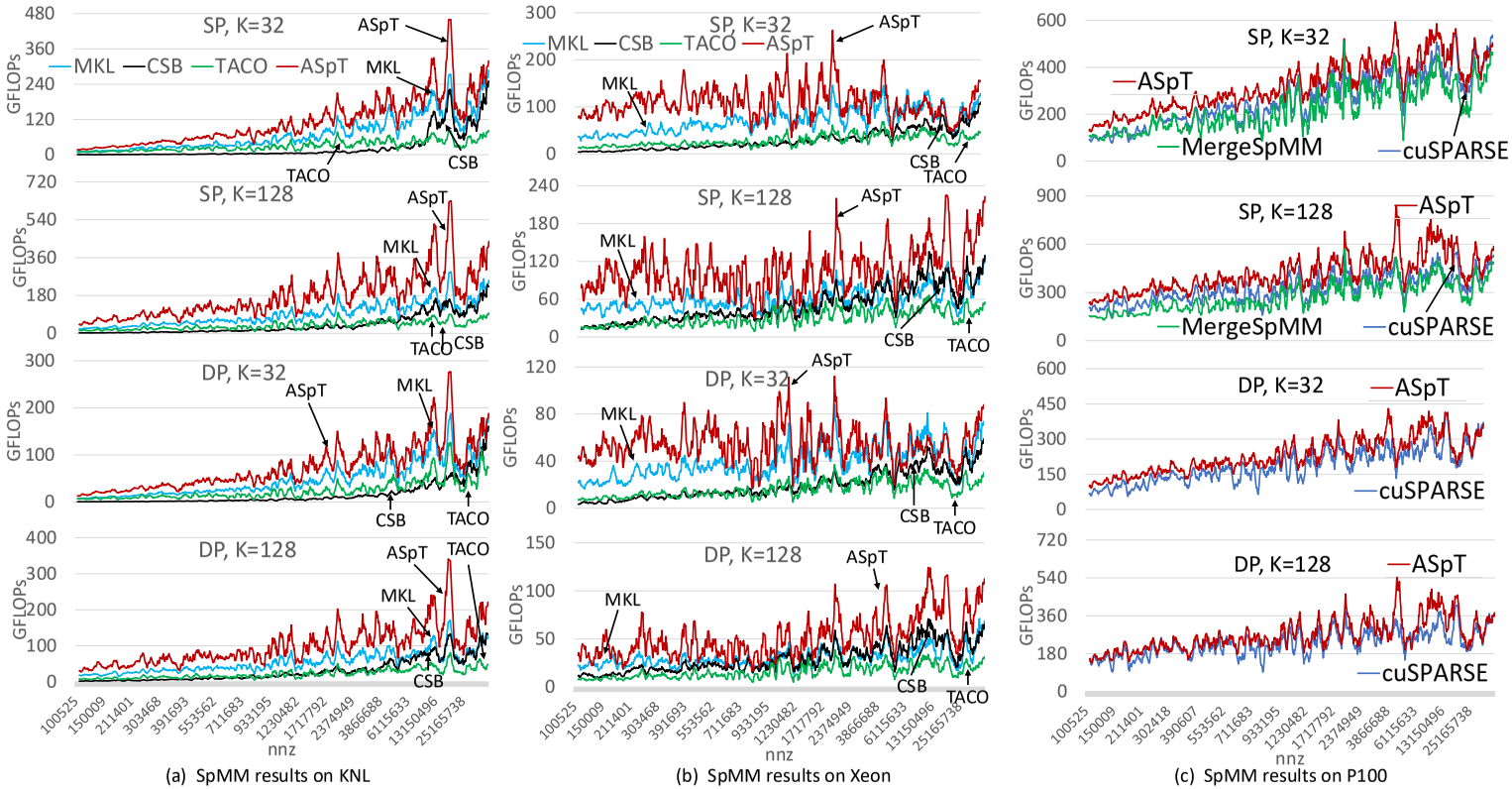


Figure 5.9: SpMM results

Table 5.1: Summary performance comparison: SpMM

		percentage											
		KNL				Xeon				GPU			
		SP		DP		SP		DP		SP		DP	
		K=32	K=128	K=32	K=128	K=32	K=128	K=32	K=128	K=32	K=128	K=32	K=128
slowdown	>100%	1.7%	1.4%	1.4%	1.7%	5.1%	1.0%	4.0%	1.2%	0.1%	0.0%	0.1%	0.1%
	50%~100%	1.9%	0.7%	1.2%	1.4%	6.7%	3.6%	7.9%	2.6%	0.1%	0.0%	0.1%	0.4%
	10%~50%	4.2%	4.0%	5.2%	6.1%	12.6%	13.9%	16.4%	10.8%	1.7%	1.7%	3.2%	10.4%
	0%~10%	3.2%	2.5%	4.1%	3.3%	6.2%	7.3%	6.3%	5.9%	11.2%	6.2%	12.3%	18.8%
speedup	0%~10%	2.6%	3.0%	3.7%	3.9%	4.2%	7.4%	7.4%	6.9%	24.9%	18.8%	21.4%	22.9%
	10%~50%	19.4%	15.8%	25.0%	20.2%	15.7%	26.7%	20.3%	32.2%	49.5%	53.5%	44.1%	34.0%
	50%~100%	35.0%	28.8%	33.1%	32.5%	18.5%	20.6%	15.3%	24.6%	6.1%	16.4%	5.1%	3.6%
	>100%	32.0%	43.8%	26.5%	31.0%	31.1%	19.5%	22.4%	15.9%	6.5%	3.5%	13.5%	9.8%

5.6.2 SpMM

Fig. 5.9 (a) shows SpMM performance with SP and DP for different K widths on the KNL. Each point in Fig. 5.9 (a) represents the average GFLOPs value for a contiguous set of 10 matrices – the matrices are sorted in ascending order by the number of non-zeros. As shown in Fig. 5.9 (a), TACO is outperformed by CSB, which is outperformed by MKL. For all configurations, ASpT outperforms other implementations. ASpT performance is improved when K is increased from 32 to 128. For matrices with a small number of non-zeros, performance is low. This is because concurrency is very low or there is not enough data reuse (i.e., $\frac{nnz}{\#\_of\_cols}$ is small). An overall summary of the relative performance across the set of matrices is presented in Table 5.1. The speedup and slowdown are defined as $\frac{GFLOPs\_with\_ASpT}{GFLOPs\_with\_best\_comparsion\_baseline} - 1$ and $\frac{GFLOPs\_with\_best\_comparsion\_baseline}{GFLOPs\_with\_ASpT} - 1$, respectively. ASpT achieves significant speedup for most matrices and only suffers slowdown for a small fraction of the matrices.

Fig. 5.9 (b) shows SpMM performance with SP and DP for different K widths on the Intel Xeon multicore CPU. As shown in Fig. 5.9, the relative performance trends of Xeon and KNL are quite similar.

Fig. 5.9 (c) shows SpMM performance on the NVIDIA Pascal P100 GPU. The performance gap between ASpT and cuSPARSE is higher for higher K widths. The performance of cuSPARSE does not improve much when K is increased. Fig. 5.9 (c) also compares ASpT with Merge-SpMM [151]. The comparison was limited to single precision as Merge-SpMM does not support double precision. For each dataset, Merge-SpMM reports GFLOPs with different strategies, and we took the best among them. Merge-SpMM's performance is slightly inferior to that of cuSPARSE, which is outperformed by ASpT.

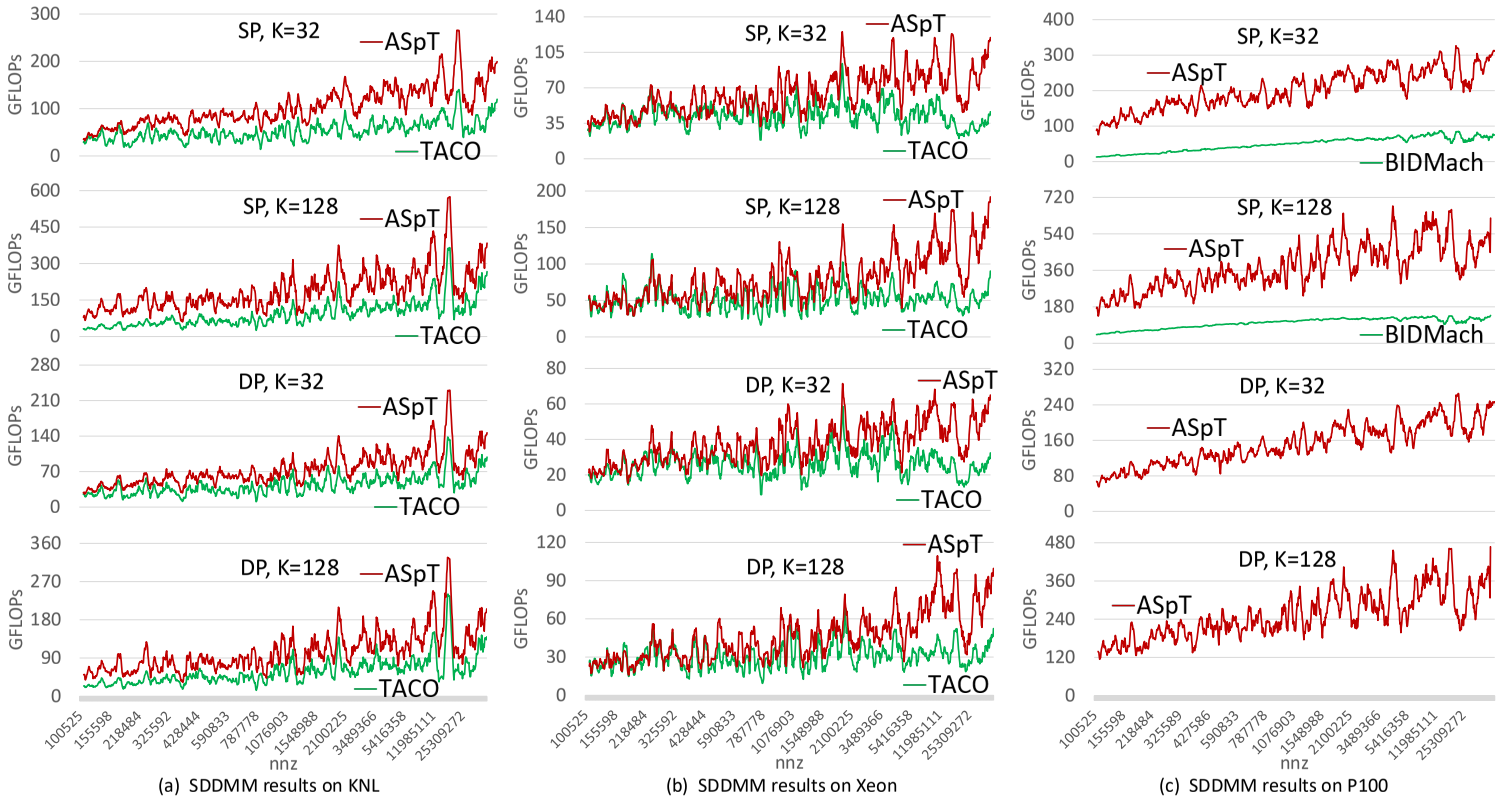


Figure 5.10: SDDMM results

Table 5.2: Summary performance comparison: SDDMM

		percentage									
		KNL				Xeon				GPU	
		SP		DP		SP		DP		SP	
		K=32	K=128	K=32	K=128	K=32	K=128	K=32	K=128	K=32	K=128
slowdown	>100%	0.0%	0.0%	0.0%	0.0%	0.0%	0.5%	0.0%	0.1%	0.0%	0.0%
	50%~100%	0.1%	0.2%	0.3%	0.1%	1.3%	2.3%	0.1%	1.5%	0.0%	0.0%
	10%~50%	2.2%	2.5%	4.2%	3.6%	15.6%	15.5%	10.6%	11.3%	0.0%	0.2%
	0%~10%	3.3%	1.4%	3.1%	2.8%	11.2%	7.0%	15.7%	9.1%	0.0%	0.0%
speedup	0%~10%	2.7%	2.2%	3.8%	3.2%	4.5%	9.0%	9.0%	9.5%	0.0%	0.1%
	10%~50%	16.3%	14.4%	18.6%	22.8%	20.5%	17.8%	20.4%	25.4%	0.2%	1.2%
	50%~100%	21.8%	24.9%	27.5%	29.3%	14.1%	16.3%	19.8%	17.8%	1.4%	11.3%
	>100%	53.6%	54.4%	42.5%	38.2%	32.8%	31.6%	24.5%	25.3%	98.4%	87.2%

5.6.3 SDDMM

Figs. 5.10 (a) and (b) show the SDDMM performance on the KNL and Xeon, respectively. The performance trend is similar to SpMM in Figs. 5.9 (a) and (b), but the absolute GFLOPs is lower. On KNL, ASpT outperforms TACO for the majority of the datasets as shown in Table 5.2. Table 5.2 also shows a similar trend on Xeon CPU.

Fig. 5.10 (c) presents SDDMM performance on GPUs. The performance trend is similar to that of SpMM in Fig. 5.9 (c), with lower absolute GFLOPs. Both BIDMach and ASpT improve when K is increased, and ASpT significantly outperforms BIDMach across all matrices. We only report BIDMach performance for single precision, since it does not support double precision.

Table 5.3: Details of preprocessing overhead

ratio (preprocessing_time /computing_time)	KNL (SP)	KNL (DP)	CPU (SP)	CPU (DP)	GPU (SP)	GPU (DP)
0~5	64.7%	89.7%	87.8%	98.8%	75.4%	83.7%
5~10	31.1%	9.4%	11.3%	1.0%	14.5%	13.6%
10~15	3.2%	0.6%	0.9%	0.1%	7.2%	2.2%
15~20	0.8%	0.2%	0.0%	0.0%	2.0%	0.4%
20~25	0.0%	0.0%	0.0%	0.0%	0.6%	0.1%
25~30	0.1%	0.0%	0.0%	0.0%	0.2%	0.0%

5.6.4 Preprocessing Overhead

Constructing additional meta-data and reordering the column indices incurs overhead. Fig. 5.11 shows the preprocessing time normalized to the execution time of one ASpT SpMM with K=128, for single precision (red curve) and double precision (blue curve).

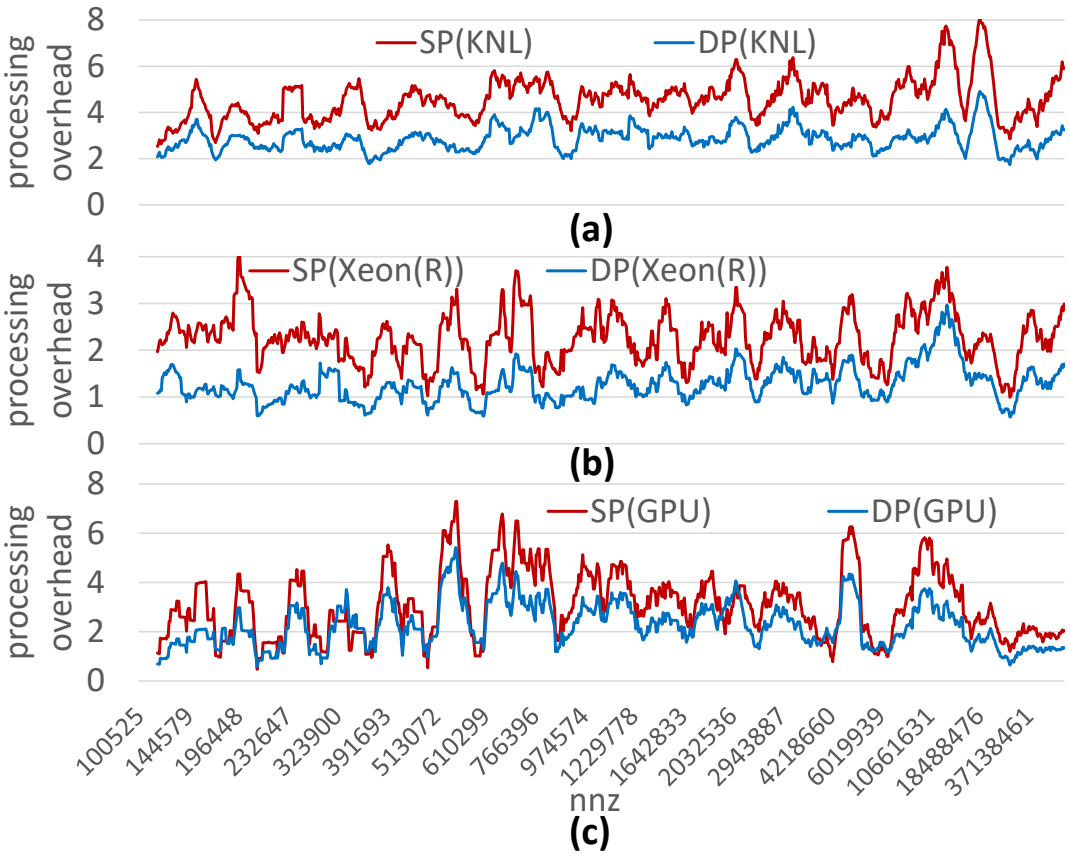


Figure 5.11: Preprocessing time (SpMM)

Typical applications that involve SpMM and SDDMM execute a large number of iterations (e.g., [105] for SpMM and [156] for SDDMM). Hence, our preprocessing overhead is negligible. DP precision has less overhead, as the preprocessing time for SP and DP is similar; but the SpMM time for DP is higher than that of SP. Table 5.3 shows that the preprocessing times are generally between 0-5× compute time.

5.6.5 Benefit from Tiling / Reordering

While tiling helps to improve the data-reuse of the dense input matrix, it has two main disadvantages: i) tiling overhead and ii) loss of concurrency for very sparse tiles. Tiling

Table 5.4: Benefit of tiling or reordering

	Tiling-Only	Tiling+Reordering
SpMM(KNL)/speed-up over MKL	1.38	2.06
SpMM(GPU)/speed-up over cuSPARSE	0.8	1.34
SDDMM(KNL)/speed-up over TACO	1.02	2.01
SDDMM(GPU)/speed-up over BIDMach	2.41	4.04

only helps to improve the data-reuse for columns with sufficient density. Hence, tiling overheads can be minimized by limiting the tiling to columns with sufficient column density. A simple 2D tiling strategy would include columns of both low and high density, which may affect performance. The performance can be improved by grouping columns with high density and limiting tiling to these high-density columns; This can be achieved using reordering. Note that reordering without tiling may not improve performance, as the data-reuse may not be improved. However, it is possible that the inherent matrix structure may already be clustered, and thus tiling without any reordering could potentially improve performance. Hence, we ran experiments with only tiling and no reordering. Results with K = 128 for single precision are presented in Table 5.4; other configurations achieved similar results. As shown in Table 5.4, the combined Tiling+Reordering strategy substantially outperforms the Tiling-Only strategy.

5.7 Conclusion

SpMM and SDDMM are key kernels in many machine-learning applications. In contrast to other efforts that use customized sparse matrix representations to achieve high-performance, our approach targets efficient implementation of these primitives using the

standard (unordered) CSR sparse matrix representation, so that incorporation into applications can be facilitated. An adaptive 2D-tiled approach exposes higher memory reuse potential, and an efficient reordering scheme enables efficient execution of 2D tiles. In comparison to the current state-of-the-art, the ASpT-based SpMM algorithm achieves a speedup of up to 13.18x, with a geometric mean of 1.65x on an Intel Xeon Phi KNL; a speedup of up to 7.26x, with a geometric mean of 1.36x on an Intel Xeon multicore processor; and a speedup of up to 24.21x with a geometric mean of 1.35x on GPUs. The ASpT based SDDMM algorithm achieves a speedup of up to 30.15x, with a geometric mean of 1.93x on the KNL; a speed of up to 22.75x, with a geometric mean of 1.52x on the Xeon; and a speedup of up to 13.74x, with a geometric mean of 3.60x on GPUs.

CHAPTER 6

Conclusion and Future Research

6.1 Conclusion

This dissertation has addressed the development of a high-performance approach for regular- and irregular applications (graph algorithms, SpMM, and SDDMM) on GPUs.

For regular applications, the dissertation has taken a different approach from prior efforts in modeling performance of GPU kernels: i) we restricted our abstract execution to a small number of thread blocks for kernels; and ii) we used a simplified modeling of each hardware resource using only the two parameters, latency and initiation interval. Sensitivity analysis with respect to resource parameters identifies the bottleneck resource(s) for a given kernel's execution on a particular GPU system. The bottleneck analysis enables an efficient search approach in a space of transformed code variants.

For graph algorithms, the dissertation has used multiple data representation and execution strategies for dense versus sparse vertex frontiers, depending on the sparsity. The sampling method based on the execution time prediction chooses one of two representations.

The dissertation has also developed a hybrid approach for SpMM for GPU's by exploiting the cluster parts in sparse matrices with a novel data structure. The approach tries to get

good reuse of the elements simultaneously from input dense matrix, output dense matrix, and sparse matrix.

Experimental results demonstrate the effectiveness of our approaches.

6.2 Future Research

Despite the contributions in this dissertation and the extensive efforts which have been made to achieve high-performance on GPUs, there remain countless research opportunities for enhancing performance on GPUs. Below, we discuss two avenues for future research.

Develop a dedicated performance model on GPUs In SAAKE, two key resources are modeled without actual execution of kernels and with conservative assumptions (irregular memory accesses are always uncoalesced and dynamic data branches are always taken). This simplification would lead to inaccurate prediction especially for kernels which have irregular memory accesses or dynamic data branches, as shown in Chapter 2.

The dissertation assumed that all shared resources such as global memory bandwidth are equally distributed to SMs to focus on a single set of active blocks on one SM. This approximation may be accurate for programs with good and dynamically smooth load balance, but may be inaccurate for load-imbalanced or dynamically irregular kernels.

GPU memory hierarchy is composed of several types of memory and cache, and is very convoluted. We have not developed a model that reflects that hierarchy, and this can cause significant inaccurate preciction for global memory latency-bound or throughput-bound kernels. Hardware counters (cache hit rates) were obtained by NVPROF to resolve this issue. However, modeling GPU memory hierarchy is still an important problem that must be addressed.

In sum, the inaccuracy of the performance model can stem from dynamic memory and control irregularities, equal treatment of different SMs, and lack of cache modeling. Various cache characteristics such as associativity and policy can be approximated as shown in [89]. We can execute a small portion (a few thread blocks) of actual kernels to extract addresses in the memory layout. That is, one can devise a memory trace generator which collects memory (load and store) operations for a small portion of thread blocks. These addresses can be used to predict conditional branches, coalescing memory accesses, serialization of atomic operations, and cache hit rate. Note that considering only a few thread blocks cannot handle dynamic memory and control irregularities for some kernels. However, the cost for considering full thread blocks can be tremendous. There is a tradeoff between efficiency and accuracy, and our primary goal is to accurately model the impact of various changes in a modest amount of time for the sake of providing beneficial optimizations. Abstract kernel emulation with all SMs can also improve accuracy, since L2 cache and DRAM are shared across SMs. In addition, considering the factors below can further improve accuracy.

- Considering kernel launch overhead: the kernel launch overhead cannot be amortized if the kernel execution time for one thread block is very short. The kernel launch overhead can be approximated by using a microbenchmark, and this overhead can be added to the predicted execution time.

- Developing a dedicated memory model: we assumed memory latency and gap as constants. However, memory latency is shown to be non-uniform [89, 57]. Row buffer hit and miss can also affect memory latency and gap [57]. Developing equations for estimating memory latency and gap would help to improve accuracy.

- Extracting properties for the actual warp scheduler: As shown in many works such as [94, 75, 60], warp scheulding can significantly affect performance. We used Greedy-then-oldest (GTO) warp scheduling, which was one of the simplest warp schedulers and has since been superceded by several warp schedulers, as established by [94], which has been outperformed by [75, 60]. We believe the warp scheduler used in current GPUs is much more complicated than GTO. We plan to develop a set of microbenchmarks to extract how warp schedulers work in current GPUs.

Develop a large-scale graph processing framework on just one GPU A large body of recent research has focused on various implementations for processing large-scale graphs in distributed systems. However, processing large-scale graphs on a single GPU has received much less attention; only a few strategies have been developed [45]. Processing large-scale graphs on a single GPU is costly and has poor energy efficiency. However, only a small fraction of those graphs (including graph structure, and values of edges/vertices) can reside in the device memory on a GPU, since the graphs usually do not fit in the device memory on a GPU. Moreover, if a small fraction of graphs is brought into the device memory, the previous data on the device memory may be evicted. Thus, processing large-scale graphs on a GPU is challenging. When a portion of the graph is loaded into the device memory, existing frameworks regard the graph algorithm as a topology-driven algorithm (i.e., all vertices and edges are active for each iteration). This strategy is not suitable for processing data-driven algorithms such as BFS, SSSP, and BC. Future research needs to pursue developing data-driven graph processing frameworks on a single GPU.

BIBLIOGRAPHY

[1] Gunrock: High-performance graph primitives on GPUs. gunrock.github.io.

[2] Nvidia. 2016. the api reference guide for cusparse, the cuda sparse matrixlibrary.(v8.0 ed.). nvidia.

[3] SuiteSparse matrix collection. https://sparse.tamu.edu, 2011.

[4] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1213–1222. IEEE, 2014.

[5] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1213–1222. IEEE, 2014.

[6] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM.

[7] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. Accelerating the lobpcg method on gpus using a blocked sparse matrix vector product. In *Proceedings of the Symposium on High Performance Computing*, pages 75–82. Society for Computer Simulation International, 2015.

[8] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. Accelerating the lobpcg method on gpus using a blocked sparse matrix vector product. In *Proceedings of the Symposium on High Performance Computing*, pages 75–82. Society for Computer Simulation International, 2015.

[9] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus for graph applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 781–792. IEEE Press, 2014.

[10] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wenmei W Hwu. An adaptive performance modeling tool for gpu architectures. In *ACM Sigplan Notices*, volume 45, pages 105–114. ACM, 2010.

[11] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*. SIAM, 2000.

[12] Allison H Baker, John M Dennis, and Elizabeth R Jessup. On improving linear solver performance: A block variant of gmres. *SIAM Journal on Scientific Computing*, 27(5):1608–1626, 2006.

[13] Muthu Baskaran, Jj Ramanujam, and P Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.

[14] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high-performance computing networking, storage and analysis*, page 18. ACM, 2009.

[15] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 235–248. ACM, 2017.

[16] A. Benatia, W. Ji, Y. Wang, and F. Shi. Sparse matrix format selection with multiclass svm for spmv on gpu. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 496–505, Aug 2016.

[17] Massimo Bernaschi, Giancarlo Carbone, and Flavio Vella. Betweenness centrality on multi-GPU systems. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 12. ACM, 2015.

[18] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Notices*, volume 43, pages 101–113. ACM, 2008.

[19] Aydin Buluc and John R Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 503–510. IEEE, 2008.

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

[21] Aydin Buluc, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. Design of the graphblas api for c. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 643–652. IEEE, 2017.

[22] Aydin Buluc, Samuel Williams, Leonid Oliker, and James Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Parallel & distributed processing symposium (IPDPS), 2011 IEEE international*, pages 721–733. IEEE, 2011.

[23] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. *The International Journal of High Performance Computing Applications*, 21(4):467–484, 2007.

[24] Carmen Campos and Jose E Roman. Strategies for spectrum slicing based on restarted lanczos methods. *Numerical Algorithms*, 60(2):279–295, 2012.

[25] John Canny. Collaborative filtering with privacy. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 45–57. IEEE, 2002.

[26] John Canny and Huasha Zhao. Bidmach: Large-scale learning with zero memory allocation. In *BigLearn workshop, NIPS*, 2013.

[27] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. Falcon: A graph manipulation language for heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 12(4):54:1–54:27, December 2015.

[28] Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687. IEEE Computer Society, 2011.

[29] CUDA occupancy calculator.

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

[31] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[32] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations. *URL: http://cusplibrary. github. io/(accessed: 01.02. 2016)*, 2014.

[33] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. *Version 0.5. 0*, 2014.

[34] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 349–359. IEEE, 2014.

[35] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[36] Jack Dongarra. Report on the sunway taihulight system. *PDF). www. netlib. org. Retrieved June*, 20, 2016.

[37] Ian J. Egielski, Jesse Huang, and Eddy Z. Zhang. Massive atomics for massive parallelism on GPUs. In *Proceedings of the 2014 International Symposium on Memory Management*, pages 93–103. ACM, 2014.

[38] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high-performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.

[39] Joseph L Greathouse and Mayank Daga. Efficient sparse matrix-vector multiplication on gpus using the csr storage format. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 769–780. IEEE Press, 2014.

[40] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75. ACM, 2014.

[41] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 24–31. ACM, 2013.

[42] Gaël Guennebaud, Benoıt Jacob, Philip Avery, Abraham Bachrach, Sebastien Barthelemy, et al. Eigen v3, 2010.

[43] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[44] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[45] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 233–245. IEEE, 2017.

[46] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320. ACM, 2012.

[47] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J Ramanujam, and Ponnuswamy Sadayappan. Effective padding of multidimensional arrays to avoid cache conflict misses. In *ACM SIGPLAN Notices*, volume 51, pages 129–144. ACM, 2016.

[48] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '18, pages 66–79, New York, NY, USA, 2018. ACM.

[49] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P Sadayappan. Gpu code optimization using abstract kernel emulation and sensitivity analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 736–751. ACM, 2018.

[50] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noel Pouchet, Fabrice Rastello, and P. Sadayappan. GPU code optimization using abstract kernel emulation and sensitivity analysis. Technical report, Ohio State University, 2018.

[51] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, Prashant Singh Rawat, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P Sadayappan. Performance modeling for gpus using abstract kernel emulation. *ACM SIGPLAN Notices*, 53(1):397–398, 2018.

[52] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. Multigraph: Efficient graph processing on gpus. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 27–40. IEEE, 2017.

[53] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314. ACM, 2019.

[54] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 349–362. ACM, 2012.

[55] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 267–276. ACM, 2011.

[56] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[57] Yingchao Huang and Dong Li. Performance modeling for optimal data placement on gpu with heterogeneous memory systems. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 166–177. IEEE, 2017.

[58] Haipeng Jia, Yunquan Zhang, Guoping Long, Jianliang Xu, Shengen Yan, and Yan Li. Gpuroofline: a model for guiding performance optimizations on gpus. In *European Conference on Parallel Processing*, pages 920–932. Springer, 2012.

[59] Wei Jiang and Gang Wu. A thick-restarted block arnoldi algorithm with modified ritz vectors for large eigenproblems. *Computers & Mathematics with Applications*, 60(3):873–889, 2010.

[60] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance. In *ACM SIGPLAN Notices*, volume 48, pages 395–406. ACM, 2013.

[61] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[62] Christoph W Keßler and Craig H Smith. The sparamat approach to automatic comprehension of sparse matrix computations. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pages 200–207. IEEE, 1999.

[63] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable SIMD-efficient graph processing on GPUs. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50. IEEE Computer Society, 2015.

[64] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pages 239–252. ACM, 2014.

[65] Jinsung Kim, Aravind Sukumaran-Rajam, Changwan Hong, Ajay Panyala, Rohit Kumar Srivastava, Sriram Krishnamoorthy, and P Sadayappan. Optimizing tensor contractions in ccsd (t) for efficient execution on gpus. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 96–106. ACM, 2018.

[66] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[67] Andrew V Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing*, 23(2):517–541, 2001.

[68] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[69] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Csx: An extended compression format for spmv on shared memory systems. *SIGPLAN Not.*, 46(8):247–256, February 2011.

[70] Sireyya Emre Kurt, Vineeth Thumma, Changwan Hong, Aravind Sukumaran-Rajam, and P Sadayappan. Characterization of data movement requirements for sparse matrix computations on gpus. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 283–293. IEEE, 2017.

[71] Junjie Lai and André Seznec. Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10. IEEE, 2013.

[72] Kartik Lakhotia, Shreyas Singapura, Rajgopal Kannan, and Viktor Prasanna. Recall: Reordered cache aware locality based graph processing. In *High Performance*

Computing (HiPC), 2017 IEEE 24th International Conference on, pages 273–282. IEEE, 2017.

[73] Seyong Lee, Jeremy S. Meredith, and Jeffrey S. Vetter. Compass: A framework for automated performance modeling and prediction. In *ACM International Conference on Supercomputing (ICS15)*, 2015.

[74] Seyong Lee and Jeffrey S Vetter. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *HPDC '14: Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, Short Paper*, 2014.

[75] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. Cawa: coordinated warp scheduling and cache prioritization for critical warp acceleration of gpgpu workloads. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 515–527. ACM, 2015.

[76] Weifeng Liu and Brian Vinter. CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication. *CoRR*, abs/1503.05032, 2015.

[77] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350. ACM, 2015.

[78] Weifeng Liu and Brian Vinter. A framework for general sparse matrix–matrix multiplication on gpus and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85:47–61, 2015.

[79] Yongchao Liu and Bertil 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*, pages 82–89. IEEE, 2015.

[80] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[81] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, and Karol Kowalski. Acceleration of streamed tensor contraction expressions on gpgpu-based clusters. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing, Heraklion, Crete, Greece, 20-24 September, 2010*, pages 207–216, 2010.

[82] Wenjing Ma, Sriram Krishnamoorthy, Oreste Villa, Karol Kowalski, and Gagan Agrawal. Optimizing tensor contraction expressions for hybrid cpu-gpu execution. *Cluster computing*, 16(1):131–155, 2013.

[83] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 455–466. ACM, 2014.

[84] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 455–466. ACM, 2014.

[85] Alberto Magni, Christophe Dubach, and Michael FP O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 11. ACM, 2013.

[86] Alberto Magni, Christophe Dubach, and Michael FP O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 11. ACM, 2013.

[87] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010.

[88] Karl Meerbergen and Raf Vandebril. A reflection on the implicitly restarted arnoldi method for computing eigenvalues near a vertical line. *Linear Algebra and its Applications*, 436(8):2828–2844, 2012.

[89] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[90] Duane Merrill and Michael 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*, pages 43:1–43:2. ACM, 2016.

[91] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[92] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. *HiPEAC*, 5952:111–125, 2010.

[93] Ronald B Morgan and Dywayne A Nicely. Restarting the nonsymmetric lanczos algorithm for eigenvalues and linear equations including multiple right-hand sides. *SIAM Journal on Scientific Computing*, 33(5):3037–3056, 2011.

[94] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Microarchitecture (MICRO), 2011 44th Annual IEEE/ACM International Symposium on*, pages 308–317. IEEE, 2011.

[95] Nervana maxas.

[96] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P Sadayappan. Sampled dense matrix multiplication for high-performance machine learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 32–41. IEEE, 2018.

[97] CUDA NVIDIA. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31, 2008.

[98] CUDA Binary Utilities, 2018.

[99] NWChem Download, 2017.

[100] Leonid Oliker, Xiaoye Li, Parry Husbands, and Rupak Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *Siam Review*, 44(3):373–393, 2002.

[101] Gloria Ortega, Francisco Vázquez, Inmaculada García, and Ester M Garzón. Fast-spmm: An efficient library for sparse matrix matrix product on gpus. *The Computer Journal*, 57(7):968–979, 2013.

[102] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19. ACM, 2016.

[103] Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Henry Wong. Micro-benchmarking the gt200 gpu. *Computer Group, ECE, University of Toronto, Tech. Rep*, 2009.

[104] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. Faster CNNs with Direct Sparse Convolutions and Guided Pruning, 2016. arXiv:1608.01409v5.

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

[106] Juan C Pichel, Francisco F Rivera, Marcos Fernández, and Aurelio Rodríguez. Optimization of sparse matrix–vector multiplication using reordering techniques on gpus. *Microprocessors and Microsystems*, 36(2):65–77, 2012.

[107] Xavier Pinel and Marc Montagnac. Block krylov methods to solve adjoint problems in aerodynamic design optimization. *AIAA journal*, 2013.

[108] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 12–25. ACM, 2011.

[109] Mahesh Ravishankar, Paulius Micikevicius, and Vinod Grover. Fusing convolution kernels through tiling. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 43–48. ACM, 2015.

[110] Prashant Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. Resource conscious reuse-driven tiling for GPUs. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 99–111, 2016.

[111] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, and P Sadayappan. Effective resource management for enhancing performance of 2d and 3d stencils on gpus. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 92–102. ACM, 2016.

[112] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[113] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Regularizing graph centrality computations. *Journal of Parallel and Distributed Computing*, 76:106–119, 2015.

[114] Isaiah Shavitt and Rodney J Bartlett. *Many-body methods in chemistry and physics: MBPT and coupled-cluster theory*. Cambridge university press, 2009.

[115] Xuanhua Shi, Junling Liang, Xuan Luo, Sheng Di, Bingsheng He, Lu Lu, and Hai Jin. Frog: Asynchronous graph processing on gpu with hybrid coloring model. *Huazhong University of Science and Technology, Tech. Rep. HUST-CGCL-TR-402*, 2015.

[116] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *ACM SIGPLAN Notices*, volume 47, pages 11–22. ACM, 2012.

[117] Markus Steinberger, Andreas Derlery, Rhaleb Zayer, and Hans-Peter Seidel. How naive is naive spmv on the gpu? In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–8. IEEE, 2016.

[118] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the gpu. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 13:1–13:11, New York, NY, USA, 2017. ACM.

[119] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the gpu. In *Proceedings of the International Conference on Supercomputing*, page 13. ACM, 2017.

[120] Bor-Yiing Su and Kurt Keutzer. clspmv: A cross-platform opencl spmv framework on gpus. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 353–364. ACM, 2012.

[121] Xiangzheng Sun, Yunquan Zhang, Ting Wang, Xianyi Zhang, Liang Yuan, and Li Rao. Optimizing spmv for diagonal sparse matrices on gpu. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 492–501. IEEE, 2011.

[122] Narayanan Sundaram and Kurt Keutzer. Long term video segmentation through pixel level spectral clustering on gpus. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 475–482. IEEE, 2011.

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

[124] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "Think like a vertex" to "Think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, November 2013.

[125] Michalis K Titsias. The infinite gamma-poisson feature model. In *Advances in Neural Information Processing Systems*, pages 1513–1520, 2008.

[126] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. Automatic restructuring of gpu kernels for exploiting inter-thread data locality. In *International Conference on Compiler Construction*, pages 21–40. Springer, 2012.

[127] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, et al. Nwchem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[128] Francisco Vázquez, José-Jesús Fernández, and Ester M Garzón. A new approach for sparse matrix vector product on nvidia gpus. *Concurrency and Computation: Practice and Experience*, 23(8):815–826, 2011.

[129] Francisco Vazquez, G Ortega, José-Jesús Fernández, and Ester M Garzón. Improving the performance of the sparse matrix vector product with gpus. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1146–1151. IEEE, 2010.

[130] Jyothi Vedurada, Arjun Suresh, Aravind Sukumaran Rajam, Jinsung Kim, Changwan Hong, Ajay Panyala, Sriram Krishnamoorthy, V Krishna Nandivada, Rohit Kumar Srivastava, and P Sadayappan. Ttlg-an efficient tensor transposition library for gpus. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 578–588. IEEE, 2018.

[131] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[132] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[133] F V'zquez, G Ortega, JJ Fern'ndez, Inmaculada García, and Ester M Garzón. Fast sparse matrix matrix product based on ellr-t and gpu computing. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 669–674. IEEE, 2012.

[134] Joerg Walter, Mathias Koch, et al. ublas. *Boost C++ software library available from http://www. boost. org/doc/libs*, 2006.

[135] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi¢*, pages 167–188. Springer, 2014.

[136] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. *SIGPLAN Not.*, 51(8):11:1–11:12, February 2016.

[137] Mark N Wegman and F Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[138] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.

[139] NVIDIA Tesla K100, 2012.

[140] NVIDIA Tesla P100, 2016.

[141] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc-first experiences with real-world applications. *Euro-Par 2012 Parallel Processing*, pages 859–870, 2012.

[142] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.

[143] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[144] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu. Efficient pagerank and spmv computation on amd gpus. In *2010 39th International Conference on Parallel Processing*, pages 81–89, Sept 2010.

[145] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. Cvr: efficient vectorization of spmv on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 149–162. ACM, 2018.

[146] Shizhen Xu, Yuanchao Xu, Wei Xue, Xipeng Shen, Xiaomeng Huang, and Guangwen Yang. Taming the "monster": Overcoming program optimization challenges on

sw26010 through precise performance modeling. In *Parallel and Distributed Processing Symposium (IPDPS), 2018 IEEE International*, page pages will be added. IEEE, 2018.

[147] Ichitaro Yamazaki, Tingxing Dong, Raffaele Solcà, Stanimire Tomov, Jack Dongarra, and Thomas Schulthess. Tridiagonalization of a dense symmetric matrix on multiple gpus and its application to symmetric eigenvalue problems. *Concurrency and computation: Practice and Experience*, 26(16):2652–2666, 2014.

[148] Ichitaro Yamazaki, Hiroto Tadano, Tetsuya Sakurai, and Tsutomu Ikegami. Performance comparison of parallel eigensolvers based on a contour integral method and a lanczos method. *Parallel Computing*, 39(6):280–290, 2013.

[149] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaspmv: Yet another spmv framework on gpus. In *Acm Sigplan Notices*, volume 49, pages 107–118. ACM, 2014.

[150] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaspmv: yet another spmv framework on gpus. In *Acm Sigplan Notices*, volume 49, pages 107–118. ACM, 2014.

[151] Carl Yang, Aydin Buluc, and John D Owens. Design principles for sparse matrix multiplication on the gpu. *arXiv preprint arXiv:1803.08601*, 2018.

[152] Hiroki Yoshizawa and Daisuke Takahashi. Automatic tuning of sparse matrix-vector multiplication for crs format on gpus. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pages 130–136. IEEE, 2012.

[153] AN Yzelman and Rob H Bisseling. Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009.

[154] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–43. ACM, 2017.

[155] Yao Zhang and John D Owens. A quantitative performance analysis model for gpu architectures. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 382–393. IEEE, 2011.

[156] Huasha Zhao, Biye Jiang, John F Canny, and Bobby Jaros. Same but different: Fast and high quality gibbs parameter estimation. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1495–1502. ACM, 2015.

[157] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.

[158] Keren Zhou, Guangming Tan, Xiuxia Zhang, Chaowei Wang, and Ninghui Sun. A performance analysis framework for exploiting gpu microarchitectural capability. In *Proceedings of the International Conference on Supercomputing*, page 15. ACM, 2017.