A Study on Reducing the Amount of Out-of-Core Data Access for GPU-Accelerated Applications

Submitted to
Graduate School of Information Science and Technology
Osaka University

April 2020

Yuechao Lu

Published Papers

Journal Papers

1. Yuechao Lu, Fumihiko Ino, and Kenichi Haighara, "Cache-Aware GPU Optimization for Out-of-Core Cone Beam CT Reconstruction of High-Resolution Volume," *IEICE Transactions on Information and Systems*, Vol.E99-D, No.12, pp.3060–3071, Dec. 2016.

2. Yuechao Lu, Ichitaro Yamazaki, Fumihiko Ino, Yasuyuki Matsushita, Stanimire Tomov, and Jack Dongarra, "Reducing Out-of-Core Data Access for GPU-accelerated Randomized SVD," *Concurrency and Computation: Practice and Experience*, accepted.

3. Yuechao Lu, Yasuyuki Matsushita, and Fumihiko Ino, "Block Randomized SVD on GPUs," *IEICE Transactions on Information and System*, accepted.

International Conference Papers

1. Yuechao Lu, Boqi Gao, and Fumihiko Ino, "GPU-Accelerated Randomized Newton Method for Fast Training Malicious Web Site Classifier," *Proceedings of the 18th International Conference on Security and Management (SAM)*, pp.95–99, Aug. 2019.

Oral Presentations

1. Yuechao Lu, Fumihiko Ino, Yasuyuki Matsushita, and Kenichi Hagihara, "RLAGPU: High-performance Out-of-core Randomized Singular Value Decomposition on GPU," *Poster in the 8th GPU Technology Conference (GTC)*, May 2017.

Abstract

The computing architectures are shifting quickly as we move forward to exascale computing. Heterogeneous computing architectures employing accelerators like graphics processing unit (GPU) have become the mainstream approach for high-performance computing (HPC) systems from supercomputers to mobile devices. However, the increased layers of the memory hierarchy in heterogeneous architectures have hindered the users to extract the full potential and maximum scalability of the hardware. Communication cost for transferring data between memory hierarchy dominates the overall processing time for most applications and the floating-point operations per second (flop/s) has become comparatively irrelevant.

In this work, we look into two applications to address the problems in accelerating applications with GPUs. The first application is the cone beam computed tomography (CT) reconstruction, which is an widely used by medical imaging devices. The second application is a matrix decomposition algorithm called randomized singular value decomposition (RSVD). We show that the CPU-GPU data transfer is the main bottleneck for processing large scale data on GPU-enabled systems. We first propose methods to accelerate RSVD by reducing the data transfer between the CPU and GPU. We then propose algorithms which modify the original RSVD to fit into the heterogeneous computing architecture. The proposed methods successfully move the performance bottleneck from CPU-GPU bandwidth bound to compute bound, so that the computation ability of the GPUs can be fully utilized for acceleration.

This thesis is divided into three parts. In the first part of this work, we propose a cache-aware optimization method to accelerate the out-of-core cone beam CT reconstruction on a GPU. Out-of-core data here are data that are too large to fit into the GPU memory at once. Utilizing the GPU in reconstructing CT images has gained its popularity for its high performance and low cost implementation compared to other methods. The proposed method extends a GPU-based previous method by increasing the cache hit rate to speed up the reconstruction of high-resolution volumes that exceed the capacity of GPU memory. More specifically, our approach accelerates the well-known Feldkamp, Davis, and Kress (FDK) algorithm by utilizing the following three strategies: (1) a loop organization strategy that identifies the best trade-off point between the cache hit rate and the number of off-chip memory accesses; (2) a data structure that exploits high locality within a layered texture; and (3) a fully pipelined strategy for hiding file input/output (I/O) time of GPU execution and data transfer time. We implement our proposed method on NVIDIA's Maxwell architecture and provide a tuning guideline for adjusting the execution parameters, which include the granularity and shape of thread blocks as well as the granularity of I/O data to be streamed through the pipeline, which maximizes reconstruction performance. Our experimental results

show that it took less than three minutes to reconstruct a 2048^3-voxel volume from 1200 2048^2-pixel projection images on a single GPU; this translates to a speedup of approximately 1.47 as compared to a previous method. We also make clear a trade-off between the texture cache hit rate and the number of memory accesses. Concerning GPU optimization, we found that it is not necessarily efficient to compact as many tasks as possible into kernel execution to decrease total execution time. Instead, proper tuning is required to identify the optimum number of tasks that will minimize the overall time. With the aid of texture interpolation and cache-aware strategies, our presented GPU implementation achieves performance advantages over other computing platforms.

In the second part, we propose two acceleration methods, namely Fused and Gram, for reducing the out-of-core data access when performing RSVD on GPUs. Both methods accelerate GPU-enabled RSVD using the following three schemes: (1) a highly tuned general matrix-matrix multiplication (GEMM) scheme for processing out-of-core data on GPUs; (2) a data-access reduction scheme based on one-dimensional (1D) data partition; and (3) a first-in, first-out (FIFO) scheme that reduces CPU-GPU data transfer using a reverse iteration. The Fused method further reduces the amount of out-of-core data access by merging two GEMM operations into a single operation. In contrast, the Gram method reduces both in-core (*i.e.*, all the working data can be held on the GPU memory) and out-of-core data access by explicitly forming the Gram matrix. According to our experimental results, the Fused and Gram methods improved the RSVD performance by up to 1.9× and 5.2×, respectively, compared with a straightforward method that deploys schemes (1) and (2) on the GPU. In addition, we present a case study of deploying the Gram method for accelerating robust principal component analysis (RPCA), a convex optimization problem in machine learning.

In the third part, we propose a two-pass RSVD, named block randomized SVD (BRSVD), designed for matrices with a slow-decay singular spectrum that is often observed in image data. BRSVD fully utilizes the power of modern computing system architectures and efficiently processes large-scale data in a parallel and batched fashion. Our experiments show that BRSVD effectively moves the performance bottleneck from data transfer to computation, so that outperforms existing RSVD methods in terms of speed with retaining similar accuracy. We also show an application of randomized SVD to convex RPCA on a GPU, which shows significant speedup in computer vision applications.

Our work demonstrates that communication cost is an important factor to influence the overall performance. The first application demonstrates that cache-aware optimization improves overall performance effectively. The second application shows that reducing the communication cost at the expense of increased computational cost is a viable approach in a computing environment where communication cost exceeds the computational cost. Furthermore, our work shows that redesigning algorithms to fit for the heterogeneous computing architecture is a feasible approach in dealing with bandwidth bound problems.

Acknowledgements

I would like to express my deep and sincere gratitude to my supervisor, Professor Fumihiko Ino, for his many valuable and insightful advices and for his continuous encouragement during my years.

I would like to sincerely thank Professor Yasuyuki Matsushita, who has guided this work with many constructive suggestions and detailed help on my thesis.

I would like to thank the member of my thesis committee: Professor Toshimitsu Masuzawa, Professor Yasuyuki Matsushita, and Professor Fumihiko Ino for their insightful comments and encouragement.

I am deeply grateful to Honorary Professor Kenichi Hagihara and Associate Professor Masao Okita for discussions, for their many valuable and thoughtful comments, and for their encouragement.

I would like to thank my collaborators: Professor Jack Dongarra, Dr. Ichitaro Yamazaki, and Professor Stanimire Tomov (affiliated with the innovative computing laboratory, University of Tennessee). They gave me a lot of motivating research ideas and guided me through the difficulties during the research.

I acknowledge the financial support from Humanware Innovation Program at Osaka University in the last five years.

Finally, I am indebted to members of Ino laboratory, for their daily support and useful comments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we first brief the history of high-performance computing (HPC). We then describe the recent heterogeneous computing architecture with GPU and the bottleneck for accelerating applications on this architecture. After that, We introduce the two topics of this work. Finally, we summarize the contributions of this thesis.

1.1 Overview of High-Performance Computing

In recent years, HPC has gradually entered almost all aspects of science and engineering. HPC enables scientists and engineers to construct and validate large simulation models like molecular simulation and plasma fusion simulation. Supercomputers now are regarded as an essential tool in driving new exploration and discovery. In 1997, the first supercomputer reached Tflop/s was called ASCI Red and was built by Sandia National Lab [21]. In 2008, a supercomputer called RoadRunner at Los Alamos National Lab first reached Pflop/s [8].

As we move forward to exascale computing [19], the supercomputing architectures are rapidly shifting. The traditional customized CPUs, networking, and storage systems for supercomputers are diminishing. Three architectures are emerging: the first is that the trend of building supercomputers with commodity CPUs like Intel Xeon. Those machines have dominated the Top 500 list[1]. The second is that systems with accelerators are increasing. Over one hundred systems out of Top 500 adopt accelerators like GPUs or field-programmable gate array (FPGA). This kind of heterogeneous architectures is expected to become mainstream soon. The third is that lightweight CPUs like advanced RISC machine (ARM) have entered the realm of supercomputing. Lightweight CPUs have much lower power consumption and heat dissipation than traditional CPUs. This kind of supercomputer fits needs like building supercomputers with a limited power supply.

Regarding the benchmark software for supercomputers, LINPACK [20] has been the de facto tool for more than 30 years. Solving a dense system of linear equation $\mathbf{Ax} = \mathbf{b}$ is used as the benchmark in LINPACK. The idea is that making matrix $\mathbf{A}$ as large as possible, then solving the equation using Gaussian elimination with partial pivoting. By measuring the performance with a maximum size of $\mathbf{A}$, an actual performance peak will be obtained.

[1] `top500.org`

Combined with the theoretical peak performance, those numbers are the benchmark values for supercomputers on the Top 500 list.

1.2 Overview of Graphics Processing Unit (GPU)

NVIDIA, AMD, and ARM are three major vendors for commodity GPUs. The former two provide their GPUs mainly for desktops and workstations, while ARM only provides GPUs for mobile devices with low power consumption. All of their products support the programming framework OpenCL [66]. In this work, we focus on NVIDIA GPUs and their programming framework called CUDA [77]. The reason is that NVIDIA GPUs yield the best performance per unit and dominate the accelerator market in HPC. The methods proposed in the work are general and can be extended to GPUs provided by other vendors.

1.2.1 Architectural Difference between CPU and GPU

The major difference between CPU and GPU is that CPU is latency-focused or serial-focused, while GPU is throughput-focused or parallel-focused. Modern CPUs have powerful arithmetic logic units (ALUs), which include complex circuits for functions like instruction reordering, branch prediction, out-of-order execution, paging, and caching. Almost all those functions aim at reducing the latency of program execution.

On the other hand, GPUs use the opposite strategy. The ALUs on GPUs have much fewer functions than those on CPUs. Those simple ALUs are called CUDA cores by NVIDIA. A typical GPU is compacted with thousands of CUDA cores. GPU hides the latency behind high throughput provided by CUDA cores and complex memory systems. CUDA cores are grouped into an array of streaming multiprocessors (SMs). To compensate for the CUDA cores, each SM is also equipped with ALUs like special function units (SFUs), double precision units, and warp schedulers. CUDA cores are in charge of processing 32-bit integer and floating-point operations. SFUs are in charge of special arithmetic operations like reciprocation. Double-precision units are in charge of double-precision operations. The number of double-precision units in a single SM varies according to the target market of a certain GPU. In the CUDA framework so far, 32 threads are packed into a single unit called *warp* for managing and scheduling execution. A hardware mechanism called warp scheduler manages the dispatch of warps to CUDA cores.

In general, using implementations based on CUDA offloads the performance bottleneck of a CPU-based sequential code. Such offloaded workloads can be implemented as *kernel functions*, which can be parallelized via millions of GPU threads for acceleration.

1.2.2 GPU Computing Model

Single instruction multiple data (SIMD) is the most widely adopted parallel computing model. SIMD generally means that all processors execute the same instruction while the data can be different on each processor. The constraint is that SIMD uses vector instructions (*e.g.* AVX for Intel CPU). Vector instructions perform the same operation on multiple data elements. However, data must be loaded and stored contiguously in vector instructions.

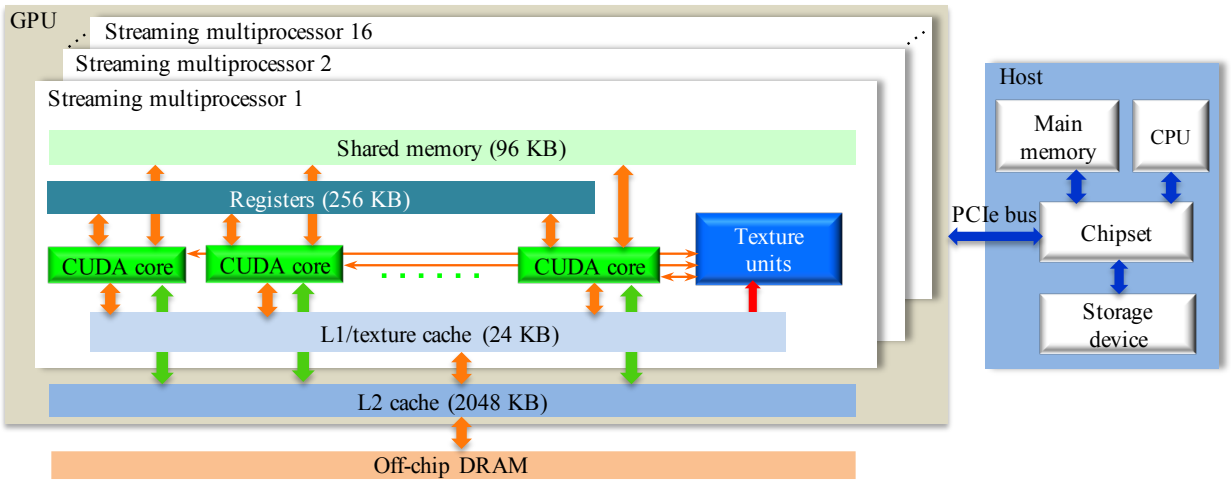

Figure 1.1: Maxwell GPU architecture [71].

NVIDIA improved the SIMD computing model and proposed single instruction multiple threads (SIMT) model for its GPU computing. SIMT relaxes the constraint of SIMD. SIMT allows threads to take different execution paths in the unit of warp.

We use the the Maxwell GPU [71] (Fig. 1.1) for the explanation of the GPU computing model. Similar to other CUDA-compatible GPUs, this architecture has an array of SMs to process millions of tasks in parallel. Regarding the memory hierarchy inside the SM, there are registers, a shared memory, and L1 cache. Registers provide the shortest access latency, analogous to CPU registers, but different in number; here there are 64K 32-bit registers. The shared memory is a software-managed cache that allows CUDA cores to more efficiently share data inside an SM so that they do not need to share through the slow off-chip memory. Finally, the L1 cache acts as a read-only cache and coalescing buffer for off-chip memory access [72]. L1 cache can be accessed by all threads assigned to the current SM. Outside the SM, there are L2 cache, memory controller, texture caches, and translation lookaside buffers. L2 cache and memory controller together controls memory request so that threads that access the same piece of memory do not need to access slow off-chip memory.

Outside of the SMs, there is an off-chip memory called GPU memory. Although GPU memory provides a large storage of up to 32 GB, its latency of several hundreds of clock cycles is much longer than that of the on-chip memory. GPU memory can be used as both texture and global memory. Here, texture memory stores read-only data that can be accessed using hardware interpolation, while global memory stores readable/writable data for CUDA cores. As shown in the figure, the L2 cache is an on-chip cache located between the SMs and global memory.

During kernel execution, threads are cyclically assigned to SMs in the unit of a thread block [77], *i.e.*, a group of threads organized by CUDA programmers. Different thread blocks must be independent of each other with respect to data dependencies; otherwise parallelization cannot be correctly achieved. Resident thread blocks that have been assigned consume SM resources, including registers and shared memory, such that there are limitations on the maximum number of resident threads and that of thread blocks. This assignment

process is repeated until all thread blocks finish execution.

Threads in a warp, which share the same instruction at each clock cycle, are processed on an SM in parallel. While threads in the same warp access global memory, memory access coalescence is critical for maximizing effective memory bandwidth [77]. In most cases, assigning multiple thread blocks to an SM is an effective approach for overlapping memory accesses with computation, because the SM has more data-independent resident warps to switch while waiting for data to be fetched from off-chip memory. Thus, if we maximize the occupancy, or ratio of the number of resident thread blocks to the maximum number of resident thread blocks, we can hide memory access latencies with computation. In other words, fewer resident thread blocks expose memory latency.

Each thread can also independently access memory in the SIMT model, which means the data can be loaded and stored non-contiguously. Although each thread is free to branch and execute independently, the threads can be stalled for reasons like data fetching latency. As we previously mentioned, the memory access has a tremendous impact on program performance. If the memory access requests from a warp have stride larger than the memory segment limit (*e.g.* 128 bytes in recent GPU memory system), those requests need to be severed by several memory transactions which degrade the performance. The best case is that the memory access stride is small enough to be severed with only one transaction, which is called memory coalescing. Despite memory coalescing, a lot of other factors like cache hit rate and shared memory bank conflicts also influence the performance significantly.

1.3 Challenges in Heterogeneous Computing Architectures

There are a few major challenges that hinder applications to fully benefit from emerging new computing architectures. First, more layers of the memory hierarchy in new architecture complicate programming, migration, and optimization. Heterogeneous architectures make the situation even worse. Tuning the granularity of tasks, the locality of data, coordination, and synchronization for those system imposes a tremendous burden on researchers.

Second, communication across distinct memory hierarchies or networks often constitutes a performance bottleneck [18, 37] due to the increasing gap between arithmetic and communication performance [15]. Supercomputers now have more than millions of computing cores. The spent time for communication between those cores usually dominates the overall processing time for large scale data. The methodology for algorithm design has shifted from reducing the computational cost to reducing the communication cost [15, 30, 113].

Figure 1.2 illustrates the memory hierarchy in a heterogeneous computing architecture with GPUs. The capacity of each layer in Fig. 1.2 is normally more than 10× larger than its upper layer. However, the access speed will also decrease to less than 1/10 for accessing the lower layer with a larger capacity.

In this work, the GPU is regarded as the computing core, and the data is stored out-of GPU memory and brought into GPU memory to be processed. Out-of-core data access here involves CPU-GPU data transfer in our target CPU-GPU system, where by contrast, in-core data can be rapidly accessed without additional CPU-GPU data transfer. Currently,

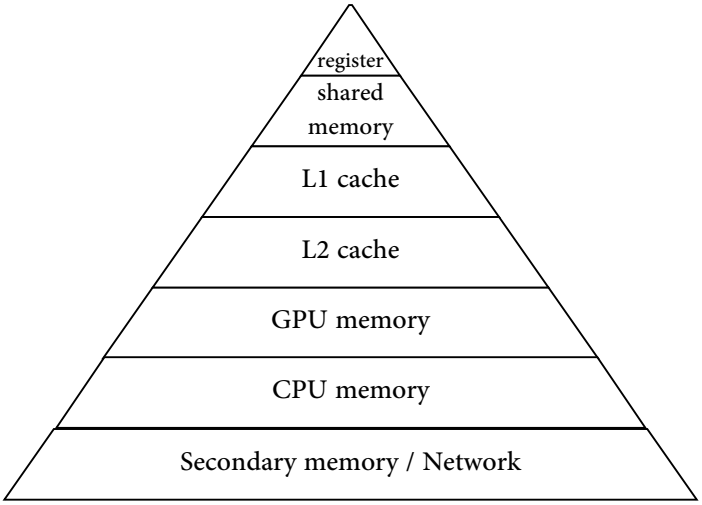

Figure 1.2: Illustration of the memory hierarchy in a heterogeneous computing architecture with GPUs.

the in-core data access speed for GPU has reached 900 GB/s (V100 GPU). Comparatively, the PCIe 3.0 interface for CPU-GPU data transfer has a theoretical maximum bandwidth of 15.75 GB/s. This large performance gap renders the out-of-core data transfer to be the bottleneck of accelerating applications with a large data set.

1.4 Contributions of This Thesis

The contributions of this thesis are divided into three parts:

In the first part, we look into an application called cone beam CT reconstruction. We analyze the bottleneck of accelerating cone beam CT reconstruction on the GPU. We propose a cache-aware optimization method to maximize the usage of the cache mechanism on GPUs.

In the second part, we investigate a randomized matrix decomposition algorithm called randomized singular value decomposition (RSVD). We focus on accelerating RSVD on GPUs. We use a roofline model [111] to show that the theoretical peak performance is bounded by CPU-GPU data transfer. We then propose out-of-core methods that reduce data transfer especially for tall-skinny matrices. Our methods do not modify the original algorithm.

In the third part, we propose an algorithm which is called Block Randomized Singular Value Decomposition (BRSVD). BRSVD is based on the original RSVD algorithm and is modified to fit for the heterogeneous computing environment with GPUs. The out-of-core methods proposed in the second part loose the constraint of CPU-GPU bandwidth, but the bandwidth remains the performance bottleneck. BRSVD successfully moves the performance bottleneck from CPU-GPU bandwidth bound to performance bound at the expense of slightly reducing the accuracy of the original algorithm.

Chapter 2

Cache-Aware GPU Optimization for Out-of-Core Cone Beam CT Reconstruction of High-Resolution Volumes

2.1 Introduction

CT reconstruction is a radiology imaging technology that converts two-dimensional X-ray projection images generated by a rotary CT scanner into a three-dimensional (3D) volume, such that CT data can be viewed using 3D visualization software. The FDK algorithm [26] is the de facto standard for cone beam CT reconstruction and is widely adopted in medical and industrial applications [24, 53, 62]. Because reconstruction time is critical, especially for real-time medical applications such as image-guided surgery [90, 108], research activities on accelerating the FDK algorithm have been ongoing ever since its advent in 1984.

With the development of parallel computing and computer graphics technologies, efforts to parallelize the FDK algorithm have included various computing devices, including a GPU [112, 117], a cell broadband engine (CBE) [52], a field-programmable gate array (FPGA) [29] and a Xeon Phi coprocessor [43]. CBE has eight synergistic processing elements (SPEs) allow for a theoretical performance of 192 Gflop/s. Data mining techniques and double buffering of input data were extensively used to optimally utilize both the memory bandwidth and the available local store of each SPE. The pixel-driven back-projection code uses floating point arithmetic and either inear interpolation or nearest neighbor interpolation between neighboring detector channels. For FPGA implementation, Gac *et al.* proposed a prediction algorithm to prefetch the data needed into cache so as to increase the spatial and temporal localities [29].

In particular, utilizing CUDA, compatible with GPUs [77], to parallelize FDK computation has gained popularity due to its high performance and low-cost implementation as compared to other devices [24, 46, 91]. Given this parallelization technique, the performance bottleneck of the FDK algorithm lies in its back-projection of projection images in which interpolated pixel values are accumulated back to form voxel values, which then compose

the 3D volume. Therefore, typical implementations store projection images in *textures* to take advantage of hardware-accelerated interpolation available on the GPU.

In addition to this fundamental implementation scheme, Okitsu *et al.* [82] presented a multiplication method that back-projects multiple projections with a single kernel invocation. This multiplication method accelerates the back-projection procedure by reducing the number of off-chip memory accesses. In [82], Okitsu *et al.* concluded that GPU memory bandwidth (*i.e.*, the memory bandwidth between SMs [77] and off-chip memory) determines reconstruction throughput on a GeForce 8800 GTX GPU. The FDK has long been proved to be a memory-bound application on parallel computers. A performance model [45] is normally used to indicate the implementation efficiency on a specific hardware. The model is given $E = 4\rho/B$, where B is the total memory bandwidth of the deployed machine, ρ is the back-projection throughput. Okitsu *et al.* emphasized memory coalescing to improve the overall performance, because the GPU memory bandwith was the bottleneck on the deployed G80 architecture [69] and the cache capacity was too small to impact on the overall performance. Therefore, in this chapter, we propose a cache-aware optimization method [61] to accelerate the FDK algorithm for handling out-of-core data on a GPU. To our knowledge, no work has been done in modeling the GPU cache performance. The reason is that the cache hit rate varies greatly with different applications. Also, the cache is transparent to the programmer, which means that the programmer has no direct control over the data movement. Regarding the general purpose modeling of the GPU memory, latency, and bandwidth, several works has been done. Those works do not take the cache into their models. For example, Nakano has proposed asynchronous memory machine model for GPU computing [67]. They applied the model in analyzing several applications like interval sum and prefix-summing. However, the proposed model is not cache-aware.

In this chapter, we extends Okitsu's method [82] by increasing the cache hit rate, thereby improving out-of-core cone beam CT reconstruction. The proposed method consists of the following three key strategies: (1) a loop organization strategy which identifies the best trade-off point between the cache hit rate and the number of off-chip memory accesses; (2) a data structure that exploits high locality within a layered texture; and (3) a fully pipelined strategy for hiding file I/O times with GPU execution and data transfer times. We analyze the underlying mechanism of these strategies and provide tuning guidelines for adjusting the execution parameters, which include the granularity and shape of thread blocks [77] and the granularity of I/O data to be streamed through the pipeline, the latter maximizing reconstruction performance on NVIDIA's latest Maxwell architecture [71].

The rest of this chapter is organized as follows. In Section 2.2 we introduce related studies regarding the acceleration of cone beam CT reconstruction. In Section 2.3 we summarize the FDK algorithm and its previous GPU-based implementation [82]. In Section 2.4, we describe our proposed method, and then present our experimental results in Section 2.5 along with discussion on tuning for the Maxwell architecture. Finally, in Section 2.6, we conclude our chapter and suggest avenues for future work.

2.2 Related Work

Table 2.1 shows a comparison of our present work with recent studies, showing the advantages of our proposed method. In the table, the back-projection throughput ρ is presented in giga voxel updates per second (GUPS), where giga denotes 10^9. In summary, our present work employs a single-GPU machine to achieve high back-projection throughput for large amounts of data that exceed not only GPU memory but also CPU memory. Further, we incorporate optimization method based on NVIDIA's Maxwell architecture [71].

To our knowledge, Scherl *et al.* [91] first proposed the use of a CUDA-based GPU to accelerate the FDK algorithm, claiming that reducing register file usage raised GPU occupancy [77] so as to accelerate reconstruction. They demonstrated that a GeForce 8800 GTX GPU achieved two times higher reconstruction performance as compared to a CBE. A similar CUDA-based approach with similar results was presented by Noël *et al.* [68].

As for kernel optimization, Okitsu *et al.* [82] extended Xu's method [112], who first proposed back-projecting multiple projections in a single kernel invocation. These multiplication schemes reduced the number of off-chip memory accesses and that of kernel invocations. They concluded that GPU memory bandwidth determines reconstruction performance; thus, the back-projection kernel should process more projections at a time. A similar scheme was presented by Papenhausen *et al.* [85], who processed 64 projections with a single kernel execution. In contrast to the above studies, we show that excessive projections result in a lower texture cache hit rate on the latest Maxwell architecture [71]. Consequently, it is important to find the best trade-off point between texture cache hit rate and the number of off-chip memory accesses.

Based on Okitsu's multiplication method [82], Zinßer *et al.* [123] swapped the nested loop structure proposed in [82] such that threads that share the same instruction can simultaneously access a single projection. Zinßer *et al.* claimed that their loop organization not only increased the texture cache hit rate but also reduced the number of off-chip memory accesses by processing 32 projections with a single kernel invocation. A key drawback of their loop organization is that it consumes more registers than the original organization. Consequently, only four xy-slices of the volume were produced by a kernel invocation, whereas the original organization produced 512 xy-slices at a time. This consumption issue must be resolved for large amounts of data, which we focus on in the present study, because 128 times more kernel invocations are required to produce the entire volume. In our work, we present a data structure capable of achieving an L1/texture cache hit rate of more than 95%, even with the original loop organization.

In contrast to the input-related optimization mentioned above, Zheng *et al.* [122] presented a cache-aware method capable of maximizing write throughput for the output volume. Their method rearranges volume data according to the back-projection angle such that a series of memory transactions can be coalesced into a single transaction. Since this data rearrangement incurs overhead, they allocated another copy of the volume to avoid rearrangement overhead; however, such duplicated data must be eliminated to handle large amounts of data on limited GPU memory. Our method realizes memory access coalescence by adopting a workload distribution scheme in which threads are responsible for angle-independent regions of the volume.

With respect to out-of-core reconstruction in which I/O data flow exceed GPU memory,

Table 2.1: Comparison of the performance of our work with relevant recent works. Note that back-projection throughput ρ which is measured in GUPS, is given by $\rho = NXYZ/T$, where N is the number of projections, $X \times Y \times Z$ is the volume size in voxels, and T is the back-projection time, which includes data transfer times between CPU and GPU. Further, U and V are the horizontal and vertical sizes of a projection, respectively. Note that efficient cache utilization achieves an efficiency of more than 100%.

Work	Platform	Memory specification		Data specification		Throughput	Efficiency
		Bandwidth (GB/s)	Capacity (GB)	$NUV \rightarrow XYZ$	Size (GB)	ρ (GUPS)	E (%)
Scherl (2007) [91]	8800 GTX	86.4	0.8	$414 \times 1024 \times 1024 \rightarrow 512^3$	1.6 → 0.5	6.2	29
Okitsu (2010) [82]	2×Tesla C870	2 × 76.8	2 × 1.5	$1024 \times 1024 \times 1024 \rightarrow 1024^3$	4.0 → 4.0	48.9	127
Ino (2010) [45]	4×Tesla S1070	4 × 102.0	4 × 4.0	$2048 \times 2048 \times 2048 \rightarrow 2048^3$	32.0 → 32.0	105.7	104
Noël (2010) [68]	GTX 280	141.7	1.0	$106 \times 1024 \times 1024 \rightarrow 512^3$	0.4 → 0.5	2.3	6
Zheng (2010) [122]	GTX 480	177.4	1.5	$364 \times 1024 \times 768 \rightarrow 512^3$	1.1 → 0.5	12.0	27
Zhang (2012) [120]	2×GTS 450	2 × 57.7	2 × 1.0	$360 \times 1024 \times 1024 \rightarrow 512^3$	1.4 → 0.5	11.1	38
Treibig (2013) [100]	4×Xeon E7-4870	4 × 34.2	N/A	$496 \times 1248 \times 960 \rightarrow 1024^3$	2.2 → 4.0	12.0	35
Papenhausen (2013) [85]	GTX 680	192.3	2.0	$496 \times 1248 \times 960 \rightarrow 512^3$	2.2 → 0.5	72.3	150
Zinßer (2013) [123]	GTX 680	192.3	2.0	$496 \times 1248 \times 960 \rightarrow 1024^3$	2.2 → 4.0	88.2	183
Blas (2014) [9]	2×GTX 680	2 × 192.3	2 × 2.0	$720 \times 1024 \times 1024 \rightarrow 1024^3$	2.8 → 4.0	117.5	122
Serrano (2014) [92]	2×GTX 680	2 × 192.3	2 × 2.0	$360 \times 512 \times 512 \rightarrow 512^3$	0.4 → 0.5	27.1	28
	2×Xeon Phi 7120P	2 × 352.0	2 × 16.0	$360 \times 512 \times 512 \rightarrow 512^3$	0.4 → 0.5	26.7	15
This work (2016)	GTX 980	224.0	4.0	$1200 \times 512 \times 512 \rightarrow 512^3$	1.2 → 0.5	116.7	208
				$1200 \times 1024 \times 1024 \rightarrow 1024^3$	4.7 → 4.0	128.5	229
				$1200 \times 2048 \times 2048 \rightarrow 2048^3$	18.8 → 32.0	92.9	166

several studies have explored a multi-GPU machine to achieve further acceleration [9, 45, 120]. Existing multi-GPU implementations adopt a pipelined approach to overlap kernel execution with data transfer between CPU and GPU; however, except for Blas *et al.* [9], file I/O overhead has not been considered in detail. Blas *et al.* [9] did indeed consider file I/O overhead, thus realizing on-the-fly reconstruction, which produces the volume immediately after image acquisition; however, cache optimization issues were not addressed. Our out-of-core pipelined strategy yields a fully pipelined cache-aware solution for processing large amounts of data on a single-GPU system. Further, although we evaluated the advantages of our method on a single-GPU machine, our method can be expanded to support a multi-GPU environment in a straightforward manner.

On the other hand, Serrano *et al.* [92] proposed using a directive-based programming approach [83] to parallelize the FDK algorithm on GPUs and Intel Xeon Phi coprocessors. Compared with CUDA, this directive-based approach provides an easy programming scheme in which parallelization is achieved by adding compiler directives to sequential code; however, using such a high-level programming style degrades the performance. Due to the same performance related reason, we prefer CUDA rather than OpenCL [56].

Finally, Treibig *et al.* [100] explored optimizing the AVX instruction set [47]. They indicated that GPU-based solutions degrade reconstruction throughput for large amounts of data, because limited GPU memory requires data transfers between CPU and GPU. Our pipelined solution overlaps these required data transfers with GPU computation, thereby achieving higher out-of-core reconstruction performance as compared to CPU-based approaches (Table 2.1).

2.3 Preliminaries

Let F be the 3D volume of size $X \times Y \times Z$ to be reconstructed. To estimate each voxel value $F(x, y, z)$ in F, where $0 \leq x < X$, $0 \leq y < Y$, and $0 \leq z < Z$, the FDK algorithm back-projects set $\mathcal{P}$ of two-dimensional projection images onto the target volume F, where

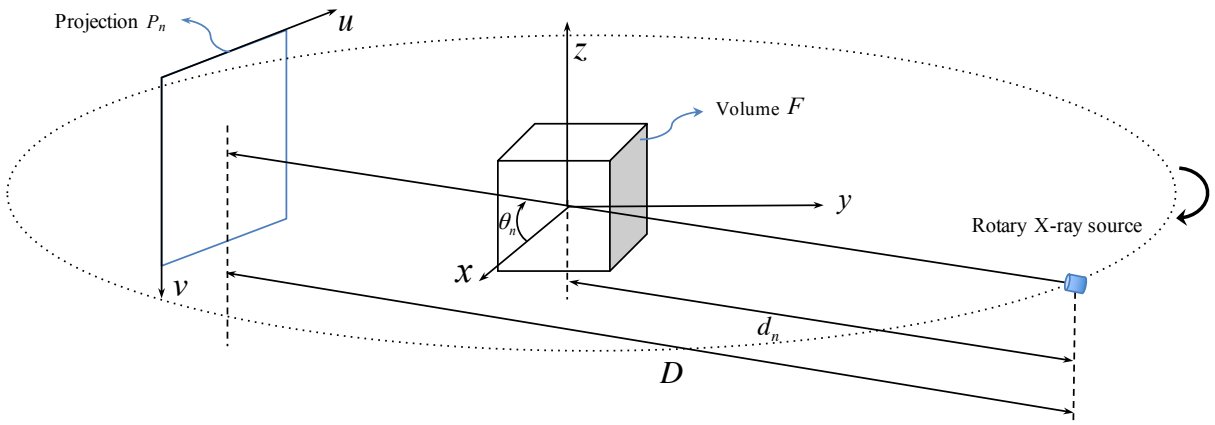

Figure 2.1: Geometry for back-projection of the n-th filtered projection, where $0 \leq n < N$.

$\mathcal{P} = \{P_0, P_1, \ldots, P_{N-1}\}$ and N is the number of projection images. As shown in Fig. 2.1, P_n represents a projection obtained with rotational angle θ_n, where $0 \leq n < N$. In the following subsections, we assume that all projection images comprise $U \times V$ pixels.

2.3.1 FDK Reconstruction Algorithm

The FDK algorithm is composed of two processing stages, *i.e.*, ramp filtering and back-projection. The ramp filtering stage performs one-dimensional convolution along the horizontal direction (*i.e.*, the u-axis). Given raw projection P_n, where $0 \leq n < N$, the pixel value $Q_n(u, v)$ of filtered projection Q_n, where $0 \leq u < U$ and $0 \leq v < V$, is given by

$$Q_n(u,v) = \sum_{r=-R}^{R} \frac{2}{\pi^2(1-4r^2)} \frac{D}{\sqrt{D^2+r^2+v^2}} P_n(r,v), \tag{2.1}$$

where R denotes the ramp filter radius and D denotes the distance between the X-ray source and the projection panel, as shown in Fig. 2.1.

Next, the back-projection stage back-projects filtered projections $Q_0, Q_1, \ldots, Q_{N-1}$ into 3D volume F. Voxel value $F(x, y, z)$ at point (x, y, z) is given by

$$F(x,y,z) = \frac{1}{2\pi N} \sum_{n=0}^{N-1} W(x,y,n)\ Q_n(u(x,y,n), v(x,y,z,n)), \tag{2.2}$$

where weight $W(x, y, n)$ and coordinates $u(x, y, n)$ and $v(x, y, z, n)$ are calculated separately by

$$W(x,y,n) = \left(\frac{d_n}{d_n - x\cos\theta_n - y\sin\theta_n} \right)^2, \tag{2.3}$$

$$u(x,y,n) = \frac{D(-x\sin\theta_n + y\cos\theta_n)}{d_n - x\cos\theta_n - y\sin\theta_n}, \tag{2.4}$$

$$v(x,y,z,n) = \frac{Dz}{d_n - x\cos\theta_n - y\sin\theta_n}, \tag{2.5}$$

where d_n denotes the distance between the X-ray source and the origin of the xyz coordinates.

According to Eq. (2.2), the time complexity of the back-projection stage is $\mathcal{O}(NXYZ)$, which represents the performance bottleneck of the FDK algorithm. In particular, 3D data accesses required to read $Q_n(u(x,y,n), v(x,y,z,n))$ and write $F(x,y,z)$ determine total execution time; as such, the back-projection stage constitutes more than half of the total execution time [82, 122]. Note that Eq. (2.5) can be efficiently computed via the following recurrence relation:

$$v(x,y,z+1,n) = v(x,y,z,n) + v(x,y,1,n). \tag{2.6}$$

Because the second term $v(x,y,1,n)$ can be precomputed for all n, this relation is useful for reducing computational cost; more specifically only an addition is needed to compute $v(x,y,z,n)$ within the z loop.

2.3.2 GPU Implementation of the FDK Algorithm

As presented in Table 2.1, Okitsu's method [82] was one of the most efficient comparative methods in terms of memory bandwidth. Figure 2.2(a) illustrates the reconstruction pipeline realized in their implementation. This pipeline has four major stages, *i.e.*, projection input, ramp filtering, back-projection, and volume output.

In the projection input stage, raw projections $P_0, P_1, \ldots, P_{N-1}$ are loaded from a storage device to CPU memory (*i.e.*, main memory), and then sequentially transferred to GPU memory. Next, the ramp filtering stage filters each projection P_n into Q_n, where $0 \leq n < N$, and transfers these filtered projections back to CPU memory. The back-projection stage then handles both filtered projections and volume via a divide-and-conquer approach to overcome limited GPU memory, as illustrated in Fig. 2.3. From the figure, volume F is partitioned along the z-axis into subvolumes $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_{Z/Z'-1}$, each with Z' xy-slices, whereas the filtered projections are partitioned into subsets $\mathcal{Q}_0, \mathcal{Q}_1, \ldots, \mathcal{Q}_{N/N'-1}$ with N' projections each, thus indicating that the back-projection kernel is launched N/N' times for each subvolume $\mathcal{F}_k$, where $0 \leq k < Z/Z' - 1$. Finally, the produced subvolumes are transferred back to CPU memory, and then written to the storage device in the volume output stage.

Algorithm 1 shows the pseudocode of the back-projection kernel, which generates subvolume F_k from subset $\mathcal{Q}_m$ of projections, where $0 \leq k < Z/Z' - 1$ and $0 \leq m < N/N' - 1$. This kernel assumes that each thread block is in charge of reconstructing a slab along the z-axis, where a thread with global index (x, y) computes $F(x,y,z)$ for all $kZ' \leq z < (k+1)Z'$ (*i.e.*, Z' voxels along the z-axis); this volume is stored in writable global memory, whereas projections are stored in textures to take advantage of hardware-accelerated interpolation. Note that this workload distribution scheme always achieves memory access coalescing when writing voxel values, because threads in the same warp access consecutive voxels on the same xy-plane, *i.e.*, threads are responsible for an angle-independent region of the volume. Similar to [85], atomic instructions are deployed to maximize the collision-free write throughput of the volume on line 14 of Algorithm 1.

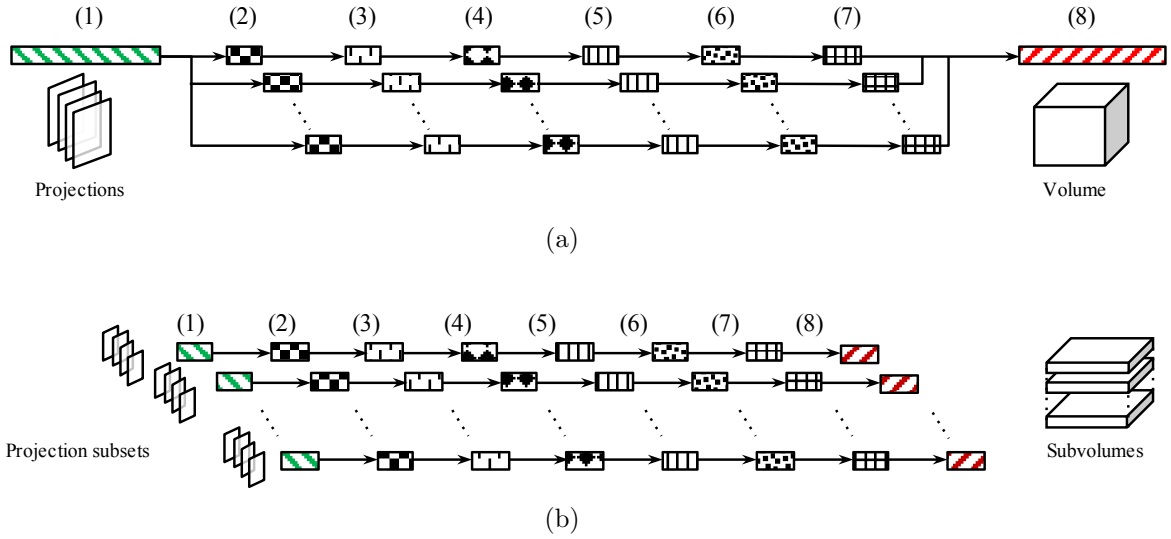

Figure 2.2: Reconstruction pipelines of (a) Okitsu's previous method [82] and (b) the proposed method. Both pipelines consist of the following eight steps, with the first and last steps not pipelined in the previous method: (1) raw projections are loaded from a storage device into CPU memory, (2) projections in CPU memory are then transferred to GPU memory; (3) ramp filtering is applied to produce filtered projections, (4) filtered projections are transferred back to CPU memory if necessary, (5) filtered projections are transferred from CPU memory to GPU memory if necessary; (6) back-projection is performed to produce a subvolume; (7) the subvolume is transferred to CPU memory, and (8) the subvolume in CPU memory is written to the storage device.

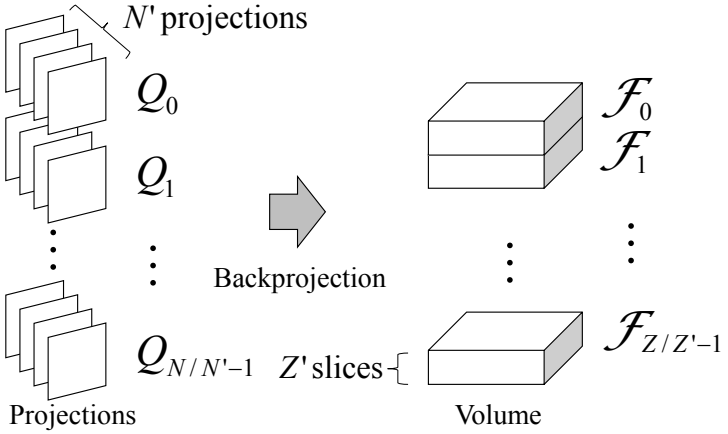

Figure 2.3: Data decomposition scheme in which the back-projection kernel is invoked for each pair $\langle \mathcal{Q}_m, \mathcal{F}_k \rangle$ of subset $\mathcal{Q}_m$ of projections and subvolume $\mathcal{F}_k$, where $0 \leq m < N/N' - 1$ and $0 \leq k < Z/Z' - 1$.

Algorithm 1: back-projection kernel

Input : subset $\mathcal{Q}_m$ of N' filtered projections, projection subset index m, and subvolume index k, where $0 \leq m < N/N' - 1$ and $0 \leq k < Z/Z' - 1$

Output: subvolume $\mathcal{F}_k = \{F_k(x, y, z) \mid 0 \leq x < X,\ 0 \leq y < Y,\ kZ' \leq z < (k+1)Z'\}$ of Z' slices

```
1  calculate responsible voxel coordinate (x, y) from thread index and thread block
     index;
2  z ← k × Z' ;                                          // first z coordinate in F_k
3  n ← m × N' ;                                          // first projection index in Q_m
4  for i ← 0 to N' − 1 do                                          // for each projection
5  |   w_i ← W(x, y, n + i); u_i ← u(x, y, n + i); v_i ← v(x, y, z, n + i) ;
   |    // Eqs.(2.3)-(2.5)
6  end
7  for j ← 0 to Z' − 1 do                                          // for each z-slice
8  |   t ← 0;
9  |   for i ← 0 to N' − 1 do                                      // for each projection
10 |   |   v_i ← v(x, y, z + j, n + i) ;                                    // Eq. (2.6)
11 |   |   r ← Q_{n+i}(u_i, v_i) ;                                // texture memory access
12 |   |   t ← t + w_i × r ;                                                // Eq. (2.2)
13 |   end
14 |   F_k(x, y, z + j) ← F_k(x, y, z + j) + t ;          // atomic write to global memory
15 end
```

2.4 Proposed Method

The main approach to optimizing GPU performance is to locate the performance bottleneck or resource constraint, and then attempt to exchange it for the use of another resource. Applying this to the FDK algorithm, lines 11 and 14 of Algorithm 1 determine the performance of the back-projection kernel, because an arithmetic instruction generally takes several clock cycles, whereas an off-chip memory access takes hundreds of clock cycles. Therefore, reducing or hiding memory access latency is pivotal to maximizing the reconstruction performance on a GPU. Our solution here is twofold: (1) maximize cache utilization to reduce memory access latency on line 11 of the algorithm; and (2) back-project multiple projections to reduce the number of off-chip memory access on line 14 [82]. Therefore, in the subsections below, we present the following three strategies to systematically optimize the reconstruction procedure:

1. Cache-aware loop organization, which identifies the best trade-off point between the cache hit rate and the number of off-chip memory accesses (Section 2.4.1)

2. A cache-aware data structure with a layered texture (Section 2.4.2)

3. A pipelined strategy that includes I/O (Section 2.4.3)

2.4.1 Cache-aware Loop Organization

As shown in Fig. 2.4, Eqs. (2.4) and (2.5) indicate that coordinates $(u(x, y, n), v(x, y, z, n))$ are similar in that they are calculated from consecutive projection angles. In other words, a series of pixels (u_n, v_n), (u_{n+1}, v_{n+1}), $\ldots$, $(u_{n+N'-1}, v_{n+N'-1})$ are intensively fetched from a small area of projections on line 11 of Algorithm 1; however, this high locality of coordinates may not be extended to that of texture pixels (*i.e.*, texels) because successive projections are stored with a stride of UV (Fig. 2.4). To handle this issue, Zinßer *et al.* [123] reversed the ji-loop of Algorithm 1 to form an ij-loop and synchronized threads before proceeding to the next projection, j; further, they changed the inter-projection first loop to an intra-projection first loop. As mentioned in Section 2.2, this method improved the texture cache hit rate by creating threads to simultaneously access the same projection within the inner j-loop, but threads consume more registers to store their responsible slabs, *i.e.*, voxel values along the z-axis. In more detail, variable t in Algorithm 1 must be extended as variables $t_0, t_1, \ldots, t_{Z'-1}$ because they cannot be flushed within the inner j-loop. Without this extension, the amount of global memory access cannot be minimized. In this study, we therefore adopt the original loop organization of the previous method [82] and present a data structure that is more tolerant to the intra-projection first loop: 3D access spreading over different projections.

As mentioned above, the previous method [82] partitions input projections into N/N' subsets to back-project each subset with a single kernel invocation. This multiplication method reduces both the number of kernel invocations and the number of global memory writes by a factor of N', because N' successive kernel executions are packed into a single execution. More specifically, the previous method [82] improves reconstruction performance by maximizing N', which leads to efficient use of registers; however, increasing N' consumes more registers for variables w_i, u_i and v_i, which decreases the occupancy on the SM.

Large N' also implies that threads can simultaneously fetch pixels from different projections because they are executed in an SIMT manner in which different warps are allowed to simultaneously process different lines in the kernel. Therefore, excessive N' decreases the L1/texture cache hit rate, particularly for high-resolution images, because stride UV between successive projections increases with image resolution $U \times V$. Note that synchronization is useful to enforce warps keeping pace with other warps, which prevents warps from accessing different projections; however, CUDA prohibits inter-block synchronization during kernel execution; thus, synchronization is not a perfect solution for this issue.

In addition to the loop organization described above, our method optimizes cache behavior by choosing the best granularity and shape of thread blocks according to the characteristics of memory access patterns. Because CUDA organizes threads in a three-level hierarchy composed of elementary threads, warps, and thread blocks, cache optimization can be achieved at each of these three levels accordingly. Sugimoto *et al.* [98] concluded that, with respect to volume-rendering applications, the most important level is the warp level, because memory access transactions are issued on a per-warp basis. Similarly, Okitsu *et al.* [82] concluded that square-shaped thread blocks are suitable for the back-projection kernel because warps are organized such that back-projection performance is averaged for arbitrary rotational angles. Refer to [82] for details.

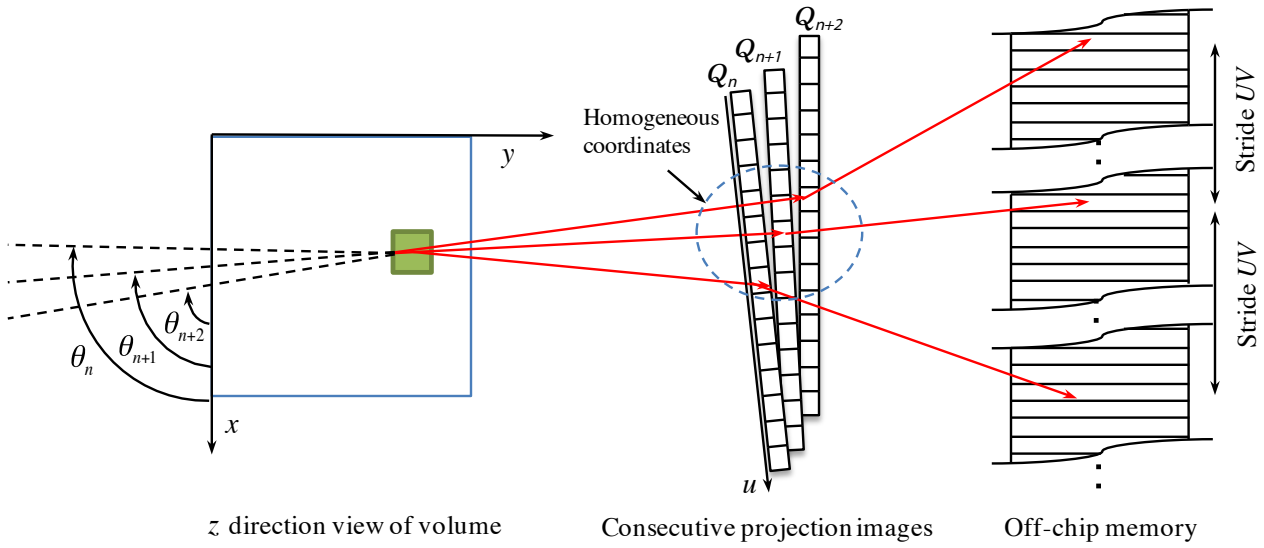

Figure 2.4: Schematic illustrating texture access. In this example, a thread block is responsible for producing the enclosed region in the volume. Given filtered projections Q_n, Q_{n+1}, and Q_{n+2}, warps in the given thread block access homogeneous texture coordinates (u_n, v_n), (u_{n+1}, v_{n+1}) and (u_{n+2}, v_{n+2}) on the filtered projections, respectively (see Algorithm 1). This locality on two-dimensional coordinates cannot extend to the locality of off-chip memory, because successive projections are stored with a stride of UV, where U and V are the horizontal and vertical resolutions of projections, respectively.

2.4.2 Cache-aware Data Structure with Layered Texture

The proposed data structure is realized via a layered texture [77], which packs equally-sized textures of the same type into a single object. Texels in a layered texture can be accessed using floating-point coordinate (x, y) and a layer specified by integer index l. Intra-layer interpolation can be applied to the xy-plane, but inter-layer interpolation is not available. Layered textures are ideal for processing multiple textures of the same size and format in that they reduce the overhead of texture access. Because its 3D locality has not been explicitly stated [77], we design a suite of micro-benchmarks to analyze its performance advantages with the memory access patterns of the FDK algorithm.

Figure 2.5 shows the pseudocode for the micro-benchmarks, which run a single GPU thread to determine whether a layered texture has been optimized for 3D locality; here, two access patterns are investigated to compare the performance of a layered texture and non-layered (*i.e.*, naive two-dimensional) textures. The first micro-benchmark, *i.e.*, Fig. 2.5(a), examines an intra-layer-first pattern in which a thread finishes fetching all texels from the current layer before accessing the next layer. Conversely, the other micro-benchmark, *i.e.*, Fig. 2.5(b), examines an inter-layer-first pattern in which a thread fetches texels from all layers at the same coordinate (u, v) before going to the next coordinate.

Figure 2.6 shows the timing and profiling results obtained using the micro-benchmarks. As shown in the figure, the inter-layer-first pattern on a layered texture achieved the best performance with a higher L1/texture cache hit rate. Consequently, we conclude that layered textures have 3D locality and are thereby optimized for 3D texture access. This performance characteristic agrees with the memory access pattern of the back-projection kernel, which is

```
for l ← 0 to L − 1 do {for each layer}
  for v ← 0 to V − 1 do
    for u ← 0 to U − 1 do
      fetch texel at coordinate (u, v) of layer l;
    end for
  end for
end for
```

(a)

```
for v ← 0 to V − 1 do
  for u ← 0 to U − 1 do
    for l ← 0 to L − 1 do {for each layer}
      fetch texel at coordinate (u, v) of layer l;
    end for
  end for
end for
```

(b)

Figure 2.5: Micro-benchmarks for evaluating texture access performance. Here two access patterns were examined with $U = V = 1024$ and $1 \leq L \leq 64$; the access patterns are (a) intra-layer-first and (b) inter-layer-first patterns. A single thread was used to fetch all texels. The innermost loop of both patterns was unrolled for optimization.

an inter-layer-first pattern as shown on lines 7–15 of Algorithm 1.

According to our benchmark results, we decided to use a layered texture with the inter-layer-first loop. We implemented the layered texture with the texture object application programming interface (API) introduced in the Kepler architecture [70]. Compared to the legacy texture reference API, the texture object API simplifies the resulting programming style and eliminates several restrictions [77]. For example, the texture object API does not require manual binding and unbinding of texture references to memory addresses. Therefore, texture references can be used in a more dynamic manner, whereas the texture reference API requires texture references to be declared as static global variables.

2.4.3 Pipelined Strategy that Includes I/O Interface

Similar to Blas *et al.* [9], our out-of-core pipelined strategy decomposes not only computation steps but also file I/O steps, namely steps (1) and (8) in Fig. 2.2(b), thus overlapping them with other steps. Such an I/O-included pipelining strategy is extremely important for large amounts of data that exceed not only GPU memory but also CPU memory. Without this strategy, the entire reconstruction throughput is bounded by file I/O time, even though the back-projection procedure is significantly accelerated on the GPU.

Algorithm 2 presents pseudocode for our pipelined FDK algorithm. As with the previous method [82], this algorithm divides the input projections and output volume into N/N'

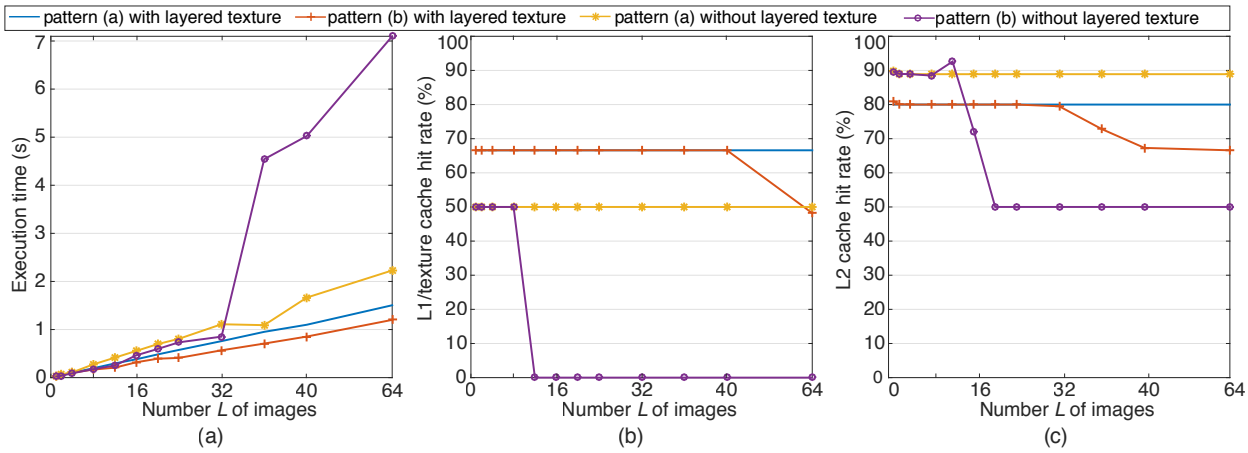

Figure 2.6: Benchmark and profiling results for different loop organizations and data structures, showing (a) execution time, (b) L1/texture cache hit rate, and (c) L2 cache hit rate. Here, intra-layer-first and inter-layer-first patterns were investigated with a layered texture and non-layered (*i.e.*, naive two-dimensional) textures.

subsets and Z/Z' subvolumes, respectively. The filtering and back-projection procedures are then carried out for each pair $\langle \mathcal{Q}_m, \mathcal{F}_k \rangle$ of projection subset $\mathcal{Q}_m$ and subvolume $\mathcal{F}_k$, where $0 \leq m < N/N' - 1$ and $0 \leq k < Z/Z' - 1$. Given the limited capacity of GPU memory, filtered projections are pushed back to CPU memory to save GPU memory consumption for the subvolume to be generated (*i.e.*, line 8 of Algorithm 2). Further, filtered projections are reused to allow us to skip the filtering step for succeeding subvolumes (*i.e.*, line 12). Note that the buffers to be used for the back-projection kernel are doubled to enable overlaps with other steps (*i.e.*, lines 12 and 16). Without these double-buffers, overlaps cannot be achieved due to the data dependence that exists between succeeding steps.

2.5 Experimental Results

We compared our proposed method with the previous method [82] in terms of reconstruction time. All timing results were measured using the NVIDIA Visual Profiler [80]. Table 2.2 lists the specifications of our experimental machine.

In our experiments, we used three sets of the Shepp-Logan phantom [94] at different resolutions: a small dataset with $U = V = X = Y = Z = 512$, a medium dataset with $U = V = X = Y = Z = 1024$ and a large dataset with $U = V = X = Y = Z = 2048$, each with $N = 1200$ projections. The middle slice of the reconstructed volume is shown in Fig. 2.7. Pixels in the projections and voxels in the volumes consist of four bytes; thus, the small, medium, and large datasets consumed 1.7 GB, 8.7 GB and 50.8 GB of memory, respectively. Therefore, the medium and large datasets could not be entirely stored in CPU memory or GPU memory (see Table 2.2).

Algorithm 2: Fully pipelined FDK reconstruction

Input : set $\mathcal{P} = \{P_0, P_1, \ldots, P_{N-1}\}$ of raw projections
Output: volume $F = \bigcup_{k=0}^{Z/Z'-1} \mathcal{F}_k$

```
1  for k ← 0 to Z/Z' − 1 do in parallel                          // for each subvolume
2      for j ← 0 to N/N' − 1 do                                  // for each projection subset
3          if k == 0 then
4              for i ← 0 to N' − 1 do in parallel                // for each projection in
                projection subset
5                  load raw projection P_{N'j+i} from the storage device;
6                  transfer P_{N'j+i} from CPU to GPU asynchronously;
7                  Q_{N'j+i} ← RampFilteringKernel(P_{N'j+i}) ;          // Eq. (2.1)
8                  transfer Q_{N'j+i} from GPU to CPU asynchronously;
9              end
10             set Q_j = {Q_{N'j}, Q_{N'j+1}, ..., Q_{N'j+N'−1}};
11         else
12             transfer Q_j = {Q_{N'j}, Q_{N'j+1}, ..., Q_{N'j+N'−1}} from CPU to GPU
                asynchronously ;                                  // Q_j is double buffered on GPU
13         end
14         F_k ← back-projectionKernel(Q_j, j, k) ;                          // Algorithm 1
15     end
16     transfer F_k from GPU to CPU ;                             // F_k is double buffered on CPU
17     store F_k to the storage device asynchronously;
18 end
```

Table 2.2: Specifications of our experimental machine.

Item	Specification
CPU	Intel Core i7-4770K
Main memory capacity	32 GB
GPU	NVIDIA GeForce GTX 980
Clock speed	1126 GHz (base), 1216 GHz (boost)
Texture fill rate	144.1 Gtexel/s (base), 155.6 Gtexel/s (boost)
GPU memory capacity	4 GB
GPU memory bandwidth	224.3 GB/s
Bus interface	PCIe 3.0 ×16 bus
Storage	Samsung 850 EVO SSD 500 GB
OS	Fedora 22 64-bit
Compiler	CUDA 7.5 and gcc 5.11
Compiler option	`-O3 -arch=compute_52 -code=compute_52 -Xcompiler -fopenmp -lgomp`
Driver	352.55

2.5.1 Parameter Configuration

We conducted preliminary experiments to identify execution parameters that achieve the highest reconstruction performance; these parameters included (1) the granularity and shape of thread blocks, (2) projection subset size N', and (3) subvolume size Z'.

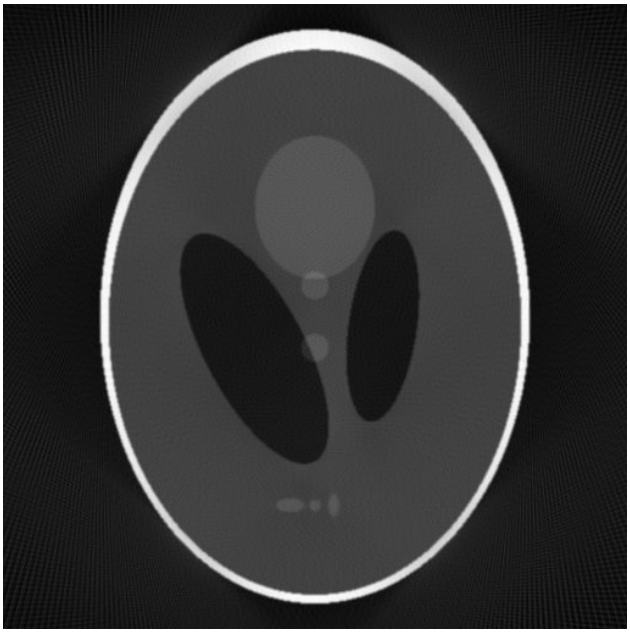

Figure 2.7: Shepp-Logan phantom [94] reconstructed by our experimental machine.

The appropriate shape of the thread blocks was firstly investigated for the back-projection kernel. Figure 2.8 shows the back-projection times, L1/texture cache hit rates, and L2 cache hit rates for all rotational angles with different thread block shapes. As expected (see Section 2.4.1), square blocks of 16×16 threads achieved the highest performance with approximately 95% L1/texture cache hit rates. In contrast, the lowest performance was obtained with a shape of 256×1 threads, which yielded a 50% L1/texture cache hit rate, observed around rotational angles of 90 and 270 degrees. With these rotational angles, resident warps in such non-squared thread blocks accessed wide rows of projections, thereby dropping the L1/texture cache hit rate.

Next, as shown in Fig. 2.9, we investigated back-projection performance with different projection subset sizes N' and thread block sizes. As shown in Fig. 2.9(a), setting $N' = 20$ and using blocks of 256 threads yielded the best performance for the medium dataset. As for projection subset size N', Figs. 2.9(b) and 2.9(c) show that there was a trade-off between the L1/texture cache hit rate and the number of global memory accesses, as stated in Section 2.3.2. The number of global memory accesses here is given by $8 \times X \times Y \times Z \times N/N'$ in bytes, because four-byte voxels are loaded and stored once per kernel invocation.

Conversely, the previous method [82] improved reconstruction performance by maximizing N' on the G80 architecture [69]; thus, this previous idea must to be adapted to the new Maxwell architecture accordingly. In other words, the previous results partially agreed with our results when $N' < 20$, where the number of global memory accesses decreased with N', but back-projection time increased slightly with N' when $N' > 20$. As shown in Fig. 2.9(b), excessive N' increased memory access strides and degraded L1/texture hit rates, which outweighed the performance gain contributed by fewer kernel invocations and fewer write accesses.

As noted earlier, global memory access bypasses the L1 caches; therefore, the L1/texture

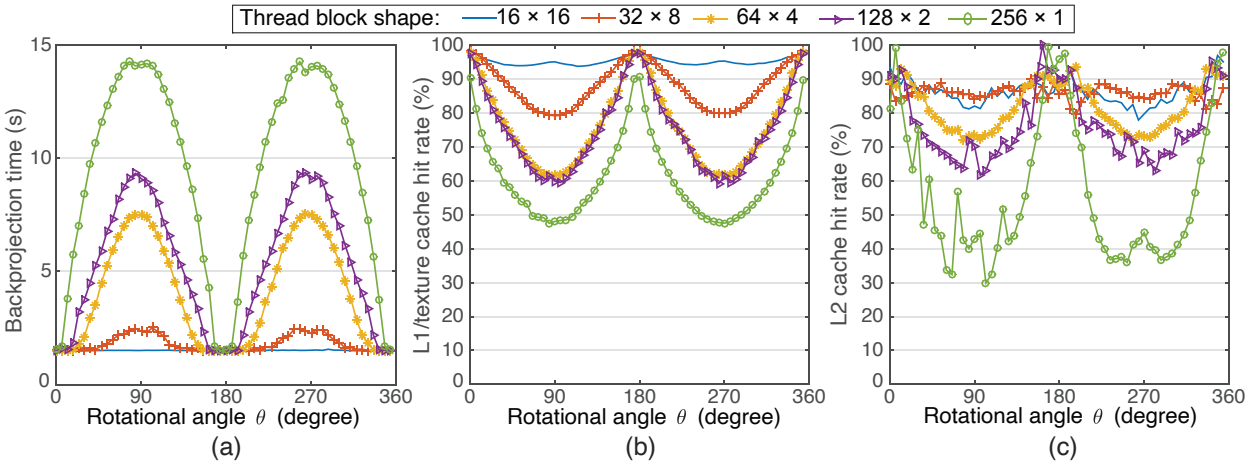

Figure 2.8: Back-projection performance and profiling results with different shapes of thread blocks and rotational angles: (a) back-projection time, (b) L1/texture cache hit rate, and (c) L2 cache hit rate. These results were obtained with a thread block size of 256 for the medium dataset, with $N' = 20$ and $Z' = 512$.

cache hit rate primarily corresponds to texture memory performance (*i.e.*, the read throughput of the back-projection kernel). In summary, an appropriate N' can be selected according to the trade-off mentioned above. Similarly, we found that $N' = 20$ and $N' = 16$ were the best sizes (*i.e.*, the trade-off point) for the small and large datasets, respectively.

Next, we discuss thread block size. As we decrease the thread block size, more thread blocks can be dispatched to each SM, which leads to higher occupancy; however, a thread block size of 64 was too small for assigning a square region to the SMs. In this case, we found that eight thread blocks were resident on each SM. Although each thread block was responsible for a square 8×8 region, these eight squares appeared in a rectangular region because of the cyclic assignment described in Section 1.2.2. Such a rectangular region cannot be efficiently back-projected from an unfavorable angle, as we presented in Fig. 2.8, for the shape of thread blocks. In contrast, the number of resident thread blocks decreases as we increase the thread block size; however, a thread block size of 512 was too large to maximize the projection subset size N', because such a large thread block consumes more registers. Execution for $N' > 8$ failed when using a thread block size of 1024. In summary, our solution is to maximize the thread block size such that its shape is kept as a square (*i.e.*, a block of 16×16 threads).

Finally, we minimized subvolume size Z' such that (1) the off-chip memory could hold both a subvolume and a projection subset and (2) at least two subvolumes were generated for overlapping file I/O time ($Z/Z' \geq 2$). Note that for all subvolumes, filtered projections must be transferred to GPU memory. Consequently, the amount of data transfer between CPU and GPU increases with Z/Z'. According to this guideline, we used $Z' = 256$, 512, and 128 for the small, medium, and large datasets, respectively.

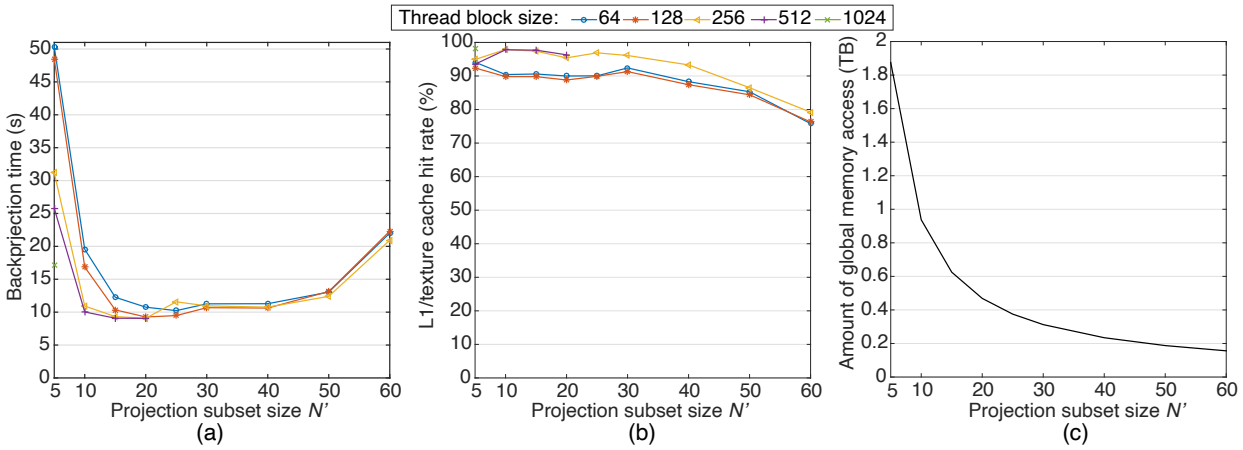

Figure 2.9: Back-projection performance and profiling results with different projection subset sizes N' and thread block sizes: (a) back-projection times, (b) L1/texture cache hit rates, and (c) number of global memory accesses. These results were obtained with a medium dataset and $Z' = 512$.

2.5.2 Breakdown Analysis

Using the best parameters identified above, we investigated the impact of our data structure and that of our I/O-included pipeline using different data sizes; results are summarized in Table 2.3. Here, the layered texture efficiently improved back-projection performance for the small and medium datasets, achieving speedups of at least a factor of 1.44 over the given baseline; however, speedup decreases to only a factor of 1.14 for the large dataset, which fetches texels from projections that are four times as large. In this case, data size $4UV$ of a projection reaches 16 MB, which immediately depletes the L1/texture and L2 caches. For such large datasets, our pipeline increased speedup from a factor of 1.14 to that of 1.47 by realizing overlapped file I/O steps (1) and (8), which consumed 31% of the overall time before such overlapping was achieved. Thus, our I/O-included pipeline complements cache-aware back-projection, thereby demonstrating large speedups for both small and large datasets.

Next, we investigated the breakdown of reconstruction time for the large dataset, with our results summarized in Table 2.4. We evaluated the impact of our pipelined strategy; thus, all comparative methods used the same cache-aware kernel during our measurements. In addition, a non-pipelined version deployed synchronous APIs such that the sum of the breakdowns never equaled the execution times of the pipelined versions, which deployed asynchronous APIs. The previous method reduced the execution time from 265.5 s to 211.8 s, and our proposed method further reduced the execution time to 159.6 s, achieving speedups of 1.66 and 1.33 times the non-pipelined method and the previous method, respectively. Thus, pipelining must be applied not only to the filtering and back-projection steps (2)–(7) but also to file I/O steps (1) and (8).

With respect to the filtering stage, *i.e.*, steps (1)–(4), the previous method had little advantage over the non-pipelined implementation in that only data transfer stages (2) and (4) were partially overlapped with filtering stage (3). In contrast, our proposed method realized a full overlap, including file I/O stage (1), such that the execution time was reduced from 59.6 s to 38.5 s.

Table 2.3: Comparing the performance of our proposed method and previous method using different data sizes. The baseline corresponds to Okitsu's method [82] with the best parameters tuned for the Maxwell architecture. Further, "Both strategies" corresponds to our proposed method.

Data size	Step	Baseline [82]	Layered texture		I/O pipeline		Both strategies	
		Time (s)	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Small	(6) Backproj.	1.7	1.1	1.55	1.7	1.00	1.1	1.55
	Total	5.4	4.6	1.17	4.3	1.26	3.9	1.38
Medium	(6) Backproj.	13.1	9.1	1.44	13.1	1.00	9.1	1.44
	Total	31.3	25.9	1.20	25.0	1.25	21.8	1.43
Large	(6) Backproj.	112.9	98.4	1.14	113.0	1.00	98.3	1.14
	Total	234.3	211.8	1.11	166.1	1.41	159.6	1.47

Table 2.4: Breakdown analysis of execution times for the large dataset. Here, "No pipeline" means that all steps were processed sequentially with synchronous APIs, whereas the "Previous pipeline" and "Proposed pipeline" were processed asynchronously. These results were obtained with the large dataset, with $N' = 16$ and $Z' = 256$.

Step	No pipeline	Previous pipeline [82]	Proposed pipeline
(1) T_1: Storage → CPU	36.9	—	—
(2) T_2: CPU → GPU	3.2	—	—
(3) T_3: Ramp filtering	16.3	—	—
(4) T_4: GPU → CPU	3.2	—	—
(5) T_5: CPU → GPU	55.1	—	—
(6) T_6: Back-projection	98.4	—	—
(7) T_7: GPU → CPU	5.4	—	—
(8) T_8: CPU → storage	46.5	—	—
(1)–(4)	59.6	58.1	38.5
(5)–(8)	205.4	153.7	121.1
Total	265.5	211.8	159.6

With respect to the back-projection stage, *i.e.*, steps (5)–(8), the previous method overlapped step (5) with step (6), such that the corresponding execution time was reduced from 205.4 s to 153.7 s. Our method further realized an overlap of steps (7) and (8), thereby reducing the execution time to 121.1 s. Note that step (8) for the last subvolume cannot be overlapped with other steps. Similarly, step (1) for the first subvolume cannot be overlapped with other steps.

2.5.3 Efficiency Analysis

To analyze the performance bottleneck of our method, we measured arithmetic performance, L1/texture cache throughput, L2 cache texture load throughput, and texture fill rate using the NVIDIA Visual Profiler; results are shown in Fig. 2.10. According to Eq. (2.2), each voxel requires one pixel per projection; thus, the effective texture fill rate can be given by $NXYZ/T_6$, where T_6 is the back-projection time as shown in Table 2.4. The peak texture fill rate was derived according to a boosted clock speed because the clock speed was boosted during back-projection.

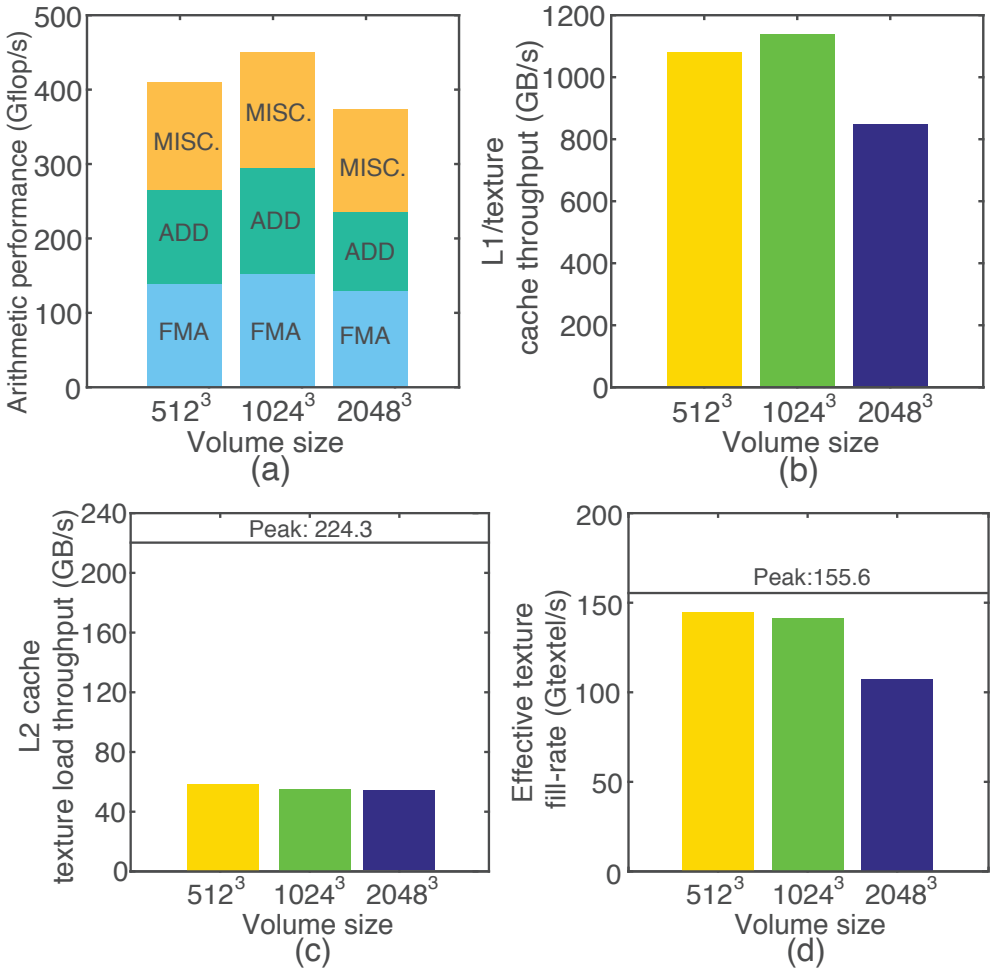

Figure 2.10: Profiling results for different data sizes: (a) arithmetic performance in Gflop/s, (b) L1/texture cache throughput, (c) L2 cache texture load throughput, and (d) effective texture fill rate. FMA, ADD, and MISC in (a) refer to fused multiply-add [77], addition, and other instructions, respectively. The horizontal lines in (c) and (d) are peak memory bandwidth and peak texture fill rate, respectively, with the latter derived according to the boosted clock speed presented in Table 2.2.

Our results indicate three key findings. First, Fig. 2.10(d) implies that the texture fill rate determined reconstruction performance for the small and medium datasets. The Maxwell architecture has eight texture units devoted to each SM; thus, 16 CUDA cores share a single texture unit. Given these limited resources, the effective texture fill rates were limited to approximately 141.2 Gtexel/s, which is 9.3% lower than the peak (boosted) value of 155.6 Gtexel/s. These effective values were close to the peak (base) value of 144.1 Gtexel/s; thus, we conclude that our back-projection kernel is highly optimized for the Maxwell architecture. As for the large dataset, we determined that the average L2 texture read cache hit rate decreased to 55.8% due to the increased projection size, which decreased the effective texture fill rate to 104.9 Gtexel/s.

Second, our cache-aware method and the rich caching mechanism of the Maxwell architecture moved the performance bottleneck from the off-chip memory bandwidth to the texture

fill rate. As shown in Fig. 2.10(c), the effective throughput was approximately one-quarter of the theoretical peak value, which indicates that off-chip memory accesses do not limit back-projection performance on the Maxwell architecture. These results were not observed in the previous study [82], which concluded that off-chip memory access was the performance bottleneck of the back-projection kernel on the G80 architecture.

Third, reconstruction performance for the large dataset was sacrificed due to limited GPU memory. As compared to the medium dataset, the large dataset consisted of four times larger xy-slices, which decreased the subvolume size Z' from 512 to 128. This decrease in Z' led to more kernel executions and reduced the amount of memory accesses per warp, *i.e.*, $32N'Z'$ in bytes, from 1.25 MB to 0.25 MB. An alternative solution for increasing Z' is to partition the volume along the x- or y-plane instead of the z-plane; however, this solution requires recombining the volume after reconstruction. Such a post-processing task will likely slow the I/O-included pipeline.

2.5.4 Estimated Performance on the Future Architecture

In this subsection, we brief the evolution of the GPU architecture and give our analysis for implementing the FDK algorithm with newer architecture. We listed three typical GPUs with different architectures in Table 2.5: G80 (previous work [69]), GTX 980, RTX 2080 ([75], released in 2018). The theoretical texture fill-rate is calculated as GPU core clock $\times$ the number of texture units. From G80 to GTX 980, the core clock improved about 10%. Comparatively, texture units, Bandwidth and L1 cache size improved about 3–4$\times$. The most significant improvement is the L2 cache which increased by approximately 21$\times$. Here we use the middle size data to illustrate how the L2 cache size increase enables the proposed cache-aware strategies. The volume is reconstructed layer by layer in z direction. Threads blocks in the same kernel invocation will use similar coordinates to access pixels on projection images. The texture units will fetch two rows from images for interpolation. The data size is 2$\times$1024$\times$4B=8KB. If 20 images are packed into a single kernel invocation, the data access will be 160KB. This is much larger than the L2 cache size on the G80 GPU, which will cause cache hit-miss anyway for thread blocks with different shapes and sizes. If we reduce the number of packed images, the kernel invocations will increase greatly. Owing to those constraints, it is impossible to employ any cache-ware strategy on the G80 GPU. With 2048KB L2 cache on the GTX 980, approximately 24 rows of 20 images can be cached. This improved cache enables the proposed cache-aware strategies to move the bottleneck from bandwidth bound to texture fill-rate.

Regarding the architecture transformation from GTX 980 to RTX 2080, the improvements are similar for all the listed items, which are around 1.5–2$\times$. We can see that although a lot of new features (*e.g.* half-precision computing [75]) have been added in the last decade, the memory hierarchy of GPUs and computing units to cache ratio has settle down. We speculate that accelerating the FDK algorithm will be a compute-bound (mainly texture fill-rate) problem.

Table 2.5: GPU architecture comparsion.

	G80 (previous work)	GTX 980	RTX 2080
Architecture	Tesla [69]	Maxwell [82]	Turing [75]
Core clock (GHz)	1.1	1.2	1.8
Texture units	32	128	184
Texture fill-rate (Gtextel/s)	35.2	155	348
Bandwidth(GB/s)	86.4	224.3	448
L1 cache (KB)	16	48	64
L2 cache (KB)	96	2048	4096

2.6 Conclusions

In this chapter, we presented a cache-aware optimization method to accelerate out-of-core cone beam CT reconstruction on a GPU. Our proposed method extended the previous method described in [82] and accelerated the FDK algorithm via three key strategies, *i.e.*, an improved loop organization strategy, an improved data structure, and an I/O-included pipeline. We also presented tuning guidelines for determining the best configuration for the granularity and shape of thread blocks, as well as the projection subset size and subvolume size, *i.e.*, the granularity of I/O data to be streamed through the pipeline.

Our experimental results showed a trade-off between the texture cache hit rate and the number of memory accesses. We also found that it took 159.6 s on a GeForce GTX 980 to reconstruct a 2048^3-voxel volume from 1200 2048^2-pixel projections, consuming 50.8 GB of memory. This reconstruction performance is approximately 1.47 times higher than that achieved by the previous method [82]. Concerning GPU optimization, we found that it is not necessarily more efficient to compact as many tasks as possible into kernel execution to decrease kernel executions. Instead, proper tuning is required to identify the optimum number of tasks that will minimize the overall time required. With the aid of texture interpolation and cache-aware strategies, our presented GPU implementation achieved performance advantages over other computing platforms.

Chapter 3

Reducing the Amount of Out-of-Core Data Access for GPU-Accelerated Randomized SVD

3.1 Introduction

SVD [34] is a matrix approximation algorithm that finds two orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ and a diagonal matrix $\mathbf{\Sigma}$ of an input matrix $\mathbf{A}$ such that $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, where $\mathbf{A}$ can be approximated with matrices smaller than itself by truncating $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{\Sigma}$. Studies on improving the performance and numerical stability of SVD algorithms have been ongoing ever since its advent [28, 33, 55], and successfully applied to various fields, such as bioinformatics [102, 107], physics [27, 41], and machine learning [10, 60, 121]. In particular, SVD of dense *tall-skinny matrices*, whose the height (m rows) is at least one magnitude larger than the width (n columns), is of great interest to researchers in computer vision [81], image compression [6], facial recognition [103], and data analysis [40].

Recently, RSVD algorithms [39, 57, 64] have been proposed to further accelerate SVD by exploiting the low-rank structure inherent in matrix data. Compared with classical deterministic algorithms [34], randomized algorithms have been shown to access the input data less number of times, while maintaining the desirable accuracy of the approximation [39, 57, 64]. RSVD is typically build upon random sampling [22, 39] and power iteration methods [39, 54, 88]. The sampling method constructs a subspace of the input matrix, which reduces dimension. The power method takes powers of $\mathbf{A}$ (*i.e.*, $\mathbf{A}^\top\mathbf{A}$) to increase the approximation accuracy. Both underlying methods can be implemented using GEMM routines, whereas general matrix-vector multiplication (GEMV) [31] is required for deterministic SVD computation. GEMM is more suitable for modern parallel computers, where it achieves 20–40 times higher flop/s than GEMV [18, 31].

While matrix approximation, such as SVD, has been made efficient based on randomization methods, modern computing architectures, such as the use of the GPU accelerators [74], enable the development of even faster algorithms by taking the advantage of parallel computing. Nevertheless, there are a few major challenges that prevent matrix approximation algorithms to fully benefit from the modern computing architectures. First, large matrix data

may not fit into the GPU memory due to its limited capacity. Second, communication across distinct memory hierarchies or networks often constitutes a performance bottleneck [18, 37] due to the increasing gap between arithmetic and communication performance [15, 93].

The above challenges have been addressed using the following two strategies: communication-avoiding algorithms [5, 42] reduce the communication of intermediate data during computation, whereas pass-efficient algorithms [22, 23] save the memory bandwidth by reducing the number of passes over data. As for the multi-pass RSVD [39], target data is passed over $2q + 1$ times to attain a high-accuracy approximation, where q denotes the number of iterations in the power method [39, 88]. By contrast, single-pass algorithms [101, 118] access the target data in just one pass. However, single-pass algorithms include iterations to construct a *sketch* [39, 101, 118], which have data dependency that avoids parallelization required by the modern computing architectures. Furthermore, there exists an accuracy-performance trade-off in single-pass algorithms [101, 118]. Due to this accuracy issue, multi-pass RSVD is preferred in convergence-sensitive applications such as RPCA, where the noise and low-rank component in input data are typically separated in an iterative manner [14].

In this study, we focus on a multi-pass RSVD algorithm proposed by Martinsson *et al.* [39, 64], which has a higher accuracy than single-pass algorithms according to solid error-bound analysis. We extend this algorithm so that large tall-skinny matrices can be rapidly decomposed using a divide-and-conquer method that reduces out-of-core data access on a CPU-GPU heterogeneous system. Compared with previous in-core algorithms [106, 114], which made the assumption that input and intermediate data can be fully stored in the GPU memory, we consider an RSVD algorithm at scale, where the matrices include more than 10^6 entries. Our out-of-core approach relaxes the limitation on the data size by allowing the data to be stored in both the CPU and GPU memories.

The main contributions of this research include:

- **Highly tuned, out-of-core GEMM with theoretical performance model**.

  Since GEMM is a building block of RSVD algorithms, we extensively tuned the GEMM operation with theoretical performance analysis based on an extension of the roofline model [111]. The extended model shows that GPU-accelerated out-of-core GEMM is bandwidth bound for tall-skinny matrices. In addition, we present experimental results where our out-of-core scheme achieved higher performance than previous GEMM schemes.

- **Two out-of-core RSVD methods, namely Fused and Gram**.

  Both methods are based on three common schemes: (1) the above-mentioned out-of-core GEMM scheme; (2) a data-access reduction scheme based on 1D data partitioning; and (3) a first-in, first-out (FIFO) scheme that reduces CPU-GPU data transfer by reverse iteration. The Fused method is a communication-avoiding algorithm because the method merges GEMM operations to reduce the amount of the CPU-GPU data transfer, (*i.e.*, out-of-core data access). By contrast, the Gram method is a pass-efficient algorithm because the method explicitly computes the Gram matrix to reduce the number of data passes (*i.e.*, both in-core and out-of-core data access) from $2q + 1$ to 3.

- **Case study with a practical application**.

  We apply the Gram method to nuclear norm minimization in an RPCA algorithm [10] that heavily relies on SVD computation. We found that our GPU-based implementation provided 23.3× faster RSVD computation compared with that of a CPU-based implementation, doubling the RPCA performance for the video background subtraction.

Our source code, which will be included into the MAGMA package [2], is freely available at `http://www-ppl.ist.osaka-u.ac.jp/research/code/`.

The rest of this chapter is organized as follows. Section 3.2 provides an overview of related studies. Section 3.3 presents a technical background regarding RSVD. Section 3.4 outlines the proposed highly tuned out-of-core GEMM scheme with its theoretical performance analysis and preliminary evaluation. Section 3.5 details our proposed methods constructed over the underlying GEMM scheme. Section 3.6 compares and contrasts the proposed methods with the existing methods. Section 3.7 describes the case study with an RPCA application. Conclusions and prospects for future work are provided in Section 3.8.

3.2 Related Work

In this section, we brief the related work regarding deterministic SVD algorithms, randomized algorithms, and GPU-accelerated randomized algorithms.

3.2.1 Deterministic SVD Algorithms

In 1965, Golub and Kahan [33] proposed the first stable SVD algorithm for computers using a bidiagonlization method. EISPACK [95] first implemented the bidiagonlization method in Fortran. EISPACK was designed to run on a single-core CPU and was replaced by LINPACK [20], which first implemented the SVD algorithm with Basic Linear Algebra Subprograms (BLAS) interface. The performance of LINPACK was limited by the level-1 BLAS (BLAS1) implementation and benefited little from multi-core architectures [18]. LAPACK [4] redesigned the SVD algorithm to use level-3 BLAS (BLAS3) routines wherever possible to improve the performance on the multi-core CPUs. Recently, a two-stage bidiagonal reduction method has been proposed to adapt SVD to new computer architectures like GPU accelerators [31].

Theoretically, deterministic SVD of a matrix $\mathbf{A}$ can be calculated with its Gram matrix $\mathbf{A}^\top\mathbf{A}$, which is a well-known method [97]. However, forming the Gram matrix is usually avoided due to its high computational cost in linear algebra packages like LAPACK [4] or MAGMA [2]. In this work, we apply this idea to out-of-core RSVD computation; we introduce a method which explicitly forms the Gram matrix to reduce the number of data passes, *i.e.*, the amount of in-core and out-of-core data access.

3.2.2 Randomized Algorithms

Randomized algorithms [28, 39, 57, 63, 64, 88, 89] have been proposed to reduce the time and space complexities required for the approximation of high-dimensional data. The efficiency

of such algorithms in tackling large scale data has led to an increased interest from the HPC community.

In general, randomized algorithms have a typical computation flow. First, they construct a subspace of the input data using random sampling. Computation is then performed only on the sampled subspace to reduce the costs of computation, communication, and storage. Randomized algorithms are popular in the field of big data analytics, where large quantities of data are missing or contain noise. There is a strong demand for low-precision approximation that is useful for the restoration of the entire data and elimination of irrelevant data hindering data analysis. Randomized algorithms typically employ one of the two sampling methods: namely, uniform and nonuniform sampling methods.

The uniform sampling method uses independent and identically distributed random numbers as the entries of sampling matrices, which are then multiplied with the target matrix. This method is also called random projection; it has a strong relative-error bound and widely used in randomized matrix factorization [39]. The major objective of random projection is to project the high-dimensional data onto a low-dimensional subspace by exploiting the low-rank characteristics of the data [39]. Deterministic decomposition is then performed on this subspace, and the decomposed results are projected back to form the full factorization. However, uniform sampling has relatively higher computational cost compared with that of nonuniform sampling. In particular, given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and sampling matrix $\mathbf{Q} \in \mathbb{R}^{n \times \ell}$, it takes $\mathcal{O}(mn\ell)$ time to compute the sampled matrix $\mathbf{AQ}$.

In contrast, the nonuniform sampling method constructs a subspace by selecting a certain set of vectors from the target data. Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, an importance sampling distribution [22] of the input matrices is first computed to perform the selection. The distribution and selection have the time complexity of $\mathcal{O}(mn)$, which is lower than the overhead of the uniform sampling mentioned above. Moreover, nonuniform sampling has a higher accuracy compared with that of uniform sampling. Hence, our base RSVD algorithm [39, 64] deploys uniform sampling.

3.2.3 GPU-Accelerated Randomized Algorithms

Randomized algorithms have been implemented on GPUs to achieve further acceleration. For example, low-rank approximations of dense matrices were computed and evaluated with a truncated SVD [51] and a truncated QR factorization with column-pivoting [65]. The RSVDPACK library [106] contains a set of randomized algorithms for computing low-rank matrix approximations on a single GPU. These studies assume that the matrix data are small enough to fit into the GPU memory. Therefore, the maximum data size was limited by the capacity of the GPU memory, which is an order of magnitude smaller than that of the CPU memory.

As for large matrices whose data size exceeds the capacity of the GPU memory, traditional matrix factorization algorithms have been investigated for more than a decade [2, 105]. For example, divide-and-conquer methods have been proposed to perform out-of-core LU [13], QR, or Cholesky factorization [115, 116]. Similarly, underlying BLAS routines have been extended to deal with large matrices. To the best of our knowledge, cuBLAS-XT [76] is the first library, implementing the out-of-core BLAS routines. BLASX [109] deploys a least recently used cache management scheme to implement the BLAS routines for out-of-core

matrices, which aims at reducing the amount of the data transfer between the CPU and GPU.

While GPUs are widely used as accelerators for memory- or compute-intensive applications, GPU programming is still not easy partly due to its complex memory hierarchy levels. In particular, application developers are required to manually manage CPU and GPU memories to gain a high performance of GPU-based systems. To deal with this issue, NVIDIA introduced Unified Memory [77], that realizes an integrated memory space available from both the CPU and GPU. This capability frees application developers from explicitly managing data movement between the CPU and GPU. Furthermore, Unified Memory accepts large data that may exceed the capacity of the GPU memory.

3.3 RSVD Algorithm

Algorithm 3 lists a pseudocode of the RSVD algorithm [39, 64]. Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the target rank k, oversampling parameter o, and power iteration count q, the algorithm outputs matrices $\mathbf{\Sigma}$, $\mathbf{U}$, and $\mathbf{V}$, where $\mathbf{\Sigma}$ is the diagonal matrix whose diagonal entries approximate the k-largest singular values of $\mathbf{A}$, and $\mathbf{U}$ and $\mathbf{V}$ approximate the corresponding left and right singular vectors, respectively. The oversampling parameter o is added to the target rank k to guarantee the approximation accuracy for matrices with a slow singular value decay.

First, the RSVD algorithm generates the basis vectors $\mathbf{P}$ and $\mathbf{Q}$ that approximate the range and domain of the matrix $\mathbf{A}$, respectively. The power iteration method in lines 2–7 improves the approximation accuracy. During these q power iterations, basis vectors of $\mathbf{P}$ and $\mathbf{Q}$ are orthogonalized to maintain the numerical stability. After the power iterations, QR factorization is performed on $\mathbf{P}$ in line 9 such that the upper triangular matrix $\mathbf{B}$ is the projected matrix of dimension $\ell \times \ell$ (*i.e.*, $\mathbf{B} = \mathbf{P}^T\mathbf{A}\mathbf{Q}$). In other words, a small matrix $\mathbf{B}$ can be created by GEMM when matrix $\mathbf{A}$ is low-rank, *i.e.*, $\mathrm{rank}(\mathbf{A}) = k \ll \min(m, n)$. This small matrix B is useful for revealing the SVD of the original matrix $\mathbf{A}$ at low cost. The SVD of $\mathbf{B}$ is then computed by deterministic SVD in line 10. Finally, the left singular vector $\widetilde{\mathbf{U}}$ and right singular vector $\widetilde{\mathbf{V}}$ are projected back onto $\mathbf{P}$ and $\mathbf{Q}$ to generate the left and right singular vectors of $\mathbf{A}$ in lines 12 and 13, respectively.

The orthogonalizations in lines 4 and 6 of Algorithm 3 ensure that the different columns of $\mathbf{P}$ or $\mathbf{Q}$ converge to different dominant singular vectors. However, the original RSVD algorithm [64] was later improved to skip the orthogonalization of $\mathbf{P}$ [39]. Halko *et al.* [39] used a diverse collection of real applications to illustrate the accuracy and stability of RSVD with skipping the orthogonalization of $\mathbf{P}$. Their test cases included the adaptive range approximation in physics, the graph Laplacian approximation in image processing and the face recognition in machine learning. Similar studies [25, 81] were presented based on the improved algorithm [64]. Thus, the orthogonalization of $\mathbf{P}$ can be skipped depending on the accuracy required by the target application; this improvement is widely accepted for practical applications.

Algorithm 3 indicates that RSVD is dominated by GEMM computation in lines 3 and 5 with access to the large tall-skinny matrix $\mathbf{A}$, which we assume not to fit in the GPU memory. Therefore, a straightforward solution deploys the out-of-core GEMM routines that

Algorithm 3: Multi-pass RSVD. Function orth$(\cdot)$ orthogonalizes the column vectors while functions qr$(\cdot)$ and svd$(\cdot)$ return the QR factorization and deterministic SVD of a matrix, respectively. We use $\widetilde{\mathbf{\Sigma}}(1:k, 1:k)$ and $\widetilde{\mathbf{U}}(:, 1:k)$ to denote the leading $k \times k$ submatrix of $\widetilde{\mathbf{\Sigma}}$ and the submatrix consiting of the first k columns of $\widetilde{\mathbf{U}}$, respectively.

Input : matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, target rank k, oversampling parameter o and power iteration count q.

Output: $\mathbf{\Sigma}$, $\mathbf{U}$ and $\mathbf{V}$ such that $\mathbf{A} \approx \mathbf{U\Sigma V}^\top$ with $k \times k$ diagonal $\mathbf{\Sigma}$, and orthonormal column vectors $\mathbf{U}$ and $\mathbf{V}$.

Generate a random matrix $\mathbf{Q} \sim \mathcal{N}(0,1)^{n \times \ell}$, where $\ell = k + o$.
for $i = 1$ **to** q **do**
  $\mathbf{P} = \mathbf{AQ}$;
  $\mathbf{P} = \text{orth}(\mathbf{P})$; `// if needed`
  $\mathbf{Q} = \mathbf{A}^\top\mathbf{P}$;
  $\mathbf{Q} = \text{orth}(\mathbf{Q})$
end
$\mathbf{P} = \mathbf{AQ}$;
$[\mathbf{P},\ \mathbf{B}] = \text{qr}(\mathbf{P})$;
$\left[\widetilde{\mathbf{U}},\ \widetilde{\mathbf{\Sigma}},\ \widetilde{\mathbf{V}}\right] = \text{svd}(\mathbf{B})$;
$\mathbf{\Sigma} = \widetilde{\mathbf{\Sigma}}(1:k, 1:k)$;
$\mathbf{U} = \mathbf{P}\widetilde{\mathbf{U}}(:, 1:k)$;
$\mathbf{V} = \mathbf{Q}\widetilde{\mathbf{V}}(:, 1:k)$;

offload heavy computation to GPUs. In this case, the amount of CPU-GPU data transfer increases linearly with the number q of power iterations because $\mathbf{A}$ is accessed $2q + 1$ times in Algorithm 3. Thus, RSVD solvers for large data rely on out-of-core GEMM, whose performance fluctuates drastically according to divide-and-conquer strategies. In the next section, we provide theoretical analysis and empirical results to reveal that out-of-core RSVD is bandwidth bound requiring a data-access reduction scheme to minimize the amount of data transfer between the CPU and GPU.

3.4 GPU-Accelerated Out-of-Core GEMM

In this section, we first extend the roofline model [111] to investigate the upper bound of the GPU-accelerated out-of-core GEMM performance. We then propose a 1D partition scheme for tall-skinny matrices and compare our GEMM implementation with several existing implementations.

The following analysis and experiments were conducted in double precision using an experimental system equipped with two Intel 8-core Xeon Silver 4110 CPUs and two NVIDIA Tesla V100 (Volta) GPUs [74]. These CPUs had 96 GB of DDR4-2133 main memory and provided a double precision peak performance of 0.67 Tflop/s in total. Each GPU had 16 GB

of GPU memory with theoretical peak performance of 7.0 Tflop/s in double precision. Each GPU was connected to the CPU via a PCIe 3.0 link allowing bidirectional transfer between the CPU and the GPU. In our system, we observed 6.9 Tflop/s for in-core GEMM computation, and the effective transfer bandwidth per PCIe link was 13.1 GB/s and 12.8 GB/s for CPU-to-GPU and GPU-to-CPU, respectively. The CPU and GPU implementations used Intel MKL 2018.0.3 [48] and cuBLAS 9.2 [76], respectively, for processing in-core GEMM operations. In addition, we used MAGMA 2.4 [2] and LAPACK 3.7 [4] to evaluate the approximation accuracy and initialize multi-threaded computation.

3.4.1 Performance Model

We now propose a performance model based on the roofline model [111] to investigate an upper bound of the GPU-accelerated out-of-core GEMM performance. The roofline model is useful for locating the performance bottleneck, *i.e.*, either arithmetic or memory operations, which limits the entire performance of linear algebra algorithms [36, 110]. Considering the roofline model, the attainable performance in flop/s can be defined as

$$T = \min\left(\frac{F}{D}B,\ C\right), \tag{3.1}$$

where F denotes the number of floating point operations, D denotes the amount of memory access, B denotes the peak memory bandwidth and C denotes the peak computational performance of the target hardware. The term F/D is called operational intensity, which represents the ratio of floating point operations to total data access. The operational intensity is the key algorithmic factor that determines attainable flop/s, whereas the remaining parameters B and C depend on the target hardware.

In the following discussion, we consider DGEMM, $\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$, to investigate the performance of the tall-skinny GEMM, $\mathbf{P} = \mathbf{AQ}$, where $\mathbf{P} \in \mathbb{R}^{m\times\ell}$, $\mathbf{A} \in \mathbb{R}^{m\times n}$ $(m \gg n)$ and $\mathbf{Q} \in \mathbb{R}^{n\times\ell}$, which appears in the power iteration in line 3 of Algorithm 3. Since our focus is on the tall-skinny $\mathbf{A}$, we assume that (1) the large matrix $\mathbf{A}$ must be partitioned into smaller blocks to be processed with a divide-and-conquer strategy, and (2) the small projection matrix $\mathbf{Q}$ fits into the GPU memory and hence is broadcast to all GPUs. In addition, we assume that (3) the input and output matrices exist in the CPU memory; (4) m, n, and ℓ are multiplies of the block dimension b to simplify the discussion; and (5) the DGEMM implementation sends the output matrix $\mathbf{C}$ to the GPU even if $\beta = 0$.

To illustrate the impact of matrix shapes on the out-of-core GEMM performance, we substituted the operational intensity with the matrix size parameters such as m, n, and ℓ. The flop F of GEMM was fixed to $2mn\ell$ for double precision GEMM. By contrast, the memory access cost D was interpreted as CPU-GPU data transfer cost to consider the out-of-core execution on the GPU. Furthermore, the cost D was appropriately selected according to the two data partition schemes as follows.

1. Row-wise 1D partition scheme with block dimension b (Fig. 3.1). According to this scheme, m/b blocks in total must be computed for the matrix $\mathbf{P}$. Each block requires $b \times n$ entries in $\mathbf{A}$ and $n \times \ell$ entries in $\mathbf{Q}$ to compute all entries in the block. Providing

$\mathbf{Q}$ can be reused on the GPU, and $\left(\sum_{j=1}^{m/b} bn\right) + n\ell + m\ell = mn + n\ell + m\ell$ entries are transferred from the CPU to GPU and $m\ell$ entries for the opposite direction. Assuming bidirectional transfer between the CPU and GPU, the data transfer cost can be calculated as $D = mn + n\ell + m\ell$. Therefore, the attainable performance T_1 for the 1D partition scheme can be written as

$$T_1 = \min\left(\frac{2mn\ell}{mn + n\ell + m\ell}B,\ C\right). \tag{3.2}$$

2. 2D partition scheme with the block dimension $b \times b$, where b is the block size. According to this scheme, $m\ell/b^2$ blocks in total must be computed for the matrix $\mathbf{P}$. Each block requires $b \times n$ entries in $\mathbf{A}$ and $n \times b$ entries in $\mathbf{Q}$ to compute all entries in the block. Providing $\mathbf{Q}$ can be reused on the GPU, and $\left(\sum_{j=1}^{m\ell/b^2} bn\right) + n\ell + m\ell = mn\ell/b + n\ell + m\ell$ entries are transferred from the CPU to GPU and $m\ell$ entries for the opposite direction. Assuming the bidirectional transfer scheme mentioned above, the data transfer cost can be calculated as $D = mn\ell/b + n\ell + m\ell$. The attainable performance T_2 for the 2D partition scheme can be written as

$$T_2 = \min\left(\frac{2mn\ell}{mn\ell/b + n\ell + m\ell}B,\ C\right). \tag{3.3}$$

Equations (3.2) and (3.3) can be further simplified by considering the shape of the matrix $\mathbf{Q}$. In more detail, the parameter ℓ can be eliminated for the following two cases: (1) square matrix ($\ell = n$) and (2) tall-skinny matrix ($\ell = n/10$, for example). After this elimination, Eqs. (3.2) and (3.3) can be rewritten as functions of m and n. Consequently, the performance upper bound can be shown on a 2D heatmap, where the vertical and horizontal axes are the matrix dimensions m and n of $\mathbf{A}$, respectively (Fig. 3.2).

Figure 3.2 clearly demonstrates that a higher performance illustrated as a red area can be expected only with the 1D partition scheme. With respect to the 2D partition scheme shown in Figs. 3.2(c) and 3.2(d), the performance upper bounds are strictly limited for both the square and tall-skinny shapes of the matrix $\mathbf{Q}$ with less than 1 Tflop/s due to the narrow bandwidth of the CPU-GPU data transfer. The same limitation can also be found for the 1D partition scheme; however, the maximum performance reached up to 7 Tflop/s in this case. Another remarkable point here is that the shape of the matrix $\mathbf{A}$ also strongly impacts the performance upper bound if the matrix $\mathbf{Q}$ is tall-skinny, which is demonstrated by a larger blue area in Fig. 3.2(b) compared with that in Fig. 3.2(a). Thus, the transfer bandwidth between the CPU and GPU limits the GPU-accelerated out-of-core GEMM performance for the tall-skinny $\mathbf{A}$ ($m \gg n$), which frequently appears in big data analytics.

In summary, we make the following observations about the extended performance model.

- Row-wise 1D data partition is a promising solution for the GEMM operations of large tall-skinny matrices because this solution minimizes the amount of CPU-GPU data transfer.

- 2D data partition inevitably increases the amount of CPU-GPU data transfer limiting the out-of-core GEMM performance. The performance will deteriorate, especially for the tall-skinny GEMM operations, which are the main focus of our research.

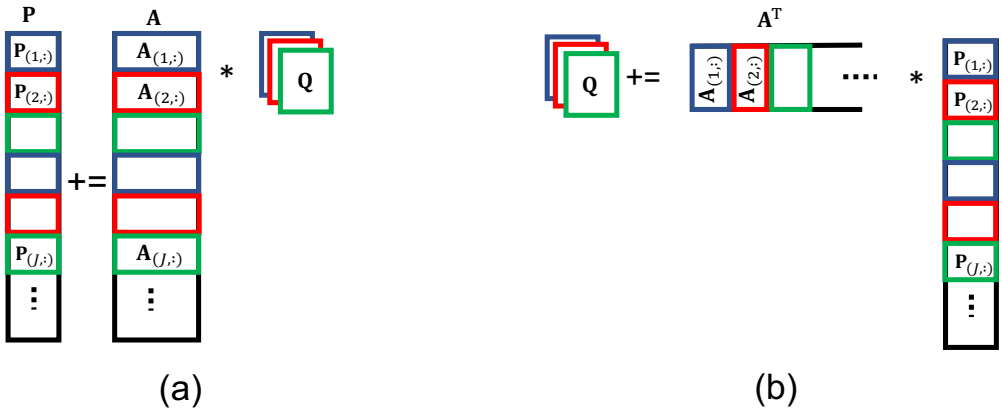

Figure 3.1: Proposed row-wise 1D partition scheme for tall-skinny GEMM. (a) $\mathbf{P} = \mathbf{AQ}$ (line 3 of Algorithm 3), where $\mathbf{A}$ is partitioned into blocks and $\mathbf{Q}$ is broadcasted among GPUs. (b) $\mathbf{Q} = \mathbf{A}^\top\mathbf{P}$, where $\mathbf{Q}$ is accumulated by reduction. Blocks are assigned to CUDA streams [77] in a round-robin fashion (*i.e.*, block-cyclic distribution as illustrated with different colors). Because $\mathbf{P}$ is computed in a block-wise manner, the GPU is allowed to store the part of $\mathbf{P}$.

3.4.2 Row-wise 1D Partition Scheme for Out-of-Core GEMM

We propose a row-wise 1D partition scheme for the two GEMM operations applied to tall-skinny matrices in lines 3 and 5 of Algorithm 3. We use row-wise partition rather than column-wise partition for the following two reasons.

- Lower data transfer cost. The outer-product update of the matrix $\mathbf{P}$ results in a large amount of CPU-GPU data transfer for each GEMM computation if the tall-skinny $\mathbf{A}$ is partitioned into 1D column blocks. Furthermore, the column-wise partition requires multiple buffers and a complicated synchronization mechanism to accumulate the updates from different blocks. In contrast, the row-wise partition allows per-block GEMM operations to be data-independent without a race condition. This asynchronous property enables the pipelined execution of partitioned blocks, where data transfers are efficiently overlapped with GEMM computation. In more detail, software pipelining can be implemented using CUDA streams [77].

- Higher GEMM performance. GEMM routines generally run faster for square matrices rather than tall-skinny matrices. Compared with the column-wise partition, the row-wise partition generates square-like blocks, which are useful for maximizing the performance of per-block GEMM operations.

Figure 3.1 illustrates how we apply our 1D partition scheme to the GEMM operations in the RSVD algorithm. As shown in Fig. 3.1(a), we partition the tall-skinny matrices $\mathbf{A}$ and $\mathbf{P}$ into blocks for the first GEMM, $\mathbf{P} = \mathbf{AQ}$. Given the block size b, this partition scheme generates m/b blocks for each $\mathbf{A}$ and $\mathbf{P}$, each having dimensions of $b \times n$ and $b \times \ell$, respectively. The small $n \times \ell$ matrix $\mathbf{Q}$ is initialized on the CPU and then broadcast to all GPUs to allow them to independently call the cuBLAS routines for applying the in-core GEMM operations to blocks. The second GEMM, $\mathbf{Q} = \mathbf{A}^\top\mathbf{P}$, is based on an outer-product of $\mathbf{Q}$. Similar to the process illustrated in Fig. 3.1(a), we broadcast $\mathbf{Q}$ to all GPUs for

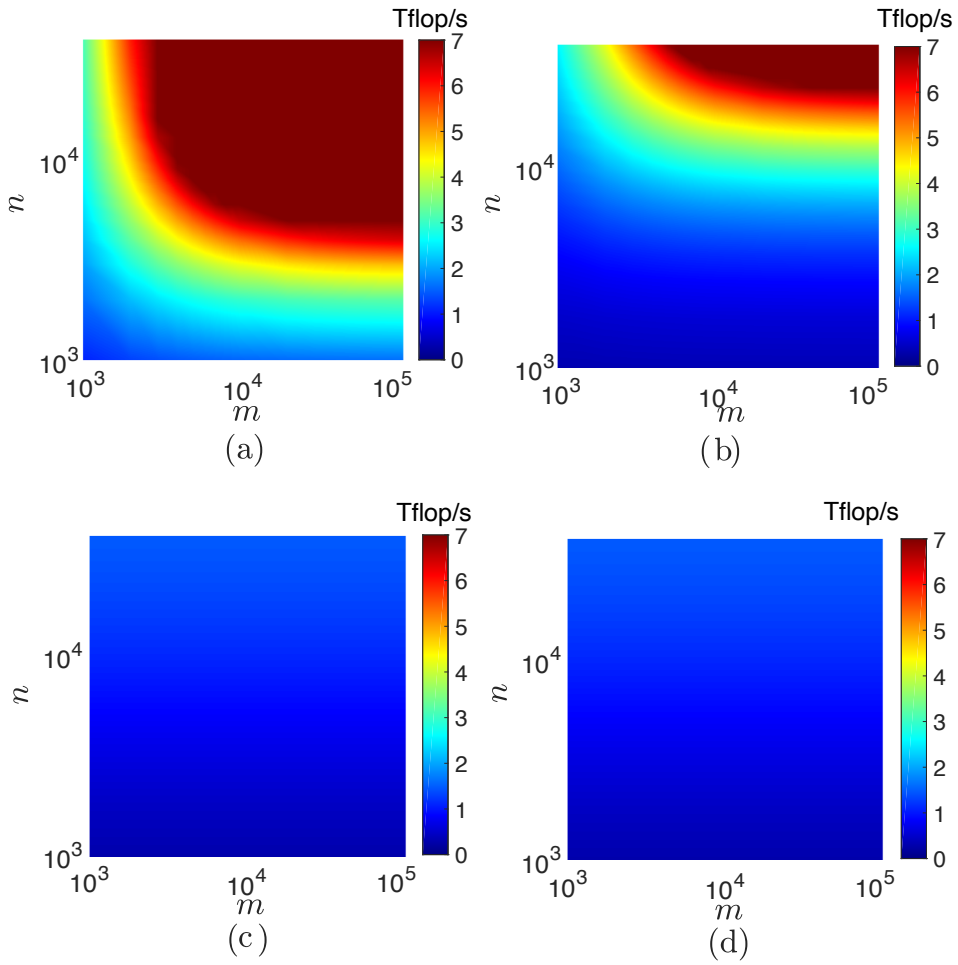

Figure 3.2: Performance upper bound of GPU-accelerated out-of-core GEMM: $\mathbf{P} = \mathbf{AQ}$, where $\mathbf{P} \in \mathbb{R}^{m \times \ell}$, $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{Q} \in \mathbb{R}^{n \times \ell}$. Row-wise 1D partitioning results for (a) square ($\ell = n$) and (b) tall-skinny matrices $\mathbf{Q}$ ($\ell = n/10$). 2D partition results for (c) square ($\ell = n$) and (d) tall-skinny matrices $\mathbf{Q}$ ($\ell = n/10$). Hardware specific parameters were set for the Tesla V100 GPU: $B = 13$ GB/s and $C = 7$ Tflop/s. Both partition schemes used the block size b of 1024, which was the default setup of cuBLAS-XT [76].

processing blocks in parallel (Fig. 3.1(b)). A reduction of small $\mathbf{Q}$ is followed after finishing all GEMM operations.

As mentioned above, GEMM is not universally efficient for tall-skinny matrices. This low efficiency is due to the computation of tall-skinny GEMM, which is closer to GEMV (BLAS2 routines) than GEMM (BLAS3 routines). The BLAS2 routines are less efficient than the BLAS3 routines due to vector accesses that degrade cache hit rate on both the multi-core CPU and GPU; BLAS3 routines are 20–40 times more efficient than BLAS2 routines [18, 31]. Regarding the in-core performance of tall-skinny GEMM, Chen *et al.* [11] achieved 1.1–3.0× speedups over cuBLAS for tall-skinny matrices with up to 16 columns. Their GEMM solution can be easily integrated into our RSVD solver, but the maximum number of columns is limited by 16, mainly due to the limitation on computational resources, such as register files.

In the following discussion, we use the sub-matrix notation [34, 39] to denote the block matrix; each block of $\mathbf{A}$ is expressed as $\mathbf{A}_{(J,:)} \in \mathbb{R}^{b \times n}$, where J is an ordered set of indices

defined as $J = [jb, jb + 1, \ldots, jb + b - 1]$ for the j-th block ($j \geq 0$).

3.4.3 Performance Tuning and Comparison of Out-of-Core GEMM

We first tuned our out-of-core GEMM implementation with respect to the block size b. Figure 3.3 illustrates the measured performance with different block sizes, ranging from 512 to 8192. The block size was also maximized to measure the performance without data partition; due to the lack of data partition, we failed to execute large matrices of $m \geq 2 \times 10^5$ entries. Similarly, we measured the performance of a multi-threaded CPU implementation for reference. All except one of the setups allocated matrices in the pinned CPU memory. Pinned CPU memory gives higher performance for CPU-GPU data transfer in GPU programming [77].

Figure 3.3 demonstrates that the pinned memory was faster than the pageable memory; the effective bandwidth from the pinned memory to the GPU memory was 1.7 times higher than that from the pageable memory. Hence, we allocated matrices in the pinned memory for the rest of the experiments. We also observed that the performance for the maximum block size ($b = \max$) was significantly degraded due to the lack of overlapped execution; there were no partitioned blocks that could be processed in the pipeline. Therefore, data partition schemes are necessary to maximize the performance of the out-of-core GEMM operations.

As mentioned in Section 3.4, the in-core GEMM computation ran at 6.9 Tflop/s, which was close to the theoretical peak of 7.0 Tflop/s. With respect to the out-of-core GEMM computation, Fig. 3.3 demonstrates that there is some margin between the theoretical upper bound and the measured results. This margin occurred due to data partition, which applies GEMM operations to the partitioned blocks. In other words, partitioned blocks are not sufficient large to maximize the effective performance; in-core GEMM runs without such data partition. However, small blocks are required to overlap CPU-GPU data transfer with GPU computation.

As for the block size, a trade-off point must be found to maximize the performance of 1D partition scheme. For small block sizes of $b \leq 2048$, we observed only 8–11 GB/s of CPU-GPU transfer bandwidth and 8.3–9.3 Gflop/s of arithmetic performance, because the parallelism in each small block was not sufficient to achieve the maximum efficiency on the GPU. On the other hand, the maximum block size $b = \max$ resulted in a poor performance due to inefficient execution of the pipeline. Weighing the trade-offs here, we selected $b = 4096$ for the following experiments.

We now compare and contrast the performance of out-of-core GEMM implementation against that obtained with previous out-of-core GEMM implementations: cuBLAS-XT [76] and BLASX [109]. Figure 3.4 illustrates the performance comparison when using a single Tesla V100 GPU. Among these implementations, the proposed GEMM was the fastest in most cases. Note that cuBLAS-XT was evaluated with the following three setups: (1) 2D partition with pinned memory (default); (2) 1D partition with pinned memory; and (3) 1D partition with Unified Memory. As for 1D partition, we set the block size as $b = n$, which enforced the row-wise 1D block partition. Among these setups, the highest performance was obtained when using the second setup, *i.e.*, 1D partition with pinned memory. The default setup failed to achieve high performance for the out-of-core GEMM operations; this behavior is consistent with Eqs. (3.2) and (3.3), which imply that 2D partition transfers more entries

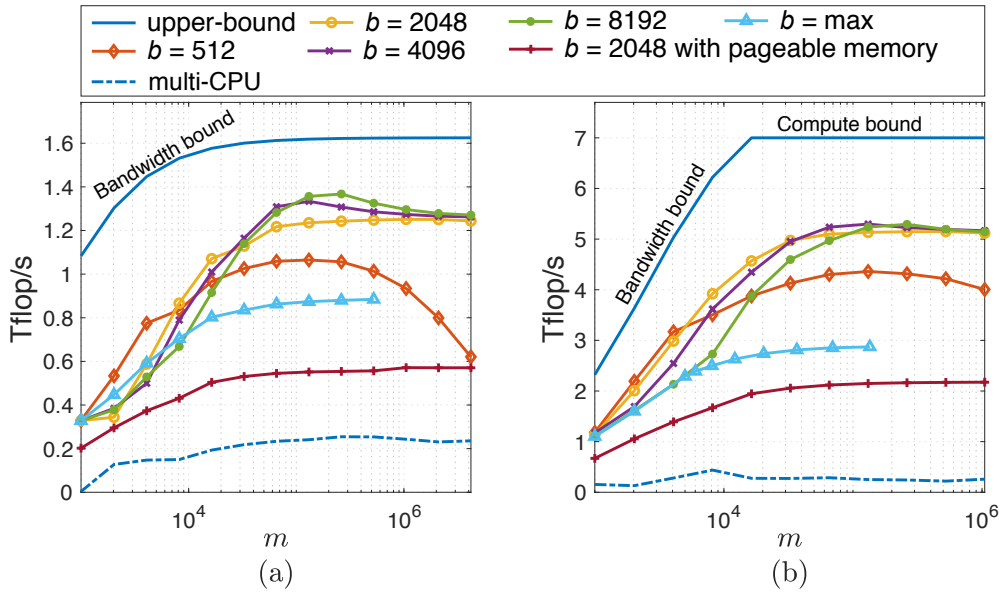

Figure 3.3: Out-of-core GEMM performance with different block sizes b on a single Tesla V100 GPU. Measured results for (a) small ($\ell = n = 1000$) and (b) large square matrices ($\ell = n = 5000$). The maximum block size ($b = \max$) indicates the performance without data partition. A multithreaded CPU version was also evaluated on two 8-core CPUs. Note that the upper-bound line is curved because the horizontal axis of our extended model is the height of matrix $\mathbf{A}$, which is different from that (*i.e.*, the operational intensity) of the original roofline model.

than 1D partition. Therefore, excessive data transfers saturated the CPU-GPU bandwidth, resulting in a lower performance. While enforced 1D partition significantly increased the performance, there was still a performance gap compared with that of the proposed GEMM. The advantage of the proposed GEMM implementation comes from manual broadcast of the small matrix $\mathbf{Q}$ to GPUs (Fig. 3.1(a)), which reduces the amount of CPU-GPU data transfer for all block GEMM operations. We obtained similar results for another GEMM operation in the power method ($\mathbf{Q} = \mathbf{A}^\top \mathbf{P}$ in Fig. 3.1(b)).

With Unified Memory, we failed to increase the performance for larger m even with 1D partition. We speculate that Unified Memory failed to efficiently deal with the complicated memory access pattern. It is not easy to automate CPU-GPU data transfer without explicitly managing the memory. Finally, BLASX performed stably for all setups (Fig. 3.4). However, the maximum matrix size was limited to $m \leq 2 \times 10^6$ entries; matrices larger than the maximum size resulted in an execution failure without any error message.

We also evaluated the speedups of the considered implementations on two Tesla V100 GPUs (Fig. 3.5). For each implementation, we used the same implementation as the baseline, which ran on a single V100 GPU. All implementations except Unified Memory demonstrated increased speedups as m grows; the speedups reached around 1.8× for $m \geq 2 \times 10^4$ demonstrating a scalable performance on two GPUs. By contrast, small matrices of $m < 1 \times 10^4$ resulted in low speedups, due to the overheads of GPU initialization that took up around 30% of the overall execution time. Similarly, cuBLAS-XT with Unified Memory degraded the GEMM performance when using two GPUs.

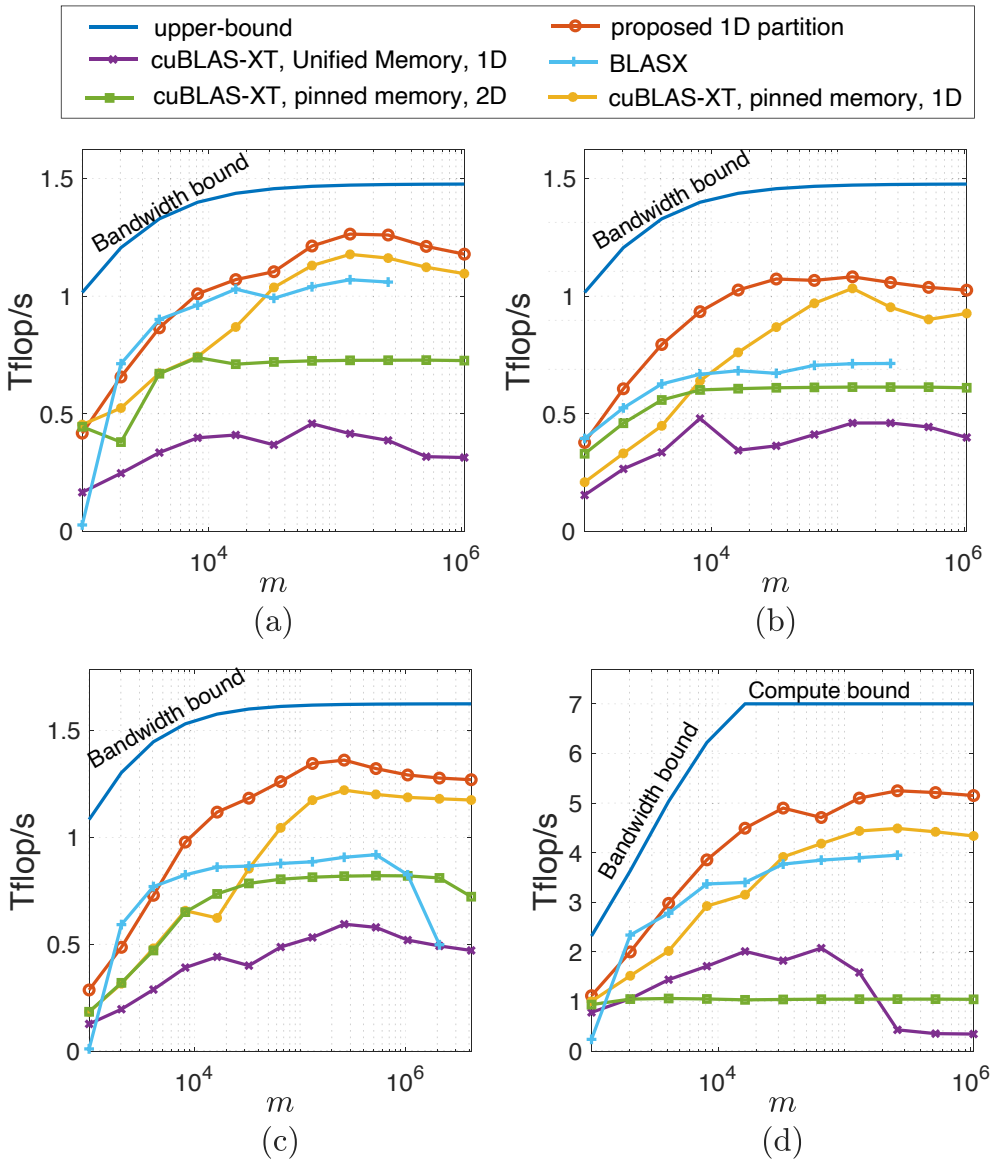

Figure 3.4: Performance comparison of the out-of-core GEMM implementations on a single Tesla V100 GPU. Results with different shapes for the matrix $\mathbf{Q}$: (a) tall-skinny ($n = 5000, \ell = 500$), (b) short-wide ($n = 500, \ell = 5000$), (c) small square ($\ell = n = 1000$) and (d) large square ($\ell = n = 5000$). BLASX is a high-level library that hides specific data partition and memory allocation methods.

3.5 Proposed Out-of-Core RSVD Methods

We first describe the basic scheme that uses 1D partition for out-of-core RSVD computation. We then present a FIFO scheme that reduces the amount of CPU-GPU data transfer by employing the reverse iteration. We further elaborate on two methods, namely Fused and Gram, which are our main contribution to this work. Both the Fused and Gram methods are built upon the basic and FIFO schemes.

Table 3.1 summarizes the computational and communication costs of all proposed vari-

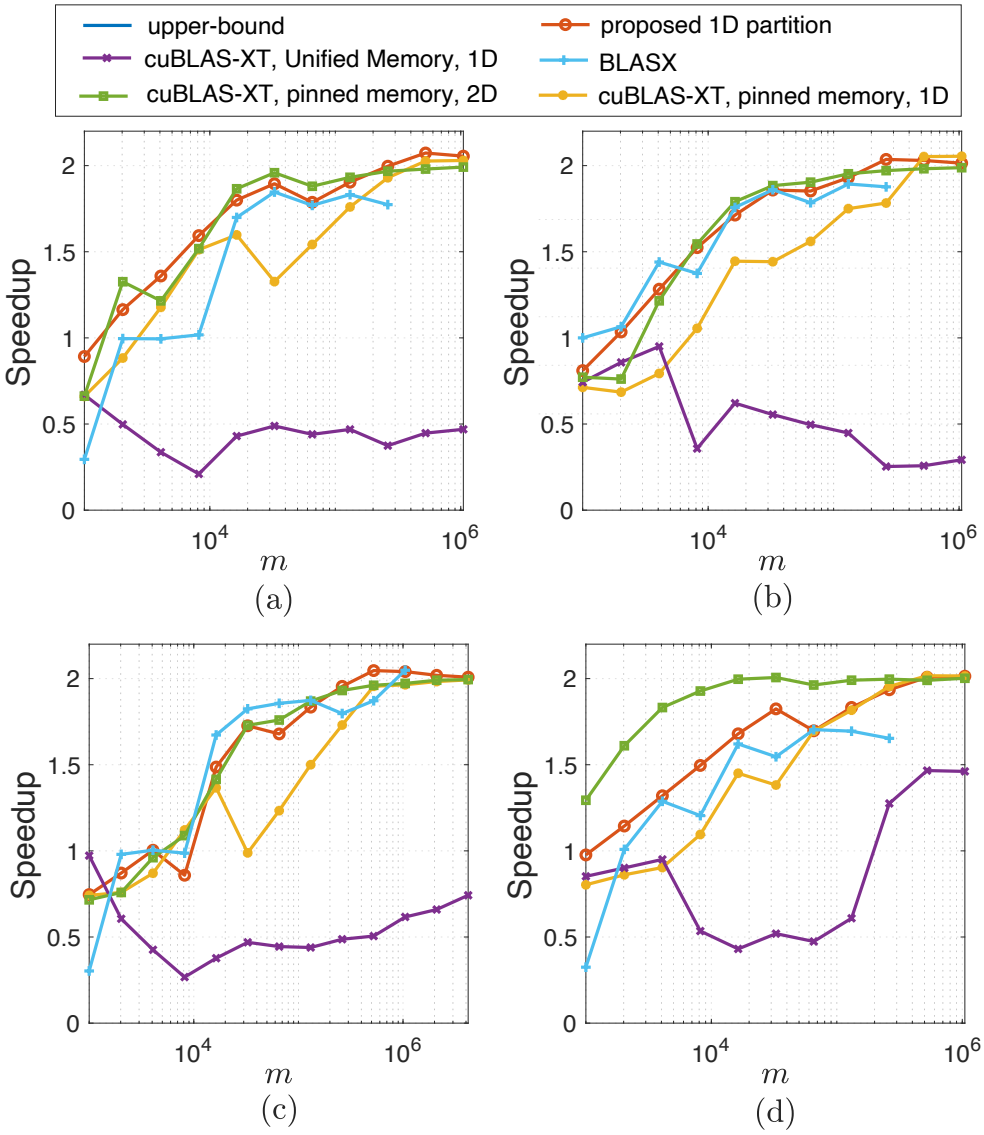

Figure 3.5: Speedup of the out-of-core GEMM implementations on two Tesla V100 GPUs. Results with different shapes for the matrix $\mathbf{Q}$. (a) tall-skinny ($n = 5000, \ell = 500$), (b) short-wide ($n = 500, \ell = 5000$), (c) small square ($\ell = n = 1000$) and (d) large square ($\ell = n = 5000$). For each implementation, we executed the same implementation on a single GPU to compute the speedup.

ations. This table considers only the power method, which is the main contribution of the chapter. As shown in Table 3.1, our main concern is to reduce the amount of CPU-GPU data transfer, which limits the performance of out-of-core computation on the GPU. We also show a pipelined mechanism to overlap kernel execution with data transfer. As for kernel optimization, our approach is to deploy vendor's optimized GEMM kernels with tuned execution setups, as investigated in Section 3.4.3.

Table 3.1: Comparison of the proposed methods in terms of computational cost, the number of data passes, and CPU-GPU data transfer cost. The number of data passes and CPU-GPU data transfer cost correspond to in-core access cost and out-of-core access cost, respectively. We consider matrix $\mathbf{A}$ to evaluate the number of data passes. Both the Fused and Gram methods adopt the FIFO scheme to reduce the CPU-GPU data transfer cost.

Method	F: # of floating point operations	# of data passes	D: CPU-GPU data transfer cost
Basic (data-access reduction)	$(2q+1)mn\ell$	$2q+1$	$(2q+1)mn$
FIFO (reverse iteration)	$(2q+1)mn\ell$	$2q+1$	$< (2q+1)mn$
Fused	$(2q+1)mn\ell$	$2q+1$	$< (q+1)mn$
Gram	$mn^2 + qn^2\ell + mn\ell$	3	$< 2mn$

3.5.1 Basic and FIFO Schemes for Reducing Out-of-Core Data Access

As summarized in Table 3.1, a straightforward divide-and-conquer implementation of RSVD passes $\mathbf{A}$ for $2q+1$ times with the proposed 1D scheme, where $\mathbf{A}$ and $\mathbf{P}$ are partitioned, whereas $\mathbf{Q}$ is broadcast to all GPUs (Fig. 3.1). The QR factorization of $\mathbf{P}$ in line 9 of Algorithm 3 is computed using Cholesky factorization [96] shown in Algorithm 4. In Algorithm 4, the GEMM operation at line 4 can be processed by calling the GEMM function, such as the `cublasDgemm()` kernel of the cuBLAS library. Algorithm 4 only transfers small $\ell \times \ell$ matrices $\mathbf{B}$ and $\mathbf{R}$ between the CPU and GPU because Algorithm 3 stores the input matrix $\mathbf{P}$ in the GPU memory (at line 8) before processing the QR factorization. Note that the data transfers are omitted in Algorithm 4 to simplify its description. Both SVD and QR procedures in line 10 of Algorithm 3 and line 6 of Algorithm 4, respectively, can be implemented using the standard LAPACK routines. We denote this implementation as the basic scheme.

The basic scheme can be easily improved by reorganizing the loop structure. Algorithm 5 presents the FIFO scheme that requires less access to $\mathbf{A}$. As shown in lines 7–9, the execution order of iterations for computing $\mathbf{Q}$ is reversed such that blocks of $\mathbf{A}_{(J,:)}$ from the first GEMM in line 5 can be reused for that for the second GEMM in line 8; the data transfer of $\mathbf{A}_{(J,:)}$ occurs only before the first GEMM in line 5, where the CUDA kernel is invoked from the CPU. As compared with the basic scheme, this data reuse on the GPU reduces the amount of CPU-GPU data transfer; at the same time, the reduced amount depends on the number of blocks that can be stored at once in the GPU memory (Table 3.1). In addition to this data reuse, blocks can be further reused across different iterations, *i.e.*, the second GEMM at the current i-th loop can be reused for the first GEMM at the next $(i+1)$-th loop. Note that the worst case of $mn + 2q(m-b)n$ occurs when the GPU memory can hold only a single block of $\mathbf{A}_{(J,:)}$; all blocks of the total size mn are sent to the GPU at the first data access to $\mathbf{A}$ ($i = 1$, lines 4–6), then a block of $b \times n$ entries is reused $2q$ times by the following GEMM. By contrast, the best case of mn can be obtained when the GPU memory can hold all blocks of $\mathbf{A}_{(J,:)}$; however, this contradicts to our assumption that data size exceeds the GPU memory capacity.

Algorithm 4: QR Factorization. Function chol(·) returns the Cholesky factorization of a matrix.

Input : $\mathbf{P} \in \mathbb{R}^{m \times \ell}$.
Output: $\mathbf{P} \in \mathbb{R}^{m \times \ell}$ and $\mathbf{R} \in \mathbb{R}^{\ell \times \ell}$.

1 $\mathbf{B} = \mathbf{0}^{\ell \times \ell}$; // Initialize $\mathbf{B}$ with all 0
2 $s = m/b$; // # of blocks
3 **for** $j = 1$ **to** s **do**
4 $\quad \mathbf{B} \mathrel{+}= \mathbf{P}_{(J,:)}^{\top} \mathbf{P}_{(J,:)}$
5 **end**
6 $\mathbf{R} = \text{chol}(\mathbf{B})$; // factored on CPU
7 **for** $j = 1$ **to** s **do**
8 $\quad \mathbf{P}_{(J,:)} = \mathbf{P}_{(J,:)} \mathbf{R}^{-1}$
9 **end**

Algorithm 5: FIFO scheme. This method replaces lines 2–8 of Algorithm 3.

Input : $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{Q} \in \mathbb{R}^{n \times \ell}$, block size b and power iteration count q.
Output: $\mathbf{P} \in \mathbb{R}^{m \times \ell}, \mathbf{Q} \in \mathbb{R}^{n \times \ell}$ and $\mathbf{B} \in \mathbb{R}^{\ell \times \ell}$.

1 $\mathbf{B} = \mathbf{0}^{\ell \times \ell}$; // Initialize $\mathbf{B}$ with all 0
2 $s = m/b$; // # of blocks
3 **for** $i = 1$ **to** q **do**
4 $\quad$ **for** $j = 1$ **to** s **do**
5 $\quad\quad \mathbf{P}_{(J,:)} = \mathbf{A}_{(J,:)} \mathbf{Q}$;
6 $\quad$ **end**
7 $\quad$ **for** $j = s$ **to** 1 **do**
8 $\quad\quad \mathbf{Q} \mathrel{+}= \mathbf{A}_{(J,:)}^{\top} \mathbf{P}_{(J,:)}$; // reuse 1D blocks
9 $\quad$ **end**
10 $\quad [\mathbf{Q}, \sim] = \text{qr}(\mathbf{Q})$;
11 **end**
12 **for** $j = 1$ **to** s **do**
13 $\quad \mathbf{P}_{(J,:)} = \mathbf{A}_{(J,:)} \mathbf{Q}$;
14 $\quad \mathbf{B} \mathrel{+}= \mathbf{P}_{(J,:)}^{\top} \mathbf{P}_{(J,:)}$;
15 **end**

3.5.2 Fused Method for Reducing Out-of-Core Data Access

The amount of CPU-GPU data transfer can be further reduced by taking the advantage of the fact that the orthogonalization of $\mathbf{P}$ in line 4 of Algorithm 3 can be omitted in many practical applications, as mentioned in Section 3.3. Algorithm 6 shows the Fused method that skips the orthogonalization of $\mathbf{P}$. Consequently, the two block GEMM operations in lines 5 and 8 of Algorithm 5 can be processed in a single Fused loop, as shown in lines 6–7 and 11–12 of Algorithm 6. After this loop fusion, the two GEMM operations are performed with the same block $\mathbf{A}_{(J,:)}$ to compute $\mathbf{P}_{(J,:)}$ and $\widetilde{\mathbf{Q}}$ before accessing the next block; the `if-else` statement in lines 4–14 is used to process $\mathbf{A}_{(J,:)}$ in inverse order such that the

Algorithm 6: Fused method. This method replaces lines 2–11 of Algorithm 5.

Input : $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{Q} \in \mathbb{R}^{n \times \ell}$, block size b and power iteration count q.
Output: $\mathbf{P} \in \mathbb{R}^{m \times \ell}$ and $\mathbf{Q} \in \mathbb{R}^{n \times \ell}$.

1 $\widetilde{\mathbf{Q}} = \mathbf{0}^{n \times \ell}$; `// Initialize` $\widetilde{\mathbf{Q}}$ `with all 0`
2 $s = m/b$; `// # of blocks`
3 **for** $i = 1$ **to** q **do**
4 **if** $i\%2 == 1$ **then**
5 **for** $j = 1$ **to** s **do**
6 $\mathbf{P}_{(J,:)} = \mathbf{A}_{(J,:)}\mathbf{Q}$;
7 $\widetilde{\mathbf{Q}} \mathrel{+}= \mathbf{A}_{(J,:)}^{\top}\mathbf{P}_{(J,:)}$; `// reuse 1D blocks`
8 **end**
9 **else**
10 **for** $j = s$ **to** 1 **do**
11 $\mathbf{P}_{(J,:)} = \mathbf{A}_{(J,:)}\mathbf{Q}$;
12 $\widetilde{\mathbf{Q}} \mathrel{+}= \mathbf{A}_{(J,:)}^{\top}\mathbf{P}_{(J,:)}$; `// reuse 1D blocks`
13 **end**
14 **end**
15 $[\mathbf{Q}, \sim] = \mathrm{qr}(\widetilde{\mathbf{Q}})$;
16 **end**

blocks can be reused in the next i loop. Consequently, the matrix $\mathbf{A}$ is transferred only once in every iteration, and thus, the amount of CPU-GPU data transfer is reduced to $(q+1)mn$. Combined with the FIFO scheme, the worst case of $mn + q(m-b)n$ occurs when only a single block of $\mathbf{P}_{(J,:)}$ is reused, whereas the best case of mn is obtained when all blocks are reused at all GEMM operations.

Recall here that each block GEMM operation is processed by calling the GEMM kernel, which runs on the GPU. Therefore, the `if-else` statement in Algorithm 6, which exists outside the CUDA kernel function, is executed on the CPU. Consequently, there is no concern on warp divergence issues [77], which degrade the performance on the GPU.

The Fused method requires an additional memory space for storing $\widetilde{\mathbf{Q}}$, which has the same size as the sampling matrix $\mathbf{Q}$. However, this additional cost is negligible because the size of $\mathbf{Q}$ is assumed to be small ($\ell \times \ell$). With respect to the number of data passes, the Fused method requires the same number $2q + 1$ of data passes as the basic scheme.

3.5.3 Gram Method for Reducing In-Core and Out-of-Core Data Access

Similar to the Fused method, the Gram method skips the orthogonalization of $\mathbf{P}$ in line 4 of Algorithm 3. Therefore, every power iteration computes the following equation:

$$\mathbf{Q} = \mathrm{orth}(\mathbf{A}^{\top}(\mathbf{A}\mathbf{Q})). \tag{3.4}$$

We explicitly form the Gram matrix as $\mathbf{G} = \mathbf{A}^\top\mathbf{A} \in \mathbb{R}^{n\times n}$, and thereby, Eq. (3.4) is mathematically equivalent to the following equation:

$$\mathbf{Q} = \text{orth}(\mathbf{A}^\top(\mathbf{AQ})) = \text{orth}((\mathbf{A}^\top\mathbf{A})\mathbf{Q}) = \text{orth}(\mathbf{GQ}). \tag{3.5}$$

As mentioned in Section 3.2.1, forming the Gram matrix is usually avoided due to its high computation cost; however, we do that to reduce the number of data passes, *i.e.*, the amount of in-core and out-of-core data access.

Algorithm 7 lists a pseudocode of our proposed Gram method. The power method is applied to the Gram matrix $\mathbf{G}$ without accessing $\mathbf{A}$ in lines 6–9 of Algorithm 7, so that the number of data passes is reduced to 3, which is independent from q. These q-independent passes prevent the performance degradation if some data present a slow singular decay pattern that requires more power iterations to form the approximation [39]. In this case, the communication cost with more passes to $\mathbf{A}$ incurs a tremendous burden on the narrow CPU-GPU bandwidth.

For 1D partition of $\mathbf{A}$, GEMM operations for forming $\mathbf{G}$ (lines 3–5 of Algorithm 7) can be processed with the data transfer cost of mn. However, the Gram method increases the number of floating-point operations from $(2q+1)mn\ell$ to $mn^2 + qn^2\ell + mn\ell$; mn^2 and $qn^2\ell$ correspond to lines 3–5 and 6–9 of Algorithm 7, respectively. Despite this extra computation, the Gram method reduces the amount of CPU-GPU data transfer to less than $2mn$ when combined with the FIFO scheme (Table 3.1). Considering the large gap between the communication cost and computational cost, we believe that forming $\mathbf{G}$ reduces the overall run time at the expense of increased floating-point operations. We experimentally validate this assumption in Section 3.6.

3.5.4 Implementation details

We implemented all of the proposed methods with the underlying GEMM operations tuned in Section 3.4.3. In addition, we integrated the following techniques into our RSVD solver.

Kernel optimization: The MAGMA library [2], which wraps the CUDA library, was deployed for all GEMM operations, GPU initialization, memory management, and data transfer. The MAGMA library assumes that (1) matrices are stored in column-major format and (2) the leading dimension of matrices are round up to multiples of 32. These assumptions are useful for maximizing effective memory bandwidth by achieving memory access coalescing [77] on the GPU. For QR and deterministic SVD computations on the CPU, we used the `potrf()` and `gesvd()` routines included in LAPACK [4], respectively.

Software pipeline: A double buffering approach was implemented to realize a software pipeline mechanism to overlap CUDA kernel execution with CPU-GPU data transfer. In more detail, our solver creates two CUDA streams [77] per GPU. Each stream, wrapped inside the MAGMA queue structure [2], runs asynchronously so that overlapping can be achieved to maximize the entire performance. For the GEMM operation at line 5 of Algorithm 5, a CUDA stream calls a single cuBLAS CUDA kernel on a buffer while another stream executes data transfer on another buffer.

Multi-GPU execution: Our solver assigns matrix blocks to GPUs in a round-robin fashion. For the FIFO and Fused methods, each GPU obtains intermediates of $\mathbf{B}$ and $\mathbf{Q}$

Algorithm 7: Gram method. This method replaces line 1–11 of Algorithm 5.

Input : $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{Q} \in \mathbb{R}^{n \times \ell}$ and block size b.
Output: $\mathbf{Q} \in \mathbb{R}^{n \times \ell}$.

1 $\mathbf{G} = \mathbf{0}^{n \times n}$; `// Initialize G with all 0`
2 $s = m/b$; `// # of blocks`
3 **for** $j = 1$ **to** s **do** `//` $\mathbf{G} = \mathbf{A}^\top \mathbf{A}$: `form Gram matrix with 1D partition of A`
4 $\quad \mathbf{G} \mathrel{+}= \mathbf{A}^\top_{(J,:)} \mathbf{A}_{(J,:)}$
5 **end**
6 **for** $i = 1$ **to** q **do**
7 $\quad \mathbf{Q} = \mathbf{G}\mathbf{Q}$;
8 $\quad [\mathbf{Q}, \sim] = \mathrm{qr}(\mathbf{Q})$;
9 **end**

after processing assigned blocks. The solver then transfers $\mathbf{Q}$ to the CPU to process the QR factorization with the `potrf()` routine on the CPU. Finally, the CPU calls the `axpy()` routine to form $\mathbf{B}$ and $\mathbf{Q}$ from all the intermediates. A similar approach was used to obtain matrix $\mathbf{G}$ for the Gram method. We used a single GPU to compute the power iteration process in lines 6–9 of Algorithm 7 because $\mathbf{Q}$ and $\mathbf{G}$ are small enough to fit into a single GPU memory.

3.6 Experimental Results

We now evaluate the proposed methods in terms of the performance and numerical stability. In the following discussion, the CPU-GPU data transfer time was taken into account for the measured performance in flop/s.

We also compare the proposed methods with the previous methods. We implemented a CPU-based method using the LAPACK library for reference. The CPU-based method was multi-threaded using OpenMP directives [84]. Intel MKL 11.3.1 was linked to LAPACK for BLAS routines. We thoroughly tuned the solver to gain the highest performance on our experimental CPUs. All implementations were compiled using GNU C++ 7.4.0 and CUDA 10.1.

All experiments were conducted in double precision using two Intel Xeon Silver 4114 CPUs and two NVIDIA Tesla V100 GPUs. These CPUs had 384 GB of DDR4-2666 main memory and provided a double precision peak performance of 0.9 Tflop/s in total.

3.6.1 Performance Evaluation

We applied the proposed schemes to the basic scheme step by step to investigate the performance impact of each scheme; (1) the basic method (the basic scheme in Section 3.5.1), (2) the FIFO method (the FIFO scheme in Section 3.5.1), (3) the Fused method (Section 3.5.2), and (4) the Gram method (Section 3.5.3). For the power iteration count, we used $q = 1, 4,$ and 8, which covers from low to high accuracy approximation. Note that with $q = 4$, RSVD

achieved almost the same level of accuracy as the deterministic SVD (Section 3.6.3). Figure 3.6 shows the RSVD performance measured with different numbers of rows m. In Fig. 3.6, the FIFO method slightly reduced the overall runtime by approximately 10%. Furthermore, the Fused method increased the basic performance by reducing the amount of CPU-GPU data transfer D from $(2q+1)mn$ to less than $(q+1)mn$, which corresponds to at most 33%, 42%, and 44% reductions for $q = 1$, 4, and 8, respectively. Accordingly, the overall execution time was reduced by 24.2%, 37.8%, and 42.5% for $q = 1$, 4, and 8, respectively. Thus, the gap between the reduction rate on execution time and that on data transfer amout became closer as we increased q. To understand this behavior, we investigated the time breakdowns of the Fused method; the proportion other than GEMM operations dropped from 12.5% to 6.7%, which implies that reducing data transfer amount increased its impact as q increased. Overall, the Fused method achieved up to 1.7× speedup over the basic method.

Regarding the performance of the Gram method, the execution time for $m = 9.2 \times 10^5$ remained at approximately 12 s, which was independent of q. In fact, the proportions of GEMM in the Gram method maintained at 93% from $q = 1$ to $q = 8$ with $m = 9.2\times10^5$. With $q = 1$, the Gram method performed slightly worse than the Fused method due to increased computation cost. Comparatively, the execution time of the Fused method increased from 12.2 s in Fig. 3.6(a) to 38.2 s in Fig. 3.6(b) with $m = 9.2\times10^5$. With $q = 8$ and $m = 9.2\times10^5$ in Fig. 3.6(c), the execution times of the Gram and Fused methods were 12.7 s and 38.2 s, respectively. This 3.0× speedup of the Gram method was achieved by further reducing D from $(q+1)mn$ to $2mn$ via Gram matrix computation. Overall, the Gram method improved the performance by up to 5.2× over the basic method.

Figure 3.7 shows the speedup of one GPU over two CPUs with different numbers of rows m. The speedup gradually increased for all methods as we increased q, because the proportion of GEMM increased with q and the GPU performed better than CPUs for GEMM operations. The Gram method achieved up to 70× speedup over two CPUs with $q = 8$ in Fig. 3.7(c).

Figure 3.8 shows that all proposed methods achieved similar speedups over one GPU, demonstrating efficient scaling on two GPUs. However, when $m \leq 1\times10^5$, the speedups were at most 1.7× because the GPU initialization time surpassed the performance gain provided by two GPUs.

We next focus on the Fused and Gram methods to investigate the performance with different numbers of columns n and different power iteration count q (Fig. 3.9). As shown in Fig. 3.9(a), the Gram performance was independent from q, as explained in Section 3.5.3. Thus, the additional computational cost needed for forming the Gram matrix had a limited impact on the GPU-based RSVD performance. In fact, the number $(qn^2\ell)$ of floating-point operations for power iteration is much smaller than that (mn^2) for forming the Gram matrix $\mathbf{G}$. Consequently, the measured run time for power iteration was less than 1% of that for forming the Gram matrix on our experimental machine.

As shown in Fig. 3.9(a), for a small power iteration number ($q = 1$), the Fused method outperformed the Gram method. In particular, their gap increased with n. The reason for this behavior is that the Gram method has a higher computation cost $(mn^2+n^2\ell+mn\ell)$ than the Fused method $(3mn\ell)$ when $q = 1$. In particular, the computation cost for forming the Gram matrix, *i.e.*, mn^2, made the Gram method slower compared with the Fused method when $n \geq 1 \times 10^4$. As for the data transfer cost D, when $q = 1$, there was no significant

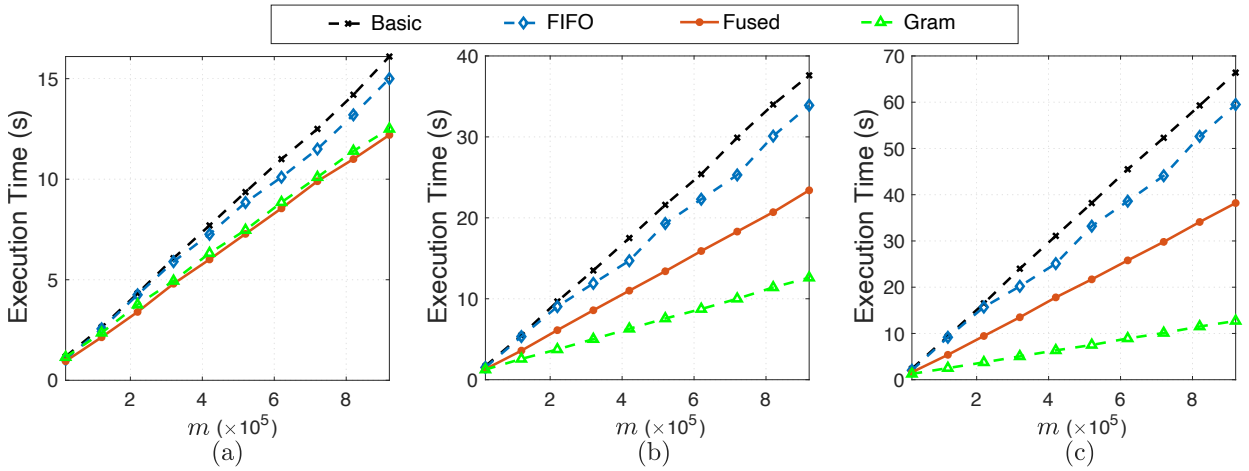

Figure 3.6: RSVD execution time a single Tesla V100 GPU with different numbers of rows m. Results for power iteration counts (a) $q = 1$, (b) $q = 4$, and (c) $q = 8$. Matrix sizes were $n = 5000$ and $\ell = 500$. The results are shown in execution time instead of flop/s because the flop counts of the Gram method are different from others. CPU-based results are omitted to focus on GPU-based results.

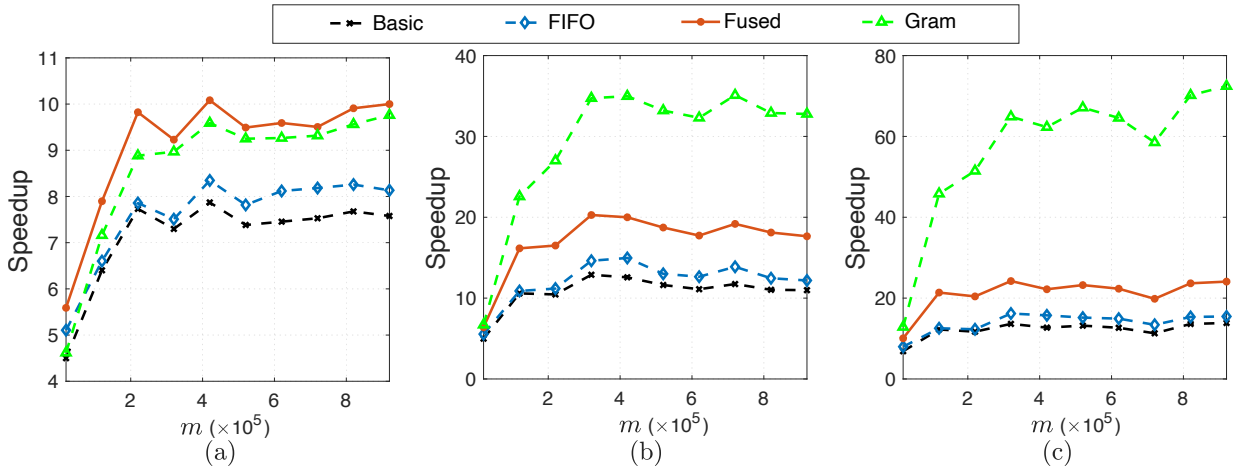

Figure 3.7: Speedup of one Tesla V100 GPU over two Xeon Silver 4114 CPUs with different numbers of rows m. Results for power iteration counts (a) $q = 1$, (b) $q = 4$, and (c) $q = 8$. Matrix sizes were $n = 5000$ and $\ell = 500$. For each method, we executed a multithreaded version of Algorithm 3 on two CPUs to compute the speedup.

difference between the Fused and Gram methods; the data transfer cost for both methods was approximately $2mn$ when $q = 1$ (Table 3.1). However, when we increased the power iteration count to $q = 4$, the cost D for the Fused method increased to approximately $5mn$, whereas that for the Gram method remained $2mn$. Consequently, the Gram method was twice as fast as the Fused method when $q = 4$ and $n < 0.6 \times 10^4$ in Fig. 3.9(a). Thus, the Gram method is robust for large q values but sensitive to large n values, whereas the Fused method is robust for large n values but sensitive to large q values. Therefore, we think that the Gram method is useful especially when a large number of iterations is required to acquire

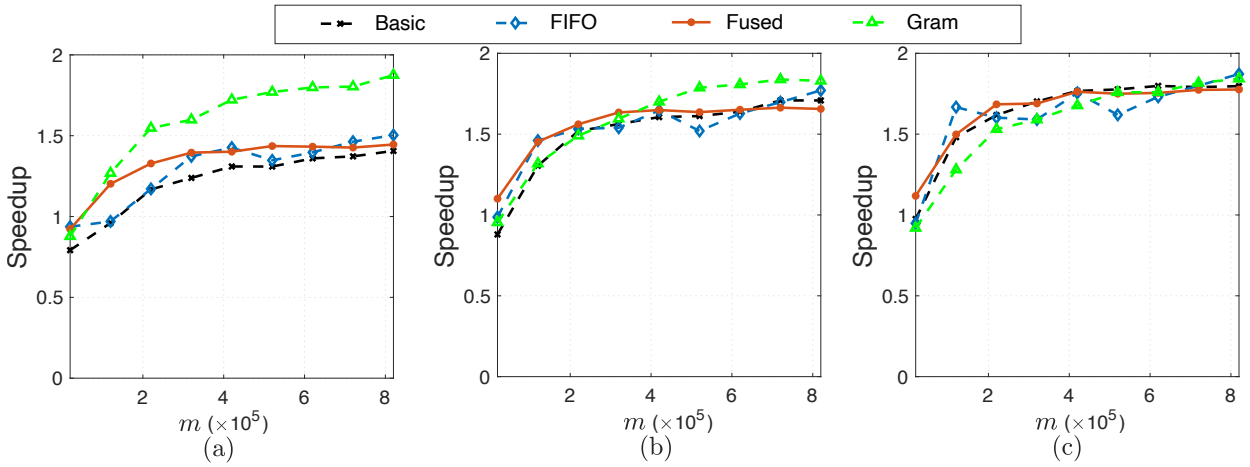

Figure 3.8: Speedup of two Tesla V100 GPUs over one V100 GPU with different numbers of rows m. Results for power iteration counts (a) $q = 1$, (b) $q = 4$, and (c) $q = 8$. Matrix sizes were $n = 5000$ and $\ell = 500$. For each method, we executed the same method on a single GPU to compute the speedup.

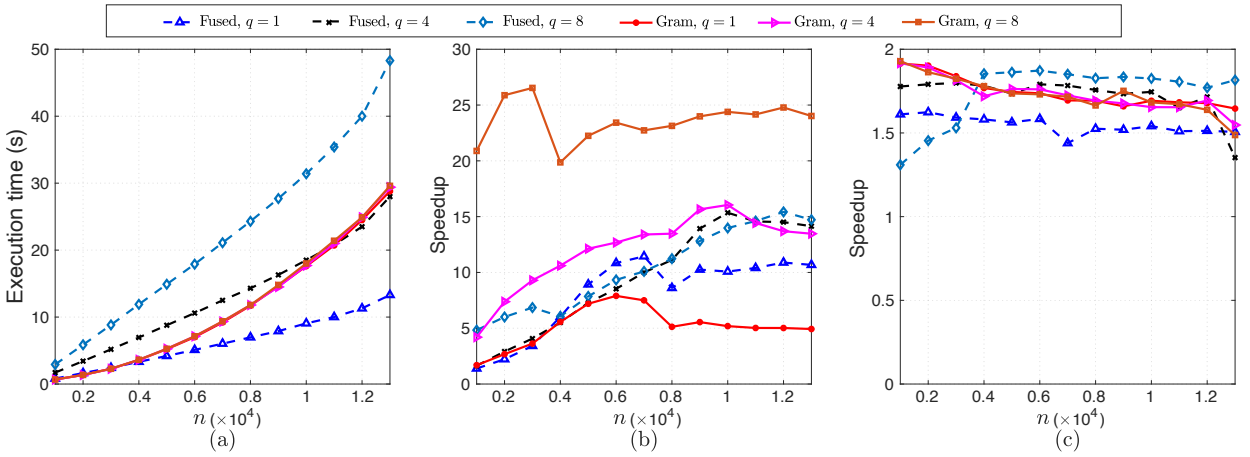

Figure 3.9: RSVD performance comparison of Fused and Gram methods with different numbers of columns n and different power iteration count q. (a) Execution time on a single Tesla V100 GPU, (b) speedup of one GPU over two Xeon Silver 4114 CPUs, and (c) speedup of two GPUs over one GPU. Matrix sizes were $m = 4 \times 10^5$ and $\ell = n/10$. We executed a multi-threaded version of Algorithm 3 to compute the speedup over two CPUs in (b). For each method in (c), the same method was executed to compute the speedup over a single GPU.

a high-accuracy approximation.

Figure 3.9(b) shows the speedup of one GPU over two CPUs. The speedup of the Fused method was up to 17× for different q. Comparatively, the speedup of the Gram method gradually increased from 7× to 24× as we increased power iteration count from $q = 1$ to $q = 8$. Figure 3.9(c) shows the speedup of two GPUs over one GPU. The performance was slightly better than the growing height case in Fig. 3.8, achieving 1.5–1.9× speedups.

Figure 3.10 shows the breakdown of the execution time obtained with the Gram method. We used two experimental setups: (a) increasing the height m of the matrix $\mathbf{A}$ with the fixed

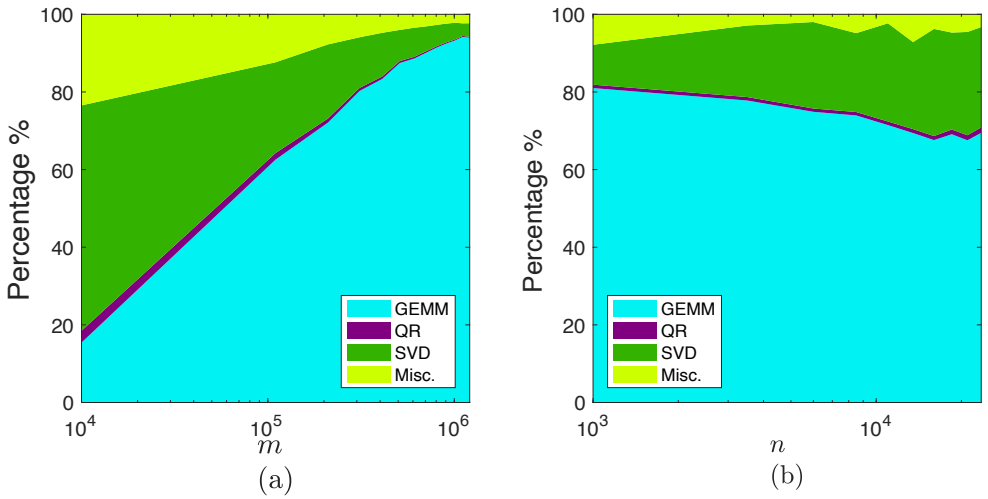

Figure 3.10: Breakdown of execution time on a single Tesla V100 GPU with $\ell = n/10$ and $q = 4$. Results of (a) growing height m of $\mathbf{A}$ with fixed width $n = 5000$ and of (b) growing width n of $\mathbf{A}$ with fixed height $m = 1 \times 10^5$. GEMM includes the time of CPU-GPU data transfer and GEMM operations. SVD denotes the deterministic SVD of matrix $\mathbf{B}$. Misc. is composed of initialization and random matrix generation.

width and (b) increasing the width n of the matrix $\mathbf{A}$ with the fixed height. Figure 3.10(a) demonstrates that the proportion of GEMM execution gradually increased with m and reached 90%. This behavior was due to the computational cost of GEMM ($mn^2+qn^2\ell+mn\ell$), which increases linearly with m, whereas that of SVD is fixed to $\mathcal{O}(n^3)$. Consequently, GEMM operations dominate the computation for extremely tall-skinny matrices.

In contrast, the proportion of SVD gradually increased with n and reached 24% of the total time when $n = 2.4{\times}10^4$ for $\mathbf{A}$ transformed from tall-skinny to square-like Fig. 3.10(b). This was due to the computational cost of SVD and GEMM, $\mathcal{O}(n^3)$ and $mn^2 + qn^2\ell + mn\ell$, respectively. Assuming that GEMM is parallelizable while other operations are not, parallel GEMM can achieve a linear speedup (Fig. 3.10(a)). By contrast, the theoretical speedup for situations similar to that illustrated in Fig. 3.10(b) is limited to at most 5× according to the Amdahl's law [3].

3.6.2 Performance Comparison

We next compared the proposed methods with the previous methods in terms of the out-of-core RSVD performance. As a comparative method, we used cuBLAS-XT [76] as a GPU-accelerated method. According to our preliminary results reported in Section 3.4, we maximized the GEMM performance by applying the row-wise 1D partition to cuBLAST-XT. With respect to the proposed method, we used the Fused method to compare the RSVD performance in Tflop/s because all methods, except the Gram method, had the same number of floating-point operations.

We varied the matrix setup $m : n : \ell$ from $100 : 10 : 1$ to $10000 : 10 : 1$ to investigate the out-of-core RSVD performance (Fig. 3.11). We first used a rectangular matrix setup where $m : n : \ell = 100 : 10 : 1$. Figure 3.11(a) demonstrates that all GPU-based methods achieved

4 Tflop/s, which outperformed the CPU-based method (approximately 0.35 Tflop/s); the basic, FIFO, and cuBLAS-XT methods showed similar performance whereas the Fused method achieved the highest flop/s. The GPU-based methods achieved approximately 14× higher flop/s than the CPU-based method. This performance gap came from that the deployed CPUs provided a theoretical peak performance of 0.9 Tflop/s, which was one-fifteenth as compared with that of a single V100 GPU. The second setup in Fig. 3.11(b) used tall-skinny matrices where $m : n : \ell = 1000 : 10 : 1$. The GPU performance dropped with the increasing ratio of $m : n$ and achieved a maximum of 2.6 Tflop/s. On the other hand, the CPUs still resulted in around 0.35 Tflop/s. The third setup in Fig. 3.11(c) was obtained with the extremely tall-skinny matrices where $m : n : \ell = 10000 : 10 : 1$. The GPU performance dropped to 0.95 Tflop/s, but the achieved performance was still higher than that on the CPUs (0.3 Tflop/s). The CPU performance stagnated and dropped slightly after $m > 1.2 \times 10^6$. Thus, the CPU performance was less sensitive to input matrix shapes than the GPU performance. In summary, the GPU-based methods outperformed the CPU-based method for square-like input matrices. However, the performance advantage over multi-core CPUs dropped as we increased the $m : n$ ratio. The main reason is that with the increasing $m : n$ ratio from 10 : 1 to 1000 : 1, the GEMM operations in RSVD get close to GEMV operations which are less efficient for GPUs.

We then used two GPUs to demonstrate the scalability of each implementation. As shown in Fig. 3.12, the proposed methods scaled well on two GPUs, achieving speedups of up to 1.9× over one GPU. The performance gap between cuBLAS-XT and our methods increased significantly on two GPUs. The speedups of cuBLAS-XT were about 0.9–1.2× in all setups. While cuBLAS-XT was enforced to 1D partition, we failed to make cuBLAS-XT to perform similar to our 1D scheme that was free of reduction between GPUs. The excessive data transfer of both CPU-GPU and GPU-GPU was the main reason for the performance degradation.

At last, we compare the out-of-core RSVD with the in-core RSVD for input data smaller than 7GB in Fig. 3.13. We also added an in-core full SVD results. We used the in-core SVD routine which was included in the MAGMA package. The results show that in-core RSVD achieved up to 226× and 32× acceleration against the in-core full SVD for square matrices (Fig. 3.13 (a)) and tall-skinny matrices (Fig. 3.13 (b)), respectively. The out-of-core RSVD is approximately 5× and 4× faster than in-core full SVD for square and tall-skinny matrices.

3.6.3 Numerical Study with Synthetic and Real Data

We used synthetic and real data to evaluate the numerical stability of the following three methods: (1) a deterministic SVD method provided by LAPACK [4], (2) the original RSVD method [39] that orthogonalizes both $\mathbf{P}$ and $\mathbf{Q}$, and (3) the proposed Gram method.

We used two different singular value distributions for the synthetic data: geometric and exponential distributions. For the geometric distribution, the *j-th* singular value σ_j was defined as $\sigma_j = \sigma_1 \gamma^{j-1}$ with the parameter $\gamma = 0.99$. For the exponential distribution, the *j-th* singular value σ_j was defined as $\sigma_j = \sigma_1 e^{-j/\beta}$ with the parameter $\beta = 160$. The matrix size $m \times n$ was set to $10,000 \times 5000$ with the sampling parameters $k = 64$ and $o = 64$. The real data came from the facial recognition technology dataset [87] containing human face images. All $25,389$ images were resized to the resolution of 512×768 pixels, and thus the

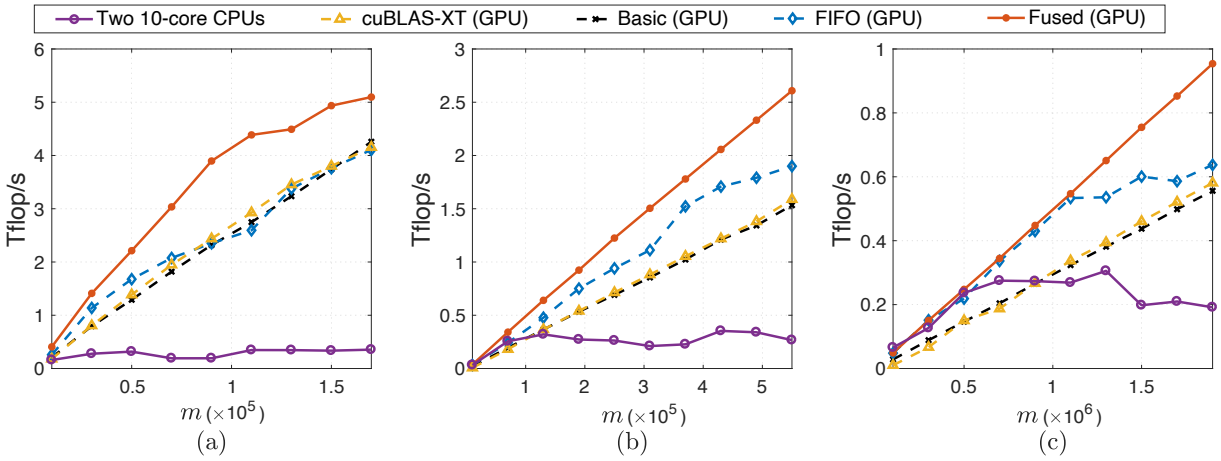

Figure 3.11: Comparison of CPU-based method, cuBLAS-XT [76], and the proposed Fused method with different matrix shapes. (a) rectangular with $m : n : \ell = 100 : 10 : 1$, (b) tall-skinny with $m : n : \ell = 1000 : 10 : 1$, and (c) extremely tall-skinny with $m : n : \ell = 10000 : 10 : 1$. Results obtained on a single Tesla V100 GPU are presented in flop/s. Power iteration count was $q = 4$. The CPU-based method was multi-threaded using two Xeon Silver 4114 CPUs. Note that the horizontal scale is different in three setups because we set the maximum input matrix size to the maximum memory size that our system can hold.

Table 3.2: Comparison of approximation error $\|\mathbf{A} - \hat{\mathbf{A}}\|_F / \|\mathbf{A}\|_F$ with different SVD methods and different power iteration count q.

Data	Deterministic SVD	Original RSVD		Gram method	
		$q = 1$	$q = 4$	$q = 1$	$q = 4$
Geometric distribution	0.5204	0.5297	0.5204	0.5302	0.5204
Exponential distribution	0.6622	0.6828	0.6623	0.6830	0.6623
FERET [87]	0.1661	0.1690	0.1661	0.1690	0.1661

input matrix consisted of $393,216 \times 25,389$ entries.

Table 3.2 shows the approximation error $\|\mathbf{A} - \hat{\mathbf{A}}\|_F / \|\mathbf{A}\|_F$, where $\|\cdot\|_F$ denotes the Frobenius norm. When $q = 4$, the Gram method achieved similar errors compared with those of the original RSVD and deterministic SVD methods. By contrast, when $q = 1$, there were small differences across these methods. Thus, increasing the number q of power iterations slightly improved the accuracy for randomized methods. Hence, the Gram method achieved the same level of accuracy as the deterministic SVD method without the orthogonalization of $\mathbf{P}$ (line 4 of Algorithm 3).

3.7 Case Study with RPCA

Finally, we demonstrate the performance of out-of-core RSVD with an RPCA application [10]. RPCA is a common method in computer vision and machine learning, which recovers a low-rank matrix with an unknown fraction of data corruption [10]. While being effective, RPCA algorithms are computationally demanding due to iterative SVD operations required to reveal the singular values for thresholding.

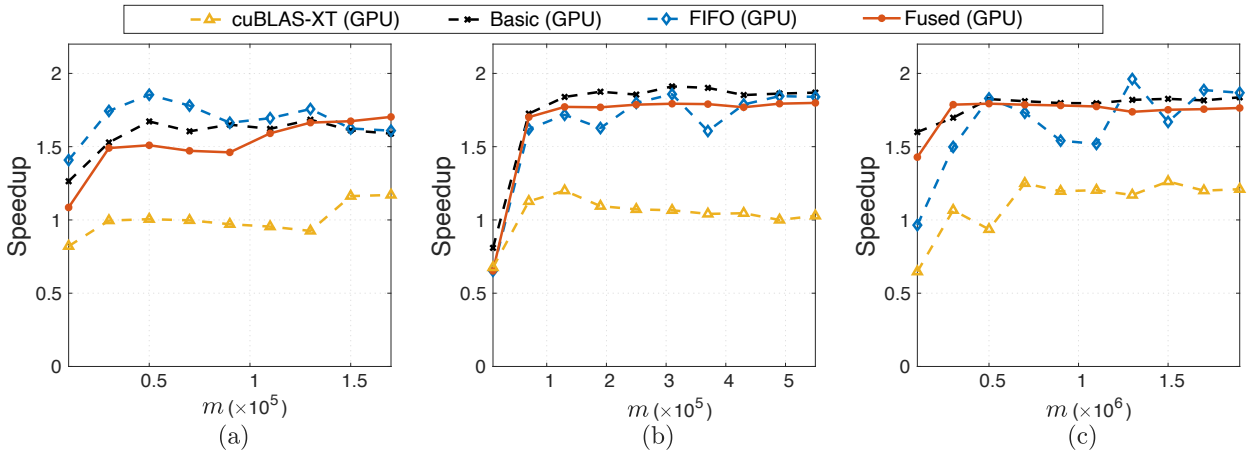

Figure 3.12: Speedup of two Tesla V100 GPUs over one V100 GPU with different matrix shapes: Results for (a) rectangular matrices with $m : n : \ell = 100 : 10 : 1$, (b) tall-skinny matrices with $m : n : \ell = 1000 : 10 : 1$, and (c) extremely tall-skinny matrices with $m : n : \ell = 10000 : 10 : 1$. Power iteration count was $q = 4$. For each method, the same method was executed to compute the speedup.

Algorithm 8 lists a pseudocode of the RPCA algorithm [10, 58], which separates the sparse corruptions $\mathbf{S}$ from the original data $\mathbf{M}$ so that a low-rank matrix $\mathbf{L}$ can be obtained as $\mathbf{L} = \mathbf{M} - \mathbf{S}$. The problem can be written as

$$\min_{\mathbf{L},\mathbf{S}} ||\mathbf{L}||_* + \lambda||\mathbf{S}||_1 \quad \text{subject to} \quad \mathbf{M} = \mathbf{L} + \mathbf{S}, \tag{3.6}$$

where $||\cdot||_*$ and $||\cdot||_1$ denote the nuclear norm and ℓ_1 norm of a matrix, respectively, and λ is a positive weighting parameter that is usually set to $\lambda = 1/\sqrt{\max(m, n)}$. Various solvers have been proposed for this convex optimization problem, and recent solution methods drastically improved the computation efficiency [58, 59, 119]. Similar to Oh *et al.* [81], we used an RSVD solver to replace the deterministic SVD solver in RPCA to accelerate the computation. The shrinkage operator $\mathcal{S}_\varepsilon[\cdot]$ in line 4 of Algorithm 8 is defined as

$$\mathcal{S}_\varepsilon[x] = \begin{cases} x - \varepsilon, & \text{if} \quad x > \varepsilon, \\ x + \varepsilon, & \text{if} \quad x < -\varepsilon, \\ 0, & \text{otherwise}, \end{cases} \tag{3.7}$$

where ε represents the threshold value.

In Algorithm 8, GEMM operations appear in lines 3–4, whereas all other computations are vector summations. GEMM operations have a higher operational intensity than vector summations, which cannot be efficiently offloaded to GPUs. Therefore, we decided to call LAPACK routines to implement vector summations on the CPU. On the other hand, we used the Gram method for RSVD in line 3 and the out-of-core GEMM with 1D partition for the reconstruction of $\mathbf{L}$ at line 4. Vector summations were implemented on the CPU with LAPACK routines.

All experiments were conducted in double precision using the same machine described in Section 3.4. The sampling parameters for RSVD were $k = 100$, $o = 100$, and $q = 4$. The

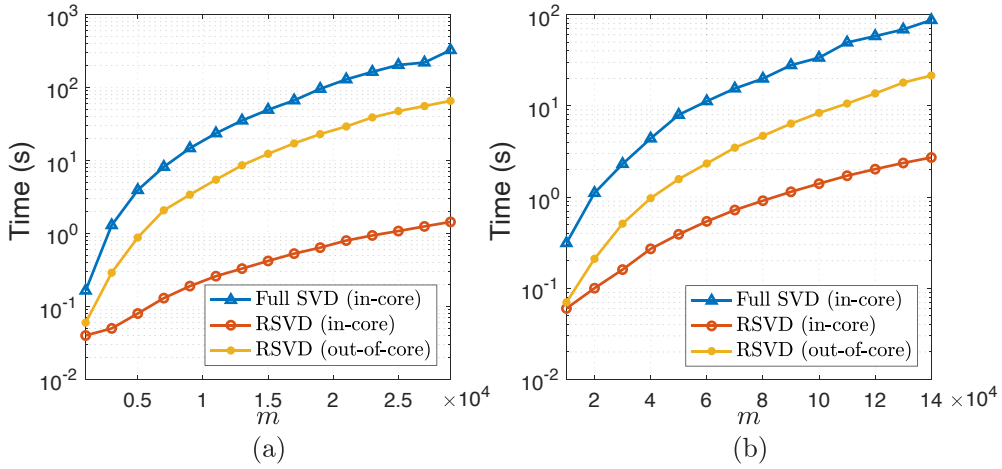

Figure 3.13: Comparison of out-of-core and in-core performance. (a) Execution time with square matrices ($m = n$). (b) Execution time with tall-skinny matrices ($n = m/10$). Note that for in-core implementation of full SVD and RSVD, the CPU-GPU data transfer time is not included.

Figure 3.14: An execution example of RPCA. (a) Input image, (b) low-rank image and (c) sparse image.

parameters in Algorithm 8 were set as $t = 10^{-6}$, $\rho = 1.5$, $\mu_0 = 1.5/||\mathbf{M}||_\infty$, and $\varepsilon = \mu_i^{-1}$. As for data, we used a high definition traffic video that was heavily corrupted by camera jittering and a large traffic volume. We extracted 1000 frames with a resolution of 1920×1080. Therefore, the input matrix $\mathbf{M}$ is of size $2,073,600 \times 1,000$. Figure 3.14 illustrates a sample frame output, which indicates that our implementation successfully separated the input data into sparse traffic noise and the low-rank background.

Table 3.3 compares the RPCA performance measured on the experimental machine. Compared with a multi-core CPU implementation, our GPU-accelerated Gram method accelerated the RSVD computation by 23.3×, which halved the total RPCA time. This speedup is reasonable according to the out-of-core GEMM performance presented in Fig. 3.7(b); the acceleration on two GPUs over two 10-core CPUs was up to 35× for out-of-core RSVD. Fig. 3.3(a); the highest performances on two GPUs and two CPUs were about 2.5 Tflop/s and 0.22 Tflop/s, respectively, demonstrating a speedup of 11.4× for out-of-core GEMM. Fig. 3.3(a); the highest performances on two GPUs and two CPUs were about 2.5 Tflop/s and 0.22 Tflop/s, respectively, demonstrating a speedup of 11.4× for out-of-core GEMM. However, the maximum performance of two Tesla V100 GPUs was close to 14 Tflop/s (Fig. 3.3(b)), implying that the data size was not sufficiently large to maximize the per-

Algorithm 8: RPCA by RSVD

Input : matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$, target rank k, oversampling parameter o, power iteration count q, convergence condition t and parameters μ_0 and ρ.

Output: low-rank matrix $\mathbf{L} \in \mathbb{R}^{m \times n}$ and sparse matrix $\mathbf{S} \in \mathbb{R}^{m \times n}$.

$\mathbf{Y}_0 = \mathbf{M} \ / \ \max(||\mathbf{M}||_2,\ \lambda^{-1}||\mathbf{M}||_\infty)$; $\mathbf{S}_0 = \mathbf{0}^{m \times n}$; $i = 0$;
while *TRUE* **do**
  $[\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}^T] = \text{rsvd}(\mathbf{M} - \mathbf{S}_i + {\mu_i}^{-1}\mathbf{Y}_i\ ,\ k,\ o,\ q)$;
  $\mathbf{L}_{i+1} = \mathbf{U}\mathcal{S}_{{\mu_i}^{-1}}[\mathbf{\Sigma}]\mathbf{V}^T$; `// shrinkage operation on Σ`
  $\mathbf{S}_{i+1} = \mathcal{S}_{\lambda{\mu_i}^{-1}}[\mathbf{M} - \mathbf{L}_{i+1} + {\mu_i}^{-1}\mathbf{Y}_i]$;
  $\mathbf{Z}_{i+1} = \mathbf{M} - \mathbf{L}_{i+1} - \mathbf{S}_{i+1}$;
  $\mathbf{Y}_{i+1} = \mathbf{Y}_i + \mu_i\mathbf{Z}_{i+1}$;
  if $(\ ||\mathbf{Z}_{i+1}||_F \ / \ ||\mathbf{M}||_F < t\ )$ **break**; `// evaluate convergence`
  $\mu_{i+1} = \rho\mu_i$; $i = i + 1$;
end

Table 3.3: Execution time of RPCA based on two Xeon Silver 4114 CPUs and two Tesla V100 GPUs. Both implementations converged with the same number (28) of iterations.

Breakdown	CPU (s)	GPU (s)
RSVD	1120	48
Miscellaneous	951	946
Total RPCA	2071	994

formance on the GPU. By accelerating RSVD with GPUs, the performance bottleneck of RPCA moved to the vector summation part, which has a low operational intensity. The acceleration of out-of-core algorithms with a low operational intensity, such as vector summation, is still limited by the CPU-GPU transfer bandwidth, which is a challenge to fully utilize GPUs.

3.8 Conclusions

Over the past decade, randomized algorithms have shown significant advancements in terms of the computation efficiency. However, there have not been many attempts to harness the modern computing architectures such as GPU accelerators. The likely explanation is that GPUs are still considered as specialized devices. At the same time, large-scale computing is now performed on supercomputers that are prevalently equipped with accelerators; hence, developing new algorithms for evolving computing architectures is becoming urgent.

We studied GPU-accelerated methods, namely, Fused and Gram, to reduce out-of-core data access for computing RSVD. The Gram method, which was especially effective for tall-skinny matrices, achieved up to 5.2× speedup compared with a straightforward method that deploys the highly-tuned GEMM scheme and the 1D data partition scheme. The Fused method effectively accelerated the RSVD up to 1.9× compared with the straightforward method. This work allows us to see the directions of the randomized algorithm development

as we move toward exascale computing. Most immediate direction is the independent block operations which fit for accelerators.

Our analysis and empirical results also revealed that CPU-GPU transfer bandwidth limits the RSVD performance on a common workstation, especially for tall-skinny matrices that limits the scalability on GPUs. This constraint is expected to be mitigated with the introduction of the next generation PCIe and NVlink [77] buses. Nevertheless, reducing the amount of data access and communication remains the major challenge in developing scalable linear algebra algorithms for both single- and multi-node systems. This is also our main focus in the future.

Chapter 4

Block Randomized Singular Value Decomposition on GPUs

4.1 Introduction

SVD is an essential tool in computer vision, machine learning, data analysis, and various other scientific computing. Studies on improving the performance and numerical stability of SVD have been ongoing ever since its advent [18, 28, 33, 55], and have been successfully applied to applications such as principal component analysis (PCA) [44, 86]. Recently, an RSVD algorithm has been proposed to further accelerate SVD by exploiting the low-rank structure of data [39, 106] and now appears a method of choice for fast approximate SVD computation.

While SVD has been made efficient in terms of its computational complexity based on these findings, the transition of modern computing architectures, such as GPUs, allow us to develop even faster methods by taking advantage of high parallelism. Nevertheless, to fully benefit from the modern computing architectures, there are a few major challenges. Although computers' arithmetic operations are becoming ever efficient, communication between memory hierarchies or through networks is emerging as the bottleneck for a lot of applications in distributed memory systems equipped with accelerators [7, 49, 50]. The gap between communication cost and computational cost is expected to increase, where arithmetic operations are fast and highly parallelized but data communication remains slow [15]. Note that the communication refers to not only the data transfer between computing nodes but also the data transfer of CPUs/accelerators to their memory. A series of communication avoiding algorithms have been proposed to tackle this gap [16, 17, 42]. These studies aim to redesign linear algebraic algorithms to reduce data communication among memory hierarchies.

For fast SVD computation, this trend requires a *locality-aware* method with less access to the input data. Especially, for large-scale data, the traditional SVD computation cannot fully benefit from a fast BLAS3 computation, or its main computational kernel GEMM, due to that (1) the data may not fit in a single memory space, (2) its computation pattern includes vast data accesses, and (3) communication between distinct memory hierarchy levels [37]. As for RSVD, input data will be accessed for $2q+2$ times, where q denotes the number of power

iterations [39, 88], to attain a high accuracy approximation. Single-pass algorithms [101, 118], also called streaming algorithms, are proposed, which access the target with *"one touch."* Mostly, they include iterations to construct a *sketch* [39, 101, 118], which have data dependency and is difficult be made in parallel. Furthermore, there exists an accuracy-performance trade-off in these algorithms [101, 118].

To our best knowledge, few works have been done to accelerate single-pass RSVD on GPUs. Regarding accelerating multi-pass RSVD [39] on GPUs, the computations of RSVD are mainly matrix-matrix multiplication. It means that implementing RSVD on multi-core CPUs or GPUs is relatively easier than implementing deterministic SVD [18]. Yamazaki *et al.* [114] proposed exploiting random sampling to update partial SVD on a hybrid CPU/GPU cluster. Their work showed that a random sampling algorithm achieved a speedup of up to 14.1× compared with a standard deterministic SVD implementation on multi-core CPUs. They assumed that GPU memory can hold all the working data. Voronin *et al.* proposed a comprehensive randomized linear algebra library named RSVDPACK [106]. While effective, their GPU implementation is in-core, and efficient computation can only be achieved when the data fits in the space of GPU memory.

This study considers redesigning the RSVD algorithm especially for large matrices that do not fit in the GPU memory and a limited CPU-GPU communication bandwidth. We propose a two-pass RSVD algorithm named block randomized SVD (BRSVD), which accesses the input data only twice in the whole computation. Similar to the GPU-only strategy [35], BRSVD uses GPUs for all computations which fully utilizes the power of accelerators and efficiently processes data without burdening the host CPU. BRSVD decomposes the original power method into independent block executions to reduce access to the target matrix. Different from the previous works [106, 114], our proposed out-of-core algorithm frees the memory capacity limitation on the input data. Furthermore, BRSVD decomposes the original power method into independent block executions to reduce the communication between CPUs and GPUs.

We compare the efficiency with an in-core implementation, which is the performance upper-bound of RSVD. For large-scale data, BRSVD achieves a significant speedup in comparison to the original algorithm. We then assess the accuracy of the proposed method using both synthetic and real data and compare with existing algorithms. Our experiment shows that with a moderate partition size of the input matrix, BRSVD gives a close approximation to the original algorithm. The empirical results also indicate that the proposed algorithm outperforms the single-pass algorithm in terms of accuracy.

Section 4.2 introduces the preliminaries regarding RSVD algorithm. We describe the proposed BRSVD algorithm in Section 4.3 and give the experimental results in Section 4.4. Section 4.6 concludes this chapter and discuss the future work.

4.2 Preliminaries

RSVD has been made popular by Halko *et al.*'s work [39] built upon the previous studies on randomized linear algebra [28, 57, 64, 88, 89]. The randomization approach outperformed classical deterministic SVD methods in terms of speed while maintaining equivalent accuracy and robustness.

As described in [39], given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, an orthonormal basis $\mathbf{Q}$ can be constructed such that $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^\top\mathbf{A}$. The factorization (SVD and QR) then can be efficiently computed using a relatively small *sketch matrix* $\mathbf{B} = \mathbf{Q}^\top\mathbf{A}$, when the basis matrix $\mathbf{Q}$ has few columns. In other words, when the rank of matrix $\mathbf{A}$ is small, *i.e.*, $\text{rank}(\mathbf{A}) = k \ll \min(m, n)$, a small matrix $\mathbf{B}$ can be created and an SVD of the small matrix $\mathbf{B}$ reveals the SVD of the original matrix $\mathbf{A}$ as long as the range of the projector $\mathbf{Q}$ retains the action of the original matrix $\mathbf{A}$.

The process of randomized factorization has two stages: (1) construction of the basis $\mathbf{Q}$ with a random projection of the original matrix $\mathbf{A}$, and (2) factorization of the small matrix $\mathbf{B}$ with a standard deterministic method. In stage (1), it is important to construct $\mathbf{Q} = (\mathbf{q}^{(1)}, \mathbf{q}^{(2)}, \ldots, \mathbf{q}^{(l)})$, in which $\mathbf{q}^{(i)}$ denotes the i-th column vector of $\mathbf{Q}$, such that $\mathbf{Q}$ covers the range of $\mathbf{A}$. To achieve this, a random vector $\boldsymbol{\omega}$ can be used to form a sample vector $\mathbf{y}$ as

$$\mathbf{y}^{(i)} = \mathbf{A}\boldsymbol{\omega}^{(i)}, \quad i = 1, 2, \ldots, l, \tag{4.1}$$

where $l = k + o$, and o denotes the oversampling parameter. With l samplings, a sample matrix $\mathbf{Y} = (\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(l)})$ can be constructed. In some cases, the singular spectrum of matrix $\mathbf{A}$ may decay slowly, power iteration is used to overcome this issue by projecting more information of $\mathbf{A}$ into the sample matrix $\mathbf{Y}$ so as to accelerate the spectrum decay:

$$\mathbf{Y} = (\mathbf{A}\mathbf{A}^\top)^q\mathbf{A}\boldsymbol{\Omega}, \tag{4.2}$$

where the random matrix $\boldsymbol{\Omega} = \left(\boldsymbol{\omega}^{(1)}, \boldsymbol{\omega}^{(2)}, \ldots, \boldsymbol{\omega}^{(l)}\right)$ is a standard Gaussian matrix of *i.i.d* normal random variables with mean 0 and variance 1. The acceleration of the power method can be achieved with

$$\begin{aligned}\mathbf{Y} &= (\mathbf{A}\mathbf{A}^\top)^q\mathbf{A}\boldsymbol{\Omega} \\ &= (\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top\mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^\top)^q\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top\boldsymbol{\Omega} \\ &= \mathbf{U}\boldsymbol{\Sigma}^{2q+1}\mathbf{V}^\top\boldsymbol{\Omega}.\end{aligned} \tag{4.3}$$

Afterward, the basis of $\mathbf{Y}$ is computed by $\mathbf{Q} = \text{orth}\,(\mathbf{Y})$, where the operator $\text{orth}\,(\cdot)$ represents orthonormalization. In stage (2), matrix $\mathbf{B}$ is formed as $\mathbf{B} = \mathbf{Q}^\top\mathbf{A}$ and factorized by a conventional deterministic factorization method. The RSVD algorithm is summarized in Algorithm 9.

4.3 Proposed Method: Block Randomized SVD (BRSVD)

In modern computing architectures, flop counts become rather irrelevant due to the greatly increased communication cost between storage and processor. A measure of algorithmic communication performance is called *pass-efficiency* [39], which counts how many times the data is accessed by a specific algorithm. In lines 2 and 4 of Algorithm 9, the power method [88] requires totally $2q + 2$ passes mainly for matrix-matrix multiplication, which translates to communication cost of $(2q+2)mn$. However, if the memory of processor cannot hold all working data for matrix-matrix multiplication, the algorithm has to be implemented with out-of-core GEMM routines, which will increase communication cost significantly. High

Algorithm 9: RSVD algorithm [39]

Input : matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, target rank k, oversampling parameter p, and power iteration exponent q.

Output: SVD of $\mathbf{A}$: matrices $\mathbf{U} \in \mathbb{R}^{m \times l}$, $\mathbf{\Sigma} \in \mathbb{R}^{l \times l}$, and $\mathbf{V}^\top \in \mathbb{R}^{l \times n}$.

1 Generate a Gaussian matrix $\mathbf{\Omega} \in \mathbb{R}^{n \times l}$, where $l = k + o$.

2 $\mathbf{Y} = (\mathbf{A}\mathbf{A}^\top)^q \mathbf{A}\mathbf{\Omega}$; `// sketch A and perform power iterations`

3 $\mathbf{Q} = \text{orthonormalize}(\mathbf{Y})$; `// form an orthonormal basis of Y`

4 $\mathbf{B} = \mathbf{Q}^\top \mathbf{A}$; `// form B`

5 $\left[\widetilde{\mathbf{U}}, \mathbf{\Sigma}, \mathbf{V}^\top\right] = \text{svd}(\mathbf{B})$; `// truncated rank-`l `SVD of B`

6 $\mathbf{U} = \mathbf{Q}\widetilde{\mathbf{U}}$; `// form U`

communication cost will be the bottleneck for processing large-scale data. We will give detailed analysis in Section 4.3.1. Our goal is to design a new RSVD algorithm by reducing the communication cost without hurting the algorithm's accuracy.

Note that the sampling process in Eq. (4.1) is a matrix-matrix multiplication, which can be decomposed into block operations and conducted independently. We wish to design a block power method so as to reduce the data access to $\mathbf{A}$. With a moderate partitioning of the input matrix, we can accelerate the singular value decay for each sub-matrix independently. Therefore, BRSVD (Algorithm 10) can avoid out-of-core GEMM in processing large-scale data.

We give our assumption before elaborating our proposed algorithm. Suppose that the input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is partitioned into s column sub-matrices. We use the sub-matrix notation in [34, 39] here, then each column sub-matrix is expressed as $\mathbf{A}_{(:,\beta_j)} \in \mathbb{R}^{m \times n'}$, where β_j is an ordered set of indices defined as $\beta_j = [jn', jn' + 1, \ldots, (j+1)n' - 1]$ for the j-th sub-matrix with n' columns. Each submatrix $\mathbf{A}_{(:,\beta_j)}$ has its own singular values $\mathbf{\Sigma}_j$. Our assumption is that the singular spectrums of sub-matrices do not highly deviate from each other. With similar singular spectrums, the spectrum decay of each sub-matrix can be accelerated independently by the power method. This assumption reflects the practical situation. In data processing, it is not recommended to combine unrelated matrices into a single matrix. Therefore, it is a rare case to have an input matrix which is composed of sub-matrices with extremely different singular values. Regarding the accuracy of BRSVD, it is experimentally confirmed in Section 4.4.4.

Figure 4.1 illustrates the overall pipeline of the proposed algorithm. A Gaussian matrix $\mathbf{\Omega}_j \in \mathbb{R}^{n' \times l}$ is drawn to sketch each $\mathbf{A}_{(:,\beta_j)}$. The resulting matrix is further refined via a power method with exponent q by reusing the transferred column block $\mathbf{A}_{(:,\beta_j)}$, and the sample matrix $\mathbf{Y}_j \in \mathbb{R}^{m \times l}$ is calculated independently as

$$\mathbf{Y}_j = \left(\mathbf{A}_{(:,\beta_j)}\mathbf{A}_{(:,\beta_j)}^\top\right)^q \mathbf{A}_{(:,\beta_j)}\mathbf{\Omega}_j, \tag{4.4}$$

which we call a *block* power method. The sub-matrix $\mathbf{A}_{(:,\beta_j)}$ can be orthonormalized before applying the power method on sample matrix $\mathbf{A}_{(:,\beta_j)}\mathbf{\Omega}_j$. The orthonormalization will reduce the magnitude of the projection so that batches with large magnitude will not overwhelm the final sample matrix $\mathbf{Y}$. Note that with the power iteration number $q = 0$, BRSVD and RSVD become equivalent.

Algorithm 10: BRSVD algorithm

Input : matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, target rank k, oversampling parameter o, power iteration exponent q, and partition number s.

Output: SVD of $\mathbf{A}$: matrix $\mathbf{U} \in \mathbb{R}^{m \times l}$, $\mathbf{\Sigma} \in \mathbb{R}^{l \times l}$, and $\mathbf{V}^\top \in \mathbb{R}^{l \times n}$.

$n' = \lceil n/s \rceil$; $l = k + o$; $\mathbf{Y} = \mathbf{0}_{m \times l}$;
for $j \leftarrow 0$ **to** $s - 1$ **do in parallel**
  Generate a Gaussian matrix $\mathbf{\Omega}_j \in \mathbb{R}^{n' \times l}$;
  $\mathbf{Y}_j = \left(\mathbf{A}_{(:,\beta_j)} \mathbf{A}_{(:,\beta_j)}{}^\top\right)^q \mathbf{A}_{(:,\beta_j)} \mathbf{\Omega}_j$; `// sampling & block power iteration`
end
$\mathbf{Q} = \text{orthonormalize}(\sum \mathbf{Y}_j)$; `// reduction and orthonormalization`
for $j \leftarrow 0$ **to** $s - 1$ **do in parallel**
  $\mathbf{B}_{(:,\beta_j)} \leftarrow \mathbf{Q}^\top \mathbf{A}_{(:,\beta_j)}$; `// form each` $\mathbf{B}_{(:,\beta_j)}$
end
$\left[\widetilde{\mathbf{U}}, \mathbf{\Sigma}, \mathbf{V}^\top\right] = \text{svd}(\mathbf{B})$; `// gather` $\mathbf{B}_{(:,\beta_j)}$ `into` $\mathbf{B}$ `and SVD`
$\mathbf{U} = \mathbf{Q}\widetilde{\mathbf{U}}$; `// form` $\mathbf{U}$

We can observe that if $s = 1$, Eq. (4.4) is equivalent to Eq. (4.2). For an extreme case where $s = n$, each sub-matrix is shrunk to a column of $\mathbf{A}$ and results in $\mathbf{Y}_j = (\mathbf{a}_j \mathbf{a}_j{}^\top)^q \mathbf{a}_j \mathbf{\Omega}_j = \mathbf{a}_j (\mathbf{a}_j{}^\top \mathbf{a}_j)^q \mathbf{\Omega}_j = |\mathbf{a}_j|^{2q} \mathbf{a}_j \mathbf{\Omega}_j$. For a moderate batch $s \in (1, n)$, the block power method will weaken the effect of accelerating the spectrum decay of sample matrix $\mathbf{Y}$. We will use extensive experiments to verify its accuracy in the next section. Because the sampling and power iteration of each block are independent operations, the data transfer and computation can be overlapped and executed concurrently.

The sample matrix $\mathbf{Y}$ can then be computed by reduction as

$$\mathbf{Y} = \sum_{j=0}^{s-1} \mathbf{Y}_j$$

After each update of $\mathbf{Y}$, the transferred column block $\mathbf{A}_{(:,\beta_j)}$ can be discarded from the GPU for avoiding memory overflow. Once the sample matrix $\mathbf{Y}$ is created, its orthonormalized basis $\mathbf{Q} \in \mathbb{R}^{m \times l}$ can be constructed by QR decomposition. After acquiring the basis $\mathbf{Q}$, the input matrix $\mathbf{A}$ is accessed for the second pass to compute a small core matrix $\mathbf{B} \in \mathbb{R}^{l \times n}$:

$$\mathbf{B}_{(:,\beta_j)} \leftarrow \mathbf{Q}^\top \mathbf{A}_{(:,\beta_j)}.$$

As the above operation shows, sub-matrix $\mathbf{B}_{(:,\beta_j)}$ also can be independently computed from the corresponding column block $\mathbf{A}_{(:,\beta_j)}$. Finally, all sub-matrices of $\mathbf{B}$ is gathered and an SVD of $\mathbf{B}$ is computed to yield its decomposition $\widetilde{\mathbf{U}}, \mathbf{\Sigma}, \mathbf{V}^\top$, and by re-projecting the obtained basis $\widetilde{\mathbf{U}}$ via $\mathbf{Q}$, the left singular vectors $\mathbf{U} = \mathbf{Q}\widetilde{\mathbf{U}}$ of the input matrix $\mathbf{A}$ can be obtained.

4.3.1 Efficiency Analysis

For now, let us look at the efficiency analysis summarized in Table 4.1. We suppose that the data size exceeds the GPU memory, therefore RSVD has to be implemented in an out-of-core

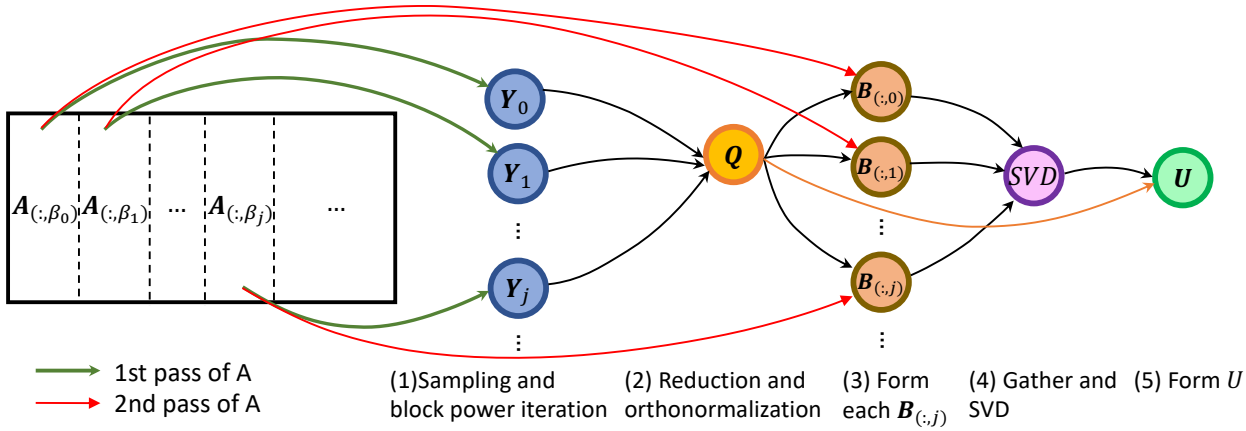

Figure 4.1: Diagram of proposed BRSVD method. Column blocks $\mathbf{A}_{(:,\beta_j)}$ in local memory are reused in the RSVD computation pipeline.

fashion. Table 4.1 compares the computational and communication costs of the proposed algorithm with a straightforward implementation for RSVD. #flops refers to the arithmetic computational cost in floating point operations. #words refers to the communication cost between CPU and GPU memory. We first show the #flops of the original and proposed algorithm. In Algorithm 9, the first GEMM operation for $\mathbf{A\Omega}$ (line 1) requires $2mnl$ flops, and the following power iteration requires $4qmnl$. The QR operation to orthonomralize $\mathbf{Y}$ requires $2ml^2$ flops. Forming $\mathbf{B}$ requires $2mnl$ flops. The SVD operation and forming $\mathbf{U}$ requires $6nl^2$ and $2ml^2$, respectively. Regarding BRSVD, each block sampling $\mathbf{A}_{(:,\beta_j)}\mathbf{\Omega}_j$ requires $2mn'l$ flops. The overall flops for s blocks are $s \times 2mn'l = 2mnl$, which equals the original algorithm. The other GEMM operations also have the same #flops as the original algorithm. The remaining operations have slightly different #flops for BRSVD, while they only contribute to a small portion of overall computation (We will show the time breakdown in the next section). Therefore, we come to the conclusion that BRSVD have the same theoretical flop counts as the RSVD algorithm.

While the #flops remains the same in both algorithms, BRSVD significantly reduces the communication cost. As shown in Table 4.1, the #words for RSVD in sampling is mnl/b, where b denotes the partition size in out-of-core GEMM. RSVD requires overall $2(q+1)mnl/b + ml$ to calculate the out-of-core GEMM. The best case is that if $b \geq l$, the communication cost is $2(q+1)mn + ml$. We leave b in those equations because GEMM library other than cuBLAS [76] may not allow users to set the partition size b. On the other hand, the communication cost of the proposed BRSVD is fixed to $2mn + ml$, which is independent of the power iteration number q. This reduction comes from the fact that BRSVD avoids out-of-core GEMM entirely and reuses the column blocks $\mathbf{A}_{(:,\beta_j)}$ on the local memory (see Fig. 4.1). As we will see in the next section, this reduction of communication cost significantly improves the efficiency of RSVD computation for large-scale data.

Table 4.1: Computational and communication costs comparison. #flops refers to the arithmetic computational cost in floating point operations. #word indicates communication cost between CPU and GPU. Line # indicates the corresponding operation blocks in Algorithm 10. The non-dominant terms of #flops for random number generation in (1), QR in (4) and SVD in (6) are dropped. (For a detailed #flops count, please refer to Matrix Computations [34] and LAPACK working note [12].) We show the commucation cost for out-of-core GEMM in item (2), (3) and (5). b denotes the partition size in out-of-core GEMM. The value of b varies in different implementations (*e.g.* LAPACK or cuBLAS) and hardware architectures. Note that if $b \geq l$, #words (RSVD) are mn, $2qmn$ and mn for item (2), (3) and (5), respectively.

	line #	#flops	#words (RSVD)	#words (BRSVD)
(1) Random number generation	3	nl	0	0
(2) Sampling	4	$2mnl$	mnl/b	mn
(3) Power Iterations	4	$4qmnl$	$2qmnl/b$	0
(4) Orthonormalization	6	$2ml^2$ (p.A208 of [16])	0	0
(5) Form $\mathbf{B}$	8	$2mnl$	mnl/b	mn
(6) SVD	10	$6nl^2$ (p.493 of [34])	0	0
(7) Form $\mathbf{U}$	11	$2ml^2$	ml	ml
Total		$4(q+1)mnl + 4ml^2 + 6nl^2 + nl$	$2(q+1)mnl/\ b + ml$	$2mn + ml$

4.3.2 Implementation Detail

Here we describe implementation details of BRSVD that will be needed to reproduce the work.

For generating Gaussian random matrices $\mathbf{\Omega}_j$ on the GPU, we have used cuRAND library [78]. The random number generation is performed in parallel with transferring submatrix $\mathbf{A}_{(:,\beta_j)}$ and sampling of the previous submatrix. The GEMM calculation sequence in line 4 of Algorithm 10 is reversed from right to left based on the associative law of matrix-matrix multiplication so as to avoid generating a large projection matrix of size $m \times m$ in the process:

$$\mathbf{Y}_j \leftarrow \underbrace{\overleftarrow{\mathbf{A}_{(:,\beta_j)}\mathbf{A}_{(:,\beta_j)}{}^{\top} \cdots \mathbf{A}_{(:,\beta_j)}\mathbf{A}_{(:,\beta_j)}{}^{\top}}}_{q \quad \text{(power iteration)}} \mathbf{A}_{(:,\beta_j)}\mathbf{\Omega}_j, \tag{4.5}$$

in which the long arrow on the top represents the order of matrix multiplication.

To orthonormalize the sample matrix $\mathbf{Y}$, instead of using a classical Gram-Schmidt (p.254 of [34]) or Cholesky QR (p.163 of [34]), we use the Communication avoiding QR (CAQR) factorization proposed by Demmel *et al.* [16]. CAQR has a lower communication cost compared to Householder QR [34]. The in-core GPU implementation [5] achieves fewer data accesses between GPU and GPU memory. In our test, the in-core CAQR runs roughly 1.5× faster than MAGMA library [99]. Note that we used the `geqrf()` routine for QR factorization and `orgqr()` routine for generating matrix $\mathbf{Q}$ in MAGMA implementation. In addition, since CAQR is built on the block Householder QR [16], it has intrinsically higher numerical stability than Gram-Schmidt and Cholesky QR.

For computing SVD of the small matrix $\mathbf{B}$ on the GPU, we compared `gesvd()` routines provided by MAGMA and cuSolver [79] libraries and found that there were not much performance difference in terms of both speed and accuracy. We therefore chose cuSolver included in the CUDA library to keep the implementation simple and portable. We implemented BRSVD as a general solver, which can be extended to different precisions. Note that all the following experiments were conducted in double-precision.

4.4 Experiments

In this section, we use numerical examples to compare the proposed algorithm with several existing ones in terms of computational performance and accuracy. Section 4.4.1 gives the comparison objects and Section 4.4.2 describes the experiment environment and setup. In Section 4.4.3, we use synthetic low-rank matrices with different sizes to compare BRSVD and RVSD in various computing environments. Section 4.4.4 focuses on validating the numerical stability using matrices with different singular spectrum decay patterns.

4.4.1 Performance Comparison

To assess the performance of the proposed BRSVD method, we compare with other RSVD implementations listed below. All the implementations are carefully optimized so as to yield the best performance in each setting.

1. `RSVD by cuBLAS-XT`: This implements Algorithm 9 using the cuBLAS-XT package [76]. cuBLAS-XT is a BLAS3 routines provided by the vendor that can process data larger than the GPU memory. cuBLAS-XT uses 2D partition in calculating out-of-core GEMM. cuBLAS-XT frees users from dealing with GPU memory allocation and CPU-GPU communication. However, users cannot control the reuse of the transferred data.

2. `CPU`: This is a straightforward implementation of Algorithm 9 on multi-core CPUs, with which all the working data is processed on CPUs.

3. `in-core on GPU`: This is an in-core GPU implementation, with which all the working data is held on the GPU memory. Since there is no data communication between CPU and GPU, we expect this implementation to give a reference to the performance peak for accelerating RSVD on the GPU. However, this implementation only works for small scale data that can well fit in the GPU memory.

4.4.2 Experiment Environment and Setup

For evaluation, we used an NVIDIA Telsa V100 (Volta) GPU with 16 GB memory. A V100 was connected to the host via PCIe 3.0 interface. The theoretical peak in double precision is 7.5 Tflop/s. Although V100 can access its own memory at 900 GB/s, the speed of CPU to GPU data transfer is limited to 15.8 GB/s at maximum. cuBLAS 10.1 and cuSolver 10.1 were used for BLAS and solver routines, respectively. For CPU implementation, we used a system equipped with two Intel 8-core Xeon Silver 4114 processors with 384 GB DDR4-2666 memory. The theoretical peak in double precision is 0.9 Tflop/s for two CPUs. Intel MKL 2018.0.3 [48] was used for CPU BLAS and solver routines.

Regarding the data, we generated matrices with various shapes and sizes with different ranks. A low-rank input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with rank-k was created by a product of two low-dimensional matrices $\mathbf{A}_l \in \mathbb{R}^{m \times k}$ and $\mathbf{A}_r \in \mathbb{R}^{k \times n}$ that were both random Gaussian matrices. For selecting the block size n', we use the maximal block size by querying available memory size at runtime.

4.4.3 Performance Comparison Results

We first tuned the performance of RSVD by cuBLAS-XT. As shown in Table 4.1, the communication cost of out-of-core GEMM for double precision is $8mnl/b$ in item (2). If $b \geq l$, cuBLAS-XT is set to do 1D partition on the large input matrix $\mathbf{A}$, which yields the minimum communication cost of $8mn$. According to the roofline model [111], the maximum flop/s = $(\#flops/\#words) \times bandwidth = (l/4) \times bandwidth$. The theoretical peak bandwidth is 15.8 GB/s for CPU-GPU communication. The largest parameters used in our experiments were set to $m = 589,824$, $n = 18,432$, and $l = 1152$ for tall-skinny test cases. The largest parameters for square cases are set to $m = 104,267$, $n = 104,267$, and $l = 814$. Therefore, the maximum theoretical peaks are limited to 4.5 Tflop/s and 3.2 Tflop/s for tall-skinny and square cases, respectively. The theoretical peak performance of V100 GPU is 7.5 Tflop/s, which means the performances in all test cases are bandwidth bound. Items (3) and (5) in Table 4.1 have the same results as item (2). In the performance tuning, we set the b with different sizes and found that $b = 4096$ gave the best performance for both tall-skinny and square cases. We then set $b = 4096$ for cuBLAS-XT in all the following experiments. Note that 4096 is larger than the maximum l value in the following setups.

Figure 4.2 shows the running time of RSVD and BRSVD with different power iteration number q values. In each experiment, the ratio of matrix dimensions (m, n) and rank k were fixed, and the performance was measured by varying the size of input data. For the attained performance, all measurements include the CPU-GPU data transfer time. As mentioned in Section 4.3, with $q = 0$, BRSVD and RSVD are exactly the same algorithm. The results for $q = 0$ in both Figs. 4.2(a) and (b) are almost overlapped, which means that the performance results of both implementations are close, with the same computational and communication cost. The BRSVD shows its advantage over RSVD with a larger q value. With $q = 3$, BRSVD achieves 1.6× and 1.7× maximum acceleration against `RSVD by cuBLAS-XT` for tall-skinny and square matrices, respectively.

Figure 4.3 summarizes the performance comparison with multi-core CPU and in-core results in Tflop/s. The performance peaks of in-core on GPU were 2.57 Tflop/s and 1.88 Tflop/s for tall-skinny and square matrices, respectively. These can be used as references for other out-of-core curious space. The experimental results in Figs. 4.3(a) and (b) show stable performance of BRSVD compared to `RSVD by cuBLAS-XT`. For data size smaller than 20 GB, BRSVD and `RSVD by cuBLAS-XT` both showed lower performance than in-core on GPU due to the high communication cost for initial and final data transfer between CPU and GPU. As we increased the data size, where the size of the working data exceeded the GPU memory capacity at 16 GB, the performance of BRSVD gradually increased and reached 2.57 Tflop/s for tall-skinny matrices and 2.17 Tflop/s for square ones. We profiled each GEMM operation by the NVIDIA Visual Profiler [80] and found the performance of GEMM peaks at 2.9 Tflop/s for tall-skinny and 2.3 Tflop/s for square. This performance drop rooted from the different matrix shapes of two setups. The performance peak of GEMM varies according to the sizes and shapes of the input matrices. For a detailed GEMM performance analysis, please refer to [105]. Results of large matrices showed slightly higher performance compared to in-core on GPU. This performance improvement came from the reduced proportion of execution overheads in processing large-scale data. Regarding `RSVD by cuBLAS-XT`, the performance peaks were 1.82 Tflop/s and 1.44 Tflop/s for tall-skinny and square matrices,

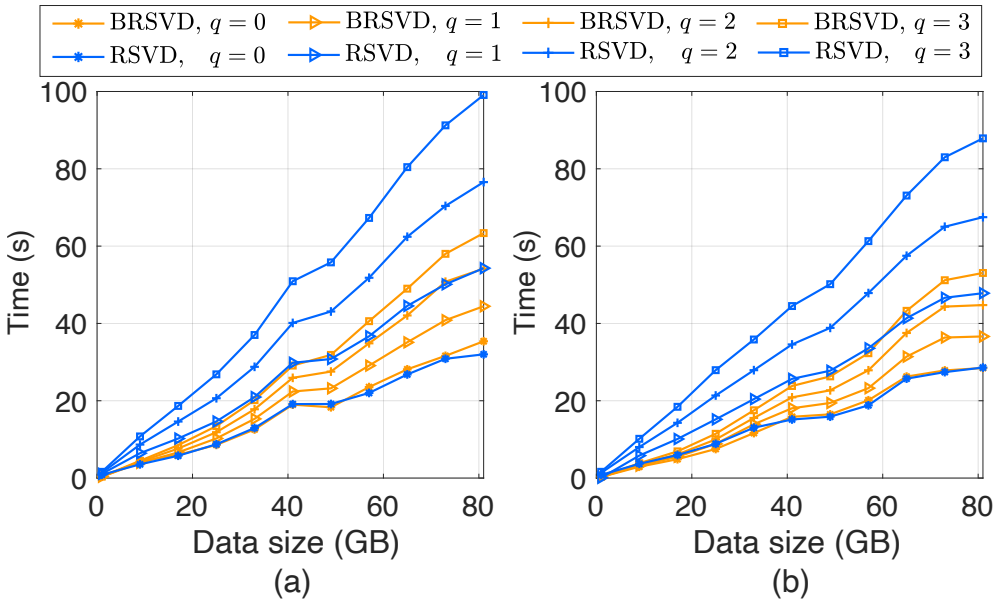

Figure 4.2: Performance comparison of BRSVD and RSVD with different power iteration number q. (a) and (b) show the execution time of tall-skinny matrices ($m : n : k = 1024 : 32 : 1$) and square ($m : n : k = 256 : 256 : 1$) ones with different q. The partition number for BRSVD was set to $s = 10$. The sampling parameters were set as: $o = k$ for all setups.

respectively.

Figures 4.3(c) and (d) show the breakdown analysis of the running time of BRSVD. Figure 4.3(c) shows that GEMM routines in sampling, power iteration, and forming $\mathbf{B}$ dominate the running time for tall-skinny matrices. For square matrices, due to the increased portion of SVD computation and declined GEMM performance which render low flop/s for both CPU and GPU, the overall performance in Fig. 4.3(d) is reduced by 15.5% compared with the tall-skinny curious space.

To further evaluate the actual amount of communication between CPU and GPU, we profiled the communication and running time of `RSVD by cuBLAS-XT` and BRSVD. Figure 4.4 gives the profiling results in sampling and power iteration. Figure 4.4(a) shows that the data transfer of `RSVD by cuBLAS-XT` fully saturated the CPU-GPU bandwidth which rendered the performance to be data transfer bound. Therefore, the performance can not be easily improved by employing more GPUs. Figure 4.4(b) shows that BRSVD effectively reduced the amount of data transfer which moved the performance bottleneck to computation. Because the parallelizability of BRSVD, its implementation on multiple GPU accelerators is expected to achieve further acceleration.

4.4.4 Accuracy Evaluation

Now we evaluate the accuracy and robustness of the proposed algorithm. In the setup, the dimension of matrix $\mathbf{A}$ was fixed to $10^3 \times 10^3$. We consider several synthetic matrices with different singular value decay patterns to assess the approximation accuracy of BRSVD.

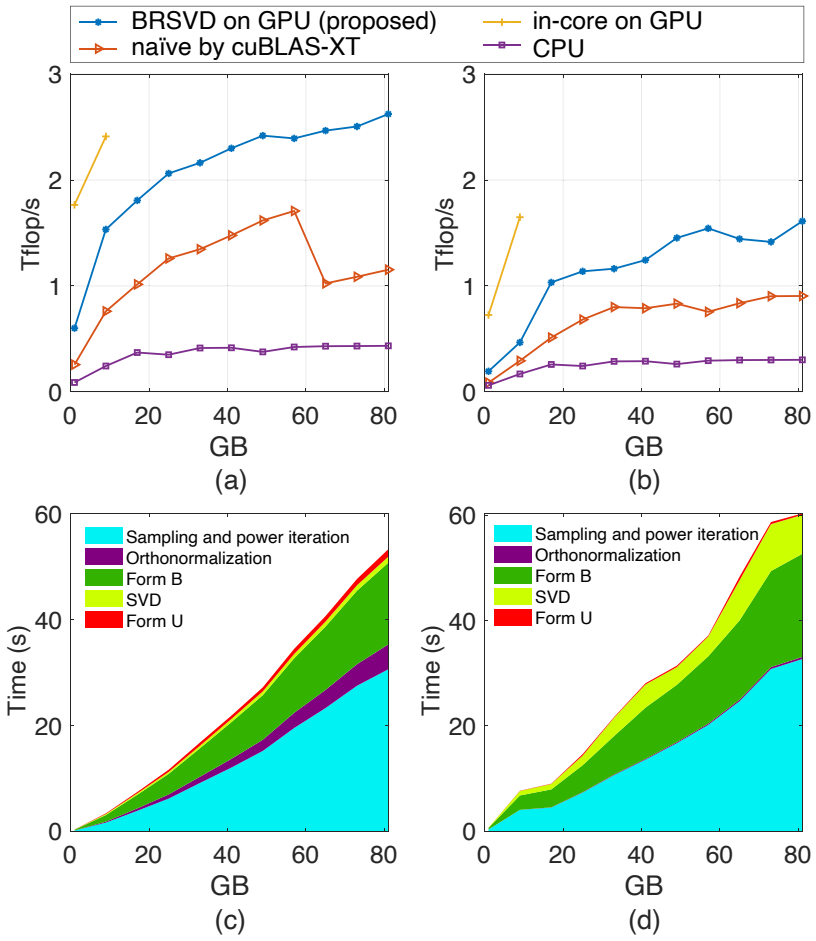

Figure 4.3: Performance comparison for different data sizes in double precision. (a) and (b) show overall performance of tall-skinny matrices ($m : n : k = 1024 : 32 : 1$) and square ($m : n : k = 256 : 256 : 1$) ones in Tflop/s. Tflop/s is calculated as #flops (Table 4.1) divided by the measured running time. (c) and (d) show the measured time breakdown of BRSVD for tall-skinny and square matrices. The parameters were set as: $o = k$ and $q = 2$.

1. Geometric and exponential decays of singular spectrum: We used two different singular value decay patterns here: geometric and exponential decays. For the geometric decay, the j-th singular value σ_j was defined to have the form of $\sigma_j = \sigma_1 g^{j-1}$. For the exponential decay, it was defined as $\sigma_j = \sigma_1 \exp(-j/w)$. As illustrated in Fig. 4.5(a), the parameters were set to $g = \{0.99, 0.9\}$ and $w = \{160, 50\}$, respectively.

2. Low-rank patterns: In addition to the geometric and exponential decays, we used the experiment setup proposed in [101]. The matrices had a fixed t leading singular values and a tail with polynomial or exponential decay. For matrices with a polynomial tail, the spectrum have the form

$$\mathbf{\Sigma} = \text{diag}(\underbrace{1, \ ..., \ 1}_{t}, \ 2^{-p}, \ 3^{-p}, \ ..., \ (n-t+1)^{-p}).$$

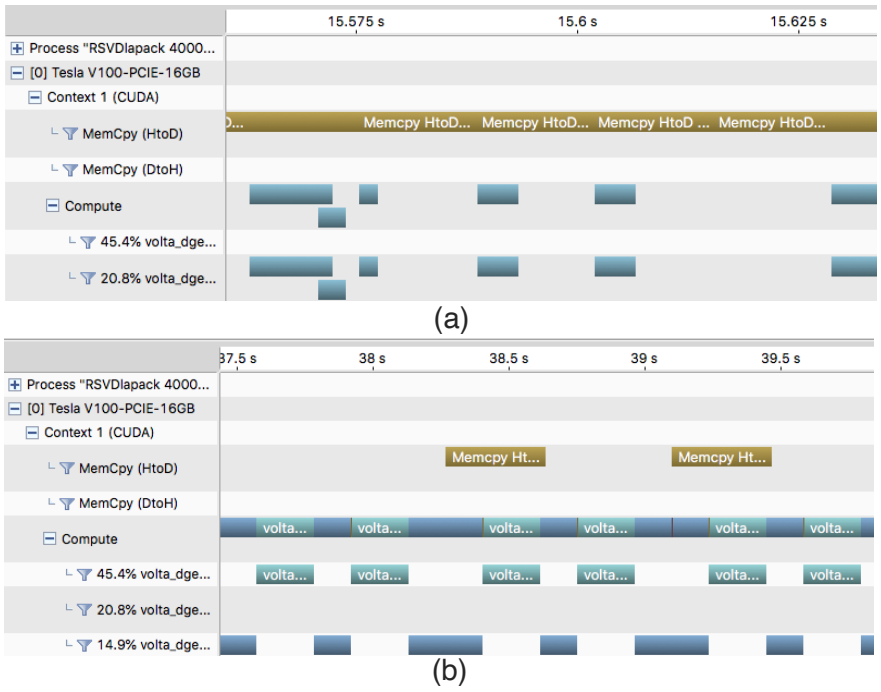

Figure 4.4: Profiling results for `RSVD by cuBLAS-XT` (a) and BRSVD (b). Beige and dark blue denote communication time and computation time, respectively. The data size was set to $m = 4 \times 10^5, n = 10^4$ and $k = 400$ (40 GB for the input matrix). Other parameter is set as: $q = 2$. Note that the time scales are different in (a) and (b) to make illustration clearer.

For matrices with a exponential tail, the form of spectrum is defined as

$$\mathbf{\Sigma} = \mathrm{diag}(\underbrace{1, \ ..., \ 1}_{t}, \ 10^{-h}, \ 10^{-2h}, \ ..., \ 10^{-(n-t)h}).$$

As plotted in Fig. 4.5(b), the parameters were set to $t = 10$, $p = \{1, 2\}$ and $h = \{0.25, 1\}$.

The left singular matrix $\mathbf{U}$ and right singular matrix $\mathbf{V}$ are generated as random orthogonal matrices. The test matrices are generated by matrix-matrix multiplication as $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$.

For assessing the accuracy, we used two different measures, namely, relative and actual approximation errors [101]. The *relative approximation error* is defined as

$$e_1 = \frac{\|\mathbf{A} - \hat{\mathbf{A}}\|_F}{\tau_{l+1}(\mathbf{A})} - 1,$$

where $\hat{\mathbf{A}}$ denotes the approximations of $\mathbf{A}$ obtained by a matrix decomposition algorithm and $\tau_{l+1}(\mathbf{A})$ denotes the root sum of squared singular values after l-th [101]. The *actual approximation error* is defined as

$$e_2 = \frac{\|\mathbf{A} - \hat{\mathbf{A}}\|_F}{\|\mathbf{A}\|_F},$$

which evaluates the difference from the ground truth.

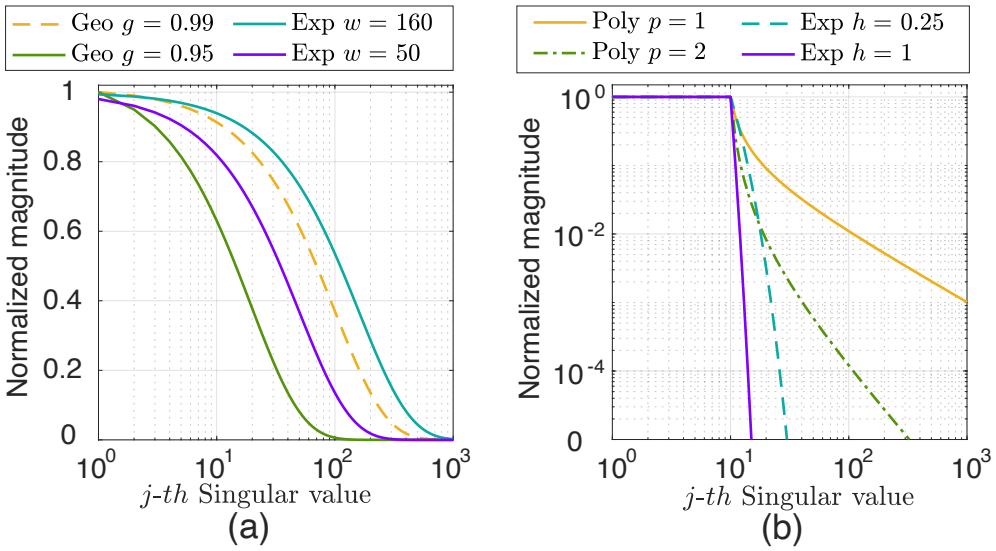

Figure 4.5: Singular spectrum decay pattern of matrices used in experiments. (a) shows geometric and exponential decay of singular value. (b) shows a low-rank with decaying tail.

Figure 4.6 depicts the relative approximation errors e_1 of the BRSVD in comparison to RSVD using (a) geometric and exponential decay patterns and (b) low-rank patterns. For the parameters of this experiment, we used $k = o = 10$ and $q = 2$ in all the conditions. The result shows that the proposed BRSVD effectively gives a close approximation to RSVD regardless of the number of batches. In particular, BRSVD is shown effective for a matrix with a slow singular value decay pattern (Exp $w = 160$). For matrices with a faster decay rate, the result indicates that a moderately small batch size can yield good approximation.

We now show three test patterns to take a closer look at actual approximation errors e_2 using different algorithms in Fig. 4.7. We include comparisons with a single-pass RSVD algorithm [118], which is the most accurate approximation among several state-of-the-art single-pass RSVD algorithms. We can see that BRSVD gives substantially closer to the optimal result (RSVD with power iterations) than RSVD without power iteration ($q = 0$ case) and single-pass algorithms. As the result shows, the proposed BRSVD achieves good accuracy with a reduced data accesses.

Table 4.2 shows the accuracy of each algorithm with different q values. We set the partition number $s = 10$ for BRSVD, which is a normal partition number for the out-of-core GPU computation. We found that for the geometric and exponential decay patterns, BRSVD requires approximately one more iteration to acquire a comparable accuracy of RSVD. For the polynomial low-rank pattern, BRSVD failed to obtain the same level of accuracy as RSVD. Regarding the exponential low-rank pattern, power iteration decreased the error of BRSVD. In contrast, the error of RSVD increased with the increasing q values. Those results also show that it requires the users to set sampling rate k and power iteration number q to find the optimal values for unknown matrix inputs.

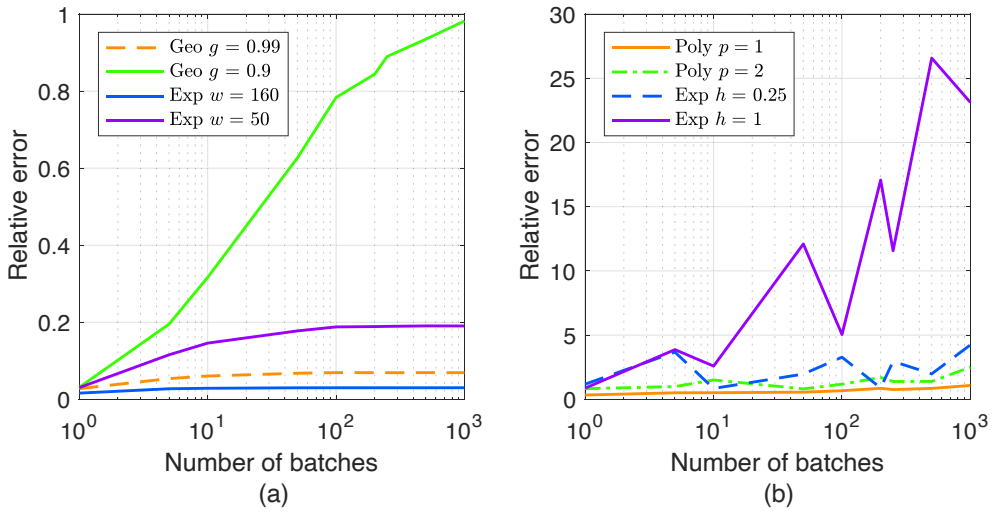

Figure 4.6: Result of relative error w.r.t. the varying number of batches. (a) Geometric and exponential decay pattern. (b) Low-rank pattern.

Table 4.2: Accuracy comparison with different power iteration number q. The parameters were set as: $k = o = 20$ and $s = 10$. The results were calculated by actual approximation error.

Data	$q = 0$		$q = 1$		$q = 2$		$q = 3$	
	RSVD	BRSVD	RSVD	BRSVD	RSVD	BRSVD	RSVD	BRSVD
Geo $g = 0.99$	0.898	0.898	0.850	0.867	0.836	0.858	0.829	0.854
Exp $\omega = 160$	0.936	0.936	0.905	0.918	0.896	0.913	0.891	0.910
Poly $p = 1$	0.178	0.178	0.093	0.125	0.091	0.128	0.09	0.131
Exp $h = 1$	4.21e-11	4.21e-11	1.42e-06	3.33e-11	1.30e-04	3.23d-11	3.36e-04	1.71e-11

4.4.5 Algorithm Comparison

In this subsection, we compared the proposed methods in Chapter 3 in Table 4.3. We first compare the Fused method (Chapter 3) and BRSVD. From Table 4.3, we can see that the computational costs for Fused and BRSVD are the same for Fused and BRSVD. However, the communication costs are different. The communication cost of the Fused method is dependent on the power iteration number q, which means the higher accuracy, the higher communication cost. Comparatively, BRSVD only access $\mathbf{A}$ with two passes which is independent of q.

We then compare the Gram method and BRSVD. Both two methods access the matrix A for 2 times. However, the gram method will have a higher computational cost than BRSVD and only works well for tall-skinny matrices.

Regarding the applications of methods proposed in Chapter 3 and BRSVD. The methods proposed in Chapter 3 mainly focused on reducing the CPU-GPU data transfer for RVSD. Those methods reduced the data transfer at the cost of increased computational cost. However, the CPU-GPU bandwidth remains the bottleneck for accelerating RSVD on GPUs. As shown in Table 4.3, the Fused and Gram are exactly the same as the original RSVD. Therefore, the proposed methods have a rigid error-bound for decomposition results. They fit for applications that require a high decomposition accuracy. This chapter focused on redesigning the RSVD algorithm to remove the bandwidth constraint. The experimental re-

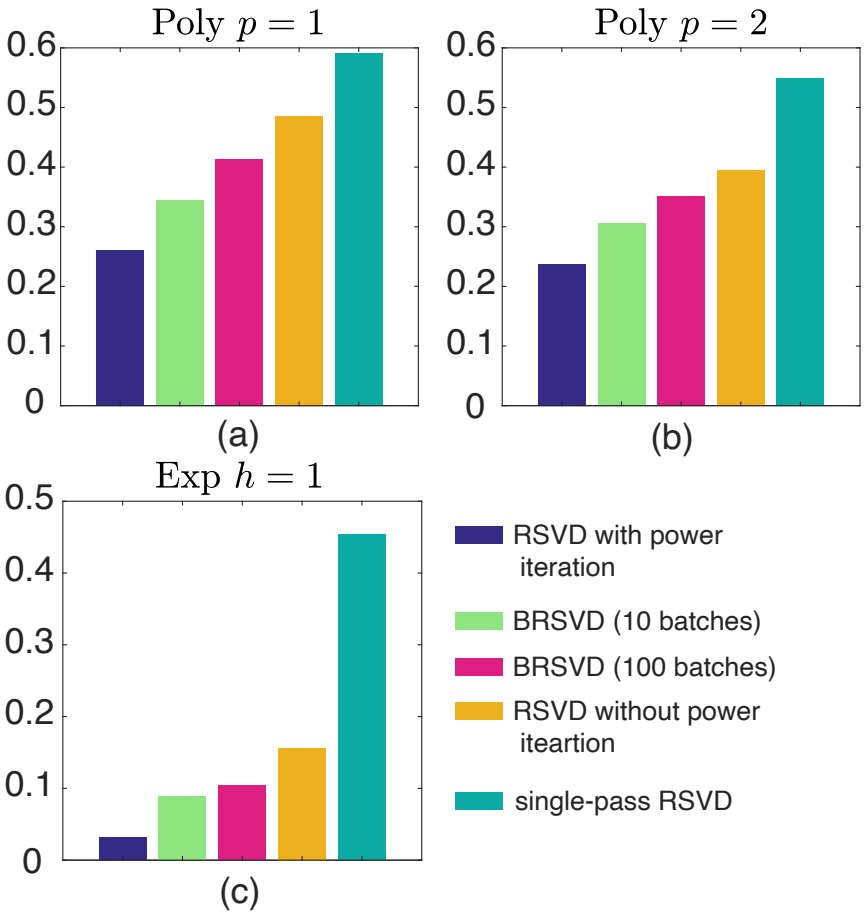

Figure 4.7: Actual error with different singular spectrum decay patterns which are (a) Poly $p = 1$, (b) Poly $p = 2$, and (c) Exp $h = 1$, respectively.

sults show that the performance bottleneck has been moved to the GPU peak performance. Since the proposed method is hardware independent, BRSVD will fit better for a multi-node computing environment where the ratio of communication cost to computational cost is much higher than a single node environment.

Table 4.3: Comparison of the proposed methods in Chapter 3 and this Chapter. #flops denotes the computational cost for the sampling and power iteration part. Pass of **A** represents the communication cost of transferring matrix **A** between CPU and GPU. q denotes the power iteration number.

	Fused (Chapter 3)	Gram (Chapter 3)	BRSVD
#flops	$(2q+1)mnl$	$mnl^2 + qn^2l + kmnl$	$(2q+1)mnl$
Pass of **A**	$q+1$	2	2
Accuracy	exact	exact	approximation

4.5 Applications of BRSVD

In this section, we demonstrate the performance of the proposed method on a diverse collection of data. All data used in the experiments exceeded the GPU memory capacity (16 GB). In Section 4.5.1, we take Eigenface [103] and computed tomography data as examples to assess the proposed BRSVD in comparison to both deterministic SVD and RSVD. The hardware used in the experiment was the same in Section 4.4.2.

4.5.1 Eigenfaces

We apply the proposed BRSVD method to two real datasets. First, we used the extended Yale face dataset [32] to assess the leading left singular vectors, which are known as eigenfaces [103]. The database contains 2383 cropped face images, and each image has the resolution of 168×192. Then a matrix of $32,256 \times 2383$ was constructed as the input matrix $\mathbf{A}$. Figures 4.8(a) and (b) show the leading 4 eigenfaces calculated by BRSVD and deterministic SVD, respectively. The results show that our proposed algorithm has no discernible deviation from the deterministic method.

Second, we used a large dataset derived from the facial recognition technology (FERET) dataset [87]. $11,333$ images are of size 512×768. The remaining $14,056$ images were resized to the resolution of 512×768 so that all images can be processed in the same matrix; thus the input matrix size becomes $393,216 \times 25,389$. This matrix with double precision entries will take up about 80 GB for storage. As shown in Fig. 4.8(c), the eigenfaces are blurred compared with the Yale dataset due to that FERET contains portraits taken from different directions and distances. Figure 4.8(d) compares the leading 32 singular values calculated by BRSVD and RSVD, which indicates that BRSVD gives a very close approximation of singular values to the RSVD method.

We compared the accuracy of approximated left singular vectors with the results calculated by deterministic SVD. The relative error of singular vectors is calculated as:

$$e_3 = \frac{\|\mathbf{v_i} - \hat{\mathbf{v}}_\mathbf{i}\|}{\|\mathbf{v_i}\|},$$

where $\mathbf{v_i}$ and $\hat{\mathbf{v}}_\mathbf{i}$ denote the i-th singular vector obtained by the deterministic SVD and the target algorithm, respectively.

As shown in Table 4.4, the accuracy of BRSVD is of the same order of magnitude as RSVD for both two datasets, which means BRSVD can accurately approximate the leading singular vectors of real data.

4.5.2 Computed Tomography

SVD has been used in CT reconstruction and denoising [1, 104]. To evaluate the performance on large scale data, we used a standard Shepp-Logan phantom [94] as the input dataset. Each entry of the data is a double precision gray-scale voxel of the phantom. 2048 slices of CT image with a resolution of 2048×2048 ($= 4,194,304$) were vectorized to form a $4,194,304 \times 2048$ double precision matrix. This matrix took up 64 GB for storage. As

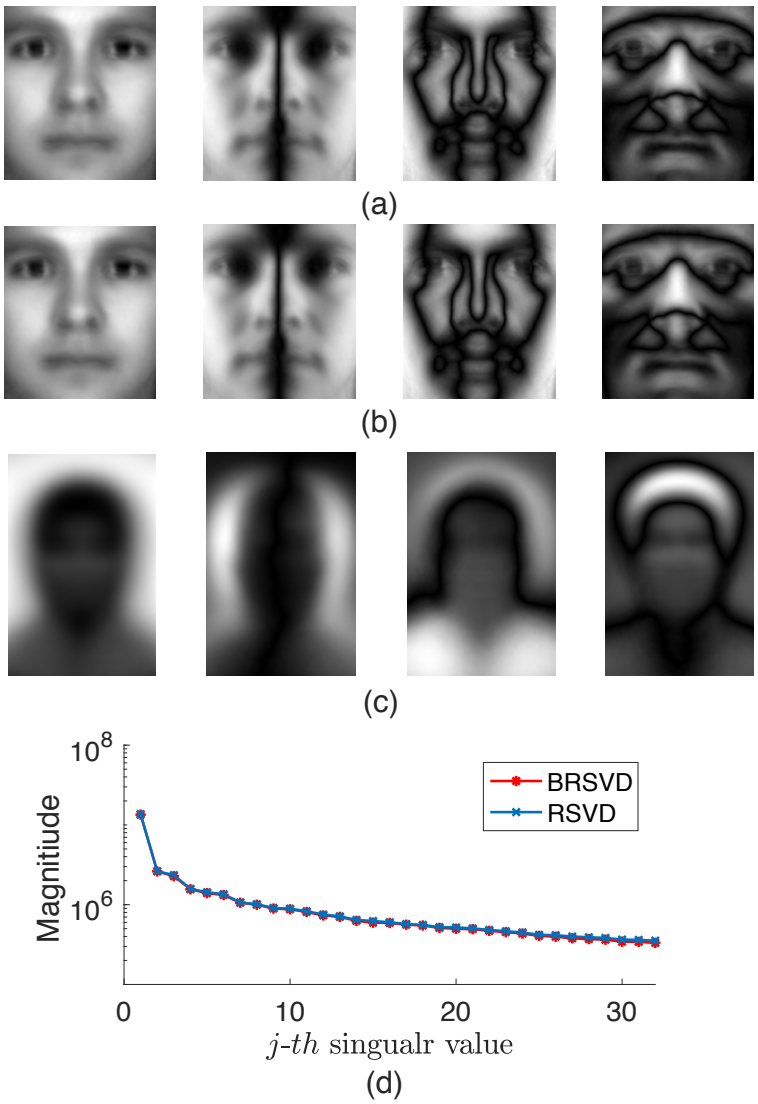

Figure 4.8: Comparison of BRSVD and deterministic rank-k SVD. (a) and (b) shows the eigenfaces from the Extended Yale Face dataset [32] approximated by BRSVD and deterministic rank-k SVD, respectively. (c) shows the eigenfaces computed from the FERET dataset [87] approximated by BRSVD. (d) shows the singular values calculated by BRSVD and RSVD for the FERET dataset.

shown in Table 4.5, BRSVD and RSVD achieved 6.3× and 3.7×, respectively, against the multi-core CPU implementation.

4.6 Conclusions

This paper presented a fast RSVD algorithm named BRSVD that fully utilizes GPU accelerators. We provided a detailed study of the data access efficiency of the proposed algorithm, and demonstrated the performance with detailed experimental results and real applications. The proposed algorithm allows us to see the directions of randomized algorithm development

Table 4.4: Accuracy comparison for the leading four left singular vectors. The sampling parameters were set as: $k = o = 32$ for Yale Face and $k = o = 128$ for FERET, respectively. Other parameters were set as: $q = 2$ and $s = 10$ for both datasets.

	Yale Face		FERET	
	RSVD	BRSVD	RSVD	BRSVD
$\mathbf{v}_1$	1.47e-5	3.51e-5	9.46e-7	4.77e-6
$\mathbf{v}_2$	1.52e-4	3.39e-4	8.22e-5	3.20e-4
$\mathbf{v}_3$	3.43e-3	9.13e-3	6.44e-5	5.21e-4
$\mathbf{v}_4$	2.92e-3	9.91e-3	5.72e-4	2.05e-3

Table 4.5: Performance comparison of BRSVD, RSVD by cuBLAS-XT, and RSVD on CPU. All experiments were conducted in double precision. The parameters used in the experiments were set as: $k = o = 64$, $q = 2$, and $s = 8$.

	BRSVD	RSVD by cuBLAS-XT	RSVD on CPU
Time (s)	23.1	39.4	145.3

in evolving computing environments. Most immediate is the independent operations which fit accelerators. Regarding the accuracy of the proposed algorithm, our proposed two-pass algorithm has a higher accuracy than the single-pass one. However, it has a slightly lower accuracy than the standard multi-pass with the same computational cost.

Our future work mainly focuses on showing the mathematical proof of BRSVD. Other work includes mixed-precision randomized algorithms and enabling BRSVD to run in a multi-GPU cluster environment to achieve further acceleration.

Chapter 5

Conclusions

In this chapter, we summarize our work and discuss future work.

5.1 Summary of This Thesis

In this work, we investigated two algorithms: cone beam CT (Chapter 2) and RSVD (Chapter 3 and 4). Both works focus on enabling those algorithms to process large-scale data on GPUs. Both our proposed methods are based on the divide-and-conquer strategy in dealing with the input and output data.

In Chapter 2, we proposed a cache-aware optimization method for cone beam reconstruction. The previous work [82] has revealed that the GPU bandwidth instead of CPU-GPU bandwidth is the performance bottleneck for the G80 GPU. Our work started with a comparatively newer GPU. We analyzed the Cone Beam CT algorithm and GPU architecture. We found the improved cache capacity can be utilized to reduce the off-chip data transfer. The proposed method accelerated the FDK algorithm via three strategies: (1) an improved loop organization strategy, (2) an improved data structure, and (3) an I/O-included pipeline. We also presented tuning guidelines for determining the best configuration for the granularity and shape of thread blocks. Through the proposed method, we successfully moved the bottleneck of this application from the bandwidth of GPU memory to the cache hardware throughput. We showed that the sophisticated cache hardware on the new GPUs gives a new perspective on accelerating cone beam reconstruction using GPU. Compared to the previous strategy of emphasizing memory coalescing to reduce GPU memory access, the proposed cache-aware strategies focus on optimizing the cache-hit rate of the GPU.

In Chapter 3, we first analyzed the performance bottleneck and data access pattern of accelerating RSVD on GPUs. The main building block of RSVD is GEMM, and GEMM has been highly optimized. Compared with the Cone Beam CT, the performance peak of GEMM is close to the theoretical peak for in-core computation, which means the GPU memory is not the bottleneck. We used a roofline model to benchmark the out-of-core GEMM performance. The benchmark results revealed that the CPU-GPU bandwidth is the performance bottleneck. Therefore, different from the Cone Beam CT in Chapter 2, our main focus in Chapter 3 was to reduce the CPU-GPU data transfer instead of improving the cache performance. The proposed methods demonstrated that reducing the communication cost at

the expense of increased computational cost is a feasible strategy to improve the overall performance. We proposed data transfer reducing methods that relieve the constraint imposed by CPU-GPU bandwidth for RSVD. The proposed Fused method effectively accelerated the RSVD up to 1.9× compared with a straightforward method that deploys the highly-tuned GEMM scheme and the 1D data partition scheme. The Gram method achieved up to 5.2× speedup for tall-skinny matrices compared with a straightforward method.

In Chapter 4, we explored potential techniques for improving the performance of RSVD on the heterogeneous architecture. Our proposed methods in Chapter 3 relaxed the constrained of CPU-GPU bandwidth. However, RSVD still remained an CPU-GPU bandwidth bound problem in Chpater 3. In Chapter 4, we redesigned the RSVD algorithm to further reduce the CPU-GPU data transfer. The proposed BRSVD algorithm successfully moved the bottleneck of RSVD from bandwidth bound to the computational ability of the GPU. Our work demonstrated a sample of reducing communication costs at the expense of losing a small fraction of accuracy.

5.2 Future Work

Recent developments in GPUs provide new possibilities for HPC. One of them is a new feature called mixed precision. Mixed-precision is a computational method that combines the use of different numerical precisions. The mixed-precision allows users to reduce the computational cost and storage cost without extra efforts for typecasting. From the Pascal GPU architecture [73], NVIDIA starts to provide support for mixed-precision computation, including double-precision (64-bit), single-precision (32-bit), and half-precision (16-bit). This function mainly targets the rapidly growing needs for accelerating neural networks on GPUs. Half-precision on GPUs has twice the throughput of single-precision arithmetic computation and 4× the throughput of double precision. The theoretical maximum performance of V100 GPU reaches 31.4 Tflop/s with half-precision, 15.7 Tflop/s with single-precision, and 7.8 Tflop/s with double-precision. For accelerating image processing applications, data from cameras or other kinds of sensors do not require high-precision floating point computation. For cone beam CT systems that do not have a high dynamic range X-ray sensor to span the range of 32-bit floating point, it is possible to replace most of 32-bit computations with 16-bit and make the output voxel 32-bit. These mixed-precision computation can be achieved in the CUDA kernel level.

For accelerating linear algebra applications, the cuBLAS library provides API level mixed-precision which allows users to access BLAS3 API without extra storage and computation for converting precisions. Based on those new functions, mixed-precision iterative refinement has been proposed to the LU factorization in solving $\mathbf{Ax} = \mathbf{b}$ by GPUs [38]. Replacing intermediate computation inside other linear algebra algorithms is in progress [30]. Besides, extending the mixed-precision computation to randomized algorithms is a possible research direction.

Bibliography

[1] K. Abhari, M. Marsousi, J. Alirezaie, and P. Babyn. Computed tomography image denoising utilizing an efficient sparse coding algorithm. In *Proceedings of the 11th International Conference on Information Science, Signal Processing and their Applications (ISSPA)*, pages 259–263, 2012.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(012037), 2009.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the the AFIPS Conference*, pages 483–485, 1967.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, third edition, 1987.

[5] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 48–58, 2011.

[6] H. Andrews and C. Patterson. Singular value decomposition (SVD) image coding. *IEEE Transactions on Communications*, 24(4):425–432, 1976.

[7] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[8] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Proceedings of the 20th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2008.

[9] J. G. Blas, M. Abella, F. Isalia, J. Carretero, and M. Desco. Surfing the optimization space of a multiple-GPU parallel implementation of a X-ray reconstruction algorithm. *The Journal of Systems and Software*, 95:166–175, Sept. 2014.

[10] E. J. Candès, X. Li, Y. Ma, and J. Wright. Robust principal component analysis? *Journal of the ACM*, 58(3):1–37, 2011.

[11] J. Chen, N. Xiong, X. Liang, D. Tao, S. Li, K. Ouyang, K. Zhao, N. DeBardeleben, Q. Guan, and Z. Chen. TSM2: Optimizing tall-and-skinny matrix-matrix multiplication on GPUs. In *Proceedings of the 33rd International Conference on Supercomputing (ICS)*, pages 106–116, 2019.

[12] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Staney, D. Walker, and R. Whaley. LAPACK working note 95 ScaLAPACK: A portable linear algebra library for distributed memory computers-design issues and performance. Technical Report UT CS-95-283, University of Tennessee, Knoxville, TN, USA, 1995.

[13] E. D'Azevedo and J. Hill. Parallel LU factorization on GPU cluster. *Procedia Computer Science*, 9:67–75, 2012.

[14] F. De la Torre and M. J. Black. A framework for robust supspace learning. *International Journal of Computer Vision*, 54(1/2/3):117–142, 2003.

[15] J. Demmel. Communication-avoiding algorithms for linear algebra and beyond. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 585, 2013.

[16] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[17] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang. Communication avoiding rank revealing QR factorization with column pivoting. *SIAM Journal on Matrix Analysis and Applications*, 36(1):55–89, 2015.

[18] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki. The singular value decomposition: Anatomy of optimizing an algorithm for extreme scale. *SIAM Review*, 60(4):808–865, 2018.

[19] J. Dongarra, S. Gottlieb, and W. T. Kramer. Race to exascale. *Computing in Science & Engineering*, 21(1):4–5, 2019.

[20] J. Dongarra, C. Moler, J. Bunch, and G. Stewart. *LINPACK Users' Guide*. SIAM, 1979.

[21] J. J. Dongarra, H. W. Meuer, and E. Strohmaier. TOP500 supercomputer sites. *Supercomputer*, 13:89–111, 1997.

[22] P. Drineas, R. Kannan, and M. W. Mahoney. Fast monte carlo algorithms for matrices I: Approximating matrix multiplication. *SIAM Journal on Computing*, 36(1):132–157, 2006.

[23] P. Drineas and M. W. Mahoney. A randomized algorithm for a tensor-based generalization of the singular value decomposition. *Linear Algebra and its Applications*, 420(2/3):553–571, 2007.

[24] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte. Medical image processing on the GPU — past, present and future. *Medical Image Analysis*, 17(8):1073–1094, Dec. 2013.

[25] N. B. Erichson and C. Donovan. Randomized low-rank dynamic mode decomposition for motion detection. *Computer Vision and Image Understanding*, 146:40–50, 2016.

[26] L. A. Feldkamp, L. C. Davis, and J. W. Kress. Practical cone-beam algorithm. *Journal of the Optical Society of America*, 1(6):612–619, June 1984.

[27] C. Frankenberg, C. O'Dell, J. Berry, L. Guanter, J. Joiner, P. Köhler, R. Pollock, and T. E. Taylor. Prospects for chlorophyll fluorescence remote sensing from the Orbiting Carbon Observatory-2. *Remote Sensing of Environment*, 147:1–12, 2014.

[28] A. Frieze, R. Kannan, and S. Vempala. Fast Monte-Carlo algorithms for finding low-rank approximations. *Journal of the ACM*, 51(6):1025–1041, 2004.

[29] N. Gac, S. Mancini, and M. Desvignes. Hardware/software 2D-3D backprojection on a SoPC platform. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC)*, pages 222–228, Apr. 2006.

[30] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra. SLATE: design of a modern distributed and accelerated linear algebra library. In *Proceedings of the 31th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, number 26, 2019.

[31] M. Gates, S. Tomov, and J. Dongarra. Accelerating the SVD two stage bidiagonal reduction and divide and conquer using GPUs. *Parallel Computing*, 74:3–18, 2018.

[32] A. Georghiades, P. Belhumeur, and D. Kriegman. From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(6):643–660, 2001.

[33] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, 2(2):205–224, 1965.

[34] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 4th edition, 2012.

[35] A. Haidar, A. Abdelfatah, S. Tomov, and J. Dongarra. High-performance Cholesky factorization for GPU-only execution. In *Proceedings of the 10th Workshop on General Purpose GPUs (GPGPU)*, pages 42–52, 2017.

[36] A. Haidar, A. Abdelfattah, M. Zounon, S. Tomov, and J. Dongarra. A guide for achieving high performance with very small matrices on GPU: A case study of batched LU and Cholesky factorizations. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):973–984, 2018.

[37] A. Haidar, K. Kabir, D. Fayad, S. Tomov, and J. Dongarra. Out of memory SVD solver for big data. In *Proceedings of the 21st High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[38] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the 30th International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, number 47, 2018.

[39] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[40] E. Henry and J. Hofrichter. [8] singular value decomposition: Application to analysis of experimental data. *Methods in Enzymology*, 210:129–192, 1992.

[41] A. Höcker and V. Kartvelishvili. SVD approach to data unfolding. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 372(3):469–481, 1996.

[42] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, 2010.

[43] J. Hofmann, J. Treibig, G. Hager, and G. Wellein. Performance engineering for a medical imaging application on the Intel Xeon Phi accelerator. In *Proceedings of the 27th International Conference on Architecture fo Computating Systems (ARCS)*, pages 222–228, Feb. 2014.

[44] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417–441, 1933.

[45] F. Ino, Y. Okitsu, T. Kishi, S. Ohnishi, and K. Hagihara. Out-of-core cone beam reconstruction using multiple GPUs. In *Proceedings of the 7th IEEE International Symposium on Biomedical Imaging (ISBI)*, pages 792–795, Apr. 2010.

[46] F. Ino, S. Yoshida, and K. Hagihara. RGBA packing for fast cone beam reconstruction on the GPU. In *Proceedings of the SPIE Medical Imaging (MI)*, Feb. 2009. 8 pages (CD-ROM).

[47] Intel Corporation. Intel architecture instruction set extensions programming reference, Dec. 2013. `http://download-software.intel.com/sites/default/files/managed/71/2e/319433-017.pdf`.

[48] Intel Corporation. Developer Reference for Intel®Math Kernel Library - C. `https://software.intel.com/en-us/mkl-developer-reference-c/`, 2019.

[49] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[50] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. *ACM SIGPLAN Notices*, 46(6):142–151, 2011.

[51] H. Ji and Y. Li. GPU accelerated randomized singular value decomposition and its application in image compression. In *Proceedings of the Modeling, Simulation, and Visualization Student Capstone Conference*, pages 39–45, 2014.

[52] M. Kachelrieß, M. Knaup, and O. Bockenbach. Hyperfast parallel-beam and cone-beam backprojection using the cell general purpose hardware. *Medical Physics*, 34(4):1474–1486, Apr. 2007.

[53] J. Kastner, B. Harrer, G. Requena, and O. Brunke. A comparative study of high resolution cone beam X-ray tomography and synchrotron tomography applied to Fe- and Al-alloys. *NDT & E International*, 43(7):599–605, Oct. 2010.

[54] J. Kuczyński and H. Woźniakowski. Estimating the largest eigenvalue by the power and Lanczos algorithms with a random start. *SIAM Journal on Matrix Analysis and Applications*, 13(4):1094–1122, 1992.

[55] R. M. Larsen. Lanczos bidiagonalization with partial reorthogonalization. *DAIMI Report Series*, 27(537), 1998.

[56] M. Leeser, S. Mukherjee, and J. Brock. Fast reconstruction of 3D volumes form 2D CT projection data with GPUs. *BMC Research Notes*, 7(582), June 2014. 8 pages.

[57] E. Liberty, F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 104(51):20167–20172, 2007.

[58] Z. Lin, M. Chen, and Y. Ma. The augmented lagrange multiplier method for exact recovery of corrupted low-rank matrices. *arXiv preprint arXiv:1009.5055v3*, 2013.

[59] Z. Lin, A. Ganesh, J. Wright, L. Wu, M. Chen, and Y. Ma. Fast convex optimization algorithms for exact recovery of a corrupted low-rank matrix. Technical Report UILU-ENG-09-2214, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2009.

[60] B. Liu, J. Huang, L. Yang, and C. Kulikowsk. Robust tracking using local sparse appearance model and k-selection. In *Proceedings of the 24th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1313–1320, 2011.

[61] Y. Lu, F. Ino, and K. Hagihara. Cache-aware GPU optimization for out-of-core cone beam CT reconstruction of high-resolution volumes. *IEICE Transactions on Information and Systems*, E99-D(12):3060–3071, Dec 2016.

[62] K. Machin and S. Webb. Cone-beam X-ray microtomography of small specimens. *Physics in Medicine and Biology*, 39(10):1639–1657, Oct. 1994.

[63] M. W. Mahoney. Randomized algorithms for matrices and data. *Foundations and Trends®in Machine Learning*, 3(2):123–224, 2011.

[64] P.-G. Martinsson, V. Rockhlin, and M. Tygert. A randomized algorithm for the approximation of matrices. Technical Report YALEU/DCS/TR-1361, Yale University, Department of Computer Science, New Haven, CT, USA, 2006.

[65] T. Mary, I. Yamazaki, J. Kurzak, P. Luszczek, S. Tomov, and J. Dongarra. Performance of random sampling for computing low-rank approximations of a dense matrix on GPUs. In *Proceedings of the 27th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, number 60, 2015.

[66] A. Munshi. The OpenCL specification. In *Proceedings of the 21st IEEE Hot Chips Symposium (HCS)*, pages 1–314, 2009.

[67] K. Nakano. Asynchronous memory machine models with barrier synchronization. *IEICE TRANSACTIONS on Information and Systems*, 97(3):431–441, 2014.

[68] P. B. Noël, A. M. Walczak, J. Xu, J. J. Corso, K. R. Hoffmann, and S. Schafer. GPU-based cone beam computed tomography. *Computer Methods and Programs in Biomedicine*, 98(3):271–277, June 2010.

[69] NVIDIA Corporation. NVIDIA GeForce 8800 GPU architecture overview. Technical Report TB-02787-001_v01, NVIDIA Corporation, Nov. 2006.

[70] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, May 2012.

[71] NVIDIA Corporation. NVIDIA GeForce GTX 980, Nov. 2014.

[72] NVIDIA Corporation. Tuning CUDA applications for Maxwell, Sept. 2015.

[73] NVIDIA Corporation. NVIDIA Tesla P100 Whitepaper. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, 2016.

[74] NVIDIA Corporation. NVIDIA Tesla V100 GPU Architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017.

[75] NVIDIA Corporation. NVIDIA TURING GPU ARCHITECTURE, 2018.

[76] NVIDIA Corporation. CUBLAS Library User Guide. http://docs.nvidia.com/pdf/CUBLAS_Library.pdf, 2019.

[77] NVIDIA Corporation. CUDA C++ Programming Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2019.

[78] NVIDIA Corporation. CURAND Library Programming Guide v10.1. http://docs.nvidia.com/pdf/CURAND_Library.pdf, 2019.

[79] NVIDIA Corporation. CUSOLVER Library v10.1. http://docs.nvidia.com/pdf/CUSOLVER_Library.pdf, 2019.

[80] NVIDIA Corporation. Profiler User's Guide Version 10.1. http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf, Aug. 2019.

[81] T.-H. Oh, Y. Matsushita, Y.-W. Tai, and I. So Kweon. Fast randomized singular value thresholding for nuclear norm minimization. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4484–4493, 2015.

[82] Y. Okitsu, F. Ino, and K. Hagihara. High-performance cone beam reconstruction using CUDA compatible GPUs. *Parallel Computing*, 36(2/3):129–141, Feb. 2010.

[83] OpenACC-Standard.org. The OpenACC application programming interface, version 2.0, Aug. 2013.

[84] OpenMP Architecture Review Board. OpenMP application programming interface, version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, Nov. 2018.

[85] E. Papenhausen and K. Mueller. Rapid Rabbit: Highly optimized GPU accelerated cone-beam CT reconstruction. In *Proceedings of the Nuclear Science Symposium on and Medical Imaging Conference (NSS/MIC)*, Oct. 2013. 2 pages.

[86] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(11):559–572, 1901.

[87] P. J. Phillips, H. Moon, P. Rauss, and S. A. Rizvi. The FERET evaluation methodology for face-recognition algorithms. In *Proceedings of the 10th IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 137–143, 1997.

[88] V. Rokhlin, A. Szlam, and M. Tygert. A randomized algorithm for principal component analysis. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1100–1124, 2009.

[89] T. Sarlós. Improved approximation algorithms for large matrices via random projections. In *Proceedings of the 47th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 143–152, 2006.

[90] W. C. Scarfe and A. G. Farman. What is cone-beam CT and how does it work? *Dental Clinics of North America*, 52(4):707–730, Oct. 2008.

[91] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger. Fast GPU-based CT reconstruction using the common unified device architecture (CUDA). In *Proceedings of the Nuclear Science Symposium on and Medical Imaging Conference (NSS/MIC)*, pages 4464–4466, Oct. 2007.

[92] E. Serrano, G. Bermejo, J. G. Blas, and J. Carretero. High-performance X-ray tomography reconstruction algorithm based on heterogeneous accelerated computing systems. In *Proceedings of the 16th IEEE International Conference on Cluster Computing (CLUSTER)*, pages 331–338, Sept. 2014.

[93] J. Shen, K. Shigeoka, F. Ino, and K. Hagihara. GPU-based branch and bound method to solve large 0-1 knapsack problems with data-centric strategies. *Concurrency and Computation: Practice and Experience*, 31(4), Apr. 2019. e4954.

[94] L. A. Shepp and B. F. Logan. The Fourier reconstruction of a head section. *IEEE Transactions on Nuclear Science*, 21(3):21–43, June 1974.

[95] B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ikebe, V. Klema, and C. Moler. Matrix eigensystem routines — EISPACK guide. *Lecture Notes in Computer Science*, 6, 1977.

[96] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM Journal on Scientific Computing*, 23(6):2165–2182, 2002.

[97] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, fifth edition, 2016.

[98] Y. Sugimoto, F. Ino, and K. Hagihara. Improving cache locality for GPU-based volume rendering. *Parallel Computing*, 40(5/6):59–69, May 2014.

[99] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1–8, 2010.

[100] J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger, and G. Wellein. Pushing the limits for medical image reconstruction on recent standard multicore processors. *International Journal of High Performance Computing Applications*, 27(2):162–177, May 2013.

[101] J. A. Tropp, A. Yurtsever, M. Udell, and V. Cevher. Practical sketching algorithms for low-rank matrix approximation. *SIAM Journal on Matrix Analysis and Applications*, 38(4):1454–1485, 2017.

[102] O. Troyanskaya, M. Cantor, G. Sherlock, P. Brown, T. Hastie, R. Tibshirani, D. Botstein, and R. B. Altman. Missing value estimation methods for DNA microarrays. *Bioinformatics*, 17(6):520–525, 2001.

[103] M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. In *Proceedings of the 4th IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 586–591, 1991.

[104] D. Verhoeven. Limited-data computed tomography algorithms for the physical sciences. *Applied optics*, 32(20):3736–3754, 1993.

[105] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 20th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, number 31, 2008.

[106] S. Voronin and P.-G. Martinsson. RSVDPACK: An implementation of randomized algorithms for computing the singular value, interpolative, and CUR decompositions of matrices on multi-core and GPU architectures. *arXiv preprint arXiv:1502.05366v3*, 2, 2016.

[107] M. E. Wall, A. Rechtsteiner, and L. M. Rocha. Singular value decomposition and principal component analysis. In *A Practical Approach to Microarray Data Analysis*, pages 91–109. 2003.

[108] M. J. Wallace, M. D. Kuo, C. A. Binkert, R. C. Orth, and G. Soulez. Three-dimensional C-arm cone-beam CT: Applications in the interventional suite. *Journal of Vascular and Interventional Radiology*, 19(6):799–813, June 2008.

[109] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang. BLASX: A high performance level-3 BLAS library for heterogeneous multi-GPU computing. In *Proceedings of the 30th International Conference on Supercomputing (ICS)*, number 20, 2016.

[110] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 19th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, number 38, 2007.

[111] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[112] F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. *IEEE Transactions on Nuclear Science*, 52(3):654–663, June 2005.

[113] I. Yamazaki, M. Hoemmen, P. Luszczek, and J. Dongarra. Improving performance of gmres by reducing communication and pipelining global collectives. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1118–1127, 2017.

[114] I. Yamazaki, J. Kurzak, P. Luszczek, and J. Dongarra. Random sampling to update partial singular value decomposition on a hybrid CPU/GPU cluster. In *Proceedings*

of the 27th International Conference for High Performance Computing, Networking, Storage and Analysis (SC), number 59, 2015.

[115] I. Yamazaki, S. Tomov, and J. Dongarra. One-sided dense matrix factorizations on a multicore with multiple GPU accelerators. *Procedia Computer Science*, 9:37–46, 2012.

[116] I. Yamazaki, S. Tomov, and J. Dongarra. Non-GPU-resident symmetric indefinite factorization. *Concurrency and Computation: Practice and Experience*, 29(5), 2017. e4012.

[117] H. Yang, M. Li, K. Koizumi, and H. Kudo. Accelerating backprojections via CUDA architecture. In *Proceedings of the 9th International Meeting Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine (Fully 3D)*, pages 52–55, July 2007.

[118] W. Yu, Y. Gu, J. Li, S. Liu, and Y. Li. Single-pass PCA of large high-dimensional data. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3350–3356, 2017.

[119] X. Yuan and J. Yang. Sparse and low-rank matrix decomposition via alternating direction methods. *Pacific Journal of Optimization*, 9(1):167–180, 2013.

[120] H. Zhang. High performance parallel backprojection on multi-GPU. In *Proceedings of the 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, pages 2693–2696, May 2012.

[121] Q. Zhang and B. Li. Discriminative k-svd for dictionary learning in face recognition. In *Proceedings of the 23rd IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2691–2698, 2010.

[122] Z. Zheng and K. Mueller. Cache-aware GPU memory scheduling scheme for CT backprojection. In *Proceedings of the Nuclear Science Symposium on and Medical Imaging Conference (NSS/MIC)*, pages 2248–2251, Nov. 2010.

[123] T. Zinßer and B. Keck. Systematic performance optimization of cone-beam backprojection on the Kepler architecture. In *Proceedings of the 12th International Meeting Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine (Fully 3D)*, pages 225–228, June 2013.