UNIVERSITY OF CALIFORNIA
RIVERSIDE

Optimizing Tall-and-Skinny Matrix-Matrix Multiplication on GPUs

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Computer Science

by

Nan Xiong

December 2018

Acknowledgments

I am grateful to my academic advisor, Dr. Chen, without whose help, I would not have been here.

To my parents for all the support.

ABSTRACT OF THE THESIS

Optimizing Tall-and-Skinny Matrix-Matrix Multiplication on GPUs

by

Nan Xiong

Master of Science, Graduate Program in Computer Science
University of California, Riverside, December 2018
Dr. Zizhong Chen, Chairperson

Linear algebra operations have been widely used in big data analytics and scientific computations. Many works have been done on optimizing linear algebra operations on GPUs with regular-sized input. However, few works are focusing on how to fully utilize the underlying GPU resources when the input size is not regular. Current optimizations lack of considering fully utilizing the memory bandwidth and computing power, therefore they could only achieve sub-optimal performance. In this thesis, we propose a performant tall-and-skinny matrix-matrix multiplication algorithm on GPUs – `TSM2`. It focuses on optimizing linear algebra operation with none regular sized input. We implement the proposed algorithm and test on three different Nvidia GPU micro-architectures: Kepler, Maxwell, and Pascal. Experiments show that our `TSM2` speedups the computation by 1.1x - 3x, improves memory bandwidth utilization by 8% - 47.6%, and improves computing power utilization by 7% - 37.3% comparing to the current state-of-the-art works.

Contents

List of Figures

Chapter 1

Introduction

Matrix-matrix multiplication (GEMM) has been one of the most extensively used linear algebra operations in big data analytics and scientific computations. Due to many factors (e.g., algorithms, input data, etc.) the size or shape of input matrices of GEMM usually varies when it is used in different applications. For example, many modern highly scalable scientific simulations packages in the field of fluid dynamics, such as Finite Element Method (FEM) needs to compute many GEMM with small-sized input matrix. Artificial neural networks (ANN) involve using GEMM with small to medium input matrices. Matrix decompositions uses GEMM with large-sized input matrices. So, besides large-sized input, which has already been extensively optimized during the past decades, GEMM with small to medium sized input has also drawn a lot of attention to recent researchers. For example, [11] proposed MAGMA-Batched, which aims to batch small input matrices into larger ones to utilize the highly optimized implementations for large input size on GPUs. [14] proposed to speed up GEMM with small input using instruction level optimization on CPUs.

Although previous works have focused on optimizing GEMM with different matrix sizes, most of them only assume that the input matrix is *regular-shaped*. In another word, the size they mentioned in their works usually refers to both dimensions of the input matrix. So, for example, a small matrix means both its width and height are small and their magnitudes are close each other. However, not much work has been done to optimizing GEMM for *non-regular shaped* input. For example, there is one particular kind of non-regular shaped input in which the magnitude of both dimensions has significant difference i.e., tall-and-skinny. To the best of our knowledge, GEMM with tall-and-skinny input has not been fully studied and optimized for. Tall-and-skinny input has been used in many applications. For example, recent highly optimized K-means implementations [10, 1] use GEMM as their core computation and input size is usually tall-and-skinny since the number of centroids is usually far less than the number of input data points. Also, when GEMM is used for encoding checksums for many ABFT applications [9, 15, 8, 17, 24, 23], the input usually involves a tall-and-skinny checksum weight matrix.

Previous efforts made for optimizing GEMM with regular-shaped input may not work for the non-regular shaped input. For instance, [9] shows that calculating GEMM with tall-and-skinny input using vendor's highly optimized linear algebra library (e.g., cuBLAS [16]) is slower than disassembling the tall-and-skinny input matrix into several vectors and then applying matrix-vector multiplications instead. However, it can be easily seen that this workaround is not efficient, since elements in input matrices are accessed by GPU more times than necessary.

Although the performance can be optimized by grouping many tall-and-skinny input matrices into large ones similar to the approach proposed, there are cases where this grouping approach is not feasible. For example, tall-and-skinny input matrices may be generated one at a time from a producer process in user's workflow. Grouping several of them into a large matrix requires extended waiting time, which is not applicable for time-sensitive applications. On the other hand, the memory space may limit the total number matrices that can fit into the memory at the same time, if the input matrices are large (e.g., multiplication between regular-shaped large matrices and tall-and-skinny matrices).

In this work, we target on optimizing the computation of GEMM with tall-and-skinny input on the GPU platform since many applications that use GEMM are deployed on GPUs. So, our optimization could greatly benefit those applications. The insight of our work is that when input matrices size is regular-shaped (e.g., an $n \times n$ matrix multiplies an $n \times n$ matrix, each element in the input matrices loaded into GPU requires $O(n)$ for computation), the computation time is usually longer than memory access time (especially for large matrices). Then the GEMM operation is compute-bound. However, when input matrices size is tall-and-skinny (an $n \times n$ matrix multiplies an $n \times k$ matrix with k much smaller than n, each element is only use for $O(k)$ times on average for computation), depending on k and the ratio between executing GPU's peak computing power and peak memory throughput, the GEMM computation can be either compute-bound or memory-bound. As k gets smaller, it moves toward memory-bound; Otherwise, it moves toward compute-bound.

To optimize GEMM with tall-and-skinny input, it is critical to design a computation algorithm that considers both compute-bound and memory-bound cases.

The main contributions of this paper include:

- We study the limitation of current state-of-the-art GEMM implementation with tall-and-skinny input. With benchmarking, we find that the under-utilization of GPU resources is the main reason that causes low performance when the input is tall-and-skinny.

- We design a GEMM on GPU optimized for tall-and-skinny input with both double and single floating point precisions. We call our optimized version as `TSM2`. By leveraging the knowledge on the input size and hardware architecture characteristic, we redesign computation algorithm to ensure high memory bandwidth utilization in memory-bound cases and high computing performance for compute-bound cases. Experiments show that our `TSM2` can obtain 1.1x - 3x speedup comparing to state-of-the-art cuBLAS library on modern GPU micro-architectures. We also replace the original GEMM operations in K-means and ABFT applications with `TSM2` and achieve up to 1.89x and 1.90x overall speed up.

Chapter 2

Backgrounds

2.0.1 Definition of tall-and-skinny input

In this work, we restrict our scope to GEMM with tall-and-skinny input on GPUs. The tall-and-skinny input size means that, for the two input matrices, at least one matrix is tall-and-skinny (one dimension is significantly smaller than the other). For example, input matrix A with size 20480×20480 and matrix B with size 20480×2 are considered as tall-and-skinny input in our work. In this paper, we focus on optimizing GEMM with one regular large input matrix and one tall-and-skinny input matrix. We refer matrix A as the larger input matrix ($n \times n$) and matrix B ($n \times k$) as the tall-and-skinny input matrix in this paper. We choose this input size and shape because we believe it can expose most of the challenges in all kinds of tall-and-skinny input, so the design idea and optimization techniques introduced in this paper can be easily applied to other cases with slight modification. Also, we choose to let the larger matrix to be in squared-shape only for simplified representation.

2.0.2 cuBLAS

One of the most commonly used standard linear algebra libraries optimized for GPU is the cuBLAS library developed by Nvidia. The cuBLAS is the core computing library of many big data and scientific computing applications. For example, it is the GPU computing library for MAGMA heterogeneous linear algebra library [18, 19, 12], cuLA library [5], and cuDNN deep learning library [4]. With deep optimization by Nvidia, the cuBLAS library is able to provide state-of-the-art performance in many use cases. For example, with large regular-shaped input matrix, their GEMM implementation is able to achieve near peak performance of GPU [2].

However, we found that the GEMM subroutine is not fully optimized with certain input matrix sizes [9]. For example, with tall-and-skinny input, the GEMM operation in current best implementation (cuBLAS 9.0 running on NVIDIA Tesla K40c GPU) only uses less than 10% of the theoretical peak memory bandwidth on average with $k = 2$ (**Fig. 4.5**). When $k = 16$, the same GEMM operation only uses less than 20% of the theoretical peak memory bandwidth on average (**Fig. 4.8**). The resource utilization is even lower with larger input dimensions. By comparing the two input sizes, it can be seen for input with smaller k values, the computation utilizes higher memory bandwidth (close to memory bound). On the other hand, for input with larger k values, the computation utilizes higher computing power (close to compute bound). However, since we are not able to analyze the implementation of GEMM in none open-sourced cuBLAS library, it is hard to tell the exact characteristic of their computation.

Chapter 3

Optimization design

3.0.1 Insight on tall-and-skinny input size

For regular-shaped GEMM ($n \times n$ matrix multiplies $n \times n$ matrix), the input matrices size is $O(n^2)$, while the computing time complexity is $O(n^3)$, so each element in input matrices is used $O(n)$ times within the entire computation process. Since loading data from GPU off-chip DRAM (i.e., global memory) to GPU is expensive and to avoid extensive data load operations, one common optimization for this kind of problem is minimizing the number of times each element needs to be loaded into the GPU by using fast on-chip memory (e.g., cache, registers) to enable data reuse. As the number of loads reduces, optimized GEMM tends to be compute bound. For example, current GEMM implementation in cuBLAS library can reach near bare-metal performance on GPUs [2].

However, unlike regular-shaped GEMM, when the input size is tall-and-skinny, the input matrices size is still $O(n^2)$, however, the computing time complexity is $O(n^2k)$. So, each element in the input matrices is used k times on average:

$$\frac{(n \times n) \times k\ times + (n \times k) \times n\ times}{n \times n + n \times k} \approx k\ times$$

Depending on the size of k and target GPU peak computing power and memory throughput ratio, the `TSM2` can be either compute bound or memory bound. When k gets smaller, the computation tends to be memory bound. Otherwise, the problem tends to be compute bound. In either case, the problem is always near the boundary between memory bound and compute bound, so it is critical to design an algorithm that is optimized for both two cases.

3.0.2 Algorithm design

As the core of our optimization, algorithm design plays an important role. First, we need to consider how to fit the workload of our `TSM2` into the programming model of CUDA (i.e., thread hierarchy). Although the workload can be easily decomposed into many independent smaller workloads, careful consideration on workload distribution is still necessary, since any unnecessary performance penalty can drastically cause GPU resource underutilization. Several factors are considered in our design:

1. Total number of global memory accesses;
2. Efficiency on global memory throughput;
3. Utilization on global memory throughput;
4. Parallelism of overall workload;

5. On-chip memory utilization;

6. Streaming Multiprocessor (SM) utilization;

7. Optimization for both compute bound and memory bound cases.

To achieve good performance, there must exist enough number of active threads in each SM of GPU to ensure proper instruction and memory access latency hiding. So, in our algorithm we divide the workload by assigning n rows of matrix A to n different threads. Each vector-matrix multiplication is assigned to one thread (i.e., $(A[i,:] \times B)$). The benefit is three-fold: First, this ensures high parallelism and high SM occupancy; Second, since the number of elements of matrix A is much higher than matrix B, this kind of distribution ensures minimum number of memory accesses in favor of matrix A; Third, it also enables high memory access efficiency and throughput, since all memory accesses to matrix A are naturally coalesced (assuming input matrices are stored in column-major by convention).

As for the vector-matrix multiplication assigned to each thread, to further reduce the number of memory accesses to matrix A, we use outer-product style computation instead of the regular inner-product style computation. As shown in **Alg. 1**, if we use inner-product, each element of matrix A is repeatedly referenced k times. On the other hand, if we use outer-product as shown in **Alg. 2**, each element of matrix A is referenced only once. (Please note, as we will discuss in later sections, when k is larger than a certain threshold, elements in matrix A still need to be referenced more than once due to the limited resources available for each thread, but it is still far lower than using inner-product). For large matrix A, the benefit is significant, since it greatly reduces the total number of global memory accesses during the entire GEMM computation. Also, the outer-product style does not bring any extra memory accesses to matrix B compared to inner-product style. The only cost for outer-product is extra registers holding k intermediate results. However, with proper tuning, they only bring little performance impact compared with extra memory accesses.

Algorithm 1 Workload of each thread with inner product

```
Require: input matrix A ($n \times n$) and B ($n \times k$)
Require: output matrix C ($n \times k$)
1: for $i = 1$ to $k$ do
2:     for $j = 1$ to $n$ do
3:         $C[thread\_id, i] += A[thread\_id, j] \times B[j, i]$
4:     end for
5: end for
```

Algorithm 2 Workload of each thread with outer product

```
Require: input matrix A ($n \times n$) and B ($n \times k$)
Require: output matrix C ($n \times k$)
1: $Reg1 \leftarrow C[thread\_id, 1]$
2: $Reg2 \leftarrow C[thread\_id, 2]$
3: ...
4: $Regk \leftarrow C[thread\_id, k]$
5: for $i = 1$ to $n$ do
6:     $tmp \leftarrow A[thread\_id, i]$
7:     $Reg1 += tmp \times B[i, 1]$
8:     $Reg2 += tmp \times B[i, 2]$
9:
10:    $Regk += tmp \times B[i, k]$
```

```
11: end for
12: $C[thread\_id, 1] \leftarrow Reg1$
13: $C[thread\_id, 2] \leftarrow Reg2$
14: ...
15: $C[thread\_id, k] \leftarrow Regk$
```

3.0.3 Efficient off-chip memory access

One key factor of optimizing memory intensive applications is ensuring high off-chip memory access efficiency. Depending on the GPU model type or runtime configurations, global memory (off-chip) accesses of threads within the same warp can to coalesced into 128 byte- or 32 byte-transactions [3] if their access addresses fall into the same 128 byte- or 32 byte-segments in global memory, which enables efficient use of memory bandwidth. Otherwise, the GPU still loads memory in 128 byte- or 32 byte-transactions, but it may contain unrequested data that are stored in neighbor addresses, which causes inefficient memory accesses.

Since each thread reads one row of matrix A and the matrix is stored in column-major by convention, memory accesses are naturally coalesced when threads within the same warp access elements on different rows but on the same column. So, 100% memory access efficiency is achieved on matrix A. However, for matrix B, all threads access the same element at the same time, which results a single memory transaction containing one requested element and several unrequested neighbor elements. So, only $\frac{8\ bytes}{128\ bytes} = 6.25\%$ or $\frac{8\ bytes}{32\ bytes} = 25\%$ memory access efficiency is achieved for accessing 64-bit double floating point elements. Although the total number of elements in matrix B is small, given that each element needs to be accessed n times, this inefficient access pattern can still greatly impact the overall performance.

To improve the efficiency of memory accesses to matrix B, we utilize shared memory in GPU. Since it is located on-chip, shared memory gives us the speed of L1 cache and it is fully programmable. Threads within one thread block can also use shared memory to share data between them. So, one key advantage of shared memory is that it eliminates the need for the consistency between patterns of data loading and data using pattern, which enables us to load global memory in the most efficient way and keep the way we use data as before.

By using shared memory for accessing matrix B, we can reduce the total number of memory accesses and enable coalesced memory access. As shown in **Alg. 3**, for each iteration, instead of letting threads request elements they need individually by themselves inefficiently, we now let a block of threads work together to fetch a tile of matrix B into the shared memory in a coalesce-compatible way (**line 13-15**). Then during computation, each thread references elements in matrix B through the shared memory instead of loading each one of them individually from global memory. This reduces the total number of accesses to matrix B from global memory (from n to n/t_1 per element). Also, threads in a same thread block (also warp) fetch elements of matrix B column by column, which enables coalesced memory access. This greatly improves memory access efficiency of matrix B to 100%.

In **Alg. 3**, we also introduce three parameters: t_1, t_2, and t_3. These parameters are used for adjusting the performance and will be discussed in later sections.

Algorithm 3 TSM2 with shared memory

```
Require: input matrix A ($n \times n$) and B ($n \times k$)
```

```
Require: output matrix C ($n \times k$)
 1: $t_1 \leftarrow tile\_size\_B$, $t_2 \leftarrow tile\_size\_C$, $t_3 \leftarrow tile\_size\_A$
 2: Register: $A_1, A_2, ..., A_{t_3}$
 3: Register: $C_1, C_2, ..., C_{t_2}$
 4: Shared Memory: currB with size $t_1 \times t_2$
 5: Threads per thread block $\leftarrow t_1$
 6: Total thread blocks $\leftarrow n/t_1$
 7: for $p = 1$ to $k$ with step size $= t_2$ do
 8:     $C_1 \leftarrow C[thread\_id, p]$
 9:     $C_2 \leftarrow C[thread\_id, p+1]$      ...
10:     $C_{t_2} \leftarrow C[thread\_id, p+t_2-1]$
11:     for $j = 0$ to $n$ with step size $= t_1$ do
            /* Load a tile of B into shared memory */ThreadsSynchronization()
            $currB[thread\_id, 1] \leftarrow B[j+thread\_id, p]$ $currB[thread\_id, 2] \leftarrow B[j+thread\_id, p+1]$      ... $currB[thread\_id, t_2] \leftarrow B[j+thread\_id, p+t_2-1]$ ThreadsSynchronization()
            for $l = j$ to $j+t_1$ with step size $= t_3$ do /* Load a tile of A into registers */
18:             $A_1 \leftarrow A[thread\_id, l]$
19:             $A_2 \leftarrow A[thread\_id, l+2]$          ...
20:             $A_{t_3} \leftarrow A[thread\_id, l+t_3-1]$
21:             $C_1 += A_{[1...t_3]} \times currB[[l...l+t_3], 1]$
22:             $C_2 += A_{[1...t_3]} \times currB[[l...l+t_3], 2]$          ...
```

```
23:         $C_{t_2} += A_{[1...t_3]} \times currB[[l...l+t_3], t_2]$
24:       end for
25:     end for
26:     $C[thread\_id, p] \leftarrow C_1$
27:     $C[thread\_id, p+1] \leftarrow C_2$      ...
28:     $C[thread\_id, p+t_2] \leftarrow C_{t_2}$
29: end for
```

3.0.4 Optimize use of shared memory

Although fast, elements in shared memory still need to be loaded into registers before using [6]. Its accessing speed can affect the overall performance. Shared memory is divided into several same-sized memory banks for fast parallel accesses. Different threads accessing different memory banks can be done simultaneously. So, total b memory banks can speedup overall shared memory throughput by up to b times compared to the throughput of one single memory bank. However, if x threads in the same warp access different data from the same memory bank, x-way bank conflict would occur and each request is processed sequentially, which dramatically reduces the accessing throughput to $1/x$ of the peak throughput.

In our algorithm, threads in the same thread block load data from global memory into shared memory column by column to enable fast coalesced global memory access. Then threads access data from shared memory row by row during computation. How we store elements in shared memory will affect how these elements are accessed from memory banks, which affects the throughput of shared memory. We have two ways of storing a tile of matrix

B in shared memory: column-major storage and row-major storage. To choose between the two ways, we need to analyze and compare which way brings the least overall bank conflict. We assume the size of one tile of matrix B is $t_1 \times t_2$ and t_1 is the multiply of total number of memory banks b for simplicity.

For column-major storage, elements (32-bit words or 64-bit words) in the same column of one tile of matrix B are stored in successive memory banks. So, for shared memory with b memory banks, every t_1 elements of one column are stored in b different successive memory banks with each bank stores at most $\frac{t_1}{b}$ elements and is accessed by at most $\frac{warp\ size}{b}$ threads at the same time, which may potentially cause bank conflict if $\frac{warp\ size}{b}$ is greater than one.

For row-major storage, elements in the same row of matrix B are stored in successive memory banks. So, elements of the same column are stored in $\frac{b}{t_2}$ different banks with each bank stores $\frac{t_1 \times t_2}{b}$ elements from one column. Since each bank has t_2 times more elements from one column, totally each bank has at most t_2 times more thread accessing at the same time: $\frac{warp\ size}{b} \times t_2$, which may also potentially cause bank conflict.

On modern Nvidia GPUs, the *warp size* is fixed to 32 and total number of banks is also 32 [3], so column-major storage does not cause bank conflict, since each bank can only have up to one thread accessing. Row-major storage can cause up to t_2-way bank conflict, which decreases overall shared memory throughput to $\frac{1}{t_2}$ of the peak throughput. As shown in **Fig. 3.1**, we load a 64×2 matrix tile into shared memory using column-major storage (left) and row-major storage (right). When using column-major storage, threads in one warp all access different banks, so no bank conflict occurs. On the other hand, when using row-major storage, 32 elements are stored in 16 banks causing 2-way bank conflict.

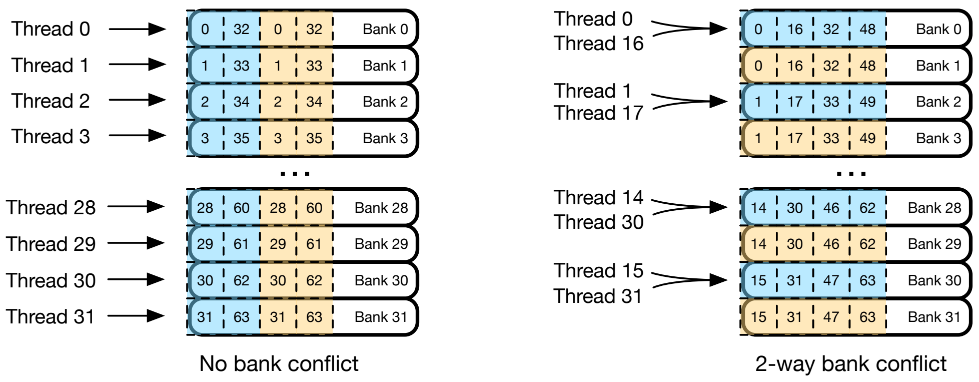


Figure 3.1: Comparing column-major (left) with row-major (right) storage for storing a 64×2 tile of matrix B in shared memory. Blue and yellow squares represent elements in the first and second column. When one warp of 32 threads accessing 32 elements in one column (e.g. element 0 to 31 of the first column), the column-major storage brings no bank conflict and row-major storage brings 2-way bank conflict, which reduces throughput by half.

When accessing elements in shared memory for computation, threads in a warp (thread block) all access the same element at a time in our algorithm. Although multiple threads are accessing one bank, they are accessing the same element, so one broadcast is initiated, which does not cause bank conflict. It is the same for both storage styles. So, we choose column-major storage as it brings no bank conflict and potentially brings the highest shared memory throughput.

3.0.5 Overlapping Computation and Memory Access Latency

During execution, for each instruction issuing moment, each warp scheduler picks an eligible warp and send it to the corresponding component for execution. A warp becomes eligible only if all operands of its next instruction are ready. However, if a warp is loading data from global memory, it would take several hundred cycles before it can be ready for execution. To hide this long latency, we can either increase the number of threads reside in each SM to ensure there always exist eligible warps [20] or put independent instructions in between data loading and data consuming operations, so that warps are also eligible for execution during memory loading time. The first approach requires us to adjust the on-chip resource usage of each thread block. We will leave that discussion in the next section. In this section, we aim to add independent instructions in between data loading and data consuming operations.

A shown in **Alg. 3**, **line 13-15 and 18-20** load data from global memory and **line 21-23** consume data once data is loaded. However, due to data dependency, there is no independent instruction in between, so once each warp issues global memory access requests, it must wait for the requested elements to be ready before it can proceed to computation.

So, to add independent instructions, we use data prefetching to mix the data loading and consuming between neighbor iterations. Specifically, instead of letting each iteration loads data that is going to be used for current iteration, we let the data needed for current iteration to be loaded by the previous iteration, so that its calculation will not be blocked by data loading (since the data are ready). When doing calculation, it also loads data that is going to be used for the next iteration. By overlapping data loading and computation, we can significantly improve memory bandwidth and SM utilization. We apply data prefetching to both matrix A and B.

Algorithm 4 `TSM2` with shared memory and data prefetching

```
Require: input matrix A ($n \times n$) and B ($n \times k$)
Require: output matrix C ($n \times k$)
```

```
1: $t_1 \leftarrow tile\_size\_B$, $t_2 \leftarrow tile\_size\_C$, $t_3 \leftarrow tile\_size\_A$
2: Register: $currA_1$, $currA_2$,...,$currA_{t_3}$
3: Register: $nextA_1$, $nextA_2$,...,$nextA_{t_3}$
4: Register: $nextB_1$, $nextB_2$,...,$nextB_{t_2}$
5: Register: $C_1$, $C_2$,...,$C_{t_2}$
6: Shared Memory: $currB$ with size $t_1 \times t_2$
7: Threads per thread block $\leftarrow t_1$
8: Total thread blocks $\leftarrow n/t_1$
9: for $p = 1$ to $k$ with step size $= t_2$ do
10:     $C_1 \leftarrow C[thread, p]$
11:     $C_2 \leftarrow C[thread, p+1]$     ...
12:     $C_{t_2} \leftarrow C[thread, p+t_2-1]$
13:     $currB[thread\_id, 1] \leftarrow B[thread\_id, p]$
14:     $currB[thread\_id, 2] \leftarrow B[thread\_id, p+1]$     ...
15:     $currB[thread\_id, t_2] \leftarrow B[thread\_id, p+t_2-1]$
16:     $currA_1 \leftarrow A[thread\_id, 1]$
17:     $currA_2 \leftarrow A[thread\_id, 2]$     ...
18:     $currA_{t_3} \leftarrow A[thread\_id, t_3]$
19:     for $j = 0$ to $n$ with step size $= t_1$ do
20:         ThreadsSynchronization() /* prefetch the next tile of B into registers */
21:         if $j + t_1 < n$ then
22:             $nextB_1 \leftarrow B[j + t_1 + thread\_id, p]$
```

```
        $nextB_2 \leftarrow B[j + t_1 + thread\_id, p + 1]$      ...
        $nextB_{t_2} \leftarrow B[j + t_1 + thread\_id, p + t_2 - 1]$
    **end if**
    **for** $l = j$ to $j + t_1$ with step size $= t_3$ **do**
          **/* prefetch the next tile of A into registers */****if** $l + t_3 < n$ **then**
          $nextA_1 \leftarrow A[thread\_id, l + t_3]$
          $nextA_2 \leftarrow A[thread\_id, l + t_3 + 1]$      ...
          $nextA_{t_3} \leftarrow A[thread\_id, l + t_3 + t_3 - 1]$
        **end if**
        $C_1 += currA_{[1...t_3]} \times currB[[l...l + t_3], 1]$
        $C_2 += currA_{[1...t_3]} \times currB[[l...l + t_3], 2]$      ...
        $C_{t_2} += currA_{[1...t_3]} \times currB[[l...l + t_3], t_2]$
        $currA_1 \leftarrow nextA_1$
        $currA_2 \leftarrow nextA_2$      ...
        $currA_{t_3} \leftarrow nextA_{t_3}$
    **end for**
    ThreadsSynchronization()
    $currB[thread\_id, 1] \leftarrow nextB_1$
    $currB[thread\_id, 2] \leftarrow nextB_2$      ...
    $currB[thread\_id, t_2] \leftarrow nextB_{t_2}$
**end for**
$C[thread, p] \leftarrow C_1$
```

```
45:     $C[thread, p+1] \leftarrow C_2$     ...
46:     $C[thread, p+t_2] \leftarrow C_{t_2}$
47: end for
```

As shown in **Alg. 4**, we design our `TSM2` with data prefetching. In **line 4 and 5**, we allocate two sets of t_3 registers for storing current tile of elements of matrix A and next tile of element of matrix A for prefetching. In **line 6 and 8**, we allocate t_2 registers for data prefetching of elements in matrix B, and allocate $t_1 \times t_2$ for storing currently loaded tile of matrix B. Note that we cannot store current tile of matrix B in registers, because elements in matrix B need to be shared between threads during computation.

Before the core computation iteration (**line 20-40**), we pre-load current tile of matrix A and B into registers and shared memory (**line 13-19**), so that computation can start immediately as soon as we enter the computation loop without being blocked by any data dependency. The main computation resides in **line 28-30**. To overlap computation with memory accesses, we initiate loading for the next tile before the computation (**line 21-23** for matrix B and **line 25-27** for matrix A). We use two loops for loading matrix A and B, because we want to have the flexibility to adjust loading pace (tile size) differently for the two matrices. We will discuss this in the next subsection. **Fig. 3.2 and 3.3** show one iteration of our optimized `TSM2` with data prefetching. `LD C` and `ST C` represent loading initial values from matrix C and storing final results back to matrix C. Each iteration we show three sub-iterations for loading matrix B. As we can see, we load pre-load the next tile of matrix B in concurrent with computation to improve memory bandwidth utilization. A threads synchronization is inserted in the end of each iteration. For the inner most iteration, we do the actual computation and pre-load elements from matrix A each time. Please note that the length of each rectangle does not accurately represent the exact execution time length and the ratio between number of `LD nextA` and `LD nextB` is not necessarily two in actual computation. Also, we show one thread block with four threads only for illustration proposes. As we will discuss in the next subsection that different parameter values can affect the length of each part and the ratio between number of `LD nextA` and `LD nextB`. Especially on the execution time of `LD nextA` and `Compute`, which will affect the characteristic of computation (i.e. memory bound or compute bound). Also, for simplicity, we ignore the part that moves data between next tile storage to current tile storage of each iteration in this figure.

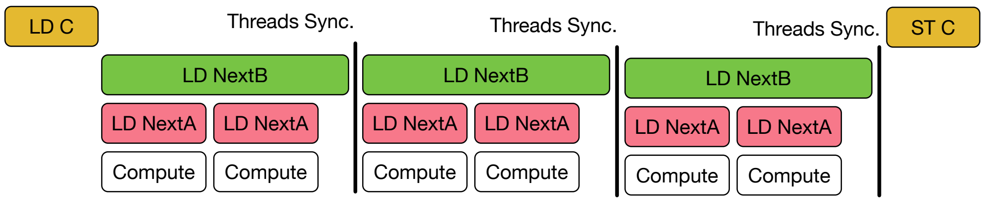


Figure 3.2: Example workload of one iteration of our optimized `TSM2` with data prefetching

3.0.6 Parameters Definition

In **Alg. 3** and **Alg. 4**, we introduced three adjustable parameters: t_1, t_2, and t_3. In this section, we first discuss how each parameter controls the computation of our `TSM2`. Then, we introduce our performance model that estimates how certain performance metrics change with these parameters. Finally, we explain our strategies of choosing values for these parameters in order to achieve high GPU resources utilization and optimized overall performance. Please note that the following discussions are all based on **Alg. 4**.

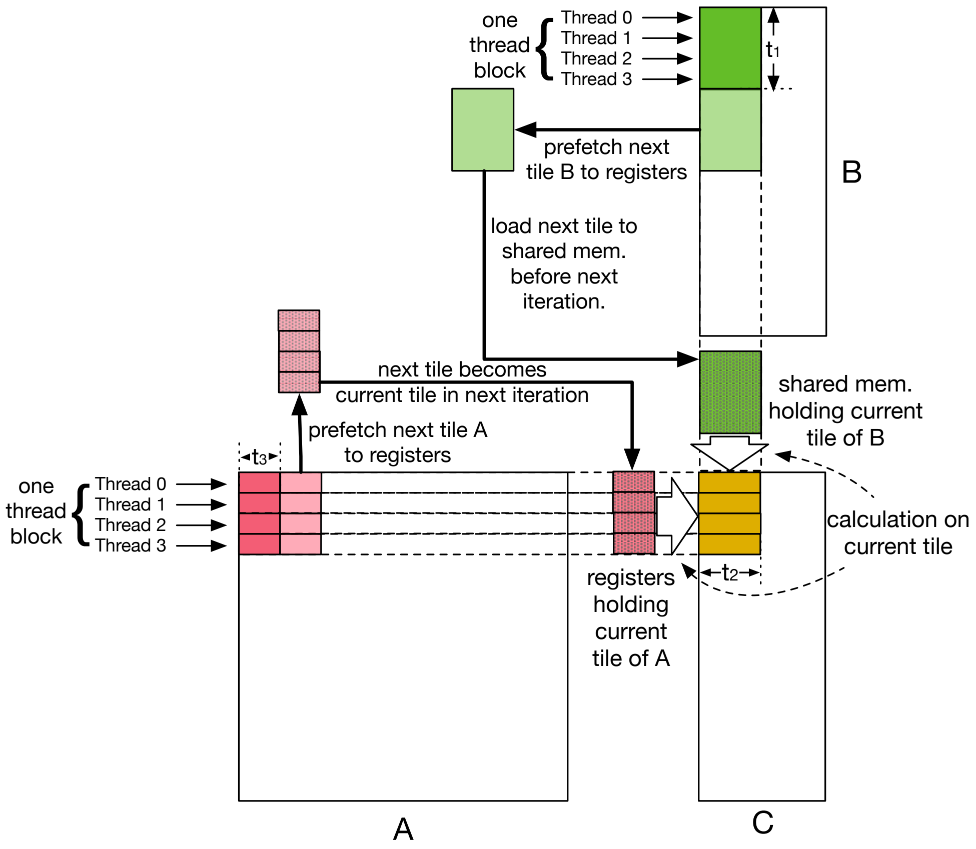

one thread block
Thread 0
Thread 1
Thread 2
Thread 3
t1
prefetch next tile B to registers
B
load next tile to shared mem. before next iteration.
next tile becomes current tile in next iteration
shared mem. holding current tile of B
prefetch next tile A to registers
t3
one thread block
Thread 0
Thread 1
Thread 2
Thread 3
calculation on current tile
registers holding current tile of A
t2
A
C

Figure 3.3: Matrix view of tall-and-skinny matrix matrix multiplication with data prefetching

Behaviors of Parameters

First, we list the behaviors of each parameter as follows:

- t_1 specifies the number of rows of one tile of matrix B. To maximize use of available active threads and to avoid any inefficient thread execution caused by warp divergence, we let all threads in each thread block participating in fetching elements of matrix B. For fast coalesced global memory access, we let each thread fetch one row, so t_1 is also the total number of threads in each thread block. Also, since we let total n threads working on the computation, the total number of thread blocks can be calculated as: n/t_1.

- t_2 specifies the number of elements in matrix C that each thread is working on at a time. It is used to divide the overall workload into several smaller workloads that are processed iteratively by each thread. Smaller workload makes each thread's SM resource usage smaller, which allows us to keep higher SM occupancy. However, dividing the workload means we need to load matrix A repeatedly for each small workload. So, there is a trade-off. t_2 also affects the ratio between total number of memory fetches and computation operations in core part of our algorithm, which allows us to adjust the computation to be compute or memory bound (will be discussed later in detail).

- t_3 specifies the number of elements in matrix A that each thread fetches at a time. Since elements fetching are independent to each other, they can be initiated without blocking each other, so t_3 can be used to adjust the memory loading concurrency.

Performance Metrics Estimation

In this section, we introduce our parameters based performance model that is used to estimate three important performance metrics: SM occupancy, memory bandwidth utilization and computing power utilization. These estimations will be used for optimizing the overall performance.

- **Max SM occupancy estimation**

  Now, with these parameters, we can calculate the max occupancy of each SM, which is defined as max number of active threads per SM. (Some works also use max number of warps, which is similar to ours. We found that using max thread is more consistent across our performance models. We also choose thread block size to be the dividend of this value to ensure expected number of threads are active.) This occupancy is mainly bound by the maximum hardware allowable number of threads (HW_MAX) and on-chip memory utilization per thread. We first calculate the total number of registers utilized per thread. Since register utilization can potentially be optimized by the `nvcc` compiler, we use maximum number of registers to estimate this value. First of all, there is a relatively fix amount of registers uses for CUDA initial setup, and we represent this amount as C. We get its amount through off-line profiling. Then, we need two sets of t_2 registers for storing elements of matrix B for both next tile fetching and current tile calculation. Please note that although the current tile of matrix B is stored in shared memory, it still needs to be transferred to registers for calculation. Next, we need t_2 registers for keeping intermediate results of matrix C. Finally, we need two sets of t_3 registers for storing elements of matrix A for both next tile fetching and current tile calculation. So, the total number of registers is:

$$R_{thread} = (t_2 \times 3 + t_3 \times 2) \times \frac{bytes\_per\_element}{bytes\_per\_register} + C$$

As for shared memory, through shared memory is allocated per thread block, we calculate the average amount of shared memory that each thread uses for consistent calculation here. Since the size of allocated shared memory per thread block is $t_1 \times t_2$, and as we will discuss earlier that we set $t_1 = threads\_per\_threadblock$, the amount of shared memory allocated for each thread on average is:

$$S_{thread} = t_2 \times bytes\_per\_element$$

So, the max SM occupancy can be calculated as:

$$MaxOccup_{SM} = min(HW\_MAX, \frac{R_{SM}}{R_{thread}}, \frac{S_{SM}}{S_{thread}})$$

In this above calculation, R_{SM} and S_{SM} stand for the max available registers and shared memory per SM.

- **Max memory bandwidth utilization estimation**

  Next, we estimate the max memory bandwidth utilization of our algorithm when the computation is memory bound. In this case, loading elements of matrix A dominates the computation instead of floating point calculations in our algorithm. So, we can estimate max memory bandwidth utilization using the maximum number of concurrent global memory accesses per SM. It can be calculated as:

$$Concurrent_{mem} \approx MaxOccup_{SM} \times t_3$$

Please note that, we only consider the memory accesses to matrix A here for simplicity. Since the majority memory accesses are for matrix A, this only brings minor inaccuracy. Then, similar to [20, 22] we calculate the least number of concurrent memory accesses per SM needed to achieve max memory bandwidth utilization using

Little's Law:

$$Throughput_{max\_mem} = \frac{Peak\ Band.}{\#\_of\_SM \times core\_clock}$$

$$Concurrent_{max\_mem} = latency_{mem} \times Throughput_{max\_mem}$$

The $latency_{mem}$ is the average global memory access latency, which is considered as a constant in our model and is obtained through offline profiling. The estimated memory bandwidth utilization is:

$$Util_{mem} = \frac{Concurrent_{mem}}{Concurrent_{max\_mem}}$$

- **Max computing power utilization estimation**

  Next, we estimate the max computing power utilization of our algorithm when the computation is compute bound. In this case, floating point calculation dominates the computation instead of memory accesses in our algorithm. So, we can estimate max computing power utilization using the maximum number of concurrent floating point operations per SM. It can be calculated as:

$$Concurrent_{comp} = MaxOccup_{SM} \times t_3 \times t_2$$

  Then, also similar to [20] we calculate the least number of concurrent floating point operations per SM needed to achieve max computing power utilization using Little's Law:

$$Throughput_{max\_comp} = \frac{Peak\ Perf.}{\#\_of\_SM \times core\_clock}$$

$$Concurrent_{max\_comp} = latency_{comp} \times Throughput_{max\_comp}$$

The $latency_{comp}$ is the average latency of floating point operations in our calculations, which is considered as a constant in our model and is obtained through offline profiling. So, the estimated computing power utilization is:

$$Util_{mem} = \frac{Concurrent_{comp}}{Concurrent_{max\_comp}}$$

- **Determine computing or memory bound**

  Given parameters and GPU specification, we can determine whether the current computation is memory or compute bound. This is mainly determined by the inner most loop (**line 24 - 34**) of **Alg. 4**. The memory loading instructions (**line 25-27**) are overlapping with computation (**line 28-30**). Since **line 31-33** depends on memory loading results, it serves as an implicit synchronization point for memory load and computation. The time takes for the two parts will determine whether the current computation is compute bound or memory bound. So, we first estimate the time takes for computation and memory access as follows:

$$time_{comp} = \frac{t_3 \times t_2}{Peak\ Perf. \times \#\_of\_SM \times Occupancy_{SM}}$$

$$time_{mem} = \frac{t_3 \times bytes\_per\_elem.}{Peak\ Band. \times \#\_of\_SM \times Occupancy_{SM}}$$

Then, by comparing the two time costs, we can determine whether the current computation is compute bound or memory bound.

$$r = \frac{time_{comp}}{time_{mem}} = \frac{t_2}{bytes\_per\_elem.} \times \frac{Peak\ Band.}{Peak\ Perf.}$$

As we can see, when r is greater than one, the computation is compute bound. On the other hand, when r is less than one, the computation is memory bound. Also, since we divide the original workload into several smaller workloads using t_2, this ratio is determined by t_2. By adjusting t_2, the actual computation can be shifted between compute and memory bound. The boundary between the two cases can be calculated by setting the ratio $r = 1$, so we get a threshold for t_2:

$$t_2^{threshold} = \frac{PeakPerf.}{PeakBand.} \times bytes\_per\_elem.$$

Similarly, we can also estimate computation characteristic of the original problem, in which the workload is not divided into smaller workloads. In this case, t_2 is always fixed to k. So, by comparing k with $t_2^{threshold}$ we can estimate the computation characteristic. If k is greater than $t_2^{threshold}$, then the original problem is compute bound. Otherwise, it is memory bound.

It can be easily seen, depending on the value of t_2 and k, the computation characteristic of the current problem and original problem can be different, which can affect the overall performance. We discuss this in later part of this section.

Choosing parameters

When choosing parameters, the first thing we should determine is whether we should optimize for computation or memory bandwidth. This is determined by whether the given TSM2 computation on the given GPU should be compute or memory bound. In the last section, we proposed to estimate this characteristic by comparing k and $t_2^{threshold}$, so that we can adjust parameters to optimize the computation in the right direction.

In the case where original problem is memory bound ($k \leq t_2^{threshold}$), we need to keep the actual computation to be memory bound also (let $1 \leq t_2 \leq k$) and optimize for memory bandwidth utilization. On the other hand, if the original problem is compute bound ($k > t_2^{threshold}$), we first try to keep the actual computation to be compute bound too (let $t_2^{threshold} \leq t_2 \leq k$) and optimize of computing power utilization. However, in the case where $t_2^{threshold}$ is too high on the given GPU, we also try to optimize it for memory bound (let $1 \leq t_2 \leq t_2^{threshold}$) and output the result parameters that deliver better performance.

Algorithm 5 Parameter Optimization for TSM2

```
1: if $k \leq t_2^{threshold}$ then
2:     $Total\_memory \approx n \times n \times \frac{k}{t_2} \times bytes\_per\_elem.$
3:     $Bandwidth = PeakBand. \times Util_{mem}$
4:     Use Gradient Descent to Optimize ($t_2$ and $t_3$): $Time = \frac{Total\_memory}{Bandwidth}$ with $1 \leq t_2 \leq k$
       and $1 \leq t_3$
5:     Output: $t_2$ and $t_3$
6: else
7:     $Total\_flops = n \times n \times k \times 2$
8:     $Compute\_power = PeakPerf. \times Util_{comp}$
```

```
9:      Use Gradient Descent to Optimize ($t_2$ and $t_3$): $Time_1 = \frac{Total\_flops}{Compute\_power}$ with $t_2^{threshold} \leq$
        $t_2 \leq k$ and $1 \leq t_3$
10:     $t_{2(time_1)} \leftarrow t_2$
11:     $t_{3(time_1)} \leftarrow t_3$
12:     $Total\_memory \approx n \times n \times \frac{k}{t_2} \times bytes\_per\_elem.$
13:     $Bandwidth = PeakBand. \times Util_{mem}$
14:     Use Gradient Descent to Optimize ($t_2$ and $t_3$) in $Time_2 = \frac{Total\_memory}{Bandwidth}$ with $1 \leq t_2 \leq$
        $t_2^{threshold}$ and $1 \leq t_3$
15:     $t_{2(time_2)} \leftarrow t_2$
16:     $t_{3(time_2)} \leftarrow t_3$
17:     **if** $Time_1 < Time_2$ **then**
18:         **Output:** $t_{2(time_1)}$ **and** $t_{3(time_1)}$
19:     **else**
20:         **Output:** $t_{2(time_2)}$ **and** $t_{3(time_2)}$
21:     **end if**
22: **end if**
```

Alg. 5 shows the parameter optimization procedure for t_2 and t_3. We first determine the computation characteristic in **line 1**. If it is memory bound, we optimize for total time cost to access needed elements from global memory (**line 4**). Otherwise, we optimize for either total computation time (**line 9**) or memory access time (**line 14**). Please note that we only count the total amount of memory accesses to matrix A for simplicity, since total accesses to matrix B is much less than matrix A, so this simplification only brings minor inaccuracy. Also, considering the total accesses to matrix B would bring one additional parameter (t_1), which can be hard to optimize since t_1 is also related to threads organization that is hard for modeling-based estimation. The memory bandwidth utilization term ($Util_{mem}$) and computing power utilization term ($Util_{comp}$) is calculated using the equation mentioned before. Since we have two parameters (t_2 and t_3) in our optimization target, we use Gradient Descent (GD) to do the optimization. In GD, based on our experience, we set initial value of both t_2 and t_3 to be 1, and step size to be 0.1. The stop threshold is set to be 1e-4, since we do not need very accurate precision. The final t_2 and t_3 are rounded to the nearest integers.

To optimize t_1, we found it only controls the number of threads in each thread block. Since the total number of threads is fixed to n, t_1 only determines how these threads are organized into thread blocks. There is trade-off: if t_1 is large, the total number of accesses to elements of matrix B is reduced, however, large thread block means large number of threads need to participate in the same synchronization, which may have impact on performance. On the other hand, if t_1 is small, the total number of accesses to elements of matrix B higher, but the smaller thread block makes scheduling more flexible and efficient. It is hard to determine the optimum value of t_1 theoretically, so we use offline profiling to choose the best value. Specifically, once t_2 and t_3 are determined, we benchmark different t_1 values that can divide $MaxOccup_{SM}$ as mentioned earlier and choose the t_1 that deliver the best performance. Although t_1 seems to have direct effect on shared memory allocation (or max SM occupancy), it actually has limited impact on it, since we fix the amount of shared memory per thread ($S_{thread} = t_2 \times bytes\_per\_element$).

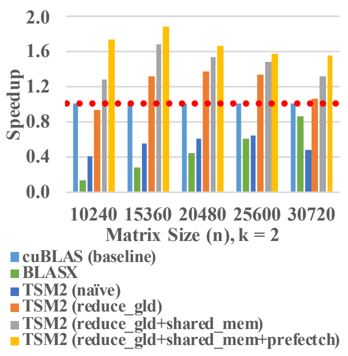
Chapter 4

Experimental evaluation

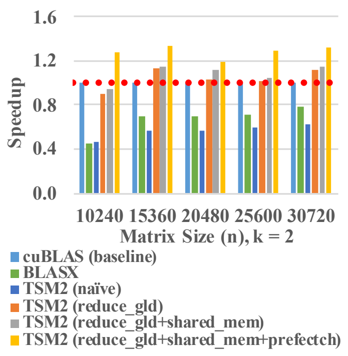
4.0.1 Experiments setup

We evaluate our optimized `TSM2` on our heterogeneous testbed cluster: Darwin. We run each test on a single GPU node with single GPU card. We conduct our tests on three different commonly used modern Nvidia GPUs with three different micro-architectures: Kepler, Maxwell, and Pascal. For Kepler GPU, we use Tesla K40c, which has 1430 GFLOPS peak double floating point performance and 288 GB/s memory bandwidth. For Maxwell GPU, we use Tesla M40, which has 213 GFLOPS peak double floating point performance and 288 GB/s memory bandwidth. For Pascal GPU, we use Tesla P100, which has 4600 GFLOPS peak double floating point performance and 720 GB/s memory bandwidth.

We implemented our `TSM2` using CUDA C for both single and double floating point input. We disabled compiler auto unrolling for better control on register allocation. For

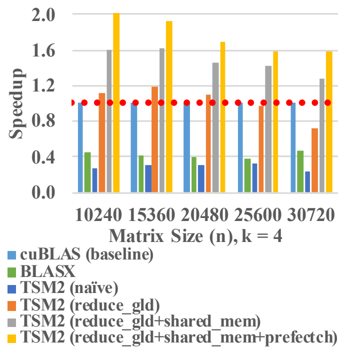


(a) Single precision

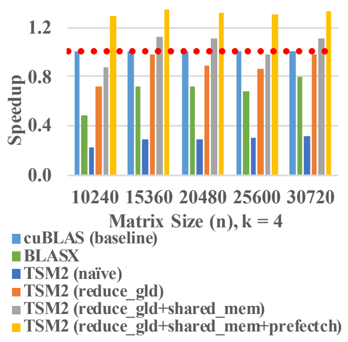


(b) Double precision

Figure 4.1: Speedup comparison for k = 2 on K40c.



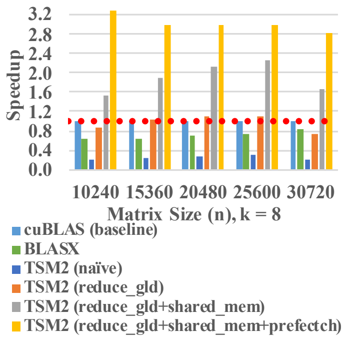
(a) Single precision



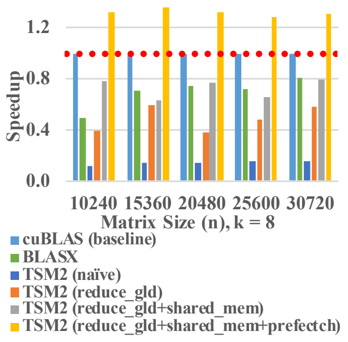
(b) Double precision

Figure 4.2: Speedup comparison for k = 4 on K40c.

comparison, we compare our `TSM2` with GEMM in the current latest cuBLAS 9.0 library and latest BLASX library [21]. Also, we try to compare our work with KBLAS [7], however since its GEMM kernel is based on cuBLAS, its performance is identical to cuBLAS, so we omitted its results. Each test is repeated multiple times to reduce noise and timed using CUDA Events API. We measure performance by calculating the performance of `FAMD` instructions. We also measure the global memory throughput using `nvprof` on the command line with `--metrics gld_throughput` option. In addition, we use `--metrics gld_efficiency` option to verify 100% global memory access efficiency is achieved in our optimization during development (we omit the presentation of result for efficiency verification due to page limit).

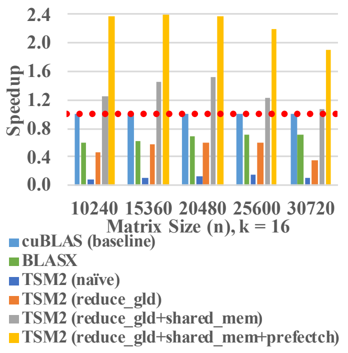


(a) Single precision

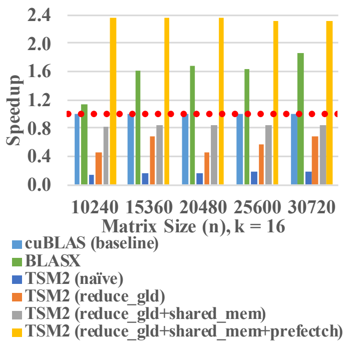


(b) Double precision

Figure 4.3: Speedup comparison for k = 4 on K40c.



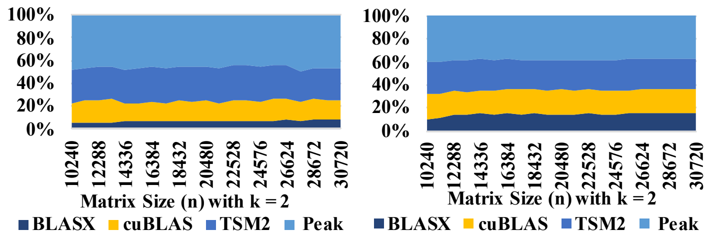
(a) Single precision



(b) Double precision

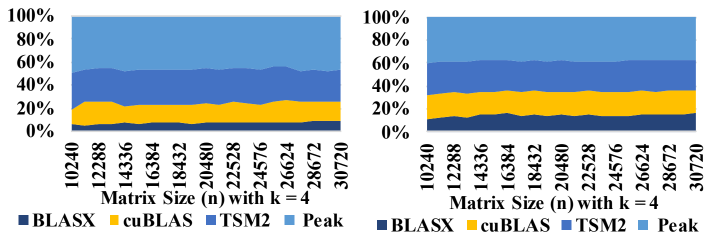
Figure 4.4: Speedup comparison for k = 16 on K40c.

Our input matrix is initialized with random floating point numbers (0 to 1). We test the multiplication between a large square sized matrix multiplies a tall-and-skinny matrix. The size of the large input matrix is from $10240 * 10240$ to $30720 * 30720$. The tall-and-skinny input matrix has size ranges from $10240 * k$ to $30730 * k$ with k equals 2, 4, 8, and 16.



(a) Single precision (b) Double precision

Figure 4.5: Memory throughput comparison for k = 2 on K40c.



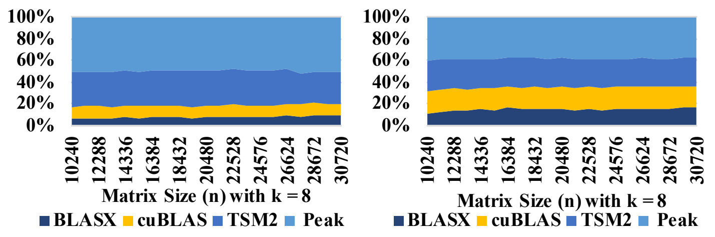
(a) Single precision (b) Double precision

Figure 4.6: Memory throughput comparison for k = 4 on K40c.

4.0.2 Tests with different combinations of optimization

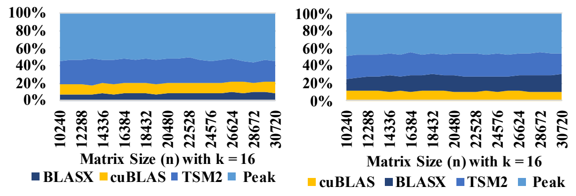
We use the GEMM in cuBLAS 9.0 as comparison baseline. We apply different combinations of optimization in `TSM2` and compare them with GEMM in cuBLAS and BLASX. We test totally four versions of `TSM2`:

- `naive`: the most straightforward inner product version as described in **Alg. 1**;
- `reduce_gld`: the outer production version as in **Alg. 2**. This version reduces the total number of global memory accesses from algorithm level;
- `reduce_gld + shared_mem`: based on outer production version as in **Alg. 2**, we add the use of shared memory, which leads to more efficient global memory access to matrix B;



(a) Single precision (b) Double precision

Figure 4.7: Memory throughput comparison for k = 8 on K40c.



(a) Single precision (b) Double precision

Figure 4.8: Memory throughput comparison for k = 16 on K40c.

- `reduce_gld + shared_mem + prefetch`: based on the outer production version as in **Alg. 2** and the use of shared memory, we add data prefetch. This is the best version of our optimized implementation, which is described in **Alg. 4**.

Limited by the page space, we only show the result on K40c GPU. Our optimization behaves similar on other GPUs. To evaluate our optimization, we need to determine by which resource our program is bounded. Since, $t_{2(k40c)}^{threshold} \approx 40$, the computation is always memory bound for the given k values. The optimized parameters are: $t_2 = k$, $t_3 = 4$, and $t_1 = 128$. The parameters are only applied to the last to versions of `TSM2`. **Fig. 4.1, 4.2, 4.3, 4.4** show the speedup of different versions in single and double precision. From the results, we can see that the `naive` version suffers from really poor performance due to the requirement of much higher number of global memory accesses in the inner product version. The `reduce_gld` version, on the other hand, significantly improve the performance compared to `naive` (2.2x - 4.7x faster), since it requires much lower number of global memory accesses. `reduce_gld + shared_mem` further improves the efficiency of global memory access to matrix B, which plays a vital role in the overall performance. In addition, the shared memory shares tiles of matrix B between threads within a thread block also reduced the total number of memory accesses to matrix B. This leads to additional 1.1x to 2.1x speedup. Finally, the data prefetch introduced in `reduce_gld + shared_mem + prefetch` version further mitigate the memory access bottleneck, which brings additional 1.3x - 3.5x speedup.

4.0.3 Memory throughput analysis

Fig. 4.5, 4.6, 4.7, 4.8 show the memory throughput of `TSM2` (with all optimizations), cuBLAS and BLASX in both single and double precision on K40c GPU. Result show that `TSM2` brings 12.5% - 26.6% (avg. 17.6%) improvement on GPU memory bandwidth utilization compare with cuBLAS and 20.1% - 38.8% (avg. 24.3%) improvement compare with BLASX.

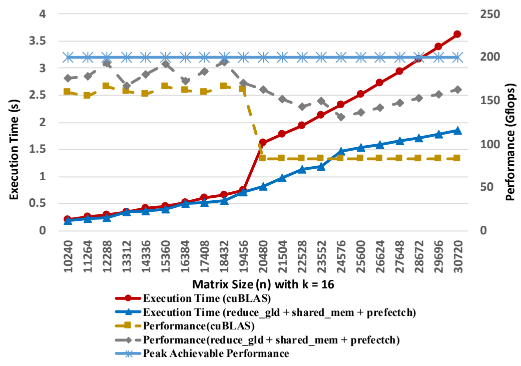


Figure 4.9: Comparing `TSM2` with cuBLAS on M40.

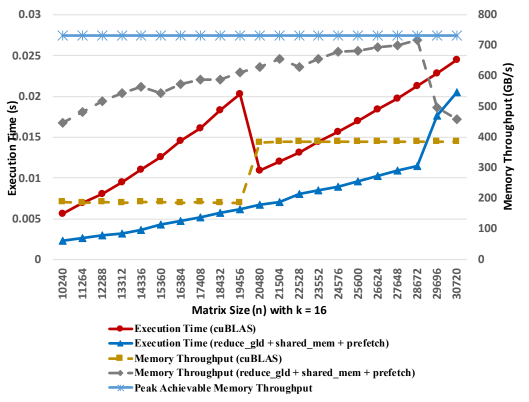


Figure 4.10: Comparing TSM2 with cuBLAS on P100.

4.0.4 Tests on different micro-architectures

In addition to Kepler micro-architecture, we also conduct test on newer Maxwell and Pascal GPUs. Similar as with Kelper GPU, we get $t_{2(m40)}^{threshold} \approx 6$ and $t_{2(p100)}^{threshold} \approx 50$

Tesla M40 has slower computing power, so the computation with input with $k = 16$ is compute bound. Our parameter optimization procedure also output parameters in favor of computing optimization: $t_2 = 8$, $t_3 = 4$, and $t_1 = 256$. As shown in **Fig. 4.9**, our optimized implementation achieves 1.1x -1.9x (avg. 1.47x) speedup on Tesla M40 with 7% to 37.3% (avg. 20.5%) computing power utilization improvement compared to the GEMM function in cuBLAS 9.0.

For P100 has much stronger computing power, as we can see the computation with input with $k = 16$ is memory bound. Our parameter optimization procedure also output parameters in favor of memory optimization: $t_2 = 4$, $t_3 = 4$, and $t_1 = 128$. As shown in **Fig. 4.10**, our optimized implementation achieves 1.1x - 3.0x (avg. 2.15x) speedup on Tesla P100 with 17% to 47.6% (avg. 34.7%) memory bandwidth utilization improvement compared to the GEMM function in cuBLAS.

Chapter 5

Conclusions

In this work, we first analyzed the performance of current GEMM in the latest cuBLAS library. We identified that current implement lack of full utilization of computing power or memory bandwidth when the input shape is tall-and-skinny. Then, we discovered the potential challenges of optimizing tall-and-skinny GEMM since its workload can varies between compute bound and memory bound. Next, we redesigned an optimized tall-and-skinny GEMM with several optimization techniques focusing on GPU resource utilization. Finally, experiment results that our optimized implementation can achieve 1.1x - 3x speedup on three modern GPU micro-architectures.

Bibliography

[1] K-means by NVIDIA.

[2] cublas benchmark, 2018.

[3] cuda programming guide, 2018.

[4] cudnn, 2018.

[5] Cula, 2018.

[6] Ptx programming guide, 2018.

[7] Ahmad Abdelfattah, David Keyes, and Hatem Ltaief. Kblas: An optimized library for dense matrix-vector multiplication on gpu accelerators. *ACM Transactions on Mathematical Software (TOMS)*, 42(3):18, 2016.

[8] Jieyang Chen, Sihuan Li, and Zizhong Chen. Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus. In *Networking, Architecture and Storage (NAS), 2016 IEEE International Conference on.*

[9] Jieyang Chen, Xin Liang, and Zizhong Chen. Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, 2016.

[10] Inderjit S Dhillon, Yuqiang Guan, and Brian Kulis. Kernel k-means: spectral clustering and normalized cuts. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 551–556. ACM, 2004.

[11] Tingxing Dong, Azzam Haidar, Piotr Luszczek, Stanimire Tomov, Ahmad Abdelfattah, and Jack Dongarra. Magma batched: A batched blas approach for small matrix factorizations and applications on gpus. Technical report, Technical report, 2016.

[12] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating numerical dense linear algebra calculations with gpus. *Numerical Computations with GPUs*, 2014.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[14] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. Libxsmm: accelerating small matrix multiplications by runtime code generation. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, 2016.

[15] Kuang-Hua Huang, Jacob Abraham, et al. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 1984.

[16] CUDA NVIDIA. Basic linear algebra subroutines (cublas) library, 2017.

[17] Dingwen Tao, Shuaiwen Leon Song, Sriram Krishnamoorthy, Panruo Wu, Xin Liang, Eddy Z. Zhang, Darren Kerbyson, and Zizhong Chen. New-sum: A novel online abft scheme for general iterative methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016.

[18] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 2010.

[19] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, Atlanta, GA, 2010. IEEE Computer Society.

[20] Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, University of California, Berkeley, 2016.

[21] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In *Proceedings of the 2016 International Conference on Supercomputing*, page 20. ACM, 2016.

[22] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, 2010.

[23] Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. Silent data corruption resilient two-sided matrix factorizations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017.

[24] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. Towards practical algorithm based fault tolerance in dense linear algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016.