International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 11, Number 2, pages 267–282, July 2021

Efficient Mixed-Precision Tall-and-Skinny Matrix-Matrix Multiplication for GPUs

Hao Tang, Kazuhiko Komatsu, Masayuki Sato and Hiroaki Kobayashi
Department of Computer and Mathematical Sciences, Graduate School of Information Sciences,
Tohoku University
2 Chome-1-1 Katahira, Aoba Ward, Sendai, Miyagi, 980-8577 JAPAN

Received: February 15, 2021
Revised: May 5, 2021
Accepted: June 6, 2021
Communicated by Yasuaki Ito

Abstract

General matrix-matrix multiplication (GEMM) is a commonly used BLAS level-3 routine in big data analysis and scientific computations. To further enhance the capability for GEMM computation on GPUs, manufacturers have introduced dedicated hardware for tensor and matrix operations into modern GPU architectures, which is called the Tensor Core unit. Mixed-precision GEMM based on the Tensor Core units has been introduced into many BLAS libraries and deep learning frameworks. However, these implementations are usually designed for large square matrices while these implementations tend to have a low performance for irregular-shaped matrices, especially for tall-and-skinny matrices.

This paper discusses on optimizing the GEMM computation suited for tall-and-skinny matrices on GPUs with three optimization methods: task mapping, memory access, and efficient use of Tensor core units by filling multiple fragments. First, the task mapping pattern of GEMM is optimized to make the implementation avoid launching too many thread blocks even when the sizes of input matrices are large. Second, the memory access pattern is optimized for half-precision tall-and-skinny matrices stored in the row-major layout. Third, Tensor Core units are effectively used even for extremely skinny matrices by filling multiple fragments into a Tensor Core operation. To examine the effectiveness of the proposed optimization methods, the experiments are conducted in two cases of GEMM that take tall-and-skinny matrices as input. With the proposed optimization methods, the evaluation results show that the optimized GEMM algorithms can make 1.07× to 3.19× and 1.04× to 3.70× speedups compared with the latest cuBLAS library on NVIDIA V100 and NVIDIA A100, respectively. By reducing the usage of the Tensor Core operations and utilizing the optimized memory access pattern, the optimized GEMM algorithms can save the energy consumptions of V100 and A100 by 34% to 74% and 62% to 82%, respectively.

Keywords: GEMM, GPU, Optimization

1 Introduction

General matrix-matrix multiplication (GEMM) is widely used in many applications, such as deep learning and scientific computing. Since the recent researches suggest that some applications have the tolerance to low-precision or mixed-precision computing, mixed-precision GEMM becomes a solution to shorten the running time of applications. Mixed-precision GEMM accepts low-precision matrices as inputs and provides outputs in high-precision. Using low-precision inputs can cut down the required data transfer, which is quite effective to improve performance when the performance is bounded by memory bandwidth.

Half-precision floating-point is one of the most commonly used low-precision formats. A half-precision element only occupies 2 bytes in memory, which significantly decreases the needed memory capacity for storage and the data transfer between the memory and the register file compared with cases of standard single-precision and double-precision data.

The advantages of low-precision and mixed-precision computing drive manufacturers to introduce mixed-precision computing units into computing platforms, such as NVIDIA's Volta, Google's TPU, and Fujitsu's A64fx [1][2][3]. To further improve the performance for applications like deep learning, these platforms feature tensor processing units providing hardware-accelerated matrix and tensor operations.

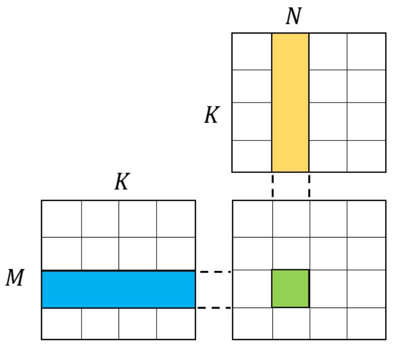
On a modern GPU, with the power of tensor processing units, the existing libraries can deliver high throughput for the mixed-precision multiplication of large matrices. For example, when the sizes of the two input matrices are $8,192 \times 8,192$, the latest vendor library delivers a performance of 83 Tflop/s and 294 Tflop/s on NVIDIA V100 and NVIDIA A100, respectively. These performances of GEMM are very close to their peak performances, 113 Tflop/s and 312 Tflop/s on V100 and A100, respectively.

There are some applications based on GEMM, in which the shape of the input matrix is tall-and-skinny, whose one dimension is small and another is much larger. For example, tall-and-skinny matrix-matrix multiplications are needed when applying the block Gram-Schmidt algorithm for Krylov basis, where the original matrix usually has 100K to 200K rows and a small number of columns [4]. Another example is the k-means algorithm, where the distance between a data point and a centroid is calculated by $||x - y||^2 = ||x||^2 + ||y|| - 2xy$. Here, x indicates the feature vector of a data point, and y indicates a centroid. The norm of vectors is usually implemented by vector-vector multiplication. Here, the xy of multiple data points and centroids are combined to make the computation become a GEMM of matrix $X(n \times d)$ by matrix $Y(d \times c)$, which can be accelerated by the BLAS routines [5][6]. In some datasets [7][8], the numbers of dimensions and centroids are much fewer than the number of data points. As the result, the computation becomes a multiplication of a tall-and-skinny matrix and a small matrix.

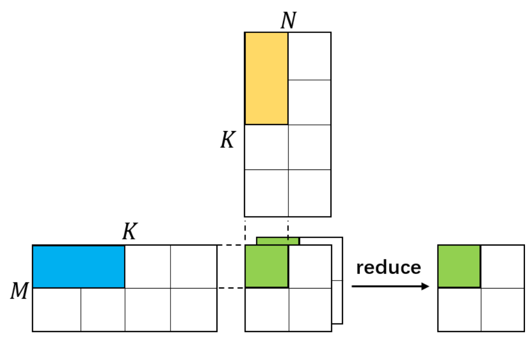
Since the conventional BLAS libraries are generally implemented for large square matrices, some designs are not suitable for tall-and-skinny matrices. First, the conventional task mapping launches a large number of thread blocks even for tall-and-skinny matrices, which brings additional reduction operations or memory accesses. Second, the row-wise access pattern cannot achieve a high bandwidth for half-precision tall-and-skinny matrices stored in the row-major layout. In addition, in current GPU architectures, the Tensor Core units can only handle matrices with a fixed size. As a result, a tall-and-skinny matrix may not be able to fully occupy the input of the Tensor Core units. Therefore, mixed-precision GEMM for tall-and-skinny matrices on GPUs should be optimized.

The contribution of this paper is the optimization of the GEMM algorithm for tall-and-skinny matrices in which the conventional implementations are inefficient. Two cases of GEMM are under consideration. The first case is the multiplications of two tall-and-skinny matrices $A^T \times B$. The second case is $A \times B$, where A is a tall-and-skinny matrix, and B is a small square matrix. By optimizing task mapping, memory access, and use of Tensor core units, the proposed GEMM algorithm can effectively achieve a speedup over the vendor library.

The rest of this paper is organized as follows. Section 2 discusses the background and reviews related work. Section 3 proposes an efficient GEMM algorithm for tall-and-skinny matrices. Section 4 provides the evaluation results in terms of performance and energy consumption. Section 5 concludes this paper.



(a) Tiling



(b) Splitting the K dimension

Figure 1: Compute GEMM with GPUs.

2 Discussion on GEMM Algorithms for Mixed-Precision Tall-and-Skinny Matrices on the GPU Architecture

2.1 Modern GPU Architecture

GPUs are widely used to accelerate computation in many applications, such as deep learning and engineering simulations. NVIDIA V100 GPU is implemented with the Volta architecture [9], which is a professional accelerator for deep learning and high-performance computing (HPC). A V100 GPU has 80 SMs, and each SM has 64 single-precision floating-point units, 32 double-precision units, and 64 integer units. Each SM is partitioned into 4 processing blocks, which can execute 4 warps simultaneously. Unlike previous generations [10], which could not execute floating-point operations and integer operations simultaneously, the Volta architecture has separated floating-point and integer cores, allowing simultaneous execution of floating-point and integer operations at full throughput. Besides, The Volta architecture is the first GPU architecture that introduces tensor processing units, which are called Tensor Core units, to accelerate tensor and matrix operations. By paring normal arithmetic units and Tensor Core units, a Tesla V100 GPU can provide 14 Tflop/s for single-precision, 7 Tflop/s for double-precision, and 113 Tflop/s for Tensor Core operations.

Ampere is the newest GPU architecture released by NVIDIA in 2020 [11]. The overall structure of Ampere is similar to Volta, while introducing some new features. One of the features is the third-generation Tensor Core units, which support high-precision inputs, for example, double-precision. The Tensor Core units in the Ampere architecture supports structured sparsity at the hardware level, which is a frequently used trick in deep neural networks (DNNs) to accelerate training and inferencing.

2.2 Computing GEMM on GPUs

Tiling is commonly used in many state-of-the-art GEMM libraries, such as cuBLAS and CUTLASS [12][13], which are the high-performance vendor libraries. Figure 1 (a) shows a common GEMM implementation with the tiling optimization on GPUs. Tiling splits a result matrix into multiple tiles and assigns the tiles as computing tasks to different thread blocks. The computation within a thread block is further assigned to threads. This strategy works well when the sizes of M and N dimensions are large because launching more thread blocks can improve the utilization of GPU hardware resources. In special cases where the sizes of M and N dimensions are not so large, both cuBLAS and CUTLASS further split computation on the K dimension to improve parallelism, as shown in Figure 1 (b). When splitting the K dimension, the tiles calculated by thread blocks are partial results of the final result. Thus, reducing operations in the global memory on these partial results are required at the end of the computation.

A commonly used memory access pattern for loading a fragment of a row-major matrix is shown

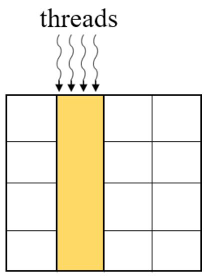


Figure 2: The row-wise access.

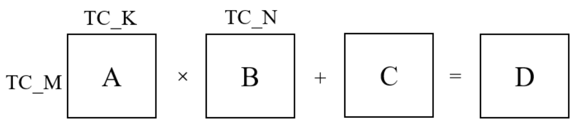


Figure 3: A Tensor Core operation.

in Figure 2. For a row-major matrix, loading elements with row-wise accesses is more efficient than loading column-wise accesses because memory accesses can be coalesced for adjacent elements. However, when the matrix is a tall-and-skinny matrix that only has few elements in each row, loading a row at a time results in a low value of accessed bytes per instruction. Therefore, such a memory access pattern becomes less efficient for tall-and-skinny matrices.

Furthermore, when the width is not multiple of 2, the starting address of each row becomes only 2-byte aligned. Alignment is crucial because it is required when using vectorized memory access on GPUs. For example, when the matrix has a size $(32 \times K)$, the beginning of each row is 16-byte aligned. Thus it can be accessed with vectorized data types such as int4 or float4. When the size is $(30 \times K)$, threads can access data with a width of 4 bytes, such as int or float. Using larger data types can reduce the required instructions to load a matrix and improve the speed of data transfer. Different memory access patterns makes the performance of cuBLAS vary a lot for different inputs. Thus, memory access should be optimized to achieve high performance for all kinds of inputs.

2.3 Half-Precision Floating-Point Computing

Many studies have shown that deep neural networks have the tolerance to low-precision computing such as [14][15]. This motivates manufacturers to introduce low-precision computing units for matrix or tensor operations, which are the most frequently required operations in deep learning. NVIDIA introduced half-precision arithmetic units in the Pascal architecture [10], and the support for half-precision arithmetic is further enhanced in the Volta and Turing architectures for GEMM computation with the Tensor Core units [9][16].

Figure 3 shows a Tensor Core operation of modern GPU architectures. A Tensor Core operation accepts matrices A, B, and C as inputs, and performs a matrix multiply-and-accumulate operation $D = A \times B + C$. Matrices A and B should be half-precision matrices, and the matrices C and D can be either half-precision or single-precision.

Using half-precision floating-point data as inputs can significantly reduce the memory transfer compared the cases of single-precision and double-precision data because the half-precision data has only two bytes per element. Moreover, it can reduce the silicon budget for arithmetic units and thus allows manufacturers to implement more units on a chip.

However, simply replacing high-precision data with low-precision data, the performance may not be improved because some operations are not optimized for half-precision data yet on GPUs. First, the optimization of *nvcc* for memory access is not available for half-precision elements [17]. Second, using the half-precision data type to access elements requires more instructions compared with using larger data types. Last but not least, the half-precision arithmetic units on GPUs are paired. Thus, instead of directly using the half-precision data type, it is recommended to use the data type that combines two half-precision elements to operate multiple half-precision elements.

2.4 Related work

Chen et al. have pointed out that the low utilization of GPU resources is the main reason for the low performance of cuBLAS for irregular-shaped inputs [18]. They have designed two classes of algorithms for GEMM that involve tall-and-skinny matrices. The first algorithm handles GEMM of a large-to-medium regular-shaped matrix and a tall-and-skinny matrix. The second algorithm handles GEMM of a tall-and-skinny matrix and a small regular shaped matrix. The key idea of the proposed algorithms is to adjust the tiling parameters to make the kernel always memory-bound. To overlap computation and memory access, they exploit a double-buffering technique for the global memory accesses. Their objective inputs are column-major matrices, which means that the threads can access elements efficiently if each thread accesses a row because a tall-and-skinny matrix has a large number of rows.

Ernst et al. have designed an optimized kernel for the multiplication of two tall-and-skinny matrices $C = A^T \times B$ [19], where 'tall-and-skinny' is defined as the width of the matrix is no larger than 64 and the number of rows is larger than 10^6. In their implementation, the computation is only parallelized over the loop of the K dimension for thread block-level parallelism to keep a good computational intensity. For thread-level parallelism, the computation is tiled on the M and N dimensions to ensure that the register usage does not exceed hardware limitation. Considering the limitation of existing compilers, they designed a code generator in Python that can unroll loops with variable iteration count and remove unnecessary guarding statements for each specific input. With the help of the code generator and selecting the optimal tile size based on evaluated results, the generated kernels can always achieve at least 2/3 of the ideal performance. Because the implementation of [19] is designed for double-precision matrices, it does not involve the usage of Tensor Core units.

Yan et al. have pointed out that there is still space for improvements in Tensor Core-based GEMM offered by the vendor library [20]. They have benchmarked the CPI of the load instructions and Tensor Core operations with Streaming ASSembler (SASS). They use the CPI of memory access and computing operations to guide the process of selecting tiling size and instructions scheduling. The evaluation results show that their kernels achieve 1.6× to 3× speedups for large matrices compared with cuBLAS 10.1. Although several studies have been done on optimizing the GEMM algorithms for tall-and-skinny matrices, Tensor Core-based mixed-precision GEMM algorithms that involve tall-and-skinny matrices still remain to be optimized, which motivates the study of this paper.

3 An Efficient mixed-precision tall-and-skinny matrix-matrix multiplication algorithm for GPUs

This section proposes an efficient GEMM algorithm for two types of GEMM by optimizing the task mapping, memory access, and the use of Tensor Core units.

3.1 Task Mapping

A CUDA kernel consists of execution units in multiple layers, such as threads and thread blocks [21]. How to map the computation tasks to these units is always a key to design a high-performance kernel. This subsection analyzes the problems of the conventional task mapping method for GEMM in two cases of tall-and-skinny matrices to provide the guideline of optimizations.

For the multiplication of two tall-and-skinny matrices $C = A^T \times B$, the computation is only parallelized over the K dimension at the thread block level to avoid decreasing computational intensity. In the conventional implementations, the computation of a thread block is further parallelized over the M and N dimensions. When the length of the K dimension varies, the number of running warps

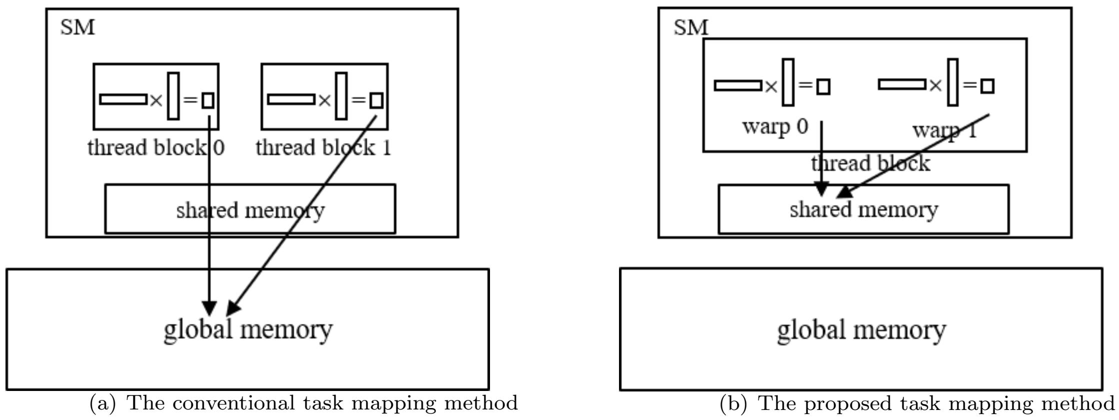


Figure 4: Task mapping for the multiplication of two tall-and-skinny matrices.

on SMs can only be adjusted by changing the number of thread blocks. However, as Figure 4 (a) shows, the partial results calculated by thread blocks need to be reduced in the global memory because thread blocks cannot directly share data even when they execute on the same SM. cuBLAS and CUTLASS use an additional kernel to reduce partial computing results. Such a reduction method introduces more memory accesses because it needs to store the partial results to the global memory so that the results can be accessed by another kernel.

To make reducing partial result efficient, this paper proposes an optimized task mapping method, as shown in Figure 4 (b). The basic idea is to parallelize the computation over the K dimension for both thread block-level and warp-level parallelisms. The partial results obtained within a thread block can be reduced within the SM.

The pseudo code of the proposed task mapping is shown in Algorithm 1. Each warp has its own buffer for matrices A and B in the shared memory. bk is the height of the fragment that a warp fetches at a time, which is set to multiples of TC_K in Figure 3. At each iteration, each warp fetches a fragment from matrices A and B, and performs the multiplication. Then, the address of each warp is shifted with a stride of $blockDim \times num\_of\_warps \times bk$ to prepare for the next iteration.

The proposed task mapping method brings two advantages: First, the number of warps running on SMs can be adjusted without introducing additional thread blocks. Second, there is no need for synchronization within a thread block because the data transfer and computation of different warps are independent.

To avoid using a additional kernel to reduce partial results, this paper uses the CUDA atomic functions to add the partial results to the global memory. Although the atomic functions that access the global memory have been optimized by performing the atomic operations in the L2 cache [22], it is still less efficient than the atomic functions that access the shared memory. Thus, the partial results are reduced in the shared memory at first, then the results of each thread block are further reduced in the global memory. Reduction can also be implemented without using the atomic operations. However, in that case, the kernel has to wait for the completion of all the SM to start the reduction. In addition, with the proposed task mapping method, the number of the partial results is limited to the number of SMs, which will not cause much congestion on the memory bus.

To fully use all the SMs on a GPU, the number of thread blocks is set to as same as the number of SMs. Unlike the conventional designs, the number of required operations for reducing the partial results of different thread blocks always keeps the same because the number of thread blocks is fixed.

For the multiplication of a tall-and-skinny matrix and a small matrix, the task mapping pattern can also be optimized by fixing the number of thread blocks. As described in Section 2.2, the number of launched thread blocks is decided by the sizes of M and N dimensions in conventional implementations. Figure 5 shows the conventional task mapping method. Since matrix A has a large number of rows, the number of launched thread blocks will be large. However, the execution of each thread block is short if the size of the K dimension is small. A large number of thread blocks

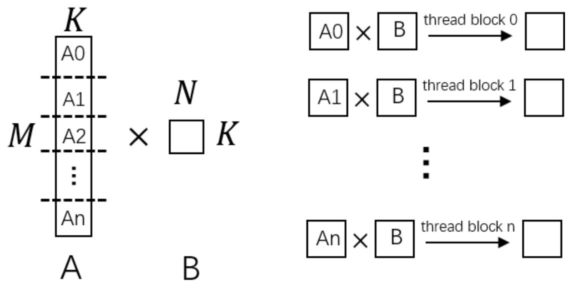


Figure 5: The conventional task mapping for the multiplication of a tall-and-skinny matrix and a small matrix.

bring much more data transfer because matrix B needs to be loaded by every thread block. For example, considering $A(32768 \times 32) \times B(32 \times 32)$ and a tile size(128×32), the computation tasks will be assigned to $32768/128 = 256$ thread blocks. As the result, the matrix B needs to be loaded 256 times.

Figure 6 shows the proposed task mapping pattern, which always sets the number of thread blocks to the number of SMs on the GPU. At beginning of the kernel, each thread block loads matrix B into the shared memory, and then it can be reused by all warps. A thread block continuously performs computation along the long M dimension. Matrix B is stored in the register files of each warp, and thus each thread block loads matrix B only once. For example, in the case of NVIDIA Tesla V100, the number of thread blocks is set to 80, which means matrix B only needs to be loaded 80 times regardless of how tall matrix A is. This design is also used in the GPU kernel optimization for RNNs [23].

3.2 Memory access for half-precision tall-and-skinny matrices

This subsection proposes two optimized memory access patterns for half-precision tall-skinny matrices. Loading a single row at a time forces threads to use the 2-byte data type to access elements when the width of the matrix is not multiple of 2. Since the computation is not parallelized on the short dimension of a tall-and-skinny matrix, loading multiple rows can solve the problem. For

Algorithm 1 : The proposed Implementation

```
INPUT: half-precision Matrix A ($K \times M$), B ($K \times N$)
OUTPUT: single-precision Matrix C
__shared__ A_buf[num_of_warps][32 × bk]
__shared__ B_buf[num_of_warps][32 × bk]
C_frag, A_frag, B_frag //Tensor Core fragments
offset ← (blockId ×num_of_warps + warpId) × bk
while offset < $K$ do
    A_buf[warpId] ← 32 × bk of A
    B_buf[warpId] ← 32 × bk of B
    for j=0 to bk by TC_K do
        load elements to A_frag, B_frag
        //Perform Tensor Core operation
        C_frag ← A_frag × B_frag + C_frag
    end for
    offset ← offset + blockDim × num_of_warps × bk
end while
C ← Reduction of partial results
```

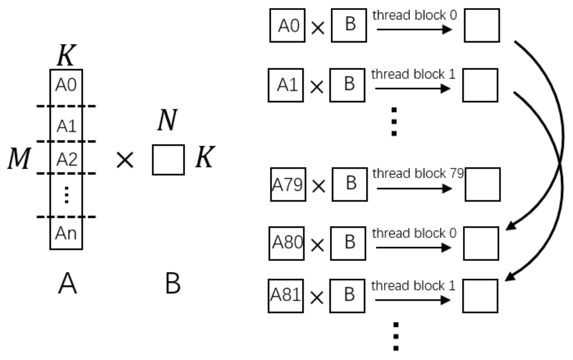


Figure 6: The proposed task mapping method for the multiplication of a tall-and-skinny matrix and a small matrix.

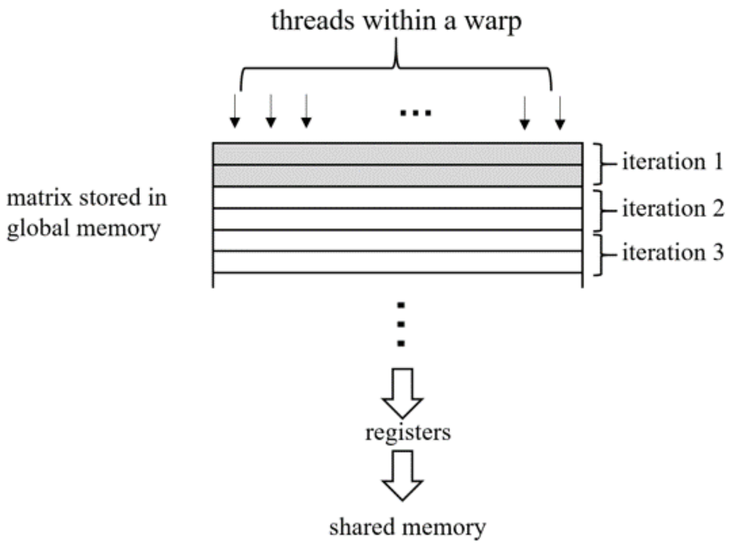


Figure 7: Load a fragment of a matrix whose width is multiple of 2.

example, the starting address of every 2 rows is always 4-byte aligned regardless of whether the width is multiple of 2 or not, thus each thread can access a 4-byte data at a time. Using larger data types to access data is a commonly used approach to the acceleration of the data transfer because it reduces the required instructions of loading data.

Figure 7 shows an implementation of loading a matrix whose width is multiple of 2. Each thread accesses 4 bytes in the global memory, and stores the data into the shared memory. The data stored in the global memory cannot be directly loaded into the shared memory. Thus, the data transfer has to involve the use of registers.

For a matrix whose width is not multiple of 2, threads cannot directly store the loaded 4-byte data to the shared memory, because the destination address is not 4-byte aligned. Thus, threads need to split the loaded data into two half-precision elements, and then store the elements into the shared memory, as shown in Figure 8.

For the matrix with a width of 32, loading 2 rows exactly needs all the threads of a warp. When the width of an input matrix becomes smaller, this memory access pattern becomes less efficient as well because more threads within a warp need to be blocked. For example, when M = 14, only 14 threads are needed to load two rows, which means that more than half of the threads within a warp are not used. Therefore, skinnier inputs should be handled specially. For matrices whose width is smaller than or equal to 16 and 8, the proposed method switches to loading 4 rows and 8 rows at

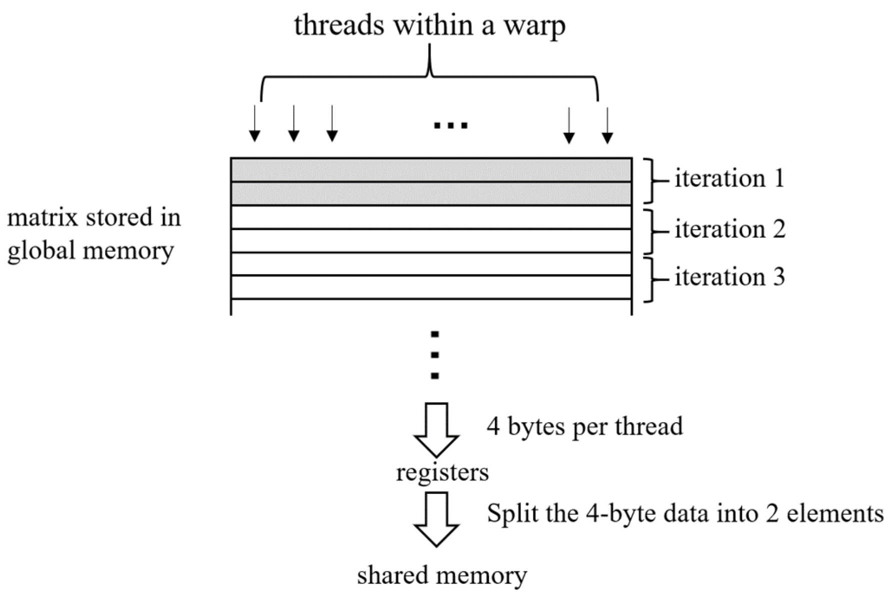


Figure 8: Load a fragment of a matrix whose width is not multiple of 2.

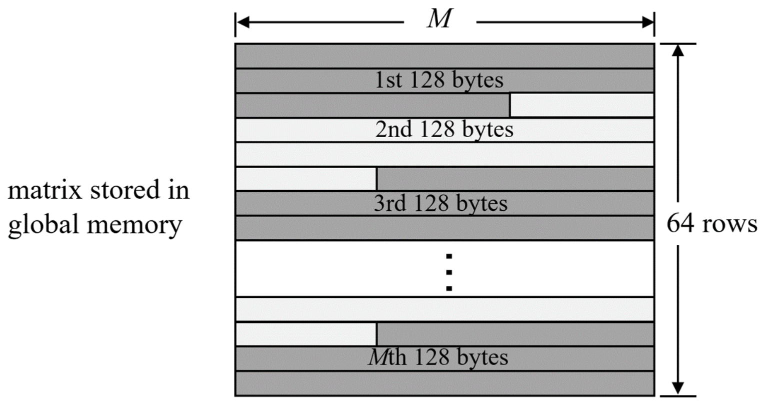


Figure 9: Load a $64 \times M$ fragment.

a time, respectively. With this optimization, the kernel can avoid blocking too many threads for different inputs when loading matrices from global memory

Another efficient memory access pattern is accessing the 2-D matrix as accessing a 1-D array, which uses a warp to load 128 bytes at a time. As Figure 9 shows, all of the threads within a warp always participate in every memory access instruction. This reduces the required instructions compared with loading a fixed number of rows at a time. More importantly, since cudaMalloc() guarantees that the starting address of an allocated buffer is 256-byte aligned, all of the accesses of a warp are 128-byte aligned. Therefore, all the memory instructions can be served by 4 transactions of the GPU DRAM and 1 transaction of L2 and L1 caches without loading redundant bytes. .

To make the size of accessed data be multiple of 128 bytes, a warp fetches a fragment with 64 rows before computation, which means bk in Algorithm 1 is set to 64 in this case. For example, when the matrix width is M, a warp takes M iterations to load $M \times 64 \times sizeof(half)$ bytes.

This memory access pattern fully uses every byte fetched by the hardware. However, it is difficult to design a general kernel with this memory access pattern because the compiler cannot unroll the loop that loads data from the global memory with a variable number of iterations.

Unrolling the loop of memory access is quite important in CUDA programs. A data assignment

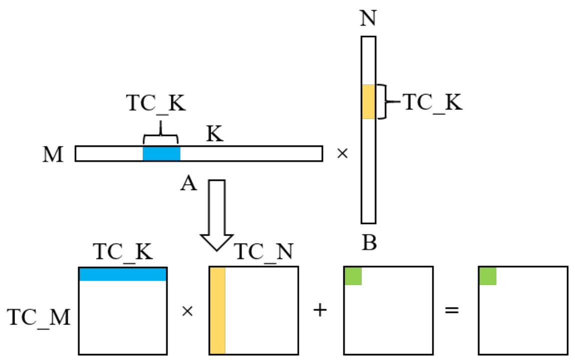


Figure 10: Extremely skinny matrices cannot fully occupy a Tensor Core operation.

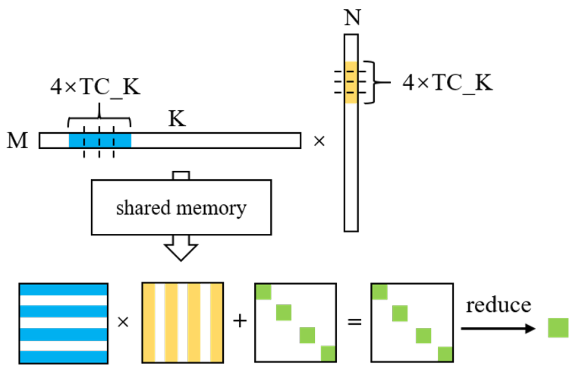


Figure 11: Fill multiple fragments into a Tensor Core operation.

statement between the shared memory and the global memory is composed of two instructions. The first instruction loads the data into a register, and the second instruction stores the data to the shared memory. Because a GPU can overlap multiple load instructions, the compiler can optimize the loop by putting all the load instructions before the store instructions. Without this optimization, the execution will stall at every iteration to wait for the data loaded from the global memory, resulting in a significant degradation in performance. To makes use of the optimization, the iterations of a loop must be constant, otherwise the number of required registers is unknown. Although the matrix width can be set as a template parameter, the implementation still cannot avoid inserting a large number of branching statements and template instances for different specific inputs.

3.3 Efficient use of Tensor Core operations for extremely skinny matrices

Due to the hardware restriction, Tensor Core units can only handle multiplications of fragments with a fixed size. As shown in Figure 10, when the input matrices are even skinnier than the Tensor Core operation, a part of the computation of the Tensor Core operation is wasted. This can be optimized by filling multiple fragments of the input matrices into a Tensor Core operation. Figure 11 shows the proposal that uses a Tensor Core operation to handle the multiplication of multiple fragments when the sizes of the M and N dimensions are smaller than TC_M and TC_N. A Tensor Core loading operation loads a small matrix from memory at a time. Thus, this optimization is implemented by rearranging the data layout in the shared memory. Then, the Tensor Core load function can load multiple fragments into a Tensor Core fragment, which is like folding the original input matrices. With the proposed method, more computations of a Tensor Core operation can be utilized, making the usage of Tensor Core operations more efficient.

Table 1: Experimental environment

CPU	Intel Xeon Gold 6126
OS	CentOS 7.7
Toolkit	CUDA 11.0
Driver	450.51.06
GPU DRAM (NVIDIA V100)	880GB/s
GPU DRAM (NVIDIA A100)	1550GB/s
Number of SMs (NVIDIA V100)	80
Number of SMs (NVIDIA A100)	108

4 Evaluation

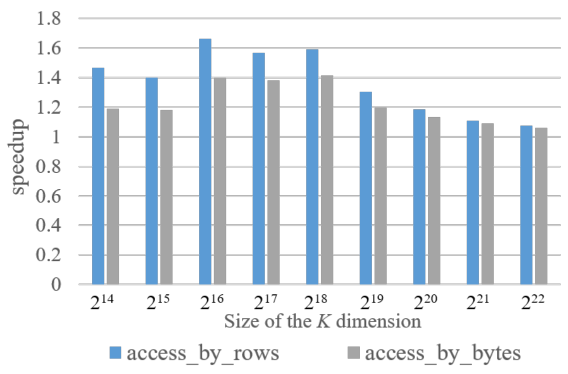
The experiments are conducted for two cases of GEMM computation. The first case is $C = A^T \times B$, where both A and B are tall-and-skinny matrices. The second case is $C = A \times B$, where A is a tall-and-skinny matrix, and B is a small square matrix. Inputs are half-precision matrices, and outputs are in single-precision matrices. All matrices are stored in the row-major layout, and the widths of matrices vary from 1 to 32. The GPU kernels are implemented with CUDA 11.0. The version of the GPU driver is 450.51.06. The experiments use cublasGemmEx() to perform the mixed-precision GEMM in cuBLAS. NVIDIA V100 and A100 are used for evaluation. The system frequencies of GPUs are fixed to 1230MHz and 1410MHz using the nvidia-smi tool for V100 and A100, respectively. The details of the hardware specification are given in Table 1.

4.1 Multiplication of two tall-and-skinny matrices

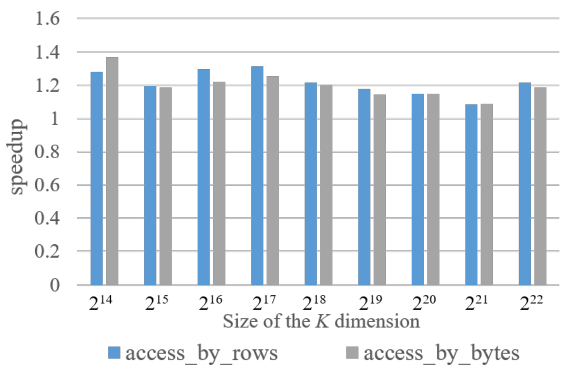
Figure 12 shows the speedup compared with cuBLAS 11. The horizontal axis shows different sizes of the K dimension, and the vertical axis represents the speedup over cuBLAS. The height of input matrices is set to 2^{14} up to 2^{22}. The sizes of M and N dimensions are set to 32. *access_by_rows* and *access_by_bytes* represent the implementations with memory access patterns of loading multiple rows at a time and loading 128 bytes at a time, respectively. For $M = N = 32$, both of the two memory access patterns load 128 bytes at a time. However, bk of access_by_rows is set to 16, and bk of access_by_bytes is set to multiple of TC_K. With a smaller bk, the first implementation can assign the workload more evenly to different warps. Therefore, *access_by_rows* performs better than *access_by_bytes*. The difference between the two implementations becomes smaller as the size of the K dimension increases because all the warps will have enough work to do with a large K dimension.

For a larger K dimension, cuBLAS launches more thread blocks, which produces more global partial results and thus makes the time for reduction longer. For the proposed method, the number of global partial results always keeps the same as the number of thread blocks is fixed. Therefore, the achieved speedup increases as K increases from 2^{14} to 2^{18}. When K is set to from 2^{18} to 2^{22}, the percentage of needed time for reduction in overall execution time becomes very low. Thus, there is only a little difference in performance between the different implementations for the case of $K = 2^{22}$ on Tesla V100. On NVIDIA A100, cuBLAS switch the number of thread blocks from 81 to 324 when $K = 2^{22}$, thus the speedup suddenly increases.

Figure 13 shows the performance of implementation with the optimization of filling multiple fragments into Tensor Core operations. The horizontal axis shows different sizes of the M and N dimensions, and the vertical axis represents the speedup against the performance of the implementation without the optimization. Both of the two implementations use the memory access pattern of loading a fixed number of rows at a time, and the result with loading 128 bytes at a time is similar. Currently, the size of the Tensor Core operation is set to 16×16, which means the proposed method is appliable only when M and N are smaller than or equal to 8. Although folding makes much more computation of Tensor Core operations useful, it brings 1% up to 24% and 1% up to 48% improvement compared with the implementation without the optimization on V100 and A100, respectively. When M and N get smaller, the required data transfer decreases. It makes less computation overlapped by data transfer. Therefore, folding is more effective for small M and N.

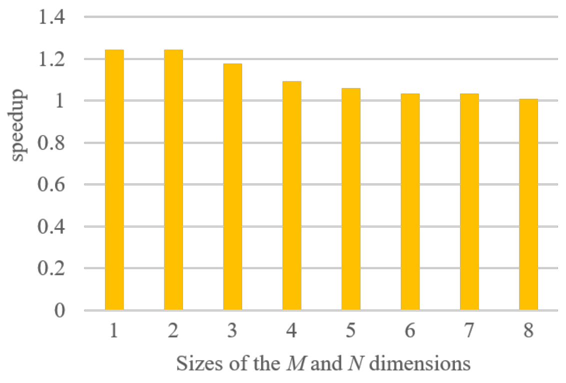


(a) Tesla V100

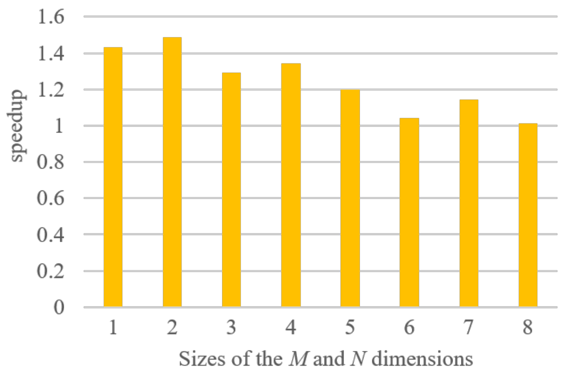


(b) NVIDIA A100

Figure 12: Comparison with cuBLAS with $M = N = 32$.



(a) NVIDIA V100

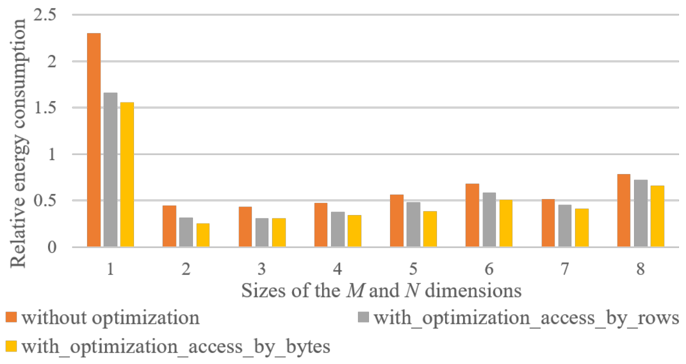


(b) NVIDIA A100

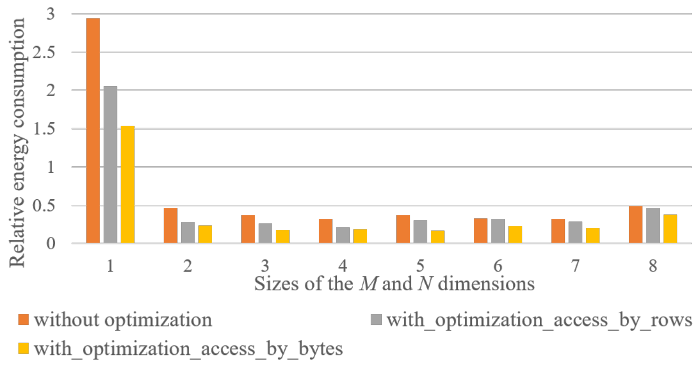
Figure 13: Speedup obtained by filling multiple fragments into Tensor Core operations with $M = N$ and $K = 2^{23}$.

Filling multiple fragments into a Tensor Core operation is effective at reducing energy consumption as well because it makes kernels use fewer Tensor Core operations. This paper profiles the energy consumption by measuring the average power draw during the execution of kernels using the nvidia-smi tool. Figure 14 shows the relative energy consumption of implementations with and without optimization compared with cuBLAS. The horizontal axis shows different sizes of the M and N dimensions, and the vertical axis represents the relative energy consumption against cuBLAS. As the result shows, the implementations of the proposed algorithm are much more energy-efficient than cuBLAS, except in the case of $M = N = 1$, where cuBLAS uses CUDA core-based kernels. The implementation optimized by filling multiple fragments into a single Tensor Core operation reduces the energy consumption by 7% to 29% and 3% to 39% on V100 and A100, respectively. Similar to the results in Figure 13, the smaller the sizes of input matrices are, the more effective the optimization will, be because filling more fragments saves more usage of the Tensor Core operations. The energy consumption can be further reduced by using the memory access pattern of loading 128 bytes at a time because the redundant data transactions are excluded.

Figure 15 shows the comparison between achieved performance and the ideal performance calculated with the roofline model [24]. The horizontal axis shows different sizes of the M and N dimensions, and the vertical axis represents the performance. Figure 15 shows that, on both of the platforms, performances of the two implementations with the proposed optimization methods are close to the ideal performance for most of the tested cases. The performance of NVIDIA A100 is



(a) NVIDIA V100



(b) NVIDIA A100

Figure 14: Relative energy consumption with $M = N$ and $K = 2^{23}$.

1.5× higher than the performance of Tesla V100, benefiting from the high bandwidth of the off-chip memory. The performance on V100 is closer to the ideal performance than the performance on A100. In A100, there is a new feature called asynchronous copy. Properly using the asynchronous copy can directly load data from the global memory to the shared memory, which accelerates the data transfer. Thus, we think if the asynchronous copy is properly used, the achieved bandwidth and the performance on A100 will be improved. When $M = N = 3$, the proposed method achieves the maximum improvements over cuBLAS, which are 3.17× and 3.70× on V100 and A100, respectively. This is because cuBLAS does not have an efficient data transfer implementation when there are only few elements in a row of the input matrices, especially when the number of elements is not multiple of 2.

Although the threads can only access the shared memory with the 2-byte data type when the sizes of M and N dimensions are not multiples of 2, the performance does not decrease a lot. This is because the bandwidth of the shared memory is much higher than the global memory, which only has little impact on the performance. As expected, the implementation with the memory access pattern of loading 128 bytes at a time has the highest performance for most cases.

4.2 Multiplication of a tall-and-skinny matrix and a small matrix

Figures 16(a) and (b) show the comparison in the case of $C = A \times B$ between the proposed method and cuBLAS on V100. The horizontal axis shows different sizes of the K and N dimensions, and the vertical axis represents the speedup over cuBLAS. The size of the M dimension is set to 2^{16} and 2^{20} in Figures 16 (a) and (b), respectively. The optimized memory access pattern can be applied to this case as well. With the optimized memory access pattern, the implementation makes 15%-41%

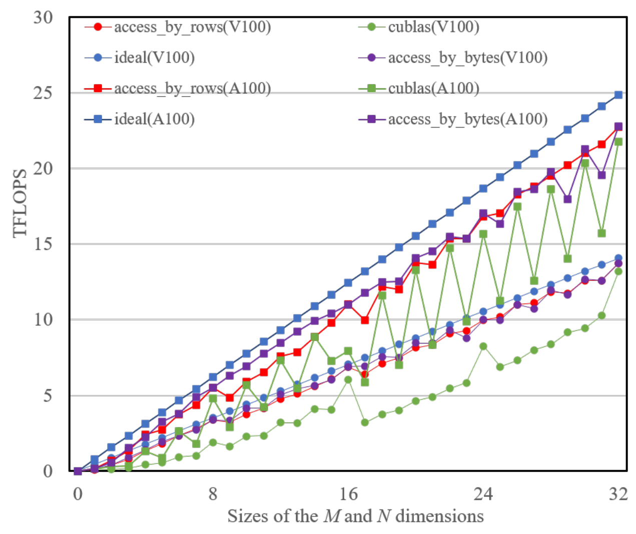
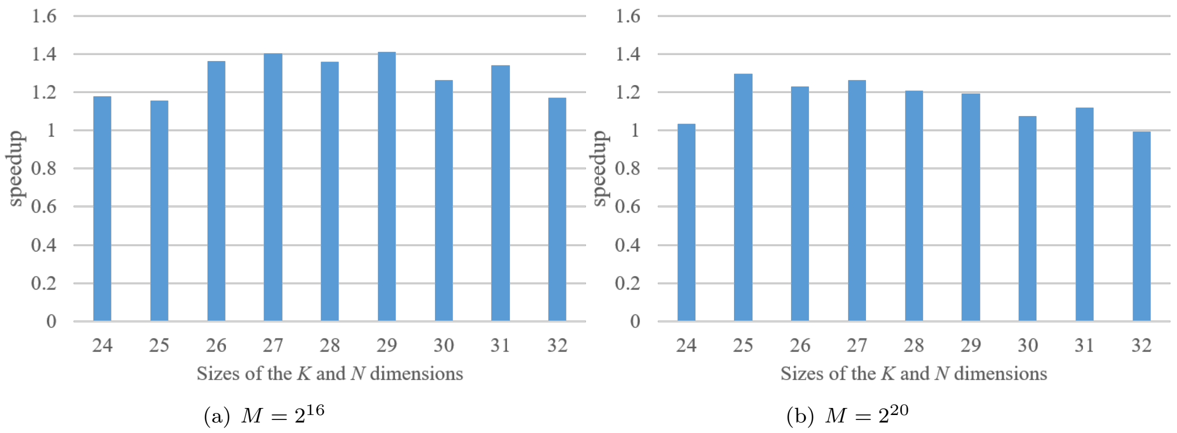


Figure 15: Comparison with ideal performance with $M = N$ and $K = 2^{23}$.



(a) $M = 2^{16}$

(b) $M = 2^{20}$

Figure 16: Comparison with cuBLAS for the multiplication of a tall-and-skinny matrix by a small matrix.

and 7%-29% improvements over cuBLAS when the size of the M dimension is set to 2^{16} and 2^{20}, respectively. When $M = 2^{16}$, cuBLAS launches $2^{16}/128 = 512$ thread blocks. When $M = 2^{20}$, cuBLAS launches $2^{20}/128 = 8192$ thread blocks, which means that the matrix B is loaded 8192 times. On the other hand, the proposed method only launches 80 thread blocks for both of the cases. However, launching over 8000 thread blocks does not significantly degrade the performance of cuBLAS. This is because the matrix B is stored in SM's L1 cache after one thread block on this SM accesses matrix B in the global memory. Therefore, the other thread blocks on the SM can directly fetch the matrix B in the L1 cache, which is much more efficient than loading data from the global memory.

5 Conclusions

Vendor libraries provide high-performance mixed-precision GEMM implementations on GPUs for large square matrices. However, these implementations are not suitable for tall-and-skinny matrices. In order to improve the performance of mixed-precision GEMM on GPUs for tall-and-skinny matrices, this paper proposes an efficient GEMM algorithm with three optimization methods. First, a method that optimizes the task mapping has been proposed to make the kernels avoid launching too many thread blocks while keeping enough warps running on SMs. Second, two efficient memory access patterns are exploited to load half-precision matrices from the global memory. Third, to improve the utilization of Tensor Core operations for extremely skinny inputs, a method that fills multiple fragments into a Tensor Core operation has been proposed. According to the evaluation results, the proposed optimization methods can make 1.07× to 3.19× and 1.04× to 3.70× speedups compared with the latest cuBLAS library on NVIDIA V100 and A100, respectively. With a higher hardware utilization, the proposed optimization methods can save 34% to 74% and 62% to 82% energy consumption on NVIDIA V100 and A100, respectively. According to the experimental results, the height of the fragment that a warp fetches at a time also has an impact on the performance, especially when the size of the K dimension is not large. This is left to be discussed and optimized in future work. As the usage of the standard arithmetic units is not considered yet, we would also like to make an implementation of the proposed method on the standard arithmetic units and compare the performance and energy efficiency with the Tensor Core in the future. Last but not least, although the output of the mixed-precision computing is still in single precision, the numerical behavior is different with the computation with half-precision inputs. Thus, it is necessary to assess the impact of mixed-precision computing in the real applications in the future.

Acknowledgments

This research was partially supported by MEXT Next Generation High-Performance Computing Infrastructures and Applications R&D Program, entitled "R&D of A Quantum-Annealing-Assisted Next Generation HPC Infrastructure and its Applications." Grants-in-Aid for Scientific Research (A)#19H01095, and Grants-in-Aid for Scientific Research (C)#20K11838.

References

[1] NVIDIA. Nvidia tensor cores. `https://www.nvidia.com/en-us/data-center/tensorcore`, 2019.

[2] Google. Google cloud tpu. `https://cloud.google.com/tpu`.

[3] Toshio Yoshida. Fujitsu high performance cpu for the post-k computer. In *Hot Chips*, volume 30, 2018.

[4] Denis Vanderstraeten. A stable and efficient parallel block gram-schmidt algorithm. *Lecture Notes in Computer Science*, 1685:1128–1135, 1999.

[5] NVIDIA. kmeans. `https://github.com/NVIDIA/kmeans`, 2020.

[6] 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, 2004.

[7] Miin-Shen Yang and Kristina P Sinaga. A feature-reduction multi-view k-means clustering algorithm. *IEEE Access*, 7:114472–114486, 2019.

[8] Pingxin Wang, Hong Shi, Xibei Yang, and Jusheng Mi. Three-way k-means: integrating k-means and three-way decision. *International Journal of Machine Learning and Cybernetics*, 10(10):2767–2777, 2019.

[9] NVIDIA. Nvidia tesla v100 gpu architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017.

[10] NVIDIA. Nvidia tesla p100 whitepaper. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`, 2016.

[11] NVIDIA. Nvidia a100 tensor core gpu architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf`, 2020.

[12] NVIDIA. cublas. `https://developer.nvidia.com/cublas`, 2020.

[13] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. Cutlass: Fast linear algebra in cuda c++. nvidia developer blog, dec 2017.

[14] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[15] Naveen Mellempudi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. Mixed precision training with 8-bit floating point. *arXiv preprint arXiv:1905.12334*, 2019.

[16] NVIDIA. Nvidia turing gpu architecture whitepaper. `https://images.nvidia.com/aem-dam/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`, 2018.

[17] Nhut-Minh Ho and Weng-Fai Wong. Exploiting half precision arithmetic in nvidia gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[18] Cody Rivera, Jieyang Chen, Nan Xiong, Shuaiwen Leon Song, and Dingwen Tao. Ism2: Optimizing irregular-shaped matrix-matrix multiplication on gpus. *arXiv preprint arXiv:2002.03258*, 2020.

[19] Dominik Ernst, Georg Hager, Jonas Thies, and Gerhard Wellein. Performance engineering for a tall & skinny matrix multiplication kernels on gpus. In *International Conference on Parallel Processing and Applied Mathematics*, pages 505–515. Springer, 2019.

[20] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 20–24, 2020.

[21] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[22] NVIDIA. Programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[23] Gregory Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2024–2033. JMLR.org, 2016.

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