Journal of Systems Architecture 160 (2025) 103336

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

LE-GEMM: A lightweight emulation-based GEMM with precision refinement on GPU

Yu Zhang [a], Lu Lu [a,b,c,*], Zhanyu Yang [a], Zhihong Liang [d,e], Siliang Suo [d,e]

[a] School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China
[b] Peng Cheng Laboratory, Shenzhen, 518055, China
[c] Pazhou Laboratory, Guangzhou 510005, China
[d] Electric Power Research Institute, CSG, Guangzhou, Guangdong, China
[e] Guangdong Provincial Key Laboratory of Power System Network Security, Guangzhou, Guangdong, China

ARTICLE INFO

Keywords:
GEMM
Thread parallelism analytic
Multi-level pipeline
Matrix core/Tensor core

ABSTRACT

Many special hardware units, such as Matrix Core and Tensor Core, have recently been designed and applied in various scientific computing scenarios. These units support tensor-level computation with different precisions on GPU. Previous studies have proposed methods for computing single-precision GEneral Matrix Multiplication (GEMM) with the half-precision matrix. However, this routine often leads to some loss of accuracy, which limits its application. This paper proposed a Lightweight Emulation-based GEMM (LE-GEMM) on GPU that includes a lightweight emulation algorithm, a thread parallelism analytic model, and an efficient multi-level pipeline implementation to accelerate the computation process without compromising the accuracy requirements. First, we propose a lightweight emulation algorithm that includes a precision transformation process and GEMM emulation calculation to achieve better computational accuracy and performance. Secondly, a thread parallel analytic model is designed to analyze and guide the selection of the optimal tiling scheme based on various computing scenarios and hardware. Thirdly, an efficient multi-level pipeline is implemented, which can maximize instruction-level parallelism and latency hiding. Several comparison experiments were conducted on two commonly used GPU platforms: AMD-platform and NVIDIA-platform. The experimental results show that the proposed method outperforms the previous approaches in terms of computational accuracy and speed.

1. Introduction

GEneral Matrix Multiplication (GEMM) has been extensively involved in High-Performance Computing (HPC) [1], deep learning applications [2], and other scientific computing fields [3]. In these fields, some applications or scenarios in which GEMM is the primary building block are referred to as GEMM-based scientific computing [4,5]. These applications involve a large number of matrix multiplication workloads, and GEMM's computation time is a significant part of the overall process. Specifically, some typical applications, such as K-Means [6], KNN [7], Feedforward Neural Networks (FFN) [8], etc., have GEMM computation times exceeding 50% in popular implementations. To cater this trend, specialized hardware accelerators (e.g., AMD's Matrix Core [9] and NVIDIA's Tensor Core [10]) on GPU have been designed and implemented to speed up GEMM, which employs low-precision inputs to achieve high performance. In contrast to the regular computation pattern, these mixed-precision cores can support tensor-level Fused Multiply-Add (FMA), dramatically speeding up the matrix computation process. For example, the half-precision performance with Matrix Core in the AMD MI210 achieves 8× higher throughput over ROCm Core [11]. Based on this feature, previous studies have proposed a way to explore low-precision matrix to obtain high-precision results and achieve high performance on GPU [12–14].

Exploiting the low-precision to speed up the GEMM is an effective route in GEMM-based scientific computing fields, where matrix multiplication operations can be tolerated for low-precision [15,16]. The precision transformation process, which has a significant impact on the accuracy of the final calculated results, is a crucial part of this route [17]. Data value errors exist during the transition from high to low precision due to the loss in the mantissa representation. After several FMA calculations, the final result has some errors compared to the standard one. This phenomenon is also known as precision loss. This restricts the application of mixed-precision calculations significantly [18,19]. Some previous studies have utilized the calculation

* Corresponding author at: School of Computer Science and Engineering, South China University of Technology, Guangzhou 510006, China.
E-mail addresses: yuzhang0722@163.com (Y. Zhang), lul@scut.edu.cn (L. Lu), yangzhanyu@hotmail.com (Z. Yang), liangzh@csg.cn (Z. Liang), suosl@csg.cn (S. Suo).

https://doi.org/10.1016/j.sysarc.2025.103336
Received 11 July 2024; Received in revised form 8 December 2024; Accepted 2 January 2025
Available online 17 January 2025

results of additional low-precision variables to reduce precision loss in the calculation results [20–23].

Low-precision variables and additional computations are utilized for *precision refinement* [10], which reduces the impact of precision loss on the computational results. The implementation of this approach on GPUs that feature mixed-precision arithmetic is still hindered by several issues. **Primarily, a crucial factor is the data transformation process**. Loss of the original data mantissa during the precision transformation process has an impact on the final outcome. As a consequence of the disparity in the mantissa length, low-precision data cannot be saved beyond the mantissa length range. This leads to numerical errors after the precision transformation process. The effect of this error will be amplified into the final calculation result after multiple FMA operations, which is a gap between calculated result and the standard result. **The second is the acceleration capabilities of these specialized hardware units.** These specialized hardware units provide GEMM with powerful acceleration capabilities, accompanied by special data layouts, which pose challenges to efficient kernel design. For GEMM operation, tiling is a common way of assigning computational tasks, where the matrix is segmented into multiple tiles (smaller matrix), and a workgroup (thread block) is responsible for computing one or several tiles. When the tile size is too large, the workload of the threads will become too heavy, reducing thread parallelism. Conversely, if the tile size is too small, there will be frequent workgroup switching, which will increase scheduling overhead. The computation allocation of tiles also affects data reuse in a workgroup. In terms of data access, these specialized hardware units perform GEMM operations with fragments, which requires higher utilization of the GPU's multi-level memory. Ineffective organization and manipulation of data frequently negate the speed benefits of the tensor-level patterns, leading to memory wall bottlenecks in the GPU.

To address the above issues, we propose LE-GEMM that includes a lightweight emulation algorithm, a thread parallelism analytic model, and a multi-level pipeline implementation to achieve better computational accuracy and speed. Our contributions can be summarized as follow:

- A lightweight emulation algorithm is proposed that employs bit-cutting and scaling techniques to preserve the original values and reduce the loss of precision while removing redundant terms in the computation process to reduce the computational overhead.
- A thread parallelism analytic model based on the lightweight emulation GEMM is proposed, which allows the selection of the optimal tiling scheme through two aspects: the number of parallel threads and the single thread performance.
- An efficient multi-level pipeline is implemented based on the access discrepancy between various levels of memory and the GEMM workflow, utilizing double buffer and pre-fetch techniques to hide access latency and improve Instruction-Level Parallelism (ILP).
- The proposed single-precision GEMM accelerated method achieves state-of-the-art speed and accuracy performance on two GPU platforms.

The rest of the paper is organized as follows. Section 2 introduces the background of these specialized hardware units (Matrix Core and Tensor Core) and matrix multiplication based on the Ozaki scheme. Section 3 provides some related work. Section 4 presents the details of the proposed accelerated method for single-precision GEMM. Section 5 demonstrates and evaluates the experimental results. The conclusions of the paper are given in Section 6. Section 7 outlines potential areas for future work that can be further explored.

2. Background

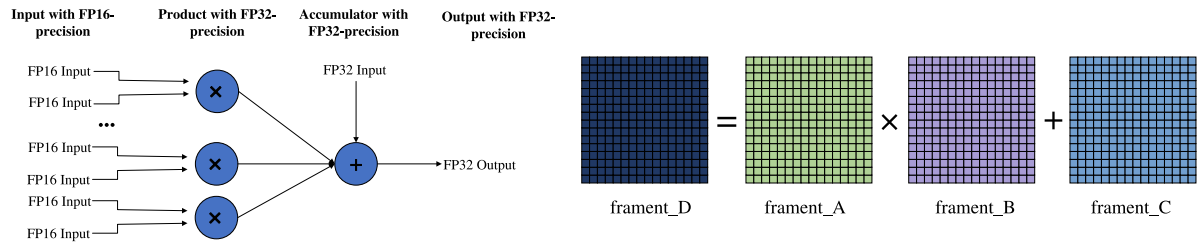
2.1. Matrix core and tensor core

The amount of computation for GEMM has gradually increased with the development of deep learning. In response to this trend, AMD and NVIDIA are adding specialized hardware units and cores to their latest GPUs that support mixed-precision computing [24,25]. In the following, we will discuss the details of the specialized cores.

Computing Pattern and Memory Layout. The GPU cores can be classified into two types based on supported precision: same-precision cores (FP16 Core, FP32 Core, etc.) that only support the same precision computation and mixed-precision cores (Tensor Core or Matrix Core) that support different precision computation [26,27]. The GEMM supported by the mixed-precision core can be expressed as $D = \alpha \times AB + \beta \times C$, where A and B can be FP16 or FP32 matrix, C and D can be configured to FP32 or double precision (FP64) matrix, and α and β denote scalar constants [28]. This means that the mixed-precision core can utilize the input of low-precision to obtain high-precision calculation results [10,29]. The details are shown in Fig. 1(a). In terms of computing patterns, the mixed-precision core utilizes a new memory layout *fragment* that stores matrix elements either in row-major or column-major order. Different from the same-precision core matrix multiplication operation, the matrix multiplication operation is performed in the mixed-precision core using the *fragment* as a unit. The fragment size is "16 × 16" or "32 × 32", and other fixed shapes (Fig. 1(b) provides an example based on a "16 × 16" fragment). Note that different hardware architectures have varying support for fragment size. For example, for NVIDIA A800, the maximum supported fragment size is "16 × 16" [10]. In AMD MI210, rocWMMA can support fragments with a size of "16 × 32" or "32 × 32" [11]. Different fragment sizes have a particular impact on the workload allocation of GEMM operations.

Programming interface and implementation. Matrix Core and Tensor Core are hardware cores delivered by two different GPU vendors (AMD and NVIDIA). GPU vendors have implemented various approaches depending on their hardware configurations to facilitate their use by researchers and engineers. On the AMD platform, the first approach is rocWMMA (ROCm Wavefront-level Matrix Multiply and Accumulate), in which AMD's C++ interface leverages Matrix Core to accelerate GEMM operations. The usage and functionality of this interface are very similar to that of WMMA [28]. The difference between the two ways is that rocWMMA calls the API handle with the keyword *rocwmma*. The second approach is the compiler instruction [9], which follows a specific syntax "*__builtin_amdgcn_wmma_*". The third approach uses rocBLAS [30], which provides a functional interface and allows the direct operation of a mixed-precision core by supplying the matrix parameter.

There are multiple ways to call Tensor Core on the NVIDIA platform, such as WMMA (Warp-level Matrix Multiply and Accumulate) [10], CUTLASS [31], cuBLAS [32], PTX instruction [33], etc. WMMA provides several specific programming interfaces to load (*wmma::load_matrix_sync*), compute (*wmma::mma_sync*), and store (*wmma::store_matrix_sync*) matrices in fragments [10]. Based on this interface, the matrix is first tiled into multiple fragments, which can be shaped to fixed sizes, and then matrix multiplication is performed. CUTLASS (CUDA Templates for Linear Algebra Subroutines) [31] is a series of CUDA template header libraries for implementing high-performance GEMM and related computations at all scales and levels within CUDA. It provides a variety of GEMM computation interfaces (*hgemm*, *sgemm*, *dgemm*, and *wgemm*) to optimize data movement and multiply-accumulate operations in fragments to achieve high performance. cuBLAS [32] is a basic linear algebra subroutines library provided and implemented by NVIDIA. cuBLAS provides two interfaces (*cublasGemmEx* and *cublasSgemm*) that support GEMM on Tensor Core. In this approach, it is necessary to set the math model of cuBLAS by using the *cublasSetMathMode* function. PTX instruction provides a series of programming APIs and instruction sets for general-purpose parallel programming. It is designed to be efficient on NVIDIA GPUs that support the defined computation functions. The terminological differences and relationships between ROCm and CUDA are also given in Table 1.



(a) Precision profiling (b) Computing pattern with a 16×16 fragment

Fig. 1. The precision profiling and the computing pattern of mixed-precision units (Fig. 1(a) shows an example of precision profiling, where input data is FP16-precision, product and accumulator is FP32-precision, and output is FP32-precision. Fig. 1(b) illustrates a fragment-based GEMM operation with a "16 × 16" fragment).

Table 1
ROCm/CUDA terminology.

ROCm	CUDA	Description
Compute Unit (CU)	Streaming Multiprocessor (SM)	One among many parallel vector processors in a GPU with parallel ALUs. All wavefronts in a workgroup are assigned to the same CU.
Kernel	Kernel	Functions initiated to the GPU that are executed by multiple parallel CUs or SMs on the GPU. Multiple kernels can work in parallel with CPU.
Wavefront	Warp	A collection of operations that executed in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked.
Workgroup	Thread block	Think of this as a vector thread. A wavefront is a 64-wide vector op in AMD GPU. A warp is a 32-wide vector op in NVIDIA GPU.
Work-item/Thread	Thread	GPU programming models can handle this as a separate thread of execution, although not necessarily get forward sub-wavefront progress.
Global Memory	Global Memory	The DRAM memory is accessed by a GPU that traverses the cache of some layers.
Local Memory	Shared Memory	Scratchpad that allows communication between wavefronts within a workgroup.
Private Memory	Local Memory	Per-thread private memory is often mapped to registers.

2.2. Matrix multiplication based on the Ozaki scheme

The Ozaki scheme is an error-free method for computing matrix multiplication [34]. It involves translating the original matrix multiplication into a sum of matrices that can be multiplied together. Fig. 2 is a schematic of the matrix multiplication based on the Ozaki scheme. The detailed procedure can be divided into three steps: splitting, computation, and summation. In the splitting step, matrix A and matrix B are first divided by element level to obtain multiple submatrix (A_i, B_j). Then, in the computation step, matrix multiplication is performed on the submatrix of A (A_i) and the submatrix of B (B_j) to obtain the submatrix of C ($C_{i,j}$). Finally, all these submatrices are summed to obtain the result matrix C.

Based on the Ozaki scheme, a single precision matrix multiplication calculation method can be described as follows. For simplicity, the matrix multiplication with FP32 is used as an example. Firstly, in the element-level splitting step, the matrix based on FP32 is converted to FP16 precision by the precision transformation function, while the error is preserved. The detailed expression is as follows:

$$A_{FP16} \leftarrow \text{toFP16}(A_{FP32}) \tag{1}$$

$$R_A \leftarrow \text{toFP16}(A_{FP32} - A_{FP16}) \tag{2}$$

where A_{FP32} and A_{FP16} denote the FP32-based and FP16-based matrix A, respectively. toFP16() is the transformation function that transforms matrix A from FP32 to FP16, and R_A represents the residual matrix between different precision. Note that this expression is analogous to matrix B. With the support of mixed-precision cores for floating-point arithmetic with different precision, $A_{FP32}B_{FP16}$ can be rewritten as follows:

$$A_{FP32}B_{FP16} = (A_{FP16} + R_A)B_{FP16} = A_{FP16}B_{FP16} + R_A B_{FP16} \tag{3}$$

Hence, the matrix multiplication for $A_{FP32}B_{FP32}$ can be expressed as:

$$\begin{aligned} A_{FP32}B_{FP32} &= (A_{FP16} + R_A)(B_{FP16} + R_B) \\ &= A_{FP16}B_{FP16} + A_{FP16}R_B + R_A B_{FP16} + R_A R_B \end{aligned} \tag{4}$$

where A_{FP16}, B_{FP16}, R_A, and R_B denote the matrix with FP16, A_{FP32} and B_{FP32} denote the matrix with FP32.

3. Related work

The advent of Matrix Core and Tensor Core has greatly accelerated the computational speed of GEMM due to its specific data layout and precision requirements. This feature allows it to be studied and applied in many GEMM-based scientific computing fields [24–27]. Therefore, previous studies have optimized the scheduling methods of these special hardware units based on different application scenarios to accelerate the computation process [4,12,15,35]. For example, a novel approach for accelerated quantization of graph neural networks, based on the support of low-precision GEMM, was proposed by Wang et al. [36], which exploits the insensitivity of graph neural networks to loss of precision. To make Tensor Core be executed more efficiently in deep learning applications, Dakkak et al. [26] have optimized the reduction and the scan terms for matrix multiplication, making their programming faster and more accessible. Specifically, Dakkak's method [26] was implemented on the Tesla V100 and achieved 89%–98% of peak memory bandwidth and reduced power consumption. Li et al. [37] proposed a half-precision Fast Fourier Transform (FFT) approach based on Tensor Core. This approach leverages the fragments in Tensor Core to support the special operations required for FFT and optimizes the data arrangement in GPU memory [38]. To further explore these hardware units' workflow and application prospects, Sakamoto et al. [39] improve energy efficiency by analyzing memory and power consumption of low-precision data during computation. The above studies aim to optimize the use of these specialized hardware units to accelerate the computing process in specific scientific computing scenarios based on their computational characteristics [40,41]. These approaches maximize the acceleration capability of matrix multiplication operation without changing the internal computation process of specialized hardware units. However, FMA operations between different precisions will inevitably result in a certain precision loss in the calculation results, which will limit the application scope of these

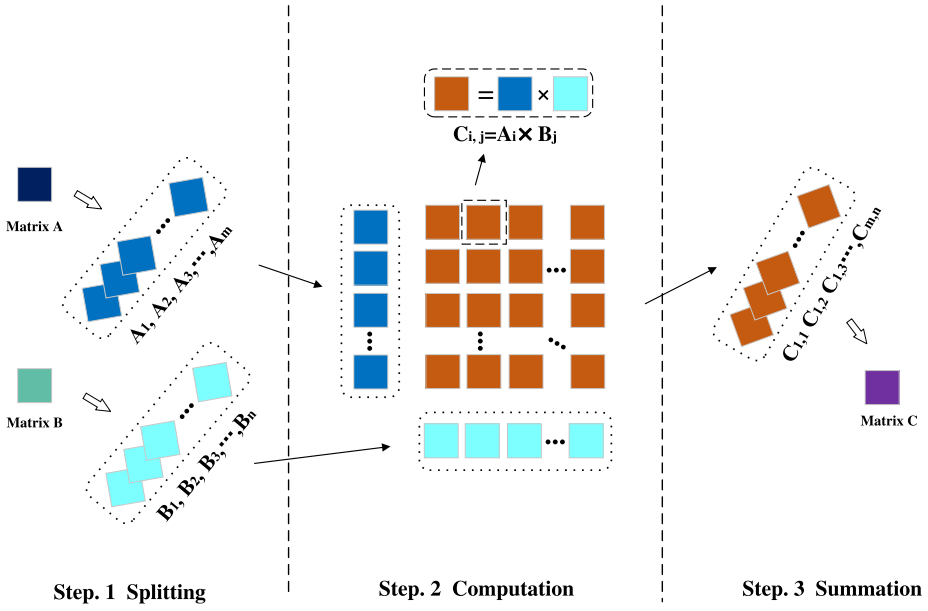


Fig. 2. The schematic of the matrix multiplication based on the Ozaki scheme ($C = AB$ as an example).

hardware units [21,22,42].

To mitigate the loss of precision, several improvement approaches have been proposed. Markidis et al. [10] analyzed the programmability and precision of Tensor Core. Then they proposed a precision refinement method for single-precision (FP32) matrix multiplication, which takes advantage of half-precision (FP16) to store residuals in transformations. Based on this research, Ootomo et al. [43] improved the rounding method of single-precision floating-point numbers. They reduced the redundancy in error-correction terms by considering the effect of the rounding method and implicit bit. Others have studied the error analysis of matrix multiplication using Tensor Core and have given theoretical error bounds for matrix multiplication with different precisions [24,44,45]. To explore the efficiency of mixed-precision cores, Jia et al. [46] studied the arrangement of matrix-to-fragment data in memory and how assembly instructions split the matrix. To support the extended-precision computing pattern, several methods have been proposed in limited-precision hardware, combining multiple low-precision computation instructions to simulate the extended-precision computing methods [47–49]. Inspired by this approach, Feng et al. [4] improved the Markidis's method in single-precision matrix multiplication, which modified the rounding method and optimized the computation of extended-precision at the instruction level. In addition, Fasi et al. [21] invested in the application of multiword methodology to improve the performance-accuracy trade-off of matrix multiplication with mixed-precision cores, focusing specifically on the Tensor Core available on NVIDIA GPUs. Following this research direction, Parikh et al. [20] presented insights and possibilities for implementing higher precision matrix multiplication with Tensor Core, and an implementation of the FP64x2 GEMM scheme with variable FP64 inputs was provided. The common feature of these studies is using low-precision inputs to obtain high-precision computational results with some additional computation workload. In this way, these specialized hardware units (Matrix Core and Tensor Core) can perform high-performance matrix multiplication operations and reduce the impact of precision loss [44,50].

4. LE-GEMM

This section gives a detailed description of LE-GEMM, which includes the lightweight emulation algorithm, the thread parallelism analysis model, and the efficient multi-level pipeline design. Firstly, we introduced the lightweight emulation algorithm, which can provide a more efficient and accurate calculation performance based on specialized hardware units. Then, a thread parallelism analytic model is proposed, which is used to analyze and select the optimal tiling scheme based on various computing scenarios and hardware. Finally, we present an efficient multi-level pipeline design for GEMM, which utilizes double buffer and pre-fetch techniques to hide access latency and improve ILP.

4.1. Lightweight emulation algorithm

Traditional emulation algorithm design assumes that the hardware has the same input and output precision. Typically, a large number of low-precision instructions are used to complete the computation and memory access process, which can result in high computational overhead and reduce performance benefits [4,40,51,52]. However, specialized hardware units, such as Matrix Core and Tensor core, have been introduced to accelerate the GEMM operation. Multiplication and addition are fused in special instruction primitives, and the process of accumulating intermediate results also supports high-precision accumulation to ensure accurate calculation results. This new hardware feature has motivated us to design a lightweight emulation algorithm that accelerates the GEMM computation process by achieving high computational throughput based on low-precision inputs. The lightweight emulation algorithm consists of the precision transformation process and the GEMM emulation calculation, as illustrated in Fig. 3. The precision transformation process is designed to make the low-precision data better preserve the value of the original data, thus achieving more accurate calculation results. The GEMM emulation calculation is designed to achieve high-precision results and speed up the calculation process by leveraging the specialized hardware units and low-precision input. Floating-point numbers of different precision types have exponent and mantissa with different lengths. For the single-precision format, 8 bits are allotted to the exponent and 23 bits to the mantissa. The half-precision representation has a smaller range than the single-precision, with the exponent and mantissa being 5 and 10 bits, respectively. In emulation algorithm, the first step is to split the original matrix, which affects the level of elements in the matrix. There are two different splitting routines. One splitting routine is to split the matrix elements without changing the precision, and the other is to split the matrix

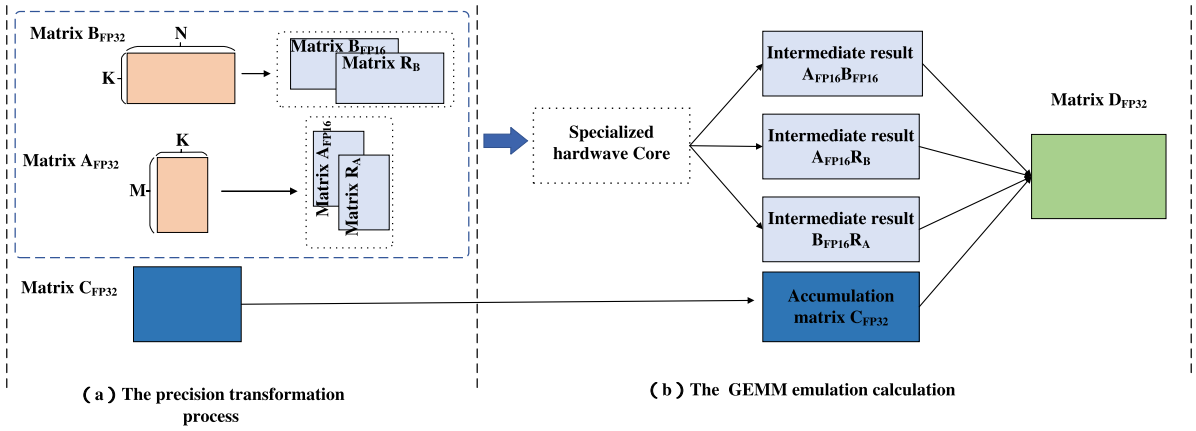


Fig. 3. The workflow of LE-GEMM.

values and change the precision type. In this paper, we focus on the implementation of the second routine. However, the transformation process inevitably leads to precision loss. Therefore, an additional FP16 is used to preserve the residual and improve the accuracy of the calculated results. Rounding is a popular precision transformation technique that has been previously employed to transform from high-precision to low-precision [20,45]. The common rounding methods include RN, RZ, RU, and RD. [1] The rounding method has the advantage of taking into account the representation of the mantissa low-bit part, which is after the rounding bit. However, it is important to note that each rounding method will inevitably introduce some errors in the final value. The primary reason for this is that the mantissa involves rounding up or down, leading to numerical deviations. For instance, when transforming from FP32 to FP16 using the RN method, the final value in the 13-bit mantissa considers the low-bit of the mantissa (0–12 bit) to achieve the nearest round. A_{FP16} and R_A are responsible for preserving the high-bit and low-bit mantissa of A_{FP32} respectively. From a numerical perspective, A_{FP16} has a larger effect than R_A on computation results. Therefore, it is important to preserve a high-bit of the mantissa during the rounding process by saving as many low-bit bits as possible.

Compared to previous work, we adopted a bit-cut to preserve the high-bit bits in FP32 during the precision transformation process, considering the impact of high-bit and low-bit bits on errors. To achieve precision transformation from A_{FP32} to A_{FP16}, a bit-cut method is employed to preserve the A_{FP32} mantissa high-bit. In this way, the higher mantissa (13–22 bits of A_{FP32} in Fig. 4) is copied into A_{FP16}. For residual R_A, introducing a scaling operation requires dividing it by a scaling factor (2^{10} in Fig. 4) in the calculation process to ensure the correct results. Note that due to the difference in the length of the exponent part between single-precision and half-precision, additional processing is required to avoid errors in the transform process of the exponent part. Considering that the bias of single-precision and half-precision sizes are 127 and 15, it is necessary to subtract the bias of the exponent part to avoid errors in the transformation exponent bit. To avoid the impact of bias, we preserve the exponent part to an intermediate variable, which requires subtracting the bias value. Then, add the corresponding bias when saving the intermediate variable for the half-precision exponent part. This approach offers the following benefits: compared to traditional rounding, this approach does not generate rounding-up or rounding-down, which can affect the stability of high-bit. This approach does not depend on the rounding API interface of the platform, which differs in the rounding regulations of different platforms. The bit-cut is based on a binary format for floating point numbers, and it is compatible across platforms. Fig. 4 illustrates the details of the precision transformation process. For the computation of R_A, since the mantissa of FP16 is of length 10, this does not fully save all the remaining mantissa. Based on these two factors, a residual is calculated from A_{FP32} to A_{FP16}. Then multiply the residual by the scaling factor to preserve the residual, which is 2^{10}. This does not alter the representation, but it maps the initial value to the larger range that can be represented by R_A. Note that since R_A was obtained by scaling, the processing used in subsequent calculations will return the original residual value. Finally, to preserve the mantissa to the maximum extent, the residuals are saved using rounding. The precision transformation process can be rewritten as follows:

$$A_{FP16} = \text{bit-cut}(A_{FP32}) \tag{5}$$

$$R_A = \text{round}((A_{FP32} - A_{FP16}) \times 2^{10}) \tag{6}$$

$$B_{FP16} = \text{bit-cut}(B_{FP32}) \tag{7}$$

$$R_B = \text{round}((B_{FP32} - B_{FP16}) \times 2^{10}) \tag{8}$$

where bit-cut() denotes executing bit-cut operation, round() represents RZ. This section presents the details of the precision transformation process involving bit-cut and scaling. In the scaling step, it is essential to divide the value by the scaling factor in the last step of the calculation to cancel the effect of the scaling factor. Therefore, the GEMM emulation calculation, which is based on the previous precision transformation process, can be described as follows.

$$\begin{aligned} A_{FP32}B_{FP32} &= (A_{FP16} + R_A/2^{10})(B_{FP16} + R_B/2^{10}) \\ &= A_{FP16}B_{FP16} + (A_{FP16}R_B + R_AB_{FP16})/2^{10} + R_AR_B/2^{20} \end{aligned} \tag{9}$$

The fourth calculation term in Eq. (9) is divided by 2^{20}, thus it has little effect on the final computation result. Moreover, additional computation items will increase the overall computation effort. For this purpose, Eq. (9) is modified as follows:

$$\begin{aligned} A_{FP32}B_{FP32} &= (A_{FP16} + R_A/2^{10})(B_{FP16} + R_B/2^{10}) \\ &\approx A_{FP16}B_{FP16} + (A_{FP16}R_B + R_AB_{FP16})/2^{10} \end{aligned} \tag{10}$$

Hence, the lightweight emulation algorithm calculation can be described as follows:

$$\begin{aligned} D_{FP32} &= \alpha \times A_{FP32}B_{FP32} + \beta \times C_{FP32} \\ &\approx \alpha \times (A_{FP16}B_{FP16} + (A_{FP16}R_B + R_AB_{FP16})/2^{10}) + \beta \times C_{FP32} \end{aligned} \tag{11}$$

[1] RN, RZ, RU, and RD represent Round-to-nearest-even, Round-towards-Zero, Round-Up, and Round-Down, respectively.

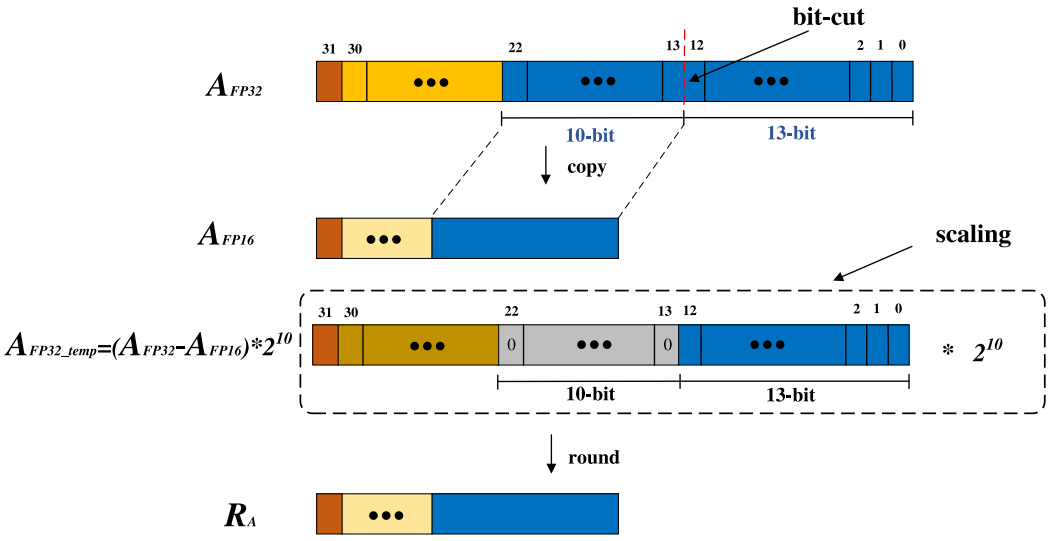


Fig. 4. The bit-cut and scaling operation.

where D_{FP32} and C_{FP32} denote FP32 matrices, A_{FP16}, R_A, B_{FP16} and R_B represent the same meaning as those in Eq. (10), α and β are scalar constants. This section introduces a precision transformation process consisting of a two-step bit-cut and rescaling. This process can effectively preserve the original data mantissa and avoid numerical fluctuations caused by rounding. At the same time, based on the above precision transformation process, we have designed a GEMM emulation calculation process, which removes the cumulative term in Eq. (9) and reduces the overall computation time of GEMM. Algorithm 1 provides an overview of the lightweight emulation algorithm. In Algorithm 1, bit-cut () denotes utilizing the bit-cut technique to preserve the high-bit bits in the mantissa, which is detailed in Section 4.1. round() represents the use of rounding to preserve the low-bit bits in the mantissa. mFma_sync() represents utilizing specialized hardware units to calculate the multiplication of two matrices, and uniformFma_sync() represents the computation process of Eq. (11) to obtain the final calculation result D_{FP32}.

Algorithm 1 The lightweight emulation algorithm

Function emulation GEMM operation(D_{FP32}, A_{FP32}, B_{FP32}, C_{FP32},α, β)
  /* The precision transformation process */
    A_{FP16}=bit-cut(A_{FP32}), R_A=round((A_{FP32}-A_{FP16})$\times 2^{10}$);
    B_{FP16}=bit-cut(B_{FP32}), R_B=round((B_{FP32}-B_{FP16})$\times 2^{10}$);
  /* GEMM emulation computation process */
    /* Initialize accumulator */
      Initialize fragsAcc0, fragsAcc1;
    /* Calculate each term in Equation (11) */
      fragsAcc0 = mFma_sync (A_{FP16}, B_{FP16}, fragsAcc0);
      fragsAcc1 = mFma_sync (A_{FP16}, R_B, fragsAcc1) ;
      fragsAcc1 = mFma_sync (B_{FP16}, R_A, fragsAcc1) ;
    /* Calculate the final result */
      D_{FP32} = uniformFma_sync (α, fragsAcc0, fragsAcc1/2^{10}, β, C_{FP32});
end Function

4.2. Thread parallelism analytic model

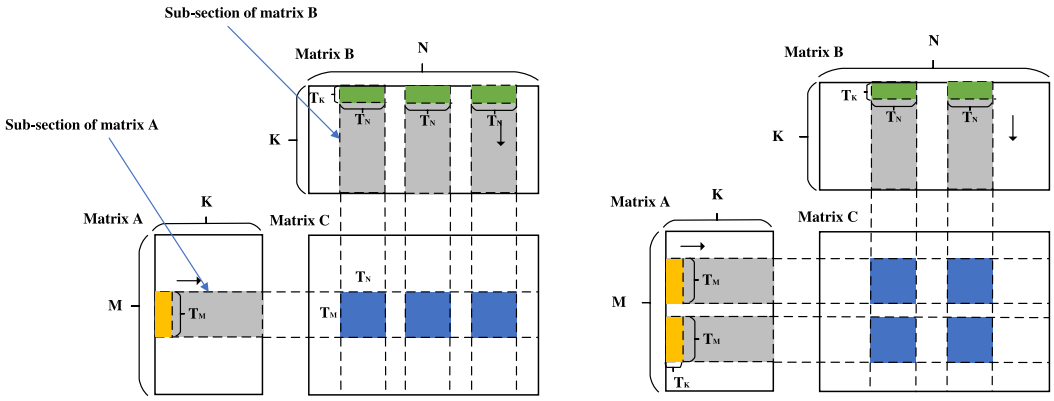
The lightweight emulation algorithm is designed to efficiently map GEMM-based scientific computing to specialized hardware cores, allowing for high-precision outputs with low-precision inputs. To fully exploit the acceleration capabilities of specialized hardware units, an efficient tiling scheme that represents the impact of the allocation of computational tasks on overall performance is critical. The tiling scheme is to segment the matrix into multiple tiles and assign each tile to a cooperative thread array (thread block) responsible for the computation. Each thread loads and completes the computation of the corresponding matrix elements. This process is based on the size of the matrix in GEMM operation and the hardware resources to perform the allocation of computational tasks.

Fig. 5 illustrates that computing a tile in matrix D requires loading sub-sections of matrices A and B (the gray area) and a tile in matrix C (the blue area) to perform GEMM operation. This allows for the computation of multiple tiles simultaneously on the GPU. To fully exploit the computing power of GPUs, our proposed thread parallelism analysis model analyses and improves the performance of GEMM computations by considering the number of parallel threads (Thread-Level Parallelism, TLP) and the single thread performance. Given the tile size, the computation of GEMM can be divided into multiple subparts, each with a set of threads responsible for the computation. This property allows for the computation of thread parallelism based on a given tiling strategy. Previous methods have been proposed based on the number of threads in a thread block simply. Because the thread block is further transformed into multiple frontwaves (warps) for execution on CU, this resulted in the previous method ignoring the underlying computational pattern on the GPU. Therefore, we use the number of wavefronts to calculate thread parallelism. This provides a more accurate indication of how active the underlying threads are in practice. Based on the matrix's tiling scheme and hardware configuration, TLP can be calculated as follows:

$$T_{TLP} = \varphi\left(\frac{M \times N}{T_M \times T_N}\right) \times T_{thread\_num} \tag{12}$$

where M and N represent the dimensions of matrix C, T_M and T_N are the tile size, φ is the transform process of workgroup to wavefront, and T_{thread_num} represents the number of threads in wavefront. Note that T_{thread_num} has different values across various GPU platforms, for example, on MI210 and A800, the values for T_{thread_num} are 64 and 32, respectively.

When considering TLP, it is important to consider the performance of a single thread, which can be expressed as a compute-to-memory ratio. Comparing the clock cycles of memory access instruction and computation instruction in threads, it is evident that data access in memory requires more time. Therefore, it is crucial to ensure that the data loaded on the thread is fully utilized. During each iteration, every thread block is required to perform two tasks. The first step involves



(a) Tiling matrix with low-reuse

(b) Tiling matrix with high-reuse

Fig. 5. The tiling scheme of a single GEMM.

reading two matrices (A_{FP16}, R_A) of size (T_M, T_K), two matrices (B_{FP16}, R_B) of size (T_K, T_N) and matrix C of size (T_M, T_N). Since matrix C of size (T_M, T_N) is loaded only in the first iteration and does not need to be loaded again in subsequent steps, its loading overhead can be neglected. Thus, the memory access instruction with each iteration can be expressed as:

$$T_{memory} = 2 \times 2 \times (T_M \times T_K + T_K \times T_N) \tag{13}$$

where T_{memory} represents memory access during each iteration, the last 2 means that the half-precision data occupies two bytes. According to the lightweight emulation algorithm, the computation instruction required for each iteration process is as follows:

$$T_{flops} = 3 \times 2 \times (T_M \times T_N \times T_K) \tag{14}$$

where T_{flops} represents computation with each iteration, $2 \times (T_M \times T_N \times T_K)$ represents the computation amount of matrix multiplication between matrix of size (T_M, T_K) and matrix of size (T_K, T_N), and the constant 3 corresponds to the three terms in Eq. (11). To this end, the ratio of computation to memory access ($T_{\frac{flops}{memory}}$) during each iteration is as follows:

$$T_{\frac{flops}{memory}} = \frac{3 \times 2 \times (T_M \times T_N \times T_K)}{2 \times 2 \times (T_M \times T_K + T_K \times T_N)} = \frac{3(T_M \times T_N)}{2(T_M + T_N)} \tag{15}$$

While pursuing higher TLP and compute-to-memory ratios, it is also important to focus on thread workload. The workload of a thread can be divided into two parts: the amount of data load and the amount of computation. To avoid excessive workload and reduce thread parallelism, it is necessary to properly set the tiling scheme and the number of tiles that the workgroup is responsible for computing. To describe a thread the amount of data load ($Work\_DL$) and the amount of computation ($Work\_CP$) more accurately, the following two formulas are given:

$$Work\_DL = \frac{T_M \times T_N + T_M \times T_K + T_K \times T_N}{Work\_num} \tag{16}$$

$$Work\_CP = \frac{T_M \times T_N}{Work\_num} \tag{17}$$

where $Work\_num$ represents the number of thread responsible for computing the tile. T_M, T_N, and T_K represent tile size, the details of which are shown in Fig. 5. $Work\_DL$ is the number of elements responsible for the computation in each thread. To ensure efficient parallelism, it is necessary to allocate a different number of threads to various shape tiles. This ensures that the workload is consistent between thread blocks. $Work\_CP$ represents the number of registers loaded by each thread. When the thread loads too many elements, it will cause the data in the register to overflow, which will increase the data access overhead. At the same time, we should also consider the correspondence between the thread block and the matrix tiles, which will also affect the efficiency of data reuse. As shown in Fig. 5(a), the calculation of 3 tiles in matrix C involves the loading of 1 sub-section of matrix A, and 3 sub-sections of matrix B, respectively. In Fig. 5(b), 4 tiles can be computed by loading two sub-sections of the matrix A and B. Comparing these two routines, it is clear that the larger the cross-sectional area (blue tile in Fig. 5) between the sub-sections loaded by matrices A and B. Fig. 5 indicates that the pattern in (b) has better data reuse.

Finally, the process of optimal tiling scheme selection with thread parallelism analytic model is formally represented as maximizing T_{TLP} and $T_{\frac{flops}{memory}}$. According to Eqs. (12) and (15), it can be found that T_{TLP} and $T_{\frac{flops}{memory}}$ show an inverse relationship given M, N, and T_{thread}. Generally, smaller tiles can achieve better thread parallelism because computational tasks are assigned to multiple threads. However, the number of activated threads is also limited by the supported number by the GPU hardware. When the tile is larger, a larger compute-to-memory ratio is often achieved, which represents a better performance for a single thread. When choosing a tiling scheme, it is also necessary to consider the workload of the thread according to Eqs. (16) and (17). Eqs. (12)–(17), take into account factors that affect the number of active threads and the performance of a single thread, allowing the optimal tiling scheme to be selected based on the various GPU platforms and computing scenarios. Our thread parallelism analysis model chooses the tiling scheme based on thread-level parallelism and single thread performance, which can be solved analytically with existing optimization solvers [53,54]. This can also be used to provide a more profitable direction when selecting the best parameters by trial-and-error. Table 2 presents the choice of the parameters for LE-GEMM. In Table 2, (T_M, T_N, T_K) denotes the size of the matrix partitioned into tiles, (f_m, f_n, f_k) denotes the fragment size for GEMM computation, and wavefront/warp size denotes the number of threads in the thread group of the two GPU platforms that is determined by their hardware architectures. Due to the various sizes and shapes of the fragment supported by different architectures, the choice of the fragment is also limited to the two GPU platforms. At the same time, the wavefront/warp size also affects the thread workload allocation. Therefore, when choosing parameters, it is essential to consider the impact of hardware differences on performance.

4.3. An efficient multi-level pipeline for GEMM

In the GPU memory architecture, various memory spaces will have different sizes and access speeds. Data access speeds can be classified from fast to slow in terms of register memory, local memory, and global memory. The speed is inversely proportional to the size of the memory

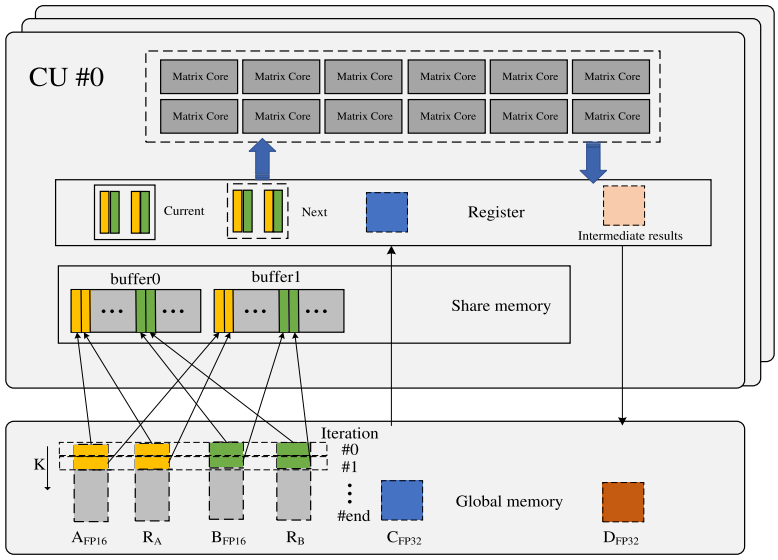


Fig. 6. The workflow of GEMM.

Table 2
The choice of the parameters on two GPUs.

Name	AMD MI210	NVIDIA A800
Matrix tile size (T_M, T_N, T_K)	(128, 128, 32)	(128, 256, 32)
Fragment size (f_m, f_n, f_k)	(32, 32, 16)	(16, 16, 16)
Wavefront/Warp size	64	32

Table 3
The configuration of platforms for evaluation.

Name	AMD-platform	NVIDIA-platform
CPU	EPYC 7663	Platinum 8358
GPU	MI210	A800
OS	Ubuntu 20.04	Ubuntu 20.04
ROCm/CUDA	ROCm 5.6	CUDA 12.0

space. Direct access to data from global registers reduces transfer efficiency and does not maximize memory bandwidth. Therefore, in this paper, double buffers are used to alleviate the problem of high differences in memory access speeds, and pre-fetch techniques are used to maximize memory bandwidth and thread instruction parallelism. The workflow for a tile in GEMM is shown in Fig. 6.

First, a double buffer (buffer0 and buffer1) was created in the share memory, and the single buffer size is related to the cumulative span of matrices A and B along the K-dimension. As shown in Fig. 6, loading instructions is applied to fetch matrix tiles A_{FP16}, R_A, B_{FP16} and R_B in iteration #0, and the data is stored in buffer0. The fetch data correspond to the yellow tiles A_{FP16} and R_A ($T_M \times T_K$) and the green tiles B_{FP16} and R_B ($T_K \times T_N$) in Fig. 6. At the same time, the data to be loaded in iteration #1 will also be fetched into buffer1. This allows the register to fetch data directly in the share memory, without waiting for the data to be loaded from the global memory. The thread can load data directly from shared memory into private registers, follow a specified data layout, and then utilize Matrix Core to complete the computation. When the intermediate result of the first iteration is complete, the register reads the data needed for the next iteration from buffer1. The thread continues to load the data needed for the next iteration from buffer1 when the intermediate result of the iteration #0 is complete. For efficient data pipelining and instruction-level parallelism, data for the next iteration can be pre-fetched from global memory into shared memory when performing data transfers between shared memory and registers. Based on the above approach, we can implement a multi-level access pipeline between different memory levels, and the access and computation instructions can be executed in parallel, improving the ILP performance of the thread. Note that a thread synchronization instruction is performed behind the store to ensure data consistency by each time a buffer is stored. Each iterative computation produces an intermediate result which is combined with the accumulation matrix C_{FP32} for FMA operation. After several computation iterations, the final calculation result D_{FP32} can be obtained and written back to the global memory.

The implementation of the multi-stage pipeline above must be aware of the size of registers and shared memory supported by the specific GPU hardware. Once the maximum capacity limit is exceeded, data overflow will occur and this will greatly increase access latency. Indeed, register and shared memory occupancy is related to T_M, T_N and T_K from the above description. When a tiling scheme is given, T_M and T_N are also determined. Therefore, data overflow can be prevented by setting T_K flexibly. For example, on the AMD platform, based on the fact that T_M and T_N are 128 and 256 respectively, T_K can be set to a smaller size such as 16 or 32 to prevent data overflow. Note that to avoid invalid computations in Matrix Core, T_K should preferably be an integer multiple of the fragment size.

5. Results

This section compares the proposed method with some state-of-the-art approaches and analyzes the experimental results on two GPU platforms. The flowchart of the comparative experiment is shown in Fig. 7. Note that the matrix size of $A \in R^{M \times K}$ and $B \in R^{K \times N}$ is represented as "$M*N*K$" in the following expressions for simplicity. The tested range of numbers was [−1,1] in comparative experiments.

5.1. Setup

Experiment platform. In this paper, we chose two most common and advanced GPUs (MI210 and A800). The configuration and details of the experimental platform are given in Tables 3 and 4 respectively.

Comparison method. To evaluate the proposed method, first, for two GPU platforms, the default single-precision GEMM methods such as rocBLAS and cuBLAS, are provided by the respective GPU vendors. These methods do not use the mixed-precision core and directly utilize the FP32 input to compute the FP32 GEMM results. We compare some methods, such as rocWMMA and WMMA, that use FP16 input to compute FP32 results on Matrix Core or Tensor Core without precision

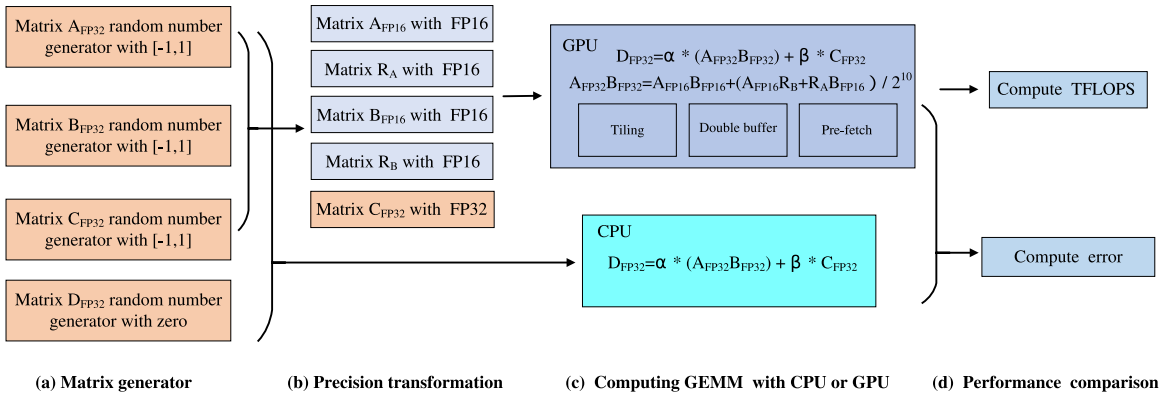


Fig. 7. The flowchart of the comparative experiment.

Table 4
The configuration of two GPUs.

Name	AMD MI210	NVIDIA A800
Architecture	CDNA 2.0	Ampere
Mixed-precision core	Matrix Core	Tensor Core
Caches	L1 16 KB(per CU) L2 16 MB	L1 192 KB (per SM) L2 40 MB
Memory	64 GB HBM2	80 GB HBM2
Bandwidth	1.6 TB/s	2.04 TB/s

Table 5
The description of the baseline method.

Name	Input	Output	Description
rocBLAS [30]	FP32	FP32	*rocblasSgemm* on ROCm Core.
cuBLAS [32]	FP32	FP32	*cublasSgemm* on CUDA Core.
rocWMMA [28]	FP16	FP32	rocWMMA implemented on Matrix Core without precision refinement.
WMMA [55]	FP16	FP32	WMMA implemented on Tensor Core without precision refinement.
Mark [10]	FP16	FP32	Markidis's implemented on two platforms.
Ooto [43]	FP16	FP32	Ootomo's implemented on two platforms.

refinement. Finally, the experiments compare Markidis's and Ootomo's methods, which calculate FP32 results based on FP16 input with precision refinement. In this paper, the experimental results are averaged over 10 experiments. The baseline methods are described in Table 5.

Evaluation criteria. To objectively evaluate the effectiveness of the proposed method, in this paper, two criteria are used to evaluate the speed and accuracy. In terms of measuring speed, the experimental results were represented by the average value of TFLOPS (Tera FLoating-point Operations Per Second), which is calculated as:

$$\text{TFLOPS} = \frac{2((M \times N \times K) + (M \times N))}{total\_time * 1.0e12} \tag{18}$$

where M, N and K represent the matrix dimension of the GEMM, and $total\_time$ represents the running time in seconds measured on GPU. To evaluate the calculated results, the max relative error (Max_error) is utilized to evaluate the error compared to the CPU computation results, which is calculated as:

$$\text{Max\_error} = Max\left(\frac{|toDouble(V_p(i,j)) - toDouble(V_{single}(i,j))|}{|toDouble(V_p(i,j))| + |toDouble(V_{single}(i,j))|}\right)$$
$$i = 1, 2, \ldots, M, j = 1, 2, \ldots, N. \tag{19}$$

where $V_p(i,j)$ denotes the computational result under the precision refinement procedure, which can achieve single-precision results with half-precision matrices in the paper, $V_{single}(i,j)$ is the result of a standard single-precision calculation used to compute Max_error, (i,j) represents the ith row and jth column element of the matrix, $toDouble()$ is a precision transformation function to obtain more accurate errors, M and N represent the dimensions of the resulting matrix, and $Max()$ is to choose the maximum value from all errors. Meanwhile, the Mean Relative Error Distance (MRED) is also used as an error evaluation metric, which is calculated as follows:

$$\text{MRED} = \frac{1}{M * N}\sum_{i=1}^{M}\sum_{j=1}^{N}\left|\frac{toDouble(V_{single}(i,j)) - toDouble(V_p(i,j))}{toDouble(V_{single}(i,j))}\right| \tag{20}$$

where M and N denote the size of the result matrix, and $V_{single}(i,j)$ and $V_p(i,j)$ denote the same meaning as in Eq. (19).

5.2. The speed improvement with various shapes

To evaluate the speed advantages of our method, we conducted comparative experiments on two GPU platforms with multiple shapes of the matrix. The results are presented in Figs. 8–9. In Fig. 8(a)–(c), our method improves performance by 1.31× compared to rocBLAS, an arithmetic library commonly used for scientific computing and provided by AMD. We also compare rocWMMA, which directly uses half-precision inputs to achieve a single-precision output without a precision refinement process. The proposed method achieves an average 1.25×, 1.24×, and 1.45× speedup in (n*n*n), (n*4n*n), and (n*n*4n), respectively. From Fig. 8(a)–(b), when n ≤ 4096, it can be seen that the performance of rocWMMA is superior to that of the proposed method. This phenomenon is mainly due to the following two reasons. First, rocWMMA is a straightforward computation way; this does not involve the accuracy refinement procedure; therefore, the workload of the calculation process is less. Second, the dimension of the matrix is minor, so the proposed method cannot fully exploit the performance improvement brought by the multi-level flow design. As the matrix dimension gradually increases, the performance advantage of the proposed method becomes increasingly apparent, as seen in Fig. 8. Compared to Markidis's and Ootomo's methods, the proposed method has a speedup of 3.05× and 3.18×, respectively. These two methods utilize precision refinement techniques but require more computation workload. The specialized hardware accelerator's data layout and instruction scheduling characteristics were not considered, leading to lower performance. The proposed method utilizes the thread parallelism analytic model to choose the tiling scheme to segment the matrix. Additionally, an efficient multi-level pipeline is used to improve the efficiency of data access and ILP. This makes it an effective solution for addressing the issue of low computation speed in precision refinement. Fig. 8 indicates that the proposed method has performance benefits when the matrix dimension is large. This serves as further evidence that the suggested LE-GEMM has significantly improved performance for both square and skewed matrices.

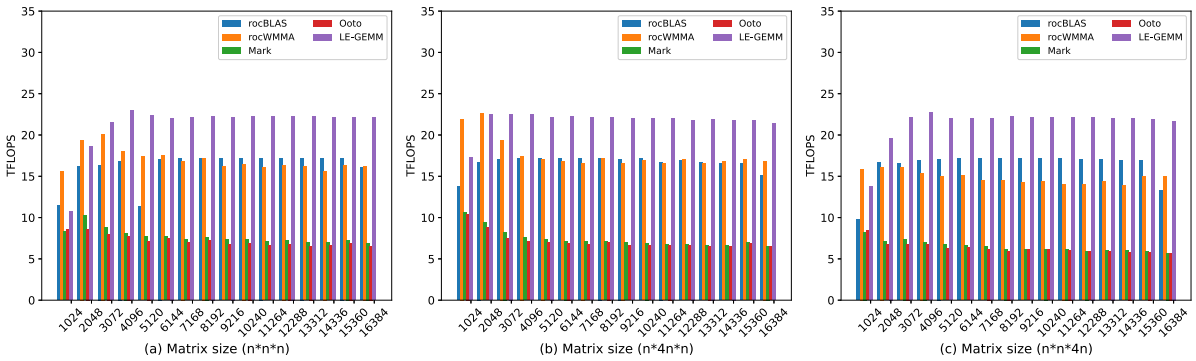


Fig. 8. The experimental results with various shapes on MI210.

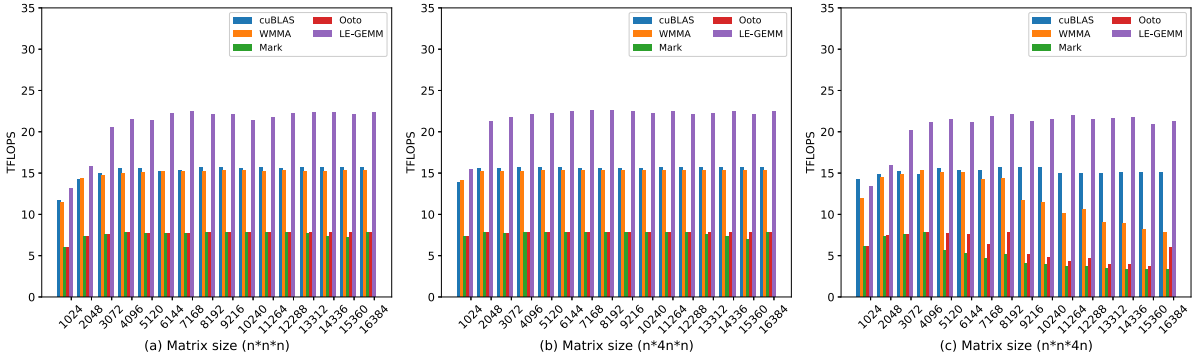


Fig. 9. The experimental results with various shapes on A800.

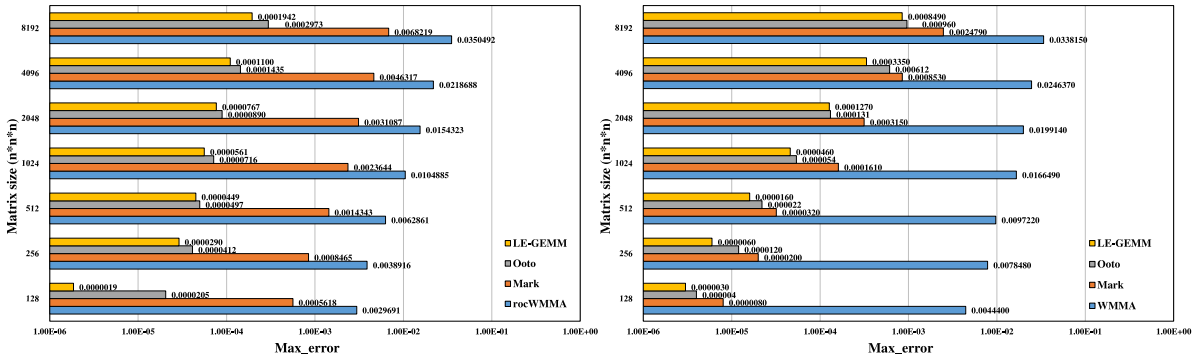
The experimental results of speed comparison on NVIDIA A800 are shown in Fig. 9. In the square matrix case (n*n*n), the proposed method achieves an average speedup of 1.37×, 1.40×, 2.78× and 2.74×, which are compared with cuBLAS, WMMA, Mark and Ooto, respectively. As illustrated in Fig. 9, it can be seen that when the matrix size is small, the performance of LE-GEMM is similar to the compared methods. For example, when n $\leq$ 2048, Fig. 9(a) shows that the proposed method performs similarly to cuBLAS and WMMA. The above phenomenon is because a smaller matrix will reduce the number of tiles and the number of iterative accumulations within the tiles, reducing the performance gain brought by the multi-level pipeline. As the size of the matrix increases, the benefits of the proposed method become more apparent. When the matrix size is large, the proposed method significantly improves performance over other methods. Experiments conducted on two GPU platforms demonstrate that the performance of the Markidis and Ootomo methods is nearly consistent. The reason is that, while Ootomo's method reduces the computational effort by 25%, Ootomo's method requires an additional accumulator to perform the accumulation. The initiation and combination stages of the accumulator also require an additional amount of time. When the matrix is skewed, the proposed method outperforms the other baseline methods significantly. Fig. 9(c), when n>7168, the performance of the WMMA method decreases because it is a simple and direct implementation that ignores the data loading pipeline and multi-level memory architecture during the computation process. This simple implementation results in lower efficiency when accumulating along the K-dimension, which is more pronounced when the matrix size is (n*n*4n). This approach makes WMMA unable to fully utilize its hardware acceleration capability. Fig. 9(b) illustrates that the proposed method achieves a speedup of 1.40×, 1.43×, 2.84× and 2.80× over the four baseline methods, respectively. As shown in Fig. 9, the proposed method performs better when the matrix size is large; this performance trend is consistent with the AMD platform. The detailed comparative experimental results of two GPU platforms are presented in Table 6.

Table 6
The speedup on two GPU platforms.

Baseline method	AMD MI210			NVIDIA A800		
	(n*n*n)	(n*4n*n)	(n*n*4n)	(n*n*n)	(n*4n*n)	(n*n*4n)
rocBLAS/cuBLAS	1.31×	1.31×	1.32×	1.37×	1.40×	1.36×
rocWMMA/WMMA	1.26×	1.24×	1.45×	1.40×	1.43×	1.81×
Mark	2.81×	2.99×	3.35×	2.77×	2.84×	4.67×
Ooto	2.99×	3.10×	3.45×	2.74×	2.80×	3.76×

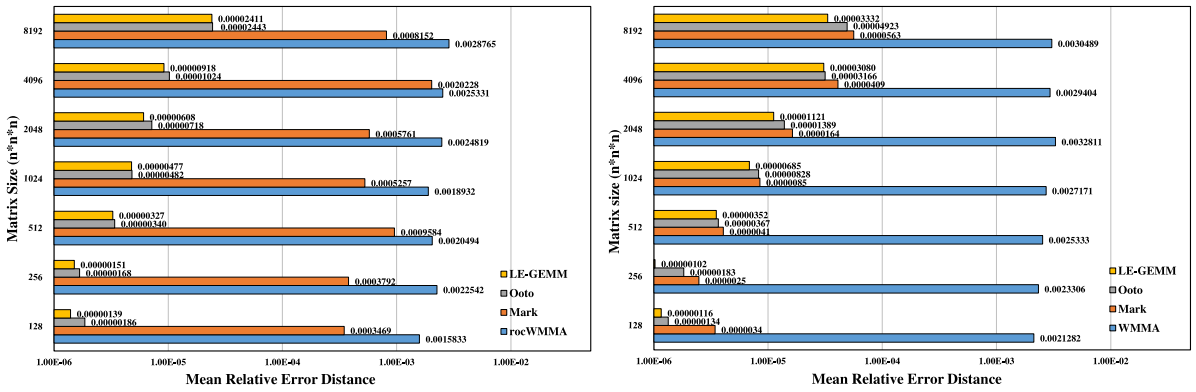
5.3. The precision error

This section compares several baseline methods in terms of their maximum error. The experiment results are shown in Fig. 10. First, we compare with rocWMMA, which computes FP32 GEMM directly from FP16 using Matrix Core. In Fig. 10(a), the proposed method reduces the average Max_error by 377.33× compared to rocWMMA. The results indicate that FP16 cannot preserve all the mantissa of FP32 due to the limitation of the mantissa length. Additionally, this method lacks precision refinement techniques, resulting in a pronounced Max_error. The comparison result shows that the proposed method can significantly reduce the Max_error compared to the default routine. Mark and Ooto methods were then compared separately. These two methods utilize precision refinement techniques to decrease Max_error. These methods have improved computational accuracy compared to the default methods. Fig. 10(a) shows that the proposed method reduces the average Max_error by 74.82× compared to Mark. Although Markidis et al. [10] employed an additional FP16 execution precision refinement process to reduce Max_error, the effect of the rounding method on FP16 was not considered, leading to certain limitations in improving accuracy. The proposed method reduces Max_error by an average of 2.69× compared to Ootomo's method. However, the addition of FP16 does not preserve the residuals of FP32 and FP16 efficiently due to the limitations of the numerical representation range. Rounding inevitably causes fluctuations in values, leading to errors in the values preserved by FP16. To



(a) AMD platform：MI210 (b) NVIDIA platform：A800

Fig. 10. The Max_error with square matrix (n*n*n) on two GPU platforms.



(a) AMD platform：MI210 (b) NVIDIA platform：A800

Fig. 11. The Mean Relative Error Distance with square matrix (n*n*n) on two GPU platforms.

alleviate this problem, the proposed method employs a bit-cut method for the high-bit of FP32 mantissa to avoid the value fluctuation caused by rounding. Fig. 10(a) shows that the Max_error increases with the matrix dimension (K-dimension). This is due to the fact that the two FP16 cannot fully preserve the FP32 values, resulting in errors in the calculated results. As the number of computations increases, the error gradually becomes more apparent. The effectiveness and advancement of the proposed method can be demonstrated by comparing with the default (rocWMMA) method and the latest (Mark and Ooto) methods on the AMD-platform.

To demonstrate the proposed method's portability, we have conducted corresponding experiments on NVIDIA A800, another popular GPU platform. The experimental results for Max_error are shown in Fig. 10(b). To ensure an objective comparison, we applied Markidis's method and Ootomo's method on the NVIDIA platform, respectively, using the WMMA interface. The interface name is the only difference between this implementation and the AMD-platform implementation. Fig. 10(b) illustrates that the proposed method significantly reduces Max_error as the matrix size increases, which is consistent with the results obtained on the AMD-platform. The proposed method also has a smaller Max_error compared to WMMA, Mark, and Ooto, reducing average Max_error by 571.75×, 2.78× and 1.41×, respectively. The experimental results of the two platforms show the same trend as the results in the AMD-platform. This indicates that the proposed method has low Max_error on mainstream GPU platforms, further confirming its effectiveness in terms of computational accuracy.

In this section, we have also provided the MRED of several comparative methods, which allows for a more comprehensive analysis of the matrix error. The detailed experimental results are presented in Fig. 11. As illustrated in Fig. 11(a), the proposed method exhibits a smaller MRED than rocWMMA, Mark, and Ooto, with a reduction of 636.92×, 178.93×, and 1.12× on MI210, respectively. By combining the experimental results in Figs. 10–11, it can be found that rocWMMA and WMMA have enormous Max_error and MRED compared to the other methods, which also shows the importance of precision refinement. In A800, the proposed method reduces the MRED by 814.87×, 1.75×, and 1.28× compared to WMMA, Mark, and Ooto, respectively. The experimental results of Figs. 10–11 demonstrate a consistent trend: the Max_error and MRED are more significant when the cumulative dimension increases. As shown in Figs. 10–11, it can be found that the proposed method has a smaller Max_error and MRED, which indicates that it has a smaller upper limit of error and mean relative error. The comparative experiment between Max_error and MRED proves that the proposed method has better computational accuracy.

This section provided error distribution histograms to demonstrate each method's error range. The error distribution histogram is implemented on two GPU platforms with a matrix size of 1024*1024*1024. The detailed error distribution results are shown in Figs. 12–13. As shown in Figs. 12–13, the error distribution of rocWMMA and WMMA is the widest, indicating that the error impact without an additional precision computation process is significant. Compared to the Mark method, the proposed method has obvious advantages in both the error range and maximum error. This precision advantage is more evident on MI210. The above gap is because although Mark considers the impact of down-conversion, a simple rounding way cannot effectively preserve low-bit bits. Moreover, the differences in the implementation of rounding API across different hardware platforms can also impact the calculation results. Compared to the Ooto method, the error distribution histograms of the two methods are similar. The error of the proposed method is more concentrated around 0 in Figs. 12 and 13, indicating that the proposed method is superior to Ooto in terms of error distribution. The error histograms shown in Figs. 12 and 13 indicate that the proposed method outperforms other methods regarding

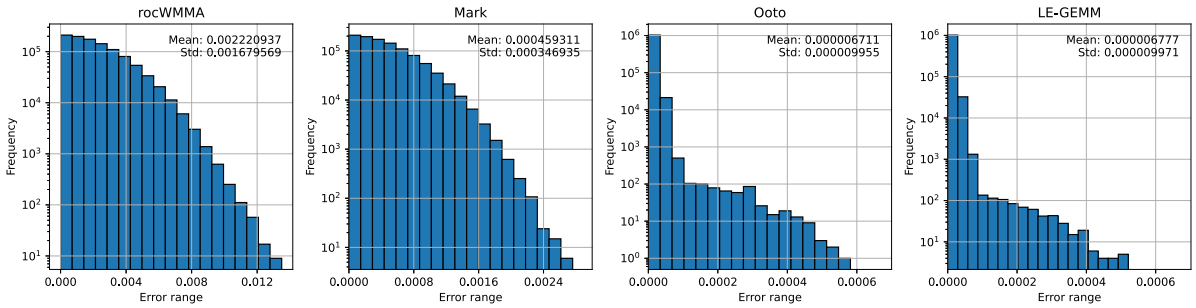


Fig. 12. Error distribution histogram on MI210.

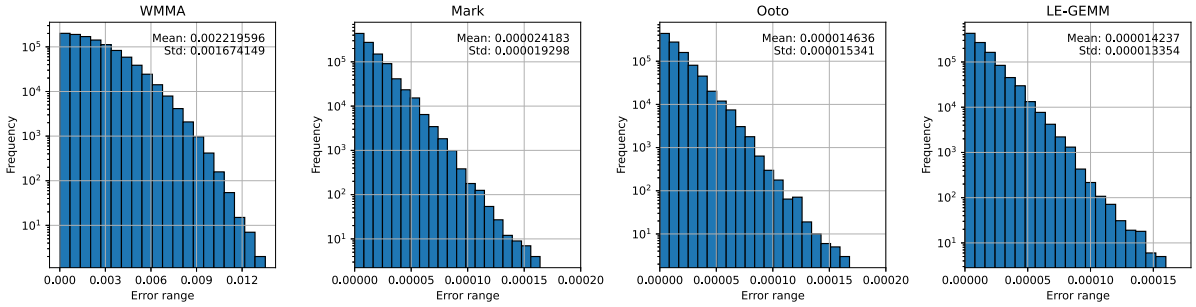


Fig. 13. Error distribution histogram on A800.

both the error upper limit and the error distribution on both platforms, proving its accuracy advantage.

5.4. The empirical Roofline analysis

To further demonstrate the effectiveness of the proposed method, we present a performance comparison and analysis of empirical Roofline performance on two GPU platforms. In the comparison experiments, GEMMs with larger dimensions are chosen for profiling, which avoids performance estimation bias due to fewer data elements in the multi-level cache. To measure performance metrics, we utilized Omniperf [56] and NCU [57] commands, which are profiling tools provided by AMD and NVIDIA, respectively. The empirical roofline performance is typically composed of two regions: the "memory-bound" and the "compute-bound", which indicate that the current kernel performance is limited by memory access and compute capacity, respectively. The specific experimental results are shown in Fig. 14. From Fig. 14, it can be seen that the proposed method is in the "compute-bound" region compared to Mark, Ooto, and rocWMMA, indicating that the proposed method has a higher bandwidth utilization for multi-level memory in the GPU. This is due to the fact that the above baseline approach does not take into account the data layout of specialized hardware units and the multi-level cache characteristics on GPUs, resulting in lower performance. To address this issue, the proposed method uses double buffers and pre-fetching techniques to establish an efficient multi-level pipeline design. This design empowers the kernel to optimize the use of data in the multi-level cache, leading to improved performance. For rocBLAS and cuBLAS, although both methods fall into "compute-bound", the performance of the proposed method is better. This also shows the computational effectiveness of the proposed method for single-precision GEMM. The comparison of the empirical Roofline performance on the two platforms can further confirm the effectiveness and advancement of the proposed method.

5.5. The improved performance for GEMM-based scientific computing scenarios

To demonstrate the proposed method's acceleration performance in GEMM-based scientific computing, we selected three real-world scientific computing scenarios: K-means [6], FFN layer in BERT [8], and Linear transformation in LLMs [58], which involve numerous GEMM operations. Fig. 15 shows the speedup ratio of the proposed method compared to the default implementation (rocBLAS and cuBLAS) on two GPU platforms. The average speedup of the proposed method in K-means for rocBLAS and cuBLAS are 1.16× and 1.33×, respectively, as shown on the left of Fig. 15. The GEMM operation's shape during computation is determined by altering the number of data points in the K-means comparison experiments. The proposed method's performance advantage increases gradually with the number of data points and then stabilizes. This suggests that the proposed method has more significant performance gains in large-scale GEMM. Fig. 15 illustrates the performance improvement of the FFN layer. The FFN has incorporated the proposed method to accelerate the computational process of the BERT layer. As illustrated on Fig. 15, the proposed method has achieved 1.16× and 1.18× speedups compared to rocBLAS and cuBLAS. Experiments on various input sequence lengths show that the proposed method can significantly improve the speed performance of the FFN computation process on two GPU platforms. Meanwhile, we also apply the proposed method to the linear transformation in LLMs to accelerate their linear mapping process, which involves various sizes of GEMM workload. During the linear transformation process of LLMs, the input features are mapped to product Q, K, and V, commonly known as Query, Key, and Value matrices. The range of matrix data from the LLMs training process is [0.5, 2.5]. In Fig. 15, for the computation process of multiple-size weight matrices, the proposed method has a speedup of 1.23× and 1.34× compared to the default implementation, respectively. Fig. 16 shows the Max_error and MRED of the proposed method in three GEMM-based applications. As illustrated in Fig. 16, the magnitude of the error associated with the proposed method is primarily contingent upon the K-dimension of GEMM. As the K-dimension increases, both Max_error and MRED of the proposed method also increase. The computation process in GEMMs with larger K-dimensions involves more FMA operations, leading to more significant errors.

5.6. The performance gain analysis

In this section, we evaluate the performance gains of two techniques, thread parallelism analytic model and multi-level pipeline, for

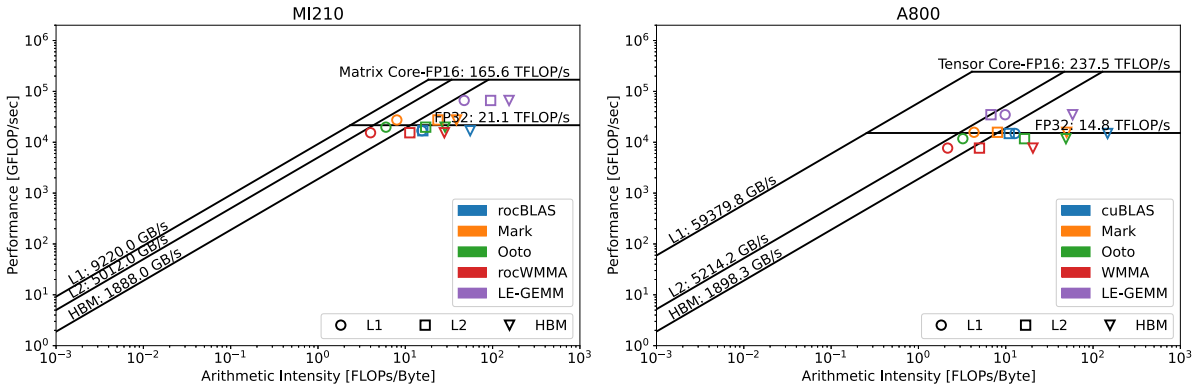
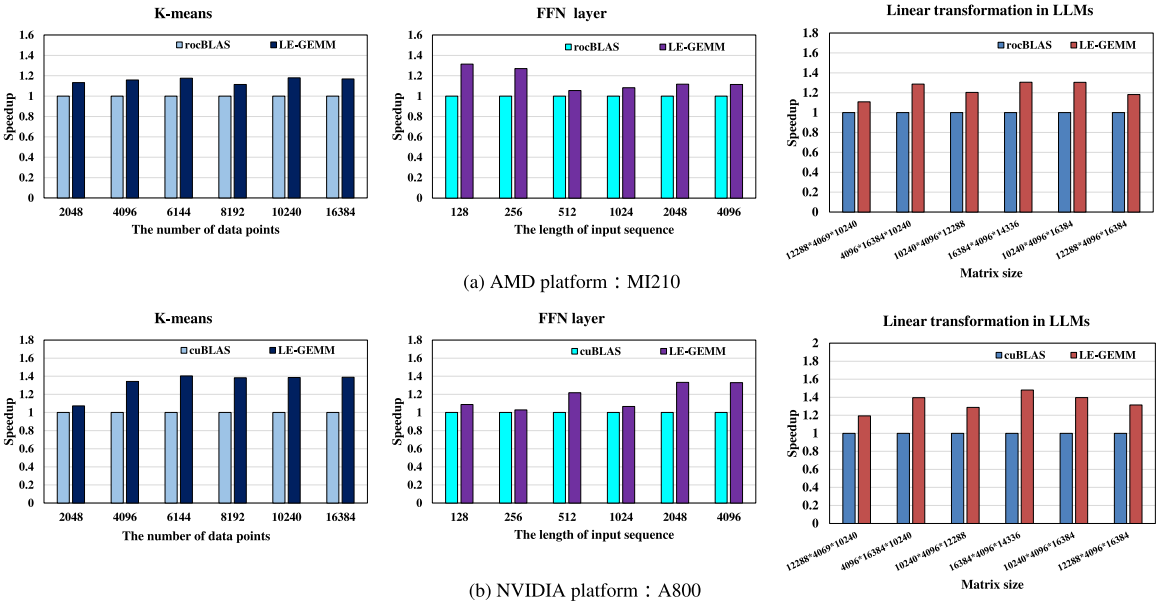


Fig. 14. The empirical Roofline performance.



(a) AMD platform：MI210

(b) NVIDIA platform：A800

Fig. 15. The acceleration performance with LE-GEMM (Note that rocBLAS and cuBLAS serve as a baseline for performance comparison. Hence, the speedup of these methods is 1.).

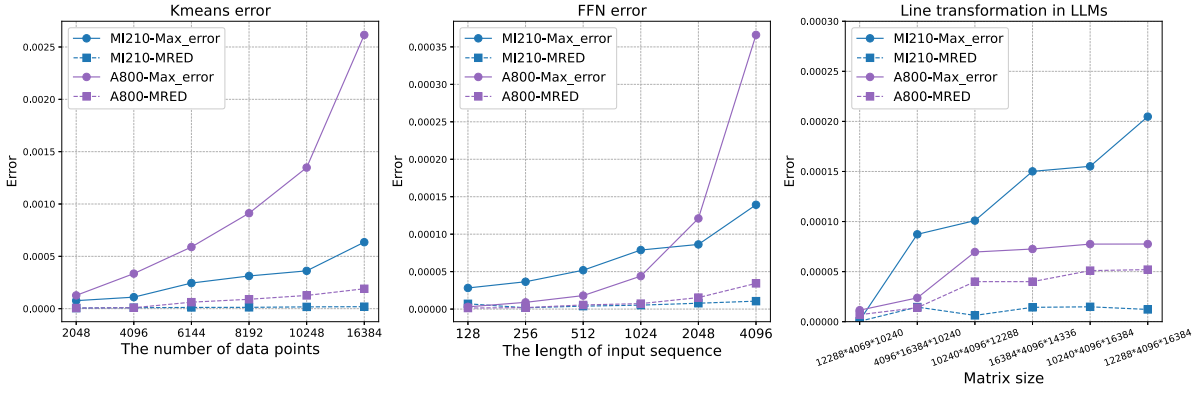


Fig. 16. The Max_error and MRED with LE-GEMM.

LE-GEMM performance. The detailed experimental results are shown in Fig. 17. In Fig. 17, "LE-GEMM manual paramete" represents the use of previous experience to select the appropriate parameters. "LE-GEMM plain implementation" is the direct use of the API interface without the multi-level pipeline. As shown in Fig. 17, the thread parallelism analytic model brings 1.36× and 1.78× performance improvement on MI210 and A800. When the matrix size gradually increases, the performance gain is more obvious. For example, when the matrix size ≥ 4000, there is a significant performance gap between the proposed method and the manual parameter implementation. This is because different parameters will have a significant performance impact for varying hardware architectures and matrix sizes. The manual parameter way

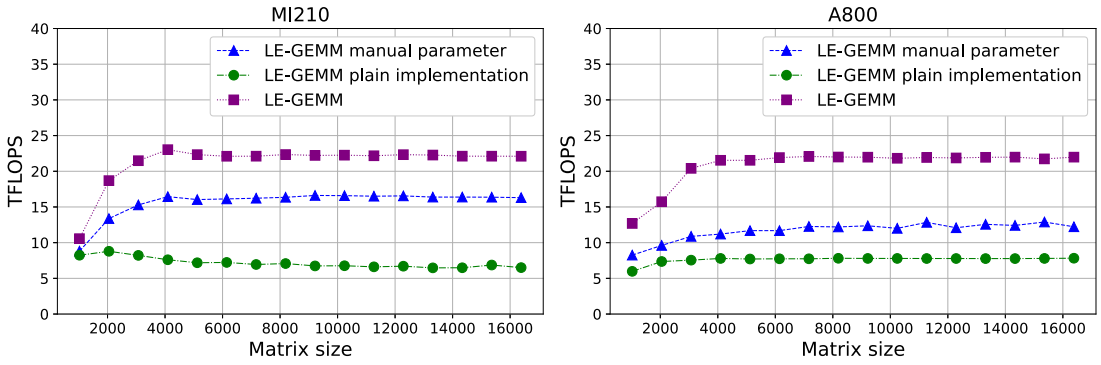


Fig. 17. The performance gain analysis of LE-GEMM ("LE-GEMM manual parameter" indicates LE-GEMM without thread parallelism analytic model and "LE-GEMM plain implementation" indicates without multi-level pipeline).

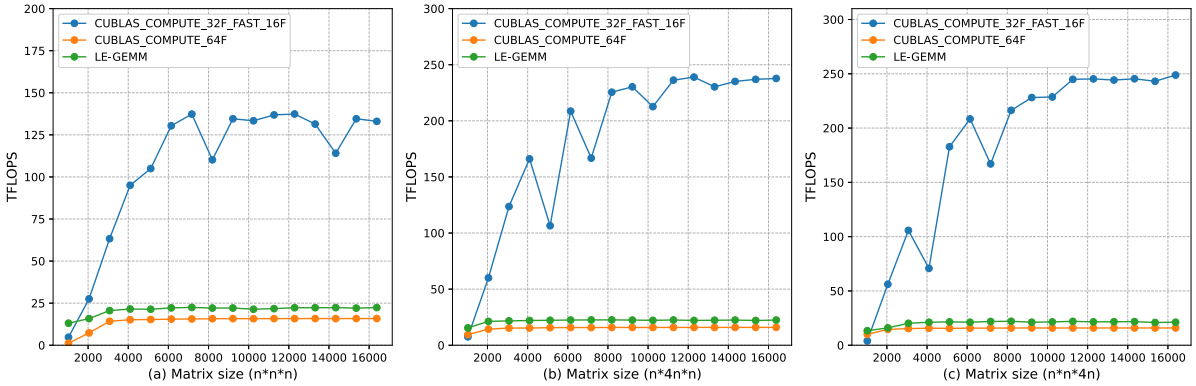


Fig. 18. The trade-off between performance and precision.

relies solely on prior experience in selecting kernel parameters. This approach often requires several trial-and-error to obtain a suitable parameter value. However, the lack of detailed analysis of the hardware architecture and the computational workflow makes it difficult to obtain the optimal parameters in the manual selection approach. We also perform LE-GEMM plain implementation, which ignores the multi-level memory architecture feature and adopts a simple and direct calling way. Fig. 17 illustrates that LE-GEMM exhibits a performance speedup of 3.02× and 2.72× over the plain implementation on two GPU platforms. In LE-GEMM plain implementation, threads are required to wait for the next data load after completing a single iteration calculation, which significantly reduces performance. Compared to plain implementation, LE-GEMM can efficiently utilize various memory spaces for pipelined access operations, reducing data loading overhead and improving CU utilization.

5.7. The trade-off between performance and precision

To explore the trade-off between high performance and precision, some comparative experiments with different precision were conducted on the A800. We used the cublasGemmEx() function in the comparative experiment to implement two different computation modes (CUBLAS_COMPUTE_32F_FAST_16F and CUBLAS_COMPUTE_64F). CUBLAS_COMPUTE_32F_FAST_16F mode means directly using FP16 to obtain single precision calculation results, during which down-conversion is performed. The detailed experimental results have been presented in Fig. 18. As shown in Fig. 18, CUBLAS_COMPUTE_32F_FAST_16F has the fastest computational performance because this method directly calls Tensor Core for matrix operations without any precision refinement process. The calculation process of CUBLAS_COMPUTE_32F_FAST_16F is consistent with WMMA, which directly uses down-conversion without a precision refinement process. Hence, the error of these two methods is also consistent. CUBLAS_COMPUTE_32F_FAST_16F often leads to significant errors, limiting its application range. The proposed method reduces the precision loss by using additional computational resources, which decrease the overall computational speed. Compared to COMPUTE_32F_FAST_16F, the proposed method has better computational accuracy. CUBLAS_COMPUTE_64F represents double-precision matrix multiplication operation. It has higher data precision and lower computational speed. The experimental results in Fig. 18 indicate that higher precision data tends to have lower computational speed. The proposed LE-GEMM achieves single-precision computational results based on low-precision computational units, essentially a compromise between computational performance and accuracy.

6. Conclusion

This paper proposed LE-GEMM for single-precision matrix multiply, which contains a lightweight emulation algorithm, thread parallel analytic model, and efficient multi-level pipeline implementation. Specifically, a lightweight emulation algorithm can provide more accurate computational results and eliminate redundant computations. Then, a thread-parallel analytical model is designed to determine the optimal tiling scheme by considering TLP and single thread performance. A multi-level pipeline design is implemented to achieve access latency hiding and higher ILP. Finally, experiments were conducted on two GPU platforms to validate the effectiveness of the proposed method and its progressiveness.

7. Future work

Future work includes exploring the impact of different types of low-precision data on computational results, allowing for better trade-offs

between performance and accuracy. Recently, low-precision hardware accelerators have been increasingly applied to various GEMM-based scientific computing scenarios to improve computational speed. The differences in hardware support and implementation can significantly impact the calculation results. In future work, we plan to explore more low-precision variables (e.g., BF16, FP8, FP4) to speed up the computational process in single-precision matrix multiplication.

CRediT authorship contribution statement

Yu Zhang: Writing – review & editing, Writing – original draft. **Lu Lu:** Writing – review & editing. **Zhanyu Yang:** Writing – review & editing, Conceptualization. **Zhihong Liang:** Writing – review & editing. **Siliang Suo:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Natural Science Foundation of Guangdong Province, China (2024A1515010204) and the Technological Research Project of Southern Power Grid Company, China (ZBKJXM20232483).

Data availability

No data was used for the research described in the article.

References

[1] M. Meyer, T. Kenter, C. Plessl, Multi-FPGA designs and scaling of HPC challenge benchmarks via MPI and circuit-switched inter-FPGA networks, ACM Trans. Reconfigurable Technol. Syst. 16 (2) (2023) 1–27.
[2] H. Martínez, S. Catalán, A. Castelló, E.S. Quintana-Ortí, Parallel GEMM-based convolutions for deep learning on multicore ARM and RISC-V architectures, J. Syst. Archit. (2024) 103186.
[3] Y.R. Choi, V. Stegailov, Multi-GPU GEMM algorithm performance analysis for NVIDIA and AMD GPUs connected by NVLink and PCIe, in: International Conference on Mathematical Modeling and Supercomputer Technologies, Springer, 2022, pp. 281–292.
[4] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, Y. Ding, EGEMM-TC: accelerating scientific computing on tensor cores with extended precision, in: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2021, pp. 278–291.
[5] N.S. Murthy, F. Catthoor, M. Verhelst, Optimization of block-scaled integer GeMMs for efficient DNN deployment on scalable in-order vector processors, J. Syst. Archit. (2024) 103236.
[6] M. Ahmed, R. Seraj, S.M.S. Islam, The K-means algorithm: A comprehensive survey and performance evaluation, Electronics 9 (8) (2020) 1295.
[7] G. Chen, Y. Ding, X. Shen, Sweet KNN: An efficient KNN on gpu through reconciliation between redundancy removal and regularity, in: 2017 IEEE 33rd International Conference on Data Engineering, ICDE, IEEE, 2017, pp. 621–632.
[8] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, 2018, arXiv preprint arXiv:1810.04805.
[9] AMD, Matrix core, 2024, https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-matrix-cores-readme/.
[10] S. Markidis, S.W. Der Chien, E. Laure, I.B. Peng, J.S. Vetter, NVIDIA Tensor Core programmability, performance & precision, in: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, IEEE, 2018, pp. 522–531.
[11] AMD, AMD CDNA 2.0 architecture, 2024, https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf.
[12] W. Sun, A. Li, T. Geng, S. Stuijk, H. Corporaal, Dissecting Tensor Cores via microbenchmarks: Latency, throughput and numeric behaviors, IEEE Trans. Parallel Distrib. Syst. 34 (1) (2022) 246–261.
[13] G. Park, B. Park, M. Kim, S. Lee, J. Kim, B. Kwon, S.J. Kwon, B. Kim, Y. Lee, D. Lee, LUT-GEMM: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models, 2022, arXiv preprint arXiv:2206.09557.
[14] J. Liu, R. Gong, X. Wei, Z. Dong, J. Cai, B. Zhuang, Qllm: Accurate and efficient low-bitwidth quantization for large language models, 2023, arXiv preprint arXiv:2310.08041.
[15] Y. Wang, B. Feng, Z. Wang, G. Huang, Y. Ding, TC-GNN: Bridging sparse GNN computation and dense Tensor Cores on GPUs, in: 2023 USENIX Annual Technical Conference (USENIX ATC 23), 2023, pp. 149–164.
[16] R. Fan, W. Wang, X. Chu, Fast sparse GPU kernels for accelerated training of graph neural networks, in: 2023 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2023, pp. 501–511.
[17] T. Utsugiri, T. Kouya, Acceleration of multiple precision matrix multiplication using Ozaki scheme, 2023, arXiv preprint arXiv:2301.09960.
[18] M. Tatsumi, S.-I. Filip, C. White, O. Sentieys, G. Lemieux, Mixing low-precision formats in multiply-accumulate units for DNN training, in: 2022 International Conference on Field-Programmable Technology, ICFPT, IEEE, 2022, pp. 1–9.
[19] Y. Tortorella, L. Bertaccini, L. Benini, D. Rossi, F. Conti, RedMule: A mixed-precision matrix–matrix operation engine for flexible and energy-efficient on-chip linear algebra and TinyML training acceleration, Future Gener. Comput. Syst. 149 (2023) 122–135.
[20] D.N. Parikh, R.A. van de Geijn, G.M. Henry, Cascading GEMM: High precision from low precision, 2023, arXiv preprint arXiv:2303.04353.
[21] M. Fasi, N.J. Higham, F. Lopez, T. Mary, M. Mikaitis, Matrix multiplication in multiword arithmetic: Error analysis and application to GPU tensor cores, SIAM J. Sci. Comput. 45 (1) (2023) C1–C19.
[22] P. Valero-Lara, I. Jorquera, F. Lui, J. Vetter, Mixed-precision S/DGEMM using the TF32 and TF64 Frameworks on low-precision AI tensor cores, in: Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, 2023, pp. 179–186.
[23] X.-Y. Liu, Z. Zhang, Z. Wang, H. Lu, X. Wang, A. Walid, High-performance tensor learning primitives using GPU Tensor Cores, IEEE Trans. Comput. 72 (6) (2022) 1733–1746.
[24] A. Abdelfattah, H. Anzt, E.G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N.J. Higham, X.S. Li, et al., A survey of numerical linear algebra methods utilizing mixed-precision arithmetic, Int. J. High Perform. Comput. Appl. 35 (4) (2021) 344–369.
[25] M.O. Lam, J.K. Hollingsworth, B.R. de Supinski, M.P. Legendre, Automatically adapting programs for mixed-precision floating-point computation, in: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, 2013, pp. 369–378.
[26] A. Dakkak, C. Li, J. Xiong, I. Gelado, W.-m. Hwu, Accelerating reduction and scan using tensor core units, in: Proceedings of the ACM International Conference on Supercomputing, 2019, pp. 46–57.
[27] M. Fasi, N.J. Higham, M. Mikaitis, S. Pranesh, Numerical behavior of NVIDIA tensor cores, PeerJ Comput. Sci. 7 (2021) e330.
[28] AMD, RocWMMA: ROCm wavefront-level matrix multiply and accumulate, 2024, https://github.com/ROCmSoftwarePlatform/rocWMMA.
[29] D. Yan, W. Wang, X. Chu, Demystifying Tensor Cores to optimize half-precision matrix multiply, in: 2020 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2020, pp. 634–643.
[30] AMD, Next generation BLAS implementation for ROCm platform, 2024, https://github.com/ROCm/rocBLAS.
[31] A. Kerr, D. Merrill, J. Demouth, J. Tran, Cutlass: Fast linear algebra in CUDA C++, NVIDIA Dev. Blog (2017).
[32] B. Tuomanen, Hands-On GPU Programming with Python and CUDA: Explore high-performance parallel computing with CUDA, Packt Publishing Ltd, 2018.
[33] D. Lustig, S. Sahasrabuddhe, O. Giroux, A formal analysis of the NVIDIA PTX memory consistency model, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 257–270.
[34] D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura, Accurate matrix multiplication on binary128 format accelerated by Ozaki scheme, in: Proceedings of the 50th International Conference on Parallel Processing, 2021, pp. 1–11.
[35] J. Huang, L. Rice, D.A. Matthews, R.A. van de Geijn, Generating families of practical fast matrix multiplication algorithms, in: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2017, pp. 656–667.
[36] Y. Wang, B. Feng, Y. Ding, QGTC: accelerating quantized graph neural networks via GPU tensor core, in: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2022, pp. 107–119.
[37] B. Li, S. Cheng, J. Lin, Tcfft: Accelerating half-precision FFT through tensor cores, 2021, arXiv preprint arXiv:2104.11471.
[38] F. Zhang, Y. Hu, H. Ding, Z. Yao, Z. Wei, X. Zhang, X. Du, Optimizing random access to hierarchically-compressed data on GPU, in: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2022, pp. 1–15.
[39] R. Sakamoto, M. Kondo, K. Fujita, T. Ichimura, K. Nakajima, The effectiveness of low-precision floating arithmetic on numerical codes: a case study on power consumption, in: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, 2020, pp. 199–206.

[40] Y. Sun, L. Zheng, Q. Wang, X. Ye, Y. Huang, P. Yao, X. Liao, H. Jin, Accelerating sparse deep neural network inference using GPU Tensor Cores, in: 2022 IEEE High Performance Extreme Computing Conference, HPEC, IEEE, 2022, pp. 1–7.
[41] Z. Ma, H. Wang, G. Feng, C. Zhang, L. Xie, J. He, S. Chen, J. Zhai, Efficiently emulating high-bitwidth computation with low-bitwidth hardware, in: Proceedings of the 36th ACM International Conference on Supercomputing, 2022, pp. 1–12.
[42] Z. Lin, A. Sun, X. Zhang, Y. Lu, MixPert: Optimizing mixed-precision floating-point emulation on GPU integer tensor cores, in: Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, 2024, pp. 34–45.
[43] H. Ootomo, R. Yokota, Recovering single precision accuracy from tensor cores while surpassing the FP32 theoretical peak performance, Int. J. High Perform. Comput. Appl. 36 (4) (2022) 475–491.
[44] P. Blanchard, N.J. Higham, F. Lopez, T. Mary, S. Pranesh, Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores, SIAM J. Sci. Comput. 42 (3) (2020) C124–C141.
[45] N.J. Higham, T. Mary, Mixed precision algorithms in numerical linear algebra, Acta Numer. 31 (2022) 347–414.
[46] Z. Jia, M. Maggioni, B. Staiger, D.P. Scarpazza, Dissecting the NVIDIA volta GPU architecture via microbenchmarking, 2018, arXiv preprint arXiv:1804.06826.
[47] M. Lu, B. He, Q. Luo, Supporting extended precision on graphics processors, in: Proceedings of the Sixth International Workshop on Data Management on New Hardware, 2010, pp. 19–26.
[48] A. Thall, Extended-precision floating-point numbers for GPU computation, in: ACM SIGGRAPH 2006 Research Posters, 2006, pp. 52–es.
[49] B. Ferrarini, M. Waheed, S. Waheed, S. Ehsan, M.J. Milford, K.D. McDonald-Maier, Exploring performance bounds of visual place recognition using extended precision, IEEE Robot. Autom. Lett. 5 (2) (2020) 1688–1695.
[50] N.J. Higham, T. Mary, Sharper probabilistic backward error analysis for basic linear algebra kernels with random data, SIAM J. Sci. Comput. 42 (5) (2020) A3427–A3446.
[51] H. Ootomo, H. Manabe, K. Harada, R. Yokota, Quantum circuit simulation by SGEMM emulation on tensor cores and automatic precision selection, in: International Conference on High Performance Computing, Springer, 2023, pp. 259–276.
[52] N. Nakasato, A fast GEMM implementation on the cypress GPU, ACM SIGMETRICS Perform. Eval. Rev. 38 (4) (2011) 50–55.
[53] M. De Santis, G. Eichfelder, J. Niebling, S. Rocktäschel, Solving multiobjective mixed integer convex optimization problems, SIAM J. Optim. 30 (4) (2020) 3122–3145.
[54] R.S. Burachik, C.Y. Kaya, M.M. Rizvi, Algorithms for generating pareto fronts of multi-objective integer and mixed-integer programming problems, Eng. Optim. 54 (8) (2022) 1413–1425.
[55] NVIDIA, Programming tensor cores in CUDA 9, 2024, https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.
[56] AMD, Advanced profiling and analytics for AMD hardware, 2024, https://github.com/ROCm/omniperf.
[57] NVIDIA, The user guide for nsight compute, 2024, https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.
[58] G. Heo, S. Lee, J. Cho, H. Choi, S. Lee, H. Ham, G. Kim, D. Mahajan, J. Park, Neupims: Npu-pim heterogeneous acceleration for batched llm inferencing, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Vol. 3, 2024, pp. 722–737.