Journal of Systems Architecture 160 (2025) 103341

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

A load-balanced acceleration method for small and irregular batch matrix multiplication on GPU

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

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

ARTICLE INFO

Keywords:
Batch GEMM
Thread workload
Multi-thread kernel
Tiling algorithm

ABSTRACT

As an essential mathematical operation, GEneral Matrix Multiplication (GEMM) plays a vital role in many applications, such as high-performance computing, machine learning, etc. In practice, the performance of GEMM is limited by the dimension of matrix and the diversity of GPU hardware architectures. When dealing with batched, irregular and small matrices, the efficiency of GEMM usually performs poorly. To this end, a common approach is to segment the matrix into multiple tiles and utilize parallelism between workgroups in GPU to compute the results. However, previous works only consider tile size and inter-workgroup parallelism and ignore the issues of low computational efficiency and hardware resource utilization caused by the difference in workloads between wavefronts. To address these issues, we propose a load-balanced batch GEMM acceleration method, consisting of a multi-thread kernel design and an efficient tiling algorithm. The multi-thread kernel design can address the workload unbalance between wavefronts in different workgroups, and the efficient tiling algorithm can choose the optimal tiling scheme with the new thread-level parallelism calculation method to achieve load-balanced task allocation. Finally, various comparative experiments were conducted on two GPU platforms: AMD and NVIDIA. Experimental results indicate the proposed method outperforms previous methods.

1. Introduction

GEneral Matrix Multiplication (GEMM) is a standard computing kernel that plays an important role in high-performance computing [1], artificial intelligence [2], image processing [3], and other research fields. With the explosive growth of data volume and the emergence of various algorithms, the demand for high-performance GEMM computing is increasing [4,5]. Additional stream processors and memory are integrated into the GPU to cater to this trend, providing tremendous computational power for GEMM acceleration. To fully utilize the hardware acceleration capability, AMD and NVIDIA, provide developers with a platform for parallel computing based on GPU (ROCm and CUDA). Based on these parallel computing acceleration platforms, various optimization algorithms and acceleration libraries have been proposed and demonstrated to have powerful effects, such as rocBLAS [6], cuBLAS [7], MAGMA [8], etc. These methods achieve optimal computational task allocation through hardware resource scheduling and thread parallelism to accelerate the matrix multiplication operation [9,10].

Many real-world applications, such as deep learning, involve irregular, small-size matrix multiplication operations in their computations [11]. For example, in Convolutional Neural Networks (CNN) [12–14], the structure of these models contains a large number of convolutional layers. The scale of the convolution kernel tends to be small (e.g. "1*1" and "3*3"). Convolution operations are converted to GEMM using *Im2col* function, and the dimension of the matrix is typically less than 1000 [15,16]. These small GEMM computations prevent the GPU from fully exploiting its hardware computing potential. In this case, the scheduling overhead between batch GEMMs and the regularity of the matrix poses challenges to computational performance [17,18]. For a GEMM, the tiling is a standard solution method. The matrix is segmented into multiple tiles, and a thread block is responsible for computing individual tiles. Since each tile is independent, multiple tiles can be computed in parallel by using multiple threads in GPU, to speed up the computation process of GEMM. The larger dimension of tile will increase the Thread-Level Parallelism (TLP) of a single tile and also will

* 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.2025.103341
Received 3 September 2024; Received in revised form 3 November 2024; Accepted 8 January 2025
Available online 23 January 2025

reduce the number of tile, resulting in the failure to fully utilize the hardware resources of GPU [19,20]. The Instruction-Level Parallelism (ILP) of a single thread is related to the K-dimension. Generally, for a large enough matrix size, it can fully use GPU hardware resources and achieve higher TLP and ILP [21,22].

To improve computational efficiency, previous studies have proposed some acceleration methods for matrix multiplication. For instance, rocBLAS [6] and cuBLAS [7] provide batch GEMM API (*rocblasSgemmBatched* and *cublasSgemmBatched*), which can support multiple GEMMs to be simultaneously calculated on GPUs. However, these APIs support only uniform matrix sizes that considerably limit these applications. NVIDIA also provides a C++-style template library, CUTLASS [23], which utilizes built-in tile templates and sorting to accelerate matrix multiplication operations. In fact, the size of matrices is variable in many real-world applications [11]. To solve this issue, a Vbatch GEMM route that supports batch GEMM in various sizes is designed and implemented by MAGMA (*magmablas_sgemm_vbatched*). It adapts to batch GEMMs with multiple tiling strategies, assigning the appropriate tile to a single GEMM for huge performance gains. Although variable sizes are supported in MAGMA, it still has some limitations. First, MAGMA only supports some coarse-grained tiling strategies that are not appropriate for all GEMM. Coarse-grained tiling results in an unbalanced kernel workload and GPU utilization reduction. Second, the grid size is determined by the tiling of the largest matrix, which leads to idle threads and a waste of GPU computing power. Third, the lack of an evaluation criterion for tiling leads to lower efficiency of strategy choice.

To thoroughly support batch GEMM with variable sizes, it is essential to design a tiling algorithm that can be adapted to all GEMMs and adaptively choose tile sizes, not limited to single size. The optimal tiling for each GEMM is different, depending on the size of the matrix dimensions (M, N, K). How to choose a suitable tile is a challenge for batch GEMM. At the same time, an evaluation criterion based on the current GPU hardware and tiling strategy is also essential. With GPU hardware, an appropriate tiling for each GEMM can be chosen to fully utilize the GPU computing capabilities and achieve better computational performance. How to measure the effectiveness of the tiling algorithm on the GPU hardware is a challenging problem. The tile with various sizes can lead to significant differences in computational effort within each workgroup, further to an unbalanced distribution of computational tasks and excessive load differences between threads. Hence, for tiles with various sizes, balancing thread computation and data loading during computation is also a challenge for batch GEMM.

To address the above challenges, we propose a batch GEMM acceleration method with a multi-thread kernel design. Furthermore, an efficient tiling algorithm is proposed to achieve load-balanced and higher hardware resource utilization. Our contributions can be summarized as follows:

- A multi-threaded kernel design scheme is proposed to balance thread computation and data loading in different workgroups to compute the various tiles.
- A novel TLP computation method is designed to select the optimal tiling algorithm by combining the kernel occupancy of the GPU and the tiling operation.
- An efficient tiling algorithm is implemented by considering the GPU hardware architecture and the batch GEMM workload.
- The proposed method can efficiently handle batch irregular GEMM and achieve state-of-the-art performance on AMD and NVIDIA GPU platforms.

The rest of the paper is organized as follows. Section 2 provides related work and motivation. Section 3 introduces background on batch GEMM, GPU architecture, and kernel occupancy. Section 4 presents the details of the multi-thread kernel design and load-balanced tiling algorithm. Section 5 demonstrates and evaluates the experimental result. Section 6 provides the conclusions of the paper and future work. The source code of this paper can be obtained in this repository link: https://github.com/zhangyu0722/BatchGEMM.git.

2. Related work and motivation

2.1. Related work

Several approaches have been proposed for batch GEMM computation, which mainly focus on algorithm-level optimization or architecture-level optimization. The former mainly explores lower bounds on the time complexity of GEMM operations at the mathematical level and optimizes the computational effort. The latter is based on different GPU architecture features and uses corresponding optimization techniques to improve the computational efficiency of GEMM. In algorithm-level optimization, Strassen et al. [24] proposed a novel GEMM algorithm based on the property that matrix addition is faster than matrix multiplication to speed up the computational process, which uses seven-time multiplications and multiple addition operations instead of eight-time multiplications. This approach mathematically reduced the time complexity of GEMM to $O(n^{2.81})$ for the first time. To reduce the requirement of Strassen's algorithm for extra memory space, three different methods were proposed in [25]: pre-additions, overwriting the input matrix, and recursive scheduling to alleviate this problem. At the same time, due to the powerful effect of deep neural networks in various domains, Alhussein Fawzi et al. [26] transformed the process of finding the optimal complexity of matrix multiplication into a tensor decomposition problem and used reinforcement learning to explore lower bounds on the complexity of matrix multiplication. In particular, for a 4×4 matrix, the multiplication number was as low as 47 multiplications. This performance was better than the two-level Strassen's algorithm, which involves 49 multiplications. Although the above approach reduces the mathematical complexity of matrix multiplication operations, it is difficult to take advantage of the performance benefits of these approach due to the neglect of computational scheduling strategies and multi-level memory architecture features on the GPU.

In architecture-level optimization, GPU vendors (NVIDIA and AMD) have designed and implemented computing libraries such as cuBLAS [6] and rocBLAS [7] based on their parallel computing platforms to improve GPU hardware utilization and parallelism. However, due to the restriction of uniform-sized matrix, the performance is poor when faced with small and irregular batch GEMMs. Although NVIDIA provides a C++-style template library, the small size of the matrix and the lack of assembly-level optimizations make it difficult for CUTLASS to fully exploit its performance advantages for irregular and small matrix multiplication [23]. These irregular and small-sized matrices often lead to unbalanced workloads among threads in different workgroups, which can reduce kernel performance. For Sparse GEneral Matrix-Matrix multiplication (SpGEMM), the matrix's sparsity leads to significant differences in thread workloads [27,28]. To address the unbalanced workload, Chen et al. [29] optimized the matrix segmentation by analyzing the distribution of the floating point calculations of the CSR-based SpGEMM, which achieves load balance and performance improvement on Sunway TaihuLight. For the issue of workload unbalance in threads, it is necessary to conduct a detailed analysis of the computation process and hardware platform characteristics to design an efficient parallel framework implementation [30,31]. Xiao et al. [32] introduce a fine-grained partitioning strategy to select appropriate segmentation dimensions, efficiently utilizing the parallelism of multi-thread and improving the performance of binary sparse tensor contracts. The diversity of matrix sizes makes it difficult to utilize a unified routine for calculations, resulting in some threads being idle in CU [33,34]. Indeed, the size of matrices is variable and irregular in various scientific computing scenarios. To overcome the matrix restriction of uniform size, MAGMA [8] proposes a Vbatch routine to support batch GEMM with various sizes. In this way, it uses a 3D grid to indicate batch GEMM's kernel design, where *grid.z* represents batch size. Each GEMM corresponds to one of the 2D-grid planes, and the size of the two-dimensional plane (*grid.x*, *grid.y*) is determined by the largest GEMM. In the case of irregular GEMM, if the dimension

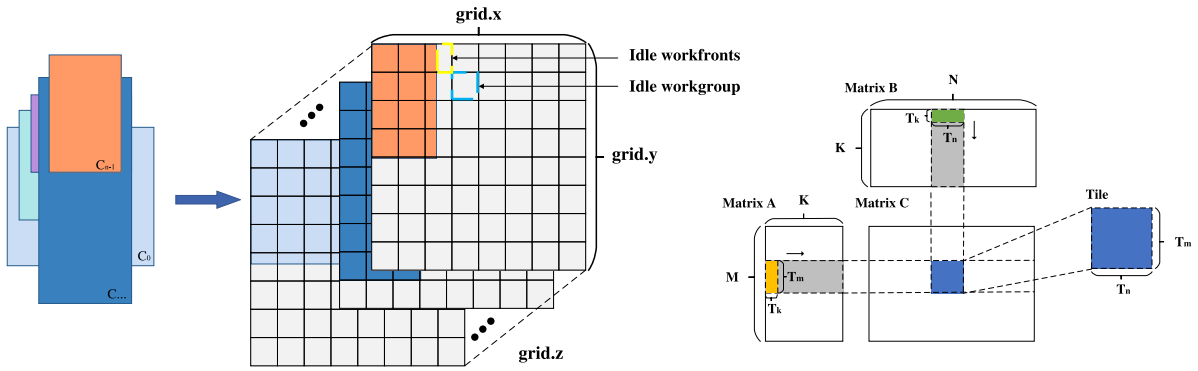


Fig. 1. GEMM and batch GEMM schematic diagram.

difference between the largest GEMM and the rest is too large, a large number of threads and workgroups will be idle, resulting in a waste of GPU computing resources. For various parallel acceleration platforms, different hardware characteristics, such as register size and number of CUs, will affect the allocation of computing resources in the kernel. To ensure kernel performance, it is necessary to flexibly set parameters based on different matrix sizes and hardware architectures [9,35]. To solve this problem, a coordinated tiling and batching strategy is proposed in [21], where a different tiling strategy is used for each GEMM in batch GEMM and appropriate batching is used according to the tile size to improve the computational efficiency of the GPU. Wang et al. [36] proposed the sort-up algorithm based on the GEMM workload and split-down in the tiling process, which can segment large tiles into multiple smaller tiles. This approach can make better use of CU utilization when the number of GEMM is limited.

2.2. Motivation

Although the above-mentioned methods improve the parallel computing efficiency of batch GEMM on GPU from various perspectives, there are two problems. One is that the workload of threads varies significantly across the kernel. In the above approach, tiles with various sizes are designed, and each tile is responsible for the corresponding kernel, where the number of threads is fixed. In general, larger tiles have better TLP. This will also increase the workload of each thread for large-size tiles, and the thread responsible for computing large tiles requires more hardware resources (VGPR, SGPR, LDS) and computing time. The other one is that differences between wavefronts within different workgroups are ignored in the TLP calculations. The workgroup will be transformed into multiple wavefronts during GPU computation and be executed in parallel on the CU. Each CU can run multiple wavefronts simultaneously, and the number of wavefronts depends on the hardware resources required by the wavefront. Thus, the TLP on the GPU should be determined by the number of threads in the wavefront that can be executed in parallel on the CU.

To solve the above problems, we propose an efficient and load-balanced batch GEMM acceleration method, which consists of two parts: a multi-thread kernel design scheme and an efficient tiling algorithm. A multi-thread kernel design is proposed to balance the amount of loading and computation in the thread corresponding to each tile. Tiles with various sizes correspond to the number of threads selected. Although this is limited by the parallel programming interfaces of the CUDA and ROCm platforms, the number of threads responsible for computing a tile is uniform. To overcome this shortcoming, we use the corresponding filtering operation in the kernel execution process to effectively alleviate this problem. An efficient tiling algorithm can choose the optimal scheme based on different GEMMs and GPUs. To measure the effect of tiling, we propose a new way of TLP computation based on wavefronts. The optimal tiling scheme is obtained by adjusting the tiling strategy according to the TLP. Finally, we obtain an efficient tiling algorithm based on the new TLP calculation method. In Section 4, the details of the proposed method are introduced.

3. Background

3.1. GEMM and batch GEMM

For a single GEMM, its accumulation routine is $C = \alpha AB + \beta C$, where $A \in R^{M\times K}$, $B \in R^{K\times N}$ and $C \in R^{M\times N}$ are dense matrices, M, N, and K represent matrix dimensions, and α and β are constant scalars. A common approach is tiling matrix C into multiple tiles [21,36], which utilizes the parallel computing of thread in GPU to calculate each tile and splices together the result. As shown in Fig. 1 (b), given a GEMM with size $M \times N \times K$, the matrix C is segmented into multiple tiles with $T_m \times T_n$. Each workgroup is responsible for the calculation of a tile and needs to access the row section of matrix A with size $T_m \times K$ and column section of matrix B with size $K \times T_n$. However, the row cross-section of A and the column cross-section of B (represented in Fig. 1 (b) by the gray parts of matrices A and B, respectively) are too large to store in shared memory and registers. Hence, the row section of A and the column section of B are segments of multiple A tiles with $T_m \times T_k$ and B tiles with $T_k \times T_n$, respectively. The partial result of C can be obtained by calculating with A tile and B tile, and accumulative partial results can obtain the final result.

To batch-run multiple GEMMs, a naive routine is computed for each GEMM individually. However, when the matrix size is small, a single GEMM does not fully utilize the GPU's computing power, leaving the CU idle [37,38]. To avoid this situation, a batch GEMM method is proposed to design multiple kernels for various GEMM in the GPUs [36,39]. Compared to GEMM, batch GEMM is expressed in $(M \times N \times K \times B_{size})$, where M, N and K represent the dimensions of the matrix, and B_{size} represents the batch size. A batch GEMM is 3D-dimension grid, where *grid.z* is batch sizes, and *grid.x* and *grid.y* are the lengths and widths of a two-dimensional plane respectively [40]. To balance the workload of a batch GEMM, a variety of tile sizes are used for GEMM tiling. The two-dimensional grid size has the corresponding matrix C and tiling strategy. Each tile is responsible for the corresponding workgroup. A workgroup is decomposed into multiple wavefronts that execute on the CU. The 3D grid of batch GEMM is shown in Fig. 1 (a).

3.2. GPU architecture and kernel occupancy

With the improvement of hardware architecture and parallel computing programming platforms (such as ROCm[1] and CUDA[2]), GPUs are becoming the most popular hardware accelerator. The two most commonly used GPUs are AMD and NVIDIA, widely used in various scientific computing platforms. However, some basic concepts of expression in ROCm and CUDA are different. We chose AMD's official

[1] https://rocm.docs.amd.com/en/latest/
[2] https://docs.nvidia.com/cuda/

Table 1
ROCm/CUDA terminology.

ROCm	CUDA	Description
Compute Unit (CU)	Streaming Multiprocessor (SM)	One of many parallel vector processors in a GPU that contains parallel ALUs. All waves in a workgroup are assigned to the same CU.
Kernel	Kernel	Functions launched to the GPU that are executed by multiple parallel workers on the GPU. Kernels can work in parallel with CPU.
Wavefront	Warp	Collection of operations that execute in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked off.
Workgroup	Thread block	Think of this as a vector thread. A 64-wide wavefront is a 64-wide vector op.
Work-item/Thread	Thread	GPU programming models can treat this as a separate thread of execution, though this does not necessarily get forward sub-wavefront progress.
Global Memory	Global Memory	DRAM memory accessible by the GPU that goes through some layers cache.
Local Memory	Shared Memory	Scratchpad that allows communication between wavefront in a workgroup.
Private Memory	Local Memory	Per-thread private memory often mapped to registers.

terminology for this paper to provide precise specifications. To clarify some differences and relationships between ROCm and CUDA terms, a comparison of terminology is given in Table 1.

A GPU is composed of multiple Shader Engines (SE) and a command processor. Each SE has its own workload manager. One SE is integrated with multiple CU and workload manager. Each CU contains an enormous amount of Arithmetic and Logic Units (ALUs), a small number of control units, and caches. Hence, GPUs are suitable for a large number of simple parallel computing tasks. A GPU kernel consists of one or multiple workgroups, the size of which is determined by the number of wavefronts and threads. On the memory hierarchy, the GPU has global memory, local memory, and private memory from slow to fast according to memory access speed, and local memory and private memory are much smaller than global memory [41,42].

Kernel Occupancy represents the actual utilization of computing unit resources by a kernel function on GPU, which is the ratio of actived wavefront to the maximum wavefront supported by CU [35,43]. An active wavefront running on CU requires resources such as Vector General-Purpose Register (VGPR), Scalar General-Purpose Registers (SGPR), Local Data Share (LDS), etc. A wavefront can be activated and run on a CU when all required resources are available. When the utilization of CU resources is low, the number of active wavefronts is small, which leads to the waste of hardware resources and the degradation of the parallel performance of the kernel. On the other hand, when the number of active wavefronts in the CU increases, the resources used by each wavefront and the available register storage space of each work-item in the wavefront decrease [44,45].

The number of active wavefronts on a CU is mainly limited by the following factors: the number of work-items in each workgroup and the sizes of VGPR, SGPR, and LDS. For example, in AMD's MI100[3] and MI210,[4] a wavefront consists of 64 work-terms. When the number of work-items in a workgroup is less than or equal to 64, only one wavefront is included. The VGPR, SGPR, and LDS sizes on the CU have a corresponding upper bound for each work-item. According to the kernel design, the resources on the CU need to be allocated before executing each work-item. When resource requirements of the work-item are satisfied, the wavefront can be active and run on the CU. Otherwise, it will not run until other wavefronts accomplish tasks and release resources. In order to fully utilize the hardware resources of the GPU and improve the efficiency of parallel computing, the kernel occupancy should be improved as much as possible without data overflow [46,47]. In batch GEMM, an efficient kernel design should properly allocate the data loading and computation workload for each work-item in the wavefront, so that the memory space and computing power on the CU can be more efficiently utilized [48,49].

[3] https://www.amd.com/system/files/documents/instinct-mi100-brochure.pdf

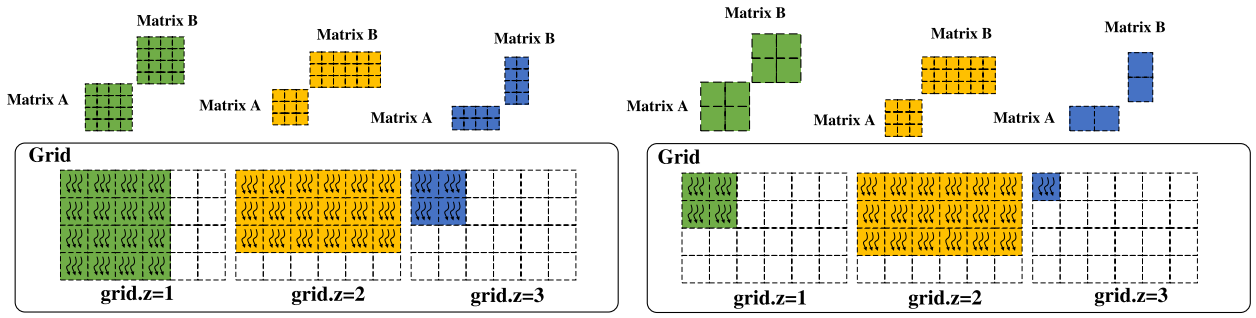
[4] https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf

4. Overview

4.1. Multi-thread kernel design

Tile size and kernel design are closely related in the design of batch GEMM algorithms, and there are two matrix tile design routes. The first way is to design a tile to adapt to all GEMMs, and the second is to design the various tiles to adapt to different GEMMs. Compared with the first method, for irregular GEMM, the latter method is more flexible and efficient to utilize the computing resources of GPU. For GEMMs with various shapes and sizes, using a single tile can easily lead to increased workload differences between threads in multiple workgroups, affecting the allocation of computing resources. In this paper, we perform a multi-thread kernel design for the second matrix segmentation method. Two different tile design strategies are shown in Fig. 2. Here we present the effect of two different tile strategies on the occupancy of the 3D grid. For the batch GEMM, different tile sizes lead to different numbers of workgroups, resulting in different 3D grid occupancies.

For a single GEMM, matrix C is tiled into multiple tiles. The tile size can be flexibly designed, and each tile can be run in parallel without data interference. Each tile is calculated by the corresponding workgroup and can be represented by a 2D-grid as a whole. When the size and number of tiles is large enough, efficient parallel execution efficiency can usually be obtained. However, in real-world cases, the size of matrices in batch GEMM tends to be small and irregular, which leads to poor performance of traditional methods. Therefore, the previous method adopts a variety of tiles to adapt to the corresponding GEMM, and each tile is based on a unified number of threads, which will lead to the workload of threads in large-scale tiles being much larger than that of small tiles. This gap in the workload of threads results in unbalanced thread loading and reduces GPU parallel computing efficiency. Table 2 lists the detailed parameters for tiles with various sizes based on the same work-item design (The number of work-items in the kernel is 128). W_{CP} and W_{DL} represent the computation



(a). Batch GEMM shares the same tile. (b). Batch GEMM uses different tiles.

Fig. 2. Two different tile design strategies for batch GEMMs. ((a) All GEMMs adopt the same tiling scheme, which is divided into multiple tiles of the same size. (b) Different GEMMs adopt different tiling schemes and are divided into multiple tiles of different sizes.).

Table 2
The common kernel design scheme for batch GEMM (There are significant workload gaps between threads).

Tile	T_m	T_n	T_k	W_{CP}	W_{DL}
small	16	16	8/16	2	4/6
medium	32	32	8/16	8	12/16
large	64	64	8/16	32	40/48

amount and data loading amount of work-item, respectively, and their calculation expressions are considered as:

$$W_{CP} = \frac{T_m \times T_n}{W_{num}} \tag{1}$$

$$W_{DL} = \frac{T_m \times T_n + T_m \times T_k + T_k \times T_n}{W_{num}} \tag{2}$$

where W_{num} represents the number of work-items responsible for computing the tile.

For different tiles, there is a significant gap in workload between threads ($W_{CP} \in [2, 32]$ and $W_{DL} \in [4, 48]$). The choice of T_k also has a certain impact on the data load of work-item. Each thread is responsible for more data loads when T_k is larger. For example, in large tile, when the value of T_k is set to 8 or 16, each work-item is responsible for loading 40 and 48 elements, respectively. The workload differences caused by these different tile sizes impact kernel performance.

To explore the impact of the number of work-items in the workgroup and the tile size on the performance of batch GEMM, some experiments are performed, whose results are given in Fig. 3. As shown in Fig. 3, under the condition that the number of GEMMs is large and M, N, and K are large enough, various thread-kernels (thread number is 64, 128, 256, and 512) are used to compute multiple tiles (The nine tiles are shown in Fig. 3). In Fig. 3, four thread kernels commonly used in previous work are selected as benchmarks [21,34,36]. We used these kernels to investigate their performance under various tiles in comparative experiments. Fig. 3 shows that the kernel's performance first increases and then decreases for different tiles. When the tile size is small, the thread's workload is also tiny. In this case, threads in the kernel only compute a few elements, which causes a lack of full utilization of threads' computing power. As the tile size increases, the number of elements that the thread needs to calculate and store is also increasing. Under the condition that the register data does not overflow, the computing efficiency of the thread is continuously improving. When the tile corresponding to the thread is too large, the register data overflows, and the data will be transferred to the global memory. For example, for a 64-thread-kernel, when computing "8*8" and "32*32" tiles, respectively, each thread needs to compute 1 and 32 elements in matrix C. It is obvious that "32*32" requires more register memory. However, the register memory of each thread is precious. When the maximum limit of the register memory is exceeded, the data will be transferred to the global memory for storage. Because the access speed of global memory is considerably lower than that of registers, threads' data access efficiency decreases, and overall time consumption increases. At the same time, since the variety of thread workloads, when a thread with a heavy workload is run on the CU, the number of active wavefronts on the CU is less, resulting in the CU's kernel occupancy (The ratio between the number of active wavefronts and the maximum number of supported wavefronts) will be reduced. The state of the CU with low kernel occupancy will be longer due to the longer work-item computation time.

To solve this problem, we propose a multi-thread kernel design, which ensures that the workload of each thread is balanced as much as possible. The experimental results in Fig. 3 show that multiple kernels' performance varies when calculating the same tile. For example, the 128-thread kernel performs best when calculating a tile with "32*32", as shown in Fig. 3. The performance gap mentioned above is mainly because of the varying workloads of threads under different kernels, which affects the overall performance. For the 128-thread kernel, when calculating a tile with "32*32", each thread needs to complete the calculation of 8 elements and the loading of 16 elements. When calculating a tile with "64*64", the workload of the threads is heavy, and each thread needs to complete the calculation of 32 elements and the loading of 64 elements. When calculating larger tiles, the workload of the thread increases significantly. To avoid significant differences in workload between threads, we used a multi-thread kernel to calculate various tiles by considering the computation amount (W_{CP}) and data loading amount (W_{DL}) of threads in the kernel. For larger tiles such as "32*64" and "64*64", a 256-thread kernel is used for computation. In this way, increasing the number of threads will reduce the thread's computation amount and data loading amount, thereby reducing the gaps between threads' workloads and achieving load balancing. There are five tiles and two kernels (W_{num}) for small and irregular batch matrix multiplication, as shown in Table 3. Compared to Table 2, we balance the thread workload by setting the tile size and number of kernel threads so that thread computation and data loading are as consistent as possible across different workgroups. In the calculation process of GEMM, five tile types are designed for GEMM calculation of different sizes, from "small" to "large". To ensure that the amount of computation and data loading for the work-item responsible for computing different tiles are as equal as possible, the number of threads varies depending on the tile size. In Table 3, two different thread numbers are used (128 and 256), respectively, and the computation amount (W_{CP}) and data loading amount (W_{DL}) of the work-item in each scheme are given. Although the current ROCm and CUDA platform programming interfaces only support the kernel design of a uniform thread number, we use a screening operation in the early stage of kernel execution to achieve the effect of kernel design of multiple threads. For example, in this paper, the number of kernel threads is set to 256. When the tiles of "small", "small-medium" and "medium" are executed, the extra threads will be terminated immediately and the corresponding computing resources will be released because these tiles only need

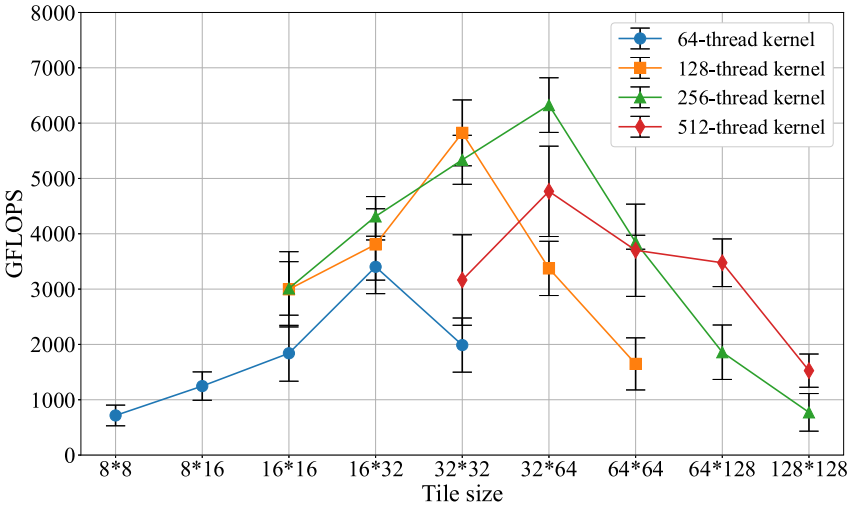


Fig. 3. Experimental results of multi-thread kernel.

Table 3
The multi-thread kernel design scheme with a more balanced workload.

Tile	T_m	T_n	T_k	W_{num}	W_{CP}	W_{DL}
small	16	16	16	128	2	6
small-medium	16	32	16	128	4	10
medium	32	32	16	128	8	16
medium–large	32	64	16	256	8	14
large	64	64	16	256	16	24

128 threads. Terminating threads early allows for a better allocation of computational resources to threads responsible for computing other tiles. With this implementation, we can achieve the effect of a multi-threaded kernel. Even though the performance may be degraded in comparison with an actual multi-threaded kernel, the experimental results in Section 5 demonstrate the excellent performance of this method.

4.2. Tiling algorithm

4.2.1. Criteria for evaluation

The tiling can be seen as a re-assignment of GEMM computation task. Efficient tiling algorithm can transform GEMM operations and improve hardware resource utilization. When various kernel designs are implemented, choosing an appropriate tiling scheme becomes a crucial issue. In general, for a GEMM, there will be better parallelism within the workgroup when the tile size is larger. However, a larger tile means that the number of tiles needs to be reduced. If the number of tiles is too few, the CU cannot be fully utilized, resulting in a waste of computing resources. Therefore, choosing a suitable tiling evaluation criteria is crucial. In the previous study, TLP was used to quantify the parallelism of tiling strategies on GPUs. Given a GEMM and a tiling strategy, its TLP can be calculated as follows:

$$TLP = \sum_i \frac{M_i \times N_i}{T_{mi} \times T_{ni}} \times T_{workgroup} \tag{3}$$

where M_i and N_i are the dimension size of matrix C of the ith GEMM, and T_{mi} and T_{ni} are the tile sizes chosen by matrix C. $T_{workgroup}$ is the number of threads in workgroup. However, the above formulation only considers TLP from the level of the workgroup. Indeed, during the computation of the GEMM, the workgroup needs to be further transformed into wavefronts and run on the CU in the form of a wavefront. The execution process of batch GEMM can be divided into four phases: segmentation, workgroup, wavefront, and execution. In the segmentation phase, the GEMM is tiling into tiles with various sizes, and each tile is computed by a workgroup. Workgroups are further transformed into wavefronts based on their hardware resource requirements and the number of work-item. Finally, these wavefronts are run in parallel on multiple CUs for batch GEMM calculations. Due to the difference between tile sizes, the computation amount and data loading amount of threads are not unified in the different wavefront, which will lead to unbalanced hardware resource requirements. The execution time of the wavefront on the CU is also different. The overall time of the batch GEMM is the maximum of all CU execution time. If the workload difference between wavefronts is too significant, the execution time of one wavefront will be excessive, increasing the overall calculation time consumption.

Therefore, Eq. (3) does not consider the workload gaps between wavefronts. To solve this problem, we propose a new TLP calculation method as follows:

$$TLP_{new} = \sum_i \varphi\left(\frac{M_i \times N_i}{T_{mi} \times T_{ni}}\right) \times T_{wavefront} \tag{4}$$

where the expression of M_i, N_i, T_{mi} and T_{ni} have the same meaning as Eq. (3), and $T_{wavefront}$ is number of work-item in wavefront, φ represents the conversion process of workgroup to wavefront.

The conversion process mainly considers the following factors: the number of workitems in the workgroup, the size of VGPR, SGPR, LDS required by a workitem, and the maximum number of wavefront supported in the CU. These factors are related to GPU hardware architecture. Next, take AMD's MI210, which is based on CDNA2.0 architecture, as an example. Under the limitation of the number of workitems in the workgroup, the number of wavefront can be calculated as follows:

$$WF_{wg} = 16 \times \text{ceil}\left(\frac{WI_{wg}}{64}\right) \tag{5}$$

where WF_{wg} is the maximum number of wavefronts under the limit of the number of work-item in the workgroup, and WI_{wg} represents the number of work-item in the workgroup. Eq. (5) indicates that when the number of work-item is less than or equal to 64, a workgroup contains only one wavefront, and the number of workgroups is limited to 16 in the CU.

Limited by the size of VGPR, SGPR, and LDS, the number of the wavefront can be calculated as follows:

$$WF_V = 4 \times \text{floor}\left(\frac{VGPR_{max}}{VGPR_{used} \times 64}\right) \tag{6}$$

where WF_V is the maximum number of wavefronts under the limit of the size of VGPR, $VGPR_{max}$ is the size of VGPR in the Single Instruction Multiple Data (SIMD) unit, and $VGPR_{used}$ is the VGPR size used by a

work-item. In the CDNA2.0 hardware architecture, each CU consists of four SIMDs.

$$WF_S = \text{floor}\left(\frac{SGPR_{max}}{SGPR_{used}}\right) \tag{7}$$

where WF_S is the maximum number of wavefronts under the limit of the size of SGPR, $SGPR_{max}$ is the size of SGPR in the CU, and $SGPR_{used}$ is the size of SGPR used by a wavefront.

$$WF_L = \text{floor}\left(\frac{LDS_{max}}{LDS_{used}}\right) \times \text{ceil}\left(\frac{WI_{wg}}{64}\right) \tag{8}$$

where WF_L is the maximum number of wavefronts under the limit of the size of LDS, LDS_{max} is the size of LDS in the workgroup, LDS_{used} is the size of LDS used by a workgroup, and the expression of WI_{wg} have same meaning as Eq. (5).

To sum up, the number of wavefronts should meet the limitations of all the above factors, and the calculation method is as follows.

$$WF = \min(WF_{wg}, WF_V, WF_S, WF_L, WF_C) \tag{9}$$

where WF is the number of activated wavefronts, WF_C is the maximum number of wavefront supported in the CU.

The number of wavefronts and the corresponding number of threads are introduced into Eq. (4) to compute the TLP more accurately and appropriately. Compared to Eq. (3), the former only considers the workload at the workgroup-level, which neglects further conversion between the workgroup and wavefront at runtime. Eq. (3) is valid only if the following two conditions are satisfied. One is that all thread computations and data load amounts are consistent. The other one is that the hardware resources required for activated wavefront do not exceed the limit in the CU. Note that for GEMM with different precision, threads have different requirements for computing resources (VGPR, SGPR, LDS) during the computation process. Therefore, for matrices with different precision, the values of $VGPR_{used}$, $SGPR_{used}$, and LDS_{used} in Eqs. (6)–(8) above are different. This will affect the number of activated wavefronts.

4.2.2. Tiling fine-tuning

For batch GEMM, an initial tiling scheme is first assigned to solve the problem of switching between contexts and low hardware resource utilization caused by the matrix's variable scale. Then, the tiling scheme is adjusted according to the TLP estimation of batch GEMM and the hardware architecture of GPU, and finally, the best tiling scheme is obtained. In the first stage, the tile size chosen by each GEMM according to the dimensions of the matrix should meet the following conditions:

$$\begin{cases} T_{mi} \le M_i \text{ and } M_i \ mod \ T_{mi} = 0 \\ T_{ni} \le N_i \text{ and } N_i \ mod \ T_{ni} = 0 \\ T_{ki} \le K_i \text{ and } K_i \ mod \ T_{ki} = 0 \end{cases} \tag{10}$$

where T_{mi} and T_{ni} represent the size of the tile dimension corresponding to the tiling scheme, and T_{ki} is the sub-tile size along the dimension of K. There are two issues. (1) After the first phase, batch GEMM is only an "initial scheme" that cannot achieve optimal parallel computing efficiency. (2) Due to the variability of matrix size in batch GEMM, one or several items of B_{size}, M, N, and K values may be particularly small in batch GEMM, which is called an extreme GEMM case. In this case, the "initial scheme" cannot get enough tiles, which will make some CU in an idle state, resulting in a waste of GPU computing power.

To solve these problems, the "initial scheme" is adjusted reasonably and efficiently in the second stage. For the larger-size matrix, smaller tiles are used to segment, and the number of tiles is increased by reducing the tile size to avoid CU being idle. The details are as follows: for a GEMM, given an appropriate "initial scheme", to avoid the waste of GPU hardware resources, some larger GEMMs are cut with smaller tiles to ensure that the number of tiles is sufficient. For example, for tiles whose initial value is "64 * 64", tiles with "32 * 32" are used for segmentation. As a result, the number of tiles increases as the tile size decreases. This fine-tuning approach ensures that the CU is not idle by increasing the utilization of hardware resources at the expense of intra-tile parallelism.

Algorithm 1 The Tiling algorithm.

```
Initialize TLP_threshold, TLP=0, total_workgroup=0, total_wavefront = 0;
for i = 0 to B_size − 1 do
    Calculate T_mi, T_ni according to equation (10);
    total_workgroup+ = (M_i * N_i)/(T_mi * T_ni);
end for
TLP_new = φ(total_workgroup) × T_wavefront;
Tile[size] represent to "large" to "small";
while ( TLP_new >= TLP_threshold ) do
    for j = 0 to B_size − 1 do
        if Tile[j] is "large" then
            Set Tile[j] is "medium-large";
        else if Tile[j] is "medium-large" then
            Set Tile[j] is "medium";
        else if Tile[j] is "medium" then
            Set Tile[j] is "small-medium";
        else if Tile[j] is "small-medium" then
            Set Tile[j] is "small";
        end if
        total_workgroup+ = (M_j * N_j)/(T_mj * T_nj);
    end for
    TLP_new = φ(total_workgroup) × T_wavefront;
end while
```

$TLP_{threshold}$ is used as a threshold to ensure parallelism among multiple tiles in fine-tuning phase. Note that $TLP_{threshold}$ has an important influence on the selection of tiling scheme for different hardware architectures. As a measure, the TLP values of the batch GEMM vary according to the different tiling schemes. The setting of the $TLP_{threshold}$ value is related to the architecture of the GPU because it uses the number of wavefront and the number of threads in the wavefront to measure the parallelism of the tiling scheme. The hardware resources and the maximum number of wavefronts supported by each CU are diverse, so corresponding $TLP_{threshold}$ should be set for different GPU architectures.

The specific process of selecting a tiling scheme for batch GEMM is given in **Algorithm 1**: (1) when batch GEMM is given, an "initial scheme" is obtained according to Eq. (10). (2) The TLP of this scheme is calculated according to the given batch GEMM and tiling scheme. (3) Compare the TLP of the current tiling scheme with the $TLP_{threshold}$. If the TLP is not reached, the fine-tuning operation will be performed, and the current tiling scheme will be changed and then returned to step (2). If the current TLP is greater than or equal to the threshold, go to step (4). (4) The batch GEMM is calculated according to the final tiling scheme. In the above procedures, the TLP is used as an evaluation criterion to measure the effectiveness of the tiling scheme on the batch GEMM. If the threshold is not reached, fine-tuning is used to adjust and improve the utilization of GPU hardware resources. The optimal tiling scheme can be obtained to ensure an optimal implementation at the GEMM and workgroup level. After the final tiling scheme, the multi-thread kernel is calculated based on the tile size so that the wavefront and work-item levels can achieve a "workload balance" state.

The proposed method is based on the GPU platforms of AMD and NVIDIA for implementation. The hardware characteristics of the GPU platform can also significantly impact GEMM performance. For example, in AMD and NVIDIA platforms, threads are based on wavefront and warp as the basic execution units containing 64 and 32 threads, respectively. The number of threads in the kernel needs to be an integer multiple of the number of threads in wavefront and warp to improve kernel occupancy. Meanwhile, the size of registers and shared memory

Table 4
The configuration of platforms for evaluation.

Platform setup	AMD-platform	NVIDIA-platform
CPU	EPYC 7763	Platinum 8358
GPU	MI210	A800
OS	Ubuntu 20.04	Ubuntu 20.04
ROCm/CUDA	ROCm 5.6	CUDA 12.0

Table 5
The configuration of GPUs for evaluation.

Name	MI210	A800
Architecture	CDNA 2.0	Ampere
Core	1700 MHz	1410 MHz
Caches	L1 16 KB (per CU) L2 16 MB	L1 192 KB (per SM) L2 40 MB
Memory	64 GB 3.2 Gbps HBM2	80 GB 2.4 Gbps HBM2
Bandwidth	1.6 TB/s	2.04 TB/s

can affect parameter settings during implementation based on different hardware architectures. Based on this difference, the proposed method considers parallelism at the wavefront or warp level when performing matrix segmentation on two GPU platforms. In this way, the proposed method can flexibly select tiling schemes based on the hardware characteristics of the GPU to achieve optimal performance. In this way, the proposed method can avoid exceeding the maximum register limit and prevent data overflow, which improves its applicability for various hardware architectures.

5. Evaluation

5.1. Setup

Experiment platform and matrix generation. The overall configuration of the experimental platform and the details of the two GPUs are shown in Tables 4 and 5, respectively. To ensure the irregularity and variability of the input matrix, the GEMM size parameters M, N, and K are randomly generated within corresponding ranges ($[Min, Max\_M(N)]$ and $[Min, Max\_K]$). Max_M, Max_N, and Max_K represent the upper bounds of M, N, and K, respectively. The lower bound for each experiment is denoted uniformly by Min. In this paper, the value of Min is set to 16. For example, Max_M(N) = 512 and Max_K = 128 indicate that the range of matrix dimensions is $M \in [16, 512]$, $N \in [16, 512]$ and $K \in [16, 128]$. Thus, multiple sets of matrix dimension ranges can be obtained, and the parameters needed for GEMM generation are chosen from the different value ranges by random selection.

Comparison method. First, for the two GPU experimental platforms, the default GEMM processing methods rocBLAS [6] and cuBLAS [7] provided by the respective GPU manufacturers are chosen as the basic comparison methods to demonstrate the effectiveness of the proposed method. Since these methods do not support the way of batch invocation, in this paper, rocBLAS and cuBLAS compute batch GEMM in a loop manner. No stream operations are used during the computation. Meanwhile, we also compared the CUTLASS [23], which supports batch GEMM based on sorting and built-in tiles. We then compare with MAGMA [8] supported by the University of Tennessee ICL Lab, which only extends the $grid.z$ to support batch GEMM but does not have a fine-grained optimization strategy. The MAGMA comparison experiments were run on two GPU platforms. Meanwhile, to show the advancement of our proposed method, we compare with the state-of-the-art methods such as Wang [36] and Li [21] on their respective platforms. All of the above methods perform a warp-up operation to eliminate the effect of the first kernel boot.

Evaluation criteria. In the following experiments, there are 12 sets of GEMM dimension ranges. The experiments with batch sizes 8, 16, 32, 64, 128, and 256 were run continuously for ten epochs under each set of value ranges. The experimental results were represented by the average value of GFLOPS (Giga Floating-point Operations Per Second), which is calculated as:

$$GFLOPS = \frac{\sum_{i=0}^{n-1} 2(M_i \times N_i \times K_i)}{total\_time * 1.0e9} \tag{11}$$

where M_i, N_i and K_i represent the matrix dimension of the ith GEMM, and $total\_time$ represents the running time on this GPU, n represents batch sizes. For simplicity, the experimental data is represented as single-precision floating-point data and the storage format is based on the row-first format. The experimental results are averaged over 10 consecutive runs. The final experimental results were rounded to preserve two decimal places.

5.2. Speed up

In the two platforms, we first compare with the default methods rocBLAS and cuBLAS. These two methods do not support batch irregular GEMMs; we convert batch GEMMs into multiple single GEMMs and compute the results. The specific experimental results are shown in Figs. 4–5. Figs. 4–5 show that the proposed method achieves 5.09× and 7.18× average speedup compared to rocBLAS and cuBLAS. This result is primarily due to the fact that this method does not support GEMMs of different scales when computing batch GEMMs, so it can only compute one GEMM simultaneously. When faced with a small matrix, the computational resources of the GPU cannot be fully utilized due to the cost of context switching between multiple GEMMs. As the batch size gradually increases, the advantage of the proposed method becomes more evident. This shows that for batch and irregular GEMMs, rocBLAS and cuBLAS are at a disadvantage in terms of computational efficiency and switching between instances. Meanwhile, we also compare CUTLASS, which handles batch GEMM, using sorting to solve the problem of significant workload differences between multiple matrix multiplications. Fig. 5 shows that the proposed method has a 4.64× speedup, which is because CUTLASS's built-in tiles are unsuitable when the matrix dimensions are small. Therefore, the proposed method performs better acceleration than CUTLASS for batch, irregular, and small-size matrix multiplication. We then perform a detailed comparison and analysis of the experimental performance based on MAGMA. The proposed method has 4.37× and 3.36× speed improvement compared to MAGMA. Figs. 4–5 show that the advantage of our method becomes more pronounced as the batch size increases. This is because MAGMA only uses the largest GEMM size in the batch GEMM to set $grid.x$. Due to the irregularity of the matrix size, a large number of computational resources in the grid will be idle. The proposed method, in this case, employs fine-grained filtering operations to ensure further efficient utilization of computational resources, which is more evident when the difference between matrix dimensions is significant.

As shown in Fig. 4, the proposed method achieves an average 1.88× speedup performance compared to Wang. It is noted that the advantage of the proposed method is more pronounced when Max_K and Max_M are small. For example, in the case of ($Max\_M(N) = 128$, $Max\_K = 128$), the average speedup can reach 1.95×. This is mainly due to the fact that when the dimension of matrix is small, there are not enough tiles to cover the time consumption of data loading in the wavefront, which is more pronounced in workgroups with heavy loads. The proposed method adjusts the wavefront workload corresponding to the tiles through a multi-thread kernel and ensures consistent computation and data loading by different workgroups. At the same time, it has also shown that the state of load and computation balancing between wavefronts is more conducive to improving the efficiency of GPU parallel computing. In the NVIDIA platform, Fig. 5 shows that the proposed method has average 1.94× speedup performance compared to Li. The advantage of the proposed method becomes clearer as the batch size increases. There are two reasons for this speedup performance :

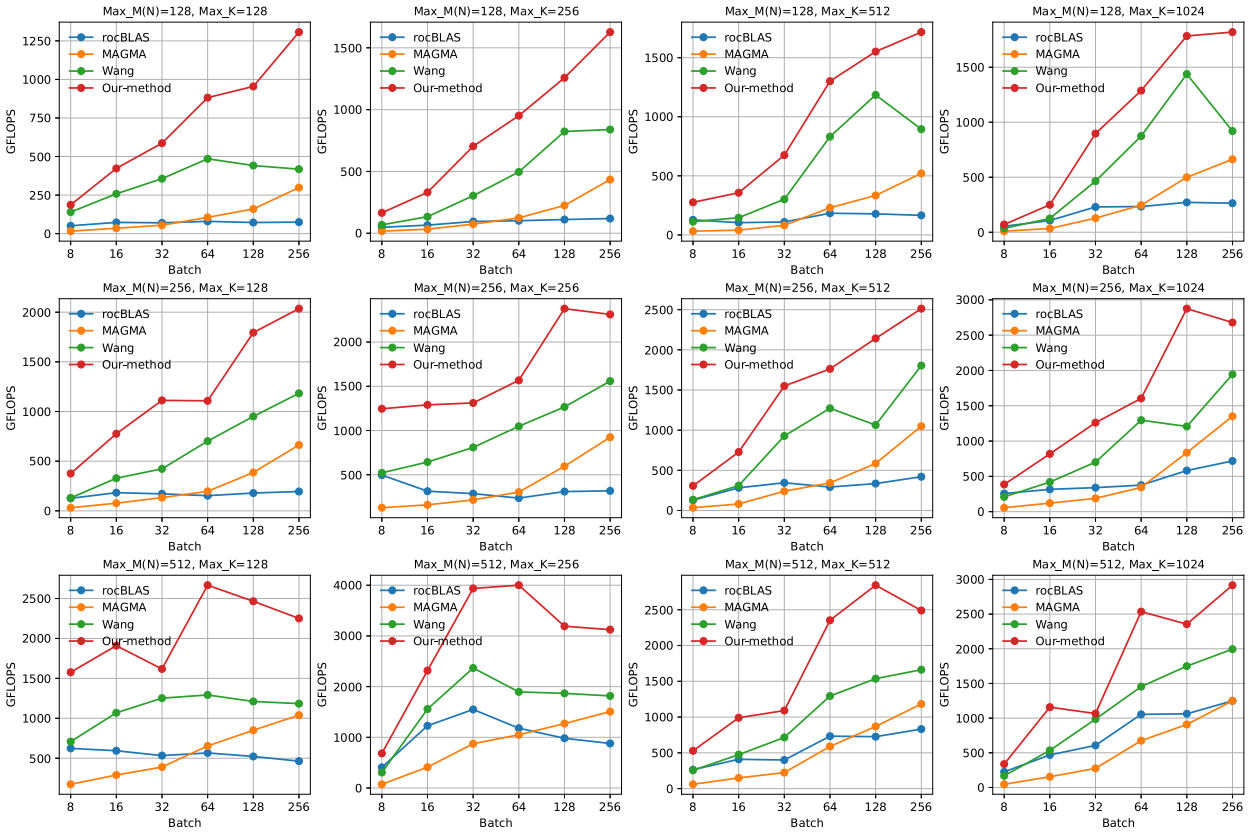


Fig. 4. The comparative results on MI210. (5.09×, 4.37×, 1.88× speedup over rocBLAS, MAGMA, Wang).

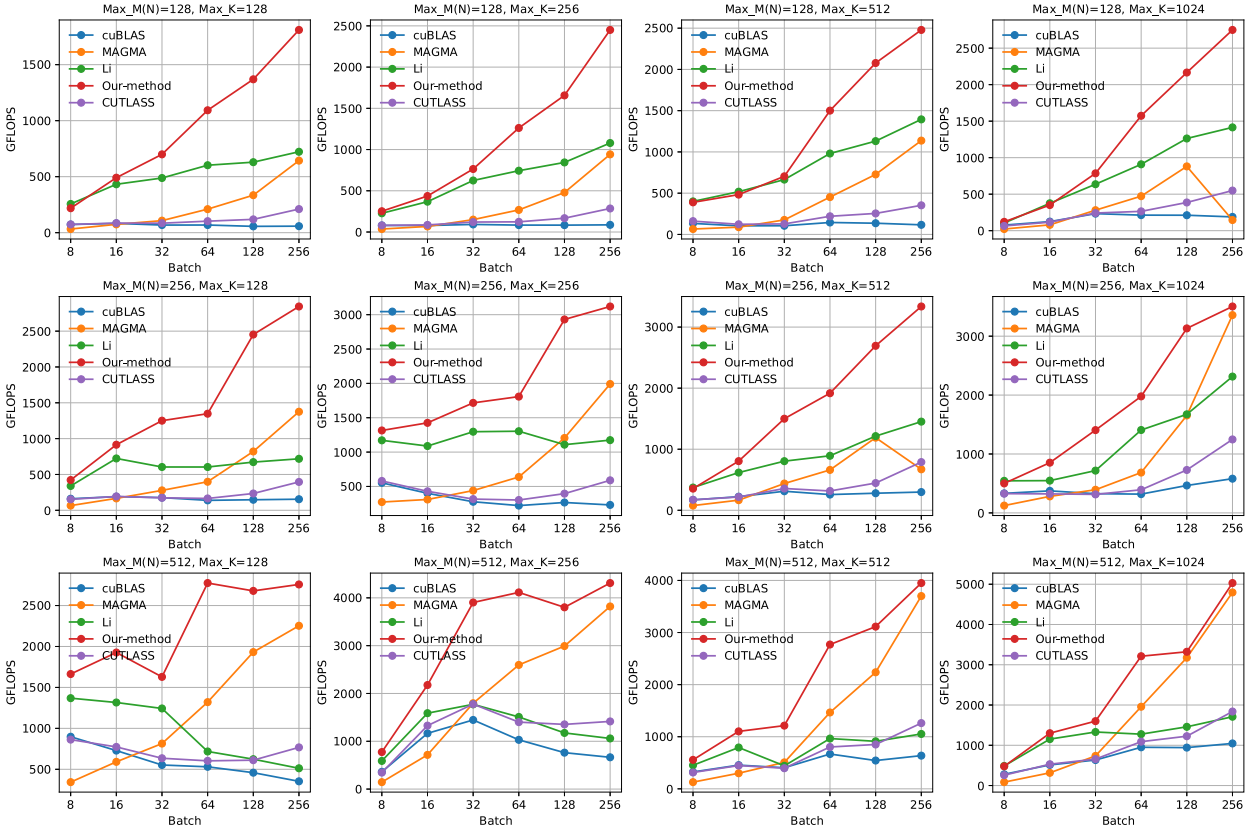
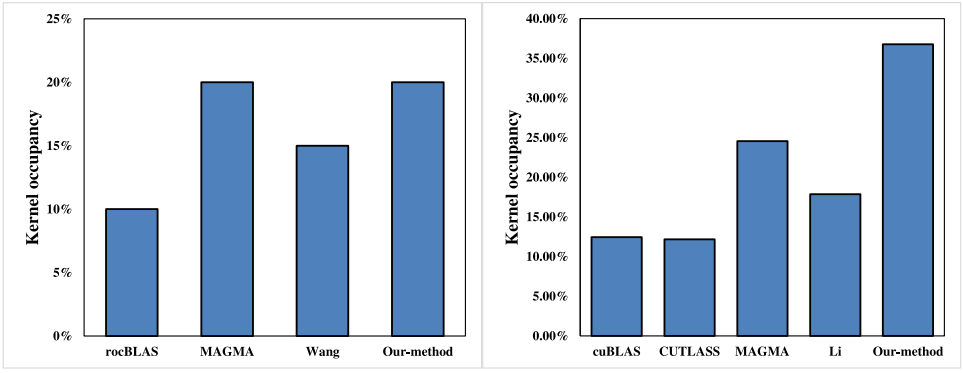
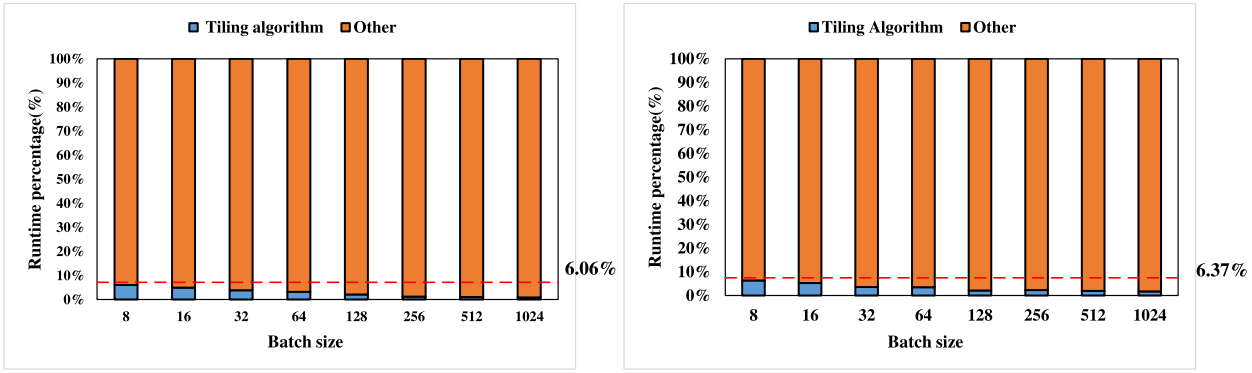


Fig. 5. The comparative results with on A800. (7.18×, 4.64×, 3.63×, 1.94× speedup over cuBLAS, CUTLASS, MAGMA, Li).



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

Fig. 6. The kernel occupancy on two GPU platforms.



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

Fig. 7. The time overhead of tiling algorithm.

(1) Li et al. used batching to balance the workload among different blocks but did not consider the difference between the workload of threads in different tiles. (2) When selecting the tiling scheme, the TLP is calculated only by considering the block, and the fine-grained warp level is neglected, which leads to the inaccurate calculation of TLP. The proposed method adjusts the wavefront workload corresponding to the tiles through a multi-thread kernel and ensures consistent computation and data loading by different workgroups. At the same time, it has also shown that the state of load and computation balancing between wavefronts is more conducive to improving the efficiency of GPU parallel computing.

5.3. Kernel occupancy

To explore the difference between the proposed method and the comparison methods in terms of GPU resource utilization, we present the kernel occupancy of the various methods on two GPU platforms. The formula for kernel occupancy can be expressed as:

$$\text{kernel occupancy} = \frac{Num\_actived}{Num\_total} \tag{12}$$

To obtain more accurate performance metrics, we utilize Omniperf[5] and Nsight[6] commands, profiling tools provided by AMD and NVIDIA, to evaluate the resource utilization of the kernel during the execution process. The kernel occupancy has distinct interpretations owing to the distinctions in GPU architecture between AMD MI210 and NVIDIA A800. On the AMD platform, $Num\_actived$ is the number of activated wavefronts and Num_total is the theoretical number of wavefronts that CU can execute simultaneously. $Num\_actived$ and Num_total represent the number of warps in activation and the number of warps that are theoretically parallelizable simultaneously in the NVIDIA platform.

The results of the experiment are shown in Fig. 6. By comparing rocBLAS and cuBLAS, it can be seen that the proposed method has a clear advantage in the case of batch GEMM. The proposed method is also in the best position compared to the other methods (CUTLASS, MAGMA, Wang, Li), showing high efficiency in terms of utilization of GPU resources. As shown in Fig. 6, the proposed method consistently maintains the optimal kernel occupancy on both GPU platforms, which indicates that the proposed method can better exploit the computing power of the GPU.

5.4. The overhead of tiling algorithm

This section presents the proportion of the runtime that is taken up by the tiling algorithm when executing the proposed method on two different GPU platforms with various batch sizes. The experimental results are presented in Fig. 7. From Fig. 7, it is evident that the tiling algorithm's runtime percentage decreases as the batch size increases. When batch size is 8, the runtime of the tiling algorithm on the two GPU platforms is 6.06% and 6.37%, respectively. As the batch size increases, more and more GEMMs are executed on the GPU, and the execution time of these GEMMs on the GPU side takes up most of the time, resulting in a smaller runtime portion of the tiling algorithm. For example, with a batch size is 1024, the tiling algorithm takes less than 1% of the runtime. The experimental results on two GPUs indicate that the time overhead of the tiling algorithm in the batch GEMM execution process is negligible, especially when the batch size is large. In real-world scenarios such as deep learning, where a large number of

[5] https://github.com/ROCm/omniperf
[6] https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

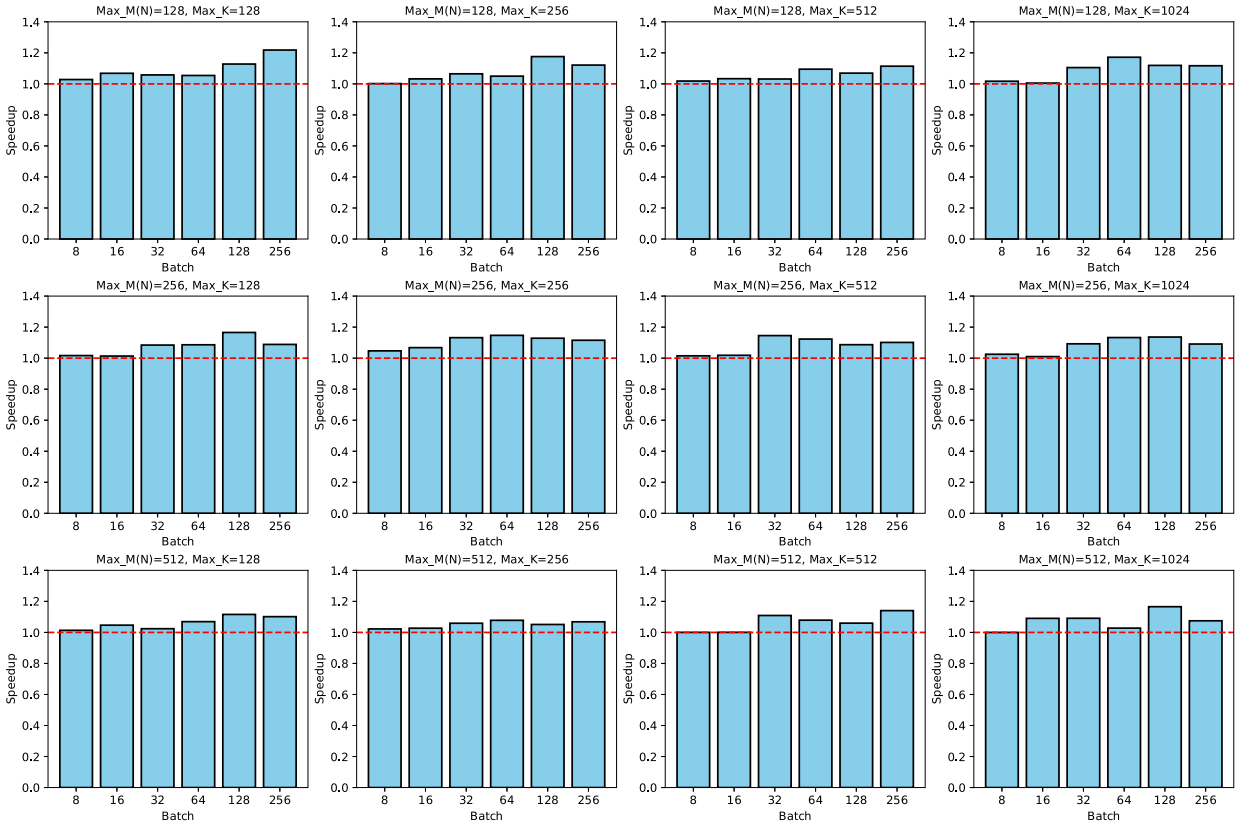


Fig. 8. The performance improvement of the proposed TLP on MI210. (1.077× average speedup).

GEMM operations are often required, the tiling algorithm will have less overhead in the execution process.

5.5. The performance benefits of the proposed TLP

This section presents the comparative experimental results on two GPU platforms to provide a more detailed evaluation of the proposed TLP. The detailed experimental results are shown in Figs. 8–9. From Figs. 8–9, it is clear that the proposed TLP performs better overall than traditional TLP. The proposed methods have a speedup of 1.077× and 1.085× on MI210 and A800, respectively. From Fig. 8, the proposed method significantly improves performance when the batch size is larger. For example, on MI210, the proposed method has an average speedup of 1.04× when batch size <= 16. When batch size >= 32, the proposed method can improve performance by 1.10×. The performance improvement gap is because when the batch size and matrix dimension are small, it is difficult to utilize hardware resources fully. When there are a large number of tiles, the proposed TLP can more accurately evaluate the thread's workload and select the optimal tiling scheme. The same performance trend is also reflected in the A800 platform. On A800, the proposed TLP has performance improvements of 1.04× and 1.11× when batch size <= 16 and batch size >= 32, respectively. The effectiveness of the proposed TLP can be further demonstrated through comparative experiment results on two GPU platforms.

5.6. The latency

This section compares kernel latency on two GPU platforms to provide a more detailed evaluation of the proposed method. We measured kernel latency with different batch sizes in the comparative experiment. The detailed experimental results are shown in Fig. 10. On MI210, the proposed method has a latency reduction of 3.87×, 4.53×, and 1.62× compared to rocBLAS, MAGMA, and Wang, respectively. The proposed method has the lowest latency performance on MI210, indicating higher computational efficiency and can effectively reduce latency. On A800, the proposed method showed performance improvements of 3.02×, 2.59×, 2.45×, and 1.89× compared to cuBLAS, MAGMA, CUTLASS, and Li, respectively. Fig. 10 shows that as the batch size gradually increases, the kernel latency increases on both GPU platforms. rocBLAS and cuBLAS have the highest latency as the batch size increases. This phenomenon is because the traditional loop scheduling method significantly increases latency consumption due to context switching between kernels when the batch size is large. From Fig. 10, it can be seen that some methods exhibit different latency performances at various batch sizes. For example, when batch size <= 16, MAGMA has the highest latency performance on two GPU platforms. When the batch size is large, its computational performance improves, indicating that the MAGMA performs better when there are many matrices. The experimental results on two platforms show that the proposed method has the lowest latency under various batch sizes, indicating better performance and broad applicability.

5.7. The improved performance on inception layers of CNN

Modern CNN model architectures often have multiple branches to capture features at different scales. Convolution operations of different scales in each branch can be represented as batch GEMM operations with various dimensions, e.g. GoogleNet [13], DenseNet [50], SqueezeNet [12], etc. To demonstrate the effectiveness of the proposed method in real-world scenarios, we use various Inception module as a typical application to perform the forward computation process on two GPU platforms. The Inception module involves a large number of irregular, small-size GEMM operations. The deep learning frameworks

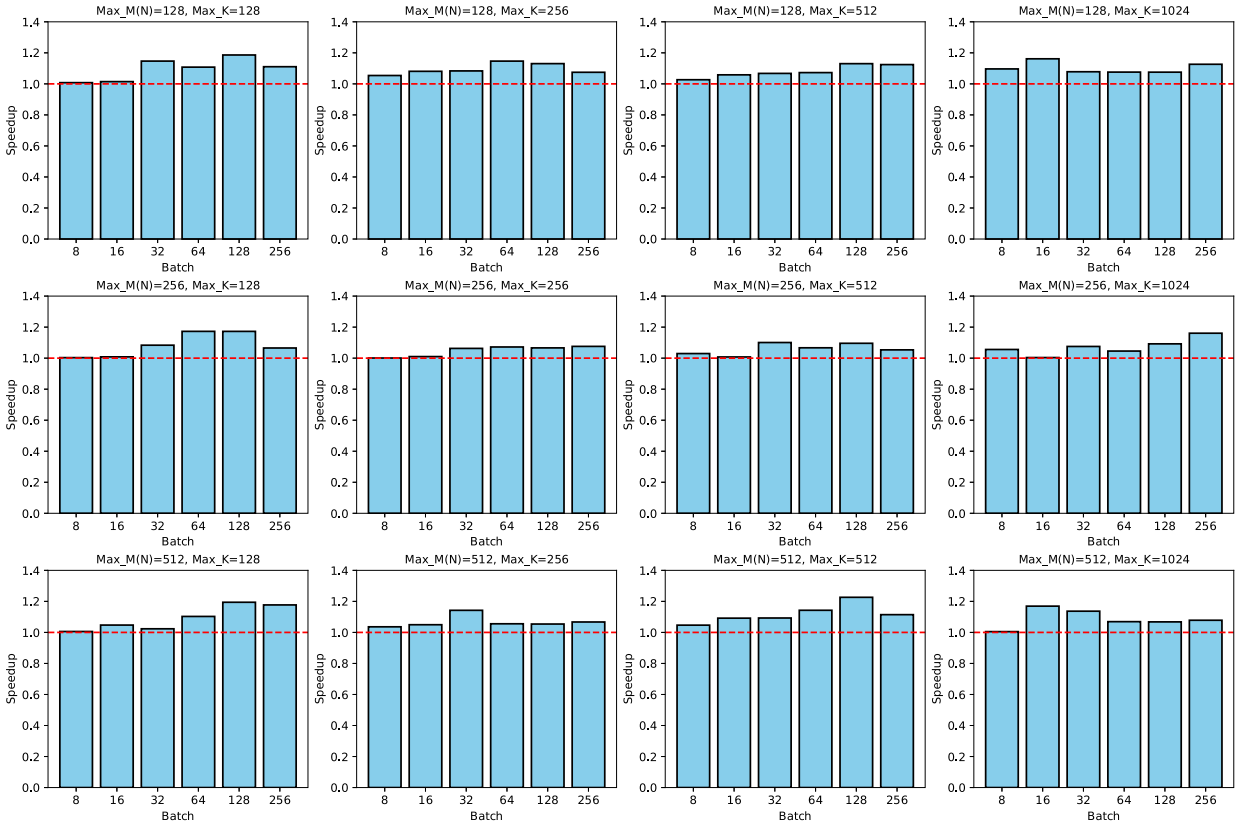


Fig. 9. The performance improvement of the proposed TLP on A800. (1.085× average speedup).

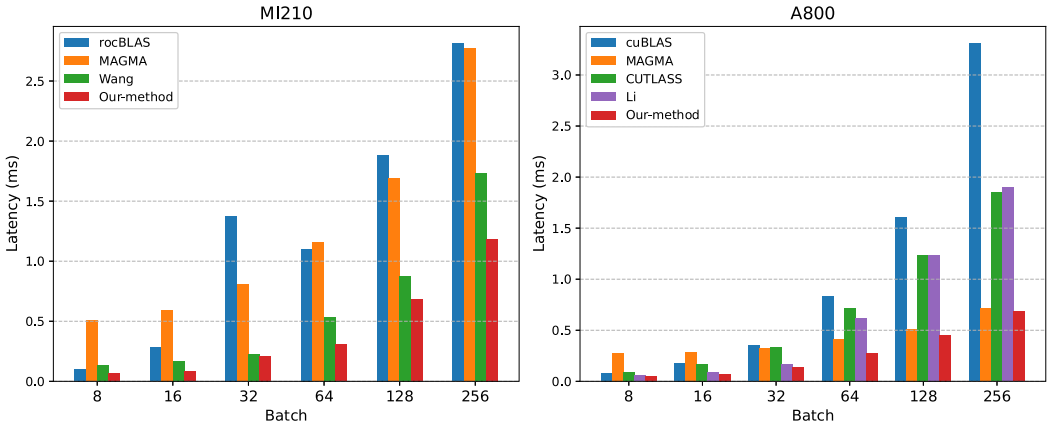


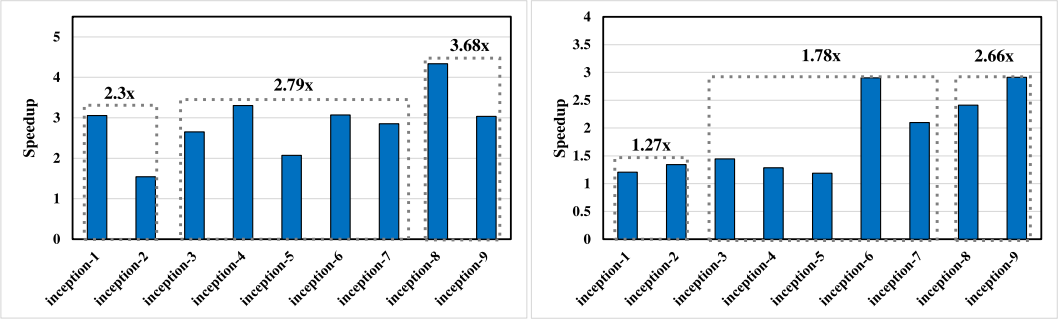
Fig. 10. The latency performance of the kernel on two GPU platforms.

MIOpen[7] and cuDNN[8] are used as benchmark implementations on both GPU platforms. In this section, we select several commonly used Inception modules to evaluate the proposed method's speedup performance. The GEMM sizes in Inception modules are shown in Table 6. Fig. 11 shows the speedup performance of the proposed method in each Inception module. As shown in Fig. 11, the average speedups are 2.88× and 1.87× respectively. The gray boxes represent the average speedup ratios of the different Inception modules in Fig. 11. The experimental results suggest that the Inception 8–9 series has the highest average speedup ratio (3.68× and 2.66× respectively) among the Inception modules, because Inception 8–9 has more matrix shapes compared to the other Inception module, and the dimensions of these matrices are smaller than the former two. Finally, the proposed method has been proven to significantly accelerate CNN models with various branch structures on two different GPU platforms, particularly in scenarios involving multiple branches, irregular shapes, and small dimensions.

6. Conclusion

In this paper, we propose a load-balanced batch GEMM acceleration method for the problem of low parallel computing efficiency and poor hardware resource utilization in batch, irregular, and variable matrix multiplication scenarios. The kernel occupancy and hardware resource utilization can be effectively improved by a multi-thread kernel design that balances the computational and data load in the work-item. A novel approach to TLP computation is devised, where the parallelism of

[7] https://github.com/ROCm/MIOpen
[8] https://github.com/NVIDIA/cudnn-frontend



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

Fig. 11. The speedup performance on Inception layers.

Table 6
The size of GEMM in various Inception modules.

Inception module	GEMM size (M ×N × K)
Inception-1	784 × 96 × 192, 784 × 64 × 192, 784 × 32 × 192, 784 × 16 × 192
Inception-2	784 × 64 × 192, 784 × 32 × 192, 784 × 128 × 192
Inception-3	196 × 192 × 192, 196 × 16 × 192, 196 × 96 × 192, 196 × 64 × 192
Inception-4	196 × 64 × 192, 196 × 24 × 192, 196 × 160 × 192
Inception-5	196 × 64 × 192, 196 × 128 × 192, 196 × 24 × 192
Inception-6	196 × 112 × 192, 196 × 144 × 192, 196 × 32 × 192, 196 × 64 × 192
Inception-7	196 × 256 × 192, 196 × 160 × 192, 196 × 128 × 192
Inception-8	49 × 160 × 192, 49 × 128 × 192, 49 × 256 × 192, 49 × 160 × 192, 49 × 32 × 192
Inception-9	49 × 192 × 192, 49 × 128 × 192, 49 × 384 × 192, 49 × 192 × 192, 49 × 48 × 192

the tiling scheme is measured by the number of activated wavefronts. This approach allows the optimal tiling scheme to be selected based on different GPU architectures. Experiments are conducted on two GPU platforms to validate the effectiveness and progress of our proposed method.

Future work includes exploring batch GEMM with various precision performances. With the development of Transformer-based, many GEMM operations are involved in the training and inference process of Large Language Models (LLMs), which often have lower accuracy, such as FP16, FP8, etc. For example, quantized LLMs often involve GEMM operations where the weight matrices and activation values have different precisions, e.g. W4A16, W8A8. More complex precisions and storage formats pose challenges to the performance of GEMM operations.

CRediT authorship contribution statement

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

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 (2024A1515010204) and the Technological Research Project of Southern Power Grid Company (ZBKJXM20232483).

Data availability

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

References

[1] 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.
[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] J. Fornt, P. Fontova-Musté, M. Caro, J. Abella, F. Moll, J. Altet, C. Studer, An energy-efficient gemm-based convolution accelerator with on-the-fly im2col, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 31 (11) (2023) 1874–1878.
[4] H. Kim, W.J. Song, Las: locality-aware scheduling for GEMM-accelerated convolutions in GPUs, IEEE Trans. Parallel Distrib. Syst. 34 (5) (2023) 1479–1494.
[5] W. Yang, J. Fang, D. Dong, X. Su, Z. Wang, Optimizing full-spectrum matrix multiplications on ARMv8 multi-core CPUs, IEEE Trans. Parallel Distrib. Syst. (2024).
[6] AMD, Next generation BLAS implementation for ROCm platform, 2024, https://github.com/ROCm/rocBLAS.
[7] B. Tuomanen, Hands-On GPU Programming with Python and CUDA: Explore High-Performance Parallel Computing with CUDA, Packt Publishing Ltd, 2018.
[8] ICL, Matrix algebra for GPU and multicore architectures, 2024, https://icl.utk.edu/magma/.
[9] T. Faingnaert, T. Besard, B. De Sutter, Flexible performant GEMM kernels on GPUs, IEEE Trans. Parallel Distrib. Syst. 33 (9) (2021) 2230–2248.
[10] W.S. Moses, I.R. Ivanov, J. Domke, T. Endo, J. Doerfert, O. Zinenko, High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs, in: Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 2023, pp. 119–134.
[11] H. Kim, H. Nam, W. Jung, J. Lee, Performance analysis of CNN frameworks for GPUs, in: 2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2017, pp. 55–64.
[12] F.N. Iandola, S. Han, M.W. Moskewicz, K. Ashraf, W.J. Dally, K. Keutzer, SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size, 2016, arXiv preprint arXiv:1602.07360.
[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.
[14] G. Pant, D. Yadav, A. Gaur, ResNeXt convolution neural network topology-based deep learning model for identification and classification of pediastrum, Algal Res. 48 (2020) 101932.
[15] S. Barrachina, M.F. Dolz, P. San Juan, E.S. Quintana-Ortí, Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors, J. Parallel Distrib. Comput. 167 (2022) 240–254.

[16] S. Rajbhandari, Y. He, O. Ruwase, M. Carbin, T. Chilimbi, Optimizing cnns on multicores for scalability, performance and goodput, ACM SIGARCH Comput. Archit. News 45 (1) (2017) 267–280.
[17] C. Rivera, J. Chen, N. Xiong, S.L. Song, D. Tao, Ism2: Optimizing irregular-shaped matrix-matrix multiplication on gpus, 2020, arXiv preprint arXiv:2002.03258.
[18] K. Matsumoto, N. Nakasato, S.G. Sedukhin, Performance tuning of matrix multiplication in opencl on different gpus and CPUs, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, IEEE, 2012, pp. 396–405.
[19] G.E. Moon, H. Kwon, G. Jeong, P. Chatarasi, S. Rajamanickam, T. Krishna, Evaluating spatial accelerator architectures with tiled matrix-matrix multiplication, IEEE Trans. Parallel Distrib. Syst. 33 (4) (2021) 1002–1014.
[20] Q. Han, H. Yang, M. Dun, Z. Luan, L. Gan, G. Yang, D. Qian, Towards efficient tile low-rank GEMM computation on sunway many-core processors, J. Supercomput. 77 (5) (2021) 4533–4564.
[21] X. Li, Y. Liang, S. Yan, L. Jia, Y. Li, A coordinated tiling and batching framework for efficient GEMM on GPUs, in: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, 2019, pp. 229–241.
[22] P. Tillet, D. Cox, Input-aware auto-tuning of compute-bound HPC kernels, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017, pp. 1–12.
[23] NVIDIA, CUDA templates for linear algebra subroutines, 2024, https://github.com/NVIDIA/cutlass.
[24] J. Huang, C.D. Yu, R.A.v.d. Geijn, Strassen's algorithm reloaded on GPUs, ACM Trans. Math. Softw. 46 (1) (2020) 1–22.
[25] B. Boyer, J.-G. Dumas, C. Pernet, W. Zhou, Memory efficient scheduling of strassen-winograd's matrix multiplication algorithm, in: Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation, 2009, pp. 55–62.
[26] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F.J. R Ruiz, J. Schrittwieser, G. Swirszcz, et al., Discovering faster matrix multiplication algorithms with reinforcement learning, Nature 610 (7930) (2022) 47–53.
[27] G. Xiao, C. Yin, T. Zhou, X. Li, Y. Chen, K. Li, A survey of accelerating parallel sparse linear algebra, ACM Comput. Surv. 56 (1) (2023) 1–38.
[28] Y. Chen, G. Xiao, K. Li, F. Piccialli, A.Y. Zomaya, fgSpMSpV: A fine-grained parallel SpMSpV framework on HPC platforms, ACM Trans. Parallel Comput. 9 (2) (2022) 1–29.
[29] Y. Chen, G. Xiao, W. Yang, Optimizing partitioned CSR-based SpGEMM on the sunway TaihuLight, Neural Comput. Appl. 32 (10) (2020) 5571–5582.
[30] Y. Chen, K. Li, W. Yang, G. Xiao, X. Xie, T. Li, Performance-aware model for sparse matrix-matrix multiplication on the sunway taihulight supercomputer, IEEE Trans. Parallel Distrib. Syst. 30 (4) (2018) 923–938.
[31] G. Xiao, K. Li, Y. Chen, W. He, A.Y. Zomaya, T. Li, Caspmv: A customized and accelerative spmv framework for the sunway taihulight, IEEE Trans. Parallel Distrib. Syst. 32 (1) (2019) 131–146.
[32] G. Xiao, C. Yin, Y. Chen, M. Duan, K. Li, Efficient utilization of multi-threading parallelism on heterogeneous systems for sparse tensor contraction, IEEE Trans. Parallel Distrib. Syst. (2024).
[33] D.E. Tanner, Tensile: Auto-tuning gemm gpu assembly for all problem sizes, in: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, IEEE, 2018, pp. 1066–1075.
[34] S. Wang, FlexGEMM: A flexible micro-kernel generation framework, in: Proceedings of the 5th International Conference on Computer Information and Big Data Applications, 2024, pp. 164–170.
[35] G. Alaejos, A. Castelló, H. Martínez, P. Alonso-Jordá, F.D. Igual, E.S. Quintana-Ortí, Micro-kernels for portable and efficient matrix multiplication in deep learning, J. Supercomput. 79 (7) (2023) 8124–8147.
[36] R. Wang, Z. Yang, H. Xu, L. Lu, A high-performance batched matrix multiplication framework for gpus under unbalanced input distribution, J. Supercomput. 78 (2) (2022) 1741–1758.
[37] Y. Zhang, Y. Wang, Z. Mo, Y. Zhou, T. Sun, G. Xu, C. Xing, L. Yang, Accelerating small matrix multiplications by adaptive batching strategy on GPU, in: 2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application, HPCC/DSS/SmartCity/DependSys, IEEE, 2022, pp. 882–887.
[38] A. Abdelfattah, S. Tomov, J. Dongarra, Matrix multiplication on batches of small matrices in half and half-complex precisions, J. Parallel Distrib. Comput. 145 (2020) 188–201.
[39] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Novel HPC techniques to batch execution of many variable size BLAS computations on GPUs, in: Proceedings of the International Conference on Supercomputing, 2017, pp. 1–10.
[40] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Performance, design, and autotuning of batched GEMM for GPUs, in: High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings, Springer, 2016, pp. 21–38.
[41] A. Li, G.-J. van den Braak, H. Corporaal, A. Kumar, Fine-grained synchronizations and dataflow programming on GPUs, in: Proceedings of the 29th ACM on International Conference on Supercomputing, 2015, pp. 109–118.
[42] J. Li, H. Ye, S. Tian, X. Li, J. Zhang, A fine-grained prefetching scheme for DGEMM kernels on GPU with auto-tuning compatibility, in: 2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2022, pp. 863–874.
[43] Z. Yang, L. Lu, R. Wang, A batched GEMM optimization framework for deep learning, J. Supercomput. 78 (11) (2022) 13393–13408.
[44] H. Mei, H. Qu, J. Sun, Y. Gao, H. Lin, G. Sun, GPU occupancy prediction of deep learning models using graph neural network, in: 2023 IEEE International Conference on Cluster Computing, CLUSTER, IEEE, 2023, pp. 318–329.
[45] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, J. Dongarra, Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices, Parallel Comput. 81 (2019) 1–21.
[46] 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.
[47] 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.
[48] G. Shobaki, A. Kerbow, S. Mekhanoshin, Optimizing occupancy and ILP on the GPU using a combinatorial approach, in: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, 2020, pp. 133–144.
[49] A.B. Hayes, L. Li, D. Chavarría-Miranda, S.L. Song, E.Z. Zhang, Orion: A framework for gpu occupancy tuning, in: Proceedings of the 17th International Middleware Conference, 2016, pp. 1–13.
[50] G. Huang, S. Liu, L. Van der Maaten, K.Q. Weinberger, Condensenet: An efficient densenet using learned group convolutions, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 2752–2761.