A Coordinated Tiling and Batching Framework for Efficient GEMM on GPUs

Xiuhong Li[1], Yun Liang[1,*], Shengen Yan[2], Liancheng Jia[1], Yinghan Li[2]
[1] Center for Energy-efficient Computing and Applications, School of EECS, Peking University
[2] SenseTime Incorporation
{lixiuhong,ericlyun,jlc}@pku.edu.cn,{yanshengen,liyinghan}@sensetime.com

Abstract

General matrix multiplication (GEMM) plays a paramount role in a broad range of domains such as deep learning, scientific computing, and image processing. The primary optimization method is to partition the matrix into many tiles and exploit the parallelism within and between tiles. The tiling hierarchy closely mirrors the thread hierarchy on GPUs. In practice, GPUs can fully unleash its computing power only when the matrix size is large and there are sufficient number of tiles and workload for each tile. However, in many real-world applications especially deep learning domain, the matrix size is small. To this end, prior work proposes batched GEMM to process a group of small independent GEMMs together by designing a single CUDA kernel for all of these GEMMs.

However, the current support for batched GEMM is still rudimentary. Tiling and batching are tightly correlated. A large tile size can increase the data reuse, but it will decrease the thread-level parallelism, which further decrease the optimization space for the batching. A small tile size can increase the thread-level parallelism and then provide larger optimization space for the batching, but at the cost of sacrificing data reuse. In this paper, we propose a coordinated tiling and batching framework for accelerating GEMMs on GPUs. It is a two-phase framework, which consists of a tiling engine and a batching engine to perform efficient batched GEMM on GPUs. Tiling engine partitions the GEMMs into independent tiles and batching engine assigns the tiles to thread blocks. Moreover, we propose a general programming interface for the coordinated tiling and batching solution. Finally, experiment evaluation results on synthetic batched GEMM cases show that our framework can achieve about 1.40X performance speedup on average over the state-of-the-art technique. We also use GoogleNet as a real-world case study and our framework can achieve 1.23X speedup.

CCS Concepts • **Computing methodologies → Massively parallel algorithms**; • **Computer systems organization → Single instruction, multiple data**;

Keywords GEMM, GPGPU, Tiling, Batching

ACM Reference format:
Xiuhong Li[1], Yun Liang[1,*], Shengen Yan[2], Liancheng Jia[1], Yinghan Li[2]. 2019. A Coordinated Tiling and Batching Framework for Efficient GEMM on GPUs. In *Proceedings of 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Washington, DC, USA, February 16–20, 2019 (PPoPP '19),* 13 pages.
https://doi.org/10.1145/3293883.3295734

1 Introduction

General Matrix Multiplication (GEMM) is a matrix multiplication and accumulation routine as follows: $\mathbf{C} = \alpha\mathbf{A}\times\mathbf{B}+\beta\mathbf{C}$, where $\mathbf{A} \in \mathbb{R}^{M\times K}$, $\mathbf{B} \in \mathbb{R}^{K\times N}$ and $\mathbf{C} \in \mathbb{R}^{M\times N}$ are matrices, and α and β are scalars. It is one of the most widely used high-performance kernels in various domains such as deep learning, signal processing, advanced physical analytics, and astrophysics [4]. The wide adoption of GEMM and its huge computation cost have led to a high demand to optimize GEMM for high performance. From the hardware perspective, GPUs have been demonstrated to be able to provide tremendous computation power for accelerating regular applications such as GEMM [21]. Furthermore, novel memory architecture HBM2 and computation engine Tensor Cores have been integrated into the latest NVIDIA Volta GPUs for even higher FP16 GEMM performance [21]. From the software perspective, a variety of optimization techniques from algorithm level [1, 9, 10, 16, 20, 22] to compilation level [5, 11, 26, 31, 33] have been developed. Many optimization efforts have also been incorporated to the widely used GEMM libraries, such as cuBLAS [21], CUTLASS [22], and MAGMA [20] on GPU platforms.

For a single GEMM, a common solution is to partition matrix $\mathbf{C}$ ($M \times N$) into multiple tiles and each thread block is responsible for a single tile. Each tile is independent of other tiles, and the parallelism within and between tiles can be exploited. M and N are closely related to the tile size and the number of tiles. They are crucial for performance as they

*Corresponding author.

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295734

impact the thread-level parallelism (TLP). The workload of a single tile is to accumulate all the sub-matrix multiplication results along the K-dimension. K determines the workload of a tile and has impacts on the instruction-level parallelism (ILP) within a single thread [21]. In general, only when M, N, and K are all large enough, the two kinds of parallelism can be fully exploited and the peak performance can be achieved. For example, when $M = 5120, N = 5120, K = 5120$, the performance of FP32 GEMM in cuBLAS can be up to 14TFlops, which is about 93% of the peak performance (15TFlops) on NVIDIA Volta 100 GPUs.

However, in many real-world applications such as deep learning and astrophysics, the matrix size is small [2, 3]. Although the existing GEMM software libraries work well for large matrices, they are inefficient for small matrices. For example, in Google-Net [25], there are 57 convolution operations, and a common algorithm to compute convolution is to transform it to GEMM. For convolution based GEMM, M refers to the number of filters, K refers to the size of filter and the number of channels, and N refers to the feature map and batch size. In general, all of these matrices' M, N and K are less than 1000, and even half of these matrices' M are less than 100. Thus, even though we increase batch size, M and K is still small. For example, for the convolution in inception3a/5x5reduce, after transforming it to GEMM, its size is $M \times N \times K = 16 \times 784 \times 192$, the FP32 performance on Volta 100 GPU is only 0.6TFlops, which is less than 1% of the peak performance. This is because the matrix is small and there are no enough tiles (thread blocks) to fully occupy the GPU after tiling.

To this end, batch execution of a group of small GEMMs has been proposed as an effective solution by merging small independent GEMMs into a single CUDA kernel [2, 21]. NVIDIA designs a batched GEMM API (*cublasSgemmBatched*) in cuBLAS [21], however, it can only batch the GEMMs with the same size (same M, N, and K). Unfortunately, in real-world cases, the matrix sizes of the batched GEMMs may vary hugely. To solve this, MAGMA proposes a batched-GEMM solution [2], called vbatched-routine, which can batch GEMMs with different sizes. It uses the gridDim.z dimension in CUDA grid to batch the small GEMMs. However, the support is still rudimentary. First, it neglects the impacts of tiling strategy. The tiling strategy suited for a single GEMM case is not necessarily good for batched GEMMs scenario. Second, the batching method can only increase the number of blocks (i.e. TLP), but cannot help to improve ILP for the case where K is small.

To fully support batched GEMMs with different sizes, the primary idea is to design a single CUDA kernel for all multiple GEMMs. The fundamental problems are tiling, batching, and their synergistic interaction. Tiling means to tile each GEMM into many tiles. We allow different GEMMs to have different tiling strategies instead of sharing a uniform tiling strategy. How to unify different tiling strategies into a single kernel is a challenge. Batching means to assign the above tiles to thread blocks. We can assign one or multiple tiles to a thread block. There is a very large space to explore. How to determine the batching method with the consideration of both TLP and ILP is another challenge.

To solve the above challenges, we propose a two-phase batched GEMM framework on GPUs, consisting of two key components: a tiling engine and a batching engine. Our contributions can be summarized as follows:

- We propose a coordinated tiling and batching framework for GEMM on GPUs.
- We design a suite of tiling strategies dedicated for batched GEMM scenario and a tiling strategy selection algorithm to determine the tiling strategy for each GEMM.
- We design a batching algorithm to assign tiles to thread blocks by balancing the TLP and ILP.
- We design a general and flexible programming interface for batched GEMM.

Evaluation results on synthetic batched GEMM cases demonstrate the proposed coordinated tiling and batching framework can achieve 1.40X speedup on average over the state-of-the-art implementation (*MAGMA* [20]) on NVIDIA Volta 100 GPU. Besides, we use GoogleNet as a real-world case study and our framework can achieve 1.23X speedup.

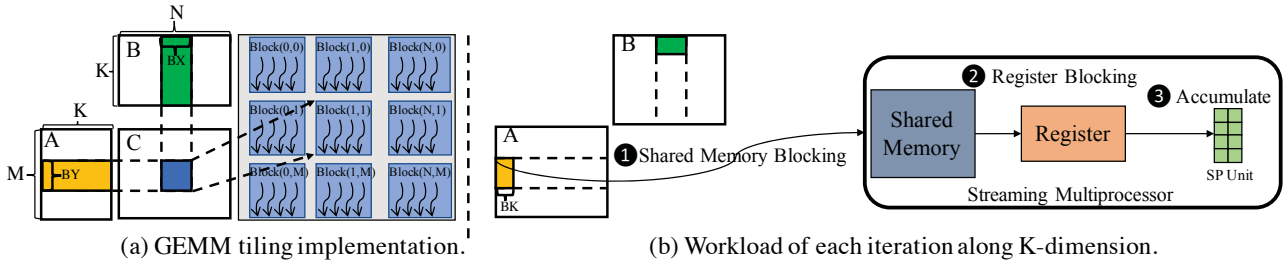
The rest of this paper is organized as follows. Section 2 presents the background of optimization for single GEMM and the baseline GPU architecture. Section 3 introduces the motivation of our proposed framework. Section 4 and Section 5 presents the details of the tiling engine and batching engine, respectively. Section 6 describes the programming interface. Section 7 evaluates the proposed framework. Section 8 discusses the related work. Section 9 concludes the paper.

2 Background

In this section, we first introduce the preliminary of GPU architecture, which is the basis for optimization on GPUs. Then, we describe the GEMM design methodology and existing popular optimization techniques on GPUs.

2.1 GPU Architecture

GPUs are becoming the most popular hardware accelerators for a wide range of applications, such as stencil, graph, finance, and machine learning. One GPU is composed of multiple *Stream Multiprocessors* (SMs) and they are connected with shared off-chip L2 cache and device memory (also known as global memory) via interconnection network. One SM contains large amounts of *SIMD* execution units: INT32 Cores, FP32 Cores, FP64 Cores, Tensor Core, and Special Function Units. The on-chip memory hierarchy consists of register file, shared memory, and L1 caches. On the latest NVIDIA Volta 100 GPUs, within each SM, the register file size is 64k 32-bit



(a) GEMM tiling implementation. (b) Workload of each iteration along K-dimension.

Figure 1. Basic GEMM implementation on GPUs.

registers and the shared memory size is configurable up to 96KB. The maximum registers per thread is 255, and they are private for each thread. If one thread needs more registers, the excess will be spilled into global memory. Shared memory is shared by the threads within a thread block. Shared memory is relatively slower than register, but much faster than global memory. Thus, proper usage of on-chip register and shared memory to exploit data reuse and locality is crucial for performance [12, 13, 15, 24, 27–30].

```
__global__ void gemm_64_64x64x8(int M, int N, int K, float *A,
float *B, float *C, float alpha, float beta){

 1:  // define double blocking register
 2:  float reg_C[64];
 3:  float reg_A[2 * BK];
 4:  float reg_B[2 * BK];

 5:  // define double shared memory
 6:  __shared__ float sh_A[2 * 64*BK];
 7:  __shared__ float sh_B[2 * 64*BK];

 8:   //load A and B from global memory to shared memory
 9:  sh_A[] = *A_start;
10:  sh_B[] = *B_start;

11:  // main loop (workload along K-dimension)
12:  for(int k=0; k<K; k+=BK){
13:      __syncthreads();

14:      for (int i=0; i<BK; ++i) {

             //load A and B from shared memory to register
15:          reg_A[] = sh_A[];
16:          reg_B[] = sh_B[];

             //compute
17:          reg_C[] = fma(reg_A[], reg_B[], reg_C[]);
18:      }

19:      // load next A tile and B tile from global memory to shared
memory
20:      if (k+BK < K){
21:          sh_A[] = *A_start;
22:          sh_B[] = *B_start;
23:      }

24:  }
25:  //write back the result
26:  *C_start= reg_C[];
}
```

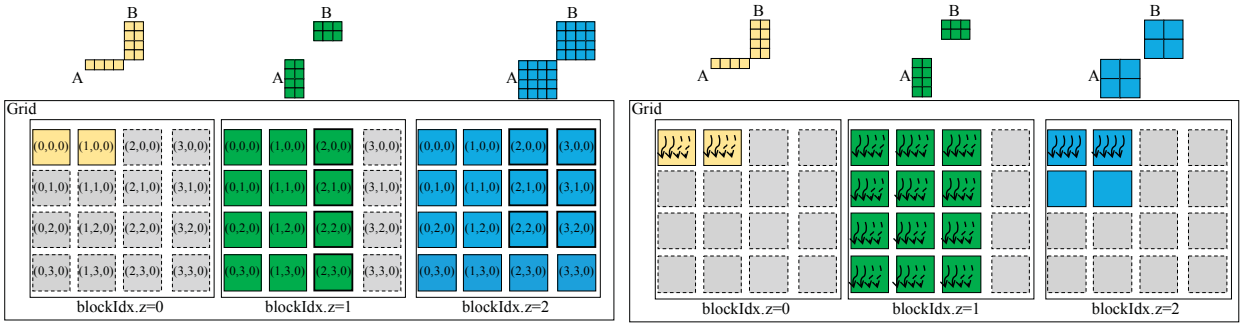
Figure 2. Code skeleton for a single GEMM.

2.2 General Matrix Multiplication

GEMM ($\mathbf{C} = \alpha \times \mathbf{A} \times \mathbf{B} + \beta \times \mathbf{C}$) has a regular and predictable data access pattern, and thus is suited for GPU acceleration. As shown in Figure 1(a), Matrix **C** is first divided into multiple tiles, and each thread block is responsible for a tile [10, 16, 26]. Tiling is an effective method to exploit parallelism on GPUs [23]. After tiling, a matrix multiplication naturally corresponds to a two-dimension grid. The computation of a thread block tile will be further decomposed into warps and threads.

Given a GEMM with size $M \times N \times K$, the **C** matrix is partitioned into multiple tiles with size $BY \times BX$. Each tile of **C** needs to access a whole row section of **A** matrix with size $BY \times K$ and a whole column section of **B** matrix with size $K \times BX$ in Figure 1(a). However, the whole row band of **A** and column band of **B** are too large to be accommodated in the shared memory and register file. To use the on-chip memory, the workload along K-dimension has to be partitioned into many segments as shown in Figure 1(b). Each segment of the row section of **A** is called an **A** tile with size $BY \times BK$, and each segment of the column section of **B** is called a **B** tile with size $BK \times BX$. The final result can be obtained by accumulating the partial result of each segment along K-dimension.

In this paper, we employ the classic techniques for GEMM such as register blocking technique and software pipelining [5, 20, 22]. Figure 2 shows a code skeleton for single GEMM using tile size with $\{BY, BX, BK\}=\{64, 64, 8\}$, and the number of threads within a block is 64. The reference for array is left blank for simplicity. We first define register block for Tile **A**, **B** and **C** from Line 1 to 4. Then, we define shared memory as the double buffer from Line 5 to 7. Next, it is the main computation along K-dimension from Line 8 to 26. Each iteration, we access an **A** tile and a **B** tile along K-Dimension, and then compute the partial multiplication result. Figure 1(b) shows the workload of each iteration. We first load **A** tile and **B** tile from global memory to shared memory ①, and then load the tiles from shared memory to register ②. Finally, we perform matrix multiplication of this tile ③. *FMA* instruction in line 17 can perform fused multiply and add operations. Thus, the partial result of the current



(a) Different GEMMs share the same tile size.

(b) Different GEMMs apply different tile size.

Figure 3. Challenges for design a single kernel for batched GEMM.

tile can be accumulated directly. In this way, the thread level parallelism between the threads can be exploited.

In addition, the instructions are pipelined to leverage instruction level parallelism within a single thread [21, 32]. Double buffer is employed for both shared memroy and registers as shown in Figure 2. More specifically, when we complete the computation of the current A(B) tile along K-dimension ③, we can load the next A(B) tile from global memory to shared memory ②. This allows us to hide the latency of loading from global memory. Register block is doubled, too. It allows to hide most of the latency of loading from shared memory, because one register block can be read while at the same time loading the next.

3 Challenge and Motivation

M and N are related to tile size and number of tiles, and they further have impacts on thread-level parallelism. Only when M and N are large enough, there are enough tiles to exploit thread level parallelism and the tile size is large enough to exploit data reuse within a tile. K decides the workload of a tile and further has impacts on instruction-level parallelism. Only when K is large enough, there are enough memory load and computation that can be pipelined. In summary, M and N are closely related to thread-level parallelism, while K is related to instruction-level parallelism. For a small GEMM, in spite of algorithm design and performance tuning, it cannot fully exploit the above two kinds of parallelism. As a result, batch execution of many small GEMMs is proposed.

Given a batch of GEMMs, in default execution mode, each GEMM corresponds to a kernel and they execute one by one. In prior works, there are two optimization directions. The first one is concurrent kernel execution based on the stream interface in recent NVIDIA GPUs [14, 17–19, 21]. Each GEMM is assigned to a different stream. For the GEMMs assigned to different streams, there are opportunities for them to execute simultaneously. However, the concurrent execution relies on kernel scheduling and the performance speedup is very limited [14, 18] due to coarse-grained scheduling overhead. The second one is to design a single kernel for all of the GEMMs [2]. In this paper, we focus on the second implementation.

NVIDIA proposes a batched GEMM API in cuBLAS library called (*cublasSgemmBatched*) [21]. However, this API can only batch the GEMMs with the same size (same M, N, and K). Unfortunately, in real-world cases, the matrix sizes of the batched GEMM may vary hugely. To this end, MAGMA group [2] proposes a scheme called vbatch to design a single kernel to batch the GEMMs with different sizes. As described in Section 2, for a single GEMM, the task of the tiles is mapped to a 2D grid, where $gridDim.x$ and $gridDim.y$ denotes the number of tiles in a row and column, respectively, and $gridDim.z = 1$. The vbatch method expands the Z-dimension of GPU kernel grid, and different GEMMs are allocated to a unique index in Z-dimension, and $gridDim.z$ is equal to the number of small GEMMs. We use an example to explain this in Figure 3(a). We have three GEMMs, their sizes ($M \times N \times K$) are $16 \times 32 \times 128$, $64 \times 48 \times 64$, and $64 \times 64 \times 128$. We use the same tile size 16×16 for the three GEMMs. The first GEMM is partitioned into 1×2 tiles, the second GEMM is partitioned into 4×3 tiles, and the third GEMM is partitioned into 4×4 tiles. The tuple within each thread block is the block index ($blockIdx.x, blockIdx.y, blockIdx.z$). The three GEMMs correspond to $blockIdx.z = 0$, $blockIdx.z = 1$, and $blockIdx.z = 2$, respectively. The size of the 2D slice is determined by the maximum matrix multiplication (4×4), and thus some of the blocks are bubble blocks (the dashed line blocks in Figure 3(a)).

There are two challenges when designing a single kernel for batched GEMM, which are left unsolved in MAGMA vbatch design. The first challenge comes from the fact that different GEMMs may have different tile sizes. A thread block is responsible for a tile. Different tile sizes may need different thread blocks. In the CUDA programming interface, all the thread blocks should use the same block size and the block size is determined by the maximum tile. This will make some of the threads in the blocks for the small tiles idle. For example, when we change tile size of the third GEMM to 32×32, shown in Figure 3(b). A 32×32 tile needs two times

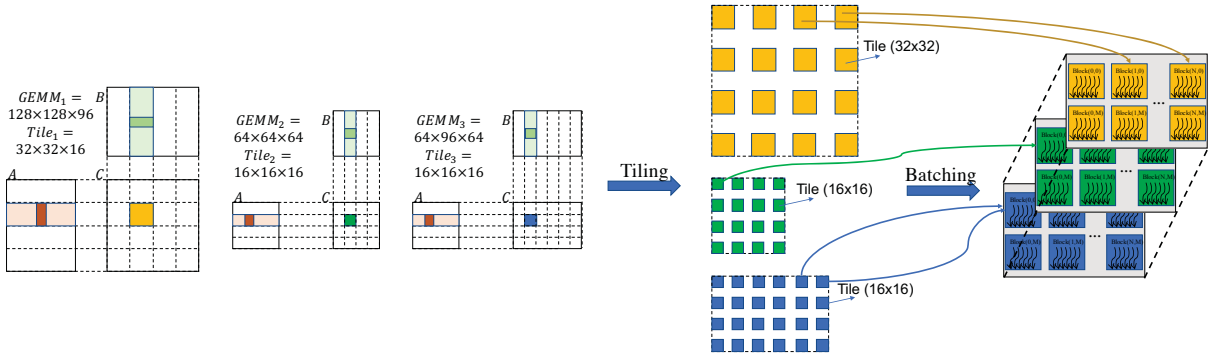


Figure 4. Overview of our proposed two-phase batched GEMM framework.

of threads than a 16×16 tile, which makes some of threads in the blocks assigned to small tiles idle (the dashed line threads). Besides, different GEMMs may have different K, and K decides the workload of each tile. Thus, there exists heavy workload imbalance between different thread blocks. In Figure 3(b), for the second GEMM, its $K = 64$ is small. For the thread blocks to execute its tiles, there is no enough workload along the K-dimension to exploit the instruction-level parallelism. Thus, how to assign tiles to thread blocks to improve instruction-level parallelism is the second challenge.

To address the above challenges, we first decompose the batched GEMM into two phases: tiling phase and batching phase shown in Figure 4. Tiling phase will partition each GEMM into many tiles, and batching phase will assign tiles to blocks. In the tiling phase , we design a series of tiling strategies dedicated for the batched GEMM scenario. Then, we present a tile strategy selection algorithm to determine tile strategy for each GEMM in batched GEMM scenario. In the batching phase, we allow a thread block to execute more than one tiles with small K one by one to improve instruction-level parallelism. Then, we design a batching algorithm to determine the batching solution and a general and flexible programming interface for any batching schemes. We will introduce them in details in Section 4 and Section 5, respectively.

4 Tiling Engine

Tiling is an important technique for optimizing parallelism and single thread performance for GEMMs on GPUs. When we increase the tile size, the number of tiles decreases and this further leads to low parallelism. When we decrease the tile size, it will lead to low data reuse within a tile. Thus, the goal of tiling engine is to strike a balance between parallelism and single thread performance.

We find that the optimal tile size for a single GEMM is not suited for the GEMM in the batched GEMM case. The reasons come from two aspects. Firstly, different GEMMs may prefer different tile size in the batched GEMM case. Different tile size need different block size (i.e. number of threads within a thread block). As discussed in Section 3, this will result in some of the threads idle. Secondly, it is more complex to select a tile size for the batched GEMM case. For a single GEMM, its tile strategy is determined by the matrix size. In the case where the matrix size is small, the optimal tile strategy is always prone to small tile. However, in the batched GEMM case, it depends on not only the size of each GEMM but also how many GEMMs are batched together. Thus, there is a larger design space to explore.

To solve the above two issues, we first design a novel tiling strategy for batched GEMM scenario. We design a unified thread structure for all the tiling strategy. With unified thread structure, we use the same thread block size for different tile sizes, but succcessfully alleviate idle threads. Then, we design a tiling algorithm to select a good tiling strategy for each GEMM.

4.1 Tiling Strategy

Table 1 lists the common tiling strategy for single GEMM scenario used in prior works [20–22]. It totally gives six tiling strategies from small to huge. A tiling strategy has two key components: the tile size ($BY \times BX \times BK$) and the number of threads for the tile. **C** tile is further partitioned into many sub-tiles, and each thread is responsible for a sub-tile. Figure 5 shows the tiling hierarchy from thread block, warp to thread. The sub-tile size of each tiling strategy is shown in the last column of Table 1. For single GEMM scenario, tile size and the number of threads for the tile can be calculated according to the global memory bandwidth and shared memory bandwidth [26]. For example, for the small tiling strategy, the tile size is $16 \times 16 \times 8$, and each thread is responsible for a 4×2 sub-tile. The number of threads the tile needs is $\frac{16 \times 16}{4 \times 2} = 32$.

To avoid the thread idle issue, the tiling strategies for batched GEMM should make sure that the thread block size is the same for all the tiling strategies. To this end, we design a series of tiling strategies dedicated for batch GEMM scenario shown in Table 2. The key idea is the unified thread structure. For different tiling strategies, we adjust the sub-tile size to ensure that all the tiling strategies share the same number of threads (either 128 or 256).

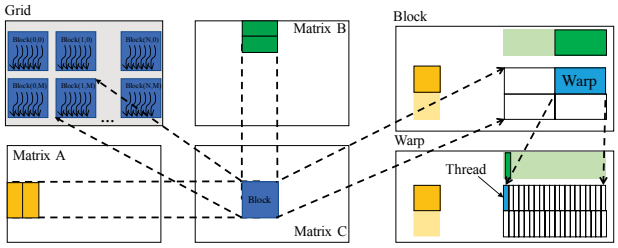


Figure 5. Tiling hierarchy from thread block, warp to thread.

Tiling Strategy	BY	BX	BK	Threads	Sub-Tile Size
small	16	16	8	32	4x2
medium	32	32	8	64	4x4
large	64	64	8	64	8x8
tall	128	64	8	128	8x8
wide	64	128	8	128	8x8
huge	128	128	8	256	8x8

Table 1. Tiling Strategy for single GEMM scenario.

Moreover, the number of threads within a thread block (denoted as T) is also a key factor to both kinds of parallelism: TLP and ILP. On one hand, when T increases, there are more threads to exploit TLP. On the other hand, when T decreases, the workload of each thread (i.e. sub-tile size) will increase. The last two columns in Table 2 show the sub-tile size when T is 128 and 256. The sub-tile size directly affects the workload of each thread, and thus further affects the ILP. Thus, for each tiling strategy, we implement multiple versions to provide more choices for TLP, such as 128-thread version and 256-thread version. One version is selected for a certain GEMM case by our tiling algorithm.

4.2 Tiling Algorithm

For single GEMM, the tiling strategy is not necessarily suited for batched GEMM scenario. We use GEMM $M = N = K = 1024$ as an example. For this case, we cannot select huge tiling strategy as shown in Table 1 with the consideration of TLP. This is because if we select huge tiling strategy, the GEMM can be partitioned into $\frac{1024\times1024}{128\times128} = 64$ tiles (i.e. 64 blocks). In general, it cannot fully occupy the current GPUs with many SMs (e.g. 80 SMs in Volta 100 GPUs). However, when we consider to batch the execution of multiple GEMMs, it is possible that the huge tiling strategy can give high number of tiles and thread blocks. Thus, how to select tiling strategy depends on how many GEMMs are batched together and what is the size of each GEMM. Next, we will first model the effects of tiling strategy on parallelism and single thread performance, respectively. Then, we design a tiling algorithm for batched GEMM case relying on the models.

Name	BY	BX	BK	Sub-Tile (128-Thread)	Sub-Tile (256-Thread)
small	16	16	8	2x1	1x1
medium	32	32	8	4x2	2x2
large	64	64	8	8x4	4x4
tall	128	64	8	8x8	8x4
wide	64	128	8	8x8	8x4
huge	128	128	8	16x8	8x8

Table 2. Tiling Strategy for batched GEMM scenario.

4.2.1 Parallelism Model

We use thread-level parallelism (TLP) to quantize the parallelism, and it is the number of threads for all the tiles of all of the GEMMs. For a given tiling strategy ($BY \times BX \times BK$) and the number of threads (T) per thread block, TLP can be calculated in the following way:

$$TLP = \sum_i \frac{M_i \times N_i}{BY_i \times BX_i} \times T \quad (1)$$

where M_i and N_i are the matrix size of matrix **C** of the i-th GEMM, BY_i and BX_i are the tiling strategy selected for this GEMM, and T is the number of threads in a thread block. We can find that on the whole, the parallelism decreases as tile size $BY \times BX$ increases. Besides tile size, the number of threads T can also affect parallelism. When T increases, there are more threads to exploit TLP.

4.2.2 Single Thread Performance Model

We use arithmetic intensity to model single thread performance. Arithmetic intensity is the ratio of the number of arithmetic instructions to memory instructions. A large arithmetic intensity means there are more arithmetic instructions per byte of memory request. It indirectly models the data reuse. The number of load instructions for each thread can be calculated as follows:

$$Num\_Load = \frac{BY \times BK + BK \times BX}{Load\_width \times T} \quad (2)$$

where the numerator means the number of float data within the tile, $Load\_width$ means the number of float data a load request can load one time. $Load\_width$ of a 16-byte load request is $16/sizeof(float) = 4$. The number of arithmetic FMA instructions for each thread can be approximated as follows[1]:

$$Num\_FMA \approx \frac{BY \times BX \times BK}{T} \quad (3)$$

So, the ratio of arithmetic instruction to load instruction is as follows:

$$Num\_FMA/Num\_Load = \frac{4 \times BY \times BX}{BY + BX} \quad (4)$$

[1]There are some arithmetic instructions for auxiliary computation, such as computing offset.

To hide the memory latency, we prefer large arithmetic intensity. When BY and BX increase, the ratio increases. BK can affect the workload of each tile, and further the number of instructions within the main loop in Figure 2. Besides tile size, the number of threads T can also affect the arithmetic intensity. When T decreases, the number of instructions will increase and it will provide more chance to hide memory latency. In this work, we only explore BY and BX and set BK as a constant (8).

4.2.3 Algorithm Design

Our algorithm gives more priority to TLP and then try to improve ILP if possible. Thus, we first consider 256-thread version tiling strategies shown in Table 2 for all the GEMMs. This is because 256-thread version has more TLP compared with 128-thread version. The algorithm consists of three steps:

1. We first obtain available tiling strategies for each GEMM. For each GEMM, its available ting strategies are the tiling strategies in Table 2 but $BY \leq M$ and $BX \leq N$. For each GEMM, we put its available tiling strategies into a priority queue. The smaller the tiling strategy is, the higher its priority is.
2. We pop one tiling strategy for each GEMM from their own priority queue. Then, we calculate the overall TLP for all the GEMMs according to Equation 1.
3. We compare the TLP with a threshold. If TLP is larger, we will try larger tiling strategy to improve ILP by repeating step 2. Otherwise, we will select the current tiling strategy as the final solution.

Note that there are two exceptions. Firstly, if there exist a GEMM which only has one tiling strategy in its own priority queue, we perform top operation instead of pop operation to guarantee every GEMM has an available tiling strategy at least. Secondly, if all GEMMs have only one tiling strategy in their priority queues, we switch to 128-thread version tiling strategies and repeat step 2. The TLP threshold in Step 3 is set empirically. It depends on the specific GPU architecture. On each platform, we determine the threshold by starting with a huge GEMM case and decreasing the TLP iteratively. We choose the inflection point with large performance degradation as the TLP threshold. The threshold is determined offline and it only needs to be done once for a particular platform.

We use an example to illustrate the algorithm. Assume that we have three GEMMs and the size of the three GEMMs is $16 \times 32 \times 128$, $64 \times 64 \times 64$, and $256 \times 256 \times 64$, respectively. We first obtain available tiling strategies for each GEMM according to step 1. For the first GEMM, it has two available tiling strategies: small and medium. For the second GEMM, it has three available tiling strategies: small, medium, and large. The third GEMM has all six tiling strategies in Table 2.

Then according to step 2, we pop the tiling strategy priority queues, and the current tiling solution is (small, small, small). Next, we calculate the overall TLP as 70144. We compare TLP with an architecture-related threshold, which is set empirically. On NVIDIA Volta 100 GPUs, we set it to 65536. In this case, TLP is larger than the threshold, this means we can choose larger tile size to trade TLP for ILP by jumping back to step 2. For all of the three tiling strategy priority queues, we pop a new tiling strategy. Note that the first GEMM has only one element in its priority queue, so its priority queue will not be popped. Then, the new solution is (small, medium, medium). The TLP is 17920, and it is smaller than the threshold. Thus, (small, medium, medium) is chosen as final tiling solution.

5 Batching Engine

After the tiling phase, the batch of GEMMs are transformed to the batch of tiles. In the prior GEMM design, a thread block is responsible for a single tile. In our design, a thread block is responsible for multiple tiles to exploit instruction-level parallelism for the tiles especially when K is small. Thus, how to assign tiles to thread blocks is a challenge. Obviously, there exists a very large exploration space. Similarly, our batching algorithm tries to strike a balance between TLP and ILP. To this end, we first propose two heuristics: threshold batching and binary batching. The first one gives more priority to TLP, while the second one gives more priority to ILP. They both rely on an architecture dependent parameter θ to quantize the workload of each thread block. We find that if the total K dimension of all the tiles assigned to a thread block is larger than a threshold θ, the benefit of batching tiles is not significant. Then, we design an algorithm to select one method from the above two heuristics based on random forest.

Threshold Batching. Threshold batching gives more priority to TLP. It first guarantees sufficient high TLP, and then tries to improve ILP. To meet the requirement of ILP, we make sure the workload of each block is not less than θ. Each time, we assign tiles to a new thread block. Before assigning, we compute the sum of the number of tiles unassigned and the number of assigned thread blocks. The sum multiplies the number of threads in a thread block to denote the TLP. If TLP is larger than the half of the threshold used in tiling engine, we assign tiles to the new thread block until the sum of their K is larger than θ. Otherwise, each of the unassigned tiles will be assigned to a thread block.

Binary Batching. Compared with threshold batching, binary batching gives more priority to ILP. To prune the exploration space and decrease the complexity of batching scheme, we only consider to batch two tiles at most each time. We combine two tiles as a pair {i,j}, and the batching problem

can be converted to the following problem:

$$minimize||\sum_{pair}(K_i + K_j - \theta)|| \quad (5)$$

where K_i and K_j is the K value of tile i and j. To reduce the overhead of batching algorithm, we sort the tiles in ascending order of K. For a thread block, we assign the first tile (with minimal K) and the last tile (with maximal K).

We have presented two different batching heuristics: threshold batching and binary batching. For the case where the batch size and the size of each matrix are fixed, for example the training process of a deep neural network, we can try both two batching heuristics and choose the better one. For the case where the batch size and the size of each matrix vary, we present a light-weight on-line batching policy to choose one from the two above heuristics. We use random forest to provide on-line selection. Random forest is an ensemble machine learning method for classification, regression and many other tasks. A random forest consists of multiple decision trees, and each tree is a weak learner. For random forest, the input feature is very important for prediction accuracy. We choose the average value of M, N, K and batch size B as prediction feature. For each tree, each node makes choice by performing a comparison. For example, for the first tree, the root node compares M with a value within this node. If M is larger than this value, the tree goes deeper along the right branch. Then, the second node compares K with a value within this node. Then, it arrives at a leaf node according to the comparison result. The leaf node is a vector with two elements. Each element corresponds to a batching heuristic, and the value represents the probability to choose this . We obtain the arrived leaf nodes of all decision trees and sum them up. Then, we choose the batching heuristic that has the maximal probability.

We use random forest algorithm as it is accurate and fast. It only needs 7-8 comparisons on average with negligible overhead. We form a training set with more than 400 samples. We test all the batching algorithms and label the sample with the best algorithm. The entire training process takes about 2 hours. It is possible to use other algorithms. We leave a more thorough investigation for future work.

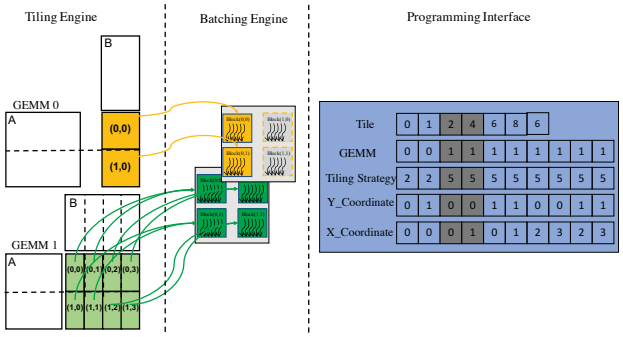


Figure 6. Auxiliary data to store the batching scheme.

6 Programming Interface

Finally, we develop a programming interface for batched GEMMs based on persistent threads [6]. Given a batching scheme, we rely on some auxiliary data structures to store the batching scheme shown in Figure 6. More clearly, we need five arrays to record the tiling and batching scheme. We first use a "Tile" array to record the Tile index assigned to each block, and the size of this array is $Block\_NUM + 1$, where $Block\_NUM$ is the number of thread blocks. The tiles assigned to each block can be calculated by $Tile[block\_id + 1] - Tile[block\_id]$. The second array we need is a "GEMM" array to record which GEMM the tile comes from. "Tiling strategy" array is used to record the tiling strategy for the GEMM that this tile comes from. We use 0-11 to represent the 12 tiling strategies in Table 2. Besides, we need another two arrays "Y_Coordinate" and "X_Coordinate" to record the position of the tile. Through the five auxiliary arrays, we can describe any possible batching schemes.

We use two GEMMs as an example, and a possible tiling and batching solution is presented in Figure 6. Assume that after tiling, there are two 128×128 tiles for the first GEMM and eight 128×64 tiles for the second GEMM. After batching algorithm, our final solution is that we have six thread blocks: two for the first GEMM and four for the second GEMM. In this case, for the third block, we can access "Tile" array to decide that it has two tiles, marked as gray in Figure 6. We still use the third block as an example. It has two tiles and their index is from the interval [2,4) in "Tile" array, and the two tiles come from GEMM 1. The tiling strategy for GEMM 1 is strategy 5 according to the "Tiling strategy" array. The coordinate of the two tiles is $(0, 0)$ and $(0, 1)$, respectively.

Figure 7 shows the code skeleton of the programming interface. It first obtain the tiles assigned to each thread block by accessing Tile array from Line 1 to 3. Then, for each tile, we access GEMM array to obtain the GEMM information of this tile Line 6 to 13. Then, we can get the position coordinate of this tile within its GEMM from Line 14 to 15. Finally, we access the Tiling strategy array to get the selected tiling strategy for this tile From line 16 to 18. After obtaining these information, the code to finish the workload of a tile is the same with algorithm in Figure 2.

7 Evaluation

In this section, we evaluate our proposed batched GEMM framework. We first evaluate tiling engine alone and then evaluate the coordinated tiling and batching engines together. Our evaluation is performed in the NVIDIA latest Volta 100 GPUs [21]. In the evaluation, the TLP threshold is set as 65536, and the threshold for threshold batching heuristic is set as 256 on Volta 100 GPUs.

To evaluate our work from a broad scale, we use synthetic batched GEMM cases with different batch sizes and different M, N, and K as input. Because MAGMA is better than cuBLAS

```
__global__ void batched_gemm(int M[], int N[], int K[],
float *A[], float *B[], float *C[], int Tile[], int GEMM[], int
Y_Coord[], int X_Coord[], int Tiling[]){

 1:  // parse the tiles assigned to this block
 2:  int begin = Tile[blockIdx.x];
 3:  int end = Tile[blockIdx.x+1];

 4:  //main loop for all tiles assigned to this block
 5:  for (int b = begin; b < end; ++b){

 6:      // parse the GEMM information of this tile
 7:      int ind = GEMM_ID[b];
 8:      int m = M[ind];
 9:      int n = N[ind];
10:      int k = K[ind];
11:     float *a = A[ind];
12:     float *b = B[ind];
13:     float *c = C[ind];

        //parse the Tile information
14:     int by = Y_Coord[ind];
15:     int bx = X_Coord[ind];

16:     int tiling = Tiling[ind];
17:     //gemm source code for this tile
18:     call the code according to tiling strategy.
    }
}
```

Figure 7. Batching programming interface based on the auxiliary data.

in the batched-GEMM cases with different M, N and K, we mainly compare with MAGMA [20] in the following evaluation with synthetic batched GEMM cases. MAGMA only provides support for batched GEMM by expanding *gridDim.z* dimension without the fine-grained tiling and batching optimizations.

Then, we demonstrate our framework can achieve performance speedup in real-world case: Google-Net. Finally, we evaluate the portability of our framework on multiple GPU architectures, such as Pascal architectures and Maxwell architecture.

7.1 Tiling Result

In the evaluation, Figure 8 shows the contribution of tiling engine. This figure contains a 2-D histogram arrays. The histograms in the same column have the same batch size, and the histograms in the same row have the same M and N. For example, the histogram in the second row and third column means the batch size is 16 and $M = N = 256$. Then, let's focus on a single histogram. The X-axis means that K increases from 16 to 2048 in logarithmic coordinate.

We can find that tiling engine can achieve about 1.20X performance speedup on average. We can have two important observations from the experiment results. First, we find that when M, N and K are fixed, the performance benefits from tile engine decreases as the batch size increases. This is because, when the batch size is small, the impact of tile selection algorithm on TLP is significant. For example, $M = N = 128$ and batch size is 4, if we select tile size $128 \times 128 \times 8$, there are only 4 blocks in total, and most of GPU is idle. When we choose small tile $16 \times 16 \times 16$, we can obtain 256 blocks in total, which can occupy most of GPU. Similarly, when batch size is fixed, the performance benefits from tile engine also decreases as the M and N increase.

Second, the impacts of K decreases as M and N increases. When $M = N = 128$, the contribution of tiling engine varies obviously as K increases. However, when M and N increases and batch size increases, the variance of contribution decreases. On the whole, the effects of tile engine is obvious when M and N are small or batch size is small.

7.2 Tiling and Batching Result

Figure 9 shows the contribution of batching engine. We can find that the coordinated tiling and batching framework can achieve 1.40X performance speedup on average. We have three important observations from the results. First, the benefit from batching engine is consistent as the batch size increases, and our batching methodology can have significant performance improvement. This mainly owes to the coordinated design of tiling and batching. When batch size is small, tiling engine can choose small tiling strategies and batching engine decrease batching depth along K dimension to guarantee thread-level parallelism. When batch size is large, batching engine can improve batching depth along K dimension to provide more chances to improve instruction-level parallelism.

Second, when K is small, the batching contribution is always higher. In fact, our flexible batching methodology can exploit both thread-level parallelism and instruction-level parallelism. By comparison, the MAGMA work can only improve thread-level parallelism but neglect instruction-level parallelism. When K is small, our flexible batching can batch tiles along K dimension to improve the workload of a thread block, which can further provide more chance to pipeline the memory accesses with computation. Thus, when K is small, the improvement is significant.

Third, on the whole the batching benefits decrease as M and N increase. This is because the motivation of batching is that the matrix is small. When M and N are large enough, the single GEMM performance is large enough.

7.3 Real-world Google-Net Result

The modern CNN models introduce many non-linear structures, such as fan-structure. More specifically, the fan-structure always spawns a few independent branches. The GEMMs in different branches can be batched together using our technique. The fan-structure is popular in other state-of-the-art

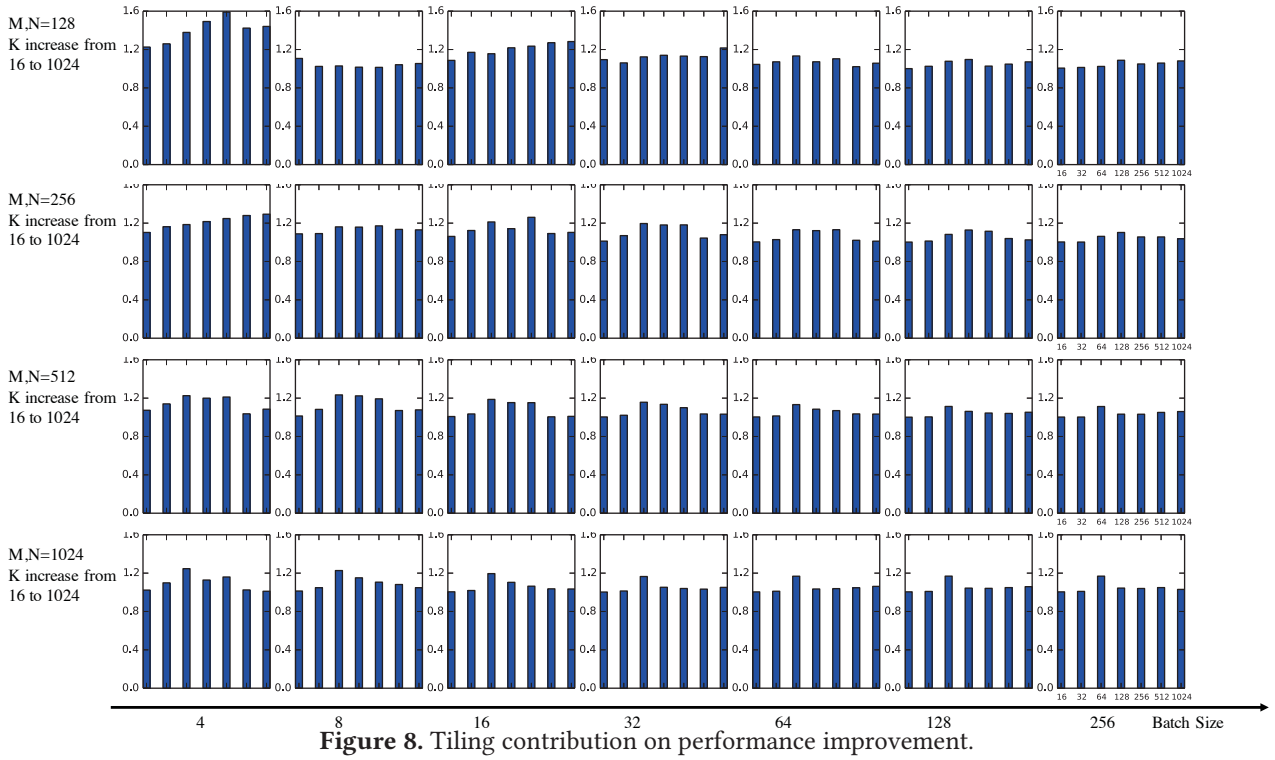


Figure 8. Tiling contribution on performance improvement.

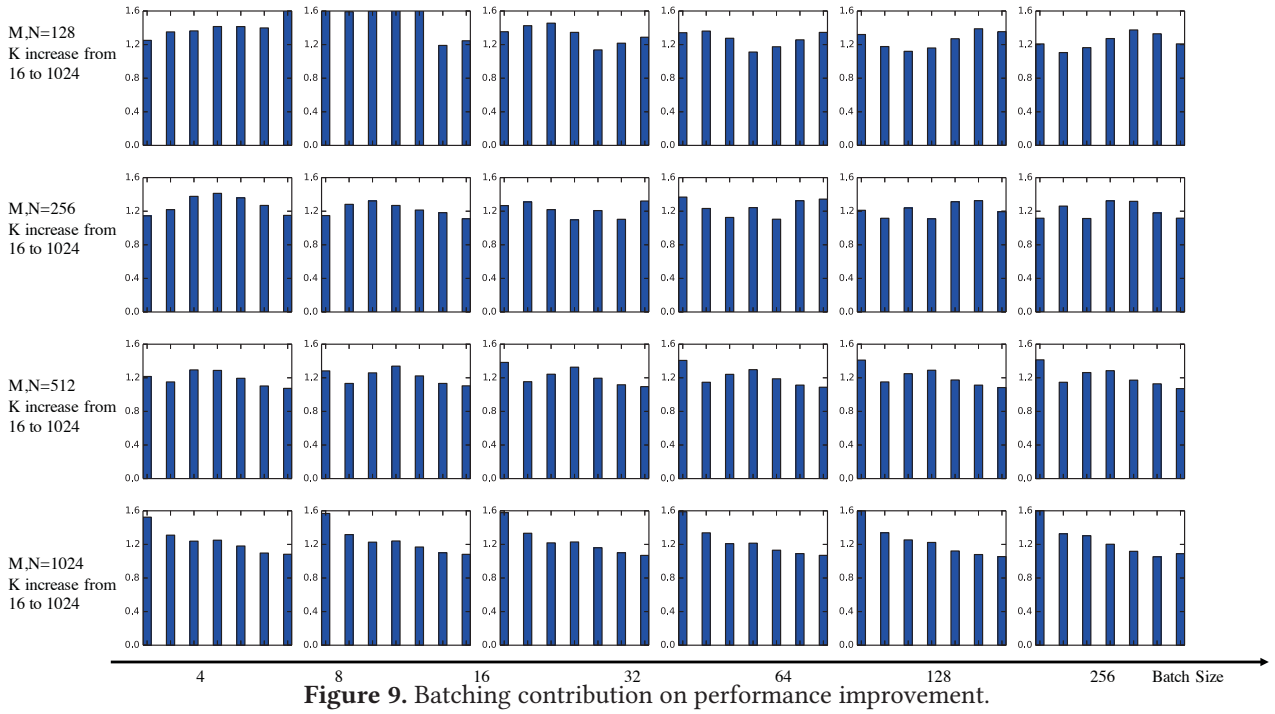


Figure 9. Batching contribution on performance improvement.

CNN models such as Squeeze-Net [8] and ResNet [7]. We use Google-Net [25] as an example to demonstrate that our proposed framework can help for this case.

GoogleNet contains 57 convolution operators. For each convolution operator, we have obtained the optimal implementation algorithm off-line. In the forward phase, we can directly select the optimal implementation algorithm for each convolution. Then, we use cuDNN implementation as

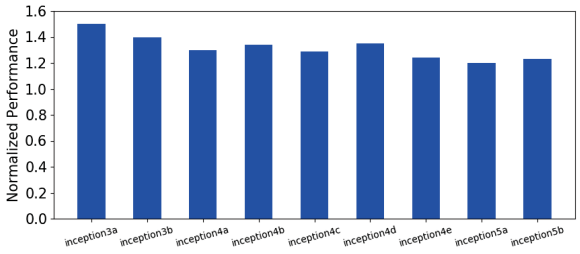


Figure 10. Batched GEMM performance speedup over state-of-the-art implementation on Google-Net inception layers.

the baseline. In addition, we also use stream interface to concurrently execute the convolution in each branch to further optimize the baseline version. We use our proposed framework to batch the four GEMMs in each inception layer. The precision we use in this experiment is FP32. The execution time of baseline version, baseline with stream optimization version and our work for a inference pass of the Google-Net is 3.18ms, 2.41ms, and 2.01ms, respectively.

Figure 10 shows the performance speedup of each inception layer than MAGMA. We can find that for inception3a and inception4a, the performance speedup can be up to 1.40X. The performance benefits come from two aspects. Tiling engine can choose proper tile strategy for each GEMM, and reduce thread under-utilization. Using inception3a as an example, the second GEMM is with $M = 16$, and the thread blocks responsible for this GEMM have large percentage of threads idle in MAGMA work. By comparison, in our framework, we can choose small tile strategy with 128-thread version for this GEMM. Besides, the batching engine can batch two tiles along K-dimension to a thread block to improve instruction-level parallelism. For the other layers, we can find that their performance speedup is about 1.25X, which is relatively lower than the first two layers. The K value for these GEMMs are large enough, so the performance benefit only comes from tiling engine. In fact, the methodology of batched GEMM is general. The other algorithm to compute convolution is implicit GEMM, which can also be batched using our proposed framework.

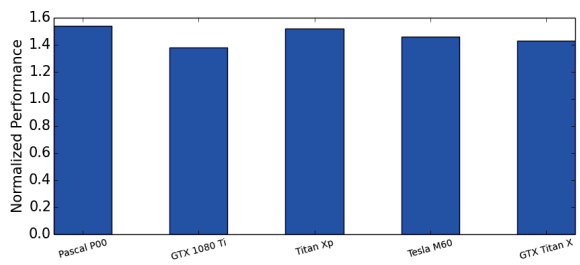


Figure 11. Sensitivity on different GPU architectures.

7.4 Sensitivity for GPU Architecture

To demonstrate the scalability on GPU architecture, we evaluate the proposed framework on Maxwell, Pascal and Volta architectures. We use three kinds of Pascal architectures: Tesla P100, GTX 1080Ti, NVIDIA Titan Xp and two kinds of Maxwell architecture: Tesla M60 and GTX Titan X. We product a evaluation for 100 randomly generated batched-GEMM cases on each of the above five architectures shown in Figure 11. On average, compared with MAGMA, our framework on these architectures can achieve 1.54X, 1.38X, 1.52X, 1.46X, and 1.43X, respectively. We can find that our framework can be well portal to other architectures and achieve a consistent performance speedup.

8 Related Work

GEMM is an important operator for many applications in a broad range of domains. Many research works focus on its optimization from algorithm level [10, 16, 26] and architecture level [11, 31, 33]. These optimization techniques have been integrated into well optimized libraries [20–22].

Full-stack Optimization. Matrix **C** is first divided into multiple tiles, and each thread block produces the partial result of each tile [10, 16, 26]. Many works focus on low level GPU micro-architecture optimization [11, 31, 33]. They hack into compiling stages and built assembly tools for Fermi and Kepler GPUs. Based on the assembly tools, they can estimate upper-bound peak performance of GEMM and perform fine-grained optimizations. Source register combinations may cause register bank conflicts, which will cause degrade in throughput. Their work proposed a register allocation technique that eliminated register bank conflict. Several control codes are included in the disassembly codes and carefully tuning the control code can result in improvement of instruction throughput. *Non-FFMA* instructions are inserted in proper positions to avoid losing performance and ensuring correctness. Finally, different load instructions from global or shared memory also influence the *GEMM* performance.

Software Projects. To fully exploit the research efforts from above algorithm and micro-architecture optimizations, many software projects are launched to pack these optimizations together in order to reduce programming difficulty [4]. Matrix Algebra on GPU and Multi-core Architectures (*MAGMA*) [20] is a dense linear library for current Multi-core GPU systems. The MAGMA research is based on the idea that, to address the complex challenges of the emerging hybrid environments, optimal software solutions will themselves have to hybridize, combining the strengths of different algorithms within a single framework. Building on this idea, they aim to design linear algebra algorithms and frameworks for hybrid many-core and GPU systems that can enable applications to fully exploit the power that each of the hybrid components offers. NVIDIA *cuBLAS* library [21] is a fast GPU-accelerated implementation of standard basic linear algebra subroutines.

Programmers can use *cuBLAS* to achieve satisfying performance with little effort. However, *cuBLAS* is not open source. CUDA Templates for Linear Algebra Subroutines (*CUTLASS*) [22] is a collection of high-performance *GEMM* templates written in pure CUDA C++. Though *CUTLASS* is written in CUDA C++ without assembly level optimization, it can still achieve around 90% of performance relative to *cuBLAS*. To use the *CUTLASS* template, programmer have to manually define the matrix blocking policy. *CUTLASS* also implemented FP16 *GEMM* with Tensor Core.

9 Conclusion

Matrix multiplication (GEMM) is widely applied in scientific computing domains, such as image processing, deep learning, and scientific computing. However, in real-world applications, the matrix size is always not large enough to drive the GPU hardware. To this end, batch execution of many small GEMMs is proposed to process many small independent GEMMs together. In this paper, we propose a coordinated tiling and batching framework for batched GEMMs. We design a series of tiling strategies dedicated for batched GEMM scenario and present a tiling strategy selection algorithm to determine tiling strategy for each GEMM. We identify that for the GEMM with small K, we can assign multiple tiles to a thread block. We design a batching algorithm to assign tiles to thread blocks with consideration of the balance between TLP and ILP. We design a general and flexible programming style for batched GEMM, which can describe any batching scheme. Finally, evaluation results show that our batching algorithm can achieve about 1.40X performance speedup over the state-of-the-art MAGMA [20] on average.

10 Acknowledgement

This work was supported by the National Natural Science Foundation China (No. 61672048 and No. 61520106004).

References

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance, Design, and Autotuning of Batched GEMM for GPUs. In *High Performance Computing*. 21–38.

[2] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2017. Novel HPC Techniques to Batch Execution of Many Variable Size BLAS Computations on GPUs. In *Proceedings of the International Conference on Supercomputing*. 5:1–5:10.

[3] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *ArXiv e-prints* (2014).

[4] Andrzej Chrzeszczyk. 2017. *Matrix computations on the GPU. CUBLAS, CUSOLVER and MAGMA by example. Version 2017.*

[5] Scott Gray. 2017. A full walk through of the SGEMM implementation. https://github.com/NervanaSystems/maxas/wiki/SGEMM. (2017).

[6] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*. 1–14.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[8] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. [n. d.]. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv e-prints* ([n. d.]), arXiv:1602.07360.

[9] Changhao Jiang and M. Snir. 2005. Automatic tuning matrix multiplication performance on graphics hardware. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 185–194.

[10] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. 2012. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems* 23, 11 (2012), 2045–2057.

[11] Junjie Lai and Andre Seznec. 2013. Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–10.

[12] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 297–311.

[13] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. 2015. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 109–118.

[14] Xiuhong Li and Yun Liang. 2016. Efficient Kernel Management on GPUs. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. 85–90.

[15] Xiuhong Li, Yun Liang, Wentai Zhang, Taide Liu, Haochen Li, Guojie Luo, and Ming Jiang. 2018. cuMBIR: An Efficient Framework for Low-dose X-ray CT Image Reconstruction on GPUs. In *Proceedings of the 2018 International Conference on Supercomputing*. 184–194.

[16] Yinan Li, Jack Dongarra, and Stanimire Tomov. 2009. A Note on Auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*. 884–892.

[17] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. 2015. Efficient GPU Spatial-Temporal Multitasking. *IEEE Transactions on Parallel and Distributed Systems* 26, 3 (2015), 748–760.

[18] Yun Liang and Xiuhong Li. 2017. Efficient Kernel Management on GPUs. *ACM Transaction on Embedded Computing System* 16, 4 (2017), 115:1–115:24.

[19] Yun Liang, Xiuhong Li, and Xiaolong Xie. 2017. Exploring Cache Bypassing and Partitioning for Multi-tasking on GPUs. In *Proceedings of the 36th International Conference on Computer-Aided Design*. 9–16.

[20] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2010. An Improved Magma Gemm For Fermi Graphics Processing Units. *International Journal of High Performance Computing Applications* 24, 4 (2010), 511–515.

[21] NVIDIA. 2018. CUDA Documentation. http://docs.nvidia.com/cuda/cublas/index.html. (2018).

[22] NVIDIA. 2018. CUTLASS: Fast Linear Algebra in CUDA C++. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/. (2018).

[23] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. 2016. Resource Conscious Reuse-Driven Tiling for GPUs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 99–111.

[24] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register Optimizations for Stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 168–182.

[25] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. *CoRR* abs/1409.4842 (2014).

[26] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast Implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 35:1–35:11.

[27] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture*. 395–406.

[28] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2018. CRAT: Enabling Coordinated Register Allocation and Thread-Level Parallelism Optimization for GPUs. *IEEE Trans. Comput.* 67, 6 (2018), 890–897.

[29] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *Proceedings of the International Conference on Computer-Aided Design*. 516–523.

[30] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *21st IEEE International Symposium on High Performance Computer Architecture*. 76–88.

[31] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 31–43.

[32] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. 2017. Versapipe: A Versatile Programming Framework for Pipelined Computing on GPU. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 587–599.

[33] Keren Zhou, Guangming Tan, Xiuxia Zhang, Chaowei Wang, and Ninghui Sun. 2017. A Performance Analysis Framework for Exploiting GPU Microarchitectural Capability. In *Proceedings of the International Conference on Supercomputing*. 15:1–15:10.

A Artifact Appendix

A.1 Abstract

This section is mainly the guideline to perform artifact evaluation for this paper. We first describe the directory tree of our code, which contains the code of both related work and our work. Then, we present the check-list for the evaluation. Finally, experiment workflow shows how to access the source code and how to use the scripts to perform detailed validation.

A.2 Description

The following is the directory tree of the code, which contains eight sub-directories as follows:

- **data.** In this sub-directory, we provide a gen_data binary to generate the data-set used in the following evaluation.
- **include.** This sub-directory contains a header file for CUDA run-time, cuBLAS and cuDNN error check.
- **default.** This sub-directory contains the source code for default execution.
- **cke.** This sub-directory contains the source code for concurrent kernel execution using stream interface.
- **magma.** This sub-directory contains the source code of the batched GEMM implementation proposed in MAGMA paper [1].
- **tiling.** This sub-directory contains the source code of the tiling engine proposed in our paper.
- **batching.** This sub-directory contains the source code of the batched GEMM implementation incorporating both of the tiling engine and batching engine in our paper.
- **google-net_cudnn.** This sub-directory contains the source code of a Google-Net inference framework built by cuDNN APIs and our batched GEMM techniques.

A.3 Artifact check-list

- **Algorithm: Multiple variations of batched-GEMM on GPUs.**
- **Program: CUDA and C/C++ code.**
- **Compilation: nvcc 9.0 with -O3 flag.**
- **Binary: CUDA executable.**
- **Data set: random matrix size and batch size.**
- **Run-time environment: Ubuntu 16.04 with CUDA SDK 9.0 installed.**
- **Hardware: Any GPUs with compute capability >=5.0 (Recommended GPU: NVIDIA V100 GPU.)**
- **How much disk space required (approximately)?: <=10Mb**
- **How much time is needed to prepare workflow (approximately)?: <=30 minutes**
- **How much time is needed to complete experiments (approximately)?: <=30 minutes**
- **Publicly available?: Yes.**

A.4 Experiment workflow

For the convenience of the artifact evaluation, we only provide a few simple scripts in each sub-directory. Below are the steps to download our code, run the experiments, and observe the results.

A.4.1 Download the code.

We provide two methods to obtain the code.

```
$git clone ppopp_ar@scc.eescer.com:/pub/tiling_batching_gemm.git
```

The password is 123456. Note that there is no dot in the password.

A.4.2 Build and Run the experiments.

Comparison. There are five variants: default, cke, and magma in related works, as well as tiling and batching in our paper. In these five sub-directories, they all contain a run.sh script and you can run it to obtain the results.

Google-net Real-world case study. We provide two different baseline implementations. We can compare the baseline version with convolution based on batched-GEMM proposed in this paper. They can be switched by modifying the variable *USE_MULTI_STREAM* in Makefile.