PA2 report

Weixiao Zhan

May 15, 2024

1 Development Flow

In this project, a general matrix multiplication (GEMM) program utilizing Compute Unified Device Architecture (CUDA) is implemented to harvest the computation power and parallel capabilities of Nvidia GPUs.

1.1 Program

In short summary, the CUDA GEMM program implemented the routine shown in Algorithm 1. The program utilized partitioning and pipelining to achieve significant performance improvement compare to a naive implementation.

Algorithm 1 CUDA GEMM

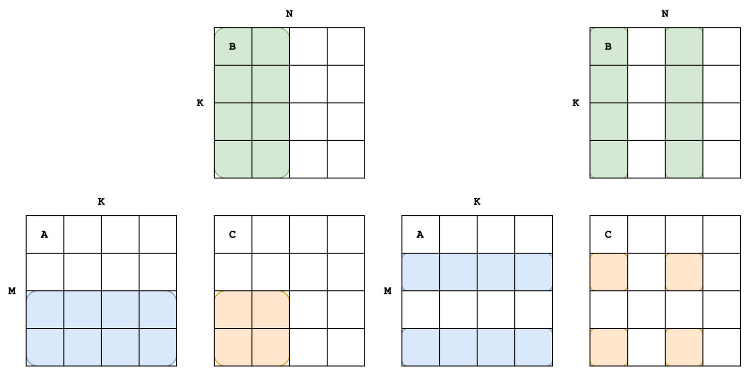
```
1: Input: $A, B, C$
2: Initialize $A\_tile$, $B\_tile$ in shared memory
3: Initialize $A\_frag$, $B\_frag$, $C\_frag$ in register
4: for each block tile in $K$ dimension do
5:     Load $A\_tile$ (global → shared memory)
6:     Load $B\_tile$ (global → shared memory)
7:     Synchronize threads
8:     for each thread tile in the $K$ dimension do
9:         Load $A\_frag$ (shared memory → register)
10:        Load $B\_frag$ (shared memory → register)
11:        $C\_frag$ += $A\_frag \cdot B\_frag$
12:    end for
13:    Synchronize threads
14: end for
15: Store $C\_frag$ to $C$ (registers → global memory)
```

1.1.1 partition

Partition, also called tiling, is one of the most important optimize technique for GEMM programs to fully utilize the memory caching hierarchy and hide memory latency.

Consider matrix multiplication $C = A \cdot B$. An $O(n^3)$ algorithm would need to load each A's element N times and each B's element M times, which, in total, is $2MKN$ times of memory load. Now suppose, the memory system has caches, i.e. is a two-level system. The program can rearrange its memory access pattern and reduce the number load from slow memory.

As shown in 1a, the program can place one row partition of A (colored in blue) and one column partition of B (colored in green) in fast memory, and perform matrix multiplication to get one partition of C (colored in yellow). In this way, despite the total number of load from fast memory is the same as an naive approach, the number of load from slow memory is reduced to $MK \cdot \frac{N}{N_{tile}} + KN \cdot \frac{M}{M_{tile}}$



(a) consecutive Partition (b) interleaved partition

Figure 1: partition methods

The partition technique is completable with CUDA paradigm: there is no dependency between partitions. Moreover, Nvidia GPUs has a three-level memory hierarchy: global memory, shared memory, registers. Thus in practice, the partition is also extended to two levels (shown in Figure 2). Correspondence between GEMM program, CUDA paradigm, and GPU memory hierarchy is summarized in Table 1. One CUDA grid handles one GEMM with A, B, and C stored in global memory; each CUDA block handles one block tile with partitions stored in shared memory; each CUDA thread handles one thread tile with finer partitions stored in registers.

GEMM	CUDA	GPU Memory Hierarchy
entire matrix	grid	global memory
block tile	block	shared memory
thread tile	thread	registers

Table 1: Correspondence between GEMM program, CUDA paradigm, and GPU memory hierarchy

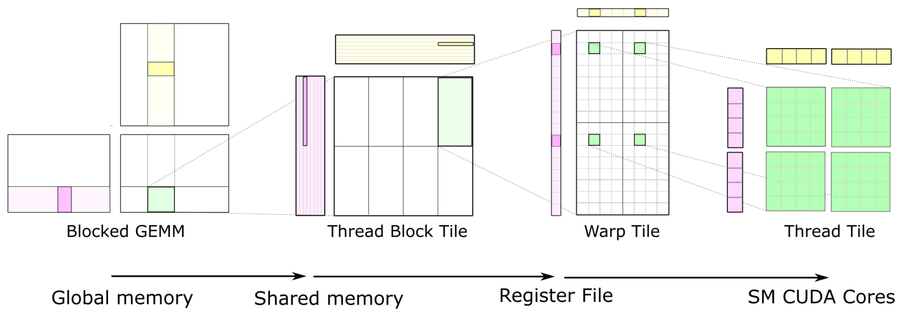


Figure 2: CUDA GEMM partition

1.1.2 interleave partition

Nvidia GPU memory supports coalesced loads. Thus its beneficial to perform an interleaved partition (shown in 1b) instead of consecutive partition for thread tile. With interleaved partition, threads within a block would simultaneously perform load operation to adjacent memory locations. The GPU would coalesce the loads and maximize the memory bandwidth.

1.1.3 boundary check

The size of matrix A, B are not always multiple of block tile sizes. Thus it is necessary to perform boundary checks to ensure correctness. In practice, there are two boundary checks are implemented:

1. when loading A_tile, B_tile from global to shared memory: set the shared memory corresponding to out-of-boundary entries to 0;
2. when storing back C_frag, skip out-of-boundary entries.

Under CUDA paradigm, it is more efficient to carry those out-of-boundary entries as zeros through out the computation instead of using branch statements to identify them. Branch statements can cause branch divergence and introduce extra cycles.

1.1.4 software pipelining

The typical workload paradigm of a CUDA block in our CUDA GEMM program is shown in Algorithm 2. Note the loads and arithmetic operations are clustered and separated by synchronization barriers. Such workload can quickly exhausted the queue of memory unit or arithmetic unit of a streaming multiprocessor (SM), causing the warp get blocked while the other queue still have capabilities. when one warp is blocked, no other warps from the same CUDA block can make progress on the same SM, since they are all waiting for the same resources. Thus the scheduler can only scheduler warps from another CUDA block and reduce the performance.

The solution is to introducing double caching and software pipelining. This method can eliminate one synchronization barriers, as shown in Algorithm 3.

Algorithm 2 Workload

```
1: repeat
2:     load one block tile                ▷ many loads
3:     synchronize threads
4:     compute one block tile             ▷ many arithmetic
5:     synchronize threads
6: until consumed all block tile
7: Store register to global memory
```

Algorithm 3 Pipelined Workload

```
1: load one block tile to one cache
2: repeat
3:     compute one block tile
4:     load next block tile to the other cache
5:     Synchronize threads
6: until consumed all block tile
7: compute one block tile
8: Store register to global memory
```

1.2 develop process

The develop process follows the order below:

1. implemented block partition.
2. implemented boundary check so that the program would work on any size matrices, including non-square ones.
3. implemented thread interleaved partition, allowing each thread processing multiple entire.
4. implemented pipelining.
5. implemented auto-tuning Python script `auto.py` to search best tile sizes for different matrix sizes.

1.3 incremental improvement

As shown in Figure 3, partition (block tiles), interleaved partition (thread tiles), pipelining, and parameter tuning give incremental performance improvement; meanwhile the necessary boundary checks costed negligible performance loss. In sum, all optimization methods are working.

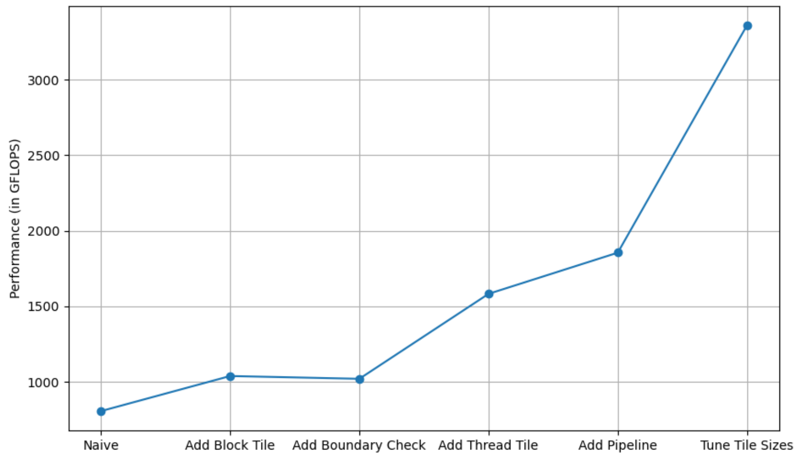


Figure 3: performance gain during develop process on $n = 1024$

2 Result

The auto-tuning script explored block tile sizes [1] $Bs = \{16, 32, 64\}$ and thread tile sizes [2] $Ts = \{1, 2, 4, 8\}$ on problem sizes $n = \{128^\ddagger, 256^\ddagger, 512^\ddagger, 768^\ddagger, 1024^\ddagger, 2048^\ddagger 4096^\ddagger\}$ [3]. The results are present in `resutls.csv`.

Figure 4 shows the best performing 4 tile size combinations out of the 12 been measured.

2.1 tile size limitation

Denote the block tile size B_s and thread tile size T_s.

Nvidia T4 GPU supports maximize 1024 threads per block. Thus the block tile size is at most $\sqrt{1024} = 32$ times greater than thread tile size:

$$Bs \leq 32 * Ts$$

Nvidia T4 GPU has the 64KB shared memory, which can hold up to 16384 floating point entries. Each block need $2\times$ space for A and B, $2\times$ space for pipelining, i.e. $2 \times 2 \times B_s^2 \leq 16384$. Thus the largest block tile size

$$Bs \leq 64$$

[1] block tile size refer to the number entries processed by one CUDA block
[2] thread tile size refer to the number entries processed by one CUDA thread
[3] $n^\ddagger = \{n-1, n, n+1\}$

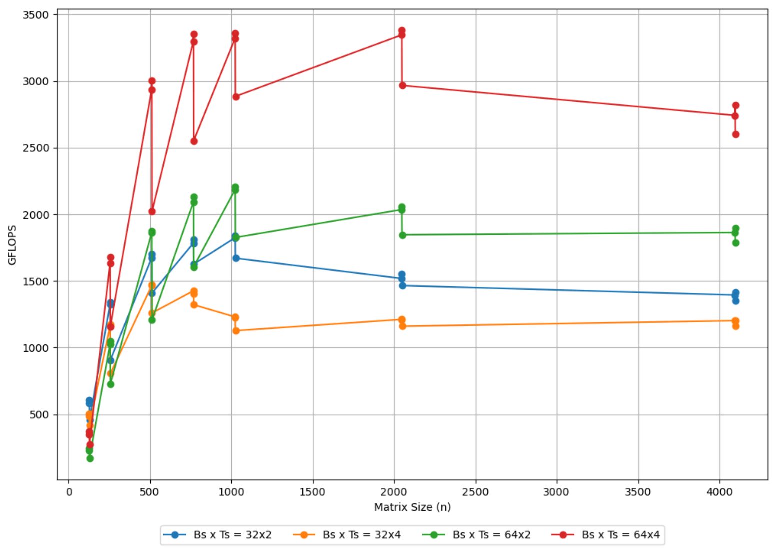


Figure 4: performance

2.2 non-multiple matrix sizes

As shown in Figure 5, all non-naive implementation suffered performance loss on non-multiple matrix sizes, especially on $n = 2^k + 1$. The optimization in those programs typically used partition to accelerate computation. Matrices with non-multiple sizes would encounter same overhead as larger multiple-sized matrices, but with fewer effective operations. Despite the performance loss on non-multiple-size matrices compare to multiple-size matrices, optimized implementations are still faster than the native implementation at all matrices sizes.

2.3 Optimal sizes

As detailed in Section 1.1.1, larger partitions result in more significant partition acceleration. However, it is crucial to maintain a sufficient number of block tiles to keep all GPU SMs fully occupied. Table 2 illustrates that for large matrices, the optimal block tile size is the maximum size permissible by the hardware, which is 64.

In contrast, for smaller matrices such as those of size 128, using a block tile size of 64 results in only 4 CUDA blocks. This quantity is inadequate for engaging all SM cores effectively. Consequently, smaller block tile sizes, such as 32, are recommended for smaller matrices.

As for thread tile sizes, the auto-tuning results suggest setting $T_s = \frac{B_s}{16}$ is an optimal balance between parallel efficiency and memory utilization. Smaller

N	CPU	GPU(CUDA)			
	BLAS	Naive	CuBLAS	My GEMM	
				Peak GF	(Block Tile) x (Thread Tile)
127		280.77		584.46	(32,32) x (2,2)
128		286.99		608.73	(32,32) x (2,2)
129		289.98		457.05	(32,32) x (2,2)
255		639.22		1632.57	(64,64) x (4,4)
256	5.84	657.96	2515.3	1677.44	(64,64) x (4,4)
257		649.01		1158.61	(64,64) x (4,4)
511		771.30		2935.53	(64,64) x (4,4)
512	17.4	788.17	4573.6	3002.40	(64,64) x (4,4)
513		775.21		2020.90	(64,64) x (4,4)
767		794.12		3297.12	(64,64) x (4,4)
768	45.3	830.18	4333.1	3352.96	(64,64) x (4,4)
769		786.03		2552.03	(64,64) x (4,4)
1023	73.7	751.69	4222.5	3319.11	(64,64) x (4,4)
1024	73.6	804.15	4404.9	3359.64	(64,64) x (4,4)
1025	73.5	735.28	3551.0	2884.39	(64,64) x (4,4)
2047	171	529.25	4490.5	3345.96	(64,64) x (4,4)
2048	182	605.77	4669.8	3382.27	(64,64) x (4,4)
2049	175	524.86	4120.7	2966.85	(64,64) x (4,4)
4095		459.44		2742.34	(64,64) x (4,4)
4096		571.76	4501.6	2822.26	(64,64) x (4,4)
4097		458.25		2601.67	(64,64) x (4,4)

Table 2: Peak GFLOPS Performance Comparison and Thread Block Sizes

thread tile size can lead to more scheduling warps and higher utilization. Larger thread tile size can improve partition acceleration.

2.4 Peak GF

Peak GP achieved are shown in Table 2.

3

Table 2 shows the Peak GF of my GEMM program along with the tile sizes.

4 Analysis

4.1 comparison between methods

Table 2 and Figure 5 shows the Peak GF of my GEMM program along with the performance from other implementations.

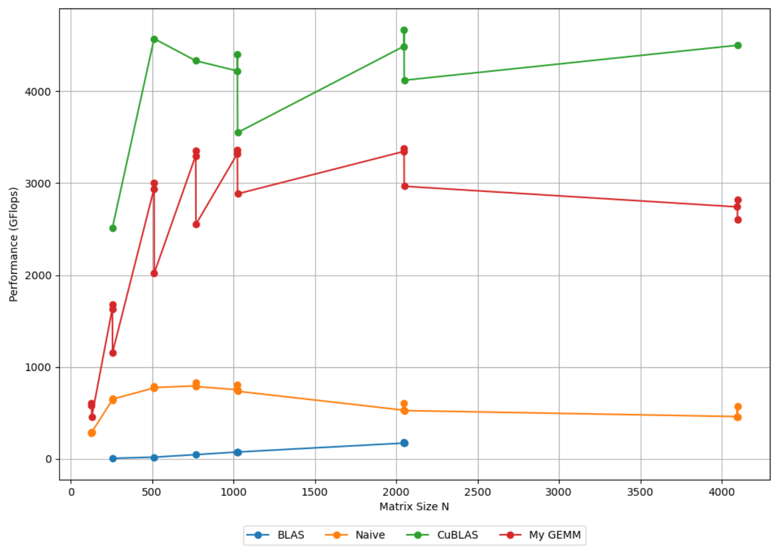


Figure 5: comparison

4.2 curve shape

As shown in Figure 5, both my GEMM and the naive program suffered some performance loss on extra large matrix sizes, The GF with $n = 409$ is less than GF with $n = 2048$. However, CuBLAS is not affected by extra large matrix sizes.

4.3 irregularities

All non-naive implementation suffered performance loss on non-multiple matrix sizes, especially on $n = 2^k + 1$. As the optimized program typically used partition to accelerate computation, and matrices with non-multiple sizes would cause some overhead. Despite the performance loss on non-multiple matrix sizes compare to multiple matrix sizes, non-naive implementations are still faster than the native program at all matrices sizes.

5

5.1

$$\begin{aligned}\text{Peak Performance} &= \text{SM Count} \times \text{Cores per SM} \times \text{Clock} \times \text{Ops per Cycle} \\ &= 40 \times 64 \times 1.5G \times 2 \\ &= 7.68T\end{aligned}$$

To arrive the peak performance, the program's operation intensity (OI) should be 24. The estimated OI of my program under theoretical memory bandwidth is 10.57.

5.2

The estimated OI of my program under actual memory bandwidth is 15.37. The estimated OI will increase as memory bandwidth decrease and the performance stay the same.

However the actual OI is inherit of program, and the estimation is based on the assumption that the computation is either memory bounded or operation bounded, i.e. the program is running on the roofline.

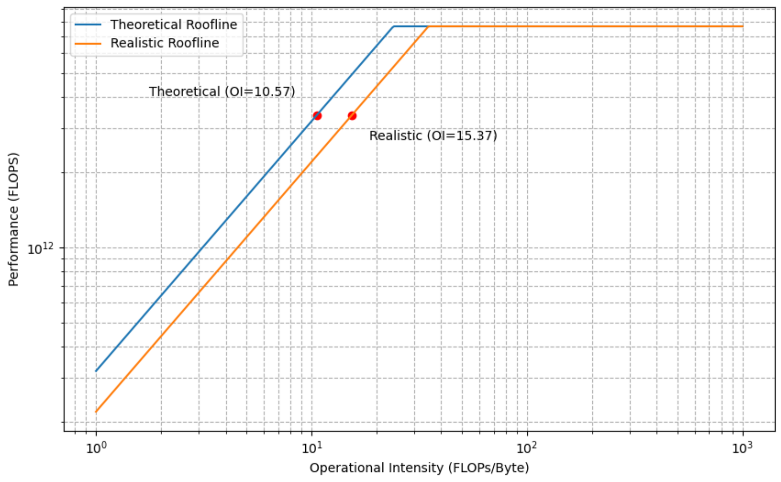


Figure 6: Theoretical and Realistic Roofline Model for NVIDIA T4 GPU

6 Potential Future Work

Use Nvidia profiling tool to learn what other factors is limiting the computation, and if there is any improvement can be made.

7 Extra Credits

Implemented a non-square matrix multiplication kernel, that is able to take 2 matrices A and B of dimensions $M \times K$ and $K \times N$ respectively and compute C of $M \times N$.

References

[1] NVIDIA. *CUTLASS: Fast Linear Algebra in CUDA C++*. `https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/`.

[2] Zhe Jia, Marco Maggioni, Jeffrey Smith, Daniele Paolo Scarpazza. *Dissecting the NVIDIA Turing T4 GPU via Microbenchmarking*. `https://arxiv.org/pdf/1903.07486.pdf`.