Optimizing Matrix Multiplication in CUDA

Matthew Zhao
mz22238
mzhao12
matthewzhao420@gmail.com

May 1, 2025

Matrix Multiplication is one of the most performance-critical kernels in many applications, from scientific computing to machine learning and computer graphics. In this project I sequentially optimize CUDA kernels from matrix multiplication, starting from a naive implementation to a more optimized tiling variant.

Given two dense matrices $A \in \mathbb{R}^{M\times K}$ and $B \in \mathbb{R}^{K\times N}$, the goal is to compute

$$C = AB, \qquad C_{ij} = \sum_{k=1}^{K} A_{ik}B_{kj}. \tag{1}$$

The GPU I run experiments on is the NVIDIA Quadro RTX 5000, which offers

- **3 072** FP32 CUDA cores across **48** Streaming Multiprocessors (SMs)
- a theoretical peak of **11.2 TFLOP/s** (FP32)
- **448 GB/s** of device memory bandwidth

The initial testing was done with multiplying 2 matrices with size 4092x4092, which requires approximately 2*4092*$(4092)^2\approx$ 137 GFLOPs to compute. Full results with varying matrix sizes are at the end of the write-up.

1 Naive Implementation

The first implementation is the following:

```
__global__ void kernel1(int M, int N, int K, const float *A, const float *B, float *C)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < N)
    {
        float sum = 0.0f;
        for (int k = 0; k < K; ++k)
        {
            sum += A[row * K + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

Launching with:

```
dim3 threadsPerBlock(32, 32);

dim3 numBlocks((N + threadsPerBlock.x - 1) / threadsPerBlock.x,
               (M + threadsPerBlock.y - 1) / threadsPerBlock.y);
```

In this kernel, each thread computes the dot product of a row from A and a column from B to compute a single element of C. This kernel takes 177 ms to perform the matrix multiplication between 2 4092x4092 matrices, which is roughly 774 GFLOPs/s.

With a closer look, we can see that this kernel is already coalescing global memory accesses - when loading B, threads with consecutive threadIdx.x in the same warp read consecutive columns. Since they are contiguous in memory, the warp scheduler can load 4 floats with a single GMEM 128B load instruction, overall allowing less DRAM traffic since each load is more efficient. We can see the degree of speedup this coalescing has by testing the same kernel with the indexing switched:

```
int row = blockIdx.x * blockDim.x + threadIdx.x;
int col = blockIdx.y * blockDim.y + threadIdx.y;
```

With this small change, loads to B are no longer coalesced, as consecutive threads in a warp now read consecutive rows, which aren't contiguous in memory. This worse kernel takes 1591 ms, or approximately 86.11 GLOPs/s.

Comparing the Nsight Compute (ncu) diagnostics for the kernels, the coalescing increases the memory bandwidth from 5.01 GB/S to 64.82 GB/S. This explains the large difference in efficiency; since the non-coalescing kernel is memory/bound, many cycles are wasted waiting for memory leading to low compute utilization. Still, the coalesced kernel still performs suboptimally.

2 Transposing of A

An obvious optimization might just be to transpose A before performing the matrix multiplication so that loads to A are also coalesced. This can be done simply with a separate transpose kernel

```
int r = blockIdx.y * blockDim.y + threadIdx.y;
    int c = blockIdx.x * blockDim.x + threadIdx.x;

    if (r < M && c < K)
    {
        A_t[c * M + r] = A[r * K + c];
    }
```

And then changing the matrix multiplication like so:

```
sum += A[k * K + row] * B[k * N + col];
```

This takes 189 ms, or approximately 724.87 GFLOPs/s. This suggests that the overhead of transposing A is too large to make the small benefits to coalescing worth it.

3 Shared Memory Tiling

According to the ncu diagnostic of the first kernel, warp stalling is a big problem, with an average of 35.2 cycles stalled between two instructions, of which 76.3% are stalled waiting for the L1 instruction queue for global memory operations. Thus, the first real optimization we can make is using shared memory. In this next kernel, each thread loads its corresponding value from A and B into shared memory, and performs as much computation as possible using only shared memory within its block. The same process is repeated by an outer loop over the K dimension, sliding the chunks horizontally for A and vertically for B. The important parts of the code:

```
    // slide over K in BLOCKSIZE-wide panels
    for (int bk = 0; bk < K; bk += BLOCKSIZE)
    {
        // transposed loads of A and B to coalesce
        if (row < M && bk + threadIdx.x < K)
            As[threadIdx.x][threadIdx.y] = A[row * K + (bk + threadIdx.x)];
        else
            As[threadIdx.x][threadIdx.y] = 0.0f;

        if (bk + threadIdx.y < K && col < N)
            Bs[threadIdx.x][threadIdx.y] = B[(bk + threadIdx.y) * N + col];
        else
            Bs[threadIdx.x][threadIdx.y] = 0.0f;

        __syncthreads();

        // compute the BLOCKSIZE-length dot product
        for (int t = 0; t < BLOCKSIZE; ++t)
            sum += As[t][threadIdx.y] * Bs[threadIdx.x][t];

        __syncthreads();
    }
```

This takes 180 ms, or 761.1 GFLOPs/s. However, this kernel doesn't completely solve the memory bottleneck. According to ncu, the warp cycles per issued instruction have actually increased to 41.67 cycles, 65.4% of which are stalled waiting for the MIO (memory input/output) instruction queue. This suggests that too many SMEM instructions are occurring.

4 Blocktiling

To address this, we can have each thread calculate multiple results in C, increasing the arithmetic intensity. The main optimization of the next kernel is introducing another loop over the multiple results each thread calculates. In this kernel, a single thread calculates NR sum values, performing the dot product between NR rows in A with 1 column in B. A small optimization is having the outer loop be the dot product, so we can reuse the values in B. Now, rather than doing 1 FMA per load, we do NR FMA's per load.

```
    for (int t = 0; t < BK; ++t) {
        float tmp = Bs[t][threadIdx.x];
        for (uint i = 0; i < NR; ++i) {
            sums[i] += As[threadIdx.y*BK+i][t] * tmp;
        }
    }
```

One consequence of this approach is that we have more flexibility for the block dimensions, since we can change the value of NR. With some simple auto-tuning, I found this configuration to perform best:

```
dim3 threadsPerBlock2(BN, BK);
dim3 numBlocks2((N + BN - 1) / BN, (M + BM - 1) / BM);
```

Where BN = 128 and BK = 8, with NR = 128/8 = 16. This configuration only takes 95 ms, or 1442.11 GFLOPs/s. This is around a 2x speedup of the naive implementation, although still far from the theoretical limit of the GPU, which is 11.2 TFLOPs/s.

5 Results

The results shown in Figure 1 show the blocktiling approach was the fastest, and the GFLOPs/s seem to scale linearly with the problem sizes increasing in powers of 2. Some of these results were unexpected, however; since I spent most of my time writing and testing the kernels on my NVIDIA RTX 3060 Laptop GPU (out of convenience), I had expected similar results as Figure 2 where the shared memory kernel performs better than the naive kernels. However, the shared memory kernel performed roughly the same as the naive kernels. Moreover, the GFLOPs/s seemed to scale differently - GFLOPs/s seem to stay mostly constant in Figure 2.

These discrepancies are most likely due to different hardware specifications. Besides the obvious difference in throughput, the laptop GPU's performance saturates quickly, starting around N=1024. This is because the laptop GPU runs out of memory bandwidth quickly compared to the Quadro RTX 5000 GPU and then faces higher fixed overheads, which makes performance even degrade slightly. The different gaps between the kernels, can also be explained by this: the lower memory bandwidth in the laptop GPU makes the naive kernels much more memory-bound, magnifying the speedups of the shared memory and blocktiling kernels, while the kernels in the Quadro RTX 5000 are not nearly as memory-bound.

Naturally, these results are far from optimal. The most natural place to continue from would be to try increasing arithmetic intensity again, by doing 2D blocktiling instead of just 1D. There are also other problems like SMEM bank conflicts worth exploring.

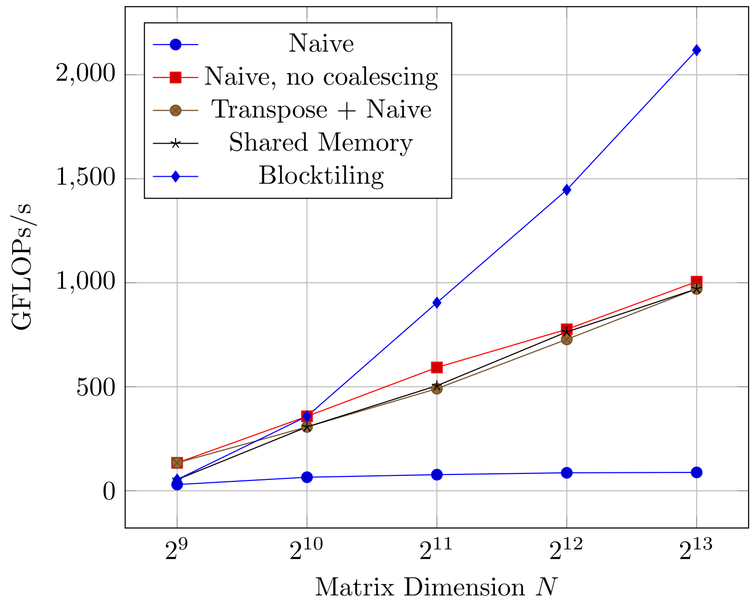

Figure 1: Performance trajectories (GFLOPs/s) vs. problem size for each kernel. Done on a NVIDIA Quadro RTX 5000 GPU

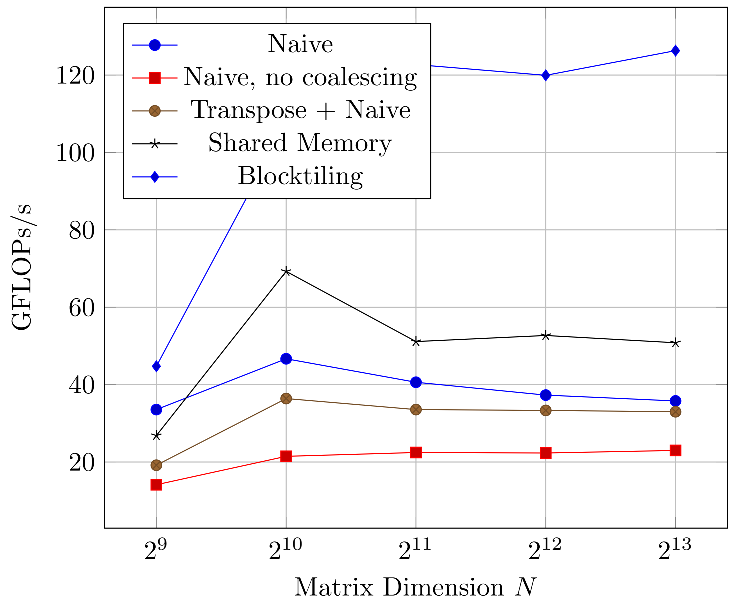

Figure 2: Performance trajectories (GFLOPs/s) vs. problem size for each kernel. Done on a NVIDIA RTX 3060 Laptop GPU