Exp1 Report

Joshua Gisiger

Introduction

Matrix multiplication has, in recent times, become the most researched operation in computer science, for the most part due to the demand for faster and more efficient machine learning training and execution algorithms. As a means of exposing us to both optimization algorithms, general capabilities of the GPU via that CUDA runtime API, and functional knowledge of GPU memory architecture, COMP 468's first assignment requires students, like myself, to perform matrix multiplication on a baseline algorithm (ran on the CPU), a naive algorithm (on the GPU), a tiled algorithm (on the GPU), and on CuBLAS (on the GPU). We are to measure the performance of each algorithm and compare them to one another. In doing so, students will get a deeper understanding of perfomance optimization on the GPU and why using shared memory improves GEMM operations.

Background

This lesson is likely heavily inspired by the seminal paper `FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, Dao et. al.` The paper explored I/O awareness into GPU algorithms and introduced tiled algorithms for GEMM operations. Specifically, it noted that by loading tiles, also known as sub-matrices, onto shared memory, algorithms could significantly limit how often they reach into global GPU memory, which is slow in comparison to shared memory (as fast as L1 cache).

Method and Design

The code was designed mostly like a script for quick development and testing convenience. It was notably decoupled into two portions: 1. The code body 2. The sweep script

Code Portions

1. The Code Body

The code body works in three sequential steps. It first parses the command line. In the command, the user is able to specify the `matrix dimensions`, `GEMM implementation/algorithm`, and a boolean variable `verify`, indicating whether we should verify the operation's result against CuBLAS. After parsing the command, the code body proceeds to running the requested `GEMM implementation`. Finally, the code copies the result back onto the host, and if specified by `verify`, compares the result to the CuBLAS result.

2. The Sweep Script

The sweep script automates benchmarking by iterating over a fixed set of square matrix sizes and GEMM implementations, invoking the compiled `dgemm` binary for each configuration. It parses the program's output into a structured CSV file containing timing and throughput metrics, enabling straightforward comparison of performance across implementations and problem scales.

GEMM Implementation

In relation to the different GEMM implementations, we built a baseline, naive, and tiled algorithm. To better understand them, we spend the next few paragraphs describing them in depth.

1. Baseline Implementation

This GEMM implementation was a simple triple for loop run directly on the CPU. It calculated the resulting matrix, element by element, meaning that the tasks were partitioned by the output data. Specifically, the two outer loops determined the index of the ouput element we were computing, and the inner loop iterated over the input data needed to calculate the input. Despite the algorithm's data partitioning, the implementation executed sequentially, making no use of any CPU parallelism.

2. Naive Implementation

The naive GPU implementation assigns each CUDA thread the responsibility of computing a single output element in the result matrix. Threads determine their corresponding row and column indices from the block and thread coordinates, and then perform a straightforward dot product over the shared dimension (K). For each iteration of this inner loop, the thread loads one element from matrix (A) and one element from matrix (B) directly from global memory, accumulating the result in a local register before writing the final value back to global memory. While this approach exposes a high degree of parallelism across output elements, it makes no attempt to reuse data between threads, causing each input element to be repeatedly fetched from global memory and resulting in a large number of slow global memory accesses that ultimately limit performance.

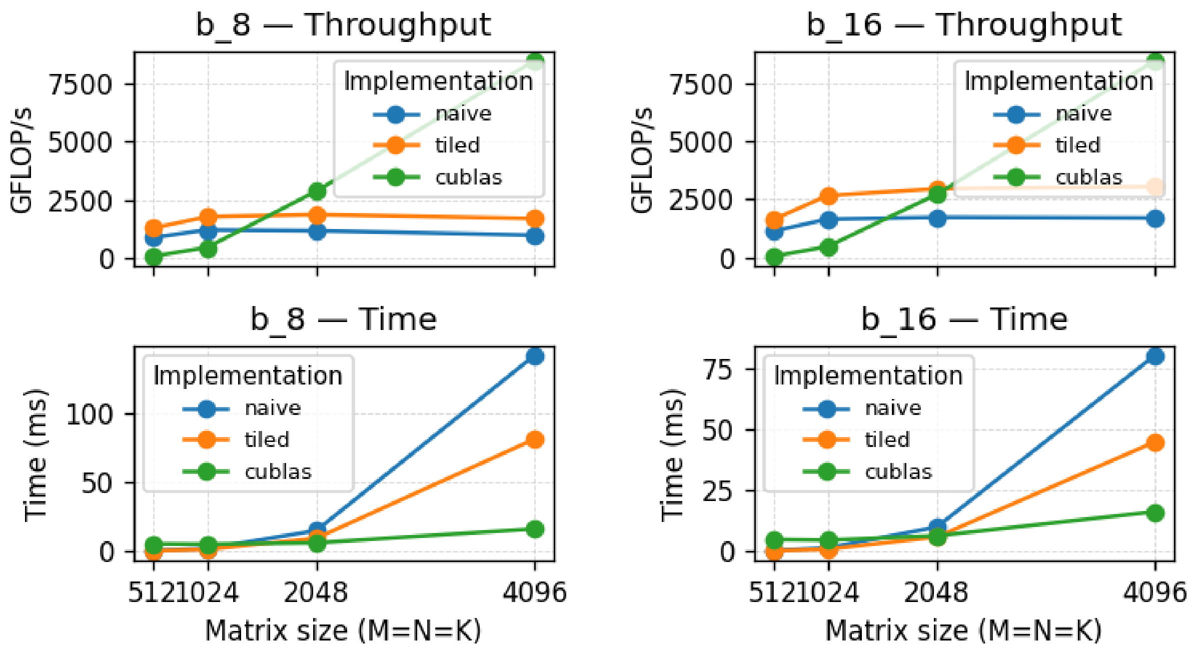
3. Tiled Implementation

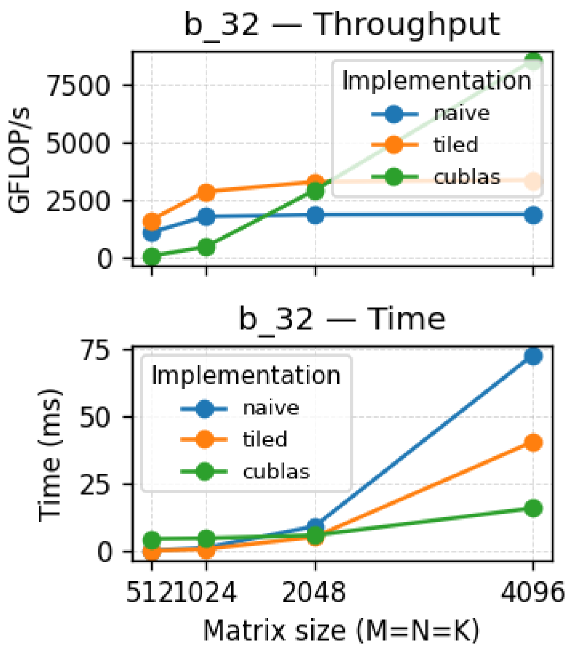
The tiled implementation preserves the same overall work partitioning as the naive kernel—each thread is still responsible for producing a single element of (C)—but it fundamentally changes *where* the data comes from during the inner product. Instead of repeatedly fetching operands from global memory, each thread block cooperatively loads a (BLOCK_SIZE X BLOCK_SIZE) tile of (A) and a matching tile of (B) into shared memory (`tile_A` and `tile_B`). After a synchronization barrier ensures the entire tile is resident on-chip, threads compute a partial dot product using fast shared-memory accesses, and then advance to the next tile along the (K) dimension.

From a memory perspective, this is the key optimization: in the naive version, every multiply-add in the inner loop triggers two global reads (one from (A), one from (B)), meaning global memory is hit (O(K)) times per output element. In the tiled version, global memory is only accessed when loading the next tiles—each thread performs at most one global read for (A) and one for (B) *per tile*, and the subsequent (BLOCK_SIZE) multiply-adds reuse those values from shared memory. This dramatically reduces redundant global memory traffic, increases data reuse within a block, and is exactly why the tiled kernel scales far better as matrix sizes grow.

Results and Discussion

From the sweep script tests we found the following:





The performance results compare three GPU-based GEMM implementations: a naive kernel, a tiled kernel, and CuBLAS. The naive GPU implementation achieved moderate throughput but scaled poorly with increasing matrix size. The tiled implementation consistently outperformed the naive kernel across all tested configurations, achieving higher GFLOP/s and lower execution times, with the performance gap widening at larger problem sizes. CuBLAS delivered the highest overall throughput for large matrices, but for smaller matrix sizes it was occasionally slower than the tiled implementation, indicating nontrivial overheads.

3. Observations

These results show that memory access efficiency plays a critical role in GPU GEMM performance. Although the naive kernel exposes sufficient parallelism, its repeated global memory accesses limit scalability. The tiled kernel mitigates this issue by leveraging shared memory to increase data reuse and arithmetic intensity. The observation that CuBLAS can underperform the tiled kernel at smaller matrix sizes suggests that its highly optimized routines incur additional overhead—such as kernel launch configuration, internal blocking, and scheduling costs—that outweigh the benefits of aggressive optimization when the problem size is small. As matrix sizes increase, these fixed overheads are amortized, allowing CuBLAS to achieve superior performance.