GPU Computing Assignment #1

Matrix Multiplication Performance Comparison (Serial vs. Naive vs. Tiled)

Due: 2025.05.30 (Fri) 23:59

Objective

The purpose of this assignment is to implement and analyze matrix multiplication using different computing strategies:

1. Serial (CPU-based) implementation
2. Naive CUDA kernel (global memory only)
3. Tiled CUDA kernel (shared memory optimization)

You will measure and compare the performance of these three implementations on matrices of various sizes.

Instructions

1. Implement the following versions of matrix multiplication:
   A. Serial Version (CPU)
      : Implement a basic nested loop algorithm on the host using for loops.
   B. Naive CUDA Kernel
      : Implement a GPU kernel that uses only global memory. Each thread computes one element of the result matrix.
   C. Tiled CUDA Kernel (Shared Memory)
      : Implement a GPU kernel that uses shared memory to load sub-tiles of the matrices, optimizing memory reuse. Your tiled implementation **must satisfy the following conditions**:
      - The shared memory size must be passed via the kernel execution configuration (i.e., using the third parameter of the <<< >>> syntax).
      - The kernel must correctly handle arbitrary matrix sizes, including cases where the matrix size is not a multiple of the tile size.
      - You must include appropriate boundary checks to avoid out-of-bounds memory access.

2. Matrix Sizes

: You must test your implementation on at least five different square matrix sizes, each size being 4×4 or larger. You are encouraged to explore additional sizes if resources allow.

3. Run performance experiments

- Run the multiplication **at least 20 times**
- Calculate and report the **average execution time** over those runs
- Use:
  - **cudaEvent_t** for timing GPU code
  - **std::chrono** for timing CPU code
- You may include a **dummy kernel** to warm up the GPU before measurements begin

4. Requirements

- Modularize your code clearly (separate functions for each kernel)
- Use float type for all matrix values
- Ensure correctness by comparing GPU results against the CPU implementation using a small epsilon tolerance.
- Your code must be **self-contained and automated**:
  When executed, it should run all required experiments (matrix sizes, implementations, and repetitions) without manual intervention or code modification.
  This includes:
  - Trying all specified matrix sizes
  - Running serial, naive, and tiled versions
  - Measuring performance (averaging over 20+ runs)
  - Verifying correctness
- You must **submit your complete solution folder** (including all .cu source files and helper files) **along with your report** in a single .zip file.
- The .zip file must be named in the following format: studentID_name.zip.
  For example: 20241234_JohnDoe.zip

5. Report

You must submit a report in **PDF format** that summarizes and analyzes your experimental results. Your report should include the following elements:

I. **Hardware and Runtime Environment**
   : GPU model, CUDA version, operating system, compiler version (e.g., nvcc --version) …

II. **Analysis and Discussion**
   : Analyze the performance of your implementations based on experimental results. Your discussion should address:

A. **Scalability:**
   How does execution time change with increasing matrix size for each implementation?

B. **GPU vs. CPU performance:**
   At what size does the GPU begin to outperform the CPU? What contributes to this difference?

C. **Tiled vs. Naive kernel:**
   How much improvement is achieved through shared memory? Under what circumstances is the improvement most significant?

D. **Speedup comparison:**
   Include calculated speedup ratios such as Serial / Naive, Serial / Tiled, and Naive / Tiled. Use visualizations (e.g., charts) that best communicate these trends.

E. **Block and tile size sensitivity:**
   Report how different configurations affect performance. Discuss which settings were optimal and why.

You are encouraged to choose chart types and visualization styles that best highlight your key findings. Your goal is not to show every number, but to present insights clearly and effectively.

Note

- If you are using **Visual Studio**, make sure to compile and run your program in Release mode, not Debug mode. Debug mode disables many optimizations and may significantly degrade performance, especially in CUDA kernels.
- If you are using **Google Colab** or a command-line environment, you should compile your CUDA code with appropriate optimization flags to reflect release-mode behavior. Use the following recommended settings with nvcc:
  nvcc -O3 your_code.cu -o your_executable
- Evaluation will be based on (1) the correctness and completeness of your implementation, and (2) how thoughtfully and systematically you analyze the results. Grading will be qualitative, not based on a checklist, but on overall effort and clarity.
- Late submissions will be accepted **until Sunday, June 1**.
  A **10% penalty** will be applied **for each day late**, based on the total score.
  Submissions **will not be accepted after June 1** under any circumstances.