GIK INSTITUTE
OF ENGINEERING SCIENCES AND TECHNOLOGY

2025/02

Student Name: Abdullah Ejaz Janjua

Reg: 2023038

Github: REPO LINK

Assignment No: 02

Instructor: Dr. Taj Khan

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Ghulam Ishaq Institue of Engineering Sciences and Technology, Topi, Swabi

1. Assuming you've setup the GPU coding environment in the last assignment, use the `(cudaGetDeviceProperties()` function to get the information about your particular GPU in the `cudaDeviceProp` structure. For each field give a one-line explanation and its value for your GPU, e.g., `devProp.clockRate`: represents GPU clock speed. Value = 2GHz. Find out also the max Global Memory memory bandwidth and the peak compute performance of this GPU.

   ANSWER: Please refer to code

2. Write a CPU program. It should multiply two, potentially non-square, matrice. The input data will be provided in a file. You can define the file structure (containing matrices' dimensions, elements) yourself and explain in your submission. The program when run will take max two filenames as argument. The first is the input data filename and the second (optional) argument is the output filename; if the output filename is absent it should write its output to stdout. The program will read input data (matrices) from the input file, store them in CPU memory, perform computations on them, write the result to output file in the same format structure you defined for input matrices. Compile run the code and push it to your github repo in the folder assignment02/task02. Use a build system i.e., Make, CMake, etc., for compilation. Write modular and parametrizable code. Activate compiler options which flag all warnings and report them. You program should compile with 0 warnings and 0 errors.

   ANSWER: The CPU program implements an $O(N \cdot M \cdot K)$ matrix multiplication. The input file structure is defined as follows to minimize parsing overhead:

   - Lines 1-3: Matrix dimensions N, K, and M on separate lines.
   - Line 4: Empty line delimiter.
   - Matrix A ($N \times K$) elements, space-separated, row by row.
   - Empty line delimiter.
   - Matrix B ($K \times M$) elements, space-separated, row by row.

   The program parses this file into 1D float arrays, computes the matrix multiplication sequentially, and outputs the result in the identical format to either an output file or `stdout`. It compiles warning-free using CMake/Make.

   The program utilizes the standard C++ `ifstream` and `ofstream` libraries to map these flat files into 1D dynamically allocated float arrays mapping to 2D indices via row-major order ($row \cdot \text{width} + col$). The build system utilizes a modular Makefile/CMake setup with `-Wall -Wextra` flags to ensure a robust, warning-free compilation.

3. Port the above program to GPU/CUDA. The program will read input data (matrices) from the input file, store them in CPU memory, copy them to GPU memory, perform operations on them, copy the result to CPU memory, and write it to output file in the same format structure you defined for input matrices. Compile run the code and push it to your github repo in the folder assignment02/task03. Use a build system i.e., Make, CMake, etc., for compilation. Write modular and parametrizable code. Activate compiler options which flag all warnings and report them. You program should compile with 0 warnings and 0 errors. Calculate also the computational intensity for your program.

   ANSWER: The program was ported to CUDA by allocating device memory (`cudaMalloc`), transferring host data (`cudaMemcpy`), and launching a 2D grid of threads (`dimBlock(32, 32)`) where each thread computes a single element of the output matrix.

   Computational Intensity (CI): CI is the ratio of floating-point operations (FLOPs) to memory traffic (bytes). In the naive kernel, calculating one output element requires K multiply-add operations ($2K$ FLOPs). To compute this, each thread reads $2K$ floats ($8K$ bytes) directly from global memory.

$$\text{CI}_{\text{naive}} = \frac{2K \text{ FLOPs}}{8K \text{ Bytes}} = 0.25 \text{ FLOPs/Byte}$$

4. Add code for measuring the computation time to both above programs. GPU time should include data transfer (to and from GPU) as well as computation time. Run both above programs for different matrix sizes and produce a graph of matrix size (x-axis) vs time (y-axis). Comment on what you see. Push it to your github repo in the folder assignment02/task04.

ANSWER: Execution time was measured using standard C++ chronological libraries for the CPU and CUDA events (`cudaEventRecord`) for the GPU. The GPU time includes both memory transfers (`cudaMemcpy`) and kernel execution.[1]

Observations: As observed in the logarithmic plot, the CPU execution time scales catastrophically as matrix dimensions increase. Both GPU implementations demonstrate orders of magnitude improvement over the CPU baseline. For the smaller dimensions, the Naive and Tiled GPU execution times are nearly indistinguishable, largely bottlenecked by the unavoidable overhead of PCI-e data transfers (`cudaMemcpy`). However, as the dimensions breach 512 and scale into the 1000+ range, a distinct divergence occurs, with the Tiled GPU consistently edging out the Naive implementation.

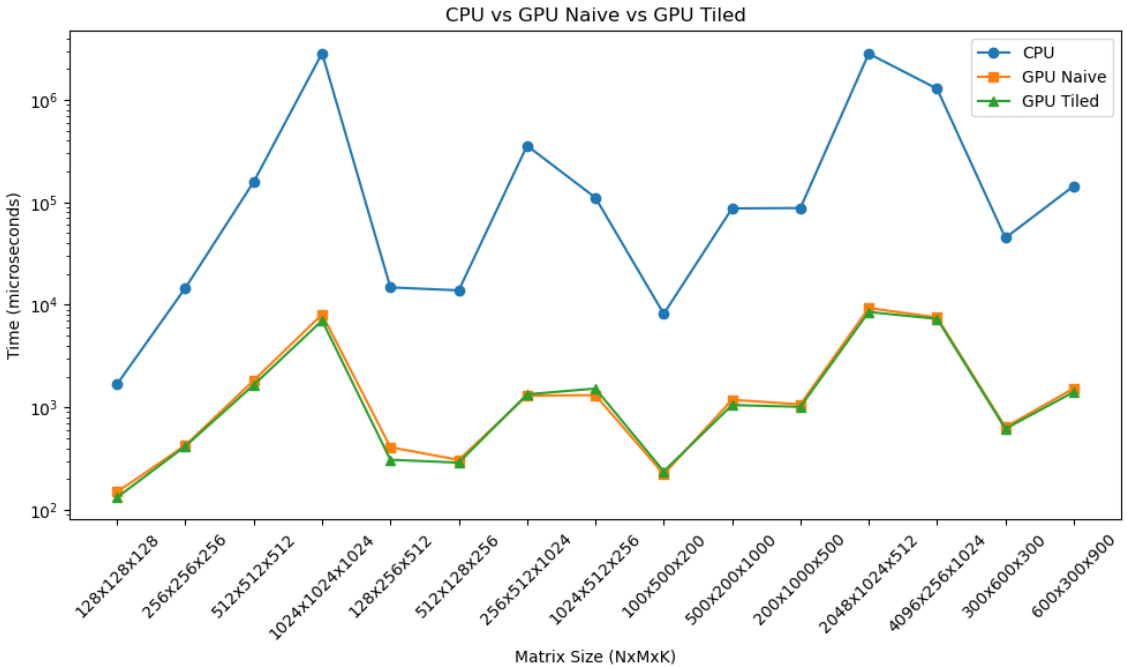

Figure 1: Matrix size vs Execution time (log scale).

5. Improve the performance of your GPU program by using tiling. Try different tile sizes to determine the maximum tile size your GPU shared memory can accommodate, and then use that maximum tile size to re-run the program with the same matrix sizes as in Task 4. Plot a time versus matrix-size plot showing the performance of three methods (CPU, GPU-Naïve, GPU-Tiled) for different matrix sizes. Comment on what you see. Push it to your github repo in the folder assignment02/task05. Calculate also the computational intensity for the tiled implementation and contrast it with the one in Task 3.

ANSWER: Shared memory tiling was implemented to optimize global memory accesses.

- **Maximum Tile Size Analysis:** According to the `cudaDeviceProp` queried in Task 1, our RTX A2000 has 48.00 KB of shared memory per block and allows a maximum of 1024 threads per block. A square tile uses two shared arrays of floats: `Mds[T][T]` and `Nds[T][T]`. The memory footprint is $2 \cdot T^2 \cdot 4$ bytes.
  - **Shared Memory Limit**: $8 \cdot T^2 \leq 49152 \implies T \leq 78$.
  - **Thread Limit**: A 2D block of size $T \times T$ requires T^2 threads. $T^2 \leq 1024 \implies T \leq 32$.

  Because we must respect the 1024 maximum threads per block limit, the absolute maximum theoretical and practical square tile size our GPU can accommodate is $\mathbf{32} \times \mathbf{32}$. This is precisely the `TILESIZE` implemented.
- **Observations:** As shown in Figure 1, the GPU-Tiled implementation outperforms the GPU-Naive implementation, especially on larger matrices, by minimizing redundant global memory fetches.
- **Computational Intensity Contrast:** With a tile size of $T = 32$, a block of $T \times T$ threads cooperatively loads tiles into shared memory. The memory traffic drops from 8 bytes per computation to $\frac{8}{T}$ bytes.

$$\text{CI}_{\text{tiled}} = \frac{2 \text{ FLOPs}}{(8/T) \text{ Bytes}} = \frac{T}{4} = \frac{32}{4} = 8 \text{ FLOPs/Byte}$$

  This represents a 32× improvement in computational intensity over the Naive implementation (0.25 FLOPs/Byte).

[1]Note: The graph generated (Figure 1) combines the performance results of the Tiled GPU implementation from Task 5 alongside the CPU and GPU-Naive results for direct comparison.