HW2: Implement Matrix Multiplication and Attention using CUDA

CSCE 654: Supercomputing
Due: `Wednesday, Sept 24 @ 11:59pm CT`
Total Points: 50

Academic Integrity & AI Tools

- **No LLM-generated code.** You may *not* use large language models (LLMs) to generate any code or scripts for this assignment (including source files, headers, build files, sbatch scripts, plotting scripts, etc.). If we determine that the code was generated by an LLM, the assignment will receive a **zero**.
- **Permitted use of LLMs.** You may use LLMs *only* to help you understand concepts (e.g., CUDA syntax, cuBLAS interfaces, algorithmic ideas) or to improve the clarity/grammar of your written report.
- **No copying from others or the internet.** Do not copy code from other students, prior solutions, or online sources (blogs, GitHub, Stack Overflow, etc.). We use automated and manual plagiarism checks across submissions and public sources.
- **Consequences.** Suspected violations will receive a zero and may be referred under university academic-integrity procedures.
- **Late policy.** Submissions lose **20%** of the total possible points for each late day, **except** when extensions are granted under university policy.

Learning Goals

- Learn to parallelize numerical kernels with CUDA.
- Understand performance trade-offs in thread organization and memory placement in the GPU.
- Reuse optimized dense matrix-matrix multiplication (called DGEMM) kernels to implement more complex operations.
- Compare library-based versus direct loop implementations.

Problem 1: Naive DGEMM (no blocking) 10 points

Goal: Implement a naive double-precision matrix multiply. Here, $A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}$, and the output $C \in \mathbb{R}^{m\times n}$.

Requirements:

- Use `dgemm_naive.cu` and write a host function that calls the device function. We provided function signatures. Feel free to modify the signature if you like.
  - In this naive version, each thread computes one entry of the output.
  - Use 2D blocks of threads (for example, 16x16).
- When you time the code, time computes only (excluding allocation/initialization).
- Compute GFLOP/s. Use $2mnk$ as the number of floating-point operations.

Experiments and report:

- In all experiments, use the input size: (m=10240, n=10240, k=10240). Generate input matrices randomly as shown in the starter code (`main.cu`).
- You can run your final experiments either on Grace or on Perlmutter. I recommend Perlmutter. You can use interactive allocation in Perlmutter.
- Ensure that your result is accurate by comparing your result with NVIDIA's cuBLAS. We provided a code to show how to call cuBLAS DGEMM and compute accuracy.
- Change your block dimensions and find the optimal block of threads. For example, use 1x1, 4x4, 16x16 blocks. Write the obtained GFLOP/s for different grids in the report. In your report, discuss which strategy worked the best.

Problem 2: Tiled DGEMM 20 points

Goal: Implement double-precision matrix multiply with tiling as discussed in the class. As before, $A \in \mathbb{R}^{m \times k}, B \in \mathbb{R}^{k \times n}$, and the output $C \in \mathbb{R}^{m \times n}$.

Requirements:

- Use `dgemm_tiled.cu` and write a host function that calls the device function.
  - In this version, each thread block computes one tile of the output.
  - Use 2D blocks of threads (for example, 16x16).
- When you time the code, time computes only (excluding allocation/initialization).
- Compute GFLOP/s. Use $2mnk$ as the number of floating-point operations.

Experiments and report:

- In all experiments, use the input size: (m=10240, n=10240, k=10240). Generate input matrices randomly as shown in the starter code (`main.cu`). Use 16x16 thread blocks by default.
- You can run your final experiments either on Grace or on Perlmutter. I recommend Perlmutter.
- Ensure that your result is accurate by comparing your result with NVIDIA's cuBLAS. We provided a code to show how to call cuBLAS DGEMM and compute accuracy.
- Change your tile size and find the optimal tiles. For example, consider tile sizes of 1, 4, 16, and 32. Write the obtained GFLOP/s for different tiles in your report. Explain which tiles work the best.

Problem 3: Attention via TiledDGEMM 20 points

Goal: Reuse your TiledDGEMM kernels to implement simple dot-product attention.

As in HW1, let $Q \in \mathbb{R}^{L \times D}$ be a dense matrix (called the query matrix), $K \in \mathbb{R}^{L \times D}$ be a dense matrix (called the Key matrix), and $V \in \mathbb{R}^{L \times D}$ be another dense matrix (called the Value matrix). For this homework, you do not need to know the meaning of these matrices. Then the attention matrix S and the transformed output O are computed as follows:

Math:

$$S = \frac{1}{\sqrt{D}} QK^\top, \qquad A = \text{softmax}(S), \qquad O = AV$$

Steps:

1. Use `attention.cu` to write your code.
2. Compute $K^\top$ explicitly (transpose $L \times D \to D \times L$). The actual computation should be done on the GPU.
3. Call your TiledDGEMM for $S = Q \cdot K^\top$.
4. Apply row-wise softmax to S (numerically stable). Write a CUDA kernel for this computation. See HW1 for an implementation of softmax.
5. Call your parallel TiledDGEMM for $O = A \cdot V$.

Experiments and report:

- Use L=10240, D=4096 in the following experiments. Generate all input matrices randomly as shown in the starter code (`main.cu`).
- You can run your final experiments either on Grace or on Perlmutter.
- Run your Attention code for various tiles (e.g., 1, 4, 16, 32) and various thread blocks (e.g., 1x1, 4x4, 16x16) and compute GFLOP/s. Summarize your results as a table or figure. Discuss the setting where the best performance is obtained.

What to Submit

- Your complete source code that can be compiled and run.
- SLURM job scripts used for the final experiment
- Your report showing the aforementioned results.
- Submit all files as a zipped folder.
- Use this naming convention for your submission: HW2_lastname_firstname.zip.

Grading

- **Correctness (35%)**: functions produce correct results.
- **Performance (40%)**: Based on obtained GFLOP/s.
- **Analysis (15%)**: clear observations about GPU programming.
- **Clarity (10%)**: clean code, reproducible runs, clear plots.