# HW1: Implement Matrix Multiplication and Attention using OpenMP

**CSCE 654: Supercomputing**
**Due:** `Monday, Sept 15 @ 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 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., OpenMP clauses, MKL 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 OpenMP (`parallel for`, `collapse`, `schedule`, `simd`).
- Understand performance trade-offs in loop order, memory access, and scheduling.
- Reuse optimized dense matrix-matrix multiplication (called DGEMM) kernels to implement more complex operations.
- Compare library-based versus direct loop implementations.

## Implementation Note

In all functions, matrices are stored as a single contiguous 1-D array and accessed through a pointer using row-major (C-style) layout. The element $A[i, j]$ is at offset `i*cols + j`, where `cols` is the number of columns. Thus we index as `A[i*cols + j]`. This layout improves cache locality.

## What We Provide

- A driver program (`hw_driver.cpp`) that generates random inputs, calls your functions, times them with `omp_get_wtime()`, and checks correctness on small cases.
- A Makefile with `-O3 -fopenmp`.
- BLAS link line for performance comparison if your cluster provides MKL.

## Problem 1: Naive DGEMM (no blocking) 10 points

**Goal:** Implement a naive double-precision matrix multiply and parallelize it with OpenMP.
**Function signature:**

```
void dgemm_naive(const double* A, const double* B, double* C,
                 int m, int n, int k);
```

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:**

- Start with a triple loop implementation.
- Add OpenMP:
  - Try `#pragma omp parallel for` over the outer loop.
  - Try `collapse(2)`.
  - Try `#pragma omp simd` in the innermost loop.
  - Experiment with `schedule(static)`, `schedule(dynamic)`, `schedule(guided)`.
- Time compute only (exclude allocation/initialization).
- Compute GFLOP/s. Use $2mnk$ as the number of floating-point operations.

**Experiments and report:**

- In all experiments, use the input size: (2048,2048,2048). Generate input matrices randomly as shown in the starter code.
- You can run your final experiments either on Grace or on Perlmutter. Please mention which supercomputer your used. You can use interactive allocation in Perlmutter. You can use your own computer for development and debugging.
- Ensure that your result is accurate by comparing your result with Intel's MKL. We provided a code to show how to call Intel's MKL DGEMM and compute accuracy.
- In your report, discuss which OpenMP strategy worked best (collapsing loops, simd, schedule, etc). Based on your experiments, just use the best strategy for the rest of the homework.
- Run your DGEMM code for threads: 1,2,4,8,16, 32 and compute GFLOP/s for each thread count. In the report, show a scaling plot (GFLOP/s vs. threads)

**Bonus Question:** Try to implement a blocked version for matrix multiplication. Please see https://malithjayaweera.com/2020/07/blocked-matrix-multiplication/. You will get an extra 5 points if you can implement a block version that is faster than the naive implementation.

## Problem 2: Naive Attention via DGEMM 10 points

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

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:**

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

**Steps:**

1. Compute $K^\top$ explicitly (transpose $L \times D \to D \times L$). Use OpenMP parallelism.
2. Call your parallel DGEMM for $S = Q \cdot K^\top$.
3. Apply row-wise softmax to $S$ (numerically stable). We provided an implementation for softmax. Add `pragma omp parallel for` in softmax over rows.
4. Call your parallel DGEMM for $O = A \cdot V$.

**Function signature:**

```
void attention_via_dgemm(const double* Q, const double* K, const double* V,
                         double* O, int L, int D);
```

**Experiments and report:**

- Use L=2048, D=1024 in the following experiments. Generate all input matrices randomly as shown in the starter code.
- You can run your final experiments either on Grace or on Perlmutter. Please mention which supercomputer your used. You can use interactive allocation in Perlmutter. You can use your own computer for development and debugging.
- Run your Attention_via_DGEMM code for threads: 1,2,4,8,16, 32 and compute GFLOP/s for each thread count. In the report, show a scaling plot (GFLOP/s vs. threads)

## Problem 3: Direct Attention Calculation without DGEM 30 points

**Goal:** Implement the same attention as Part 2 directly with loops, without DGEMM.
**Function signature:**

```
void attention_direct(const double* Q, const double* K, const double* V,
                      double* O, int L, int D);
```

**Requirements:**

- Compute scores row-by-row:
$$S[i,j] = \frac{Q[i,:] \cdot K[j,:]}{\sqrt{D}}$$
Here $Q[i,:] \cdot K[j,:]$ computes dot product of two $D$-dimensional vectors. You can use `#pragma omp simd` in computing dot products.
- Apply softmax row-wise.
- Accumulate for the $i$th row of the output.
$$O[i,:] = \sum_j A[i,j] V[j,:]$$
- Use OpenMP:
  - Parallelize over rows $i$,
  - Use `#pragma omp simd` in inner dot products. Avoid accessing any matrix along the column.

**Experiments and report:**

- Use L=2048, D=1024 in the following experiments. Generate all input matrices randomly as shown in the starter code.

- Compare the output from Attention_direct with Attention_via_DGEMM to measure correctness.
- You can run your final experiments either on Grace or on Perlmutter. Please mention which supercomputer your used. You can use interactive allocation in Perlmutter. You can use your own computer for development and debugging.
- Run your Attention_direct code for threads: 1,2,4,8,16, 32 and compute GFLOP/s for each thread count. In the report, show a scaling plot (GFLOP/s vs. threads)
- Which attention implementation is faster and why? Write your answer in the report.

## What to Submit

- `dgemm_naive.cpp`, `dgemm_blocked.cpp` **(optional)**, `attention_via_dgemm.cpp`, `attention_direct.cpp`.
- SLURM job scripts used for the final experiment
- Scaling plots and other answers (PDF).
- Submit all files as a zipped folder.
- Use this naming convention for your submission: HW1_lastname_firstname.zip.

## Grading

- **Correctness (35%)**: functions produce correct results.
- **Performance (40%)**: demonstrates speedup vs. 1 thread; DGEMM-based attention competitive.
- **Analysis (15%)**: clear observations about OpenMP.
- **Clarity (10%)**: clean code, reproducible runs, clear plots.