Introduction

- **Title:** Accelerating Scaled Dot-Product Attention using OpenMP and CUDA
- **Course:** High Performance Computing
- **Presented By:** Saurabh Pal, Sundar R, Aryan Sharma
- **Date:** April 5th, 2025

Project Introduction

- **Project Overview:**
  - Aims to accelerate the Scaled Dot-Product Attention mechanism, a core component of modern AI models like Transformers.
  - We will focus on three common variants: Self-Attention, Cross-Attention, and Multi-Head Attention.
  - The goal is to develop and compare parallel implementations using OpenMP (for CPUs) and CUDA (for GPUs) against a sequential C++ baseline.
  - Why both CPU and GPU? Because low powered devices need inference too, such as RPIs.
- **Significance in HPC:**
  - Attention mechanisms are computationally demanding, especially with large datasets and models.
  - They scale quadratically with the sequence length.
  - Efficient parallel implementations are crucial for both training and inference after deploying these models.
  - This project explores applying HPC techniques (CPU/GPU parallelism) to solve a real-world computational bottleneck in AI.
  - Provides practical experience in performance optimization using standard parallel programming models.

Background: Attention Mechanisms

- **Core Concept:** Scaled Dot-Product Attention allows a model to weigh the importance of different parts of an input sequence (Values) based on their relationship with a query (Query) and other parts of the sequence (Keys).
  - Calculates attention scores:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- **Paper Studied:** We based our initial understanding on the concepts presented in the foundational paper:
  - "Attention Is All You Need" by Vaswani et al. (2017). This paper introduced the Transformer architecture and scaled dot-product attention.
  - "Attention in transformers, step-by-step | DL6" video by 3Blue1Brown
- **Variants Explored:**
  - **Self-Attention:** Input sequence attends to itself (Q, K, V derived from the same input).
  - **Cross-Attention:** One sequence attends to another (Q derived from one input, K and V from another).
  - **Multi-Head Attention:** Performs attention multiple times in parallel with different learned projections, allowing the model to focus on different aspects simultaneously. Like over-powered Self-Attention.

Project Objectives

- **Overall Goal:** To significantly improve the performance of scaled dot-product attention computations through parallelization.
- **Specific Aims:**
  - Develop a correct sequential C++ implementation of Self, Cross, and Multi-Head Attention to serve as a baseline. (Completed for Eval 1)
  - Implement parallel versions using OpenMP for multi-core CPU execution.
  - Implement parallel versions using CUDA for execution on NVIDIA GPUs.
  - Benchmark and analyze the performance (speedup) of the parallel implementations compared to the sequential baseline.
  - Investigate the impact of problem parameters (sequence length, embedding dimension, number of heads) on performance across different implementations.

Progress So Far (Evaluation 1)

- **Conceptual Understanding:** Gained familiarity with the mathematical formulation and purpose of Self, Cross, and Multi-Head Attention mechanisms via the Vaswani et al. paper.
- **Sequential Implementation:**
  - Developed C++ code implementing the core logic for all three attention types.
  - Focused on correctness and establishing a functional baseline.
- **Initial Benchmarking:**
  - Measured the execution time of the sequential C++ code using `std::chrono`.
  - Tested various configurations of sequence length, embedding dimensions, and number of heads to understand performance characteristics.
- **System Setup:** Prepared the development environment on the target hardware.

Methodology & Setup

- **Core Algorithms Involved:**
  - Matrix Multiplication (for Q*K^T and the final weighted sum)
  - Matrix Transposition (for K^T)
  - Softmax function (for normalization of attention scores)
  - Vector Embeddings / Projections (Implicit in generating Q, K, V - handled by matrix operations in this context)
  - Element-wise operations (scaling by `sqrt(d_k)`)
- **Tools & Technologies:**
  - Programming Language: C++ (`-std=c++20`)
  - CPU Parallelism: OpenMP
  - GPU Parallelism: CUDA
  - Compiler: g++ (with flags `-O3 -march=native -Wall -Wextra`)
- **Hardware Resources:**
  - CPU: Intel Core i5 (12th Generation)
  - GPU: NVIDIA GeForce RTX 3050

Initial Benchmarking: Sequential Performance

- **Purpose:** To establish baseline performance figures for future comparison.
- **Compilation Command:** `g++ -std=c++20 -O3 -march=native -Wall -Wextra benchmark.cpp -o benchmark`
- **Sequential Execution Results:** *(Measured using `std::chrono`)*

Seq Length	Emb Dim	Num Heads	Self (ms)	Cross (ms)	Multi (ms)	Memory
256	128	8	11.24	11.12	97.69	660 KiB
256	128	32	11.36	11.24	366.46	640 KiB
256	512	8	335.79	277.70	2679.66	4 MiB
256	512	32	271.68	250.36	8696.79	4 MiB
2048	128	8	90.88	90.97	801.00	18 MiB
2048	128	32	90.55	90.28	2944.49	18 MiB
2048	512	8	2346.03	2446.14	21036.67	25 MiB
2048	512	32	2292.63	2340.37	76268.52	25 MiB

- **Observations:** Performance degrades significantly with increased sequence length, embedding dimension, and number of heads. Multi-Head attention shows the highest computational cost.

Future Work & Next Steps

- **Immediate Goals (Towards End Evaluation):**
  - Implement parallel Self, Cross, and Multi-Head Attention using OpenMP directives.
  - Develop efficient CUDA kernels for the core matrix operations and softmax function.
  - Integrate CUDA kernels into complete parallel implementations for the GPU.
  - Perform thorough benchmarking of OpenMP and CUDA versions.
  - Analyze speedup and efficiency compared to the sequential baseline.
- **Potential Optimizations & Exploration:**
  - **CUDA Kernel Fusion:** Combine multiple small GPU operations (e.g., scaling, softmax) into single kernels to reduce memory transfer overhead.
  - Investigate memory access patterns and optimize for cache locality (OpenMP) or shared memory usage (CUDA).
  - (If time permits) Explore concepts from optimized libraries/algorithms like FlashAttention (focuses on reducing memory I/O) or linear attention mechanisms as potential further improvements.