Introduction

- **Title:** Accelerating Scaled Dot-Product Attention using OpenMP and CUDA
- **Course:** High Performance Computing
- **Presented By:** Saurabh Pal, Sundar R, Aryan Sharma
- **Date:** May 10th, 2025

Project Introduction

- 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 to HPC

- Attention mechanisms scales quadratically with the sequence length.
- Efficiency is crucial for both training and inference after deploying these 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:

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

- **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 an over-powered self-Attention.

Literature Review

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.

Some other resources:

- "Attention in transformers, step-by-step | DL6" lecture by 3Blue1Brown
- "GPT from scratch" lecture by Dr. Andrej Karpathy

Project Objectives

- Develop a correct sequential C++ implementation of Self, Cross, and Multi-Head Attention to serve as a baseline.
- Implement parallel versions using OpenMP for multi-core CPU execution.
- Implement parallel versions using CUDA for execution on NVIDIA GPUs.
- Profile 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.

Methodology & Setup

- **Tools & Technologies:**
  - Programming Language: GNU C++ (`-std=c++23 -O2 -Wall -Wextra`)
  - CPU Parallelism: GNU OpenMP v4
  - GPU Parallelism: CUDA 12.8
- **Hardware Resources:**
  - CPU: Intel Core i5 12th Gen
  - GPU: NVIDIA GeForce RTX 3050

Correctness Verification

- The sequential version is a port of a very simple implementation of single and multi-head attention mechanism.
- The output attention matrices from OpenMP and CUDA implementations is compared against the output of sequential version.
- Shapes of matrices are compared and each value is required to have absolute difference <= 1e-4.
- Verified all attention types produce matching results using a `verify.py` script.

OpenMP Parallel Implementation

- Utilize `#pragma omp parallel for` to parallelize compute-heavy loops and leverage multi-core CPUs for efficient execution.
- Ensure thread safety by using local accumulators (e.g., per-thread variables for summing during matrix multiplication and softmax) and by avoiding shared writable state inside parallel regions.
- Apply parallelism to key operations:
  - **Matrix multiplication ($Q \times K^t$)**:
    - Parallelize the outer row loop in `matmul`, allowing each thread to compute a different row of the result matrix independently.
  - **Scaling and softmax**:
    - Use `#pragma omp parallel for collapse(2)` for (row, col) parallelism in attention score scaling and masking.
    - Compute softmax per row in parallel, enabling each thread to normalize one row of the score matrix.

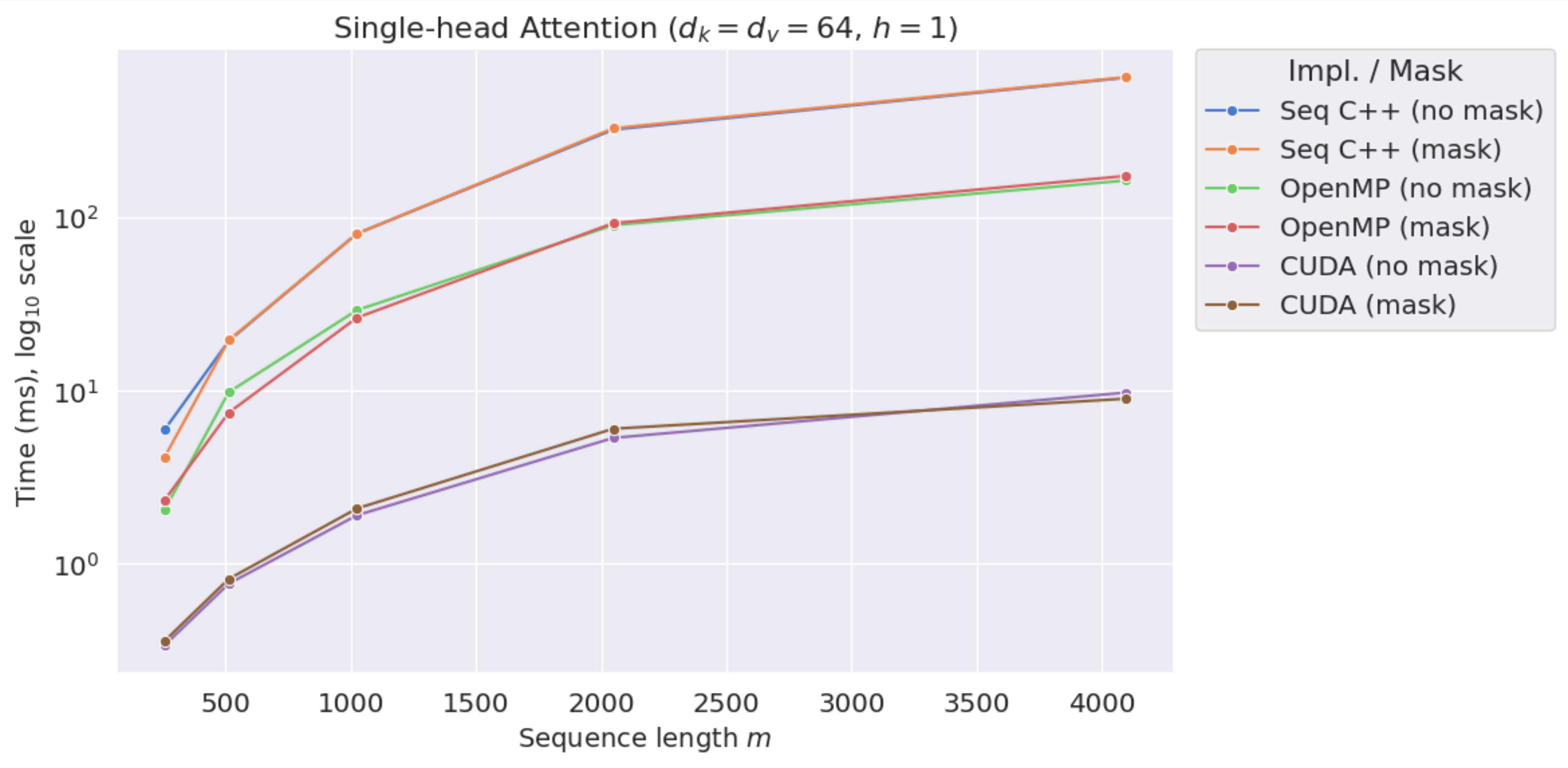
Thread Optimization

- Tuned the `OMP_NUM_THREADS` environment variable to match hardware capabilities, maximizing throughput without oversubscribing CPU resources.
- Observed that performance scales well up to 8 threads on a 12-core CPU, with diminishing returns or saturation beyond this point due to memory bandwidth and thread management overhead.
- Experimented with thread counts to find the optimal balance between parallel speedup and resource contention, ensuring stable and reproducible results across different runs.

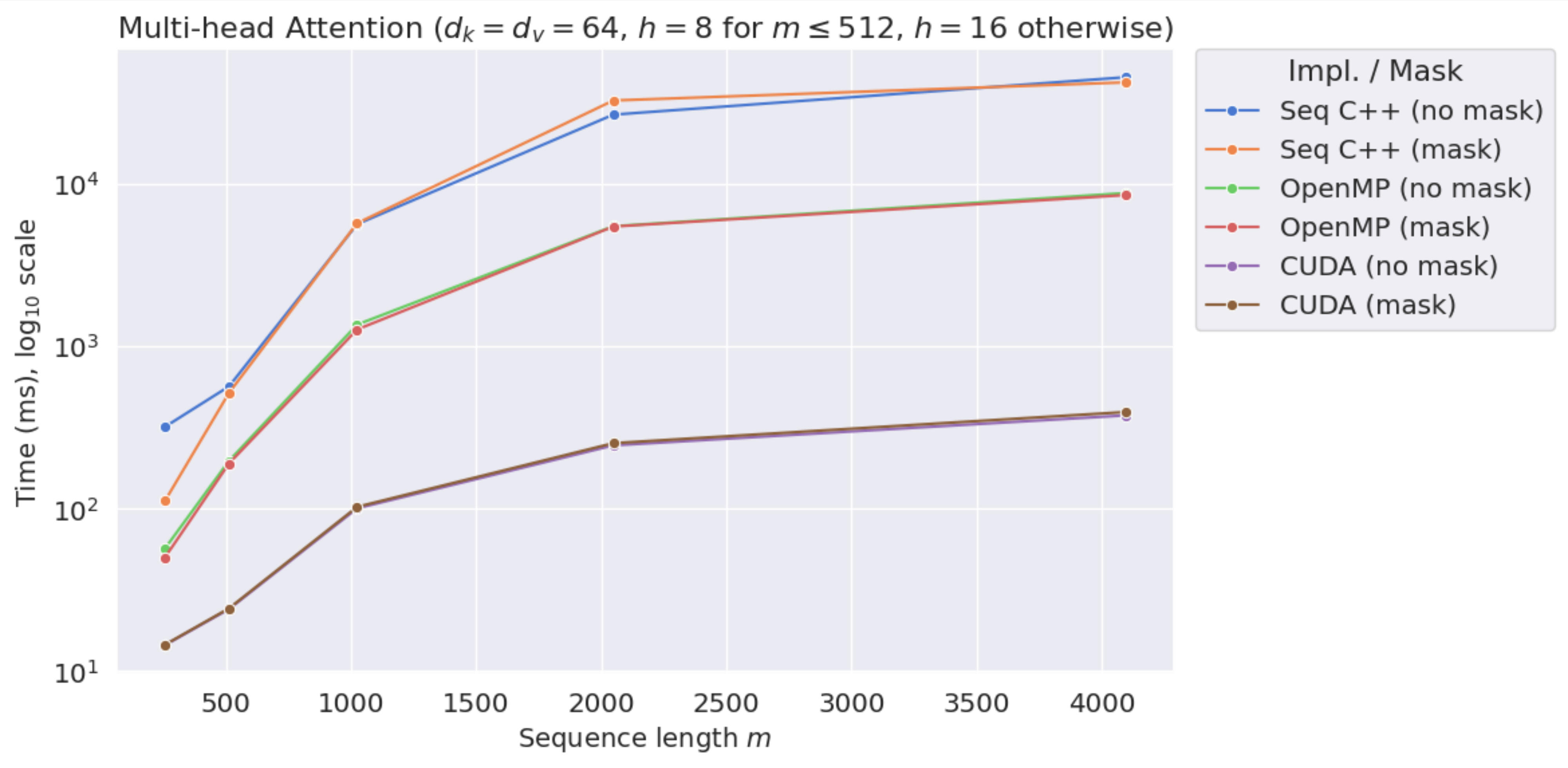
CUDA Parallel Implementation

- **CUDA Kernel Breakdown**
  - `matmul(int M, int N, int K, const float* A, const float* B, float* C)`
  - `softmax(int M, int N, const float* A, float* B)`
  - `scale(int M, int N, const float* A, float* B, float scale)`
  - `mask(int M, int N, const float* A, float* B)`
  - `transpose(int M, int N, const float* A, float* B)`
- **Optimization Techniques**
  - Utilized `__shared__` memory for block-level caching.
  - Applied `cudaDeviceSynchronize` for synchronization.
  - Ensured memory coalescing for all device loads/stores.

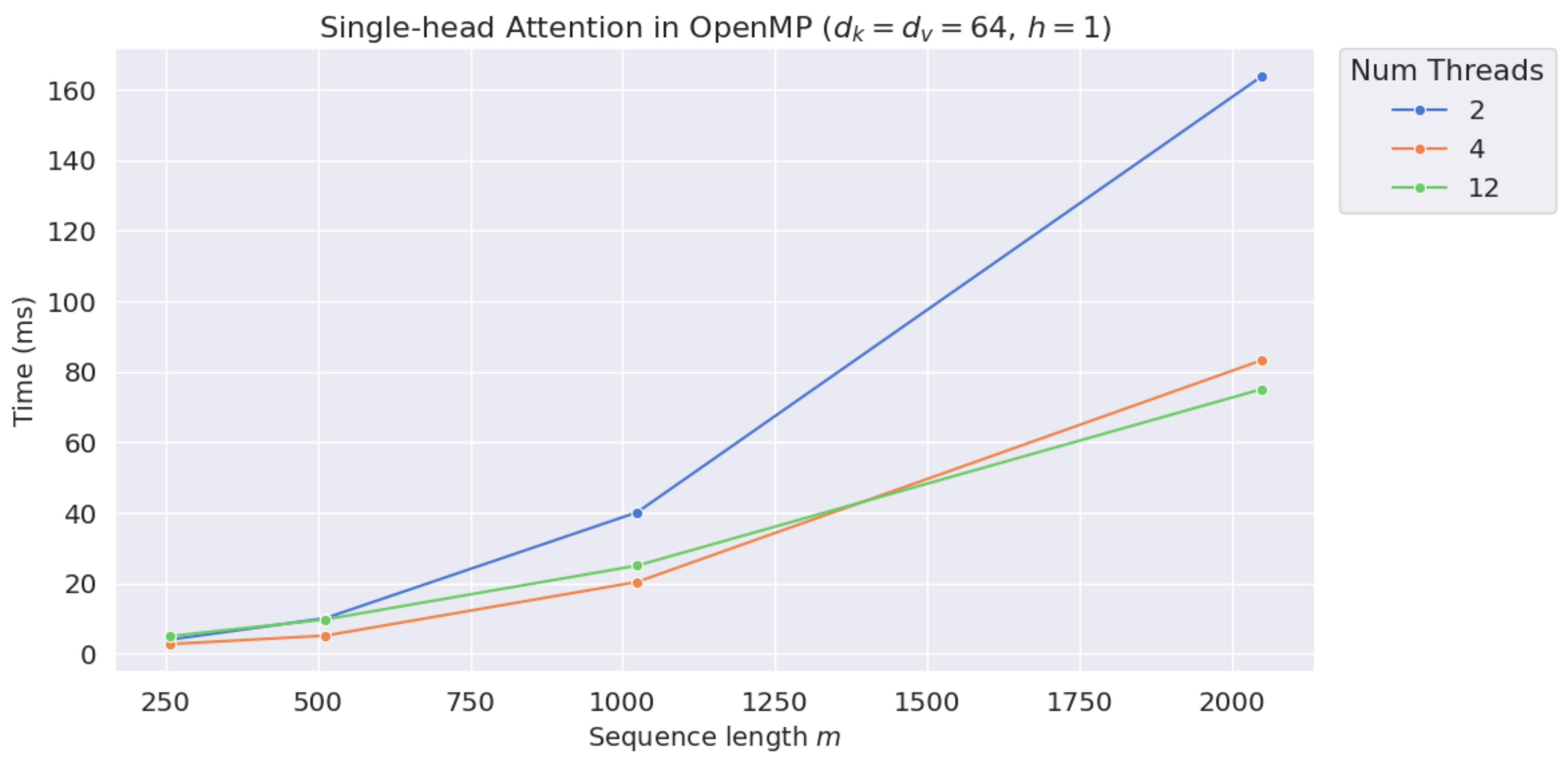
Single-Head Attention Mechanism Benchmarks



Multi-Head Attention Mechanism Benchmarks



Single-Head Attention Mechanism in OpenMP with Varying Num Threads



Single-Head Attention Mechanism in CUDA with Varying blockDim