MultiLLM-Using Multiple Cores in a CPU for Communication-Aware LLM Inference Acceleration

SUPERVISED BY *Prof. Sumit Kumar Mandal*

TEAM MEMBERS *Gayatri Madduri (25804)*
Kaushikkumar Rathva (25630)
Siddharth Sethi (25248)
Vedant Balasubramaniam (25024)

Blueprint of the Presentation

1. LLM: Problems and Challenges
2. Landscape of Distributed LLM Inferences
3. Overview of MultiLLM
4. Methodology
5. Results
6. References

LLM: Problems and Challenges

Large Language Models(LLMs) are "large" models with long inference times. Hence, acceleration is essential.

! High Memory

! Slow Inference

! High Energy Use

Multicore CPUs are optimized for low-latency, high-frequency applications, have higher memory capacity, and are more energy-efficient.

Multiple cores are required for leveraging *parallelism* effectively.

Many cores with lower performance can yield better results than a faster single core in the case of highly parallelizable code.

Problem: Communication *between* CPU cores can be slow and varies significantly depending on the core pair. This becomes a major performance limiter.

GOAL Develop a **Communication-Aware Mapping Strategy**.

- ✓ **Communication-Aware** inference ensures minimization of synchronization-time overhead.
- ✓ Profile the target CPU to understand inter-core latencies.
- ✓ Map LLM layers intelligently, placing frequently communicating layers on low-latency core pairs.
- ✓ Minimize communication overhead for faster, real-time inference.

Landscape of Distributed LLM Inference

Models often too large for one core; parallelism needed for speed.

Common Types of Parallelism

Data Parallelism

Tensor Parallelism

Pipeline Parallelism

Existing Optimization Approaches

1 **Overlap Communication & Computation:** Hide data transfer time while performing computations

2 **Reduce Communication Volume:** Use quantization or compression

3 **Advanced Communication Primitives:** Hierarchical All-to-All, Collective Decomposition

4 **Heterogeneous Scheduling:** Leverage CPU + GPU

CHALLENGES: Imperfect overlap, Bandwidth limits

TRADE-OFF: Potential accuracy loss

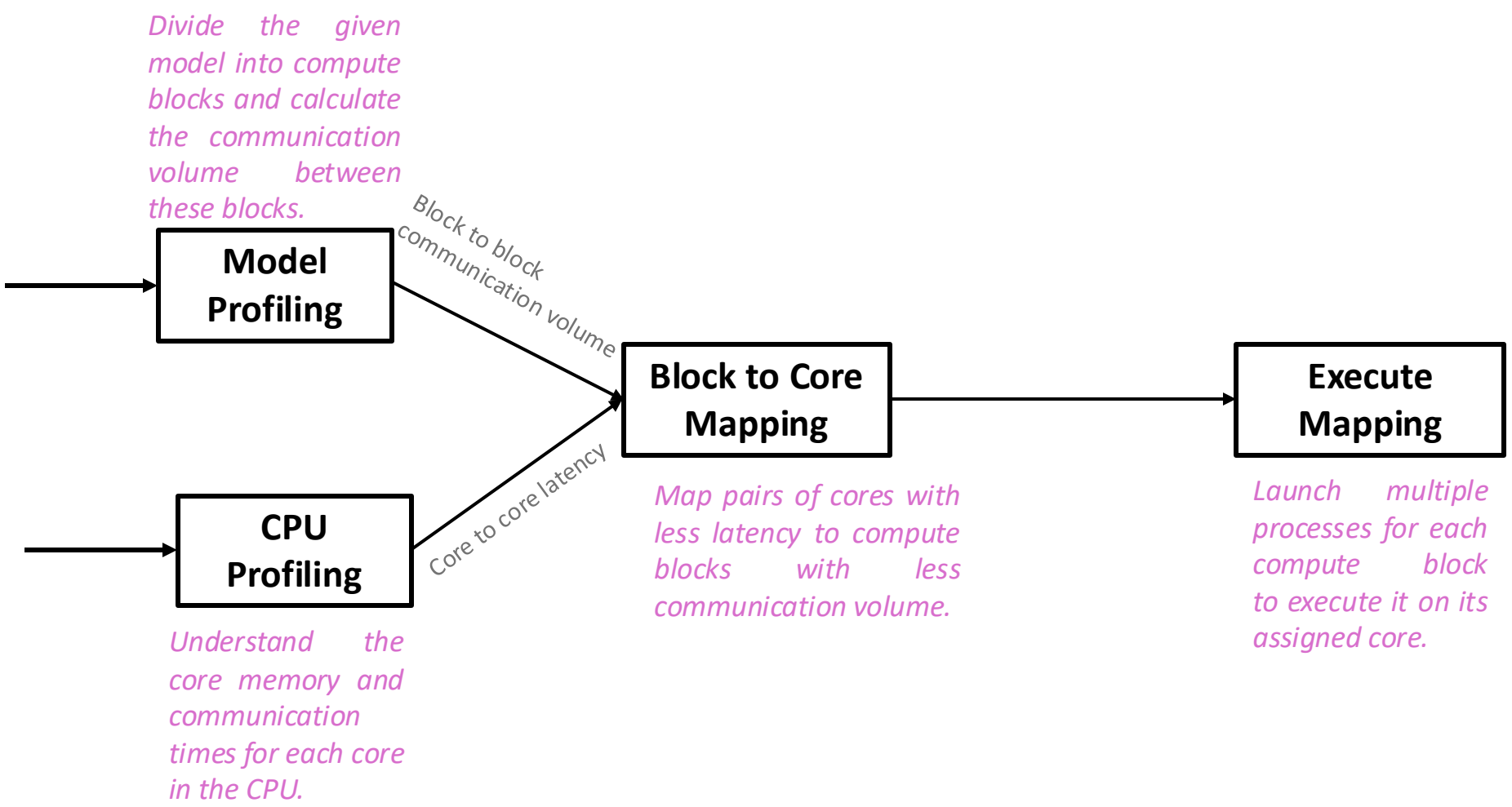
Often for large multi-node training

Focus on real-time DNNs

How our we doing it different?

- Focus on **Intra-CPU Inference Mapping**.
- Use **Empirical, Hardware-Specific Inter-Core Latency Measurements** to guide mapping decisions.
- Directly tackle the often-overlooked communication bottleneck *within* a single multicore chip.

Overview of MultiLLM



Continue the process for all layers in an LLM.

Methodology

CPU Profiling

- We measure the communication time between two cores in CPU by transmitting an unsigned integer to the other core.
- This module doesn't have any inputs.
- The output is the number of cores, and pair-wise communication latency between the cores.
- We also take note of the memory within each core to determine if a compute block can fit in a core.

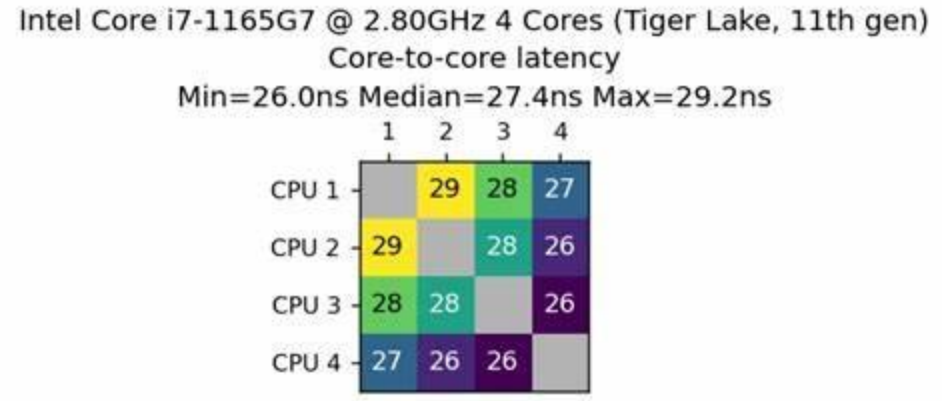
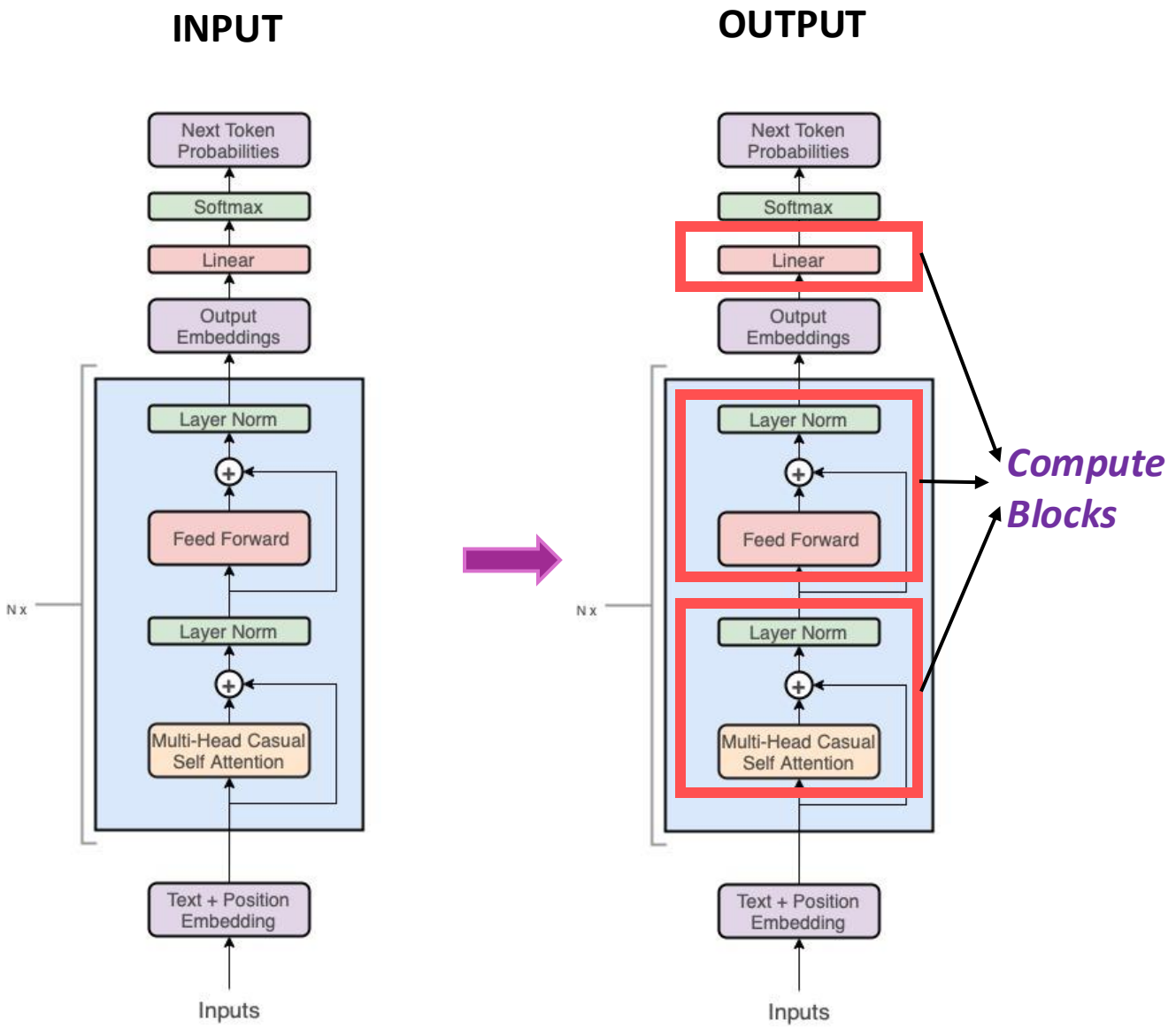


Figure : *An example of output from CPU profiling: A matrix,$[a_{ij}]$, of size N×N, where N is the total number of cores.*

Methodology

Model Profiling

- Identify sub-layers within a single Linear Layer(Self-Attention, RMS Norm, Feed-Forward, SwiGLU, Softmax, Linear).
- Group into **Compute Blocks** (units for mapping). Split large sub-layers if needed (memory constraint).
- Determine for each block:
  - **Memory Requirement:** Weights + Activations.
  - **Communication Volume:** Size of data transferred to the *next* block (e.g., activation tensor size).



Methodology

Layer to Core Mapping

Goal Assign each compute block to a CPU core.

Constraints Core memory capacity, execution order dependencies.

Objective: Minimize total communication overhead $\sum Comm_{Vol} - Comm_{Latency}$

MAPPING APRROACHES

Exhaustive Search

1. Try *all* valid block-to-core assignments.
2. Calculate total communication cost for each.
3. Select the absolute minimum.

Drawback: Computationally infeasible (combinatorial explosion).

Heuristic Method (Algorithm 2):

💡 **Idea:** Place blocks with high communication volume (consecutive blocks) on core pairs with low latency.

1. For each core c_1, find closest (c_2) and 2nd closest (c_3) cores (latency-wise).
2. When mapping block k to c_1, try mapping block *k+1* to c_2. If c_2 is taken, try c_3.

Benefit: Fast, practical approximation.

Output: A dictionary {Compute Block ID: Assigned Core ID}

Methodology

Bringing it Together - Multi-Core Execution

Algorithm 3: Process Creation & Execution

Input: The Block-to-Core mapping dictionary.

1. For each Compute Block: Create a dedicated OS Process (`multiprocessing.Process`) targeting the layer's function.
2. Pin Processes: Use **CPU Affinity** (`os.sched_setaffinity`) to lock each process to its assigned core ID from the mapping. ***Crucial step!***
3. Setup Communication: Create IPC Pipes/Queues between processes for connected blocks.

Execution Pipeline Flow

1. Input sent to Process 1 (on assigned Core A).
2. Process 1 computes, sends output via Pipe to Process 2 (on assigned Core B).
3. Continues sequentially through all blocks/processes/cores.
4. Final output collected from the last process.

BENEFIT: Enables continuous, pipelined processing for real-time applications..

System Specifications

- Architecture: x86_64
- CPU(s): 256
- Thread(s) per core: 2
- Model name: AMD EPYC 9554 64-Core Processor
- L1d cache: 4 MiB
- L1i cache: 4 MiB
- L2 cache: 128 MiB
- L3 cache: 512 MiB

Results

For Llama

Without Parallelization		With Parallelization	
Quantity	**Value**	**Quantity**	**Value**
Cycles	4,076,951,477,277	Cycles	10,259,596,330,964
Instructions	1,113,062,923,380	Instructions	1,713,560,742,716
Total time Elapsed	85.116757868 s	Total time Elapsed	159.833707000 s
Page Faults	11,509,267	Page Faults	14,470,785

Results

For GPT2_PICO

Without Parallelization		With Parallelization	
Quantity	**Value**	**Quantity**	**Value**
Cycles	6,356,796,306,034	Cycles	24,020,593,767,842
Instructions	2,882,026,219,170	Instructions	16,195,943,974,725
Total time Elapsed	31.674932681 s	Total time Elapsed	38.135483314 s
Page Faults	820,826	Page Faults	1,053,585

References

- Z. Wang, H. Lin, Y. Zhu, and T. S. E. Ng, “Hi-speed dnn trainingwith espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies,” EuroSys ’23.
- N. Viennot, “Core-to-core latency,” https://github.com/nviennot/core-to-core-latency.git 2025.
- W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, “Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks,”RTSS
- L. Zhao, W. Gao, and J. Fang, “Optimizing large language models on multi-core cpus: A case study of the bert model,” 2024.