WaferLLM: Large Language Model Inference at Wafer Scale

Congjie He[1] Yeqi Huang[1] Pei Mu[1] Ziming Miao[2] Jilong Xue[2] Lingxiao Ma[2] Fan Yang[2] Luo Mai[1]

[1]*University of Edinburgh* [2]*Microsoft Research*

Abstract

Emerging AI accelerators increasingly adopt wafer-scale manufacturing technologies, integrating hundreds of thousands of AI cores in a mesh architecture with large distributed on-chip memory (tens of GB in total) and ultra-high on-chip memory bandwidth (tens of PB/s). However, current LLM inference systems, optimized for shared memory architectures like GPUs, fail to exploit these accelerators fully.

We introduce WaferLLM, the first wafer-scale LLM inference system. WaferLLM is guided by a novel PLMR model (pronounced as "Plummer") that captures the unique hardware characteristics of wafer-scale architectures. Leveraging this model, WaferLLM pioneers wafer-scale LLM parallelism, optimizing the utilization of hundreds of thousands of on-chip cores. It also introduces MeshGEMM and MeshGEMV, the first GEMM and GEMV implementations designed to scale effectively on wafer-scale accelerators.

Evaluations show that WaferLLM achieves up to 200× higher accelerator utilization than state-of-the-art methods. Leveraging a wafer-scale accelerator (Cerebras WSE2), WaferLLM delivers GEMV operations 606× faster and 16× more energy-efficient than on an NVIDIA A100 GPU. For full LLM inference, WaferLLM achieves 10–20× speedups over A100 GPU clusters running SGLang and vLLM. These advantages are expected to grow as wafer-scale AI models, software, and hardware continue to mature. WaferLLM is open-sourced at https://github.com/MeshInfra/WaferLLM.

1 Introduction

Large Language Model (LLM) inference is a rapidly growing workload. It has two phases [15]: (i) the *prefill phase*, which processes input tokens (the prompt) and spends most of its cycles on General Matrix Multiply (GEMM); and (ii) the *decode phase*, which generates tokens one by one in an autoregressive manner, primarily performing General Matrix-Vector Product (GEMV). Decode requires repeatedly loading the entire LLM model into on-chip memory, with GEMV dominating its cycles. Since LLMs generate many tokens, especially in the test-time scaling scenario, such as the OpenAI-o1/o3 [16, 33] and DeepSeek-R1 [13], inference is constrained by GEMV latency, making it inherently memory-bandwidth-bound.

To address memory bandwidth bottlenecks, AI accelerators are increasingly adopting system-on-wafer integration [21]. This approach scales chip area to a full wafer, up to 100× larger than a typical GPU die, enabling significantly more on-chip cores, memory, and bandwidth. Examples include Cerebras WSE [24] and upcoming Tesla Dojo [41]. The Cerebras WSE-2, for instance, integrates 850,000 cores with 40GB of on-chip memory, 1,000× more than GPUs, and provides 22PB/s memory bandwidth, 7,000× higher than GPUs. TSMC anticipates widespread adoption of system-on-wafer integration, citing advantages in performance, energy-efficient die-to-die communication, and cost reduction. IEEE similarly forecasts a surge in wafer-scale computing by 2027 [21]. Wafer-scale accelerators are already seeing real-world deployment, particularly in model serving. In February 2025, Mixtral and Perplexity adopted wafer-scale chips, achieving cost parity with GPUs in terms of tokens per dollar [40, 44]. G42 now operates data centers fully outfitted with wafer-scale accelerators, and Cerebras has secured major commercial contracts [39].

Unlocking the potential of wafer-scale accelerators is challenging because current LLM systems rely on *shared memory architectures* typical of GPUs and TPUs. Wafer-scale accelerators, however, adopt *network-on-chip* (NoC) designs that interconnect millions of cores with local memory in a *massive-scale, mesh-based memory architecture*. This architecture far exceeds the scale of on-chip crossbars (e.g., one-hop NUMA such as GraphCore IPU), multi-socket NUMA [2], and high-density AI clusters (hundreds of GPUs per pod) [17]. Without fully addressing this fundamental shift in memory architecture, directly applying designs from state-of-the-art systems like T10 [25] and Ladder [45] to wafer-scale devices often results in extremely poor performance.

To address these challenges, we propose a *device model* that captures the critical hardware properties of wafer-scale accelerators, highlighting key differences from shared-memory devices. This model enables us to evaluate current LLM inference design principles, identify non-compliant areas, and pinpoint where new approaches are required. Guided by this model, we can achieve an ambitious system design: *running complete LLM inference on a single chip*, minimizing costly off-chip communication and maximizing on-chip memory bandwidth utilization.

The above idea motivates WaferLLM, the first wafer-scale LLM inference system, yielding several contributions:

(1) Device model for wafer-scale accelerators. We propose the PLMR model[1], which captures the key hardware properties of wafer-scale accelerators: (i) Massive **P**arallel cores (P): Millions of cores can be integrated on a large wafer, requiring systems to effectively partition LLMs and their operations. (ii) Highly non-uniform memory access **L**atency (L): Inter-core data access exhibits significant variation, with latency differences up to 1,000×, necessitating the system to mitigate this. (iii) Constrained per-core local **M**emory (M): Each core has limited memory (tens of KBs to several MBs), requiring efficient memory usage. (iv) Limited hardware-assisted **R**outing (R): The NoC routing hardware is constrained by the area size per core and can only support a limited number of routing paths, e.g., less than 2^5 on Cerebras WSE-2.

(2) Wafer-scale LLM parallelism. We propose an effective, PLMR-compliant LLM parallelism policy for wafer-scale accelerators. In the prefill phase, we design fine-grained partitioning to achieve million-core parallelism. For the decode phase, where tensor dimensions are insufficient for partitioning, we design fine-grained replication to enable parallelism with minimal communication costs. As a result, WaferLLM achieves larger-scale and finer-grained parallelism (satisfying P in PLMR) than GPU-based approaches. Additionally, we replace conventional GPU-based GEMM and GEMV operators with new algorithm designed for the PLMR model (satisfying L, M, and R) and propose tensor placement strategies that eliminate matrix transpositions, which are costly with a mesh NoC (satisfying L).

We also designed a scalable KV-cache management method for wafer-scale devices. This approach features a novel KV cache shift method to ensure balanced core usage (satisfying P and M), avoiding skewed utilization of cores caused by KV cache concatenation methods common on GPUs.

(3) Wafer-scale GEMM. We propose MeshGEMM, a scalable GEMM algorithm for wafer-scale devices, accelerating the prefill phase. Unlike conventional distributed GEMM algorithms, MeshGEMM achieves full PLMR compliance by leveraging two key operations: *cyclic shifting* and *interleaving*. Cyclic shifting ensures algorithm correctness while maintaining bounded usage of local memory (satisfying M). The interleaving operation minimizes communication latency in the mesh NoC, effectively reducing the overhead of highly non-uniform memory latency (satisfying L) and routing resources (satisfying R).

(4) Wafer-scale GEMV. We propose MeshGEMV, a scalable GEMV algorithm for wafer-scale devices, accelerating the decode phase. Unlike existing GEMV implementations, MeshGEMV uses a novel *K-tree allreduce* algorithm to aggregate local GEMV results across massive cores. This algorithm ensures routing resource usage meets the hardware's limitation (satisfying R) and reduces communication latency (satisfying L).

[1]PLMR model can be pronounced as "Plummer"

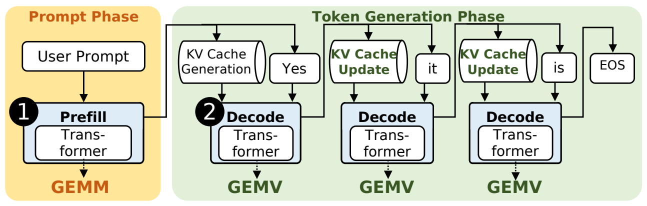

Figure 1: Key components in LLM inference

We implemented WaferLLM on the Cerebras WSE engine using approximately 7,000 lines of CSL (a C-like programming language) for LLM parallelism, MeshGEMM, and MeshGEMV, and 2,000 lines of Python for loading LLM checkpoints, launching inference, and execution parallelism policy.

We conducted end-to-end LLM inference experiments with various models, including full LLaMA3-8B and LLaMA2-13B, as well as subsets of layers of CodeLLaMA-34B, and QWen2-72B. By combining wafer-scale LLM parallelism, GEMM and GEMV, WaferLLM outperforms state-of-the-art (SOTA) systems: (i) 100-200× faster than T10 [25], the SOTA system for massive cores with a distributed on-chip memory architecture, and (ii) 200-400× faster than Ladder [45], the SOTA system for shared-memory architectures.

Micro-benchmarks further show that MeshGEMM is 2-3× faster than SUMMA [42], the default optimized GEMM for Cerebras WSE, and Cannon [6], the SOTA GEMM for supercomputers with large-scale mesh architectures. MeshGEMV achieves 4-8× speedups over Cerebras's optimized GEMV, and 606× than a single A100 GPU. Additionally, WaferLLM 's cache shift method is up to 400× more scalable than the KV cache SOTA on GPUs, such as PagedAttention [20].

Combined, WaferLLM (on Cerebras WSE-2) outperforms SGLang (on single A100) by 30-40×. Compared to the optimal performance of SGLang on A100 multi-GPUs connected with NVLink and RDMA, WaferLLM delivers a 10-20× faster end-to-end speed and is 2.5× more energy-efficient. The reduced gains from GEMV to LLM are due to current limitations in software, hardware, and existing LLM model designs. We anticipate stronger performance as wafer-scale AI computing matures and these limitations are addressed.

2 Background and Motivation

2.1 LLM inference and its key constraint

An LLM inference system typically performs auto-regressive token-by-token generation, as illustrated in Figure 1. The model comprises multiple transformer layers, dominated by self-attention and feedforward blocks. Inference operates in two phases: prefill and decode. The total cycles of the prefill phase are dominated by GEMM operations (shown by ❶). While the total cycles of the decode phase are dominated by

	System-on-Die	System-on-Wafer
Area (mm^2)	Up to 858	Up to 73062
# Transistors (TSMC n3)	~1s Tillion	~10s Trillion
# Cores	1,000s-10,000s	100,000s-1,000,000s
On-Chip Memory	10s-100s MB	~10s GB
Memory Bandwidth	1s TB/s	~10s PBs/s
Attached HBM	~10s-100s GB	10s TB (via TSMC SoW) [21]
Die-to-Die Bandwidth (TB/s)	~1s-10s (via off-chip)	~10s-100s (via on-chip)
Die-to-Die Latency (ns)	~100s (via off-chip)	~1s (via on-chip)
Die-to-Die Power (pJ/bit)	~10s (via off-chip)	~0.1s (via on-chip)

Table 1: System-on-Die vs. System-on-Wafer

GEMV operations (shown by ❷).

LLM inference is memory bandwidth-bound. Model weights (10-1,000 GB) are fetched repeatedly from external memory during inference, as GPUs typically have only 100 MB of on-chip memory. For per request, generating thousands of tokens per second demands hundreds of TB/s bandwidth, far exceeding the capabilities of HBM (high bandwidth memory) on current GPUs.

While tensor parallelism across GPUs can increase bandwidth, mitigating communication bottlenecks in a large GPU cluster remains challenging. Also, adding GPUs improves throughput for concurrent queries but does not reduce time per output token (TPOT), as each query is still memory bandwidth-limited.

2.2 Reasons for wafer-scale accelerators

To increase memory bandwidth, accelerator designers are increasingly adopting system-on-wafer integration [21] for several reasons:

Performance advantages. System-on-wafer technology allows trillions of transistors to be integrated into a single wafer-scale chip, up to 100× more than a typical GPU die, shown in Table 1. This enables millions of AI-optimized cores, providing tens of GBs of on-chip memory and up to tens of PB/s memory bandwidth, 1,000× higher than a standard GPU's several TB/s. Future wafer-scale chips can also attach 40-80× more HBM chips to their edge compared to a standard die [21].

Integration efficiency. System-on-wafer excels at integrating massive parallel cores, with wafer-based die-to-die connections offering up to 10× more bandwidth per unit area, 100-300× latency benefit, and nearly 100× better power efficiency per bit than conventional PCB-based chip-to-chip interconnection [24] (e.g., NVIDIA NVLink, PCIe), shown in Table 1. As noted earlier, LLM inference is primarily constrained by memory bandwidth due to intensive data access. In distributed settings, inter-chip communication overhead from remote memory access, especially during decode-phase GEMV, limits scalability. Thus, system-on-wafer integration offers lower-latency and high-bandwidth interconnects and improved efficiency over conventional chip-to-chip links.

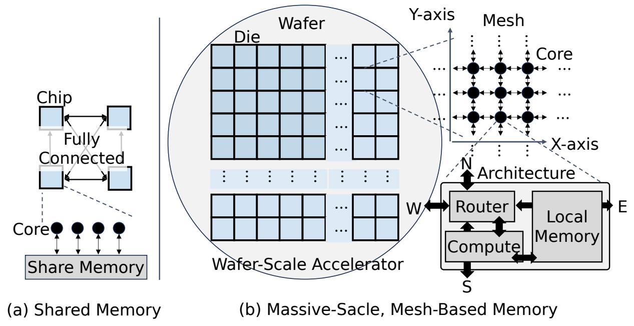

Figure 2: Massive-scale mesh-based memory architecture

Lower cost. Wafer-scale integration can lower the manufacturing cost, since a significant fraction of the cost of fabrication (typically 30-50%) is related to testing and packaging the individual chips [48]. Additionally, wafer-scale integration has made notable progress in yield improvement. Companies such as TSMC are also developing techniques to integrate fully tested dies on a single wafer, further enhancing yield.

2.3 Challenges for wafer-scale LLM inference

The key challenge in leveraging wafer-scale accelerators for LLM inference is their shift to a distributed, non-uniform memory architecture on a single chip. Current LLM systems are optimized for shared memory (single chip) or fully connected architectures (e.g., GPU pods), as shown in Figure 2(a). However, as on-chip memory size grows, these architectures face exponential manufacturing costs and performance degradation, driving the need for a distributed on-chip architecture.

AI accelerator designers predominantly use a **mesh-like network-on-chip (NoC)** to connect **massive cores** (ranging from hundreds of thousands to millions), as shown in Figure 2(b). The mesh topology is favored for its efficiency in core arrangement, enabling effective cooling [29], power delivery [19], and cost-efficient wiring [38, 46], with each core communicating only with nearby neighbors, as shown in Figure 2(b). Alternative topologies, such as 3D torus or tree structures, are impractical due to high on-chip wiring costs. Therefore, wafer-scale chip makers such as Cerebras WSE [24] and Tesla Dojo [41] adopt massive-scale mesh architectures. Even non-wafer-scale accelerators such as Meta MTIA [30], Tenstorrent [18], and others [4, 31] use mesh to scale cores on a chip.

The massive-scale mesh architecture presents challenges for several LLM operations due to their high data movement demands: (i) managing LLM models and KV cache, (ii) GEMM operations during the prefill phase, and (iii) GEMV operations during the decode phase. Other operations, such as element-wise computations such as dot-product and activation functions, require no data movement and naturally benefit from parallelism. Operations needing allreduce, such as RMSNorm and Softmax, can leverage GEMV solutions.

3 Device Model for Wafer-Scale Accelerators

3.1 The PLMR model

We develop the PLMR model to capture the unique hardware properties of wafer-scale accelerators and to motivate system requirements needed for utilizing this emerging hardware.

(1) **Massive Parallelism (P)**: A wafer-scale accelerator can accommodate millions of parallel cores, compared to thousands in GPUs. Each core features a local hardware pipeline that overlaps data ingress, egress, computation, and memory access at the cycle level. This requires the computation to be partitioned at a massive scale and a fine-grained schedule to overlap computation, memory access, and NoC communication.

(2) **Highly non-uniform memory access Latency (L)**: Accessing memory across cores in a mesh incurs highly non-uniform latency. In a mesh with $N_w \times N_h$ cores, organized as a rectangle, the maximum number of NoC hops between two cores is $N_w + N_h$, and the worst-case memory access latency is given by $\alpha(N_w + N_h) + \beta r$, where $r < N_w + N_h$ denotes the number of routing stages along the communication path. Here, α denotes the per-hop transmission latency, the cost incurred when a message is directly forwarded at a core according to the router hardware's pre-configured rules, increasing with hops. β represents the per-routing latency, the overhead when a message is involved in header parsing and rewriting by software at a core when forwarding [27]. Typically, $\alpha < \beta$. In a mesh with a million cores, the maximum number of hops and routing can reach up to several thousand, resulting in up to a thousand times latency gap between local and remote memory access. Consequently, minimizing long-range communication is critical for performance.

(3) **Constrained per-core local Memory (M)**: Each core has a small local memory (tens of KBs to several MBs), as performance and energy efficiency decline with larger capacities [47]. As a result, computation data must be explicitly partitioned into fine-grained chunks to fully fit within the constraints of each core's local memory.

(4) **Constrained Routing resources (R)**: Wafer-scale accelerators integrating millions of cores impose strict constraints on each core's routing circuit complexity or routing table size. For example, on the Cerebras WSE-2, each core can recognize only message headers with a 5-bit address code. Consequently, each core can support at most 2^5 distinct routing paths, and the software system must carefully plan these paths. For long-distance remote communication, a pair of cores can consume routing resources to establish a direct routing path, incurring only α latency. However, if the number of routing paths exceeds hardware limits, messages must be relayed through multiple intermediate cores, introducing additional β latency.

We expect these properties to remain relevant, as they are rooted in the fundamental characteristics of hardware and its manufacturing process. The PLMR model applies to both current (Cerebras WSE) and future (Tesla Dojo) wafer-scale devices. Even some non-wafer-scale devices with mesh-based NoC architectures, such as Tenstorrent Blackhole [18], can be represented by PLMR with adjusted parameters for parallelism (P), the size of the mesh (L), or relaxed constraints on local memory (M) and routing resources (R).

3.2 Limitations of state-of-the-art approaches

Leveraging the PLMR model, we analyze why existing AI systems fail to fully utilize wafer-scale accelerators. To run an LLM on a wafer-scale accelerator, we generally have two choices: (i) abstract the distributed local memory in each core as a shared memory and directly access data placed in a remote core through NoC; and (ii) explicitly partition computation into distributed cores and use message passing to exchange necessary data. We analyze two types of representative systems: LLM runtime or DNN compilers for shared memory architecture such as GPUs, e.g., Ladder [45]; and the SOTA compiler for distributed on-chip memory architectures, e.g., T10 [25] for GraphCore IPU.

Shared-memory system. A shared-memory-based DNN compiler such as Ladder usually assumes a uniform memory access pattern within the underlying memory hierarchy, which cannot tolerate the thousands of times latency variance in wafer-scale accelerators when accessing data from remote memory (failing in L). Moreover, these compilers [10, 28, 37, 45, 51, 54, 57] often focus primarily on partitioning computation, with less emphasis on optimizing data partitioning. This approach can easily lead to significant data duplication and violate the memory constraint requirements (failing in M). Finally, these compilers are unaware of the communication distance of each core, poorly addressing the constraint of routing resources.

Distributed-memory system. The T10 system [25] is designed for AI accelerators with an on-chip crossbar, which ensures a constant latency of memory access to any other cores on the same chip. T10 handles small local memory and balances communication loads, addressing memory constraints (M) and routing resource limitations (R). However, on a PLMR device, it fails to account for varying memory access latency (failing in L) and scales to thousands, not millions, of cores (failing in P).

4 Wafer-Scale LLM Parallelism

We present wafer-scale LLM parallelism, featuring new designs across prefill, decode, and KV cache management.

4.1 Prefill parallelism

The parallelism for LLM prefill must ensure compliance with the PLMR model. Key challenges include: (i) Handling multiple large matrices during prefill, requiring effective dimension

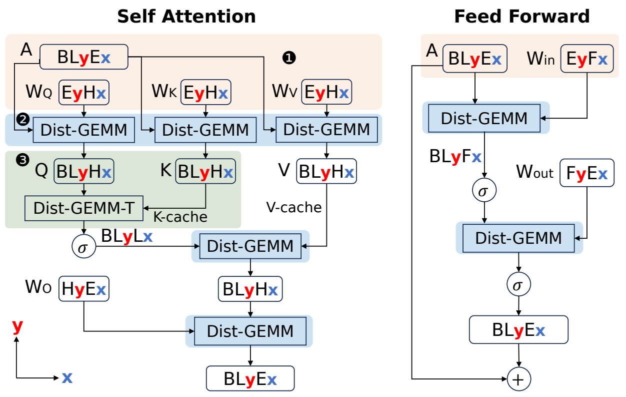
Figure 3: Prefill parallelism plan. E_xF_y represents a matrix of shape EF, where the E dimension is partitioned along the x-axis of cores, and F along the y-axis of cores on a mesh.

partitioning to achieve million-core parallelism (P); (ii) Optimizing GEMM operations, which involve further partitioning and overlapping computation and communication, to minimize long-range communication latency (L), respect local memory constraints (M), and account for limited routing resources (R); and (iii) Handling matrix transposes, which are costly on a mesh NoC (L) but often required for sequential GEMM operations.

Designing fine-grained partitioning for million-core parallelism. To achieve high chip utilization, we propose partitioning two dimensions of the input activation and weight matrices along both the X- and Y-axes of cores. This approach enables finer-grained, million-scale parallelism compared to existing methods [12,15,32,35], which typically partition only the embedding dimension, resulting in insufficient parallelism on PLMR devices.

We illustrate this partitioning using self-attention and feed-forward, as shown in Figure 3. For this discussion, we define the following annotations: the input activation A and weight W are multi-dimensional tensors during the prefill process. B represents the batch size, L the sequence dimension [2], E the embedding dimension, H the head dimension, and F the hidden dimension in the feedforward block. As shown by ❶, the partitioning layout of A is represented as BL_yE_x, where the L dimension is partitioned along the Y-axis of cores, and the E dimension along the X-axis of cores. Similarly, all weight matrices (W_Q, W_K, W_V, W_{in}, and W_{out}) are partitioned across both dimensions.

Designing PLMR-compliant distributed GEMM. We propose replacing conventional GEMM operators, which are designed for shared memory architectures, with a newly designed PLMR-compliant distributed GEMM during the prefill phase (as shown in ❷ of Figure 3). Unlike TPU and GPU systems that primarily rely on allgather operations for GEMM, PLMR-compliant distributed GEMM algorithms achieve high NoC bandwidth utilization while respecting local memory and routing constraints, ensuring compliance with the L, M, and R properties. This PLMR-compliant distributed GEMM is fully described in Section 5.

Using transposed distributed GEMM to avoid matrix transpose. We propose a transpose-free parallelism plan for prefill to avoid matrix transpose, a common operation in LLM systems designed for shared memory architectures. The L property in PLMR highlights that matrix transposition is particularly costly on a wafer-scale device. It requires a core on one corner of the mesh to send data to the opposite diagonal corner, creating a long-range communication path.

Our transpose-free parallelism plan leverages transposed distributed GEMM (denoted as dist-GEMM-T) [11,42] to compute $Q@K^T$ during LLM prefill, as shown by ❸ in Figure 3. Specifically, the intermediate Q and K tensors, generated by multiplying X with W_Q and W_K, require transposing K before proceeding with dist-GEMM operations due to the on-chip partition shape.

[2] L: sequence dimension; L: memory access latency in PLMR.

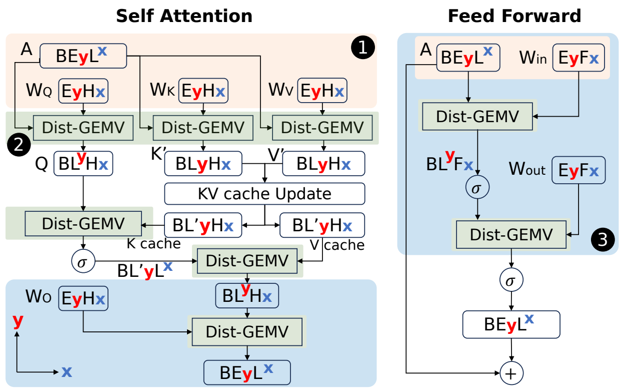
Figure 4: Decode parallelism plan. E^yF_x indicates the E dimension is replicated along the y-axis, and F is partitioned along the x-axis.

4.2 Decode parallelism

The parallelism strategy for LLM decode must address its memory-bandwidth-intensive nature, presenting several challenges: (i) Decode uses smaller matrices than prefill due to limited input sequences and batch sizes, requiring careful parallelization when dimensions are insufficient for partitioning; (ii) The phase heavily relies on GEMV operations, which are less compute-intensive than GEMM, resulting in short computation phases with limited overlap with communication, making GEMV vulnerable to long-range communication latency on a mesh NoC (L) and requiring adherence to local memory and routing constraints (M and R); and (iii) Sequential GEMV operations introduce costly matrix transpose on a NoC, risking violation of the L property.

Designing fine-grained replication to enable parallelism at minimal communication cost. When tensor dimensions are insufficient to achieve the high parallelism required for decode, we propose fine-grained replication of tensors in LLMs,

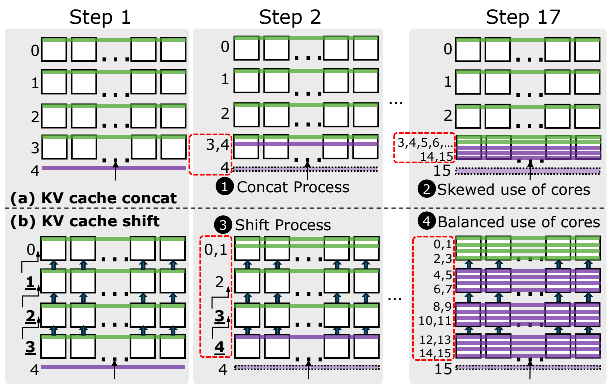

Figure 5: KV cache concatenation vs. KV cache shift

specifically replicating the sequence dimension, where the sequence length equals the prompt length during prefill phase and equals 1 during the decode phase. This approach offers two key advantages: (i) it improves parallelism and ensures balanced loads across all cores, and (ii) it avoids additional communication operations such as allreduce across all cores. As shown by ❶ in Figure 4, the E dimension is partitioned along the y-axis, and the L dimension is replicated along the x-axis, represented as BE_yL^x. Weight matrices W are partitioned across both dimensions, consistent with the prefill phase.

Our fine-grained replication differs from recent work on long-context/sequence inference systems [49, 55], which selectively replicate certain dimensions during the prefill phase rather than the decode phase.

Designing PLMR-compliant distributed GEMV. We found that existing GEMV implementations fail to fully comply with PLMR requirements due to long-range communication and excessive routing resource consumption at each core. To address this, we propose a PLMR-compliant distributed GEMV, utilizing this new implementation throughout the decode phase (as detailed in ❷ of Figure 4). A comprehensive description of this GEMV design is provided in Section 6.

Pre-optimizing model weight placement to avoid matrix transpose. To avoid matrix transpose during decode, we pre-optimize the model weight layout for decode, particularly for the distributed GEMV operation, to eliminate matrix transpose. While this introduces re-placement overhead between prefill and decode phases, the overhead is far smaller than that of sequential matrix transpose during token generation.

Figure 4 illustrates this proposal, detailed in ❸. Specifically, we optimize the placement of weights such as W_O and W_{out} for distributed GEMV in decode, differing from their layout in the prefill phase. This approach also removes transpose operations in calculating $Q@K^T$ during decode self-attention.

4.3 Shift-based KV cache management

KV cache management on PLMR devices is challenging as it requires storing large data across distributed cores while adhering to local memory constraints (M) and distributing KV cache computations to achieve high parallelism (P). To address these, we have the following insights:

Existing concatenate-based management causes skewed core utilization. Current KV cache management methods primarily concatenate the most recently generated KV vectors to the existing cache. Though efficient in shared memory architectures, this concatenate operation leads to highly skewed core utilization on PLMR devices, as shown in ❶ of Figure 5, where only the core in a row is responsible for storing and computing over the generated KV vector. After several token generation steps, this only core quickly becomes the bottleneck, as depicted in ❷ of Figure 5, causing skewed memory usage and violating the M in PLMR. Moreover, the imbalanced KV cache distribution across cores results in inefficient parallelism, violating the P property.

Proposing shift-based management for balanced core utilization. We propose a shift-based KV cache management strategy that evenly distributes cache data across all cores. Instead of concatenating new KV cache vectors at the end, this method performs a balancing shift operation, where each row transfers the oldest KV cache data to the row above, as shown in ❸ of Figure 5. When new KV data arrives, each core checks its local capacity against its neighbors. If equal, upward shifts are triggered, with each row receiving data from below and passing some to the row above. As illustrated in ❹, this ensures even KV cache distribution across all cores.

The upward shifts utilize NoC links in parallel, maintaining high performance and satisfying the P property. The physical placement of KV cache aligns with logical continuity and only involves data movement between the adjacent cores, adhering to the L property. This method also fully resolves the M violation issue observed in the last row of cores with the concatenate-based approach.

4.4 Implementation details

We outline several implementation details below:

Prefill and decode transition. Prefill and decode require distinct strategies. To handle the transition efficiently, we reshuffle KV cache and weights through the fast NoC, which often provides 100s Pbits/s aggregated bandwidth, completing instantly without relying on slower off-chip memory.

Parallelism configuration. WaferLLM uses offline autotuning to select core counts for each model, optimizing latency based on model size, input/output length, memory per core, and prefill/decode phases. It chooses different core counts for each phase, with fast dynamic remapping enabled by high NoC bandwidth. For models with variable input/output lengths, average values are used to maintain near-peak performance. Autotuning runs separately per model to adapt to specific needs.

Variations of self-attention. WaferLLM supports variations of Self-Attention, including Grouped Query Attention [3], Multihead Attention [43], and Multi-query Attention [5].

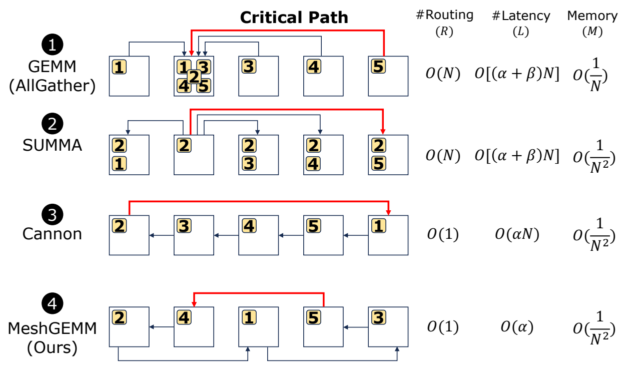
Critical Path
#Routing (R)
#Latency (L)
Memory (M)
❶ GEMM (AllGather)
$O(N)$ $O[(\alpha+\beta)N]$ $O(\frac{1}{N})$
❷ SUMMA
$O(N)$ $O[(\alpha+\beta)N]$ $O(\frac{1}{N^2})$
❸ Cannon
$O(1)$ $O(\alpha N)$ $O(\frac{1}{N^2})$
❹ MeshGEMM (Ours)
$O(1)$ $O(\alpha)$ $O(\frac{1}{N^2})$

Figure 6: PLMR compliance in distributed GEMM

These differ by performing dist-GEMM, dist-GEMV and dist-GEMM-T locally after grouping by head dimensions.

5 Wafer-Scale GEMM

In this section, we introduce MeshGEMM, a scalable distributed GEMM for massive-scale, mesh architectures.

5.1 PLMR compliance in distributed GEMM

To identify a scalable distributed GEMM for PLMR devices, we define the following metrics: (i) *Routing paths per core*: The number of routing paths per core, with fewer paths ensuring compliance with the R property. (ii) *Latency of critical path*: Maximal latency among all communication paths in each step to transmit submatrix (as the red lines in Figure 6), with less latency adhering to the L property. (iii) *Memory per core*: The memory required per core, with lower usage ensuring the M property.

We analyze current distributed GEMM methods and show how MeshGEMM meets these metrics:

(1) **GEMM via Allgather** is commonly used in GPU and TPU pods for distributed GEMM [32, 35, 56]. Its critical communication path in each step is one core gathering data from the farthest cores, shown as the red line in Figure 6 ❶, and N steps to complete the allgather. Each core creates N routing paths to neighbors in its row and column (violating R). The constraint of R necessitates step-by-step submatrix transmission via intermediate cores, introducing a communication latency of $O[(\alpha+\beta)N]$ along the critical path (violating L), and each core uses $O(1/N)$ memory due to inflated working buffers (violating M).

(2) **SUMMA** is Cerebras' default choice for distributed GEMM on its wafer-scale engine [7]. Its critical communication path in each step is where one core broadcasts data to the farthest core along the column or row, shown by the red line in ❷ of Figure 6. Same to the allgather communication in ❶, each core creates N routing paths (violating R) and spans the critical path with $O[(\alpha+\beta)N]$ latency (violating L) in each step. While SUMMA improves memory usage compared to allgather, requiring only a working set equal to the size of locally partitioned submatrices, it still doubles the peak memory usage.

(3) **Cannon** is mesh-optimized choice for distributed GEMM [6], popular in supercomputers and distributed cluster. Its critical communication path in each step is the head cores send data to the tail cores shown in ❸ of Figure 6. Each intermediate core communicates with two neighbours in a 2D torus and passes through the submatrix from head to tail, which only needs $O(1)$ communication paths and optimal memory usage of $O(1/N^2)$.
Notably, Cannon involves only a constant number of routing paths per core, allowing static routing rules to be assigned for both neighbor communication and the critical path. At each step, submatrices can pass directly from the head to the tail core through intermediate cores, unlike GEMM via allgather or SUMMA, which require step-by-step message transmission. As a result, the critical path incurs only per-hop latency α, without additional per-routing overhead β. However, since the critical path spans N hops, it still incurs $O(\alpha N)$ latency, shown as the red line in ❸, violating the L.

(4) **MeshGEMM (Ours)** is a distributed GEMM which complies with the PLMR model. The critical communication path in each step, which we named as two-hop transmission, is shown as the red line in ❹ of Figure 6, which is significantly shorter than the others. Each core communicates with two two-hop away neighbors(proven in later sections to be scalable for larger-sized mesh architectures), and passes through a one-hop neighbor's communication. This design achieves $O(1)$ communication paths per core needed and optimal memory usage of $O(1/N^2)$, similar to Cannon. Crucially, it bounds the critical path to constantly 2 hops with $O(\alpha)$ complexity, making it uniquely capable of addressing the L property.

5.2 Design intuitions and scalability analysis

Our design involves two phases: (i) We ensure algorithm correctness using a cyclic shifting process for GEMM, and (ii) then use interleaving to bound the critical path latency to a constant. Based on the cyclic shifting and interleaving, we get the **two-hop transmission** communication path, and we prove that it, on this cycle, is the minimal distance required to satisfy the L property.

Cyclic shifting. Cyclic shifting enables MeshGEMM to satisfy the M and R properties by limiting communication to two neighbors and minimizing memory usage. It ensures correct GEMM results, following reasoning similar to Cannon [6]. As illustrated in ❸ of Figure 6, a logical circle of 5 cores is flattened into the physical communication mapping, with a critical path from head core to tail core.

Interleaving. For the flattened communication plan, we would like to minimize the length of the critical path further, thus satisfying the L property. Our key intuition here is

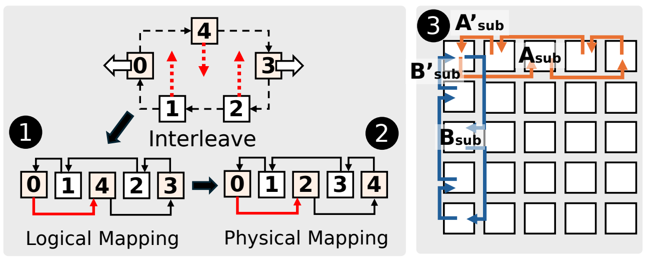

Figure 7: Design intuitions and scalability analysis.

Algorithm 1: INTERLEAVE

```
Input: index, N
Output: send_index, recv_index
if index mod 2 == 0 then
    recv_index = Max (index - 2, 0);
    send_index = Min (index + 2, N - 1);
else
    recv_index = Min (index + 2, N - 1);
    send_index = Max (index - 2, 0);
if index == 0 then recv_index = 1;
if index == N - 1 then
    if N mod 2 == 0 then recv_index = N - 2;
    else send_index = N - 2;
Return send_index, recv_index;
```

to introduce an INTERLEAVE operation to find the mapping relationship from logical to physical, defined in Algorithm 1. As shown by ❶ of Figure 7, MeshGEMM first insert core 1 in between core 0 and 4 and core 2 in between core 4 and 3 to form a logical mapping, and then call the INTERLEAVE operation to get the send to and receive from neighbours' index, resulting in a permutated, equivalent communication plan as shown by ❷ in Figure 7. For example, there are 5 cores total (N=5), so physical core 2 (index=2) sends data to physical core 4 (send_index=4) and receives from physical core 0 (recv_index=0). Figures 6 and 7 illustrate the case with 5 cores as an example, while Algorithm 1 demonstrates that the two-hop communication pattern in MeshGEMM generalizes to mesh architectures of arbitrary size with $N \geq 3$.

Scalability analysis. We can prove that the two-hop distance created by INTERLEAVE cannot be further reduced. The proof relies on the fundamental properties of sequential arrangements: if we attempt to create a circular sequence where each number differs from its neighbors by exactly one hop, we encounter a mathematical impossibility. This can be understood by visualizing the numbers as points on a line - while adjacent numbers can be connected, the endpoints of the sequence cannot simultaneously maintain single-hop differences with their neighbors while forming a circle.

Note that our discussion, based on a 1D array, naturally extends to a 2D mesh, as the 1D array corresponds to the mesh's X-axis and Y-axis due to their symmetry.

5.3 The MeshGEMM algorithm

We outline the key steps of MeshGEMM below:

(1) **Initialization:** Consider $C = A \times B$. MeshGEMM will partition A and B into tiles A_{sub} and B_{sub} along two dimensions, forming $N \times N$ tiles, which are distributed across the cores. Each core receives one tile of A_{sub} and one of B_{sub}. MeshGEMM will then use INTERLEAVE to initialize the neighbors' positions for each core.

(2) **Alignment:** Each core will then align with neighbors to align the input submatrices in a way that ensures every core in the distributed system begins with the appropriate operands for the matrix multiplication process.

(3) **Compute-shift loop:** Each core operates with a compute-shift loop involving N steps of communication and computation. In each step, every core computes the partial sum of its corresponding $C_{sub} = A_{sub} \times B_{sub} + C_{sub}$. Meanwhile, shift A_{sub} along the X-axis and B_{sub} along the Y-axis to get new A'_{sub} and B'_{sub} for the next step computation as ❸ we shown in Figure 7. After N steps, the accumulated C_{sub} is returned.

5.4 Implementation details

Handling non-square mesh. For a non-square mesh $N_h \times N_w$ ($N_h \neq N_w$), the A and B matrices can be logically partitioned into $N_{lcm} \times N_{lcm}$ cores, where N_{lcm} is the least common multiple of N_h and N_w.

Transposed distributed GEMM. The above algorithm can be applied to the computation of $C = A \times B^T$, the dist-GEMM-T in Figure 3 to avoid transposing B on mesh. It does not require alignment before computation and only necessitates N steps of two-hop compute-shift for the right matrix B along the Y-axis. After each shift step, each core computes $C_{\text{sub}} = A_{\text{sub}} \times B_{\text{sub}}$, followed by a ReduceAdd of C_{sub} along the X-axis. After N steps, the final matrix C is obtained.

6 Wafer-Scale GEMV

In this section, we describe MeshGEMV, a scalable GEMV algorithm for PLMR devices.

6.1 PLMR compliance in distributed GEMV

The completion time of a distributed GEMV is primarily determined by an element-wise computation to generate a partial sum result at each core and then an allreduce operation that aggregates partial results from all selected cores and then sends the aggregated results back to all cores for the continuous GEMV. As the analysis in GEMM, in wafer-scale GEMV, we also define the following metrics: (i) *Routing paths per core*: The number of routing paths per core, with fewer paths ensuring compliance with the R property (ii) *Latency of critical*

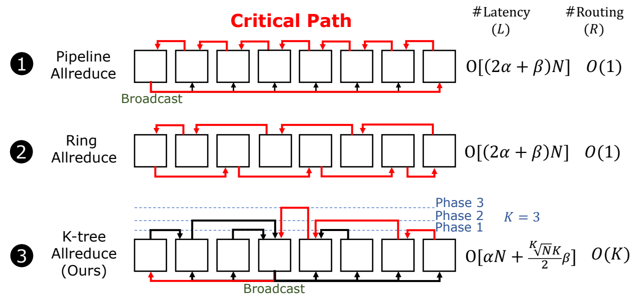

Figure 8: PLMR compliance in distributed GEMV

path: Maximal latency of the whole allreduce communication, mainly determined by the time from when the partial sum is sent from the farthest core to when the aggregated sum is received by the farthest core.

We analyze current distributed GEMV methods and show how MeshGEMV meets these metrics:

(1) **GEMV with pipeline allreduce** is commonly used in TPU pod systems [35] and as the default in Cerebras demo [9]. As illustrated by the red line in the Figure 8 ❶, the critical path of the pipelined all-reduce starts from the farthest core, where partial sums are reduced step-by-step toward the root core, and the aggregated result is then broadcast from root to all cores. While this approach ensures that the routing path per core remains within the R constraint, $O(1)$ per-core routing resource usage, it incurs a total of $2N$ hops and N routing stages, violating the L property.
(2) **GEMV with ring allreduce** is commonly used in GPU pod systems, where it is the default configuration, especially for a large amount of data [27]. As shown by ❷ in Figure 8, the critical path of ring allreduce involves each partial sum traversing all cores in the ring. Similar to pipelined allreduce, this approach maintains $O(1)$ routing paths per core, but incurs a communication latency of $O[(2\alpha+\beta)N]$, thus violating the latency constraint L defined in PLMR.
(3) **GEMV with K-tree allreduce (Ours)**. As analyzed above for pipelined and ring allreduce, the sequential execution of reduce-add operations leads to N routing stages along the critical path. In contrast, K-tree allreduce organizes the reduce-add path as a balanced K-tree, enabling K phases grouped parallel reductions with $O(\sqrt[K]{N})$ cores per group and reducing the critical path to only $\frac{\sqrt[K]{NK}}{2}$ times routing and N hops. However, this comes at the cost of requiring $O(K)$ routing paths per core.

6.2 The MeshGEMV algorithm

We will outline the key steps of MeshGEMM below:

(1) **Initialization:** Consider $C = A \times B$ and A is a vector. MeshGEMV will partition B into tiles B_{sub} along two dimensions, forming $N \times N$ tiles and distributed across the cores. For A, MeshGEMV will partition it along the vector length, forming N tiles distributed on one axis and replica A on another axis. Each core receives one tile of A_{sub} and one of B_{sub}. Then we determine which cores form a group to obtain aggregated results in each phase based on the K-tree.
(2) **Parallel computation:** In this stage, each core performs a local GEMV $A_{\text{sub}} \times B_{\text{sub}}$ to obtain C_{sub} partial sum.
(3) **Aggregation:** The aggregation step primarily involves using the K-tree allreduce we design. The key steps are as follows: (i) In the 1st-phase, each group performs group reduction and obtains the partial sum of C_{sub} at the root core of each group. (ii) In the kth-phase, the results from the $(k-1)$ th-phase are reduced to the root cores of each group in the kth-phase. After K times repeating, C can be obtained by concatenating the C_{sub} from all K-tree root cores. (iii) Optionally, a broadcast operation from the root core of the K-tree may follow, depending on whether continuous GEMV is required.

Scalability Analysis. As shown in ❸ of Figure 8, this method scales efficiently with parallelism and meets the L property by selecting an appropriate K. It requires $K+1$ paths at the tree root core but allows flexible adjustment of K to address R based on hardware limitations. Compared to pipeline and ring all-reduce, K-tree all-reduce essentially builds pass-through paths between distant cores by consuming routing resources, thereby reducing routing latency.

However, a larger K is not always better, as it depends on N and R constraints. Additionally, larger K increases routing complexity and overhead. Considering these factors, we have chosen $K = 2$ for our current implementation, evaluated in the following sections.

7 Evaluation

We extensively evaluated WaferLLM against various state-of-the-art methods and systems. Our results show that:

(1) WaferLLM achieves orders of magnitude speedup over T10 and Ladder in LLM inference (§7.1);
(2) WaferLLM's MeshGEMM and MeshGEMV perform and scale strongly over the state-of-the-art (§7.2);
(3) WaferLLM's shift-based KV cache management enables over 360-385× more token capacity (§7.4);
(4) WaferLLM on Cerebras WSE-2 achieves 10-20× e2e speedup compared to the optimal performance of SGLang on A100 GPUs-cluster, and about 30-40× speedup compared to a single A100 GPU. WaferLLM also achieves approximately 2.5× better energy efficiency (§7.1,§7.5).

Experiment setup. We evaluate WaferLLM on a server with Cerebras WSE-2. WSE-2 has 850,000 Cores, each with a Compute Engine (CE) operating at a maximum of 1.1 GHz. Each clock cycle can fetch two 32-bit operands from SRAM, perform a multiply-accumulate operation, and then write back to SRAM. Each core also has a fabric router that can send or receive 32-bit messages from neighbouring cores with a single clock cycle. Additionally, each core contains 48KB of SRAM, with the chip totalling 40GB of aggregated SRAM [24].

We compare WaferLLM with two DNN compilers: (i)T10 [25], the state-of-the-art compiler for AI accelerators with inter-core connections and distributed on-chip memory, and (ii)Ladder [45], the state-of-the-art compiler for shared memory architectures. For T10, we implemented it on WSE-2, treating each core as part of a distributed memory system interconnected by a crossbar, despite the actual mesh topology. T10 maps data to core IDs and fetches data from local SRAM as required. For Ladder, we treated the distributed memory architecture of the chip, interconnected by mesh, as unified memory, requiring collective communication over the NoC to access data.

We use the Nvidia A100 for GPU comparison experiments, which shares the same 7nm process as the WSE-2 for fairness. The experiments utilize up to 16 GPUs across two nodes (2×8 configuration). Within a single node, the eight A100 GPUs are interconnected via NVLink, while inter-node communication is handled through a high-performance InfiniBand (IB) network. We use SGLang [53], one of the highest-performing LLM inference systems for GPU-based experiments.

Experiment metric. To evaluate the maximum per-request throughput performance of LLM inference systems on different hardware, we define Throughput per Request (TPR) as a key metric. TPR is derived from the more widely used Time per Output Token (TPOT), with TPR $= \frac{1}{\text{TPOT}}$.

LLM models. Our evaluation includes various representative LLMs of different sizes and architectures. Specifically, LLaMA3-8B and LLaMA2-13B are widely used open-source LLMs, with LLaMA3 using group-query attention instead of multi-head attention to reduce KV cache usage. For LLaMA2-13B, we modified the model to remove the 4K context length limitation to evaluate system performance across different input and output lengths. Additionally, due to the tensor parallelism constraints, the number of attention heads of a model must be divisible by the number of GPUs used. We did not conduct the multi-GPU experiment of LLaMA-2 on a 16-GPU cluster. CodeLLaMA-34B is a specialized LLM for coding tasks, while QWen2-72B, another popular LLM, is renowned for its high model quality.

7.1 LLM inference

We first report the end-to-end performance of WaferLLM compared to T10 and Ladder on WSE-2 and SGLang on the A100 GPU cluster. Then we further analyze the performance to provide deeper insights by breaking down the execution into prefill and decode phases.

End-to-end throughput. Table 2 shows the inference TPR of LLaMA3-8B and LLaMA2-13B on WSE-2 and A100 with different input and output sequence lengths. Here, the end-to-end throughput is calculated as the total number of tokens generated during the decode phase divided by the total time spent in the prefill and decode phases. WaferLLM uses core configurations optimized for the best performance with each model. In LLaMA3-8B, we use 660×660 cores for prefill and 360×360 for decode. In LLaMA2-13B, we use 750×750 cores for prefill and 375×375 for decode. CodeLLaMA-34B and QWen2-72B are not included due to the memory constraint of a single WSE-2 chip.

Model	Device		Input/Output Seqence Length			
			2048/128	4096/128	2048/2048	4096/4096
LLaMA3-8B	WSE-2	WaferLLM	764.4	604.4	2370.3	2459.0
		T10	4.6	4.5	58.3	94.6
		Ladder	1.2	1.1	7.4	8.7
	A100 GPU (SGLang)	1	34.8	31.1	36.5	78.4
		8	117.2	109.0	128.4	256.1
		2×8	73.7	70.2	79.3	162.5
LLaMA2-13B	WSE-2	WaferLLM	473.9	414	1690.3	1826.0
		T10	2.6	2.5	35.0	58.3
		Ladder	0.7	0.7	4.9	6.1
	A100 GPU (SGLang)[3]	1	20.4	17.1	21.1	47.9
		8	79.6	70.5	86.9	172.4

Table 2: End-to-end LLM inference TPR

Compared to T10, WaferLLM achieves 160× speedup on average, up to 180×, for short sequence generation tasks such as 4096 and 2048 input context lengths with 128 tokens output. For longer tasks, with input context lengths of 4096 and 2048 tokens and output lengths of 4096 and 2048 tokens, WaferLLM achieves 36× on average and up to 48×. Although T10 designs the compute-shift model that considers the memory constraints (M) and routing resource limits (R) of a PLMR device, it does not account for the cores interconnected by a mesh NoC. Thus, failing to address varying hop distances (L) and scale to millions of cores (P), highlighting the need for new system designs in massive-scale NUMA architectures.

Compared to Ladder, WaferLLM achieves 625× speedup on average, up 677×, for short sequence generation tasks such as 4096 and 2048 input context lengths with 128 tokens output. For longer sequence generation tasks, with input context lengths of 4096 and 2048 tokens and output lengths of 4096 and 2048 tokens, WaferLLM achieves 312× on average and up to 342×. That is because Ladder is designed for shared memory architecture and does not consider the characteristics of the PLMR device, resulting in failure in partitioning LLMs across millions of cores (P), incurring costly long-range NoC communication (L), failure in handling local memory constraints (M) and limited routing resources (R).

Prefill throughput. Table 3 shows the prefill performance for an input sequence length of 4096, using core configurations from 480×480 to 720×720. For CodeLLaMA-34B and QWen2-72B, which exceed the memory capacity of WSE-2 and a single A100, we evaluate a subset of layers and scale the results proportionally due to their uniform layer structure.

WaferLLM achieves significant speedups over T10 and

[3]No 2×8 GPUs, due to model architecture and tensor parallelism.

Model	Methods	WSE-2 Cores			# A100 GPUs (SGLang)		
		480×480	600×600	720×720	1	8	2×8
	WaferLLM	20320.6	25037.2	27686.5			
LLaMA3-8B	T10	175.0	156.6	132.8	13988.3	17361.6	13994.2
	Ladder	61.8	42.3	31.3			
	WaferLLM	13685.1	16854.2	17498.3			
LLaMA2-13B	T10	121.3	100.6	81.3	7805.1	12287.1	/[3]
	Ladder	47.3	33.1	24.2			
	WaferLLM	5471.4	7540.1	8526			
CodeLLaMA-34B	T10	49.1	46.8	41.2	5382.5	7155.5	6409.2
	Ladder	30.1	23.1	17.7			
	WaferLLM	2785.2	3775.5	4421.6			
QWen2-72B	T10	24.9	23.5	21.5	1677.3	3803.8	3750.5
	Ladder	16.8	12.8	10.1			

Table 3: Prefill Throughput per Request (TPR)

Model	Methods	WSE-2 Cores			# A100 GPUs (SGLang)		
		420×420	540×540	660×660	1	8	2×8
	WaferLLM	2699.9	2501.5	2243.3			
LLaMA3-8B	T10	418.3	339.4	265.1	78.9	260.4	164.6
	Ladder	14.6	13.1	11.4			
	WaferLLM	2039.2	1899.4	1739.8			
LLaMA2-13B	T10	341.8	270.8	233.7	48.7	175.8	/[3]
	Ladder	11.0	9.9	9.0			
	WaferLLM	1450.8	1407.7	1359.2			
CodeLLaMA-34B	T10	278.2	222.4	193.1	26.1	100.4	84.5
	Ladder	6.1	6.2	5.8			
	WaferLLM	839.7	824.3	787.1			
QWen2-72B	T10	168.5	133.0	114.6	10.6	51.2	48.7
	Ladder	3.2	3.3	3.4			

Table 4: Decode Throughput per Request (TPR)

Ladder by effectively addressing all PLMR properties, an average speedup of 160× (up to 178×) over T10 and 270-450× over Ladder. As discussed in Section §2, GEMM is the primary bottleneck, and MeshGEMM substantially enhances WaferLLM's prefill performance, analyzed in detail in Section §7.2.

Additionally, WaferLLM scales throughput with increasing cores across all models. For instance, WaferLLM achieves a 1.6× scaleup on QWen2-72B and a 1.4× scaleup on LLaMA3-8B when scaling from 480×480 to 720×720 cores. In contrast, T10 and Ladder fail to scale effectively, with throughput even declining as more cores are added. This is mainly because T10 and Ladder do not account for the spatial locality characteristic (L) of highly uniformly distributed memory, which can lead to frequent long-distance data reads across cores and increased communication overhead.

Decode throughput. Table 4 shows decode throughput for core configurations from 420×420 to 660×660. For CodeLLaMA-34B and QWen2-72B decode benchmark on WSE-2 and single A100, we evaluate a subset of layers and scale the results.

By addressing all PLMR properties, WaferLLM achieves an average speedup of 5.7× (up to 6.5×) over T10 and 217× (up to 260×) over Ladder.

Compared to the 160× speedup WaferLLM achieves over T10 during prefill, it offers only about a 6.5× speedup during decode. However, WaferLLM maintains a consistent 200-500× advantage over Ladder in both prefill and decode stages. This is mainly because prefill involves dist-GEMM computations, which require each data element to traverse cores along the same row or column sequentially. In contrast, the dist-GEMV allreduce operation in decode only requires traversal across cores in the same row or column, without enforcing a

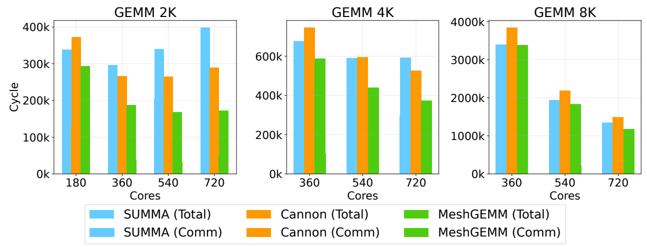

Figure 9: MeshGEMM vs. SUMMA & Cannon

strict access order. While T10's compute-shift paradigm does not consider the spatial locality of cores on a mesh NoC, it still accounts for distributed memory, which allows performance gains when memory (communication) accesses are order-independent. In contrast, Ladder is entirely designed for a shared memory architecture, leading to severe communication overhead regardless of whether memory access order is required.

Comparison with SGLang. As shown in Table 2,3 and 4, WaferLLM demonstrates clear advantages over SGLang + A100 GPU clusters in prefill, decode tasks, and also the end-to-end performance across models ranging from 8B to 72B. More detailed comparisons with GPUs are discussed in Section §7.5.

It is worth noting that T10 and Ladder fail to leverage the powerful hardware capabilities of the WSE-2 when compared to SGLang. For example, in prefill tasks and for Ladder in decode tasks, their performance is even worse than that of SGLang running on a single A100. This highlights that designing high-performance systems and algorithms on wafer-scale chips must carefully consider the PLMR model; otherwise, not only will the massive compute potential of the chip go underutilized, but performance may even degrade due to the highly uniform distributed memory resulting from the large-scale mesh NoC.

7.2 MeshGEMM

We compare MeshGEMM with Cannon [6] and SUMMA [42] on WSE-2 across different core scales and matrix sizes.

Scaling core count. Figure 9 shows that across matrices of different sizes, MeshGEMM can achieve the lowest latency than SUMMA and Cannon by scaling up the number of cores. Compared to the pipeline broadcast in SUMMA and head-to-tail transmission in Cannon, interleave transmission in MeshGEMM can minimize per-step communication overhead and, as much as possible, overlap communication with computation during large-scale fine-grained parallel execution. It demonstrates stronger scalability, maintaining over 70% computational efficiency even near the hardware limit. In contrast, SUMMA and Cannon exhibit poor scalability, with computational efficiency falling below 50% with 720×720 cores, primarily due to the communication overhead in SUMMA and Cannon, increasing with the parallelism scale.

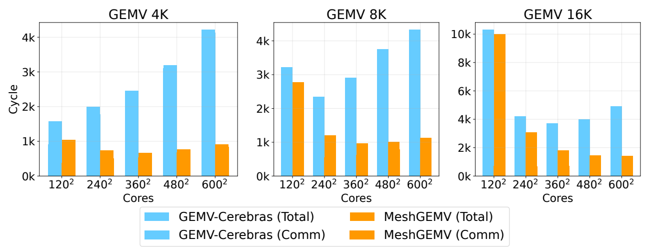

Figure 10: MeshGEMV vs. GEMV-Cerebras

Additionally, increasing the number of cores does not always bring performance gains, especially for small matrix GEMM tasks. For example, in GEMM 2K, when scaling from 360×360 to 720×720 cores, the end-to-end latency of SUMMA and Cannon increases instead of decreasing. This is because the per-core computation cost drops sublinearly as cores increase beyond a threshold (due to fixed overheads like function calls and logic checks). At the same time, communication overhead grows linearly, as seen from the shaded areas in Figure 9, leading to worse overall performance. In contrast, MeshGEMM only shows a slight communication overhead increase at 720×720 cores and maintains stable end-to-end latency. This is because its interleave communication bounds the per-step communication cost to a constant, independent of core count (the total communication overhead grows only because the number of steps increases in all dist-GEMM algorithms). Thus, MeshGEMM can achieve a better overlapping between communication and computation even under large-scale fine-grained parallelism (each core is only responsible for a small amount of data).

Scaling matrix size. We also evaluate MeshGEMM with larger matrix sizes, transforming GEMM into a more computationally intensive operation. At large scales, though the cost of communication becomes less significant, MeshGEMM maintains its scalability and outperforms SUMMA and Cannon by a wide margin, reducing total cycles by around 17%.

An interesting observation in Figure 9 is that for GEMM 8K, communication cycles decrease as core count increases. This occurs because when processing large data volumes, communication is bandwidth-bound rather than latency-bound. Increasing the number of cores not only boosts aggregate compute power to reduce computation overhead but also increases aggregate bandwidth to lower overall communication cost.

7.3 MeshGEMV

We evaluate MeshGEMV and the default GEMV implementation on Cerebras (pipeline allreduce) across various core scales and matrix shapes.

Communication latency bottleneck. Figure 10 shows that communication becomes the primary bottleneck for dist-GEMV computations on WSE-2, especially when the parallelism scale is large relative to the computation per core, where communication overhead can dominate 90% of the total. However, MeshGEMV significantly reduces communication overhead compared to the Cerebras baseline GEMV, achieving about 4.6× higher end-to-end performance. This is because large-scale dist-GEMV requires long-distance allreduce communication, and MeshGEMV uses K-tree Allreduce to maximize parallelism in the allreduce process, minimizing communication and computation costs along the critical path.

Model	LLaMA3-8B	LLaMA2-13B
Concat-based (PagedAttention)	382	16
Shift-based (WaferLLM)	137548	6168

Table 5: Maximum decode output length

GEMV	[1,16K]×[16K,16K]			[1,32K]×[32K,32K]		
TP in SGLang(A100)	1 GPU	8 GPUs	2×8 GPUs	1 GPU	8 GPUs	2×8 GPUs
Time (ms)	0.336	0.253	0.340	1.231	0.341	0.339
MeshGEMV (WSE-2) Time (ms)	0.0012			0.00203		
A100/WSE-2 Energy Ratio	**7.47**	**44.97**	**120.88**	**16.17**	**35.83**	**71.25**

Table 6: Comparing MeshGEMV (WSE-2) with TP in SGLang(A100) of GEMV latency and energy.

Scaling core count and matrix size. We also evaluate MeshGEMV with larger matrix sizes. For GEMV 8K and 16K compared to 2K, the baseline shows a trend that the end-to-end overhead first decreases and then increases. The initial decrease comes from increased core count, where the higher aggregate compute power and bandwidth reduce computation and communication costs. However, as the core count grows, the communication latency of allreduce eventually overtakes computation and bandwidth as the dominant bottleneck. In contrast, MeshGEMV's inflection point appears later, which means that for the same data size, MeshGEMV's communication overhead grows much more slowly with increasing core count compared to the baseline, demonstrating better scalability. This enables more flexible on-chip mapping policies, such as using more cores to store additional LLM parameters to meet memory limitations (M) without introducing significant extra overhead.

7.4 Shift-based KV cache management

We also compare the shift-based KV cache management with the concat-based KV cache management in PagedAttention. We evaluate KV cache capacity on LLaMA3-8B and LLaMA2-13B using the same settings as the end-to-end inference evaluation in Section §7.1. Table 5 shows that WaferLLM's shift-based KV cache management supports 360× and 385× more tokens than the concat-based method for LLaMA3-8B and LLaMA2-13B, respectively. This improvement results from balanced core utilization and the resolution of skewed data issues achieved by the shift-based approach.

7.5 Comparison with GPUs

We compare WaferLLM against the state-of-the-art LLM inference system on GPUs using Cerebras WSE-2 and NVIDIA

Prefill (4K CTX)	LLaMA3-8B			LLaMA2-13B	
SGLang(A100) TPR	1 GPU	8 GPUs	2×8 GPUs	1 GPU	8 GPUs
	13988	17361	13994	7805	12287
WaferLLM WSE-2 TPR	27686			17498	
A100/WSE-2 Energy Ratio	**0.05**	**0.34**	**0.84**	**0.06**	**0.30**

Table 7: Comparing WaferLLM (WSE-2) with SGLang(A100) in prefill throughput and energy.

Decode (4K CTX)	LLaMA3-8B			LLaMA2-13B	
SGLang(A100) TPR	1 GPU	8 GPUs	2×8 GPUs	1 GPU	8 GPUs
	78	260	164	48	175
WaferLLM WSE-2 TPR	2700			2039	
A100/WSE-2 Energy Ratio	**0.92**	**2.22**	**7.02**	**1.13**	**2.49**

Table 8: Comparing WaferLLM (WSE-2) with SGLang(A100) in decode throughput and energy.

A100. To compare against the H100 fairly, we would need access to the WSE-3, which is a 5nm manufacturer but unavailable to us.

GEMV. We compared MeshGEMV with the GEMV parallelization strategy that follows SGLang's [53] multi-GPU tensor parallelism, while the computation on each GPU is accelerated using the cuBLAS library. Shown by Table 6, compared to a single A100 GPU, MeshGEMV outperforms by 280-606× with different matrix sizes, showcasing the advantages of providing substantial memory bandwidth through wafer-scale devices. This also translates to 7.5-16× greater energy efficiency, reflecting the benefits of wafer-based connections (connecting on-chip memory) over PCB-based ones (connecting off-chip HBM) in GPUs.

Distributed GEMV on multi-GPUs shows limited scalability: a single A100 yields the highest energy efficiency per FLOP, and performance improves only 1.32× from one to eight GPUs, then degrades at sixteen. This inefficiency stems from memory intensity and communication overhead over NVLink and IB. In contrast, MeshGEMV exploits WSE-2's low-latency NoC to scale across hundreds of thousands of cores, achieving 166-210× higher performance and 45-70× better energy efficiency at peak compared to the best GPU cluster results. Moreover, GEMV exhibits similar performance for 16K and 32K tasks on two machines with 16 GPUs, mainly because communication overhead dominates the computation in distributed GEMV across two nodes.

Despite these advantages, MeshGEMV does not achieve the theoretical 7,000× improvement. Profiling identifies three contributing factors: (i) WSE-2 cores, still in their second generation, cannot fully overlap memory access and computation; (ii) edge cores are underutilized; and (iii) NoC long-range communication overhead persists, despite MeshGEMV mitigating it effectively. We anticipate these gaps will continue to narrow as wafer-scale accelerators mature.

LLM inference. The maximum TPR achievable on GPU clusters is significantly lower than on WSE-2. We compare WaferLLM with SGLang on an A100 multi-GPUs cluster and find that WaferLLM delivers a 6-20× higher TPR across a range of input/output lengths and model sizes from 8B to 72B, with the gap widening for longer outputs and larger models (Table 2). As shown in Tables 7 and 8, scaling SGLang from 1 to 8 GPUs yields only 1.2-1.6× prefill and 3.3-3.6× decode speedups, far below ideal linear scaling. Performance further degrades when scaling to 16 GPUs, due to inter-node communication bottlenecks. As a result, peak TPR for dense models is typically reached within a single 8-GPU node, where WaferLLM still outperforms SGLang up to 20×. For use cases demanding high per-request throughput, wafer-scale accelerators are substantially more capable than conventional xPU-based systems.

In terms of energy efficiency, though a WSE-2 chip has 47× the area and 37× the power and cost of an A100, WaferLLM still achieves a 2-2.5× energy efficiency advantage at SGLang's optimal multi-GPU result, owing to GPUs' nonlinear scaling on decode. This advantage is especially valuable for long-output scenarios like test-time scaling, where serving cost is critical. However, compared to GEMV, WaferLLM's decode energy efficiency advantage is reduced due to: (i) limited local SRAM (48KB) on WSE-2 cores, which hinders efficient tensor parallelism and necessitates pipeline parallelism, causing up to 5× underutilization; and (ii) GPU-optimized LLaMA models with narrow layers that constrain layer placement and worsen utilization on WSE-2.

8 Implementation Detail and Future Direction

We discuss the current limitations of WaferLLM and wafer-scale accelerators and envision their future solutions:

Hardware architecture. The performance of WaferLLM is currently constrained by execution bubbles caused by the need for pipeline parallelism. Increasing a core's local memory by 5-6× could eliminate the need for pipeline parallelism, enabling full tensor parallelism, as on vLLM and SGLang. Wafer-scale chip designers are already moving in this direction. Cerebras WSE-3 retains the same NoC configuration but improves per-core efficiency and local memory, while Tesla's Dojo incorporates 1MB of per-core memory.

Memory-to-Compute Ratio. LLM decoding demands a near 1:1 memory-to-compute ratio. However, GPUs like A100 have limited on-chip SRAM, forcing frequent off-chip memory access and yielding a poor ratio of 1:312 (FP16). Multi-GPU setups exacerbate this due to heavy inter-GPU communication. In contrast, Cerebras WSE-2 maps most model weights onto on-chip memory via a low-latency NoC, achieving near-ideal locality and approaching a 1:1 ratio. To fully realize this balance on mesh-based NoCs, adherence to the PLMR model is essential, enabling WaferLLM to outperform GPU-based systems by orders of magnitude in TPR.

Handle reliability issues. Currently, Cerebras WSE-2/3 handles faults by hardware, only exposing healthy cores (organized in a mesh) to software, with no explicit handling required at the software level. Moreover, redundant cores and links are built in at fabrication, and the on-chip SoC dynamically remaps and reroutes around defects at runtime, ensuring minimal performance impact at a low redundancy cost. Meanwhile, wafer-scale chip makers recently reported a 93% functional wafer area, higher than 70–80% in commercial GPUs [8], due to the smaller area per core design. Over two years of WSE-2 deployment, we have observed high reliability, confirming the effectiveness of these fault-tolerance mechanisms in real-world use.

Various model architecture. WaferLLM is also beneficial for MoE as it shares key operators with dense LLMs, including MeshGEMM, MeshGEMV, and shift-based KV cache management. The main difference is the all-to-all communication between attention and expert layers, which we implement using WSE-2's NoC multi-cast operations. Further optimizations for sparse models, such as offloading and sparse attention, are among our future research.

Beyond Cerebras WSE. While evaluated on Cerebras WSE, the PLMR model generalizes to emerging mesh-like architectures such as Tesla Dojo, which also feature hundreds of thousands of cores with local memory and constrained NoC routing. Variants like 2D torus or hybrid mesh-switch topologies also conform to PLMR. Our design for MeshGEMM and MeshGEMV targets worst-case 2D mesh and remains competitive across such platforms. Beyond on-chip meshes, chip-to-chip mesh interconnects, as seen in Tenstorrent's core- and card-level meshes, also align well with PLMR. Looking ahead, advancements in wafer-scale integration, such as TSMC's projected 40× density increase by 2027, further reinforce the long-term applicability of our approach.

9 Related Work

Deep learning frameworks and compilers. Current deep learning frameworks and compilers, such as PyTorch, TensorFlow, and XLA [1, 10, 28, 34, 36, 37, 45, 51, 54, 57], are designed for shared memory architectures and use a tile-based "load-compute-store" computation model. While effective for shared memory, this model ignores the unique characteristics of PLMR devices, making it inefficient for wafer-scale AI chips. LLM frameworks such as vLLM and TensorRT-LLM [20,56] have emerged to support modern LLMs but rely on frameworks and compilers designed for shared memory architectures (e.g., PyTorch [34]), inheriting similar limitations on wafer-scale chips.

Distributed GPU and TPU systems. The on-chip distributed memory architecture could theoretically be treated as a distributed LLM system, as studied in prior works [20, 23, 35, 49, 52, 56]. However, such systems, designed for GPU and TPU pods (up to thousands of nodes), rely on more capable routers and lack local memory constraints, making them misaligned with the PLMR model. These approaches are complementary to our focus on on-chip scaling.

Systolic array. Systolic array architectures [22], used in AI accelerators such as Amazon Trainium and Google TPU, focus on the design of small cores rather than larger wafer-scale accelerators. With limited processing elements (usually up to hundreds) in a core, they are not PLMR devices but complement WaferLLM. For example, a Cerebras WSE core could employ a systolic array to accelerate local GEMM operations.

Dataflow architectures. Prior research has explored computation on dataflow architectures that account for inter-core connections. TENET [26] maps computation spatially and temporally to connected cores in a dataflow pattern. DISTAL [50] enables scheduling over distributed clusters using a dataflow approach. SambaNova [14] combines model and pipeline parallelism for DNN execution. However, none of these works scale computation to wafer-scale chips.

Wafer-scale allreduce. Recent research [27] has investigated wafer-scale allreduce, but a single allreduce cannot fully parallelize GEMV or support full LLM inference as achieved by WaferLLM. Additionally, this prior work is a specific instance of the K-tree allreduce proposed in WaferLLM.

10 Conclusion

We envision this paper as a foundational step in exploring the potential of wafer-scale computing for LLMs. The simple yet effective PLMR model has revealed significant opportunities, guiding the development of the first wafer-scale LLM parallelism solution and scalable GEMM and GEMV algorithms for wafer-scale accelerators. Despite the limitations of the current software stack for wafer-scale devices, our approach achieves orders-of-magnitude improvements in both performance and energy efficiency. We hope this work inspires greater focus on wafer-scale computing and advances the path toward a more sustainable future for AI.

11 Acknowledgments

We sincerely thank our shepherd, Marco Canini, and the OSDI reviewers for their insightful feedback that greatly improved this paper. We are also grateful to Yuqing Xia (Microsoft) for her valuable input during the development of this work.

We acknowledge the hardware resources provided by the Edinburgh International Data Facility (EIDF) and the Edinburgh Parallel Computing Centre (EPCC), particularly their support in granting access to the Cerebras WSE systems. We thank Nick Johnson for coordinating access and Mark Parsons for supporting the hardware acquisition and our project.

Finally, we appreciate the generous assistance of Leighton Wilson and Mathias Jacquelin from Cerebras for their patience and detailed responses to our many questions regarding CSL and the Cerebras WSE.

References

[1] M. Abadi, P. Barham, J. Chen, et al. TensorFlow: A system for large-scale machine learning. *OSDI 2016*, 2016.

[2] Advanced Micro Devices. AMD optimizes EPYC memory with NUMA. White paper, Advanced Micro Devices, Inc., 2023.

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[4] AMD. AMD XDNA adaptive architecture, 2023. Accessed: 2024-11-29.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[6] Lynn Elliot Cannon. *A cellular computer to implement the kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[7] Cerebras Systems. GEMM with collective operations. Accessed: 2024-10-05.

[8] Cerebras Systems. 100× defect tolerance: How cerebras solved the yield problem, 2022. Accessed: 2025-04-29.

[9] Cerebras Systems. Benchmark GEMV collectives, 2023. Accessed: 2024-11-29.

[10] T. Chen et al. TVM: An automated end-to-end optimization stack for deep learning. *SSP 2018*, 2018.

[11] Jaeyoung Choi, Jack J. Dongarra, and David W. Walker. Parallel matrix transpose algorithms on distributed memory concurrent computers. *Parallel Computing*, 21(9):1387–1405, 1995.

[12] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2024.

[13] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[14] Mark Harris. SambaNova's new AI chip and the quest for efficiency, 2023. Accessed: 2024-11-29.

[15] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Hanyu Dong, and Yu Wang. Flashdecoding++: Faster large language model inference on GPUs. *arXiv preprint arXiv:2311.01282*, 2023.

[16] Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

[17] Norman Jouppi, Cliff Young, et al. Tensor processing units for machine learning: An introduction. Technical report, Google Inc., 2017.

[18] Patrick Kennedy. Tenstorrent Blackhole and Metalium for standalone AI processing, 2024. ServeTheHome, Hot Chips 2024 Coverage.

[19] Jinwoo Kim, Venkata Chaitanya Krishna Chekuri, Nael Mizanur Rahman, Majid Ahadi Dolatsara, Hakki Mert Torun, Madhavan Swaminathan, Saibal Mukhopadhyay, and Sung Kyu Lim. Chiplet/interposer co-design for power delivery network optimization in heterogeneous 2.5-d ICs. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 11(12):2148–2157, 2021.

[20] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[21] Mark LaPedus. TSMC bets big on advanced packaging, 2023. Accessed: 2024-11-29.

[22] Ching-Jui Lee and Tsung Tai Yeh. ReSA: Reconfigurable systolic array for multiple tiny DNN tensors. *ACM Transactions on Architecture and Code Optimization*, 21(3):43:1–43:24, 2024.

[23] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[24] Sean Lie. Cerebras architecture deep dive: First look inside the hardware/software co-design for deep learning. *IEEE Micro*, 43(3):18–30, 2023.

[25] Yiqi Liu, Yuqi Xue, Yu Cheng, Lingxiao Ma, Ziming Miao, Jilong Xue, and Jian Huang. Scaling deep learning computation over the inter-core connected intelligence processor with T10. In *Proceedings of the ACM*

SIGOPS 30th Symposium on Operating Systems Principles, pages 505–521, 2024.

[26] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. TENET: A framework for modeling tensor dataflow based on relation-centric notation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 720–733, 2021.

[27] Piotr Luczynski, Lukas Gianinazzi, Patrick Iff, Leighton Wilson, Daniele De Sensi, and Torsten Hoefler. Near-optimal wafer-scale reduce. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, page 334–347. ACM, June 2024.

[28] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[29] Xiaoning Ma, Qinzhi Xu, Chenghan Wang, He Cao, Jianyun Liu, Daoqing Zhang, and Zhiqiang Li. An electrical-thermal co-simulation model of chiplet heterogeneous integration systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 32(10):1769–1781, 2024.

[30] Meta AI. Introducing MTIA: Meta's next-generation training and inference accelerator for AI, 2024. Accessed: 2024-12-10.

[31] Microsoft Azure. Azure Maia: For the era of AI from silicon to software to systems, 2023. Accessed: 2024-11-29.

[32] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[33] OpenAI. Openai o3 and o4-mini system card. https://openai.com/index/o3-o4-mini-system-card/, 2025. Accessed: 2025-05-03.

[34] A. Paszke, S. Gross, S. Chintala, et al. Automatic differentiation in PyTorch. *NIPS 2017*, 2017.

[35] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[36] J. Rock et al. XLA: Optimizing TensorFlow for high performance. *Google Research*, 2017.

[37] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, 2023.

[38] Shukri J. Souri, Kaustav Banerjee, Amit Mehrotra, and Krishna C. Saraswat. Multiple Si layer ICs: motivation, performance analysis, and design implications. In *Proceedings of the 37th Annual Design Automation Conference*, pages 213–220, 2000.

[39] Cerebras Systems. Cerebras and g42 break ground on condor galaxy 3, an 8 exaflops ai supercomputer. https://cerebras.ai/press-release/cerebras-g42-announce-condor-galaxy-3, 2024. Accessed: 2025-05-03.

[40] Cerebras Systems. Cerebras powers perplexity sonar with industry's fastest ai inference. https://www.cerebras.ai/press-release/cerebras-powers-perplexity-sonar-with\-industrys-fastest-ai-inference, 2025. Accessed: 2025-05-03.

[41] Emil Talpes, Douglas Williams, and Debjit Das Sarma. DOJO: The microarchitecture of Tesla's exa-scale computer. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–28, 2022.

[42] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, pages 5998–6008. Curran Associates, Inc., 2017.

[44] James Wang. Cerebras brings instant inference to mistral le chat. https://cerebras.ai/blog/mistral-le-chat, 2025. Accessed: 2025-05-03.

[45] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan

Yang, Ting Cao, et al. Ladder: Enabling efficient low-precision deep learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 307–323, 2024.

[46] Tianqi Wang, Fan Feng, Shaolin Xiang, Qi Li, and Jing Xia. Application defined on-chip networks for heterogeneous chiplets: An implementation perspective. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1198–1210, 2022.

[47] Wikipedia contributors. Static random-access memory, 2024. Accessed: 2024-12-10.

[48] Wikipedia contributors. Wafer-scale integration, 2024. Accessed: 2024-12-10.

[49] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. LoongServe: Efficiently serving long-context large language models with elastic sequence parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 640–654. ACM, 2024.

[50] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. DISTAL: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 286–300, 2022.

[51] Y. Zhao et al. Ansor: A compiler stack for auto-tuning tensor programs. *IEEE Transactions on Software Engineering*, 2020.

[52] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[53] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024.

[54] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[55] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210. USENIX Association, 2024.

[56] Yuxiao Zhou and Kecheng Yang. Exploring TensorRT to improve real-time inference for deep learning. In *2022 IEEE 24th International Conference on High Performance Computing & Communications*, pages 2011–2018. IEEE, 2022.

[57] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.