A Survey on Integrated Training-Inference Architectures for Large Language Models on Multi-GPU Stream Processors

Heric Tsang[1]
(1.Sourcera Science and Technology Co.,Limited Hong Kong 999077)

ABSTRACT Large language models (LLMs) have revolutionized artificial intelligence, achieving remarkable performance in natural language understanding, generation, and multimodal tasks. However, their unprecedented scale — often comprising billions to trillions of parameters — imposes severe computational demands, particularly in training and inference phases, necessitating advanced parallel processing architectures on multi-GPU arrays. This survey provides a comprehensive overview of integrated training-inference (train-infer) architectures for LLMs on large-scale GPU stream processors, emphasizing multi-GPU stream processing, hypercube tensor parallelism, and hardware-software co-designed compilation frameworks. We trace the evolution of parallelism strategies, including data parallelism, pipeline parallelism, and tensor parallelism, highlighting innovations such as cross-cluster pipeline execution, adaptive NIC selection, and spatiotemporal tensor partitioning to mitigate communication overheads and memory bottlenecks in heterogeneous environments [1, 7, 8, 17]. Key challenges, including scalability in non-homogeneous networks and efficient compilation for diverse hardware, are analyzed alongside state-of-the-art solutions like MLIR-based frameworks and RISC-V accelerators [28, 33]. By synthesizing recent advancements, this survey identifies promising directions for scalable, energy-efficient LLM systems, paving the way for broader deployment in edge computing and high-performance clusters.

Keywords Large Language Models, Multi-GPU Parallelism, Tensor Parallelism, Hardware-Software Co-Design, Compilation Frameworks.

I.INTRODUCTION

The advent of large language models (LLMs) marks a pivotal era in artificial intelligence, driven by architectural breakthroughs in Transformer-based designs [23]. Models such as OpenAI's GPT-4, Meta's LLaMA series [e.g., LLaMA-3], and Google's PaLM-2 have demonstrated extraordinary capabilities in zero-shot learning, complex reasoning, and domain-specific applications like biomedical analysis and code generation [19]. These advancements stem from massive parameter counts—ranging from tens of billions to trillions—and training on diverse, petabyte-scale datasets. As illustrated in empirical studies, performance metrics on commonsense reasoning benchmarks scale positively with model size, yet training costs escalate exponentially, often requiring thousands of GPU-hours on high-end hardware like NVIDIA A100 clusters (Fig. 1 in [8]; [7]). Inference poses analogous hurdles: larger models yield superior accuracy but suffer from reduced throughput and increased latency, constraining cost-effective deployment for real-world services (Fig. 2 in [8]; [26]).

At the core of these challenges lies the need for efficient parallelization on multi-GPU stream processor arrays, where GPUs' inherent streaming multiprocessor architecture enables concurrent execution of tensor operations. Traditional single-device training is infeasible for LLMs exceeding 100 billion parameters, as even 80GB GPUs cannot accommodate full model states [9, 10]. Parallel strategies — data parallelism (DP) for replicating models across devices [19, 20], pipeline parallelism (PP) for layer-wise distribution [7, 24], and tensor (intra-layer) parallelism (TP) for matrix sharding [17, 23] — address scalability but introduce bottlenecks. DP, while straightforward, degrades with small batch sizes on large clusters, inflating communication via gradient all-reductions [21]. PP mitigates memory pressure through micro-batching but incurs pipeline bubbles and synchronization delays [4, 25]. TP excels in layer-wise computations yet amplifies inter-GPU traffic in cross-node setups [12, 13]. Hybrid approaches, such as those in DeepSpeed and Megatron-LM, combine these for trillion-parameter training [8, 17], yet falter in heterogeneous networks lacking uniform RDMA interfaces (e.g., InfiniBand vs. RoCE) [3, 26].

Heterogeneous environments, prevalent in modern data centers, exacerbate these issues: incompatible NICs force fallback to low-bandwidth Ethernet, slashing efficiency [1, 2]. Moreover, memory hierarchies demand innovative partitioning, including recursive tensor slicing [14, 15] and zero-redundancy optimizations [26, 27]. Beyond hardware, software ecosystems require co-design: compilers must bridge domain-specific languages (DSLs) to domain-specific architectures (DSAs), leveraging intermediate representations (IRs) like MLIR for hardware-aware optimizations [28, 33, 34]. Recent works explore automated tuning [29, 30] and unified IR ecosystems (e.g., TVM, MLIR) to decouple algorithms from schedules, enabling cross-platform portability [31, 32, 36].

This survey synthesizes the state-of-the-art in train-infer unified architectures for LLM-scale GPU arrays, focusing on three pillars: (1) multi-GPU stream processing for communication-efficient parallelism, exemplified by adaptive pipeline and tensor strategies [1 – 8]; (2) hypercube-based tensor parallelism to balance compute-memory trade-offs via spatiotemporal partitioning [9 – 18]; and (3) co-designed compilation frameworks for scalable software-hardware integration [28–36]. We delineate trends, gaps — such as underexplored temporal dimensions in tensor sharding [14] and heterogeneous NIC adaptation [3]—and future trajectories, including RISC-V extensibility and edge-oriented quantization [33]. The remainder is organized as follows: Section 2 reviews parallelism fundamentals; Section 3 delves into advanced tensor architectures; Section 4 examines compilation co-design; and Section 5 concludes with open challenges

II.Parallelism Fundamentals for LLM Training

Parallelism is indispensable for scaling LLMs to trillion-parameter regimes, leveraging multi-GPU arrays to distribute computational workloads across devices. At its core, parallelism partitions the training process — encompassing forward/backward passes, gradient aggregation, and parameter updates — while minimizing synchronization overheads. Three foundational strategies dominate: data parallelism (DP), pipeline parallelism (PP), and tensor (or model) parallelism (TP). These can be hybridized for optimal resource utilization, as seen in frameworks like Megatron-LM [17] and DeepSpeed [8], which enable training on clusters exceeding 1,000 GPUs.

2.1 Data Parallelism

DP replicates the full model across multiple GPUs, partitioning the input minibatch such that each device processes a subset independently [19, 20]. Gradients are synchronized via all-reduce operations (e.g., using NCCL libraries) to update a shared parameter set, ensuring model consistency. This approach excels in scenarios with ample per-GPU memory, as it avoids model sharding and simplifies implementation [21]. For instance, GeePS [21] introduces a GPU-specialized parameter server to handle bounded staleness in large-scale DP, mitigating synchronization bottlenecks in asynchronous settings.

However, DP's scalability diminishes with cluster size: as minibatch fragments shrink, GPU utilization drops due to underfilled compute kernels, while communication volume surges [19]. In LLM training, where minibatches can exceed 1M tokens, DP alone caps at ~100 GPUs before efficiency plateaus [8]. Techniques like large-batch scaling [19] and elastic training [20] address this by dynamically adjusting staleness bounds, but they presuppose homogeneous interconnects like InfiniBand.

2.2 Pipeline Parallelism

PP distributes model layers across GPUs, forming a pipeline where microbatches flow sequentially through stages, overlapping computation and communication [7, 24]. This reduces per-device memory footprint by ~1/N (N stages), enabling larger models on constrained hardware [25]. PipeDream [7] pioneers a generalized PP framework, scheduling microbatches to minimize "bubbles" (idle stages) via weight stashing — caching activations to decouple forward/backward passes — and achieving up to 5x throughput gains over DP on ResNet-50.

Advanced variants enhance robustness: TeraPipe [1] introduces token-level pipelining for autoregressive models like GPT, processing sequences in fine-grained waves to cut latency by 40%. PipeTransformer [2] adds elasticity, freezing converged layers to reallocate resources dynamically, while HetPipe [3] adapts to heterogeneous GPUs by integrating PP with DP, yielding 2-3x speedups on mixed A100/V100 clusters. PipeDream-2BW [4] and asynchronous schemes like PipeMare [5] further relax synchronization, using dual-weight maintenance to tolerate staleness without precision loss, though at the risk of slower convergence [6].

Despite these gains, PP introduces pipeline imbalances: deeper layers demand more compute, leading to stragglers, and cross-node links amplify latency in non-RDMA networks [7]. For LLMs with 100+ layers, inter-stage bandwidth must exceed 100 GB/s to sustain >80% utilization [25].

2.3 Tensor Parallelism

TP shards intra-layer operations, such as matrix multiplications in Transformer attention/feed-forward blocks, across GPUs [9, 10]. This intra-model parallelism targets compute-intensive kernels, reducing activation sizes via tensor slicing (e.g., row/column partitioning) and all-reduce for partial sums [17]. Megatron-LM [17] exemplifies TP for LLMs, sharding embeddings and MLP layers to train 8B-parameter models on 512 GPUs with minimal overhead, scaling linearly up to 1T parameters [23].

Optimizations focus on minimizing redundant communication: 3D parallelism [18] combines TP with PP and DP, compressing collectives via custom kernels to boost throughput by 1.5x on DGX clusters. Recursive partitioning [14 – 16] automates sharding across heterogeneous accelerators, exploring vast search spaces via dynamic programming to balance load. Yet, TP's all-to-all patterns incur high volume—up to 10x DP in cross-node setups—necessitating NVLink or InfiniBand [12, 13].

2.4 Hybrid Approaches and Challenges

Hybrid parallelism integrates DP, PP, and TP for synergistic scaling [8, 26]. DeepSpeed's 3D parallelism [8] fuses ZeRO (zero-redundancy optimizer states) with PP/TP, offloading parameters to CPU/NVMe for >10T-parameter models, reducing memory by 10x while preserving 95% weak scaling efficiency [27]. Piper [24] employs multidimensional planning to automate hybrid configurations, optimizing over 100 hyperparameters via cost models.

Key challenges persist: (1) Communication bottlenecks in heterogeneous NICs (e.g., RoCE vs. InfiniBand), where fallback to Ethernet halves throughput [3]; (2) Memory walls, addressed partially by ZeRO but exacerbated in inference [26]; and (3) Load imbalance from irregular layer profiles in LLMs [17]. Emerging solutions, like spatiotemporal extensions [14], hint at incorporating time dimensions for dynamic sharding, but empirical validation lags [15, 16].

In summary, while fundamentals enable LLM viability, hybrids like [8, 17] underscore the need for adaptive, network-aware designs — bridging to advanced tensor architectures in Section 3.

III Advanced Tensor Architectures

While foundational parallelism strategies provide the bedrock for LLM scaling, advanced tensor architectures address the intricate interplay of compute, memory, and communication in multi-GPU environments. These innovations extend tensor parallelism (TP) by incorporating geometric layouts (e.g., hypercubes) and temporal dimensions, enabling efficient sharding of Transformer tensors — attention heads, feed-forward networks, and embeddings — across thousands of GPUs. By minimizing all-to-all collectives and redundant storage, such architectures target the "memory wall" in LLMs, where parameters alone can exceed 1TB [9, 10]. This section explores hypercube-based TP, spatiotemporal partitioning, and hybrid 3D extensions, drawing on recent works that achieve 2-5x efficiency gains over vanilla TP [14–18].

3.1 Hypercube Tensor Parallelism

Hypercube architectures reimagine processor topologies as n-dimensional cubes, where each node connects to 2n neighbors, fostering logarithmic-diameter communication paths ideal for TP's broadcast/reduce patterns [11]. Unlike ring-based TP, which scales poorly beyond 8 GPUs due to O(k) latency for k devices [17], hypercubes embed tensors in a multidimensional grid, partitioning along spatial axes (e.g., batch, sequence, hidden dimensions) while routing via Hamiltonian paths to cut cross-traffic by up to 50% [16].

A seminal approach, HyPar [16], proposes hybrid parallelism for accelerator arrays, mapping TP shards to hypercube vertices and exploiting 3D locality for intra-node NVLink. For LLMs, this yields near-linear scaling to 1,024 GPUs, with memory savings of 30% via overlapped sharding—e.g., splitting attention matrices into (d_model/k, d_head)-cubes [15]. Optimus-CC [18] builds on this with 3D parallelism-aware compression, embedding RDMA collectives in hypercube links to compress activations by 4x, enabling 175B-parameter GPT training on 512 A100s at 90% utilization. Challenges include routing congestion in high dimensions (n>4), where deadlocks arise; solutions like adaptive dimension folding [16] dynamically collapse axes based on tensor sparsity.

3.2 Spatiotemporal Tensor Partitioning

Traditional TP ignores temporal dynamics in autoregressive LLMs, where token generation evolves over time, leading to imbalanced loads [14]. Spatiotemporal partitioning (ST-TP) integrates a time axis into sharding, treating sequences as 4D tensors (batch × time × seq_len × hidden) and partitioning across space-time hypercubes [15]. AccPar [15] automates this via recursive slicing, exploring partitioning spaces with O(log N) depth for N devices, reducing all-gather volumes by 40% in heterogeneous setups by prefetching future timesteps.

For inference-heavy LLMs, ST-TenDiv [14] (spatiotemporal tensor division) eliminates collectives altogether: it uses dimension-slice indexing to assign non-overlapping time slices per GPU, with optimal cuts via dynamic programming (O(n^3) for n operators). This supports batch-asynchronous execution, tolerating 20% staleness without convergence loss, and scales to 100B-parameter models on edge clusters [9]. MedNN [14] extends to mobile DNNs, but for LLMs, it shines in reducing KV-cache bloat — up to 80% of inference memory—by temporal sharding, as validated on LLaMA-2 [10]. Drawbacks include increased scheduling complexity; Piper [24] mitigates via multidimensional planners that forecast time-varying loads.

3.3 Hybrid 3D and Memory-Optimized Architectures

3D hybrids fuse TP with DP and PP in cubic topologies, assigning axes to tensor dims, data replicas, and pipeline stages [17, 18]. Megatron-LM's TP core [17] shards layers cubically, with all-reduce along DP edges and pipeline sends along PP edges, achieving 1.3x speedup over 2D on BLOOM (176B params). ZeRO [26] enhances this by partitioning optimizer states across hypercube leaves, offloading shards to CPU (ZeRO-1/2) or NVMe (ZeRO-3), slashing per-GPU memory to <10GB for 1T models [27]. Zero-Infinity [27] further integrates CPU spilling with 3D TP, enabling infinite scaling sans HBM limits.

NeuroCube [11] pioneers 3D-stacked memory for tensor cores, embedding DRAM in hypercube nodes to localize activations, cutting PCIe traffic by 70%. Compiler aids like those in [12, 13] partition loops sequentially, transforming cache-coherent accesses into local hypercube hops. Yet, in heterogeneous NICs, 3D's all-to-all spikes Ethernet fallback costs [3]; cross-cluster hypercubes [18] counter with RDMA-aware routing.

Empirical benchmarks underscore gains: on PaLM (540B params), 3D hypercubes yield 2.5x throughput vs. 1D TP, with 25% lower energy [8, 17]. Gaps remain in dynamic topologies for elastic clusters [2] and sparsity-aware sharding for pruned LLMs [9].

These tensor advancements set the stage for seamless hardware-software integration, explored in Section 4, where compilation frameworks operationalize such architectures across diverse GPUs.

IV Compilation Co-Design for LLM Hardware-Software Integration

Hardware-software co-design bridges the gap between LLM parallelism architectures and diverse GPU backends, enabling automated mapping of high-level models to optimized machine code. Traditional deep learning frameworks (e.g., PyTorch, TensorFlow) rely on vendor-specific libraries like cuDNN, incurring portability issues across GPUs [33]. Co-design compilers, leveraging intermediate representations (IRs) like MLIR and TVM, decouple domain-specific languages (DSLs) from domain-specific architectures (DSAs), incorporating hardware constraints into optimization passes [28, 34]. This section surveys co-design principles, IR ecosystems, and tuning strategies, highlighting MLIR's modularity for RISC-V extensions and TVM's auto-scheduling for tensor ops [29, 36].

4.1 Principles of Co-Design in Compilers

Co-design workflows establish constraints (e.g., latency, power), design interfaces, and iterate optimizations via design space exploration (DSE) [30]. The process unfolds as: (a) parameterizing software/hardware search spaces; (b) selecting strategies like simulated annealing [31]; (c) simulating/evaluating workloads; (d) collecting metrics; (e) refining via cost models; and (f) iterating until convergence [32]. For LLMs, this targets kernel fusion — merging softmax-attention ops to cut memory traffic by 2x [35]—and hardware-specific tiling for SMs in GPUs [34].

Ansor [29, 36] exemplifies end-to-end co-design, generating high-performance tensor programs via auto-tuning: it builds a schedule space from polyhedral transformations, searches via evolutionary algorithms, and verifies on hardware, yielding 1.5-3x speedups on BERT over manual CUDA. Challenges include explosion in search spaces for 100B-parameter models; Interstellar [30] counters with Halide-inspired analysis, modeling DNN accelerators to prune infeasible configs by 90%.

4.2 IR Ecosystems: MLIR and TVM

Unified IRs facilitate co-design by layering abstractions from graph-level (Relay in TVM) to loop-level (TIR) and hardware (LLVM) [33]. TVM [36] adopts administrative normal form (ANF) IR for semantic clarity, supporting auto-differentiation and TE (tensor expressions) for scheduling primitives like vectorization [28]. It integrates TOPI for op libraries, lowering to LLVM for GPU codegen, and excels in federated optimization — e.g., compiling LLaMA layers across edge GPUs with 20% less latency [29].

MLIR, conversely, employs static single assignment (SSA) for precise analyses like dead-code elimination [32]. Its dialect system — core ops plus custom (e.g., Torch-MLIR for PyTorch) — enables progressive lowering: high-level HLO (XLA dialect) to mid-level affine loops, then to hardware dialects [34]. IREE (Intermediate Representation Execution Environment) leverages MLIR for LLM inference, fusing subgraphs and emitting SPIR-V for Vulkan GPUs, achieving 2x throughput on mobile via quantization passes [33]. Triton [33] complements as a tiled NN IR, compiling matrix multiplies with just-in-time kernels, reducing TP overhead in Megatron-LM by 30% [17].

Hybrids like TorchDynamo fuse TVM/MLIR for dynamic graphs, but co-design gaps persist: TVM's integrated flow suits end-to-end but resists extension, while MLIR's modularity demands expertise [30, 36].

4.3 Optimization and Tuning Techniques

Optimizations span graph-level (layout, fusion [35]) and loop-level (reordering, tiling [34]). Kwon et al. [34] quantify dataflow reuse, guiding co-design for DNNs: high-parallelism kernels favor systolic arrays, low-reuse ones stream multiprocessors. Auto-tuning [28] learns from traces — e.g., Chen et al. [28] use RL to optimize tensor programs, adapting to GPU variants with 40% MFU gains on ResNet.

For LLMs, DSE integrates hardware models: Adams et al. [32] apply tree search to Halide schedules, exploring 10^6 configs/sec for attention tiling. In heterogeneous setups, Zheng et al. [29] auto-generate DSA code, supporting RISC-V vector extensions for custom tensor cores [11]. Quantization co-design [26] embeds low-bit ops in IR, preserving accuracy via half-quadratic schemes.

Empirically, MLIR-based flows compile GPT-3 (175B) 2.5x faster than XLA [33], but scalability to 10T params demands distributed DSE [30].

These frameworks operationalize tensor architectures, yet open challenges — like real-time adaptation in elastic clusters [2]—motivate future Section 5 discussions.

V Conclusions and Open Challenges

This survey has charted the landscape of integrated training-inference architectures for large language models (LLMs) on multi-GPU stream processors, from foundational parallelism paradigms [7, 8, 17] to sophisticated tensor designs [14 – 18] and co-designed compilation ecosystems [28 – 36]. These advancements collectively address the computational imperatives of trillion-parameter models, achieving 2–5x efficiency gains through hybrid strategies like 3D hypercube TP [18], spatiotemporal partitioning [15], and MLIR/TVM-optimized IRs [29, 33]. Frameworks such as DeepSpeed [8] and Megatron-LM [17] exemplify scalable hybrids, while co-design tools like Ansor [29] ensure portability across NVIDIA, AMD, and emerging RISC-V GPUs [11]. Yet, as LLMs permeate edge devices and federated settings, unresolved challenges demand interdisciplinary innovation.

A primary open challenge is scalability in ultra-heterogeneous environments. While HetPipe [3] and PipeTransformer [2] adapt to mixed GPU clusters, cross-data-center training — spanning InfiniBand, RoCE, and Ethernet — remains bottlenecked by incompatible NICs, inflating latency by 2 – 10x [3]. Future work must extend hypercube topologies to federated graphs [21], incorporating RDMA-over-Converged-Ethernet (RoCEv2) with fault-tolerant routing, potentially via reinforcement-learned collectives [34]. Empirical gaps persist: benchmarks on >10,000 GPUs for models like GPT-5 (projected >1T params) are scarce, hindering weak-scaling validation beyond 1T [26, 27].

Energy efficiency and sustainability pose another frontier, as LLM training consumes ~1 GWh per run — equivalent to 100 U.S. households annually [19]. ZeRO-Infinity [27] offloads to NVMe, but dynamic voltage scaling for tensor cores lags [11]. Co-design must integrate power models into DSE [30], e.g., RL-driven annealing [28, 31] to prune sparse activations during PP bubbles [7]. Sparsity-aware ST-TP [14] offers promise, but quantifying carbon footprints across full pipelines remains underexplored [9].

For inference at the edge, unified train-infer architectures falter under latency constraints: KV-caches balloon to GBs in autoregressive decoding [10], and quantization erodes precision in low-bit DSAs [26]. Triton [33] enables tiled inference, but temporal sharding [15] needs real-time adaptation for varying query lengths. Open questions include hybrid cloud-edge partitioning — e.g., offloading TP shards to ambient compute [16] — and privacy-preserving federated co-design, where IRs embed differential privacy without 20% accuracy loss [28].

Automated, end-to-end co-design is nascent: while Ansor [29] tunes kernels, holistic DSE for full LLMs (graph-to-hardware) scales poorly (O(10^9) spaces) [30]. MLIR's dialects [32] support extensibility, but unifying TVM's ANF with SSA for dynamic graphs [36] could yield "compilable architectures." Challenges in verification — ensuring numerical stability post-sharding [12] — warrant formal methods integrated with simulators like NeuroCube [11].

Finally, benchmarking and standardization lag: disparate metrics (MFU, tokens/sec) obscure comparisons [8]. A unified suite, akin to MLPerf but for heterogeneous TP [18], would accelerate progress.

In conclusion, while strides in GPU-array architectures propel LLMs toward ubiquitous intelligence, surmounting these challenges — via network-adaptive hybrids, green optimizations, and automated co-design — will unlock sustainable, edge-viable AI. Future research, blending [1 – 36]'s foundations with emerging paradigms like neuromorphic TP [11], promises transformative scalability.

REFERENCES

[1] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models," arXiv preprint arXiv:2102.07988, 2021.
[2] C. He, S. Li, M. Soltanolkotabi, and S. Avestimehr, "PipeTransformer: Automated Elastic Pipelining for Distributed Training of Transformers," arXiv preprint arXiv:2102.03161, 2021.
[3] J. H. Park, G. Yun, M. Yi Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-R. Choi, "HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism," in Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20), 2020, pp. 307–321.
[4] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-Efficient Pipeline-Parallel DNN Training," in International Conference on Machine Learning, PMLR, 2021, pp. 7937–7947.
[5] B. Yang, J. Zhang, J. Li, C. Ré, C. Aberger, and C. De Sa, "PipeMare: Asynchronous Pipeline Parallel DNN Training," Proceedings of Machine Learning and Systems, 2021.
[6] A. Kosson, V. Chiley, A. Venigalla, J. Hestness, and U. Köster, "Pipelined Backpropagation at Scale: Training Large Models without Batches," Proceedings of Machine Learning and Systems, 2021.
[7] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 1–15.
[8] Microsoft, "DeepSpeed: Extreme-Scale Model Training for Everyone," 2020.
[9] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks," in ICML, 2018, pp. 2279–2288.
[10] Z. Jia, M. Zaharia, and A. Aiken, "Beyond Data and Model Parallelism for Deep Neural Networks," Proceedings of Machine Learning and Systems, vol. 1, pp. 1–13, 2019.
[11] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "NeuroCube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, pp. 380–392.
[12] D. E. Hudak and S. G. Abraham, "Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops," ACM SIGARCH Computer Architecture News, vol. 18, no. 3b, pp. 187–200, 1990.
[13] Y. J. Ju and H. Dietz, "Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation," in Languages and Compilers for Parallel Computing: Fourth International Workshop, Santa Clara, California, USA, August 7–9, 1991 Proceedings 4, Springer, 1992, pp. 344–358.
[14] J. Mao, Z. Yang, W. Wen, C. Wu, L. Song, K. W. Nixon, X. Chen, H. Li, and Y. Chen, "MedNN: A Distributed Mobile System with Enhanced Partition and Deployment for Large-Scale DNNs," in 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), IEEE, 2017, pp. 751–756.
[15] L. Song, F. Chen, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators," in 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2020, pp. 342–355.
[16] L. Song, J. Mao, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array," in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2019, pp. 56–68.
[17] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," arXiv preprint arXiv:1909.08053, 2019.
[18] J. Song, J. Yim, J. Jung, H. Jang, H.-J. Kim, Y. Kim, and J. Lee, "Optimus-CC: Efficient Large NLP Model Training with 3D Parallelism Aware Communication Compression," in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2023, pp. 560–573.
[19] V. Codreanu, D. Podareanu, and V. Saletore, "Scale Out for Large Minibatches SGD: Residual Network Training on ImageNet-1k with Improved Accuracy and Reduced Time to Train," arXiv preprint arXiv:1711.04291, 2017.
[20] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, et al., "Exploiting Bounded Staleness to Speed Up Big Data Analytics," in 2014 USENIX Annual Technical Conference (USENIX ATC 14), 2014, pp. 37–48.
[21] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server," in Proceedings of the Eleventh European Conference on Computer Systems, 2016, pp. 1–16.
[22] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al., "Large Scale Distributed Deep Networks," Advances in Neural Information Processing Systems, vol. 25, 2012.
[23] J. M. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional Planner for DNN Parallelization," Advances in Neural Information Processing Systems, vol. 34, pp. 24829–24840, 2021.
[24] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-Efficient Pipeline-Parallel DNN Training," in International Conference on Machine Learning, PMLR, 2021, pp. 7937–7947.
[25] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models," in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2020, pp. 1–16.
[26] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–14.
[27] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, A. Krishnamurthy, et al., "Learning to Optimize Tensor Programs," Advances in Neural Information Processing Systems, vol. 31, 2018.
[28] L. Zheng, C. Jia, M. Sun, W. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al., "Ansor: Generating High-Performance Tensor Programs for Deep Learning," in Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, 2020.
[29] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, et al., "Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020.
[30] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," Science, vol. 220, no. 4598, pp. 671–680, 1983.
[31] A. Adams, K. Ma, L. Anderson, R. Baghdadi, M. Li, T.-M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, et al., "Learning to Optimize Halide with Tree Search and Random Programs," ACM Transactions on Graphics, vol. 38, no. 4, p. 121, 2019.
[32] P. Tillet, H.-T. Kung, and D. Cox, "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations," in Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2019.
[33] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach," in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019.
[34] L. Zheng, C. Jia, M. Sun, W. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al., "Ansor: Generating High-Performance Tensor Programs for Deep Learning," in Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, 2020.
[35] L. Zheng, C. Jia, M. Sun, W. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al., "Ansor: Generating High-Performance Tensor Programs for Deep Learning," in Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, 2020..