A Cross-Stack Approach to Efficient and Scalable Generative AI

Special Session Paper

Weitian Wang [§], Dhingra, Pratyush [*], Asif Ali Khan [†], Arkapravo Ghosh [‡],
Partha Pratim Pande [*], Jeronimo Castrillon [†], Priyadarshini Panda [‡], Cecilia De La Parra [§], Shubham Rai [§]
[*]Washington State University, Pullman, USA [†]TU Dresden, Germany [‡]University of Southern California, USA
[§]Robert Bosch Corporate Research, Renningen, Germany

***Abstract*—This paper presents a holistic, cross-stack vision for efficient and scalable generative AI, motivated by the view that efficiency is most readily improved when hardware, compiler, and algorithmic techniques are pursued together rather than in isolation. We connect four complementary perspectives: (i) foundational hardware innovations, including heterogeneous 2.5D/3D integration and in-memory computing, that reduce the compute, memory, and energy footprint of billion-parameter models; (ii) extensible compiler infrastructures, built around MLIR, that bridge emerging in-memory hardware to real applications and automate design-space exploration; (iii) cross-layer and algorithmic techniques—quantization, sparsification, and dataflow optimization—that compress and accelerate large language models from the cloud to the edge; and (iv) token reduction methods, such as token merging and clustering, that attack the quadratic cost of the transformer attention mechanism at the model level. By integrating hardware, compiler, and algorithm viewpoints into a single narrative, we outline how co-designed solutions across the stack can compound to enable affordable, scalable, and democratized access to generative intelligence.**

***Index Terms*—Generative AI, large language models, edge inference, in-memory computing, 2.5D/3D integration, MLIR compilers, quantization, sparsification, token merging, hardware-software co-design.**

I. Introduction

Over the past few years, generative AI has rapidly become a foundational technology for science, engineering, industry, and daily life. Transformer-based models—spanning text, images, audio, and video—have scaled to billions and now trillions of parameters, and their two core workflows, inference and fine-tuning, sit at the heart of a rapidly expanding set of applications. This progress has been driven largely by scale: larger models trained on more data tend to yield better capabilities. Scale, however, comes at a cost that is growing faster than the efficiency of the platforms meant to support it. The resulting computational, memory, and energy demands are especially challenging at the edge, where memory capacity, bandwidth, and power are tightly constrained, yet where privacy, latency, and availability make on-device deployment increasingly valuable.

Efficiency has been pursued vigorously at every layer of the computing stack, and each layer has delivered substantial gains on its own [1]. This motivates a cross-stack perspective, in which we explore *co-design*—decisions made jointly across hardware architecture, compiler infrastructure, and model-level algorithms—in the expectation that improvements at one layer can amplify, rather than be absorbed by, the others. To that end, this paper assembles four complementary perspectives, from device and architecture through compilers to algorithms, into a single, coherent view of how efficient and scalable generative AI can be achieved from the cloud to the edge.

The remainder of this paper follows the stack from the bottom up.

- **Section II (Hardware).** We describe how heterogeneous 2.5D/3D systems and in-memory computing can dramatically reduce the compute, memory, and energy footprint of billion-parameter generative models, enabling scalable and affordable inference and fine-tuning.
- **Section III (Compilers).** We present extensible compiler flows, built around the MLIR infrastructure, that abstract over individual in-memory technologies—crossbars, content-addressable memories, and bulk-wise logic—and automate mapping and design-space exploration, bridging emerging hardware to real applications.
- **Section IV (Cross-Layer Algorithms).** We discuss algorithm–hardware co-design techniques—quantization and sparsification with symmetric calibration, together with dataflow and packing optimizations—that compress LLMs and cut data-transfer overhead for efficient edge execution and low-memory fine-tuning.
- **Section V (Token Reduction).** We examine token merging and clustering methods that attack the quadratic cost of attention at the model level, compare their strengths and limitations across vision and language tasks, and introduce a learnable token clustering module for flexible compression.

Read together, these sections support a stronger claim than the simple accumulation of independent gains: cross-stack optimizations can mutually reinforce one another. For example, although INT4 quantization reduces model size and memory traffic, QServe reports 20–90% dequantization overhead on GPUs; co-designing the quantization scheme with data layout and kernels instead yields up to (2.4×) higher serving throughput than TensorRT-LLM on A100 [2]. More generally, token reduction and model compression decrease computation and data movement; a technology-aware compiler preserves these savings when mapping the transformed workload; and heterogeneous hardware provides an efficient substrate. A compressed, token-reduced model (Sections IV–V), mapped through Section III onto the platform of Section II, can therefore reach efficiency targets unavailable to any single layer in isolation. Section VI concludes with the resulting open challenges.

II. Co-Design of Heterogeneous 2.5D/3D Systems

The two core workflows of generative AI models, inference and fine-tuning, impose computational and energy demands that conventional platforms such as GPUs and Tensor Processing Units (TPUs) are unable to sustain [3], [4]. This burden arises from a fundamental mismatch between the heterogeneous computational kernels of generative AI and the homogeneous computing substrates. We now outline design principles for heterogeneous 2.5D/3D hardware platforms that reduce the computational footprint of generative AI by aligning each computational kernel with suitable memory and compute technology. We begin by characterizing the kernel heterogeneity inherent in LLMs and Mixture-of-Experts (MoE) models; next, we present our kernel-to-substrate mapping methodology alongside a co-designed on-die communication fabric.

A. Computational Heterogeneity in Generative AI Workloads

Inference in transformer-based LLMs operates in two distinct phases: prefill and decode. The prefill phase processes the input prompt in parallel and exhibits high arithmetic intensity, whereas the autoregressive decode phase generates a single token at a time and is bound by memory bandwidth, since every token requires fetching the weights and Key-Value (KV) cache entries [5]. Consequently, the two phases demand fundamentally different arithmetic intensity. Additionally, fine-tuning for these models require parameter update through gradient backpropagation, which is prohibitive for billion-parameter models. Parameter-efficient fine-tuning mitigates this by updating only a small subset of the parameters, of which Low-Rank Adaptation (LoRA) is a representative method [6]. LoRA freezes the pre-trained weights and introduces small trainable low-rank matrices, as described in Eq. 1:

$$Y = (W_O + AB)\, X = W_O X + ABX \tag{1}$$

where W_O denotes the pre-trained weights and $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times d}$ are the two trainable low-rank matrices where r is the rank and d represents the pre-trained weight matrix dimension. These low-rank adapters are much smaller in size compared to the original weights with $r << d$ [6]. This disaggregation into frozen and trainable weights during fine-tuning confines the updates to a small fraction of the model.

A comparable disaggregation arises architecturally in MoE inference, where the monolithic feed-forward network is replaced by N independent experts, of which a gating network activates only the top-k per token, as expressed in Eq. 2:

$$y = \sum_{i\in\mathcal{T}} G\left(x\right)_i \cdot E_i\left(x\right) \tag{2}$$

where $G\left(x\right)$ is the gating function and $\mathcal{T}$ is the set of selected expert indices. Although this sparsity keeps the active parameters per token small, the expert weights often constitute more than 90% of the model and impose GB-scale on-chip storage requirements.

Consequently, across both inference and fine-tuning workflows, the majority of weights remains static. Mapping these static weights to the memory-centric substrate is optimal in the decode phase given the low-arithmetic intensity. However, resident kernels during the decode phase impose conflicting requirements. The static weights are read operations and demand storage density, whereas the KV cache necessitates frequent read-modify-write updates and demands write endurance. Emerging non-volatile memory (NVM) devices such as ferroelectric field-effect transistors (FeFET) provide high density storage and 3D integration along with superior energy efficiency compared to conventional DRAM. However, their write endurance is typically limited to 10^8–10^{11} program/erase cycles [7]. In contrast, volatile memories such as SRAM and DRAM tolerate frequent writes but incur area, leakage, and refresh overheads [8]. Consequently, the memory-centric substrate and the mapping must itself be heterogeneous.

B. Heterogeneous Integration and Kernel Mapping

Heterogeneous integration, in which 2.5D interposers and 3D stacking combine disparate memories and computing cores within a single package, fulfills the requirements imposed by the generative workloads [9]. The advantages of using heterogeneity are twofold. First, the weight-stationary computation, which represents most of the compute operations, is performed in an energy-efficient manner by utilizing high-density memory-centric architectures. Three-dimensional vertical stacking of memory layers addresses the density requirement, necessary for the GB-level storage of the static weights

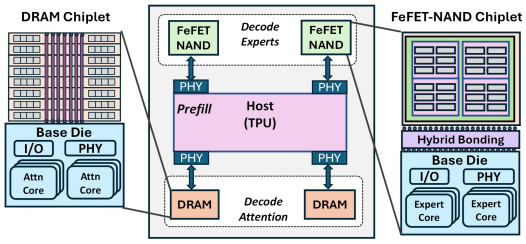


Fig. 1. A representative heterogeneous 2.5D/3D system. Memory-centric chiplets, each stacking a 3D memory array over a CMOS base die housing processing-near-memory cores, are integrated with a compute-centric host chiplet on a silicon interposer.

[10]. Second, the limitations of memory-centric architectures with dynamic operand computation are mitigated by offloading these computations to CMOS-based computing cores.

For inference, the compute-bound prefill phase and the lightweight gating network are mapped to a CMOS-compute chiplet, while the memory-bound decode phase is distributed across the memory-centric substrates. Similarly, fine-tuning using LoRA confines the updates to the low-rank matrices, the dense substrate holding the frozen weights is never written during adaptation, and the trainable parameters reside entirely on the CMOS computing cores that execute the backpropagation [11]. The same kernel-to-substrate mapping therefore serves both workflows without exposing the dense substrate to the write-centric traffic that would exhaust its endurance, while more than 90% of the matrix multiplications still execute on the energy-efficient memory-centric substrate. This principle holds for all heterogeneous architectures that pair a dense static substrate with endurance-tolerant computing cores, independent of the device chosen for either. Finally, since only the low-rank matrices differ across downstream tasks, a single resident backbone serves many tasks, and switching tasks transfers the low-rank parameters rather than the base-weights of the model.

Several three-dimensional (3D)-HI architectures consisting of NVM devices and CMOS-based computing cores have been developed, such as H3D-Transformer, and Atleus [11]–[13]. For MoE serving, Stratum employs tiered monolithically stacked 3D DRAM to hold the expert weights, whereas A3D-MoE and CHIME combine logic dies with DRAM through 3D heterogeneous integration [14]–[16].

A representative memory-centric chiplet vertically integrates the memory array over a CMOS base die housing systolic-array compute cores, coupled through Cu-to-Cu hybrid-bonding links that supply the requisite vertical bandwidth (∼5 TB/s) [17], [18]. These chiplets are integrated on a 2.5D silicon interposer through die-to-die links standardized by the Universal Chiplet Interconnect Express (UCIe) [19]. Here, the expert weights reside in high-density 3D FeFET-NAND PNM chiplets, whereas the KV cache and the attention-score computation execute within 3D DRAM-PNM chiplets, as illustrated in Fig. 1 [20]. The FeFET-NAND chiplet employs the mature vertical stacking technology, providing an aggregate 64 GB of on-chip storage that accommodates the multi-billion-parameter footprints of representative MoE models [21].

C. Communication Co-Design

Aligning kernels with substrates addresses the memory and compute bottleneck. However, a heterogeneous manycore architecture additionally necessitates a communication backbone co-designed with the traffic pattern. For the deterministic dataflow of dense LLMs, activations propagate unidirectionally from one layer to the next across the memory cores. In contrast, active experts are dynamically

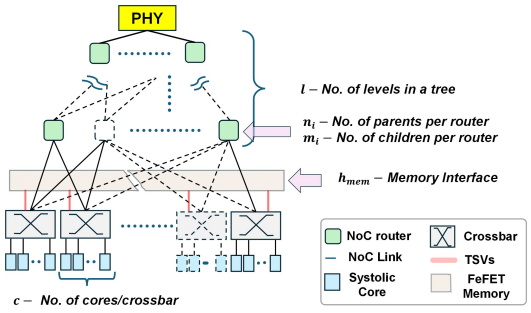


Fig. 2. Hierarchical NoC for the memory-centric base die. Local crossbars group c cores each and confine intra-expert traffic, while an upper-level tree (l levels, with m children and n parents per router) carries the scatter-gather traffic, with the memory interface injected at level h_{mem}.

allocated in MoE LLM models. The gating mechanism dictates the workload distribution for a batch of input tokens. This results in highly variable, non-deterministic traffic patterns that change based on the semantic content of the batch. Since expert selection is input-dependent, source-destination pairs and traffic volumes cannot be determined prior to inference in manycore architectures. The activations must be routed from the host to the selected experts (a one-to-many pattern), and the computed outputs aggregated back (a many-to-one pattern). The allocation among computing cores for the active experts creates distinct spatial traffic pattern for each user-dependent input. From a combinatorial perspective, allocating q cores among the active experts corresponds to partitioning the integer q, so the total number of unique traffic scenarios ($\mathcal{S}$) is given by the sum of partition functions in Eq. 3:

$$\mathcal{S} = \sum_{j=1}^{p} p_j\left(q\right) \tag{3}$$

where $p_j\left(q\right)$ is the number of integer partitions of q into exactly j positive parts, and p is the maximum number of experts that can be active concurrently. This combinatorial explosion of traffic patterns, combined with the synchronous gather that stalls the batch on the slowest expert, imposes a design challenge unique to MoE inference. Conventional Network-on-Chip (NoC) frameworks optimize interconnects for a single, deterministic compute mapping [22]. However, it is essential to optimize the NoC for the entire combinatorial space of traffic scenarios.

The one-to-many dispatch followed by a many-to-one aggregation makes a tree-based NoC a natural architectural fit [23]. However, connecting individual cores directly to a monolithic tree introduces congestion, particularly for the latency-sensitive intra-expert traffic generated when an expert is partitioned across multiple cores. To resolve this, a hierarchical NoC organization can be employed, as illustrated in Fig. 2. At the lowest level, cores are grouped and interconnected via local crossbars that provide non-blocking communication and confine intra-expert partial-sum and activation traffic locally. A tree topology forms the upper-level backbone, handling inter-crossbar traffic and communication with the host. The design is parameterized by the number of cores per crossbar (c), the number of tree levels (l), the number of children and parents per router at each level (m, n), and the level h_{mem} at which the memory interface injects the expert-weight traffic.

We formulate the NoC design as a Multi-Objective Optimization (MOO) problem over three objectives across the traffic space: the mean link utilization $\mu\left(d\right)$, the standard deviation of link utilization $\sigma\left(d\right)$, and the total NoC area $A\left(d\right)$ of each design d. Minimizing $\mu\left(d\right)$ preserves global bandwidth to prevent congestion, whereas minimizing $\sigma\left(d\right)$ bounds the utilization variance that would otherwise create tail-latency stalls. Since the topology is fixed at fabrication and must serve arbitrary models and routing behaviors over the lifetime of the chip, the optimization treats every traffic scenario as equally probable. Solving the MOO with a solver such as Archived Multi-Objective Simulated Annealing (AMOSA) yields a Pareto-optimal set of hierarchical topologies, from which a design is selected by trading NoC area against the latency sustained across the traffic space [24]. Evaluated across this combinatorial traffic space for a 32-core base die, the resulting hierarchical topology confines the compute–communication mismatch to under 2% for over 60% of valid configurations and reduces average NoC latency by 4.1× over a Ring interconnect, bounding the tail-latency stalls that a uniform mesh cannot avoid [17].

III. Compilers for In-Memory Computing Systems

The previous section focused on the technology advancements related to 2.5D/3D integration. Exploiting the potential of these novel systems, particularly when used with novel compute paradigms, requires automatic compilation flows that can target this heterogeneous landscape without requiring a new compiler for every technology.

A number of compilation flows for these computing in-memory novel architectures have emerged in recent years. Broadly, these efforts can be divided into three different levels of abstraction: assembly-level frameworks that consume pre-scheduled code [25], [26], loop-level frameworks employing polyhedral or pattern-based analysis [27]–[29], and graph-level frameworks that partition ML operator graphs across computing devices [30], [31]. A smaller set of tools bridges multiple levels through a hierarchical IR, including SongC [32], CIM-MLC [33], CIMFlow [34], OptiPIM [35], and ATiM [36]. Most of these frameworks specialize in a single CIM paradigm [37]–[40] or a single compute-near-memory (CNM) target [41]–[43]. Only a few frameworks namely PIM-DL [44], UniNDP [45], and CINM [46] support more than one CIM and CNM device families, and to our knowledge CINM is the only framework that seamlessly unifies both CIM and CNM execution paths in a retargetable MLIR pipeline.

In this section, we first provide a short background on the MLIR infrastructure and then present the CINM framework and its application to different CIM and CNM targets.

A. The MLIR Framework

MLIR is a compiler infrastructure for representing and progressively lowering intermediate representations (IRs) across abstraction levels and hardware targets [47]. Rather than fixing a single IR, MLIR provides a small set of *builtins* and lets compiler developers plug in custom *dialects* – logical groupings of operations, types, and attributes tied to a domain (e.g., `linalg` for linear algebra), a representation (e.g., `affine`), or a target (e.g., `gpu`). Programs are lowered from high-level, domain-specific dialects toward low-level, platform-specific ones through a sequence of dialect-to-dialect conversions, with each dialect's own type system and verifiers enforcing the conditions that must hold for it to be lowered correctly.

B. An Extensible MLIR-Based CINM Compilation Framework

CINM is an MLIR-based compilation framework for CIM and CNM systems (collectively referred to as CINM) [46]. As shown in Fig. 3, it introduces a device-agnostic `cinm` abstraction that implements the set of operations supported by the CIM and CNM targets. It also implements a set of reusable compiler transformations and analyses that are common to both CNM and CIM targets. The framework provides a placeholder to plug cost models such as [48]

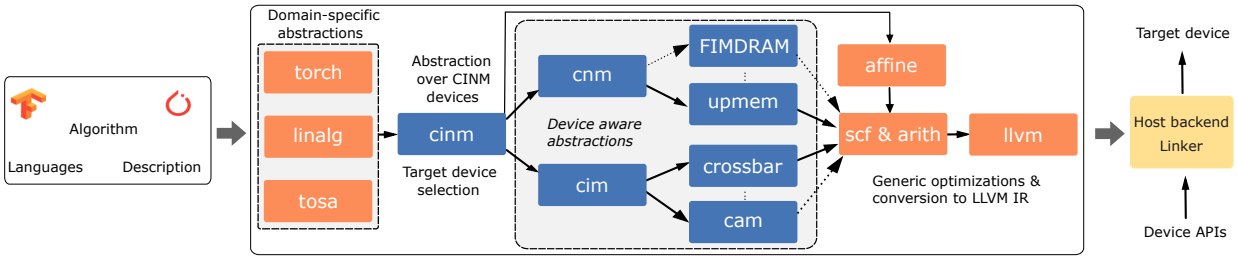


Fig. 3. A high-level overview of the CINM [46] lowering pipeline.

that can eventually be used to guide offloading and optimization decisions. The `cnm` and `cim` dialects abstract over the different types of CNM and CIM targets, respectively, and model operations that are common to these different paradigms. These include, for instance, the scatter/gather models for CNM systems and the compute and synchronization primitives for the fixed-function CIM targets. These abstractions still express computations independently of any specific memory technology. A set of backend dialects eventually model device-specific execution details (e.g., DPU/tasklet placement for UPMEM, crossbar tiling for a resistive CIM array).

An input application enters the CINM pipeline at a high-level dialect such as `linalg`, which is reachable from widely-used front-ends such as TensorFlow, ONNX, or PyTorch. The `linalg-to-cinm` conversion then (re)writes the program into a sequence of operations that the `cinm` dialect recognizes and from there, it is lowered to the device-aware `cim`, `cnm`, or `affine` dialects, and progressively down through `scf`/`arith` to `llvm` for code generation. Along this pipeline, CINM partitions the input program into CIM- and CNM-amenable regions and lowers each region independently via its target device's dialect.[1] Because the device-agnostic dialects abstract over all device-specific ones, the same input program can, in principle, be re-targeted to a new accelerator by adding a new lowering path (device-dialect) rather than rewriting the program. CINM has been shown to match hand-tuned UPMEM implementations on the PrIM benchmark suite [49], without exposing device-specific detail to the programmer.

C. Optimizing Flows: Crossbars, CAMs, Bulk-Wise Logic

This section presents the application of CINM to different CIM accelerators, focusing on the different paths through the CINM pipeline in Fig. 3.

1) Compilation for CIM-Crossbars: CINM's memristor backend builds on OCC [28], which targets memristor crossbars specifically. For a given input application, CINM first applies a set of hardware-agnostic analyses and rewriting passes at the `linalg` abstraction to detect matrix-vector (`gemv`) and matrix-matrix (`gemm`) operations – the only primitives that can be executed directly on a crossbar in constant time. These same passes also reduce arbitrary tensor contractions and convolutions down to `gemm` operations wherever possible. A *TTGT* (transpose-transpose-GEMM-transpose) pass automatically detects contractions and rewrites them as `gemm` (with the requisite set of pre- and post-processing operations). Only the `gemm` operation is offloaded to the crossbar, while the surrounding operations follow the ordinary CPU code-generation path. An analogous detection and rewriting is applied to spatial convolutions using the *Im2Col* transformation.

[1]Partitioning and offloading to CIM and CNM targets is presently semi-automated: the framework either greedily detects and offloads operations (e.g., `matmul` to CIM crossbars) or leaves the decision to the compiler developer.

```
// original tiled GEMM
scf.for %i = %c0 to %tiledRows step %c1 {
  scf.for %j = %c0 to %tiledCols step %c1 {
    %tileC = cim.copyTile(%C, %i, %j)
    scf.for %k = %c0 to %numTiles step %c1 {
      ...
      %tileB = cim.copyTile(%B, %k, %j)
      cim.write(%id, %tileB)
      ...
    }
    cim.storeTile(%tileC, %C, %i, %j)
  }
}
                    ⇓ transforms to
// loop interchanged GEMM
scf.for %k = %c0 to %numTiles step %c1 {
  scf.for %j = %c0 to %tiledCols step %c1 {
    %tileB = cim.copyTile(%B, %k, %j)
    cim.write(%id, %tileB)
    scf.for %i = %c0 to %tiledRows step %c1 {
      %tileC = cim.copyTile(%C, %i, %j)
      ...
      cim.storeTile(%tileC, %C, %i, %j)
    }
  }
}
```

Fig. 4. Loop interchange to reduce the number of writes to the CIM crossbar during a tiled GEMM computation.

Once a kernel has been reduced to GEMM form, the device-aware `cim` and `memristor` abstractions apply concrete device-specific transformations and optimizations. Since a crossbar has fixed, bounded dimensions, a *tiling* pass splits GEMM operands that exceed the array size into smaller, array-sized tiles before copying them onto the accelerator's tiles. Similarly, since `write` operations to memristive devices are considerably more costly than reads and also wear the device, a *loop interchange* pass reorders the computation to reduce the total number of writes. Fig. 4 shows the transformed, tiled loop nest. In the original tiled version (top), the tile `tileB` is copied and written once per iteration of the innermost `k`-loop, i.e., once for every row tile `i`. After interchanging `i` and `k` (bottom), the copy and write operations instead happen once per `k`, significantly reducing the number of writes. To maximize parallelism across a device's parallel-computing CIM tiles, a *loop unrolling* pass unrolls the loop over crossbar tiles.

2) Compilation for CAMs: Building on CINM, C4CAM [37] specializes the framework's `cim` and `cam` abstractions to target content-addressable memories (CAMs), whose fundamental primitives are search-based operations (e.g., computing distances, similarities, or comparisons). The entry point remains the high-level domain-specific abstractions, e.g., in this case, the `torch-mlir` dialect which accepts TorchScript input and uses the conversion passes in CINM to enter the compilation flow. Once in the `cim` abstraction, a dedicated `cim-fuse-ops` pass performs pattern matching over the kernel to automatically detect sequences of operations that correspond to a supported similarity search – dot-product similarity, Euclidean distance, cosine similarity, and Hamming distance – and rewrites

each matched sequence into a single `cim.similarity` operation. Operations that do not match any supported pattern are left to execute on the host CPU.

In addition to the input application, the C4CAM flow also takes a CAM specification and an optimization objective as input. The `cim` and `cam` dialects use this information to apply device-specific optimizations and transformations to achieve the desired objective. For instance, based on the array dimensions, the `cim` abstraction applies a compulsory partitioning transformation whenever a kernel's operands exceed this capacity, splitting the computation into subarray-sized tiles. A `cim.merge_partial` operation, parameterized by both the type of partial result and the direction of accumulation, then recombines the per-tile results, initially within a single serial loop[2]. From this partitioned `cim` representation, a `cim-to-cam` conversion pass lowers the program to the `cam` dialect, which encodes the concrete hierarchy of a CAM accelerator – banks, mats, arrays, and subarrays – and the specific CAM technology being targeted (e.g., analog CAM, ternary CAM, or multi-bit CAM), together with its associated search type (exact, best, or threshold match) and distance metric. This conversion replaces generic `cim` operations (e.g., `cim.acquire`, `cim.execute`, etc.) with the fine-grained `cam` operations (e.g., `cam.alloc_bank`, `cam.alloc_mat`, `cam.alloc_array`, etc.). A separate mapping pass then maps the input operands to the CAM structure in a way that achieves the desired objective (e.g., reduce CAM arrays for power-cap, maximize parallelism for reduced execution time).

3) Compilation for Bulk-Bitwise Logic: This CIM paradigm is fundamentally different from both the crossbar and CAM accelerators described previously. Unlike these fixed-, single-function accelerators, bulk-bitwise CIM devices typically support fundamental logic operations which can be used to synthesize multiple bitwise arithmetic and logic operations. However, they impose additional constraints on data mapping and alignment within the memory array (operands must be column- or row-aligned). Additionally, the logic can be implemented in different ways (stateful vs. non-stateful, using volatile vs. non-volatile memories, etc.).

Sherlock [39] is a compilation and optimization framework for one specific class of CIM logic, namely scouting logic [50]. Unlike CINM and C4CAM, it is not built on the MLIR/CINM dialect stack; instead, it operates directly on a data-flow graph (DFG) extracted from a `C` input via `pycparser`, where nodes represent operands or intermediate results and edges represent operation dependencies.

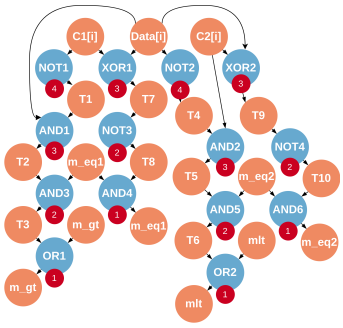


Fig. 5. DFG of a single iteration of the Bitweaving application [39].

Fig. 5 shows the DFG of a single iteration of the Bitweaving application. The mapper module in Sherlock takes this as input and greedily clusters DFG nodes into independent groups, where the size of each group is constrained by the column size (the array's per-column capacity). More specifically, by computing the b-level (priority, indicated in red circles in Fig. 5) of each node, the heuristic processes nodes in order of decreasing priority, i.e., a node with no predecessor seeds a new cluster, while a node with one or more already-clustered predecessors is assigned to whichever cluster minimizes cross-column dependencies. This assignment favors clusters with similar size and priority, closer priority proximity to the node (since this likely lies on that cluster's critical path), and, when dependencies are split evenly across clusters, the smaller cluster (to balance load). When these simple rules don't resolve the assignment, the algorithm falls back on a scoring function that combines cluster size and the summed priority differences to the node's predecessors in that cluster, assigning the node to the highest-scoring cluster. Once every node has been assigned, if the resulting number of clusters exceeds the target column count k, clusters with the strongest mutual dependencies are greedily merged (subject to a max-size constraint) until exactly k clusters remain, each of which is then mapped to one CIM array column. A final pass merging compatible instructions across clusters to reduce total instruction count and further reduces data movement. Sherlock also targets reliability optimization. Since combining multiple two-operand operations into fewer, wider multi-operand operations improves performance but increases the probability of decision failure, Sherlock explicitly co-optimizes this tradeoff rather than maximizing performance alone.

D. Design-Space Exploration

So far, we presented individual accelerators, each with a dedicated compilation flow that already supports some degree of design-space exploration (DSE) on its own. In C4CAM, for instance, CINM lets the user specify an optimization objective, and the framework finds a mapping and generates code that satisfies it – e.g., minimizing latency, power, or maximizing array utilization, as discussed in Sec. III-C. Similarly, in Sherlock, the compilation flow can be configured to optimize for performance, reliability, or to co-optimize both jointly, letting the user trade one off against the other depending on the target application's tolerance for error.

Beyond exploring the design space for a single accelerator, CINM is specifically designed to target heterogeneous systems, where an application's kernels may be distributed across multiple different CIM/CNM devices, or between such devices and the host. The `cinm` abstraction provides the necessary infrastructure to explore these trade-offs before making any offloading decision. This is, however, currently constrained by the unavailability of mature evaluation and modeling frameworks and tools for many of the devices CINM targets. In general, this abstraction provides the infrastructure to plug in device-specific cost and performance models, such as [48], [51], and to explore the design space in detail before committing to an offloading decision. For individual devices, this exploration includes finding the right tile sizes, loop transformations such as the interchange and unrolling discussed for crossbars, or fine-grained data-flow analysis to minimize both off-chip and on-chip data movement.

IV. Cross-Layer Techniques for Efficient and Scalable LLM Deployment

The rapid growth of large language models (LLMs) and generative AI has increased inference compute and memory demands, exposing three layers of LLM inference—memory, compute, and design optimization. General matrix multiply (GEMM) dominates inference, creating bottleneck across all three layers. At the *memory* layer, attention and projections are memory-bound due to repeated off-chip accesses to intermediate tokens and weights [52]. At the *compute* layer, weight-only quantization reduces memory traffic but not computation, since packed integer (INT) weights are unpacked

[2]Compared to other CIM accelerators, reduction in CAMs is non-trivial and depends on multiple factors including the search and merge types.

and dequantized to floating-point (FP) before GEMM execution [53]. At the *design optimization* layer, accelerator design space exploration (DSE) is challenging due to an exponentially large, non-convex, many-to-one, and non-differentiable search space [54]. Optimizing each layer in isolation cannot guarantee end-to-end inference benefits. We unify three works targeting these layers: **MEADOW** [52] introduces Token-Parallel Head-Sequential (TPHS) dataflow and lossless weight packing; **PacQ** [53] preserves packed weights through GEMM via co-designed dataflow and FP–INT multiplication; **DiffAxE** [54] enables rapid accelerator optimization through diffusion-based generative DSE.

A. MEADOW: Memory Layer Optimization

At the memory layer, transformer inference is dominated by off-chip data movement rather than computation. During prefill, repeated transfers of intermediate attention tensors dominate latency, while during decode, weight fetches become the primary overhead. MEADOW addresses both through a restructured dataflow and lossless weight compression. Fig. 6(a) illustrates the **Token-Parallel Head-Sequential (TPHS) dataflow**, which pipelines the memory-bound attention layers instead of executing independent general matrix multiply (GEMM) operations. TPHS processes input tokens (IP) in parallel while executing attention heads (H) sequentially, i.e., all tokens for head `H1` are completed before proceeding to `H2` and `H3`. This head-sequential ordering maximizes on-chip reuse of query (Q) weight, key (K) and value (V), avoiding repeated weight and KV reloads. The pipeline comprises the *Q*, *QKT*, *SM*, and *SMV* stages, each taking `T` cycles, while the sequential softmax (SM) stage is further pipelined into *MAX*, *EXP*, and *DIV* stages. After pipeline fill, the *Q*, *QKT*, *SM*, and *SMV* stages operate concurrently on different token groups, while intermediate $\mathtt{QK}^{\mathtt{T}}$ and `SM` values are forwarded directly between stages through on-chip registers instead of being spilled to DRAM. The remaining layers within the decoder block of LLM continue to execute as standard GEMMs.

To reduce weight-fetch latency, MEADOW uses a lossless **weight packing** pipeline (Fig. 6(a)) with a dedicated on-chip unpacking unit. MEADOW replaces repeated quantized weight chunks with IDs referencing a compact *Unique Matrix*, achieving 10^2–$10^3\times$ compression on OPT-2.7B and 6.7B.

MEADOW employs a tiled processing-element (PE) array with dedicated softmax, layer normalization, and activation units, combining parallel and broadcasting PEs to support both GEMM and TPHS execution. MEADOW achieves $\mathbf{2.5}\times$ and $\mathbf{1.5}\times$ lower prefill and decode latency, respectively, with over $\mathbf{40}\%$ lower end-to-end latency compared to prior works.

Although MEADOW reduces off-chip traffic through dataflow restructuring and weight packing, these benefits translate into latency savings only on hardware with register-level forwarding across pipeline stages, coupling the memory-layer optimization to the compute substrate discussed in Section II. MEADOW's Unique-Matrix encoding reduces weight volume, but unpacking before GEMM incurs additional latency, motivating the packed-format compute architecture discussed next.

B. PacQ: Compute Layer Optimization

While MEADOW addresses edge memory bottlenecks, cloud LLM inference remains compute-bound under multi-batching [53]. Weight-only quantization reduces memory footprint, but conventional single instruction multiple threads (SIMT) hardware unpacks and dequantizes INT weights to FP before GEMM, adding latency and losing compute benefits. PacQ preserves packed INT weights throughout for direct FP–INT multiplication, enabling hyper-asymmetric GEMM through a co-designed dataflow, parallel FP–INT multiplier, and fused dequantization datapath.

PacQ (Fig.6(b)) proposes a **parallel FP-INT multiplier** that directly consumes packed INT weights without dequantization, exploiting the FP16 representation of small integers (Fig. 6(b)). Any integer in $[1024, 2048)$ has a fixed exponent and a mantissa determined only by its offset from 1024. Thus, a signed INT4 weight $\mathtt{B} \in [-8, 7)$ is shifted to $\mathtt{B} + 1032$, yielding an FP16 mantissa with a fixed pattern and a 4-bit variable field $\mathtt{yyyy} = \mathtt{B} + 8$. This enables a single FP16 activation to multiply four packed INT4 (or eight INT2) weights per cycle. The multiplier combines activation mantissas with packed weight fields using a shared adder network to compute multiple products in parallel while reusing FP hidden-bit logic. This enables reusing 73% of a standard FP16 multiplier, achieving $3.38\times$ ($6.75\times$) higher throughput for INT4 (INT2) weights.

Since PacQ directly consumes packed INT weights, it packs along the output-feature (`n`) dimension to maximize activation reuse, unlike conventional channel (`k`)-dimension packing. Combined with an output-stationary schedule, this keeps packed weights in registers, reducing register-file accesses by 54.3%. PacQ modifies only the GEMM core with parallel FP–INT units, widened adder trees, and accumulators that fuse the $\mathtt{B}+1032$ shift into $\sum_k \mathtt{A}_k\mathtt{B}_k - 1032\sum_k \mathtt{A}_k$ before scaling (Fig. 6(b)). This requires no quantization changes and achieves iso-perplexity with W4A16. The weight packing and parallel computing strategies enable PacQ to achieve up to $\mathbf{1.99}\times$ speedup with $\mathbf{69}\%$ hardware reuse over hyper-asymmetric GEMM baseline.

PacQ's packed-format datapath complements MEADOW's memory-layer compression by enabling compressed weights to remain packed throughout GEMM execution rather than being unpacked beforehand. Co-designing these techniques allows their benefits to compound across the stack, rather than being realized independently. This dependency highlights the importance of cross-layer optimization as the compression benefits rely on datapath supporting direct computation on compressed representations.

C. DiffAxE: Rapid Design Optimization

While MEADOW and PacQ address memory and compute bottlenecks, efficient design space exploration (DSE) is required to identify optimal hardware configurations. A systolic array-based accelerator can be broadly defined with the following design parameters—array dimensions (`R`, `C`), buffer sizes (`IPSz`, `WTSz`, `OPSz`), DRAM bandwidth (`BW`), and loop order. Unfortunately, this creates an $O(10^{17})$ design space with many-to-one and non-differentiable hardware-performance mappings [55], prohibiting time-consuming exhaustive search techniques. Iterative black-box methods [56], [57] require costly simulator iterations, or approximate the performance using inaccurate differential surrogates [58] that lead to suboptimal designs. DiffAxE addresses search-space irregularity and long search times by casting DSE as a conditional generation problem. It first encodes the irregular hardware–performance space into a smooth latent space (**Phase-1**), followed by learning the latent space distribution with a diffusion model (**Phase-2**) (Fig. 6(c)). Given a workload w and target performance p, DiffAxE performs one-shot conditional generation of hardware (HW_{gen}), greatly reducing search time over iterative methods such as Bayesian optimization (BO).

During **Phase-1**, DiffAxE learns a **performance-structured latent space** (Fig. 6(c)). Each hardware design is encoded as a 7-D vector with loop-order embeddings and numerical parameters, then compressed by an autoencoder into latent vector v. A jointly trained performance predictor `PP` enforces performance awareness through the following loss: $\mathcal{L} = \|HW - \widehat{HW}\|^2 + \|p - \hat{p}\|^2$, leading to performance-aware clustering. This is essential for accurate hardware generation, as raw hardware (HW) lacks performance-aware semantic structure in the latent space (v) (Fig. 6(c)). **Phase-2** trains a conditional diffusion model over the latent space. Gaussian

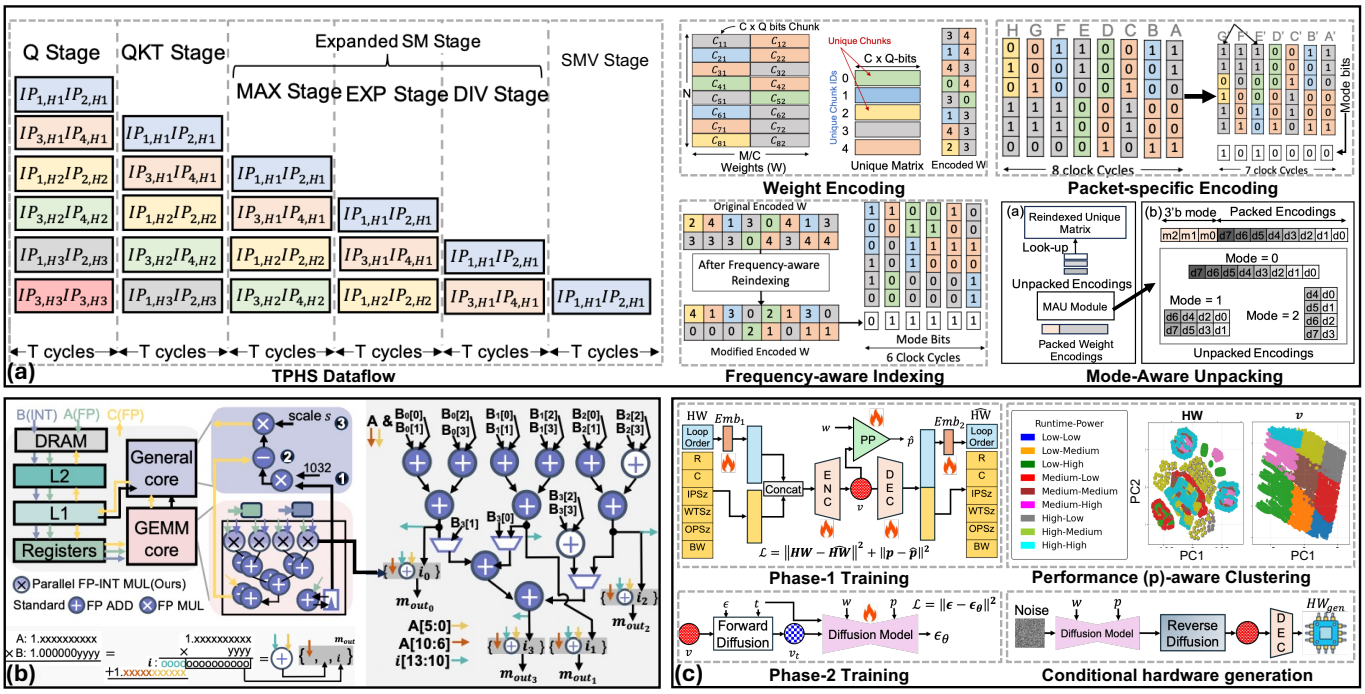


Fig. 6. Overview of the three works spanning the memory, compute, and design layers of the inference optimization stack.(a) MEADOW — the token-parallel head-sequential (TPHS) dataflow, with the lossless weight packing pipeline. (b) PacQ— the SIMT microarchitecture with the parallel FP-INT multiplier that produces four output mantissas m_{out} in a single cycle by reusing the standard FP16. (c) DiffAxE — phase-wise training to enable performance-aware latent-space clustering of hardware configurations, followed by workload (w) and performance-specific hardware generation.

noise is progressively added to latent v to obtain v_t, and the model learns to recover the noise ϵ_θ conditioned on timestep t, workload w, and target performance p using denoising loss $\|\epsilon - \epsilon_\theta\|^2$. Since conditioning requires no differentiable performance function, DiffAxE naturally handles non-differentiable DSE spaces.

During inference, DiffAxE performs **conditional hardware generation** (Fig. 6(c)) by reversing diffusion from noise given a target performance and workload. The decoder maps the generated latent vector to a valid hardware configuration (HW_{gen}). For EDP optimization, DiffAxE conditions on joint power–runtime signals, generating candidates across nine power–runtime classes and selecting the minimum-EDP design. Runtime optimization reuses the same generator by conditioning on the lowest EDP range (top 10%) and selecting the minimum-runtime design.

This dual-phase approach enables DiffAxE to achieve competitive hardware designs at a much lower search time over existing works. For GEMM workloads, DiffAxE finds designs with $\mathbf{9.8\%(10\%)}$ lower EDP (runtime) at $\mathbf{45.6\times(1312\times)}$ faster search time, while lowering LLM inference EDP by up to $\mathbf{7.75\times}$ using only $\mathbf{5.4M}$ trainable parameters. DiffAxE is well positioned to explore the resulting cross-layer design space, jointly encompassing dataflow, packed-datapath width, and buffer sizing, rather than treating accelerator microarchitecture independently of the dataflow and precision choices in Sections IV-B and IV-C. In future, DiffAxE can be extended to co-optimize hardware configurations conditioned on TPHS-style dataflows and packed-format GEMM configurations, enabling accelerator exploration jointly informed with these architectural choices.

V. Token Merging and Clustering

While the cross-layer techniques discussed in Section IV establish the macro-level co-design infrastructure—optimizing execution across compilers, memory hierarchies, and hardware accelerators—the computational footprint of Large Language Models (LLMs) remains fundamentally bound by the quadratic complexity of the self-attention mechanism relative to sequence length. The present section transitions from hardware-stack integration to active context reduction, presenting token merging and clustering as powerful algorithmic methods that prune redundant visual or textual tokens directly within the attention layers to maximize throughput.

Computational and memory costs of self-attention are significant bottlenecks in transformer architectures: for N tokens, it evaluates $O(N^2)$ pairwise interactions and, when the attention matrix is materialized, uses $O(N^2)$ storage [59], [60]. Reducing N therefore lowers computation and memory traffic. Spatial and semantic redundancy often enables this reduction with little loss of representation quality. These savings complement the preceding layers: smaller tensors reduce pressure on memory-centric hardware (Section II), allow the compiler to map lower-dimensional operations with fewer tiles and transfers (Section III), and reduce the work addressed by the compression and dataflow optimizations in Section IV. Realizing these benefits requires mappings and kernels that preserve the reduced dimensions rather than restore dense intermediates. Two common approaches are *token merging*, which performs local pairwise aggregation, and *token clustering*, which constructs representatives from a global partition.

A. From Local Merging to Global Clustering

Token merging techniques replace a set of similar tokens with one or more representatives. A prominent merging method is ToMe [61], which partitions tokens into two sets, connecting each token in one set to its most similar token in the other, and selecting the r highest-scoring source-to-destination matches. Each selected source token is then aggregated into its matched destination token, while a *size* vector tracks how many original tokens each merged token represents.

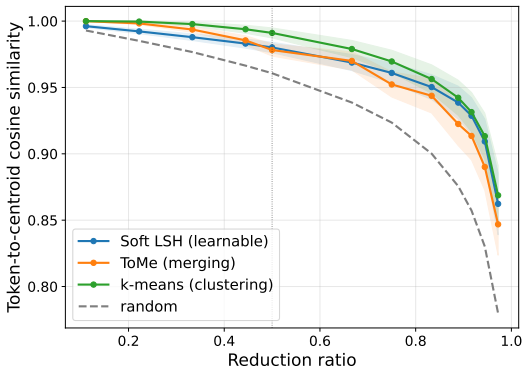


Fig. 7. Mean token-to-representative similarity for LLaVA visual tokens. Under aggressive reduction, global clustering preserves higher cosine similarity compared to ToMe.

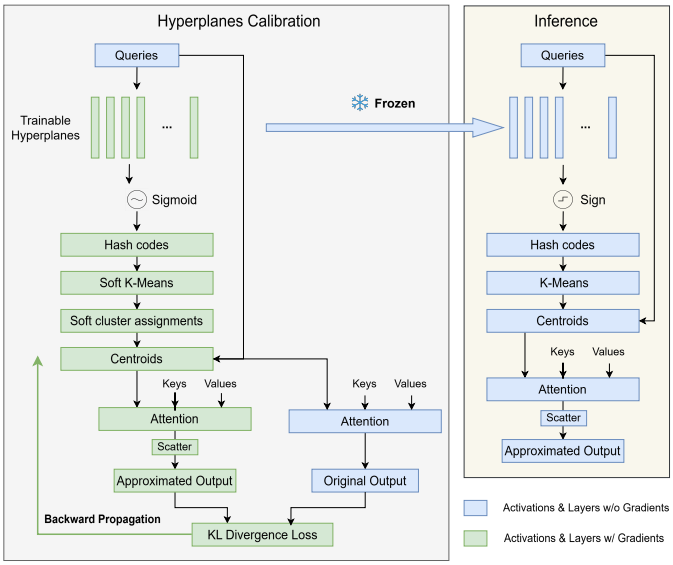


Fig. 8. Differentiable hyperplane calibration (left) and inference with frozen hyperplanes and hard assignments (right) [68]

This method requires no iterative optimization and introduces no learnable parameters, utilizing prescribed schedules that facilitate deployment in ViTs and diffusion models [62]. Related methods reduce visual context in vision–language models [63], and 3D–reconstruction models [64], among others.

Merging selection strategies are, however, local and greedy; as the reduction ratio increases, the resulting representatives do not necessarily minimize a global distortion objective. Furthermore, since neither the matching criterion nor schedule is learned from task-specific data, standard merging lacks explicit learned adaptation to task-specific activation distributions. Clustering, instead, optimizes token assignments and representatives under a global distortion objective. The k-means objective minimizes within-cluster distortion. When k centroids replace N queries, the query–key interaction cost decreases from $O(N^2)$ to $O(kN)$ [65]. Fig. 7 reports the mean cosine similarity between LLaVA visual tokens and their assigned representatives [66]. While both methods achieve similar performance at low compression ratios, k-means clustering gradually outperforms ToMe, widening the performance gap as the reduction ratio increases.

A drawback of global clustering is that the clustering step requires repeated assignment–update iterations and is less amenable to fused attention kernels. *SimHash* [67] proposes a locality-sensitive hashing approach which provides a lower-cost similarity approximation. SimHash maps each token to an m-bit code using random hyperplanes. If D is the Hamming distance between two codes, the corresponding cosine similarity satisfies $\cos(\pi\mathbb{E}[D]/m)$. While m-bit code generation costs $O(Nm)$ for fixed m, total cost still depends on subsequent grouping. Data-independent hyperplanes may nevertheless be poorly aligned with the model's activation distribution.

B. A Learnable Token Clustering Layer

To improve hyperplane alignment, we optimize the hashing hyperplanes using the differentiable surrogate in Fig. 8. This implementation is derived from the block-wise clustered attention technique introduced in [68]. During calibration, a sigmoid replaces the non-differentiable sign function, and soft-assignment k-means replaces hard grouping [69], [70]. A KL-divergence objective between the approximate and reference attention outputs provides gradients to the hyperplanes. At inference, the hard hashing and clustering operations are restored and the calibrated parameters are frozen. The surrogate therefore changes the learned partition without introducing parameter updates or surrogate operations during deployment.

In the evaluated configuration, the clustering layer introduces only a few thousand parameters, which can be optimized on a small calibration set. Alternatively, the differentiable clustering layer can be incorporated into end-to-end fine-tuning, specializing the grouping rule to a downstream objective while either retaining or jointly updating the pretrained backbone.

TABLE I
LLaVA-1.5-7B AFTER REDUCING 576 VISUAL TOKENS TO 64 AT LAYER 2. POPE IS MACRO F1; † DENOTES REPORTED BASELINES.

Method	Tokens	GQA ↑	ChartQA ↑	OCRBench ↑	POPE ↑
LLaVA-1.5-7B	576	62.0†	14.0	301	85.9†
+ DART	64	54.8	**11.4**	**271**	74.0
+ clustering layer	**64**	**57.7**	10.8	225	**81.7**

Table I reports an end-to-end evaluation with LLaVA-1.5-7B [71]. We compare our results with DART [72], a similarity-based token pruning technique. Both reduction methods retain 64 of 576 visual tokens after decoder layer 2 and leave the pretrained backbone unchanged. At this 88.9% reduction, clustering exceeds DART by 2.9 points on GQA [73] and 7.7 points on POPE macro F1 [74]. DART achieves higher scores on ChartQA [75] and OCRBench [76] by 0.6 and 46 points, respectively.

The task-level results further characterize this difference. Relative to DART, clustering improves GQA accuracy on attribute (63.6 vs. 59.6), object (83.4 vs. 80.7), and relation (50.2 vs. 47.5) questions. On POPE, it increases accuracy from 78.7 to 83.2 and recall from 60.1 to 73.9, while DART attains higher precision (96.3 vs. 91.3). The higher precision and lower recall of DART are consistent with a more conservative prediction pattern. Conversely, the ChartQA and OCRBench results indicate that the present clustering configuration is less effective for text- and detail-sensitive inputs; establishing the cause requires targeted ablations on token selection and assignment.

These results suggest that learnable clustering may balance representative quality and grouping overhead, although direct latency and memory measurements are still required. Future work should examine content-adaptive cluster counts, joint backbone optimization, modality-aware objectives, and transfer across vision, language, diffusion, and multimodal architectures.

VI. Conclusion and Outlook

Generative AI is constrained less by ideas than by the cost of running them. This paper has argued that the resulting sustainability barrier is best approached across the layers of the computing stack rather than from any single one in isolation, and has drawn together four complementary perspectives on how to do so. Energy-efficient heterogeneous 2.5D/3D and in-memory hardware (Section II) provides the physical substrate; MLIR-based compiler infrastructure (Section III) makes that substrate programmable and explorable; quantization, sparsification, and dataflow co-design (Section IV) shrink the work the hardware must perform; and token merging and clustering (Section V) attack the quadratic cost of attention at the model level. These techniques are complementary, and we expect that applying them together across the stack can compound their individual gains rather than merely adding them.

Several open challenges remain. Cross-layer co-optimization still lacks common abstractions and benchmarks, making end-to-end gains difficult to attribute and reproduce. Emerging in-memory devices introduce non-idealities and reliability concerns, while multimodal models and on-device fine-tuning continue to raise the efficiency bar. The barriers are also organizational: end-to-end co-design requires model developers, compiler teams, hardware vendors, and system integrators to align their incentives, interfaces, and development cycles. Open abstractions, shared benchmarks, and viable partnerships may therefore be as important as further technical advances. We hope this cross-stack synthesis helps the community address these challenges jointly and advances affordable, scalable generative intelligence from the cloud to the edge.

References

[1] S. Rai *et al.*, "Accelerating automated driving and adas using hw/sw codesign," in *2024 IEEE 37th International System-on-Chip Conference (SOCC)*, 2024, pp. 1–6.

[2] Y. Lin *et al.*, "Qserve: W4a8kv4 quantization and system co-design for efficient llm serving," *Proceedings of Machine Learning and Systems*, vol. 7, 2025.

[3] M. Khairy *et al.*, "A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity," *Journal of Parallel and Distributed Computing*, vol. 127, pp. 65–88, 2019.

[4] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annual International Symposium on Computer Architecture (ISCA)*. Toronto, ON, Canada: ACM, 2017, pp. 1–12.

[5] Y. Zhong *et al.*, "DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving," in *Proc. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024, pp. 193–210.

[6] E. J. Hu *et al.*, "LoRA: Low-rank adaptation of large language models," in *Proc. International Conference on Learning Representations (ICLR)*, 2022.

[7] T. Shiokawa *et al.*, "High-endurance (>10^{11} cycles) and thermally-stable sub-100nm TiO_2 channel FeFET for low-power memory centric 3d-LSI applications," in *Proc. IEEE Electron Devices Technology & Manufacturing Conference (EDTM)*. Seoul, Republic of Korea: IEEE, 2023, pp. 1–3.

[8] S. Yu *et al.*, "Compute-in-memory chips for deep learning: Recent trends and prospects," *IEEE Circuits and Systems Magazine*, vol. 21, no. 3, pp. 31–56, 2021.

[9] M.-F. Chen *et al.*, "System on integrated chips (SoIC) for 3d heterogeneous integration," in *Proc. IEEE 69th Electronic Components and Technology Conference (ECTC)*. Las Vegas, NV, USA: IEEE, 2019, pp. 594–599.

[10] A. Goda, "3-d NAND technology achievements and future scaling perspectives," *IEEE Transactions on Electron Devices*, vol. 67, no. 4, pp. 1373–1381, 2020.

[11] P. Dhingra *et al.*, "Atleus: Accelerating transformers on the edge enabled by 3d heterogeneous manycore architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 8, pp. 2842–2855, 2025.

[12] Y. Luo *et al.*, "H3D-Transformer: A heterogeneous 3d (H3D) computing platform for transformer model acceleration on edge devices," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 3, pp. 1–19, 2024.

[13] P. Dhingra *et al.*, "Focus session: Hardware/software co-design to accelerate generative ai workloads on heterogeneous architectures," in *2026 Design, Automation Test in Europe Conference (DATE)*, 2026, pp. 1–7.

[14] Y. Pan *et al.*, "Stratum: System-hardware co-design with tiered monolithic 3d-stackable DRAM for efficient MoE serving," in *Proc. 58th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Seoul, Republic of Korea: ACM, 2025, pp. 1–17.

[15] W.-H. Huang *et al.*, "A3D-MoE: Acceleration of large language models with mixture of experts via 3d heterogeneous integration," arXiv preprint arXiv:2507.19142, 2025.

[16] Y. Chen *et al.*, "CHIME: Chiplet-based heterogeneous near-memory acceleration for edge multimodal LLM inference," in *Proc. Design, Automation & Test in Europe Conference (DATE)*. Verona, Italy: IEEE, 2026, pp. 1–7.

[17] P. Dhingra *et al.*, "Thame: 3d memory-enabled heterogeneous accelerator for llm mixture of experts," 2026. [Online]. Available: https://arxiv.org/abs/2607.17074

[18] Y. H. Chen *et al.*, "Ultra high density SoIC with sub-micron bond pitch," in *Proc. IEEE 70th Electronic Components and Technology Conference (ECTC)*. Orlando, FL, USA: IEEE, 2020, pp. 576–581.

[19] D. Das Sharma *et al.*, "Universal chiplet interconnect express (UCIe): An open industry standard for innovations with chiplets at package level," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 12, no. 9, pp. 1423–1431, 2022.

[20] J. Park *et al.*, "AttAcc! unleashing the power of PIM for batched transformer-based generative model inference," in *Proc. 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. La Jolla, CA, USA: ACM, 2024, pp. 103–119.

[21] W. Shim *et al.*, "Ferroelectric field-effect transistor-based 3-d NAND architecture for energy-efficient on-chip training accelerator," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 7, no. 1, pp. 1–9, 2021.

[22] B. K. Joardar *et al.*, "Learning-based application-agnostic 3d NoC design for heterogeneous manycore systems," *IEEE Transactions on Computers*, vol. 68, no. 6, pp. 852–866, 2019.

[23] S. R. Ohring *et al.*, "On generalized fat trees," in *Proc. 9th International Parallel Processing Symposium*. Santa Barbara, CA, USA: IEEE Comput. Soc. Press, 1995, pp. 37–44.

[24] S. Bandyopadhyay *et al.*, "A simulated annealing-based multiobjective optimization algorithm: AMOSA," *IEEE Transactions on Evolutionary Computation*, vol. 12, no. 3, pp. 269–283, 2008.

[25] N. Hajinazar *et al.*, "Simdram: A framework for bit-serial simd processing using dram," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 329–345.

[26] X. Peng *et al.*, "Chopper: A compiler infrastructure for programmable bit-serial simd processing using memory in dram," in *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1275–1288.

[27] K. Vadivel *et al.*, "Tdo-cim: transparent detection and offloading for computation in-memory," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1602–1605.

[28] A. Siemieniuk *et al.*, "Occ: An automated end-to-end machine learning optimizing compiler for computing-in-memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[29] J. Han *et al.*, "Polyhedral-based compilation framework for in-memory neural network accelerators," *J. Emerg. Technol. Comput. Syst.*, vol. 18, no. 1, sep 2021. [Online]. Available: https://doi.org/10.1145/3469847

[30] X. Sun *et al.*, "Pimcomp: an end-to-end dnn compiler for processing-in-memory accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 5, pp. 1745–1759, 2024.

[31] S. Y. Kim *et al.*, "Optimal model partitioning with low-overhead profiling on the pim-based platform for deep learning inference," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 2, pp. 1–22, 2024.

[32] J. Lin *et al.*, "Songc: A compiler for hybrid near-memory and in-memory many-core architecture," *IEEE Transactions on Computers*, vol. 73, no. 10, pp. 2420–2433, 2024.

[33] S. Qu *et al.*, "Cim-mlc: A multi-level compilation stack for computing-in-memory accelerators," in *Proceedings of the 29th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2024, pp. 185–200.

[34] J. Cubero-Cascante *et al.*, "Cimflow: Modelling dataflow in cross-layer compute-in-memory deep learning accelerators," *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 5s, Sep. 2025. [Online]. Available: https://doi.org/10.1145/3760780

[35] J. Liu *et al.*, "Optipim: Optimizing processing-in-memory acceleration using integer linear programming," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 867–883. [Online]. Available: https://doi.org/10.1145/3695053.3731041

[36] Y. Shin *et al.*, "ATiM: Autotuning Tensor Programs for Processing-in-DRAM," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. Tokyo Japan: ACM, Jun. 2025, pp. 899–915.

[37] H. Farzaneh *et al.*, "C4cam: A compiler for cam-based in-memory accelerators," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2024. New York, NY, USA: Association for Computing Machinery, 2024.

[38] K. Kourtis *et al.*, "Compiling neural networks for a computational memory accelerator," *CoRR*, vol. abs/2003.04293, 2020. [Online]. Available: https://arxiv.org/abs/2003.04293

[39] H. Farzaneh *et al.*, "Sherlock: Scheduling efficient and reliable bulk bitwise operations in nvms," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3658485

[40] Y. Bai *et al.*, "A compilation framework for sram computing-in-memory systems with optimized weight mapping and error correction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 8, pp. 2379–2392, 2024.

[41] J. Chen *et al.*, "Simplepim: A software framework for productive and efficient processing-in-memory," in *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2023, pp. 99–111. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/PACT58117.2023.00017

[42] G. F. Oliveira *et al.*, "Dappa: A data-parallel programming framework for processing-in-memory architectures," 2025.

[43] P. Yang *et al.*, "A tensor compiler for processing-in-memory architectures," 2025. [Online]. Available: https://arxiv.org/abs/2511.15503

[44] C. Li *et al.*, "Pim-dl: Expanding the applicability of commodity dram-pims for deep learning via algorithm-system co-optimization," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 879–896.

[45] T. Xie *et al.*, "Unindp: A unified compilation and simulation tool for near dram processing architectures," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 624–640.

[46] A. A. Khan *et al.*, "CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. Hilton La Jolla Torrey Pines La Jolla CA USA: ACM, Apr. 2024, pp. 31–46.

[47] C. Lattner *et al.*, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 2–14.

[48] H. Farzaneh *et al.*, "CoMoNM: A cost modeling framework for compute-near-memory systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, Sep. 2026.

[49] J. Gómez-Luna *et al.*, "Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system," *IEEE Access*, vol. 10, pp. 52 565–52 608, 2022.

[50] L. Xie *et al.*, "Scouting logic: A novel memristor-based logic design for resistive computing," in *ISVLSI*. IEEE, 2017, pp. 176–181.

[51] J. Rogers *et al.*, "Aio: An abstraction for performance analysis across diverse accelerator architectures," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 487–500.

[52] A. Moitra *et al.*, "Meadow: Memory-efficient dataflow and data packing for low power edge llms," *Proceedings of Machine Learning and Systems*, vol. 7, 2025.

[53] R. Yin *et al.*, "Pacq: A simt microarchitecture for efficient dataflow in hyper-asymmetric gemms," in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2025, pp. 1–7.

[54] A. Ghosh *et al.*, "Diffaxe: Diffusion-driven hardware accelerator generation and design space exploration," 2026. [Online]. Available: https://arxiv.org/abs/2508.10303v2

[55] J. Seo *et al.*, "Airchitect v2: Learning the hardware accelerator design space through unified representations," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.

[56] B. Reagen *et al.*, "A case for efficient accelerator design space exploration via bayesian optimization," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2017, pp. 1–6.

[57] Q. Huang *et al.*, "Learning a continuous and reconstructible latent space for hardware accelerator design," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2022, pp. 277–287.

[58] L. Feng *et al.*, "Gandse: Generative adversarial network-based design space exploration for neural network accelerator design," *ACM Transactions on Design Automation of Electronic Systems*, vol. 28, no. 3, pp. 1–20, 2023.

[59] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[60] A. Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations (ICLR)*, 2021.

[61] D. Bolya *et al.*, "Token merging: Your ViT but faster," in *International Conference on Learning Representations (ICLR)*, 2023.

[62] ——, "Token merging for fast stable diffusion," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2023.

[63] Y. Shang *et al.*, "LLaVA-PruMerge: Adaptive token reduction for efficient large multimodal models," *arXiv preprint arXiv:2403.15388*, 2024.

[64] W. Wang *et al.*, "Httm: Head-wise temporal token merging for faster vggt," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2026, pp. 26 379–26 388.

[65] A. Vyas *et al.*, "Fast transformers with clustered attention," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[66] H. Liu *et al.*, "Visual instruction tuning," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[67] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, 2002, pp. 380–388.

[68] W. Wang *et al.*, "Hardware-aware calibrated clustered attention for efficient visual geometric transformers," in *International Joint Conference on Neural Networks (IJCNN)*, 2026.

[69] J. Xie *et al.*, "Unsupervised deep embedding for clustering analysis," in *International Conference on Machine Learning (ICML)*, 2016, pp. 478–487.

[70] Z. Cao *et al.*, "HashNet: Deep learning to hash by continuation," in *IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 5608–5617.

[71] H. Liu *et al.*, "Improved baselines with visual instruction tuning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024, pp. 26 296–26 306.

[72] Z. Wen *et al.*, "Stop looking for "important tokens" in multimodal language models: Duplication matters more," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2025, pp. 9961–9980.

[73] D. A. Hudson *et al.*, "GQA: A new dataset for real-world visual reasoning and compositional question answering," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 6700–6709.

[74] Y. Li *et al.*, "Evaluating object hallucination in large vision-language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2023, pp. 292–305.

[75] A. Masry *et al.*, "ChartQA: A benchmark for question answering about charts with visual and logical reasoning," in *Findings of the Association for Computational Linguistics: ACL 2022*. Association for Computational Linguistics, 2022, pp. 2263–2279.

[76] Y. Liu *et al.*, "OCRBench: On the hidden mystery of OCR in large multimodal models," *Science China Information Sciences*, vol. 67, no. 12, 2024.