

TranSAGE: Automating Spatial Accelerator Generation for Large-Scale Transformer Models

Chi Zhang
Shanghai Jiao Tong University
Shanghai, China
zhang-chi@sjtu.edu.cn

Yi Sun
School of Microelectronics, South
China University of Technology
Guangzhou, China
misunyi@mail.scut.edu.cn

Weichuang Zhang
Shanghai Jiao Tong University
Shanghai, China
1064080006@sjtu.edu.cn

Quan Chen
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Jieru Zhao*
Shanghai Jiao Tong University
Shanghai, China
zhao-jieru@sjtu.edu.cn

Abstract

Large-scale Transformers have achieved remarkable success across NLP, computer vision, and robotics, but their growing computational and memory demands challenge efficient inference. Spatial accelerators offer advantages in on-chip bandwidth and energy efficiency, yet designing them for large models remains costly and time-consuming. Existing accelerator generators typically focus on single-operation support, overlooking the benefits of multi-operation pipelining, which limits throughput and energy efficiency.

We propose TranSAGE, a compiler-based framework that jointly optimizes scheduling for large-scale transformers under multi-operation pipelining and generates corresponding accelerator implementations. Our approach integrates a DSE engine with multi-operation dataflow optimization and a dynamic programming-based graph partitioning algorithm to efficiently identify pipeline-friendly operation clusters and their optimal dataflow parameters. We further design spatial architecture primitives and lowering passes to automatically generate efficient RTL implementations.

Evaluations on four representative LLMs demonstrate that TranSAGE achieves up to $2.41\times$ higher throughput and $13.04\times$ higher energy efficiency over commercial GPU/NPU baselines. Compared with state-of-the-art accelerator generators, TranSAGE achieves up to $11.33\times$ higher throughput and $6.07\times$ higher energy efficiency. Our code is available at <https://github.com/sjtu-zhao-lab/TranSAGE.git>.

Keywords: Transformer Accelerators, Spatial Architectures, Accelerator Generation, Compiler Optimization, Dataflow Optimization

1 Introduction

Large-scale transformers, including large language models (LLMs), Diffusion Transformers (DiTs), vision-language models (VLMs), and vision-language-action (VLA) models, have

demonstrated remarkable capabilities across a broad spectrum of applications, such as conversational agents [10, 24, 41, 49], multimedia content generation [17, 18, 38, 42], multimodal understanding [27, 40], and robotics [6, 12]. However, the continued growth in model scale increases the computational and memory demands of these models, making low-latency and energy-efficient inference increasingly challenging. These challenges are further amplified by inference-time scaling techniques, such as Chain-of-Thought (CoT) reasoning [52] and agent-based methods [45, 56].

The challenges have driven the need for architectural innovations. Although GPUs remain the most widely adopted platform for deploying such models, they suffer from limited energy efficiency and suboptimal throughput. Recent research has increasingly focused on spatial accelerators to improve the efficiency of large-scale transformer inference [2, 9, 13, 23, 28, 50]. Unlike conventional architectures that heavily rely on deep memory hierarchies and temporal locality—often leading to bandwidth bottlenecks—spatial accelerators emphasize distributed memory and localized point-to-point communication. This design significantly enhances on-chip bandwidth and aligns well with the computational characteristics of transformer models. In particular, tensor operations in transformers exhibit high spatial data reuse with minimal need for global synchronization, making them especially well-suited for spatial execution paradigms.

However, maximizing the performance of large-scale transformer models on spatial accelerators requires substantial expertise in both hardware and software and is highly time-consuming, thereby hindering their broader adoption. Numerous mapping and compiler frameworks have been proposed to identify optimal mappings of large-scale neural networks onto spatial accelerators [5, 7, 15, 19, 25, 26, 30, 32, 37, 53, 57, 59, 60]. Although these frameworks report promising results, they primarily target predefined architectures with fixed or limited configurability, and thus lack the ability to jointly explore optimizations across both mapping strategies and architectural designs.

*Corresponding author

Table 1. Comparison with representative spatial accelerator automation frameworks.

Work	Productivity		Dataflow Support		Architecture Support		
	Input	Auto-Tuning Support	Adaptive Dataflow	Inter-Op Dataflow	PU	Intra-PU Topology	Inter-PU Topology
AutoSA [51]	C Code	✓	✗	Sequential	SA ² Only	Fixed	/
TensorLib [22]	STT ¹	✗	✗	Sequential	SA ² Only	Fixed	/
Gemmini [14]	C Code	✗	✗	Limited Fusion	SA ² + PPU ³	Fixed	Fixed
LEGO [29]	Relation Matrix	✗	✓	Limited Fusion	SA ² + PPU ³	Reconfigurable	Fixed
TranSAGE (Ours)	PyTorch Code	✓	✓	Tile-based Pipeline	Reconf. PU	Reconfigurable	Flexible

¹ Spatial-Temporal Transformation² Systolic Array³ Post Processing Unit

In contrast, a growing body of automated frameworks has emerged to generate optimized architectures and scheduling strategies for target workloads. We summarize these approaches in Table 1. AutoSA [51] proposes a polyhedral-based framework that automatically transforms C code into optimized HLS implementations for constructing systolic arrays. TensorLib [22] introduces a novel mathematical representation of spatial-temporal transformations (STT), and generates input, output, and interconnect components based on analysis of the STT matrix using Chisel [4] templates. However, they are primarily designed for systolic arrays, which can limit their efficiency on diverse operations in large-scale transformer models.

Gemmini [14] proposes a system-integrated, template-based generator that employs a systolic array as the core matrix processing unit, augmented with post-processing units to support a broader range of operations. LEGO [29] further extends this architecture by introducing adaptive dataflows on systolic arrays to mitigate the low utilization caused by rigid dataflow mappings. Although they target compound workloads, the generated accelerators may still incur high memory access overhead and achieve limited throughput. This limitation primarily arises from their support for only fixed-pattern fusion (e.g., GEMM combined with simple element-wise operations), which restricts their ability to fully exploit cross-operator optimization opportunities—such as operation fusion and pipelining—that have been widely demonstrated to be effective [3, 7, 8, 46, 60, 61].

Main idea. The root cause of the inefficiency in the aforementioned generators is that they treat individual operations as the basic execution unit and generate architecture for supporting them. In contrast, our key idea is to elevate clusters to first-class citizens rather than individual operations. Specifically, we fuse operations that are well-suited for parallel execution into clusters, which are then generated as hardware pipelines. The execution of the entire model is thus decomposed into the sequential execution of multiple clusters. Meanwhile, all clusters share and reuse the same on-chip compute and memory resources.

Challenge. However, such a design is challenging due to two aspects. (1) *Difficulty in identifying optimal dataflow:* For

large-scale transformer models, which typically consist of dozens of operations, determining which operations should be fused into a pipeline cluster is non-trivial. Furthermore, even after fusion, it remains challenging to assign dataflow parameters (e.g., tiling sizes, unrolling factors, and execution ordering) for each operation within a complex pipeline, making it difficult to identify the optimal dataflow. We elaborate on how these factors expand the design space and introduce potential pitfalls in Section 2.3. (2) *Difficulty in generating flexible architectures:* Even after determining the optimal dataflow for compound operations, generating corresponding hardware architectures remains challenging. It is impractical to implement architectures for every pipeline configuration, whether manually or using template-based methods. Moreover, all clusters must be mapped onto constrained hardware resources while minimizing overhead, further complicating the design.

Our solution. To address these challenges, we propose TranSAGE, an MLIR-based compiler framework that automatically extracts optimal operation clusters with corresponding dataflow parameters for large-scale transformer models, and generates the associated RTL implementations.

We introduce a design space exploration (DSE) engine that formulates multi-operation pipeline scheduling as two subproblems: graph partitioning and dataflow optimization. For graph partitioning, TranSAGE employs a dynamic programming-based method to determine the optimal operation fusion under a given topological order (Sec. 4.1). For each cluster, TranSAGE searches for the optimal dataflow parameters of its operators to form a pipelined cluster, using a decomposition-based approach (Sec. 4.2).

Furthermore, we propose a set of spatial architecture primitives (SAPs) that abstract arithmetic, data storage, and routing components commonly found in spatial accelerators. These primitives serve as a bridge between high-level representations (e.g., *MLIR linalg*) and RTL implementations. Based on dependency and data-sharing analysis, high-level *linalg* operations are lowered to SAPs and subsequently mapped to corresponding RTL implementations (Sec. 5).

Our main contributions are summarized as follows:

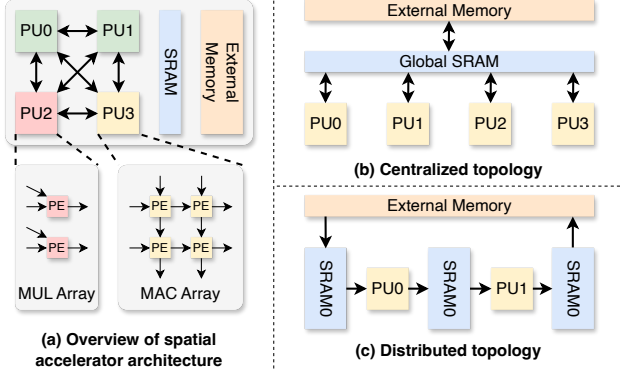


Figure 1. Spatial accelerator architecture.

- We propose TranSAGE, an end-to-end framework that automatically generates RTL implementations of spatial accelerators from PyTorch programs.
- We integrate a DSE engine that automatically extracts optimal operation fusion and dataflow parameters, enabling efficient mapping to hardware pipelines for maximum performance.
- We propose a set of abstractions for spatial accelerator hardware components, along with corresponding lowering passes, which serve as a bridge between high-level IR and RTL implementations.

Evaluation shows that TranSAGE delivers up to $2.42\times$ higher throughput and $13.04\times$ greater energy efficiency than commercial GPU/NPU baselines. Against state-of-the-art accelerator generators, TranSAGE further improves throughput by up to $11.33\times$ and energy efficiency by up to $6.07\times$.

2 Background and Motivation

2.1 Spatial Accelerators

Spatial accelerators represent a broad class of architectures characterized by two key features: (1) *distributed compute and memory resources* and (2) *asynchronous communication patterns*. These characteristics manifest across multiple levels of the system hierarchy. At the processing element (PE) level, designs such as Eyeriss [9] and TPU [23] organize PEs with local register files and localized interconnects (e.g., systolic arrays), enabling data reuse across neighboring PEs while reducing the overhead associated with global synchronization. At higher levels, such as the core or package level, systems like Simba [44] and Cerebras [1] preserve these principles while scaling out through network-on-chip (NoC) or network-on-package (NoP) interconnects.

TranSAGE Scope. The architectural scope considered in TranSAGE is illustrated in Fig. 1 (a). Within this scope, we focus on a single compute core equipped with on-chip compute resources and a hierarchical memory system. As discussed earlier, this design can be further scaled through inter-core or inter-package interconnects. The design space of spatial

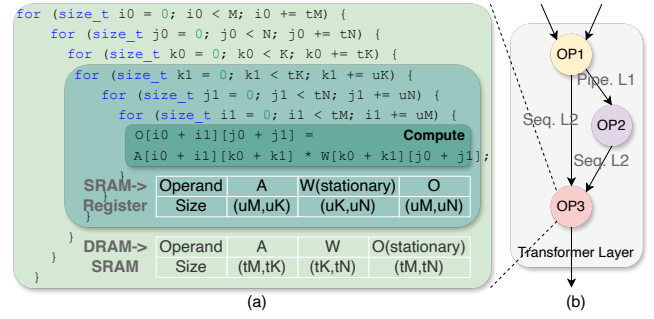


Figure 2. Representation for (a) intra-operation dataflows and (b) inter-operation dataflows. Seq. and Pipe. denote sequential and pipelined execution, and L1 and L2 denote data transfers through SRAM and DRAM, respectively.

accelerators within this scope consists of two primary dimensions. (1) *Organization of PEs*. For each type of compute resources (e.g., multipliers, adders, and nonlinear arithmetic units), we must determine the size of the PE array and how PEs are connected to local registers and to each other. (2) *Interconnect of PUs*. Furthermore, we need to define the topology of processing units (PUs). One option is a centralized topology, as shown in Fig. 1(b), where all PUs are connected to a global SRAM, enabling flexible data exchange. Alternatively, specialized PU topologies can be employed, as illustrated in Fig. 1(c), where connectivity between PUs is restricted to reduce communication overhead.

2.2 Dataflow and Representation

Dataflow refers to the scheduling strategy that describes how tensor operations are temporally and spatially mapped onto a spatial accelerator. It typically specifies the size and order of data movement between components, both within the same hierarchy level and across different hierarchy levels. In addition, it characterizes how computations are executed, whether in parallel or sequentially.

Intra-operation dataflow and representation. In the early research stage, tensor operations such as matrix multiplication and convolution are executed sequentially on spatial accelerators. In this setting, it is sufficient to represent the dataflow of each operation independently, which we refer to as intra-operation dataflow. The most commonly used method for representing intra-operation dataflow is a nested loop with annotations. Fig. 2(a) shows an example of the dataflow for a matrix multiplication operation on a spatial accelerator with a two-level memory hierarchy.

As illustrated, the six-level nested loop is derived from the original three-level nested loops over the M , N , and K dimensions. Specifically, for data movement across memory hierarchy, the representation explicitly captures the data size through tiling along each dimension. For example, the tiling at the first three levels represents data transfer between

DRAM and SRAM with a size of (tM, tN, tK) , while the rest levels represent data transfer between SRAM and registers with a size of (uM, uN, uK) . We refer to these as the **tiling vector** and **unroll vector**, respectively.

Furthermore, within each hierarchy, the ordering of the three dimensions is a permutation of the original (M, N, K) order, which determines the access pattern of each tensor. For instance, an order of (M, N, K) at the first three levels implies that $A(m, k)$ and $W(k, n)$ are accessed in a strided manner, while $O(m, n)$ is accessed contiguously, indicating that O is stationary in SRAM. Similarly, at the rest levels, $W(k, n)$ exhibits a stationary pattern in the register file. We encode these permutations as index vectors relative to the original order, referred to as the **outer-order vector** and **inner-order vector** for the first three and rest levels, respectively.

Inter-operation dataflow and representation. As techniques such as layer fusion and pipeline execution have demonstrated advantages in reducing external memory access and improving PE utilization [11, 33, 34, 54, 58], increasing attention has been devoted to representing dataflows that span multiple operations, i.e., inter-operation dataflow. For example, methods in [7, 60] adopt tree-based representations with annotations to describe how resources are temporally and spatially allocated across operations. In contrast, the table-based method in [15] provides a more formal mathematical abstraction. Despite differences in representation, the key objective of inter-operation dataflow modeling is to capture how data is transferred between operations (e.g., through DRAM, SRAM, or registers) and how operations are scheduled (e.g., sequentially, in parallel, or in a pipeline). Therefore, a directed acyclic graph (DAG) with annotated edges is often a natural and effective choice for representing inter-operation dataflow. As illustrated in Fig. 2, the DAG shows that OP1 and OP2 are pipelined through SRAM, while both execute sequentially with OP3 via DRAM.

2.3 Issues and Motivation

As discussed in Sec. 1, the common limitation of existing spatial accelerator generators is the lack of support for efficient multi-operation dataflows. Specifically, they either target a single operation on a single systolic array or support fixed fusion patterns such as GEMM followed by activation. Such designs are insufficient for large-scale transformer models, which consist of complex compositions of diverse operations with varying topologies. As a result, these generators often execute operations sequentially, leading to underutilization of PEs and incurring significant memory access overhead.

To address this limitation, our key idea is to group multiple operations and execute them in a pipelined manner, thereby maximizing throughput while minimizing external memory accesses. Instead of generating hardware for individual operations, we propose synthesizing architectures tailored to such operation clusters. However, this approach introduces several issues in designing efficient spatial accelerators.

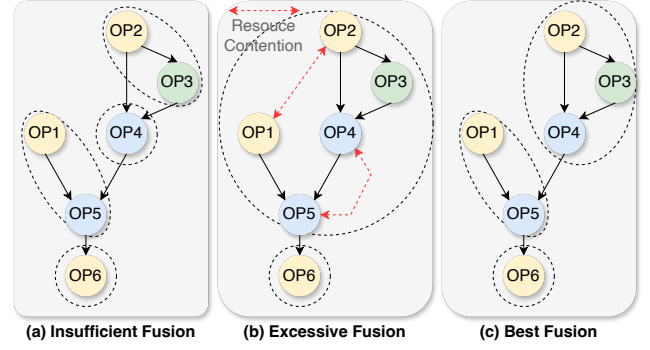


Figure 3. Example of different fusion results.

Pitfall 1: Suboptimal operation fusion. Given the large computation graph of transformers, determining how to fuse operations for optimal performance is highly challenging. Both insufficient fusion and excessive fusion can lead to suboptimal results. Fig. 3 illustrates three fusion configurations explored by our DSE engine (to be detailed in Sec. 4) for the FFN layer in the prefill stage of LLaMA3-8B. Configuration (a) represents insufficient fusion: although OP2, OP3, and OP4 utilize different compute resources, they are divided into separate clusters. In contrast, (b) illustrates excessive fusion, where OP1–OP5 are fused into a single cluster. Under pipelined execution, operations within the same cluster compete for compute and storage resources (e.g., SRAM), leading to what we term resource contention. This results in PE underutilization for two reasons: (1) imperfect PE allocation cannot fully utilize available resources. For example, assuming a total of 64 PEs, no valid allocation can fully utilize all PEs for two pipelined operations with a FLOP ratio of 2:1; and (2) reduced data reuse, because the limited per-operation SRAM capacity prevents each operation from storing sufficient inputs and partial sums, potentially pushing the computation into a memory-bound regime.

Interestingly, the optimal fusion strategy is workload-dependent. When the workload shifts from the prefill stage to the decoding stage, the same computation graph may favor configuration (b), because fusing two memory-bound operations improves overall resource utilization.

Pitfall 2: Large and complex dataflow design space. Even with an ideal operator fusion, identifying the optimal dataflow parameters for each cluster remains challenging due to the enormous design space. Consider a cluster containing N operations, denoted as OP_i , where each operation is represented by an M_i -level nested loop with bounds $(b_{i0}, b_{i1}, \dots, b_{iM_i-1})$. As introduced in Section 2.2, the dataflow design for each OP_i requires determining: (1) a tiling vector $(t_{i0}, t_{i1}, \dots, t_{iM_i-1})$, with a search space of $\prod_{j=0}^{M_i-1} b_{ij}$; (2) an unrolling vector $(u_{i0}, u_{i1}, \dots, u_{iM_i-1})$, which, for each tiling configuration, has a search space of $\prod_{j=0}^{M_i-1} t_{ij}$; (3) an

outer loop ordering, with a space of $M_i!$; and (4) an inner loop ordering, also with a space of $M_i!$.

When combined across all operations in the cluster, the total design space grows exponentially. In practice, for a cluster of six operations (our default maximum cluster size in experiments) in a sequence-length-1024 attention layer, the overall design space can reach up to 10^{21} possible configurations.

Pitfall 3: Difficulty in generating hardware for diverse operation clusters. After scheduling, another major challenge lies in generating RTL code for the extracted clusters. As discussed earlier, the resulting clusters may exhibit diverse topologies, along with varying SRAM and PE resource allocations. Such diversity significantly complicates RTL development. Moreover, conventional approaches such as hardware templating are difficult to apply due to the lack of flexibility required to accommodate these heterogeneous designs. Furthermore, all clusters must ultimately be mapped onto a unified architecture with shared resources, introducing additional constraints and design complexity.

3 Framework Overview

To address all the challenges, we propose TranSAGE, an automated framework that takes PyTorch code as input, automatically performs multi-kernel pipeline scheduling to extract clusters with optimized dataflow, and generates RTL code for hardware implementation. The overview is shown in Fig. 4. The framework follows a two-stage workflow.

Generation stage. TranSAGE first takes the PyTorch [39] code of the input model and uses Torch-MLIR to generate the corresponding MLIR representations. The frontend begins with Linalg canonicalization and optimization, transforming the input IR into a representation composed solely of `linalg.generic` operations and auxiliary tensor operations. Based on it, TranSAGE fuses memory-oriented generic operations to eliminate redundant memory accesses. TranSAGE then performs multi-operation pipeline scheduling with a DSE engine to identify optimal operation fusion and corresponding dataflow parameters (*Challenge 1*).

Afterwards, TranSAGE performs backend lowering to generate binary and RTL code from the Linalg IR of these fused clusters (*Challenge 2*). We introduce a set of spatial architecture primitives as a bridge between Linalg and RTL implementation. Based on these primitives, we construct a primitive graph that maps all supported clusters onto shared hardware and lowers it to binary and RTL code. Notably, different shapes may favor different schedules. Therefore, TranSAGE independently runs the frontend for representative shapes, incorporating all distinct clusters into the mapping stage. Details are described in Sec. 5.

Inference stage. To serve models on the generated accelerator, TranSAGE uses a host runtime with a dispatch table that maps input shapes to binary code. For each request, the runtime selects the binary for the corresponding shape

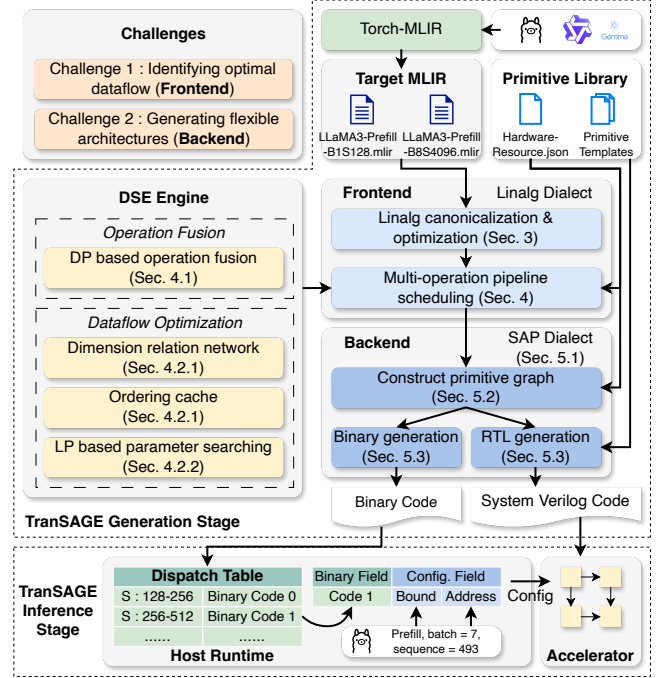


Figure 4. Framework overview.

range. It then fills in the runtime loop bounds and memory addresses, supporting any shape within the range without architectural modification. For example, as shown in Figure 4, a request with a sequence length of 493 uses *Binary Code1*, which corresponds to the sequence length range from 256 to 512. The runtime then configures the accelerator accordingly.

4 Multi-Operation Pipeline Scheduling

Problem Formulation. The canonicalized Linalg IR consists of `linalg.generic` operations connected by tensor operations. During multi-operation pipeline scheduling, we partition these generic operations into clusters, each representing a candidate pipelined grouping. For each cluster, we characterize the pipelined dataflow using parameters including the tiling vector, unrolling vector, and outer/inner loop ordering (Sec. 2.2), which define the execution behavior of each cluster. Therefore, the scheduling problem becomes a joint optimization over fusion schemes and dataflow parameters, aiming to find a configuration that optimizes the target metric, namely, *energy delay product (EDP)*.

4.1 Operation Fusion

We formulate operation fusion as a graph partitioning problem, where the objective is to partition the computation graph into several clusters. This problem has a natural recursive structure: given a topologically ordered sequence of operations, once the optimal final cluster is determined, it can be removed, reducing the original problem to a smaller subproblem over the remaining operations.

Algorithm 1 DP-based Operation Fusion

Require: Operation list $[0 \dots N - 1]$, max cluster size K
Ensure: Cutting index for optimal fusion and metrics

```

1: Initialize  $dpStatus[0] \leftarrow \infty$ 
2:  $cutIndex[0] \leftarrow 0$ 
3: for  $i = 1$  to  $N$  do
4:    $bestEDP \leftarrow \infty$ ,  $bestIndex \leftarrow 1$ 
5:   for  $j = 1$  to  $i$  do
6:      $cluster \leftarrow \text{operations}[j, i]$ 
7:     if  $\text{connectivity}(cluster) \wedge i - j + 1 \leq K$  then
8:        $EDP \leftarrow \text{SOLVE}(cluster)$ 
9:        $newEDP \leftarrow \text{MERGE}(dpStatus[j - 1], EDP)$ 
10:      if  $newEDP < bestEDP$  then
11:         $bestEDP \leftarrow newEDP$ 
12:         $bestIndex \leftarrow j$ 
13:      end if
14:    end if
15:  end for
16:   $dpStatus[i] \leftarrow bestEDP$ 
17:   $cutIndex[i] \leftarrow bestIndex$ 
18: end for

```

Therefore, we adopt a dynamic programming (DP)-based method for graph partitioning. First, all operations are sorted in topological order. For a total of N operations, we define an array $dpStatus[0 \dots N]$, where $dpStatus[i]$ records the optimal EDP for partitioning the subgraph consisting of operations 0 to i . We also maintain an array $cutIndex[0 \dots N]$, where $cutIndex[i]$ records the corresponding partition point.

Our graph partitioning algorithm is presented in Algorithm 1. Starting from the first node, for each node i , the algorithm enumerates all possible cut points $j \leq i$ to identify a partition whose operations form a fully connected cluster. This is because fusing operations without data dependencies provides no theoretical benefit. Among valid partitions, the algorithm selects the one that minimizes the overall EDP. The corresponding optimal EDP is stored in $dpStatus[i]$, while the selected cut point is recorded in $cutIndex[i]$. After processing all nodes, we reconstruct the final partitioning by backtracking. Starting from $i = N$, where i denotes the current cut index, we iteratively set $i = cutIndex[i]$ to obtain the preceding cut index until reaching the beginning of the graph. During backtracking, the optimal dataflow parameters for each cluster are determined (Sec. 4.2) and attached to the corresponding operations as attributes, enabling subsequent stages to leverage them for RTL generation.

4.2 Intra-Cluster Dataflow Optimization

The objective at this stage is to determine the tiling vector, the unroll vector, the outer-order vector and the inner-order vector for each operation (Sec. 2.2) within a cluster to maximize the performance. As discussed in Sec. 2.3, the design

space is extremely large, making brute-force search infeasible. However, we observe that unroll vectors and inner-order vectors of different operators are relatively independent (corresponding to the higher memory hierarchy $L1$ between SRAM and registers), whereas tiling vectors and outer-order vectors are tightly coupled (corresponding to the lower memory hierarchy $L2$ between DRAM and SRAM). Therefore, we propose a decomposed optimization approach. We first enumerate all possible combinations of tiling vectors and outer-order vectors, which we refer to as $L2$ parameters. For each $L2$ configuration, we determine the optimal unroll factors and inner orders—denoted as $L1$ parameters—using a well-formulated and efficient solving method.

4.2.1 Search Space Pruning for $L2$ Parameters. Despite the decomposition, the search space of $L2$ parameters remains prohibitively large. To this end, we prune the search space through: a *dimension-relation network representation* and an *equivalence-based ordering cache*.

Dimension-Relation Network Representation. Within each cluster, due to the pipelined data transfer between operations through SRAM, there exist inherent constraints on the tiling sizes of producer and consumer operations. For example, in Fig. 5(a), if the producer MM1 generates data in 64×128 tiles, it is desirable for the consumer MM2 to process data at the same granularity. This implies strong dependencies among tiling sizes at corresponding dimensions across different operators. Therefore, instead of exhaustively enumerating all possible tiling sizes for each dimension, we can enumerate only a small set of representative dimensions and infer the rest, thereby reducing the search space greatly.

Based on this observation, we propose a *dimension-relation network* representation, as shown in Fig. 5. It models the dependencies among operator dimensions as a directed graph, where nodes correspond to operation dimensions or relation nodes, and edges capture the dependency relationships. Each *relation* node is associated with a forward function that infers the tiling size of output dimensions based on the input dimensions. We define three fundamental types of relations which are typical in real-world LLMs.

(1) *Equal Relation.* As shown in Fig. 5(a), this common relation in pipelined execution arises from producer-consumer dependencies. It enforces that the tiling size of one dimension must equal that of the corresponding dependent dimension. The forward function directly propagates the tiling size from the input dimension to the output dimension.

(2) *Collapse Relation.* It originates from `tensor.collapse_shape` and `tensor.expand_shape`. These operations reinterpret tensors without modifying the underlying data. As illustrated in Fig. 5(b), an expand operation decomposes a dimension N_1 into two dimensions H_2 and D_2 . Under this relation, the tiling sizes of the decomposed dimensions must satisfy a consistency constraint such that their product equals

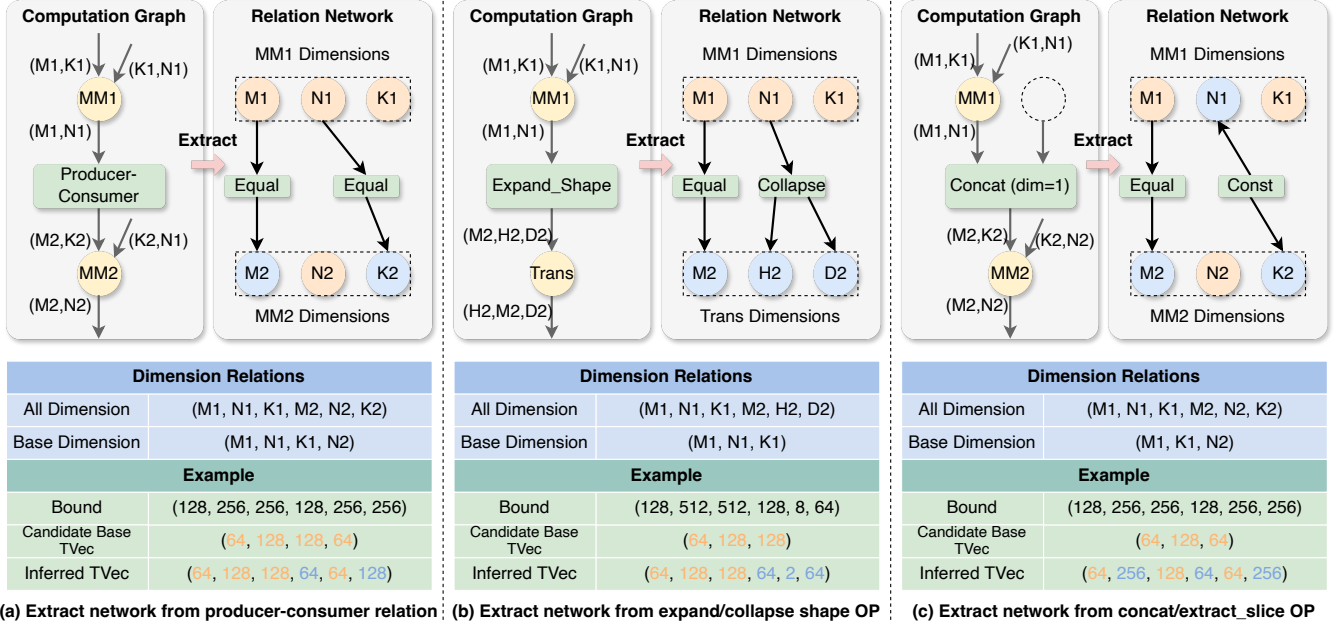


Figure 5. An illustrative example of a dimension-relation network. MM denotes a matrix multiplication operation, Trans denotes a transpose operation, and TVec denotes the tiling vector. Orange: base dimensions; blue: inferred dimensions.

the original tiling size. Moreover, to ensure contiguous memory access, we adopt a low-to-high dimension allocation strategy, i.e., higher dimensions can only be tiled once lower dimensions reach their bounds. As shown in Fig. 5(b), for N_1 , H_2 , and D_2 , a tiling configuration (128, 2, 64) is valid because D_2 reaches its bound, leading to contiguous access. In contrast, (128, 4, 32) is an invalid configuration.

(3) *Constant Relation.* As shown in Fig. 5(c), it is introduced by operations such as concat and extract_slice, which either merge tensors along a dimension or extract slices. To ensure correctness and on-chip execution, the involved dimension and all lower dimensions must be fully tiled (i.e., equal to their bounds).

The relation network is constructed by traversing operations in a cluster in topological order. For each operation, TranSAGE examines its inputs and establishes the corresponding dimension relations. If an input is the direct output of a preceding operation, TranSAGE adds equal relations between the corresponding dimensions. If it is produced by collapse_shape and expand_shape operations, TranSAGE adds collapse relations for the affected dimensions and equality relations for the others. If the input is generated by concat and extract_slice operations, TranSAGE assigns constant relations to the involved dimensions by fixing them to their bounds, while preserving equality relations for the remaining dimensions. After the traversal, the dimension-relation network is obtained, as shown in Fig. 5(a)–(c).

After constructing the relation network, we identify the base dimensions that cannot be inferred from others. We

then enumerate tiling configurations only for these base dimensions and propagate each candidate through the network to infer the remaining tiling sizes, yielding a complete tiling vector for performance evaluation.

Equivalence-Based Ordering Cache. The outer-order vector is another critical L2 parameter, as it determines memory access patterns and thus DRAM efficiency. As with tiling sizes, operation ordering within a cluster is constrained. To enable direct tensor forwarding between producer and consumer, without buffering or reordering, the access order for a tensor must satisfy two conditions. First, the reuse distance of each tile must be 1. Figure 6(a) illustrates an example of the access order for each operand. With reuse distances of 2 and 1 for the activation and output tensors, respectively, the output tensor can be directly pipelined to the next operation, whereas the activation tensor cannot be forwarded directly from the previous operation. Second, the producer and consumer must access the tensor in the same order. For example, an operation that directly consumes the output in Figure 6(a) must access the output tiles in the same order in which they are produced, i.e., Tile0, Tile0, Tile1, Tile1, ...

The tiling size further complicates this constraint, as it can alter the access consecutivity of tensors. Taking Figures 6(b) and (c) as examples, both cases follow the outer order $[M, K, N]$. However, different tiling sizes lead to different output access patterns: Figure 6(b) is non-consecutive, whereas Figure 6(c) is consecutive. Thus, we must generate all valid outer-order vectors for each candidate tiling size. Although this can be achieved by performing a depth-first

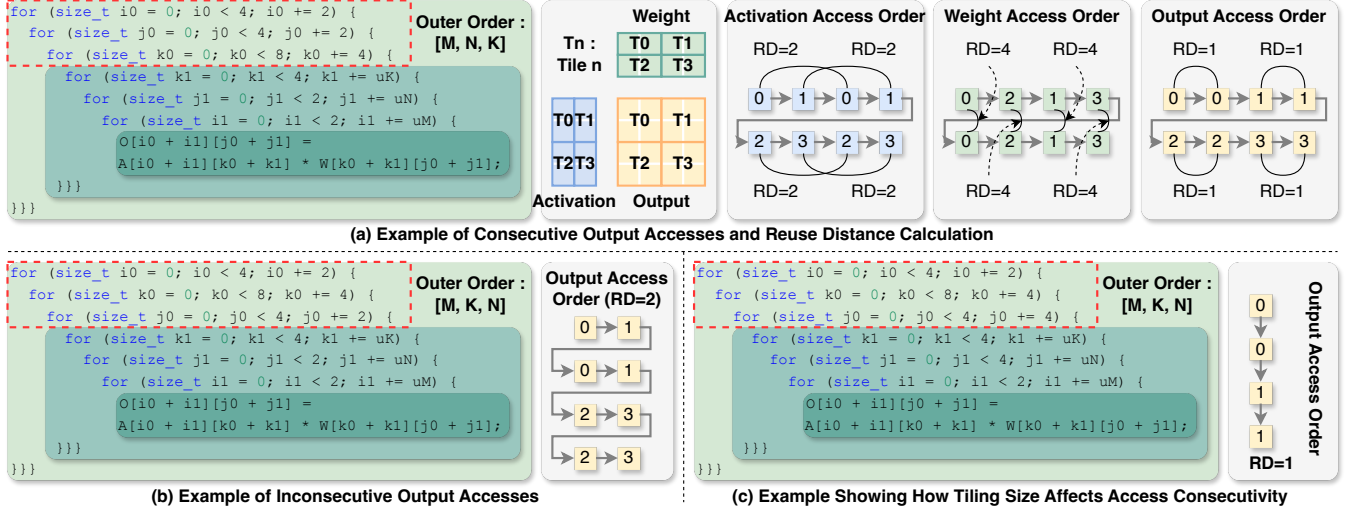


Figure 6. Examples of different tensor access orders. RD denotes reuse distance.

search (DFS), the search space grows exponentially with both the cluster size and the number of candidate tiling vectors, making exhaustive enumeration impractical.

Our key observation is that different tiling configurations can fall into the same *equivalence class* when searching for valid outer-loop orders, thereby significantly reducing the search cost. Consider the tiling size (2, 2, 4) in Figure 6(a) and (b). The only outer-loop orders that preserve consecutive output accesses are (M, N, K) and (N, M, K). If the tiling size is changed to (2, 2, 2), the set of valid outer-loop orders remains unchanged. More generally, any tiling configuration that does not fully tile any dimension has the same set of valid outer-loop orders. We therefore group such tiling configurations into the same *equivalence class*.

Based on this observation, we define a *dimension mask* to indicate whether each dimension is fully tiled. All tiling configurations sharing the same dimension mask belong to the same equivalence class and thus share identical candidate order sets. Leveraging this property, we propose an equivalence-based ordering cache implemented as a hash map. During L2 parameter enumeration, for each tiling configuration, we compute its dimension mask (encoded as a 64-bit integer) and check whether the corresponding equivalence class has already been explored. If so, we directly reuse the cached candidate order set; otherwise, we perform DFS-based enumeration and store the results in the cache.

4.2.2 Solver-based Search for L1 Parameters. The *L1 parameters* of each operation are independent and can thus be optimized separately under shared system constraints. We model the execution of a cluster as a fully pipelined dataflow system, where each operation produces one output tile per iteration. The steady-state pipeline latency is determined

by the slowest stage. Accordingly, we first optimize the per-operation parallelism p_i to minimize the overall pipeline latency. Let f_i denote the computational cost (in floating-point operations) required to produce one output tile of operation i . The per-tile latency of operation i is f_i/p_i , yielding the minimax optimization objective: $\min_{\{p_i\}} \max_i \left(\frac{f_i}{p_i} \right)$, subject to global resource constraints: $\sum_i n_{ij} \leq N_j$.

Here, n_{ij} denotes the amount of resource type j required by operation i , and N_j is the corresponding available resource budget. By introducing an auxiliary variable t , the problem can be equivalently reformulated as:

$$\min_{t, \{p_i\}} t \quad \text{s.t.} \quad \frac{f_i}{p_i} \leq t, \forall i, \quad \sum_i n_{ij} \leq N_j, \forall j,$$

which yields a standard form amenable to linear programming. This formulation can be efficiently solved using off-the-shelf linear programming solvers such as GLPK [31].

After obtaining the optimal parallelism configuration $\{p_i^*\}$, TranSAGE maps it to loop unrolling factors over dimensions. We restrict unrolling to at most two dimensions to match the 2D PE array. We prioritize dimensions that preserve contiguous memory accesses for each operand tensor, improving spatial locality and memory bandwidth utilization. The selected unroll dimensions are then placed at the outer loop levels of the L1 memory hierarchy to maximize data reuse across iterations.

5 Hardware Generation

In this section, we introduce the abstraction termed spatial architecture primitives (SAPs) and describe how high-level representations are progressively lowered to SAPs, ultimately generating binary and RTL code.

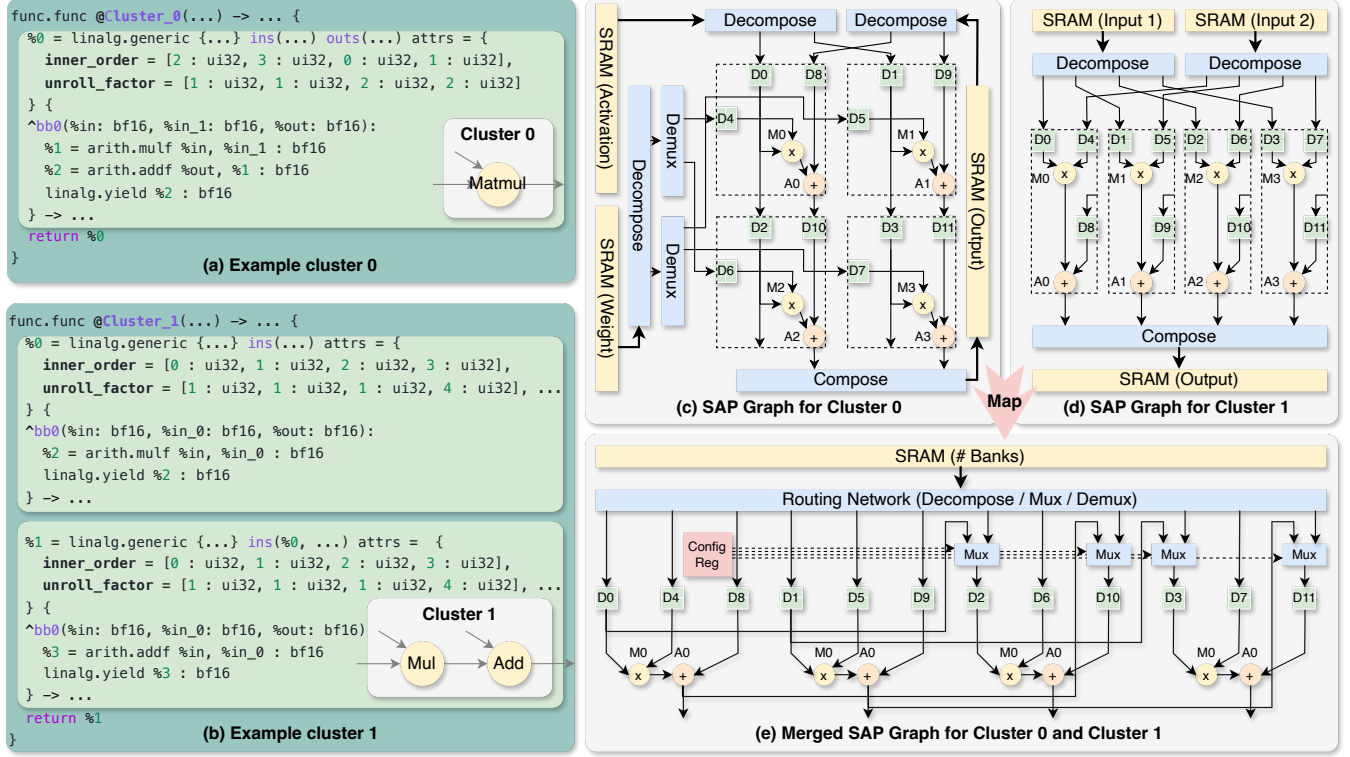


Figure 7. Example of generated primitive graph. D_n denotes DataNode n , M_n denotes Multiplier n , and A_n denotes Adder n .

5.1 Spatial Architecture Primitives

Spatial architecture primitives (SAPs) are proposed to address the difficulty of generating hardware for multi-operation pipelined clusters with heterogeneous PEs and flexible topologies. The key idea is to model accelerators as directed cyclic graphs, where nodes represent storage, computation, or routing components, and edges represent dataflow dependencies.

To enable automated graph construction, we define four classes of primitives. For edges, DataPath represents connections between source and destination ports. For nodes, we define three categories: data, routing, and arithmetic primitives. Data primitives include FIFO- and SRAM-based storage components. Routing primitives handle data movement, such as Decompose, Compose, Mux, and Demux, by transforming data stream formats or directing data paths. Arithmetic primitives represent computation operations. All SAPs are summarized in Appendix A.

5.2 SAP Graph Construction

Taking scheduled clusters as input, the TranSAGE backend constructs a SAP graph to represent a reconfigurable architecture that supports all required dataflows of clusters. The construction process consists of three stages.

Primitive allocation. First, based on the hardware resource configurations specified in the primitive library, we

pre-instantiate storage and arithmetic primitives and organize them into a resource pool. For each cluster, we allocate arithmetic and DataNode resources to each linalg operation based on its unrolling factors, and SRAM to each producer-consumer relation based on its tiling size to enable data pipelining. This design maps resources required by different dataflows onto a fixed set of physical primitives. As illustrated in Figures 7(c) and (d), two different dataflows physically share the same set of DataNodes ($D0-D11$) and arithmetic primitives ($M0-M3$ and $A0-A4$).

Datapath construction. After allocation, each linalg operation is equipped with a set of DataNodes, arithmetic primitives, and SRAM primitives. We organize the DataNodes and arithmetic primitives as a 2D array according to the unroll factors, while organizing the SRAM primitives as 1D vectors. The datapath construction stage connects these primitives by establishing the required data paths.

This process consists of three sub-steps. (1) *Routing data from SRAM to DataNodes.* Since SRAM and DataNode array shapes are typically incompatible, we reshape and distribute SRAM data to the DataNodes using routing and datapath primitives. In Figures 7(a) and (c), a *Decompose* primitive partitions the activation input into 1×2 blocks. With shared input dependencies, datapath primitives connecting adjacent PEs broadcast each element along the row to form the 2×2 input required by the DataNode array. For weights, a

Decompose primitive similarly partitions the SRAM output, while a *Demux* primitive preloads each 2×2 weight block into different rows, as the iterations have no shared input dependencies. Finally, partial sums are initialized by reading a 1×2 block; thus, only a *Decompose* primitive is needed to directly feed the first row of output DataNodes.

(2) *Synthesizing the arithmetic datapath.* For the arithmetic datapath, we analyze the data dependencies within the linalg body and connect the corresponding DataNodes and arithmetic primitives accordingly.

(3) *Routing data back to SRAM.* Similar to the input path, the DataNode array and output SRAM may have incompatible shapes. We therefore insert routing primitives based on output dependencies. In Figures 7(a) and (c), the 1×2 output blocks and 2×2 DataNode array imply a row-wise reduction, so datapath primitives forward partial sums along the row. A *Compose* primitive then packs the outputs onto the wide SRAM bus for writing.

Datapath reorganization. Since each linalg operation independently connects its storage and arithmetic primitives, the resulting datapaths may contain redundant primitives and lack support for reconfiguration. We therefore further reorganize the datapath to eliminate redundancy and enable reconfigurability. *First*, we eliminate redundancy. When multiple instances of the same primitive connect to a single port, we reorganize the datapath to share one instance. A typical example is eliminating redundant *Decompose* primitives connected to the same SRAM port. For linalg operations with the same unroll factor and inner order, we similarly eliminate intermediate SRAM and directly forward results. For example, in Cluster 1 of Figures 7(b) and (d), the matching configurations allow the Adder output to be directly forwarded to the Multiplier. *Second*, we introduce reconfigurability by inserting a *Mux* whenever a primitive has multiple possible input sources, allowing the source to be selected at runtime through configuration. As illustrated in Figure 7(e), configuring these *Mux* primitives with configuration registers enables the accelerator to execute either of the dataflows shown in Figure 7(c) and (d). This reconfigurability incurs only minor overhead (Sec. 6.4).

5.3 Binary and RTL Code Generation

After constructing the SAP graph, TranSAGE generates the binary and RTL code. We map each cluster to a binary which encodes parameters including tiling, unrolling, loop orders, and multiplexer control bits, while runtime fields specify loop bounds and memory addresses. These binaries are stored in the dispatch table for runtime execution. Meanwhile, the RTL code is generated by replacing each primitive in the SAP graph with its RTL template or hardware macro.

6 Evaluation

6.1 Experimental Setup

Workloads. We select Llama3-8B [16], Qwen3-8B [55], Gemma-7B [47], Llama3-70B [16], and Qwen3-235B-A22B (balanced routing, abbreviated as Qwen3-MoE) [48] as representative LLM workloads. Due to space constraints, we report the Llama3-8B results in Appendix C.

GPU and NPU Baselines. We use the NVIDIA Orin NX [36] and NVIDIA A100 GPU [35] as baselines. We also compare against the Ascend 910B3 NPU [28], with BF16 peak throughput and memory bandwidth comparable to the A100. We use implementations from Transformers library [21], with native PyTorch [39] and Torch-NPU [20] backends for NVIDIA and Ascend devices, respectively, leveraging optimized device-specific kernels. To obtain accurate latency measurements, we explicitly synchronize device execution using `torch.cuda.synchronize()` and `torch.npu.synchronize()` before recording execution time. We measure power consumption using `tegrastats`, `nvidia-smi`, and `npu-smi` for the Orin NX, A100, and Ascend 910B3, respectively.

Accelerator Generator Baselines. We compare TranSAGE with two accelerator generators: Gemmini [14] and LEGO [29]. For Gemmini, we consider an architecture consisting of a systolic array and a vector processing unit operating under output-stationary and weight-stationary dataflows, denoted as GEMMINI-OS and GEMMINI-WS, respectively. For LEGO, due to its flexibility, we instantiate an accelerator composed of a systolic array and a vector processing unit with a configurable dataflow that can dynamically switch among input-, weight-, and output-stationary modes.

We model the baselines using Timeloop and Accelergy [37], which support accelerator architectures and SRAM-/DRAM modeling. For each architecture, we perform mapping exploration using Timeloop and select the mapping that minimizes EDP under the same optimization criteria and search cost, reporting corresponding performance results. Since Timeloop does not support modeling multiple pipelined operations, we extract latency from our performance model for generated accelerators and combine it with energy estimates from Accelergy to obtain the final metrics.

All experiments are conducted on a system equipped with an Intel Xeon Gold 5218 CPU. We use edge/server configurations for accelerator-generator comparisons and the largest valid PE configuration with BF16 peak throughput no greater than the target device for commercial-devices comparisons. All TranSAGE configurations run at 1 GHz using a 45 nm technology model, with parameters summarized in Appendix B. We use DeepScaleTool [43] to scale energy estimates to 8 nm for Orin NX and 7 nm for A100 and Ascend 910B3.

6.2 Comparison with GPU and NPU

Throughput. As shown in Figure 8, for prefill, TranSAGE achieves average throughput improvements of 2.05 $\times$, 1.65 $\times$,

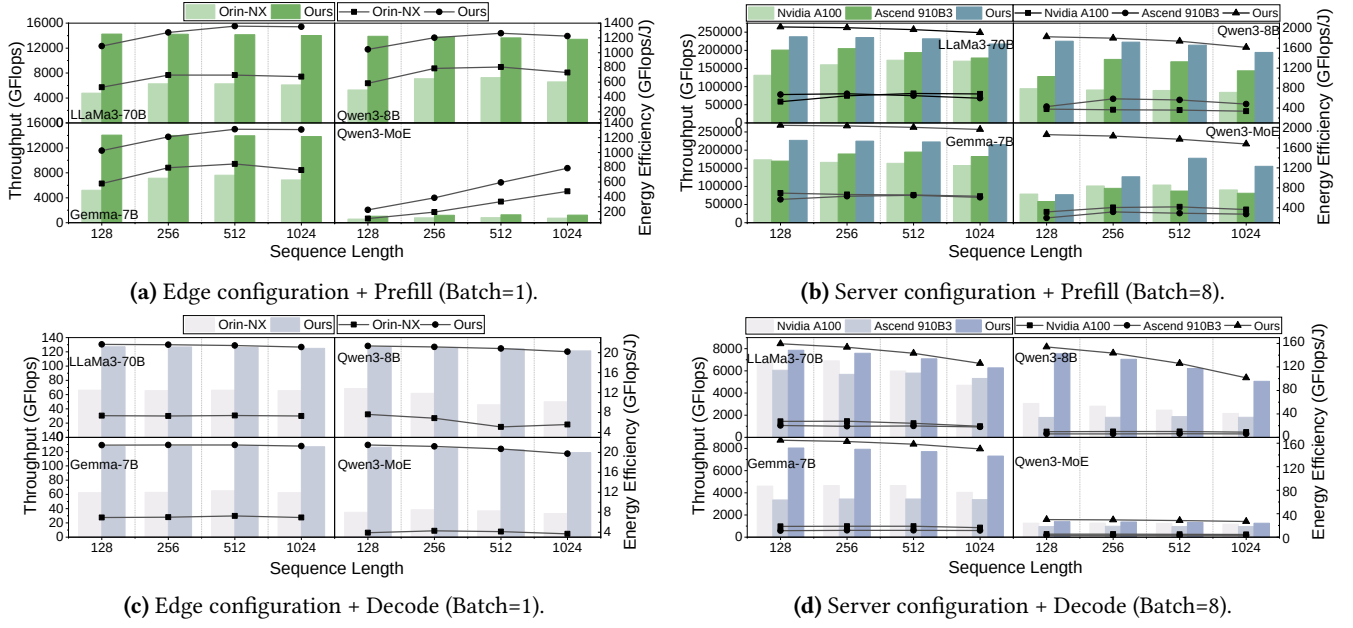


Figure 8. Throughput and energy-efficiency comparison across GPUs and commercial accelerators. Bars show throughput (left y-axis), and lines show energy efficiency (right y-axis).

and 1.42 $\times$ over the Orin NX, A100, and Ascend 910B3 NPU, respectively. For dense models, including LLaMa3-70B, Qwen3-8B, and Gemma-7B, the improvement mainly comes from pipelining auxiliary operations, which further increases overall throughput even when the matrix multiplications (MM) already achieve relatively high utilization. For Qwen3-MoE, its MoE structure results in low compute density in the FFN operations, allowing TranSAGE to further benefit from fusing short-sequence MM operations. For the memory-bound decode workload, TranSAGE achieves average throughput improvements of 2.41 $\times$, 1.62 $\times$, and 2.12 $\times$ over the Orin NX, A100, and Ascend 910B3, respectively, demonstrating the benefits of operation fusion for memory-bound workloads.

Energy efficiency. TranSAGE achieves average energy-efficiency improvements of 1.78 $\times$, 3.46 $\times$, and 3.69 $\times$ over Orin NX, A100, and 910B3, respectively, on prefill, demonstrating the energy-efficiency advantage of our specialized architecture. For the decode workload, where memory accesses dominate energy consumption, TranSAGE achieves average energy-efficiency improvements of 3.65 $\times$ over the Orin NX and 8.57 $\times$ and 13.04 $\times$ over the A100 and 910B3, respectively. These gains stem primarily from the reduction in memory traffic enabled by operation fusion.

6.3 Comparison with SOTA Accelerator Generators

Throughput. We next compare TranSAGE with SOTA accelerator generators. As shown in Figure 9, for the prefill workload, TranSAGE achieves average throughput improvements of 1.58 $\times$, 2.22 $\times$, and 1.82 $\times$ over GEMMINI-OS, GEMMINI-WS, and LEGO, respectively, under the edge configuration.

Under the server configuration, TranSAGE achieves 4.14 $\times$, 3.28 $\times$, and 5.50 $\times$ improvements over the same baselines. Similar to the GPU comparison, these gains mainly come from pipelining auxiliary operations and reducing memory accesses for memory-bound operations. Moreover, TranSAGE maintains consistent throughput improvements across different sequence lengths, demonstrating the effectiveness of its adaptive dataflow. For the decode workload, TranSAGE further achieves average throughput improvements of 3.13 $\times$, 1.46 $\times$, and 1.46 $\times$ over GEMMINI-OS, GEMMINI-WS, and LEGO, respectively, under the edge configuration. Under the server configuration, the improvements increase to 11.33 $\times$, 3.37 $\times$, and 3.37 $\times$, respectively.

Utilization. Using the throughput reported in Figure 9, we analyze effective BF16 arithmetic utilization. TranSAGE achieves up to 96.15% and 90.20% utilization on the prefill workload under the edge and server configurations, respectively. This high utilization is enabled by the high compute density and the sharing of the same set of hardware resources across all clusters. In contrast, on the decode workload, TranSAGE achieves 3.16% and 3.05% utilization under the edge and server configurations, respectively, primarily due to the limited memory bandwidth.

Energy efficiency. Operation fusion also improves energy efficiency. For the prefill workload, TranSAGE achieves average energy efficiency improvements of 1.46 $\times$, 1.56 $\times$, and 2.53 $\times$ over GEMMINI-OS, GEMMINI-WS, and LEGO, respectively, under the edge configuration. It increases to 2.13 $\times$, 2.15 $\times$, and 3.13 $\times$ under the server configuration. For the decode workload, TranSAGE achieves up to 1.73 $\times$ and 6.07 $\times$

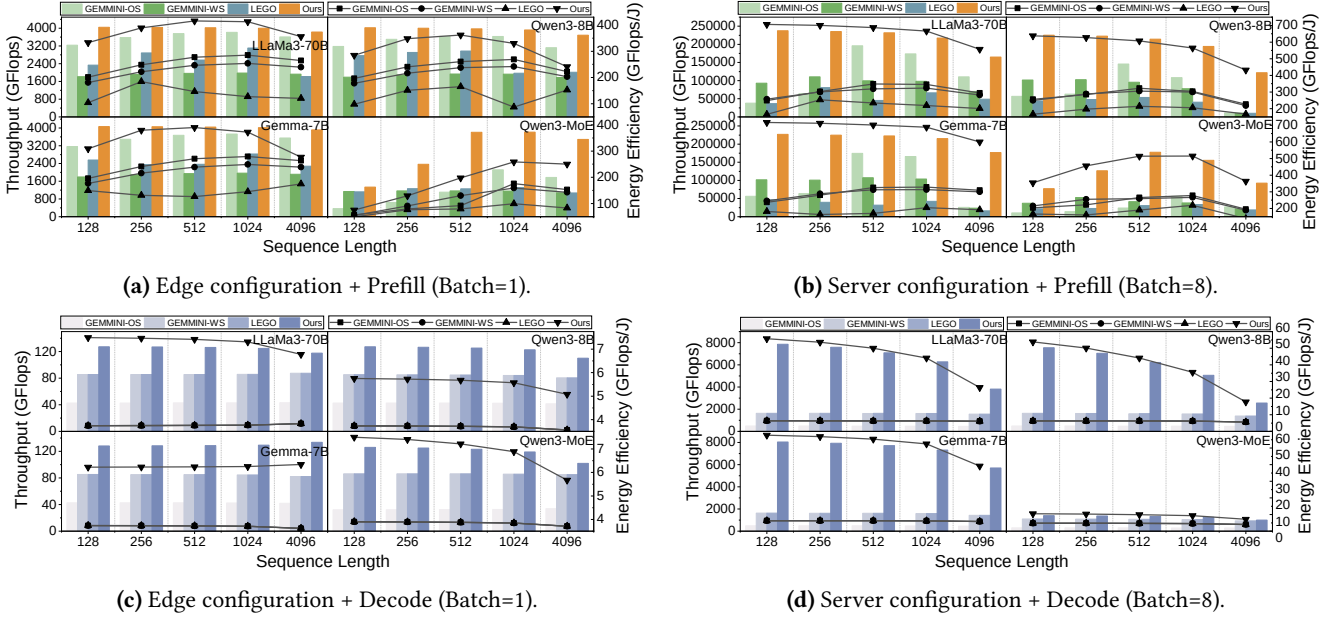


Figure 9. Throughput and energy-efficiency comparison with state-of-the-art accelerator generators. Bars show throughput (left y-axis), and lines show energy efficiency (right y-axis).

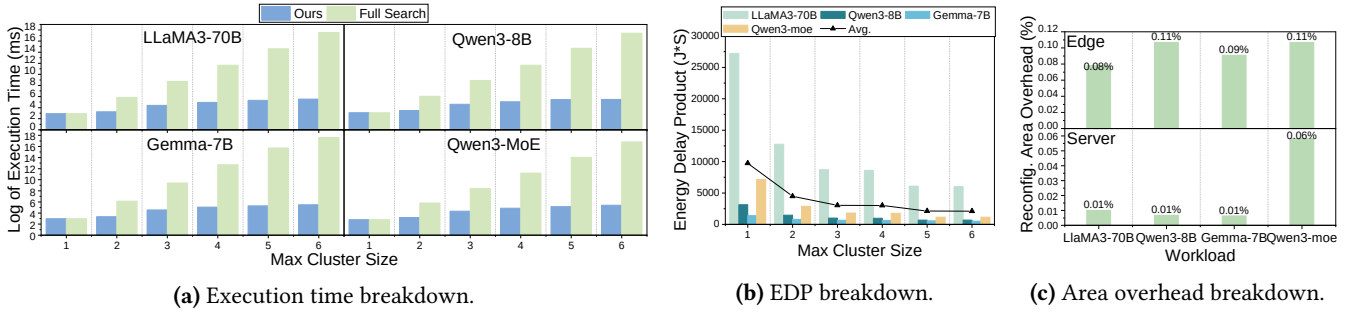


Figure 10. Ablation study of the maximum cluster size and reconfiguration area overhead.

energy efficiency improvement over the baselines under the edge and server configurations, respectively.

6.4 Ablation Study

To further demonstrate the feasibility and performance of TranSAGE, we provide a breakdown analysis of the impact of the maximum cluster size, which limits the number of operations that can be grouped into a cluster, as well as the overhead introduced by reconfiguration.

Execution Time. First, we evaluate how the maximum cluster size affects the execution time of TranSAGE in solving the scheduling problem for multi-operation clusters. As shown in Fig. 10 (a), we compare the DSE time of TranSAGE with that of a full-search method which exhaustively enumerates all parameters in the prefill stage. We observe that the execution time of full search grows exponentially with the maximum cluster size, becoming impractical once the

cluster size exceeds 4. In contrast, TranSAGE, which relies on search-space pruning and decomposition, exhibits an almost linear increase in search time. Even with a maximum cluster size of 6, the search completes within tens of minutes, demonstrating the efficiency of our framework.

EDP. We further evaluate how the maximum cluster size affects the EDP. Fig. 10 (b) presents the EDP breakdown under varying maximum cluster sizes in the prefill stage with sequence length 4096. When the maximum cluster size is 1, i.e., without pipelining, all models exhibit relatively high EDP. As the maximum cluster size increases, EDP drops rapidly, further confirming the effectiveness of multi-operation pipelining. However, the improvement gradually saturates. This is because beyond a certain cluster size, pipelining can no longer effectively improve PE utilization; instead, operations begin to compete for shared PE and SRAM resources, leading to contention, lower utilization, and reduced data reuse.

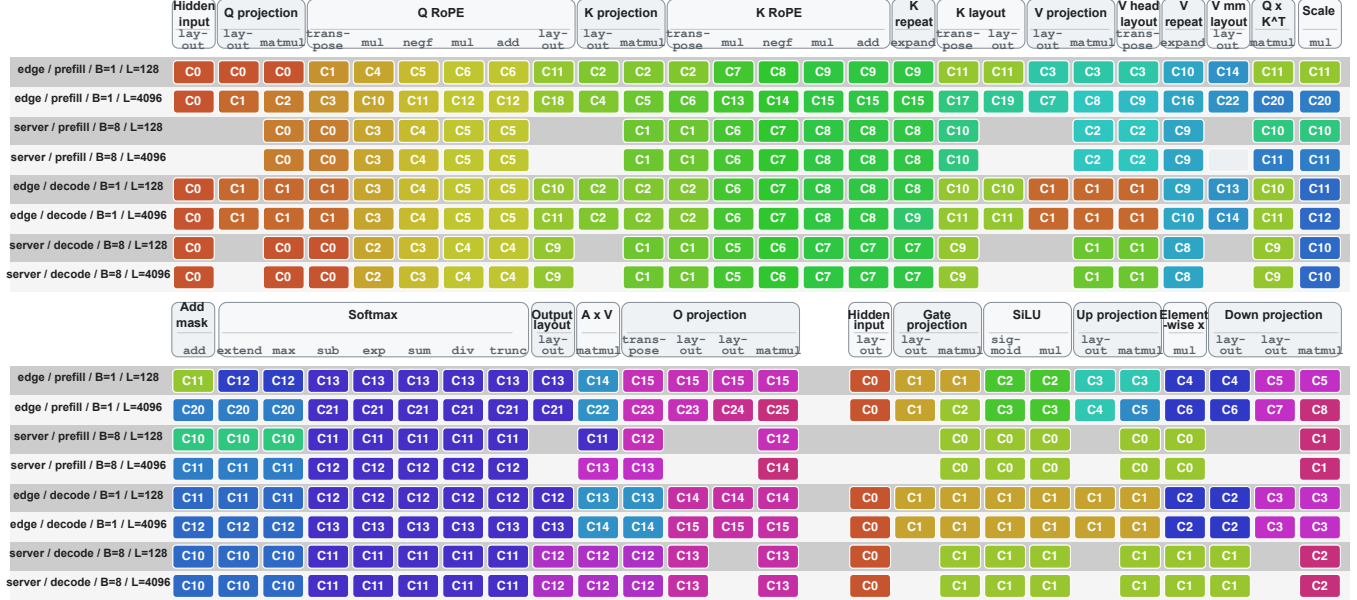


Figure 11. Illustration of the resulting clusters from DSE across eight representative workloads. Each C_n , shown in a unique color, indicates that the corresponding operation is fused into cluster n for the given workload.

Reconfiguration overhead. As discussed in Section 5.2, multiplexers are introduced to support dataflow reconfiguration. We therefore analyze the resulting area overhead. As shown in Figure 10 (c), the area overhead due to reconfiguration is negligible across all evaluated models and configurations, reaching at most 0.11%. This is because, for the same bit width, multiplexers occupy substantially less area than arithmetic PEs such as MAC units. Moreover, each model requires only a limited number of distinct dataflow configurations, further limiting the overall reconfiguration overhead.

7 Discussion

7.1 Analysis of Fused Clusters

Fusion patterns largely determine topology flexibility and thus the resulting architectures. To characterize which operations are suitable for fusion, Figure 11 presents the resulting clusters for LLaMA-8B across representative edge/server, prefill/decode, and short/long-sequence workloads.

Observations. Figure 11 reveals two notable features of the fused clusters. First, fusion patterns are workload- and resource-dependent: under different configurations, the same model exhibits different fusion patterns. In general, edge configurations tend to produce more clusters than server configurations, while long-sequence workloads tend to result in more clusters than short-sequence workloads. This difference stems from variations in internal memory requirements and the available hardware resources. Second, effective fusion crosses conventional operator boundaries. Operations that are typically treated as indivisible units in conventional designs, such as Softmax, RoPE and Gate projection, are often split

across multiple clusters or fused with operations from other stages. These observations make it challenging to manually design an accelerator for each workload, highlighting the capability of TranSAGE to enable rapid design exploration across diverse workload and hardware configurations.

Design insights. Nevertheless, the generated partitions also reveal recurring principles that can guide future spatial-accelerator designs. First, fusing data-dependent operations that leverage complementary compute resources is generally beneficial. Such fusion can reduce memory accesses while improving resource utilization, making these operations ideal candidates for fusion. As shown in Figure 11, consecutive elementwise operations, such as Sub, Exp, Sum, Div, and Trunc in Softmax, as well as Mul and Add in RoPE, utilize different compute resources and can therefore be naturally fused to form an efficient pipeline.

Second, fusing data-dependent operations that compete for compute resources is beneficial only when they can form a balanced pipeline. More fusion is not necessarily better, especially when the fused operations compete for the same resources. A typical example is Q/K/V projection. Q and K/V projections share the same input and thus have potential for fusion, but they are split into separate clusters under all prefill configurations, as shown in Figure 11. This is because both operations consume substantial arithmetic resources during prefill, while group-query attention (GQA) results in imbalanced workloads and resource underutilization, as discussed in Section 2.3, Pitfall 1. In contrast, as shown in Figure 11, the gate and up projections in the FFN can be

efficiently fused under server-prefill configurations. The reason is that these operations have identical output shapes and balanced computational demands, enabling even resource allocation and a balanced pipeline.

7.2 Limitations and Future Directions

TranSAGE has two limitations. First, its physical compute, storage, and interconnect resources are fixed after generation. Despite runtime reconfiguration, evolving model structures may require hardware regeneration. Second, TranSAGE optimizes single-chip architectures. While it can be extended to multi-chip systems through data parallelism, it does not explore other parallelism strategies that account for inter-chip communication. Future work will aim at providing greater flexibility in generated architectures, as well as communication-aware multi-chip DSE.

8 Conclusion

We present TranSAGE, an MLIR-based framework for automatically generating spatial accelerators for large-scale transformers. By treating operation clusters as execution units, TranSAGE enables efficient pipelining and improves throughput and energy efficiency over GPUs, NPUs, and prior accelerator generators for both prefill and decoding.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant 62622214 and Grant 62472273. Codex were used for language and code assistance.

A Spatial Architecture Primitives

Primitive	Op	Operand	Results	Description
Link	DataPath	src, dest	/	A point-to-point connection that transfers data between two ports.
Data	DataNode	/	din, dout	A FIFO-based storage node that buffers streaming data.
	SramBank	/	write, waddr, din, read, raddr, dout	An SRAM-based storage node supporting indexed reads and writes.
Routing	Mux	/	dins[...], dout	Selects one input and forwards it to the output.
	DeMux	/	din, douts[...]	Routes one input to one of multiple destinations.
	Broadcast	/	din, douts[...]	Replicates input data to multiple output ports.
	Reduction	/	dins[...], dout	Aggregates multiple inputs using an operation such as sum or max.
	Compose	/	dins[...], dout	Combines multiple elements into a structured output.
	Decompose	/	din, douts[...]	Splits structured input data into multiple elements.
Arithmetic	Mulf	/	lhs, rhs, result	Floating-point multiplication.
	Addf	/	lhs, rhs, result	Floating-point addition.
	Subf	/	lhs, rhs, result	Floating-point subtraction.
	Divf	/	lhs, rhs, result	Floating-point division.
	Maxf	/	lhs, rhs, result	Floating-point maximum.
	Negf	/	src, result	Floating-point negation.
	Truncf	/	src, result	Floating-point truncation.
	Extf	/	src, result	Floating-point extension.

B System Configurations

Resource	Edge	Orin NX matched	Server	A100/Ascend 910B3 matched
DRAM bandwidth	128 GB/s	128 GB/s	1024 GB/s	1024 GB/s
SRAM width	512 bit	512 bit	512 bit	512 bit
SRAM depth	256	256	256	256
SRAM banks	256	256	1024	1024
BF16 multipliers	2112	8320	131584	131584
BF16 adders	2112	8320	131584	131584
Miscellaneous PEs	64	128	512	512
Peak BF16 throughput ¹	4.224 TFLOP/s	16.640 TFLOP/s	263.168 TFLOP/s	263.168 TFLOP/s

All configurations operate at 1 GHz. Device-matched configurations use the largest valid PE organization whose peak throughput does not exceed that of the target device. ¹ One multiplication and one addition count as two floating-point operations. Each PE issues one operation per cycle.

C LLaMA3-8B Results

Sequence length	Edge (batch 1)				Server (batch 8)			
	Prefill-TP	Prefill-EE	Decode-TP	Decode-EE	Prefill-TP	Prefill-EE	Decode-TP	Decode-EE
128	4,002.75	317.05	126.27	7.44	238,287.84	669.32	7,560.52	53.41
256	3,980.38	374.15	125.59	7.38	235,341.01	659.07	7,060.47	49.73
512	3,952.80	394.61	124.29	7.28	226,956.91	638.69	6,243.70	43.78
1024	3,908.76	384.40	121.84	7.09	200,261.22	606.23	5,097.66	35.52
4096	3,667.25	306.37	108.75	6.16	130,890.76	459.65	2,586.75	17.78

TP denotes throughput in GFLOP/s; EE denotes energy efficiency in GFLOP/J.

D Artifact Appendix

D.1 Abstract

The artifact provides Accelgen, the compiler framework for TranSAGE. Starting from PyTorch model code, the workflow generates Linalg-based MLIR, optimizes Linalg operations, schedules kernels, and lowers the scheduled computation to a Spatial Architecture Primitive (SAP) graph for performance modeling. The outputs include intermediate MLIR and performance metrics such as latency, energy, FLOPs, throughput, and energy efficiency. These outputs can be inspected independently of the figures and tables in the paper.

D.2 Artifact Checklist

The artifact produces the following outputs to support reproducibility and inspection of each compilation stage.

- **Generated Linalg MLIR.** The Linalg representation generated from the input model.
- **Optimized Linalg MLIR.** The Linalg representation after Linalg-level optimizations.
- **Scheduled Clusters.** Computation clusters with their schedules and associated dataflow parameters.
- **SAP Graph.** The generated SAP graph representing the accelerator architecture.
- **Performance Report.** Modeled performance metrics for the generated accelerator.

D.3 Availability and Installation

The artifact is permanently archived on Zenodo and is available at <https://doi.org/10.5281/zenodo.23085661>. The artifact can be downloaded directly from Zenodo. A corresponding GitHub release is also provided and matches the archived version:

```
$ git clone --branch ae \
https://github.com/XiaoyeAsZ/accelgen.git
```

We provide two Docker-based methods for installing and running TranSAGE.

Using the Prebuilt Docker Image (Recommended).

Pull the prebuilt image and launch a container with the repository mounted as the working directory.

```
$ cd accelgen
$ docker pull xiaoyeasz/accelgen:latest
$ docker run --gpus all -it --rm \
  -v $(pwd):/home/accelgen \
  -w /home/accelgen \
  xiaoyeasz/accelgen:latest
```

Building from the Dockerfile. Alternatively, build the image locally and launch a container using the same repository mount.

```
$ cd accelgen
$ docker build -f accelgen.dockerfile -t accelgen:latest .
$ docker run --gpus all -it --rm \
```

```
-v $(pwd):/workspace/accelgen \
-w /workspace/accelgen \
accelgen:latest
```

D.4 Experimental Workflow

The workflow consists of five stages that can be run separately to inspect and reuse intermediate outputs. Each stage requires the output of the preceding stage. Two evaluation modes are available: `rapid` for a quick evaluation of the block 0 FFN workload in Llama3-8B, and `full` for the complete evaluation suite. The full mode can take substantially longer, particularly during kernel scheduling.

Inside the container (accelgen folder), build the compiler and enter the artifact evaluation directory:

```
$ cmake -S . -B build -G Ninja
$ cmake --build build
$ cd ae
```

In the commands below, replace `MODE` with `rapid` or `full`. All output directories are relative to `ae/results/MODE/` in the repository.

1. Generate MLIR.

```
$ bash MODE/01_generate_mlir.sh
```

The script invokes `benchmark/dump_mlir.py` for each selected combination of model, action, block, layer, batch size, and sequence length. The generated MLIR files are written to `mlir/`.

2. Optimize Linalg.

```
$ bash MODE/02_linalg_opt.sh
```

The optimizer performs mixed-precision resolution, named-operation generalization, generic marking, dead-operation elimination, generic fusion, and tensor-operation folding. The optimized MLIR files are written to `generic/`.

3. Schedule Kernels.

```
$ bash MODE/03_kernel_schedule.sh
```

The `func.func(kernel-schedule{...})` pass searches over tiling, loop, unrolling, and cluster choices using the selected edge or server resource configuration. The scheduled MLIR files are written to `scheduled/`. By default, each cluster contains at most six operations (`MAX_OPERATION=6`).

4. Construct SAP graph.

```
$ bash MODE/04_lower_to_dap.sh
```

The `convert-to-dap{config-path=...}` pass lowers each scheduled workload to the hardware-facing DAP dialect. The resulting MLIR files are written to

dap/ and contain operations such as `dap.sram`, `dap.data_path`, arithmetic operations, and routing operations.

5. **Model Performance.**

```
$ bash MODE/05_model_performance.sh
```

The `model-performance{config-path=...}` pass estimates latency, energy, and FLOPs and emits annotated MLIR files in `performance/`. The accompanying `logs/performance.log` contains workload headers and metric records suitable for processing with the repository's analysis scripts.

References

- [1] 2019. Wafer-Scale Deep Learning. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–31. doi:10.1109/HOTCHIPS.2019.8875628
- [2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. 2020. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 145–158. doi:10.1109/ISCA45697.2020.00023
- [3] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. doi:10.1109/MICRO.2016.7783725
- [4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzyniak, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. 1212–1221. doi:10.1145/2228360.2228584
- [5] Suhail Basalama and Jason Cong. 2025. Stream-HLS: Towards Automatic Dataflow Acceleration. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Monterey, CA, USA) (FPGA '25)*. Association for Computing Machinery, New York, NY, USA, 103–114. doi:10.1145/3706628.3708878
- [6] Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Joseph Dabis, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Tomas Jackson, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Isabel Leal, Kuang-Huei Lee, Sergey Levine, Yao Lu, Utsav Malla, Deeksha Manjunath, Igor Mordatch, Ofir Nachum, Carolina Parada, Jodilyn Peralta, Emily Perez, Karl Pertsch, Jornell Quiambao, Kanishk Rao, Michael Ryoo, Grecia Salazar, Pannag Sanketi, Kevin Sayed, Jaspiar Singh, Sumedh Sontakke, Austin Stone, Clayton Tan, Huong Tran, Vincent Vanhoucke, Steve Vega, Quan Vuong, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Tianhe Yu, and Brianna Zitkovich. 2023. RT-1: Robotics Transformer for Real-World Control at Scale. arXiv:2212.06817 [cs.RO] <https://arxiv.org/abs/2212.06817>
- [7] Jingwei Cai, Yuchen Wei, Zuo Tong Wu, Sen Peng, and Kaisheng Ma. 2023. Inter-layer Scheduling Space Definition and Exploration for Tiled Accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (Orlando, FL, USA) (ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 13, 17 pages. doi:10.1145/3579371.3589048
- [8] Chia-Wei Chang, Jing-Jia Liou, Chih-Tsun Huang, Wei-Chung Hsu, and Juin-Ming Lu. 2023. MultiFuse: Efficient Cross Layer Fusion for DNN Accelerators with Multi-level Memory Hierarchy. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. 614–622. doi:10.1109/ICCD58817.2023.00097
- [9] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. doi:10.1109/JSSC.2016.2616357
- [10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. doi:10.18653/v1/N19-1243
- [11] Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Hao Yu, Yida Wang, and Gennady Pekhimenko. 2025. Tilus: A Tile-Level GPGPU Programming Language for Low-Precision Computation. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (Pittsburgh, PA, USA) (ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 281–297. doi:10.1145/3760250.3762219
- [12] Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. 2023. PaLM-E: An Embodied Multimodal Language Model. arXiv:2303.03378 [cs.LG] <https://arxiv.org/abs/2303.03378>
- [13] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: shifting vision processing closer to the sensor. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 92–104. doi:10.1145/2872887.2750389
- [14] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 769–774. doi:10.1109/DAC18074.2021.9586216
- [15] Yu Gong, Lingyi Huang, Haodong Chang, Rongjian Liang, Cheng Yang, Zhexiong Tang, Jiang Hu, and Bo Yuan. 2025. Crane: Inter-Layer Scheduling Framework for DNN Inference and Training Co-Support on Tiled Architecture. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 1250–1263. doi:10.1145/3725843.3756023
- [16] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] <https://arxiv.org/abs/2407.21783>
- [17] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising Diffusion Probabilistic Models. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 6840–6851. https://proceedings.neurips.cc/paper_files/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Paper.pdf
- [18] Jonathan Ho, Tim Salimans, Alexey Gritsenko, William Chan, Mohammad Norouzi, and David J Fleet. 2022. Video Diffusion Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 8633–8646. https://proceedings.neurips.cc/paper_files/paper/2022/file/39235c5aef13fb05a6adc95eb9d8d66-Paper-Conference.pdf
- [19] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjolstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 710–726. doi:10.1145/3582016.3582051
- [20] Huawei. 2026. TorchNPU Documentation. <https://www.hiascend.com/document/detail/zh/Pytorch/2610/index/index.html>. Accessed: 2026-08-30.
- [21] Hugging Face. 2026. Transformers Documentation. <https://huggingface.co/docs/transformers/main/en/index>. Accessed: 2026-08-30.
- [22] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. 2023. Automatic Generation of Spatial Accelerator for Tensor Algebra. *IEEE*

- Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 6 (2023), 1898–1911. doi:10.1109/TCAD.2022.3209949
- [23] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuoyuan Liu, Kyle Lucke, Alan Lundin, Gordon Mackean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagaran, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) (ISCA '17). Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3079856.3080246
- [24] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG] <https://arxiv.org/abs/2001.08361>
- [25] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszal, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: a language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) (PLDI 2018). Association for Computing Machinery, New York, NY, USA, 296–311. doi:10.1145/3192366.3192379
- [26] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. 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* (Columbus, OH, USA) (MICRO-52). Association for Computing Machinery, New York, NY, USA, 754–768. doi:10.1145/3352460.3358252
- [27] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. 2023. BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 19730–19742. <https://proceedings.mlr.press/v202/li23q.html>
- [28] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. 2021. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 789–801. doi:10.1109/HPCA51647.2021.00071
- [29] Yujun Lin, Zhekai Zhang, and Song Han. 2025. LEGO: Spatial Accelerator Generation and Optimization for Tensor Applications. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1335–1347. doi:10.1109/HPCA61900.2025.00101
- [30] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. 2021. TENET: A Framework for Modeling Tensor Dataflow Based on Relation-centric Notation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 720–733. doi:10.1109/ISCA52012.2021.00062
- [31] Andrew Makhurin. 2024. GLPK (GNU Linear Programming Kit). <https://www.gnu.org/software/glpk/>. Accessed: 2026-04-16.
- [32] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Ritvik Sharma, Clark Barrett, Mark A. Horowitz, Pat Hanrahan, and Priyanka Raina. 2023. APEX: A Framework for Automated Processing Element Design Space Exploration using Frequent Subgraph Analysis. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 33–45. doi:10.1145/3582016.3582070
- [33] Nandeeeka Nayak, Xinrui Wu, Toluwanimi O. Odemuyiwa, Michael Pellauer, Joel S. Emer, and Christopher W. Fletcher. 2024. FuseMax: Leveraging Extended Einsums to Optimize Attention Accelerator Design. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1458–1473. doi:10.1109/MICRO61859.2024.00107
- [34] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) (PLDI 2021). Association for Computing Machinery, New York, NY, USA, 883–898. doi:10.1145/3453483.3454083
- [35] NVIDIA. 2026. NVIDIA A100 Tensor Core GPU. <https://www.nvidia.com/en-us/data-center/a100/>. Accessed: 2026-08-30.
- [36] NVIDIA. 2026. NVIDIA Jetson Orin. <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/>. Accessed: 2026-08-30.
- [37] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Bruce Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. doi:10.1109/ISPASS.2019.00042
- [38] William Peebles and Saining Xie. 2023. Scalable Diffusion Models with Transformers. arXiv:2212.09748 [cs.CV] <https://arxiv.org/abs/2212.09748>
- [39] PyTorch. 2026. PyTorch Documentation. <https://docs.pytorch.org/docs/>. Accessed: 2026-08-30.
- [40] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 8748–8763. <https://proceedings.mlr.press/v139/radford21a.html>
- [41] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019). <https://openai.com/blog/better-language-models/>
- [42] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-Resolution Image Synthesis with Latent Diffusion Models. arXiv:2112.10752 [cs.CV] <https://arxiv.org/abs/2112.10752>
- [43] Satyabrata Sarangi and Bevan Baas. 2021. DeepScaleTool: A Tool for the Accurate Estimation of Technology Scaling in the Deep-Submicron Era. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5. doi:10.1109/iscas51556.2021.9401196
- [44] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Bruce Khailany, and Stephen W. Keckler. 2019. Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) (MICRO-52). Association for Computing

- Machinery, New York, NY, USA, 14–27. doi:10.1145/3352460.3358302
- [45] Significant Gravitass. [n.d.]. *AutoGPT*. <https://github.com/Significant-Gravitass/AutoGPT>
- [46] Arne Symons, Linyan Mei, Steven Colleman, Pouya Houshmand, Sebastian Karl, and Marian Verhelst. 2025. Stream: Design Space Exploration of Layer-Fused DNNs on Heterogeneous Dataflow Accelerators. *IEEE Trans. Comput.* 74, 1 (2025), 237–249. doi:10.1109/TC.2024.3477938
- [47] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botet, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikula, Mateo Wirth, Michael Sharman, Nikolai Chirnaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruiho Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. 2024. Gemma: Open Models Based on Gemini Research and Technology. arXiv:2403.08295 [cs.CL] <https://arxiv.org/abs/2403.08295>
- [48] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] <https://arxiv.org/abs/2505.09388>
- [49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
- [50] Huizheng Wang, Jiahao Fang, Xinru Tang, Zhiheng Yue, Jinxi Li, Yubin Qin, Sihan Guan, Qinzhe Yang, Yang Wang, Chao Li, Yang Hu, and Shouyi Yin. 2024. SOFA: A Compute-Memory Optimized Sparsity Accelerator via Cross-Stage Coordinated Tiling. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1247–1263. doi:10.1109/MICRO61859.2024.00093
- [51] Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Virtual Event, USA) (FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 93–104. doi:10.1145/3431920.3439292
- [52] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf
- [53] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: synthesizing programmable spatial accelerators. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (Virtual Event) (ISCA '20)*. IEEE Press, 268–281. doi:10.1109/ISCA45697.2020.00032
- [54] Lei Xu, Chen Yin, Zelong Yuan, Weiguang Sheng, Jianfei Jiang, Qin Wang, and Naifeng Jing. 2025. Principle-based Dataflow Optimization for Communication Lower Bound in Operator-Fused Tensor Accelerator. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. 1–7. doi:10.1109/DAC63849.2025.11132765
- [55] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] <https://arxiv.org/abs/2505.09388>
- [56] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] <https://arxiv.org/abs/2210.03629>
- [57] Hanchen Ye and Deming Chen. 2025. StreamTensor: Make Tensors Stream in Dataflow Accelerators for LLMs. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 201–216. doi:10.1145/3725843.3762817
- [58] Linxuan Zhang, J. Nelson Amaral, and Di Niu. 2025. TransFusion: End-to-End Transformer Acceleration via Graph Fusion and Pipelining. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 1491–1504. doi:10.1145/3725843.376105
- [59] Weichuang Zhang, Yiquan Wang, Xinzhou Zhang, Chi Zhang, Yu Feng, Xiaofeng Hou, Chao Li, Jieru Zhao, and Minyi Guo. 2026. CODO: An Automated Compiler for Comprehensive Dataflow Optimization. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. 1–17. doi:10.1109/ISCA66397.2026.00018
- [60] Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, and Yun Liang. 2023. TileFlow: A Framework for Modeling Fusion Dataflow via Tree-based Analysis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (Toronto, ON, Canada) (MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1271–1288. doi:10.1145/3613424.3623792
- [61] Xilang Zhou, Shuyang Li, Haodong Lu, and Kun Wang. 2024. PipeFuser: Building Flexible Pipeline Architecture for DNN Accelerators via Layer Fusion. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 921–926. doi:10.1109/ASP-DAC58780.2024.10473790