Streaming Tensor Programs: A Streaming Abstraction for Dynamic Parallelism

Gina Sohn
Stanford University
Stanford, CA, USA
ginasohn@stanford.edu

Genghan Zhang
Stanford University
Stanford, CA, USA
zgh23@stanford.edu

Konstantin Hossfeld
Stanford University
Stanford, CA, USA
hossfeld@stanford.edu

Jungwoo Kim
Stanford University
Stanford, CA, USA
jungwkim@stanford.edu

Nathan Sobotka
Stanford University
Stanford, CA, USA
nsobotka@stanford.edu

Nathan Zhang
SambaNova Systems
Palo Alto, CA, USA
stanfurd@stanford.edu

Olivia Hsu
Stanford University
Stanford, CA, USA
Carnegie Mellon University
Pittsburgh, PA, USA
owhsu@stanford.edu

Kunle Olukotun
Stanford University
Stanford, CA, USA
kunle@stanford.edu

Abstract

Dynamic behaviors are becoming prevalent in tensor applications, like machine learning, where many widely used models contain data-dependent tensor shapes and control flow. However, the limited expressiveness of prior programming abstractions for spatial dataflow accelerators (SDAs) forces these dynamic behaviors to be implemented statically and/or unoptimized. To address these challenges, we present Streaming Tensor Programs (STeP), a streaming abstraction that enables dynamic tensor workloads to run efficiently on SDAs. STeP introduces flexible routing operators, an explicit memory hierarchy, and symbolic-shape semantics that expose dynamic data rates and tensor dimensions. These capabilities unlock new optimizations, like dynamic tiling, dynamic parallelization, and configuration time-multiplexing, that adapt SDA execution to dynamic behaviors while preserving dataflow efficiency. Using a cycle-approximate simulator on representative LLM layers and a full model with real-world traces, STeP enables: dynamic tiling that breaks the Pareto-optimal frontier from prior work, dynamic parallelization that improves latency by ~2.72x, and configuration time-multiplexing that increases compute utilization by ~2.64x over prior SDA abstractions and their implementations.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790229

CCS Concepts: • **Computer systems organization** → **Data flow architectures**; • **Theory of computation** → **Streaming models**; *Abstract machines.*

Keywords: Streaming Abstraction; Dataflow Programming; Spatial Dataflow Accelerator; Dynamic Tensor Applications

ACM Reference Format:
Gina Sohn, Genghan Zhang, Konstantin Hossfeld, Jungwoo Kim, Nathan Sobotka, Nathan Zhang, Olivia Hsu, and Kunle Olukotun. 2026. Streaming Tensor Programs: A Streaming Abstraction for Dynamic Parallelism. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3779212.3790229

1 Introduction

The widespread use of compute- and memory-intensive tensor applications has increased the demand for performant hardware backends. This need for performance now drives the widespread adoption of high-throughput, highly parallel machines (like GPUs and dataflow architectures) for many tensor workloads, particularly large language models (LLMs) [21, 37, 41, 57]. Under such circumstances, Spatial Dataflow Accelerators (SDAs) [12, 20, 33–35, 38] are emerging as a promising hardware architecture. SDAs are reconfigurable architectures composed of spatially distributed compute and memory units. By mapping computation onto a spatial fabric of pipelined compute and memory units, SDAs avoid several control overheads and enable aggressive operator fusion, pipelining, and fine-grained parallelism. Prior work has empirically demonstrated that these architectural

Abstraction	Data Flow	Explicit Data Rate	Explicit Memory Hierarchy	Dynamic Routing & Merging	Dynamic On-chip Tiling
Spatial [18]	✗	✗	✓	✗	✗
Revet [38]	✗	✗	✓	✓ (limited)	✗
StreamIt [46]	✓	✓	✗	✗	✗
SAM [15]	✓	✗	✗	✓ (limited)	✓ (limited)
Ripple [11]	✓	✗	✗	✓	✗
STeP	✓	✓	✓	✓	✓

Table 1. Landscape of programming abstractions for SDAs

features enable SDAs to outperform state-of-the-art GPUs while delivering higher energy efficiency [5, 19, 39].

However, unlike the ample software support for static workloads on SDAs, existing SDA programming abstractions have limited support for accelerating dynamic workloads, as shown in Table 1. The importance of supporting dynamic behaviors is increasing in many widely used tensor applications due to data-dependent tensor dimensions [31, 32, 42, 50] and control flow [8, 27, 52]. Many such dynamic workloads can be characterized as asynchronously executing blocks with corresponding communication. This characterization aligns well with the execution model of SDAs, where compute and memory units run asynchronously and communicate via hardware FIFOs. As such, current SDA abstractions leave performance on the table even though the SDA hardware itself naturally maps well to these workloads.

Most existing SDA programming abstractions fall into either imperative or dataflow-based designs. While imperative abstractions [12, 18, 38, 56] offer high generality, dataflow designs, such as StreamIt [46], SAM [15], and Ripple [11], have emerged as they align better with the hardware's execution model. However, they do not model an explicit memory hierarchy, and many were designed for a specific domain, limiting their ability to capture the broader range of dynamic tensor workloads. SAM is limited to sparse tensor algebra kernels, and StreamIt adopts a synchronous dataflow model, making it challenging to express dynamic behaviors. Ripple adopts a design based on asynchronous blocks that can contain arbitrary imperative code. However, Ripple leaves the memory hierarchy implicit, making it difficult to express and discover efficient implementations of many important applications whose performance is dominated by data movement across the memory hierarchy [1, 8, 27, 43, 47, 52]. Furthermore, opaque data rates at the abstraction level require lifting the imperative code within each asynchronous block to analyze the program in terms of data rates.

To address the limitations of prior SDA dataflow abstractions in expressing and optimizing dynamism, we propose Streaming Tensor Programs (STeP), a new streaming abstraction for accelerating dynamic tensor applications on SDAs. STeP expresses data as streams, where tiles and buffers in the stream can have dynamic shapes. It consists of asynchronous dataflow blocks that provide three key properties: explicit memory hierarchy, symbolic data consumption and production rate, and data-dependent control flow operators.

These properties give STeP unique capabilities that are unavailable in prior abstractions for SDAs. First, STeP captures performance-critical metrics such as off-chip traffic, on-chip memory requirement, and operational intensity at the abstraction level. We show how STeP provides insight into memory-bound tensor applications and validate the captured metrics with a cycle-accurate simulator (Section 4). STeP also enables expressing optimizations such as dynamic tiling, configuration time-multiplexing, and dynamic parallelization (Section 5), which are not expressible in prior abstractions for SDAs. We evaluate each optimization on representative layers from open-source LLMs with real-world traces using a cycle-approximate simulator. Our evaluation shows that these optimizations break the Pareto-optimal frontier from prior work by delivering speedups and/or resource savings. Specifically, dynamic tiling delivers a Pareto Improvement Distance [9, 26, 51][1] of 1.33×~2.11×; Configuration time-multiplexing delivers 2.51×~2.64× higher compute utilization; Dynamic parallelization achieves 1.14×~2.72× speedup. We also evaluate the optimizations on end-to-end models, achieving upto 1.27× speedup while using 69% less on-chip memory and 54% fewer compute resources on Qwen3-30B-A3B. Lastly, we discuss future compilation to STeP and approaches for supporting the dynamic features of STeP in SDA hardware (Section 6).

Overall, our contributions are:

- An asynchronous dataflow abstraction for SDAs (STeP) with first-class support for dynamism (Section 3).
- A symbolic system based on STeP's shape semantics to extract performance-critical metrics (Section 4).
- Optimizations that exploit the dynamic features and explicit memory hierarchy of STeP (Section 5) and an outline of how those abstract dynamic features would be supported in SDA hardware (Section 6).
- A performance and resource utilization investigation on the impact of dynamic optimizations enabled by STeP on representative LLM applications (Section 5).

2 Background

This section provides background on the application, hardware, and programming abstractions discussed in this paper.

2.1 Dynamism in Machine Learning (ML)

Although dynamism appears in many tensor applications, we will use ML workloads to illustrate real-world examples of dynamic behavior throughout this paper. Modern ML models exhibit diverse forms of dynamism and represent one of the

[1]The Pareto Improvement Distance measures the distance from a new design point p to a reference Pareto frontier P. For more detail, see Section 5.2.

most widely used tensor applications. ML workloads also demand high-throughput hardware backends, making them a primary driver for accelerators.

A prominent source of dynamism in recent ML workloads is the heavy use of data-dependent control flow. Mixture-of-Experts (MoE) is a model architecture where a subset of parameters, called experts, are activated for each input activation. With every top-ranked open-source model now adopting the MoE architecture [6, 13, 23, 23, 43–45, 52],[2] efficiently handling such control flow has become increasingly important. ML workloads also frequently exhibit dynamic/ragged tensor shapes, driven by runtime parameters such as the number of requests, input resolution, and sequence length [32, 42, 50, 58]. Data-dependent control flow further amplifies this by making expert input shapes data-dependent.

2.2 Spatial Dataflow Accelerators

Spatial dataflow accelerators [12, 20, 35, 38, 48, 49] are programmable architectures with spatially distributed hardware resources. A typical SDA consists of an array of reconfigurable compute units and memory units that communicate via hardware FIFOs and a network-on-chip. Instead of executing a sequential instruction stream as in the von Neumann model, SDAs represent programs as dataflow graphs, where nodes denote operations and edges represent explicit data dependencies. The nodes in a dataflow program graph are mapped to distributed compute and memory units, and the edges are mapped to hardware FIFOs and network-on-chip. The storage in SDAs is organized into multiple tiers, such as local PE storage, on-chip memory units, and off-chip memory. Most SDAs rely on the compiler or runtime to explicitly orchestrate and schedule the data movement from one storage tier to another [14, 18].

These architectural features allow SDAs to avoid the instruction decode, cache hierarchy, and control-flow divergence overheads of general-purpose processors and GPUs. Their dataflow execution model further enables aggressive operator fusion, pipelining, and fine-grained parallelism, which reduces off-chip memory traffic and synchronization overhead. Prior work shows that these advantages translate into both higher performance and energy efficiency compared to state-of-the-art GPUs [5, 19, 39]. For example, GPUs utilize less than half of their peak HBM bandwidth on Llama-3.1-8B and Llama-3.1-70B workloads (Figure 1), whereas the SN40L [33]—a recent commodity SDA—achieves a higher fraction of peak HBM bandwidth during token generation. As this phase is heavily memory-bound, SN40L attains up to a 2× speedup with half the peak HBM bandwidth (SN40L-8) and a 3.7× speedup with comparable bandwidth (SN40L-16) [5, 19]. Beyond throughput, SDAs also deliver higher task-level energy efficiency: the SN40L achieves 3.8× and 4.6× higher *Intelligence per Joule* than NVIDIA B200 GPUs for Qwen3-32B and GPT-OSS-120B, respectively [39].

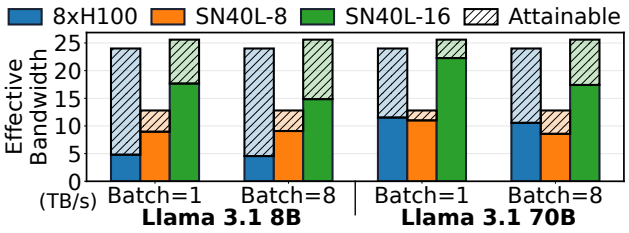

Figure 1. Comparison of SDAs versus GPUs. We express the effective bandwidth of each platform as solid bars, with slashed background bars indicating its peak HBM bandwidth. Effective bandwidth is calculated using Roofline modeling and the percentage of peak throughput reported by [19]. [3]

2.3 Programming Abstractions for SDAs

SDAs can be programmed with either an imperative [12, 18, 38] or a dataflow programming abstraction [11, 15, 46] as listed in Table 1. While imperative abstractions offer high generality, they enforce a sequential instruction order, which makes it challenging to exploit the inherent parallelism in the application [56]. Furthermore, they lack explicit primitives for asynchronous execution or queueing, which are crucial for optimizing dynamic workloads.

Spatial [18] is an imperative programming abstraction for FPGAs and SDAs. It uses nested loops and provides explicit control over the memory hierarchy. However, control flow is only permitted in restricted regions of the program, and all memory constructs must be statically sized. Furthermore, transforming imperative loops into dataflow graphs that can be mapped to hardware introduces complexity in the compiler [56], and potentially results in suboptimal schedules.

Revet [38] is an imperative programming abstraction and compiler for expressing irregular applications on SDAs. It supports more flexible data-dependent control flow than Spatial via new dynamic primitives. However, its Dataflow Thread model restricts these primitives to scalars, which limits data reuse and prevents vectorized or tiled computation. As a result, many large, memory-bound tensor applications in Revet are forced to use only static primitives to achieve high performance. Revet also cannot dynamically group scalar streams into dynamically sized tiles, making it unable to express optimizations that combine data-dependent control flow with dynamically-sized tiles.

Dataflow abstractions address these limitations with built-in support for dataflow and queueing. However, prior work

[2] According to https://lmarena.ai/leaderboard, accessed on Nov. 12, 2025.

[3] All models use a sequence length of 4K. GPU numbers were obtained by executing the models using TensorRT-LLM. The figure is reproduced with numbers from prior work with the original authors' permission [19].

either focuses only on a specific domain or lacks visibility and control over performance-critical decisions in many dynamic tensor applications. Throughout the paper, *asynchronous dataflow* refers to an execution model in which dataflow blocks execute without global synchronization, and each block may exhibit dynamic data rates and latencies.

StreamIt [46] is a synchronous dataflow abstraction used to map stream applications. It is not an abstraction dedicated to SDAs and can be used to target various streaming backends. Each node in the program graph has a fixed rate for consuming and producing data in the stream. While this design enables optimizations based on known data rates, this limits its ability to capture dynamic applications.

SAM [15] is the first asynchronous streaming tensor abstraction for SDAs. It introduces a clean dataflow model with primitives that can express the full space of sparse tensor algebra computations as streaming dataflow graphs. However, SAM is limited to sparse tensor operators, making it well-suited for exploring sparse workloads but not for dense dynamic tensor applications.

Ripple [11] is an asynchronous dataflow abstraction and architecture that expresses the asynchronous pipeline parallelism enabled by SDAs. It has an implicit memory hierarchy and offers high generality by representing programs as asynchronous blocks that can contain any imperative code. While this model is sufficient for graph analytics and sparse workloads with inherently low reuse, dynamic tensor applications, such as dense ML, exhibit high temporal and spatial reuse. Therefore, visibility and explicit control over data movement across the memory hierarchy are crucial for performance; the lack of such control makes it challenging to analyze and express efficient schedules. Furthermore, its design makes the data rates of each asynchronous block opaque, requiring the compiler to infer them from the imperative code.

3 Streaming Tensor Programs

Streaming Tensor Programs (STeP) is a streaming abstraction for dynamic applications running on SDAs. In this section, we describe its stream representation and operators.

3.1 Stream-centric Design

As an asynchronous dataflow model, STeP uses streams as the primary representation for data. Each stream has a compile-time determined rank and data type.

Data Type. The data type of a stream can either be a tile, a selector, a reference to on-chip memory, or a tuple of these data types. A tile is a two-dimensional regular matrix. STeP allows tiles to have dynamically defined shapes. Supporting dynamically-sized tiles is crucial for maximizing data reuse without excessive on-chip memory requirements when tiling tensors with runtime-determined shapes. A selector is a multi-hot vector, which can express various routing and merging operators to support control flow (Section 3.2.3). STeP also enables read-only reference (i.e. addresses) to on-chip memory as a stream data type (Section 3.2.2). The flexibility in data type enables lowering STeP to a broader range of SDAs more easily. For example, when the stream data type is restricted to only scalars, it cannot be directly mapped to SDAs with tiled processing units like systolic arrays without complicated lifting (e.g. auto-vectorization).

Stop Tokens. STeP streams are logically equivalent to zero or more tensors. STeP streams embed the logical structure of the corresponding tensor into the data stream through *stop tokens*. STeP uses a similar stop token design to that of SAM [15] as it was designed for asynchronous dataflow abstractions and allows for dimensions to be dynamic. The end of each dimension of the corresponding tensor is annotated with a stop token $S_N (N \geq 1)$, where N denotes the rank of that dimension (e.g. $N = 1$ denotes the end of a vector) and at the end of multiple dimensions, STeP only emits the highest-level stop token. The *Done* token (D) at the end indicates stream termination.

Stream Shape. The logical correspondence between a tensor and a STeP stream provides a foundation for defining shape semantics for streams. These semantics enable analyses and optimizations and improve debuggability by exposing dataflow block behaviors at the tensor level. Unlike the shape semantics of streams in synchronous dataflow [46], which are straightforward due to fixed data rates, the shape semantics in asynchronous dataflow require careful design.

Each STeP stream has a *rank* which is determined by the dimensionality of the corresponding tensor(s) in the stream. A rank-N stream with a data type T is a stream of zero or more N-dimensional tensors of T and has a shape $[D_N, \cdots, D_1, D_0]$. Throughout the remainder of this section, we will use red, gray, and black to denote the shape of the stream, the stream's data type, and the tensor, respectively. STeP allows each D_i to be either a static-regular, a dynamic-regular, or a ragged dimension. A dimension is *dynamic-regular* if its shape is a data-dependent constant. A dimension is *ragged* if its shape can be various values (e.g. inner-most dimension in example 1). A ragged dimension may be either dynamic or static, depending on whether its set of values is data-dependent. We will refer to either dynamic-regular or dynamic-ragged dimensions as *dynamic dimensions*. The shape of dynamic-regular and ragged dimensions are expressed as equations and symbols (e.g. D_0 in example 1).

$$\xleftarrow{1, 2, S_1,\ 3, S_2,\quad 4, S_1,\ 5, 6, 7, S_2,\ D} \quad \text{Shape: } [2, 2, D_0] \tag{1}$$

Ragged dimensions have an absorbing property in the equations. If a dimension's shape equation contains a ragged dimension, that dimension will be treated as a new ragged dimension. For instance, when the inner-two dimensions are flattened in example (1), the resulting stream shape is $[2, D_0']$

instead of $[2, 2 \times D_0]$ where D_0' is a newly introduced symbol for the new ragged dimension.

Certain STeP operators have restrictions on stream or data type shapes. Regular dimensions are more constrained than ragged dimensions, and static dimensions are more constrained than dynamic dimensions.[4] Thus, if the operator accepts a dimension of a certain type, it also accepts more restrictive dimension types.

3.2 STeP Operators

STeP's operators fall into five categories: (1) off-chip memory operators that stream tiled tensors between off-chip and on-chip memory; (2) on-chip memory operators that convert between streams and on-chip buffers; (3) dynamic routing and merging operators that implement data-dependent control flow; (4) higher-order operators that apply functions over stream elements; and (5) shape operators that modify stop tokens to change the stream's logical tensor structure.[5]

3.2.1 Off-chip Memory Operators.

Off-chip memory operators express the interface between on-chip and off-chip memory. Coupled with shape semantics, the off-chip memory operators can capture metrics such as off-chip memory traffic and operational intensity, exposing performance-critical metrics to the programmer or compiler.

LinearOffChipLoad As shown in Figure 2, this operator loads an input tensor with a specific shape (*in_mem_shape*) from off-chip memory to on-chip memory in tiles. It supports affine reads of the stored tensor using the stride and the shape arguments. The operator takes in a reference stream that controls how the stored tensor is repeatedly read. The reference stream's shape can contain any type of dimensions (static/dynamic-regular, and ragged). For each element in the reference stream, an affine read over the tensor is triggered (the correspondence is shown with black bold lines in Figure 2). Because the reference stream serves as a trigger, its contents do not matter.

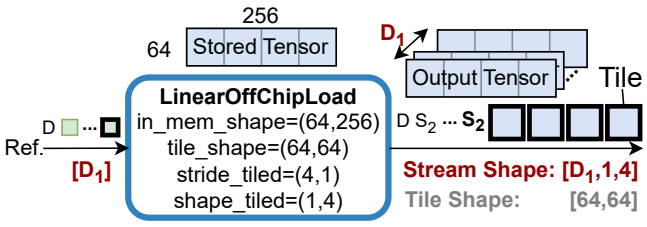

Figure 2. An example of a LinearOffChipLoad operator. The stored tensor is read in a row-major order D_1 times, where D_1 is the shape of a dynamic dimension. The stride and shape are expressed in terms of tiles. Therefore, the output stream shape is $[D_1, 64//64, 256//64] = [D_1, 1, 4]$.

[4]Regularity and data-dependence are orthogonal.
[5]Supplementary syntax and shape semantics are provided in the Appendix.

LinearOffChipStore linearly stores the input stream's tiles to off-chip memory at the given address.

RandomOffChipLoad & RandomOffChipStore support random access to tensors stored in off-chip memory. Both take the base address, tile shape, and the in-memory tensor shape as arguments. As inputs, the RandomOffChipLoad operator has a read address stream, and the RandomOffChipStore has a write address stream and write data stream.

3.2.2 On-chip Memory Operators.

These operators allow programs to leverage on-chip scratchpads and avoid off-chip memory accesses or recomputation. This expressiveness exposes a large design space of implementations that trade off on-chip memory usage against off-chip traffic.

Bufferize stores portions of the stream to on-chip memory in linear order and outputs a stream of *buffers* (a read-only reference to the allocated on-chip memory). The stream of buffers created by Bufferize becomes the input to Streamify, where any control-flow operator or shape operator (except Reshape) can be inserted between the two. The amount of data to be stored in on-chip memory is determined by the *bufferize rank* argument. As shown in Figure 3, STeP allows the bufferized inner dimensions to be dynamic-regular dimensions and the outermost bufferize dimension can be a dynamic-ragged dimension.

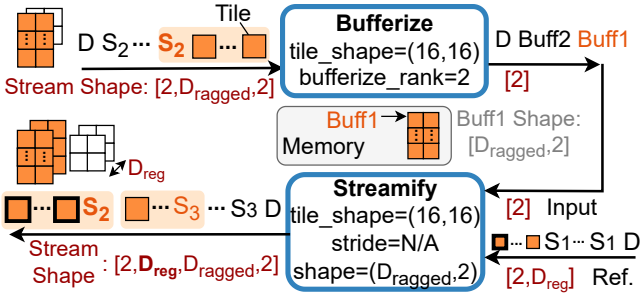

Figure 3. Bufferize stores the input stream tiles to on-chip memory until it sees a stop token larger than or equal to the bufferize rank. Then, a buffer is enqueued to the output stream, and the operator begins accumulating into a new on-chip memory (location).

Streamify supports reading data stored in on-chip memory by a dynamic number of times using a reference stream as shown in Figure 3. When the buffer shape contains only static-regular dimensions, it supports affine reads over the buffer using its stride and shape arguments (similar to LinearOffChipLoad); otherwise, it linearly streams the tensor referenced by each buffer.

3.2.3 Dynamic Routing and Merging Operators.

These operators capture the essential routing and merging patterns to express data-dependent control flow and dynamic parallelism efficiently. STeP's dynamic routing and merging operators allow for flexible data types, while many prior SDA

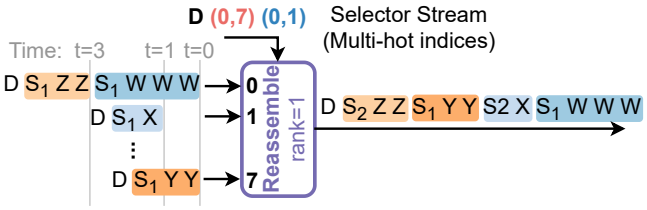

Figure 4. An example of Reassemble. The figure expresses multi-hot vectors in the selector stream as tuples of the indices of the nonzero elements. W-Z are data values.

abstractions [18, 38, 46] either lack support altogether or only support it under specific restrictions that significantly limit available parallelism.

Reassemble merges data from many input streams based on the selector stream, as shown in Figure 4. The input streams must all have the same rank, which is the *reassemble rank* b (In Figure 4, b=1). On every multi-hot vector in the selector stream, data up to the first S_b from the selected input stream is merged to the output stream. When multiple input streams are selected by the multi-hot vector (selector), data is collected in the order the input is available. For the second multi-hot selector $(0, 7)$ in Figure 4, data is collected from input stream 7 first. While routing inputs from one expert to the output stream, input streams don't get interleaved even though the other selected stream becomes available (e.g. in $t = 1$). After collecting data from all selected input streams, the operator adds a new dimension by incrementing the stop token.

EagerMerge is similar to Reassemble except that it collects data in the order they arrive. The operator has two output streams: the data stream and the selector stream, which denotes the index of the input stream from which each chunk of the stream was collected. For the input streams of Figure 4, EagerMerge will output the data stream in $W \rightarrow Y \rightarrow X \rightarrow Z$ or $Y \rightarrow W \rightarrow X \rightarrow Z$ order.

Partition is the inverse of Reassemble and routes data up to the first S_b from the input stream to the selected output streams (b is the *partition rank*).

3.2.4 Higher-order Operators.

These operators take a function supported by the hardware as an argument.

Accum reduces over multiple inner dimensions of a stream. The operator takes the reduction rank, an initialization function, and an update function as arguments. Accum can express higher-order reductions by using an accumulator tile that is larger than the input tile. Similar to Figure 3, the accumulator for Accum can have a dynamic size. Together with Bufferize, this capability enables maximizing data reuse while using minimal on-chip memory when the application involves dynamically-sized tensors (we discuss related optimizations in Section 5.2).

Scan is similar to Accum but emits the state of the accumulator on every input element. Therefore, the input and output streams have the same shape.

Map applies element-wise functions without changing the stream shape.

FlatMap expands each element in the stream to a stream of rank b by applying the supplied function. The resulting streams are concatenated into a single output stream.

3.2.5 Shape Operators.

These operators only modify stop tokens and do not alter the data contents of stream elements.

Flatten takes the indices of two dimensions, which specify the range of dimensions that will be flattened.

Reshape splits a dimension into statically-sized chunks. When splitting the inner-most dimension, the operator takes in a padding value as an argument. The operator has two output streams: the data stream and the padding stream. The padding stream specifies whether each element in the output data stream was padded.

Promote adds a new outermost dimension to the input stream. Given an input stream's outermost dimension D_a, the new outermost dimension is $(1\ if\ (D_a > 0)\ else\ 0)$ to handle cases where the input stream is an empty stream.

Expand repeats each element in the input stream based on the reference stream as shown in Figure 5.

Figure 5. The *expand rank* argument is set to the smallest stop token level of the input stream. The output stream has the same shape as the reference stream.

Zip groups two streams with the same shape into a single stream with a tuple data type.

3.3 Putting it All Together: Simplified MoE

To demonstrate how STeP combines to implement dynamic tensor applications, we will walk through a simplified MoE example. For this example, we use a simplified two-expert MoE layer, where each is a single matrix multiplication. Input rows are dynamically routed to one of the two branches, and their outputs are gathered back together after processing. Figure 6 expresses the computation at the tensor level with tiling, and Figure 7 expresses the corresponding STeP graph. We will explain how the labelled operator regions in Figure 7 relate to the changes in the stream and the datatype shape.

Route: Partition takes in a `[10,64]` tensor, which is streamed as a `[10,1]`-shaped stream of `[1,64]` tiles. The output stream shapes are expressed symbolically as $[D_i, 1]$ since each (i) branch receives a dynamic number of rows.

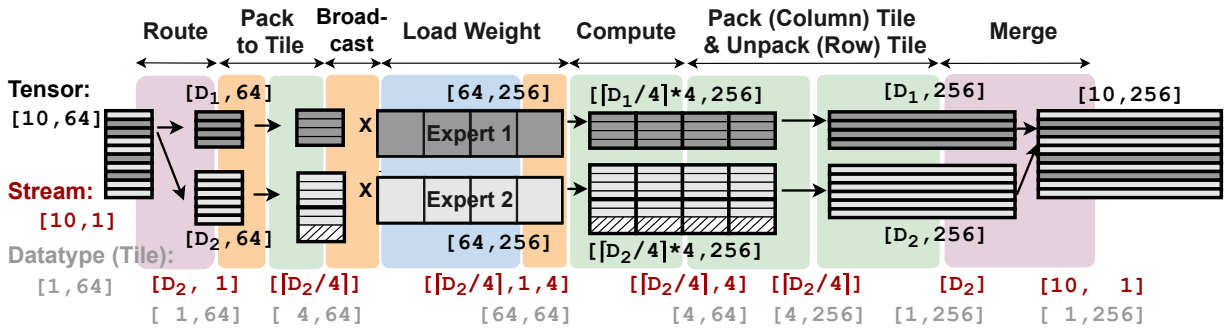

Figure 6. A simplified tensor-level MoE expressed example. Bold black lines mark tile boundaries that are streamed in row-major order. Black, dark red, and gray lists denote the shape of the tensor, stream, and the stream's data type, respectively.

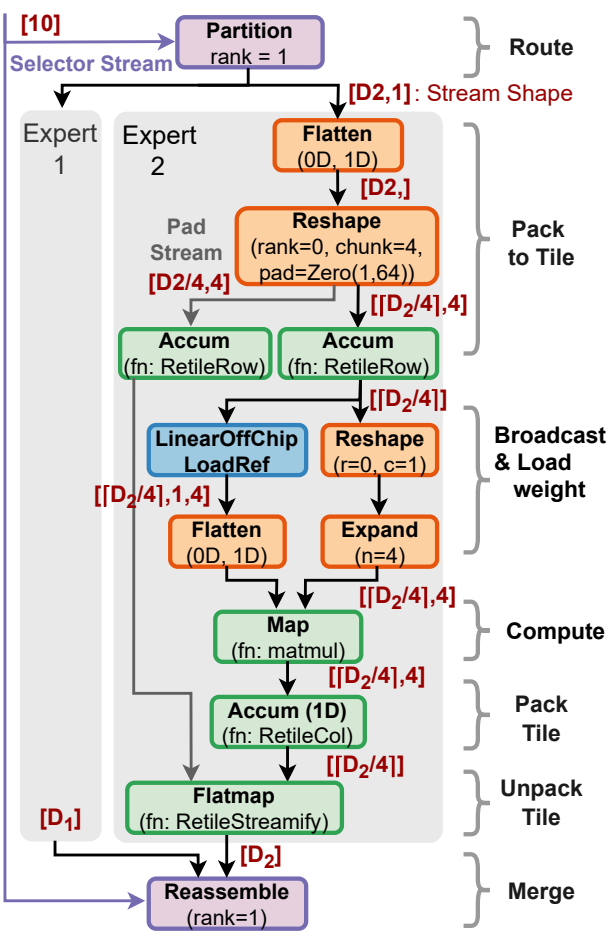

Figure 7. The STeP graph for a simplified MoE. The background colors indicate corresponding regions in Figure 6.

```python
partition = Partition(in_stream, selector,
    rank=1, num_consumers=N_ROUTED_EXPERTS)

expert_streams = []
for i in range(N_ROUTED_EXPERTS):
    flatten_in = Flatten((partition,i),0,1)
    reshape_to_tile = Reshape(...)
    collect_rows = Accum(...)
    collect_masks = Accum(...)
    weight_load = LinearOffChipLoadRef(
        ref=collect_rows,
        underlying=torch.randn(64,256),
        stride=(4,1),
        shape=(1,4),
        tile_row=64, tile_col=256)
    flatten_w = Flatten(...)
    reshape_in = Reshape(...)
    expand_in = Expand(...)
    matmul_map = Map((expand_in, flatten_w),
        map_fn.Matmul(), compute_bw=1024)
    collect_cols = Accum(...)
    retile_streamify = Flatmap(...)
    expert_streams.append(retile_streamify)

output = Reassemble(expert_streams, selector
    , rank=1)
output.stream.shape = in_stream.stream.shape
print(output.stream.shape)
```

Listing 1. The code snippet for Figure 7 written in STeP's symbolic Python frontend.

Pack to Tile: With [1,64] tile shapes, this region packs them into [4,64] tiles to execute matrix-matrix multiplication rather than multiple matrix-vector operations. To do so, Flatten and Reshape first convert the stream shape from $[D_i,1]$ to $[\lceil D_i/4\rceil,4]$. To pack a dynamic number of tiles in a stream into statically defined chunks, the Reshape operator pads the stream with the given pad value, which is [1,64] zero-value tiles. The tiles in the stream are then

packed into a larger tile using the Accum operator with the `RetileRow` function, which concatenates tiles row-wise.

Broadcast: Since the matrix multiplication will be in inner-product dataflow order, each element in the input stream has to be broadcast by the number of tiles in the column dimension of the weight matrix. Therefore, we use Reshape and Expand[6] to do the stream shape conversion: `[⌈Di/4⌉]` → `[⌈Di/4⌉,1]` → `[⌈Di/4⌉,4]`.

Load weight: The weight matrix is tiled along the column dimension and has to be loaded $\lceil D_i/4 \rceil$ times. Since D_i is dynamically determined, we must use a LinearOffChipLoad-Ref and feed the output stream of Accum (shape: `[⌈Di/4⌉]`) to its reference stream. This invokes reading the weight tensor $\lceil D_i/4 \rceil$ times as a `[1,4]` stream of `[64,64]` tiles.

Compute: The matrix multiplication is done using a Map operator as we do not tile the reduction dimension.

Pack Tile, Unpack Tile, & Merge: To merge the streams in `[1,256]` tile granularity, Accum first packs multiple tiles horizontally and then each tile is split row-wise into multiple smaller tiles using a FlatMap with the RetileStreamify function. Lastly, Reassemble gathers `[1,256]` tiles based on a selector stream.

4 Symbolic STeP Frontend and Performance Model

In this section, we describe the programmability of STeP's symbolic frontend and how it combines with our simulator to capture performance-critical metrics. We then present the performance model used in the STeP cycle-approximate simulator. We validate the metrics captured in the symbolic frontend and the simulator against a cycle-accurate hardware description language (HDL) simulation. Lastly, this section contains a discussoin on how the symbolic frontend and simulator can be adapted to target different SDAs.

4.1 Programmability of the Symbolic Frontend

Continuing with the example in Section 3.3, Listing 1 is the equivalent code snippet for STeP's symbolic Python frontend. Writing programs in STeP is similar to writing programs in PyTorch, but with schedules, such as parallelization, tiling, and memory placement. Instead of PyTorch operators, the programmer uses STeP operators, and the result of an operator is a stream instead of a tensor. From the imperative coding perspective, writing programs in STeP means giving each instruction its own asynchronously executing loops connected via streams.

STeP's stream-centric design enables operator fusion by construction and eliminates complicated compiler passes for extracting parallelism from imperative code. However, as tensors between operators are expressed as streams, both the corresponding tensor shape and the stream shape (i.e., how the elements in the tensor are being streamed) have to align between operators. STeP's shape semantics enable the symbolic frontend to internally verify that stream shapes align between the producer and consumer, and allow programmers to inspect the stream shape (see line 27 of Listing 1). Stream shapes can also be used to exploit known program properties. For example, the new dynamic dimension introduced by Reassemble can be substituted with the input stream's shape as shown in line 26 of Listing 1.

STeP provides two sets of memory operators (off-chip and on-chip) with similar interfaces, allowing programmers to select operators based on the desired memory placement. For instance, line 10 of Listing 1 uses off-chip memory to load weights, but if those weights are already resident in on-chip memory, the operator can be replaced with Streamify. Such visibility and control over the memory hierarchy enable analyzing and exploring different tiling schemes, which are a crucial scheduling knob for many tensor applications.

4.2 Analysis with Symbolic Shape Semantics

The symbolic frontend implements symbolic expressions for off-chip memory traffic and on-chip memory requirements for each operator using SymPy [28]. The total off-chip memory traffic and on-chip memory requirement of a program is obtained by summing the expressions for every operator in the program graph. Throughout this subsection, we use $||X||$ to denote the cardinality of a buffer or stream X, defined as the product of its dimension sizes. *dtype* denotes data type, and $|x|$ to denote the size of a data type x.

Off-chip Traffic. As the off-chip memory traffic only occurs in off-chip memory operators (Section 3.2.1) the equation for other operators is zero, and the equation for off-chip memory operators is:

$$||output\ stream|| \times |ouput\ stream\ dtype|$$

If the target SDA assumes that no other STeP operators spill to off-chip memory, the summed off-chip traffic equation represents the program's total off-chip memory traffic and can be used to compute operational intensity. Otherwise, it provides a lower bound on off-chip traffic and thus an upper bound on achievable operational intensity.

On-chip Memory Requirement. In our simulator, we use the following equations for each operator. Other operators return zero because they can be fully streamed without materialization.

Off-chip memory operators: $|output\ dtype| \times 2$

Bufferize: $|input\ dtype| + ||buffer|| \times |input\ dtype| \times 2$
We multiply by two, assuming double buffering.

Accum, Scan, Expand: $|output\ dtype|$

Map, Accum with matrix multiplication:
$(16 \times in\_tile\_col + |weight\ tile| + |output\ tile|)$
The in_tile_col denotes the input tile's innermost dimension size. The output tile size is included only for Accum. We account for the storage of partial-input tiles and the

[6] All STeP operators with an input reference stream have a static variant.

full weight tile because we use inner-product matrix multiplication. We multiply by 16 to mirror the decomposition of STeP-level tiles into 16×16 tiles that align with the hardware's compute-unit tile size.[7]

Handling data dependencies. When dynamic-regular or ragged dimensions are present, the metrics produced by the symbolic frontend include symbolic variables.[8] By substituting these symbols with different input shapes or control-flow decisions, programmers can quickly analyze off-chip traffic and on-chip memory requirements.

The exact values for each metric are obtained by invoking the simulator explained in the following subsection. The symbolic frontend tracks operators whose metrics depend on runtime data and enables off-chip traffic or on-chip memory measurements for those operators during simulation. The values from the symbolic frontend and the simulator are then aggregated to produce the final concrete metrics.

4.3 Performance Model for the Simulator

Since the symbolic STeP frontend has no timing information, we implement a simulator backend for STeP in Rust using the Dataflow Abstract Machine [55] simulation framework.

To model the data transfer between off-chip memory and on-chip memory, the simulator implements an HBM node that emulates the timing behavior of Ramulator 2.0 [25], a cycle-accurate DRAM simulator. The latency of accessing on-chip memory is factored into the higher-order operators that execute arithmetic functions, using Roofline modeling. Each higher-order operator is allocated a compute bandwidth (FLOPs/cycle). On each input element in the stream, the operators increment cycles based on the following equation:

$$\max\left(\frac{size\ of\ inputs}{on\text{-}chip\ mem\ BW}, \frac{total\ FLOPs}{compute\ BW}, \frac{size\ of\ outputs}{on\text{-}chip\ mem\ BW}\right)$$

As shown by the `matmul_map` in Listing 1, the *compute BW* is provided by the programmer, and *total FLOPs* is computed within the function supplied to the higher-order operators, as this value depends on the specific computation the function performs. The first and last entries in the equation are zero when the input and output are streamed directly between compute units via FIFOs.

4.4 Portability

As an abstraction, STeP is not tied to a specific hardware implementation [15] and is portable across diverse SDA implementations with software-managed scratchpads [20, 33–35, 38, 48], which we discuss in Section 6.2. The equations in the symbolic frontend can be customized to capture hardware-specific operator details, such as hardware tile sizes and matrix-multiplication implementation. If performance bottlenecks shift, additional cost functions can be added to STeP operators to obtain performance-correlated metrics (e.g. on-chip traffic, compute). In the simulator, operator initiation intervals and latencies can be adjusted to match hardware characteristics [55]. For example, when multiple Bufferize operators share an on-chip memory unit, a scratchpad simulator can replace the Roofline model to capture on-chip memory contention. Also, different memory technologies can be modeled by reconfiguring or replacing the simulator's HBM node [25].[9]

4.5 Validation

We validate the simulator by comparing the performance to a cycle-accurate HDL simulation. We also compare the off-chip traffic captured in the symbolic STeP frontend with the performance to validate the usefulness of the metrics captured in the symbolic frontend.

Workload and Hardware Model. We use a SwiGLU [40] layer as the workload since it contains representative computations in ML models such as matrix multiplication, activation function, and row-wise reduction. We choose a spatial architecture of compute units that operate on 16×16 BFloat16 tiles, each having an initiation interval of one. We pair compute tiles with distributed on-chip memory units, each capable of reading and writing one tile per cycle.

The HDL model is implemented in Bluespec SystemVerilog and executed in a cycle-accurate BlueSim simulator [3, 29]. Off-chip access delays are integrated using the Ramulator2 library calls using a configuration of an HBM2 subsystem with 8 stacks. The on-chip memory bandwidth is configured as 256 (bytes/cycle), and the cycle-approximate STeP simulator uses the same memory configurations. We measure the total execution time from the first off-chip read to the last off-chip write.

Mapping Methodology. We apply a graph transformation to partition tiles into smaller physical tiles that match the fabric's compute tile size.[10] After the transformation, every node in the graph maps to a dedicated unit in the HDL design, which we attach to a congestion-free interconnect. The programmer-specified compute bandwidth determines how many compute units are mapped to each STeP node.

Results. As shown in Figure 8, the STeP simulator's cycle-count closely matches that of the HDL simulator, with a Pearson correlation of 0.99, when sweeping different tile sizes. As the application is memory-bound in the given hardware configuration, decisions on data transfer across the memory hierarchy significantly impact performance, highlighting the importance of having visibility and control over these decisions in the abstraction. The high correspondence

[7] Refer to Section B.2 for more detail on hierarchical tiling.

[8] The symbolic frontend also introduces symbols for static ragged dimensions, as explicitly tracking all ragged values would significantly increase the complexity of metric computation.

[9] Detailed instructions for customizing the frontend and simulator can be found in the artifact repository described in Section A.2.

[10] An example transformation is shown in Figure 18 in the Appendix.

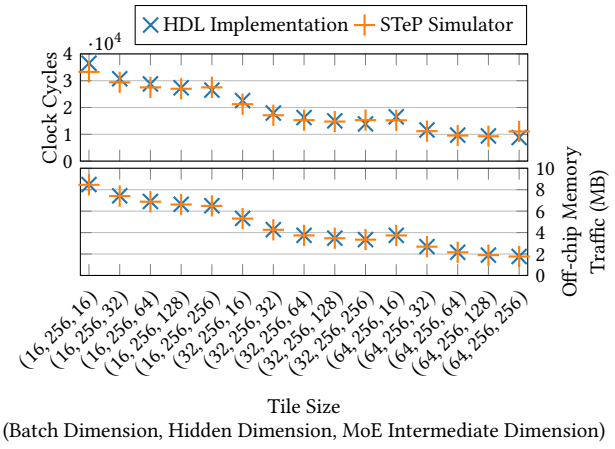

Figure 8. Cycle-count and memory traffic comparison of a SwiGLU Layer with different tile sizes. The full sizes of the batch dimension, hidden dimension, and MoE intermediate dimension are 64, 256, and 512, respectively.

between the off-chip traffic captured in the symbolic STeP frontend and the HDL simulator's performance and incurred off-chip traffic suggests that the metrics captured in the symbolic frontend and the STeP simulator can provide valuable insights into the performance of a given STeP graph.

5 Evaluation

In this section, we evaluate STeP's ability to explore efficient schedules for dynamic ML models by implementing optimizations that were not expressible in prior abstractions for SDAs. Key STeP features that enable each optimization are listed in Table 2. Our evaluation shows that these optimizations enable Pareto-optimal design points over prior work and deliver speedups or resource savings. To show how the optimizations integrate with full LLM inference, we also evaluate them on end-to-end models.

5.1 Methodology

Workload. Our workloads consist of two layers: Mixture-of-Experts (MoE) with SwiGLU [40] experts and attention [47]. We focus on these layers because they dominate end-to-end inference latency. For example, when running DeepSeek-R1 on 64× B200 GPUs, attention and MoE layers together account for approximately 80% of the total latency [54]. We use configurations from Qwen3-30B-A3B and Mixtral-8x7B. We choose Qwen3-30B-A3B because it shares a common architecture with many of the top-20 open-source models [6, 8, 13, 23, 43–45, 52] on the LM Arena leaderboard [24]. Although Mixtral is relatively older, we include it to demonstrate the impact of our optimizations across varied expert-activation patterns in MoE models.

Dataset For the attention layer, the KV cache length for each batch is sampled from the AzureLLMInference dataset [32].

Optimization	Related STeP Features
Dynamic Tiling	Dynamic tile shape Explicit memory hierarchy Accum of dynamically-sized tiles
Configuration Time-multiplexing	Explicit memory hierarchy Dynamic Routing and Merging Operators
Dynamic Parallelization	Dynamic Routing and Merging Operators

Table 2. Key STeP features that enable the optimizations

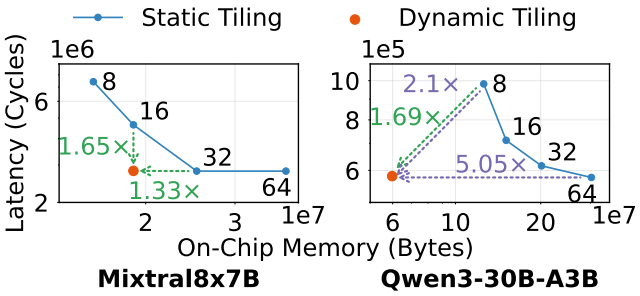

Figure 9. Performance and memory requirements of tiling strategies for the batch dimension of each expert (batch = 64). The numbers on the static tiling curve denote tile size.

For the MoE layer, we use expert-routing data collected by running the models on the HH-RLHF [10] request traces.[11]

Simulator Setup. We use the STeP simulator described in Section 4.3. The bandwidth of each on-chip memory unit is set to 64 bytes/cycle, and the off-chip memory bandwidth is set to 1024 bytes/cycle, matching the configurations of recent reconfigurable dataflow accelerators [33, 35].

Baseline Design. We chose Revet [38] as our baseline since it has the most extensive support for dynamic behaviors among SDA programming abstractions with explicit memory hierarchy. However, as discussed in Section 2.3, restrictions in Revet's dataflow thread model and its lack of support for dynamically-sized tiles make it impossible to express the proposed optimizations. Therefore, we use STeP to implement schedules that are expressible in Revet and treat these implementations as the baseline. All other scheduling and mapping decisions are identical between the baseline and optimized implementations. We do not compare against Ripple as it targets SDAs without on-chip scratchpads (see Section 2.3).

5.2 Dynamic Tiling

Dynamic tiling is a scheduling strategy where the size of a tile in a data stream is determined at runtime. When applying dynamic tiling to the batch dimension for MoEs, tokens are grouped into tiles whose size adapts to the number of tokens per expert in each batch. On the contrary, static tiling pads the tokens for each expert into statically-sized tiles. Dynamic

[11] More details on the datasets can be found in Section B.3 of the Appendix.

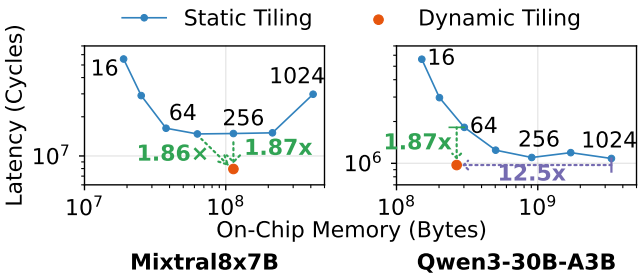

Figure 10. Performance and memory requirements of tiling strategies for the batch dimension (batch = 1024).

tiling can be expressed in STeP by replacing the first Reshape in Figure 7 with a Promote. This enables the following Accum to accumulate dynamically-shaped tiles.

Dynamic tiling extends the Pareto frontier beyond what is achievable with static tiling. We demonstrate this using the performance improvements (green dashed arrows) and on-chip memory savings (purple dashed arrows) against the closest static-tiling Pareto point along each axis in Figures 9 and 10. As shown in Figure 9, for Mixtral8x7B, dynamic tiling achieves a 1.65× speedup while using the same on-chip memory as $tile = 16$. It also reduces on-chip memory by 1.33× while delivering performance comparable to $tile = 32$ (within +0.26% cycles). Similarly, for Qwen3-30B-A3B, dynamic tiling achieves a 1.69× speedup while using 2.1× less on-chip memory than $tile = 8$, and reduces on-chip memory by 5.05× while maintaining performance comparable to $tile = 64$ (within +0.76% cycles). The larger savings for Qwen3-30B-A3B are due to its higher number of experts.

At larger batch sizes, dynamic tiling enables design points that achieve performance unattainable using any static tile size, as shown in Figure 10. For Mixtral8x7B, static tiling saturates beyond $tile = 128$, where increasing the tile size yields little to no additional speedup. In contrast, dynamic tiling improves performance beyond $tile = 128$, delivering a 1.86× speedup while using 1.79× more on-chip memory, and a 1.87× speedup over $tile = 256$ with the same on-chip memory requirement. For Qwen3-30B-A3B, dynamic tiling achieves a 1.87× speedup while using 1.13× less on-chip memory than $tile = 64$. It also delivers a 1.12× speedup while reducing on-chip memory by 12.5× relative to the best-performing static configuration ($tile = 1024$).

In summary, dynamic tiling breaks the prior Pareto-optimal frontier, achieving Pareto Improvement Distances (PID) of 1.33× and 2.11× for Mixtral-8×7B and Qwen3-30B-A3B, respectively, in Figure 9, and 1.86× and 1.87× in Figure 10.[12] PID for a design point p with respect to the Pareto-optimal subset of baseline points F_B is defined as:

$$\mathrm{PID}(p) = \min_{q \in F_B} \max\left(\frac{\mathrm{cycles}(q)}{\mathrm{cycles}(p)}, \frac{\mathrm{mem}(q)}{\mathrm{mem}(p)}\right).$$

[12]For more details on PID, see Section B.4.

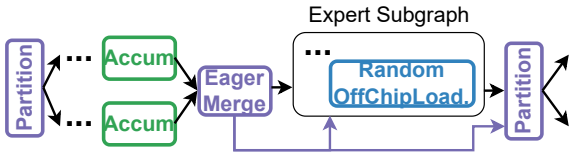

Figure 11. STeP graph with configuration time-multiplexing. Certain operators are omitted for simplicity.

The static-tiling Pareto curve reflects a trade-off between on-chip memory usage and off-chip traffic: small tiles incur frequent off-chip reloads, while large tiles waste on-chip memory due to padding and unused space.[13] Because the application is memory-bound under our simulated hardware configuration, this trade-off in off-chip traffic directly translates to a trade-off in performance. Dynamic tiling removes the need to choose between frequent reloads and wasted capacity by adapting tile sizes to the active workload. By achieving high performance with smaller or comparable on-chip memory, dynamic tiling frees up space that can be repurposed to further improve performance, for example by increasing data reuse via larger tiles or more aggressive operator fusion.

5.3 Configuration Time-multiplexing

Configuration time-multiplexing is an optimization that time-multiplexes a configuration across branches with shared computation structure in applications with data-dependent control flow. In the context of executing MoEs, a configuration is dynamically time-multiplexed across experts by routing inputs and weights accordingly. For the simplified MoE example in Figure 7, configuration time-multiplexing can be expressed by inserting a pair of control-flow operators around the time-multiplexed region, as shown in Figure 11. EagerMerge forwards inputs for each expert to the time-multiplexed region as soon as they become available. RandomOffChipLoad fetches the weight for the selected expert dynamically, instead of using LinearOffChipLoad.

Configuration time-multiplexing avoids allocating dedicated compute and memory resources for every possible branch. To quantify the resource savings, we sweep the number of experts sharing the same configured region for the MoE layer in Qwen3-30B-A3B using static tiling ($tile = 32$) and dynamic tiling. With static tiling, compute utilization improves by 2.64× with under 1% performance overhead (Figure 12(a)). With dynamic tiling, compute utilization improves by 2.51× with about 5% overhead (Figure 12(b)). As shown in Figure 13, this optimization achieves comparable performance while freeing up 62% allocated on-chip compute and 46% memory resources, which can be reallocated to support more concurrent requests or larger models.

[13]The corresponding Pareto curve for off-chip traffic versus on-chip memory is shown in Section B.4 of the Appendix.

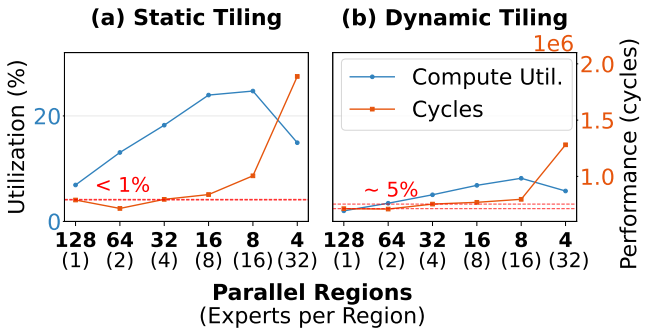

Figure 12. Resource utilization for the MoE layer in Qwen3-30B-A3B with different tiling strategies for the batch dimension of each expert (batch size = 64). Dynamic tiling has lower compute utilization when compared to static tiling because static tiling has 3.81× more total FLOPs due to padding. The tile size used in static tiling is 32.

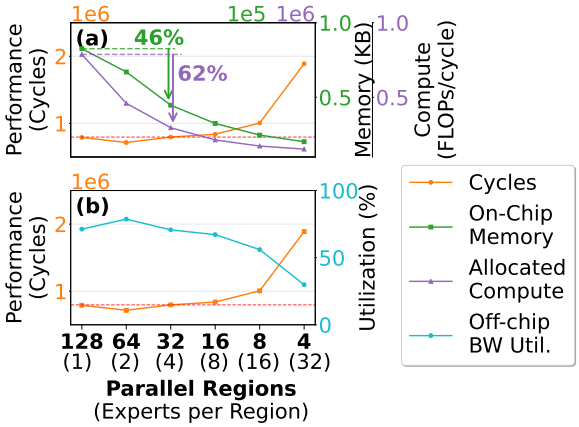

Figure 13. Resource usage and performance for the MoE layer in Qwen3-30B-A3B with time-multiplexing (tile size =32, batch size = 64). The compute utilization drop in Figure 12 is due to decreased off-chip memory bandwidth utilization because the number of parallel regions are not large enough to saturate the off-chip memory bandwidth.

5.4 Dynamic Parallelization

Dynamic parallelization is an optimization that dispatches work as soon as downstream parallel pipelines become available. This can improve performance by balancing load across spatially parallel regions when parallelizing workloads whose size or distribution can vary. In ML workloads, unevenly sized workloads arise in attention computation during decoding, since KV cache lengths vary across requests. The number of requests within a batch also varies dynamically due to optimizations such as continuous batching [22, 53]. For attention, STeP implements dynamic parallelization as illustrated in Figure 16. Each request is routed to one of several parallel regions using Partition. The selector stream for Partition is formed by merging two streams: One for round-robin assignment of the initial iteration (FlatMap) and another signaling the availability of parallel regions (EagerMerge).

We compare parallelization strategies across varying batch sizes and KV cache length distributions. We parallelize the batch dimension by four and use two tiled static parallelization baselines: coarse-grained and interleaved parallelization. Static coarse-grained parallelization fixes the number of requests assigned to each parallel region (16 in our implementation), whereas static interleaved parallelization distributes requests across regions in a round-robin fashion.

As shown in Figure 14, dynamic parallelization consistently outperforms static interleaved parallelization as KV cache length variation increases. With low variation, dynamic parallelization achieves 1.14×–1.26× speedup; with high variation, 1.47×–1.57× speedup. This is because under a larger KV cache length variation, static interleaved parallelization suffers from blocking when long requests increase load imbalance, leaving resources idle.

Dynamic parallelization also maintains high utilization across parallel regions as batch size varies. In Figure 15, it achieves a 2.72× speedup over static coarse-grained parallelization at batch=16 because several parallel regions remain idle under static coarse-grained parallelization. Although static performance improves with larger batches, it remains 1.43× slower at batch=64 due to persistent load imbalance.

5.5 End-to-end Model

To evaluate end-to-end impact, we implement the full Qwen3-30B-A3B and Mixtral-8x7B models in STeP, with and without the proposed optimizations presented above. Each Transformer decoder layer is fused into a single STeP graph and executed repeatedly with layer-specific weights. Each layer comprises of STeP graphs for QKV generation, attention, and MoE; we parallelize the batch dimension by a factor of four for QKV generation and attention, and use expert parallelism for MoE. We use static interleaved parallelization for attention, which performs best overall across batch sizes and KV-cache lengths in our ablation study (Figure 21 in the Appendix). For MoE computation, we use static tiling with performance- and memory-matched tile sizes, which are the same closest points along each axis, from Figure 9.

As shown in Figure 17, our proposed optimizations improve performance while using comparable or fewer on-chip resources. Mixtral-8×7B and Qwen3-30B-A3B achieve 1.27× and 1.15× speedups, respectively, over the memory-matched implementation, driven by fewer off-chip accesses from dynamic tiling and better load balancing from dynamic parallelization. Qwen3-30B-A3B achieves this speedup while using 69% less on-chip memory and 54% fewer compute resources through configuration time-multiplexing.

Compared with the performance-matched static implementation, on-chip memory usage is reduced by 20% for

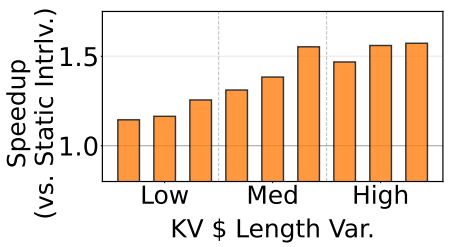

Figure 14. Speedup of dynamic parallelization over static interleaved parallelization across batches with different KV-cache length distributions (batch size=64).

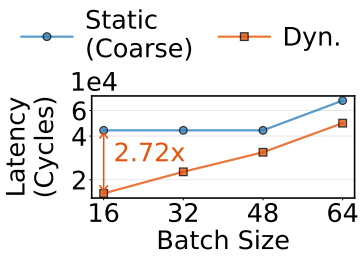

Figure 15. Performance comparison between coarse-grained and dynamic parallelization strategies.

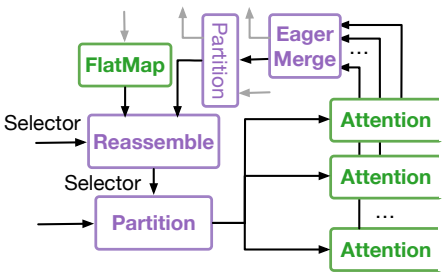

Figure 16. The STeP graph for dynamic parallelization. Shape operators omitted for simplicity.

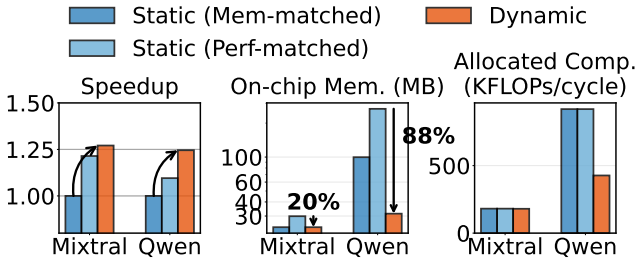

Figure 17. End-to-end result for Qwen3-30B-A3B and Mixtral8x7B. We use KV-cache length traces with median standard deviation and expert routing traces whose expert bin count standard deviation matches the overall average.

Mixtral-8×7B and by 88% for Qwen3-30B-A3B due to dynamic tiling, with additional savings for Qwen3-30B-A3B from configuration time-multiplexing. Even with the static performance-matched MoE implementation in Figure 9, the dynamic implementation still deliver 1.05× (Mixtral-8×7B) and 1.14× (Qwen3-30B-A3B) speedups due to dynamic parallelization. We do not apply configuration time-multiplexing to Mixtral-8×7B, since all experts are active at a batch=64. However, many recent MoE models activate only a small fraction of a large expert pool (128+ experts) per token [1, 17, 27, 44], indicating that the resource savings observed for Qwen3-30B-A3B generalize to modern MoEs.

5.6 DSE with the Symbolic Frontend and Simulator

The experiments in the preceding subsections not only demonstrate new optimizations, but also illustrate how STeP's symbolic frontend and simulator can be used for design space exploration (DSE). In the dynamic tiling experiments (Figures 9 and 10), static tiling requires sweeping tile sizes to identify the optimal trade-off between on-chip memory and performance. If the hardware supports only static tiling, the symbolic frontend and simulator can be used to search for the optimal tile size for a given model, batch size, and expert distribution. Similarly, for configuration time-multiplexing (Figures 12 and 13) and dynamic parallelization (Figures 14 and 15), the symbolic frontend and simulator enable comparison of design points across different schedules, such as varying degrees of time-multiplexing and parallelization strategies. They also allow evaluation under input variations, including batch size and KV-cache length distributions.

6 Discussion on Future Work

This section discusses future support for compiling models defined in a high-level framework to STeP, and potential approaches to support the dynamic features of STeP on SDAs. We leave an optimal hardware SDA design and a high-level compiler for STeP as future work.

6.1 Compiling from High-level Frameworks to STeP

Although we present STeP as a programming abstraction in this paper, it can also serve as an intermediate representation for compilers. For example, ML models defined in high-level frameworks such as PyTorch can be compiled to STeP using the `torch.compile` interface [2]. `torch.compile` captures the model into an FX graph expressed in terms of tensor-level operators, which a compiler can traverse and systematically lower into corresponding STeP subgraphs. Optimization schedules—including parallelization, tiling, and configuration time-multiplexing—can be specified over the index variables of the program. These schedules guide how each FX node is translated into STeP and determine which optimizing rewrites are applied during lowering.

6.2 Supporting Dynamic STeP features in SDAs

Prior SDAs have already demonstrated architectures that process stop-token-embedded data streams [20, 38]. A few bits in the datapath are used to identify the stop tokens and their level, and the streams are processed either by repurposing existing hardware units [38] or by designing a new dedicated state machine to process stop tokens [20].

STeP's control flow operators (Section 3.2.3) can be implemented by spatially laying out all branches and activating the appropriate ones data-dependently at runtime. Routing

can be implemented either through predication in compute units [56] or within the network-on-chip interconnect [12].

To handle dynamic tensor sizes, the memory system must support virtualization by allocating space at a fixed granularity independent of stream length and maintaining mappings between stream references and their memory addresses. Noncontiguous allocation is also required to avoid fragmentation. This can be implemented using a hardware-managed mapping cache (e.g. a linked list) that translates stream references into a sequence of noncontiguous physical addresses. With 512 KB of local memory per unit [33], the mapping cache requires less than 30 KB of metadata (≈6% overhead), comparable to the tag overhead in conventional caches. Furthermore, arbitrary tensor sizes without an upper bound can be supported via spilling mechanisms demonstrated in prior SDAs [11], where data is automatically spilled and metadata for accessing the spilled data remains on-chip.

7 Related Work

This section presents work related to STeP beyond SDAs.

CUDA graphs and conditional nodes [30] enable conditional or repeated execution of subgraphs without returning control to the CPU. While similar to STeP's control-flow operators, they do not support dynamic shapes. Common workarounds map dynamic shapes to an enumeration of many static GPU kernels, which adds overhead as the dynamic dimension's range increases [36].

Dynamic task-parallelism frameworks [4, 7, 16] share several themes with STeP but operate at different granularities and use different mechanisms. TaskStream [7] and STeP both support asynchronous units, dynamic work distribution, and dynamic data reuse. However, STeP realizes these ideas at the tile granularity via dynamic dataflow blocks, whereas TaskStream targets coarser task instances via hardware scheduling. Taskflow [16] also provides in-graph control flow and graph-based parallel abstractions, but targets CPU/GPU systems and supports dynamic parallelism via work stealing. Cheng et al. [4] similarly target irregular workloads on scratchpad-managed architectures, but focus on a Cilk/TBB-style work-stealing runtime for manycore systems.

8 Conclusion

We introduced the Streaming Tensor Programs, a streaming abstraction for dynamic tensor applications on spatial dataflow accelerators. STeP expresses optimizations that are Pareto-optimal over prior spatial dataflow abstractions, delivering speedups and/or with fewer resources. Its stream structure and shape semantics expose performance-critical metrics, creating new opportunities for optimization. We envision that STeP will enable richer forms of dynamic applications and architectures.

9 Acknowledgments

We thank Paul Mure, Rubens Lacouture, Christophe Gyurgyik, Suguna Velury, Tanmay Garg, Benjamin Driscoll, Raghu Prabhakar, Alex Rucker, Fredrik Kjolstad, Tian Zhao, Shiv Sundram, Qizheng Zhang, and Sally Wang for discussion and their helpful feedback. Gina Sohn was supported by the Stanford Graduate Fellowship and NSF GRFP. This work was supported in part by DARPA under the Machine learning and Optimization-guided Compilers for Heterogeneous Architectures (MOCHA) program (award number HR00112520038), and by the Naval Surface Warfare Center under Agreement No. N00164-23-9-G057-01. This research was also supported in part by the Stanford Data Analytics for What's Next (DAWN) Affiliate Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925* (2025).

[2] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947.

[3] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The essence of Bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 243–257.

[4] Lin Cheng, Max Ruttenberg, Dai Cheol Jung, Dustin Richmond, Michael Taylor, Mark Oskin, and Christopher Batten. 2023. Beyond static parallel loops: Supporting dynamic task parallelism on manycore architectures with software-managed scratchpad memories. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 46–58.

[5] Krishna Teja Chitty-Venkata, Siddhisanket Raskar, Bharat Kale, Farah Ferdaus, Aditya Tanikanti, Ken Raffenetti, Valerie Taylor, Murali Emani, and Venkatram Vishwanath. 2024. Llm-inference-bench: Inference benchmarking of large language models on ai accelerators. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1362–1379.

[6] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261* (2025).

[7] Vidushi Dadu and Tony Nowatzki. 2022. TaskStream: accelerating task-parallel workloads by recovering program structure. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3503222.3507706

[8] Damai Dai, Chengqi Deng, Chenggang Zhao, RX Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Yu Wu, et al. 2024.

Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *arXiv preprint arXiv:2401.06066* (2024).

[9] Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca P. Carloni, and Laura Pozzi. 2020. Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3736–3747. doi:10.1109/TCAD.2020.3012750

[10] Deep Ganguli, Liane Lovitt, Jackson Kernion, Amanda Askell, Yuntao Bai, Saurav Kadavath, Ben Mann, Ethan Perez, Nicholas Schiefer, Kamal Ndousse, et al. 2022. Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned. *arXiv preprint arXiv:2209.07858* (2022).

[11] Souradip Ghosh, Yufei Shi, Brandon Lucia, and Nathan Beckmann. 2025. Ripple: Asynchronous Programming for Spatial Dataflow Architectures. *Proc. ACM Program. Lang.* 9, PLDI, Article 157 (June 2025), 28 pages. doi:10.1145/3729256

[12] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2022. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 546–564. doi:10.1109/MICRO56248.2022.00046

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

[14] Olivia Hsu, Alexander Rucker, Tian Zhao, Varun Desai, Kunle Olukotun, and Fredrik Kjolstad. 2025. Stardust: Compiling sparse tensor algebra to a reconfigurable dataflow architecture. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 628–643.

[15] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 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

[16] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2021. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (2021), 1303–1320.

[17] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088* (2024).

[18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, 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

[19] David Koeplinger, Darshan Gandhi, Pushkar Nandkar, Nathan Sheeley, Matheen Musaddiq, Leon Zhang, Reid Goodbar, Matthew Shaffer, Han Wang, Angela Wang, et al. 2024. Kernel Looping: Eliminating Synchronization Boundaries for Peak Inference Performance. *arXiv preprint arXiv:2410.23668* (2024).

[20] Kalhan Koul, Maxwell Strange, Jackson Melchert, Alex Carsello, Yuchen Mei, Olivia Hsu, Taeyoung Kong, Po-Han Chen, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Zhouhua Xie, Christopher Torng, Joel Emer, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina. 2024. Onyx: A 12nm 756 GOPS/W Coarse-Grained Reconfigurable Array for Accelerating Dense and Sparse Applications. In *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 1–2. doi:10.1109/VLSITechnologyandCir46783.2024.10631383

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*. 611–626.

[22] Woosuk Kwon, Yifan Shen, Zifan Xiao, Zhifeng Yao, Ce Zhang, Ion Stoica, Hao Chen, and Matei Zaharia. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. *arXiv preprint arXiv:2309.06180* (2023). https://arxiv.org/abs/2309.06180

[23] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[24] LM Arena. 2025. LM Arena Leaderboard. https://lmarena.ai/leaderboard. Accessed: Aug. 2025.

[25] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, , and Onur Mutlu. 2023. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator.

[26] Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner. 2016. Adaptive Threshold Non-Pareto Elimination: Re-thinking machine learning for system level design space exploration on FPGAs. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 918–923.

[27] Meta AI. 2025. The Llama 4 Herd: The Beginning of a New Era of Natively Multimodal Models. https://ai.meta.com/blog/llama-4-multimodal-intelligence/

[28] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. doi:10.7717/peerj-cs.103

[29] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 69–70.

[30] NVIDIA Corporation. 2024. *CUDA C++ Programming Guide*. NVIDIA. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ Release 12.6.

[31] Bowen Pang, Kai Li, and Feifan Wang. 2025. Optimizing LLM Inference Throughput via Memory-aware and SLA-constrained Dynamic Batching. arXiv:2503.05248 [cs.DC] https://arxiv.org/abs/2503.05248

[32] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[33] Raghu Prabhakar. 2024. SambaNova SN40L RDU: Breaking the Barrier of Trillion+ Parameter Scale Gen AI Computing. In *2024 IEEE Hot Chips 36 Symposium (HCS)*. 1–24. doi:10.1109/HCS61935.2024.10664717

[34] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. 2022. SambaNova SN10 RDU: A 7nm Dataflow Architecture to Accelerate Software 2.0. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 350–352. doi:10.1109/ISSCC42614.2022.9731612

[35] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association

for Computing Machinery, New York, NY, USA, 389–402. doi:10.1145/3079856.3080256

[36] PyTorch Contributors. 2025. CUDAGraph Trees. https://docs.pytorch.org/docs/stable/torch.compiler_cudagraph_trees.html. PyTorch 2.9 documentation, section "Reasons for Skipping CUDAGraph". Accessed: 2025-12-03.

[37] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 3505–3506.

[38] Alexander C. Rucker, Shiv Sundram, Coleman Smith, Matthew Vilim, Raghu Prabhakar, Fredrik Kjolstad, and Kunle Olukotun. 2024. Revet: A Language and Compiler for Dataflow Threads . In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–14. doi:10.1109/HPCA57654.2024.00016

[39] Jon Saad-Falcon, Avanika Narayan, Hakki Orhun Akengin, J. Wes Griffin, Herumb Shandilya, Adrian Gamarra Lafuente, Medhya Goel, Rebecca Joseph, Shlok Natarajan, Etash Kumar Guha, Shang Zhu, Ben Athiwaratkun, John Hennessy, Azalia Mirhoseini, and Christopher Ré. 2025. Intelligence per Watt: Measuring Intelligence Efficiency of Local AI. arXiv:2511.07885 [cs.DC] https://arxiv.org/abs/2511.07885

[40] Noam Shazeer. 2020. GLU Variants Improve Transformer. arXiv:2002.05202 [cs.LG] https://arxiv.org/abs/2002.05202

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[42] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2024. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. arXiv:2408.00741 [cs.AI] https://arxiv.org/abs/2408.00741

[43] GLM-4.5 Team. 2025. GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models. https://arxiv.org/abs/2508.06471

[44] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534* (2025).

[45] Tencent Hunyuan Team, Ao Liu, Botong Zhou, Can Xu, Chayse Zhou, ChenChen Zhang, Chengcheng Xu, Chenhao Wang, Decheng Wu, Dengpeng Wu, et al. 2025. Hunyuan-TurboS: Advancing Large Language Models through Mamba-Transformer Synergy and Adaptive Chain-of-Thought. *arXiv preprint arXiv:2505.15431* (2025).

[46] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, Berlin, Heidelberg, 179–196.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[48] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. 2021. Aurochs: an architecture for dataflow threads. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 402–415. doi:10.1109/ISCA52012.2021.00039

[49] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. 2020. Gorgon: Accelerating Machine Learning from Relational Data. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 309–321. doi:10.1109/ISCA45697.2020.00035

[50] Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. 2024. Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution. arXiv:2409.12191 [cs.CV] https://arxiv.org/abs/2409.12191

[51] Zhiyao Xie, Guan-Qi Fang, Yu-Hung Huang, Haoxing Ren, Yanqing Zhang, Brucek Khailany, Shao-Yun Fang, Jiang Hu, Yiran Chen, and Erick Carvajal Barboza. 2020. FIST: A Feature-Importance Sampling and Tree-Based Method for Automatic Design Flow Parameter Tuning. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 19–25. doi:10.1109/ASP-DAC47756.2020.9045201

[52] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).

[53] Sungjin Yu, Jaehong Jeong, et al. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. https://www.usenix.org/conference/osdi22/presentation/yu

[54] Sungmin Yun, Seonyong Park, Hwayong Nam, Younjoo Lee, Gunjun Lee, Kwanhee Kyung, Sangpyo Kim, Nam Sung Kim, Jongmin Kim, Hyungyo Kim, et al. 2025. The new LLM bottleneck: A systems perspective on latent attention and mixture-of-experts. *arXiv preprint arXiv:2507.15465* (2025).

[55] Nathan Zhang, Rubens Lacouture, Gina Sohn, Paul Mure, Qizheng Zhang, Fredrik Kjolstad, and Kunle Olukotun. 2024. The Dataflow Abstract Machine Simulator Framework. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 532–547. doi:10.1109/ISCA59077.2024.00046

[56] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: scaling a reconfigurable dataflow accelerator. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 1041–1054. doi:10.1109/ISCA52012.2021.00085

[57] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. 2024. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems* 37 (2024), 62557–62583.

[58] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 11, 18 pages.

A Artifact Appendix

A.1 Abstract

This appendix describes how to set up and run programs written in the Symbolic Python frontend for the Streaming Tensor Program (STeP) using the STeP Rust simulator and the Bluespec SystemVerilog HDL simulator. The artifact provides a Docker image containing all required dependencies (Python, Rust, Bluespec, protobuf, etc.) and scripts to reproduce the experimental results reported in the paper. The artifact can be executed on any x86-64 machine with Docker, Python 3, Git, and Bash support, at least 32 GB of RAM, and more than 20 GB of disk space.

A.2 Artifact Check-List (Meta-Information)

- **Data set:** AzureLLMInferenceDataset [32], expert routing data collected by running Qwen3-30B-A3B [52] and Mixtral8x7B [17] models using the `hh-rlhf` serving trace [10], and synthetically generated tensors. To reduce artifact size, we include only the data used to generate the plots in the paper from the AzureLLMInferenceDataset and the expert routing data.
- **Run-time environment:** Docker, Git, Python 3, and Bash must be installed on the local machine. Proficiency in Bash and Git is recommended.
- **Hardware:** Any conventional x86-64 CPU with at least 32 GB of RAM should work.
- **Metrics:** Cycles, off-chip memory traffic, on-chip memory requirements, allocated compute resources, compute resource utilization, and off-chip memory bandwidth utilization.
- **Output:** Terminal output, files, and graphs (PDF figures).
- **How much disk space is required (approximately):** Approximately 20 GB of disk space is sufficient.
- **How much time is needed to prepare the workflow (approximately)?:** About 10–15 minutes.
- **How much time is needed to complete experiments (approximately)?:** The total time to complete all experiments is approximately 24.5 hours when measured on a Google Cloud C4-standard-8 instance (8 Intel Emerald Rapids vCPUs, 30 GB memory). The breakdown is as follows:
  - Figure 8: 2 hours
  - Figure 9: 2 hours 30 minutes
  - Figure 10: 17 hours 10 minutes
  - Figure 12: 1 hour 40 minutes
  - Figure 13: 50 minutes
  - Figure 14: 4 minute
  - Figure 15: 1 minutes
  - Figure 21: 15 minutes
- **Publicly available?:** Yes, on GitHub at step_artifact and step-artifact-hdl.
- **Code licenses (if publicly available)?:** MIT License
- **Workflow framework used?:** Docker
- **Archived (provide DOI)?:** Yes. The DOI is https://doi.org/10.6084/m9.figshare.31095274

A.3 Description

A.3.1 How to Access. The code for this submission can be downloaded from the step_artifact and step-artifact-hdl repositories. The step_artifact repository includes a Dockerfile that can be used to build the Docker image for the full evaluation of the artifact. The Docker image is also available at https://doi.org/10.6084/m9.figshare.31095274.

A.3.2 Hardware Dependencies. We recommend using an x86-64 machine with at least 32 GB of memory.

A.3.3 Software Dependencies. The artifact requires a machine with Docker, Git, and Python 3 installed. We evaluated the artifact using the following configuration: Ubuntu 24.04 LTS, Docker 29.1.3, and Python 3.12 (Intel-based machine).

A.3.4 Data Sets. The experiments for Figures 9, 10, 12 and 13 use expert routing data collected by running the Qwen3-30B-A3B [52] and Mixtral8x7B [17] models with the `hh-rlhf` serving trace [10]. To select representative cases, we measure the standard deviation of expert bin counts across iterations and layers and choose the case whose deviation is closest to the overall average.

The experiment for Figures 14, 15 and 21 uses KV cache lengths sampled from the AzureLLMInferenceDataset [32]. We analyze 5,000 requests within a time window, forming batches with varying prompt-length distributions. We experiment with batches whose prompt-length standard deviation matches that of the full set, as well as batches with the most and least variability. The KV cache length data and expert routing data used for the experiments are included in the step_artifact repository.

A.4 Installation

To install the artifact, first clone the step_artifact and step-artifact-hdl repositories to the local machine. Then, build the Docker image using the following commands (the build can take up to 5 minutes):

```
$ git clone --recursive https://github.com/
    stanford-ppl/step_artifact.git
$ git clone https://github.com/stanford-ppl/
    step-artifact-hdl.git
$ docker build -f step_artifact/Dockerfile
    -t step_artifact .
```

The Docker container can be started with the following command, which will print the container ID:

```
$ docker run -dit step_artifact bash
```

The container can be attached by running:

```
$ docker attach <CONTAINER_ID>
```

Once inside the Docker container, move into the directory `step_artifact` and run the following command to set up the environment:

```
### Inside the Docker container ###
$ cd /root/step_artifact
$ source setup.sh
```

A.5 Experimental Workflow

The experimental workflow for this artifact consists of running scripts inside the Docker container to execute experiments and generate the figures in the paper. Detailed instructions can be found in the `README.md` files within the step_artifact and step-artifact-hdl repositories.

A.6 Evaluation and Expected Results

All experiments and figures can be reproduced using the following commands. In total, the workflow takes approximately 7 hours when tested on a Google Cloud C4-standard-8 instance (8 Intel Emerald Rapids vCPUs, 30 GB memory).

```
### In the Docker container ###
$ cd /root/step_artifact
# Figures 9, 10, 12, 13, 14, 15, and 21
$ source ae_cmd.sh
# Figure 8
$ cp /root/step_artifact/hdl_validation/fig8
    .csv /root/step-artifact-hdl/
    step_reference.csv
$ cd /root/step-artifact-hdl
$ ./run_dse_and_figure.sh
# ctrl+p ctrl+q
```

Once the experiments finish, detach from the container by pressing `Ctrl+P` followed by `Ctrl+Q`. To copy the experiment results and figures from the container, move into the cloned `step_artifact` repository on the local machine and run the following commands. The `CONTAINER_ID` is the same ID used to attach to the container; it can also be retrieved by running `docker ps`. The results and figures will be copied to `step_artifact/results`.

```
### On the local machine ###
$ cd step_artifact
$ mkdir -p results
$ python copy_from_docker.py --docker_id
    <CONTAINER_ID> --output_dir ./results
```

The expected directory structure under `step_artifact/results` is as follows:

```
step_artifact/results
|_ step-artifact-hdl
|_ step_artifact
    |_ dyn_tiling
    |_ dynamic_par
    |_ timeshare_mem_bound
```

- Figure 8: The reproduced figure and experiment results can be found in the `step-artifact-hdl` directory. The file `validation.pdf` should match Figure 8. The values used to generate the plot are provided in the other two CSV files in the same directory.
- Figure 9: The reproduced figure and experiment results can be found in the `dyn_tiling` directory. The file `figure9.pdf` should match Figure 9. The values used to generate the plot can be found in `figure_9_mixtral_b64_raw.csv` and `figure_9_qwen_b64_raw.csv`.
- Figure 10: The reproduced figure and experiment results can be found in the `dyn_tiling` directory. The file `figure10.pdf` should match Figure 10. The values used to generate the plot can be found in `figure_10_mixtral_b1024_raw.csv` and `figure_10_qwen_b1024_raw.csv`.
- Figure 12: The reproduced figure and experiment results can be found in the `timeshare_mem_bound` directory. The file `figure12.pdf` should match Figure 12. The values used to generate the plot are provided in the remaining CSV files in the same directory.
- Figure 13: The reproduced figure and experiment results can be found in the `timeshare_mem_bound` directory. The file `figure13.pdf` should match Figure 13. The values used to generate the plot are provided in the remaining CSV files in the same directory.
- Figure 14: The reproduced figure and experiment results can be found in the `dynamic_par` directory. The file `figure14.pdf` should match Figure 14. The values used to generate the plot are provided in `batch64_interleave_dynamic.csv`.
- Figure 15: The reproduced figure and experiment results can be found in the `dynamic_par` directory. The file `figure15.pdf` should match Figure 15. The values used to generate the plot are provided in `batch_sweep_coarse_vs_dynamic.csv`.
- Figure 21: The reproduced figure and experiment results can be found in the `dynamic_par` directory. The file `figure21.pdf` should match Figure 21. The values used to generate the plot are provided in the remaining CSV files in the same directory.

A.7 Toolchain Customization

Details on how to customize the toolchain (the Python frontend and Rust simulator) can be found in the To customize or extend the toolchain section of the `README.md` file in the step_artifact repository.

B Appendix

B.1 STeP Operator Syntax and Shape Semantics

This section contains the syntax and shape semantics of STeP operators. We express stream types in the form of `Strm<T,a>` where `T` is the data type of the stream and `a` is the rank of the

stream. We will use uppercase letters in the angle brackets (<,>) to denote the data type of the stream and lowercase letters for the stream rank.

We use different uppercase letters to express the available data types for each operator.

- R,R′: Any data type
- A,B: Non-buffer type
- S: Statically sized tile
- SEL: Selector type
- I: [1, 1] tile of integer address data type.

For dynamic routing and merging operators (Table 6), the subscript i in the input and output stream shape is used to specify the shape of the i-th input or output stream. For the Reshape operator, when splitting a dimension higher than the innermost (scalar) dimension, it should be a static dimension divisible by the chunk size. When splitting the innermost (scalar) dimension, there is no restriction on the dimension shape, and it will be accordingly padded.

B.2 Hierarchical Tiling

When mapping to the HDL simulator described in Section 4.5, we apply hierarchical tiling to the tiles in each stream. The larger logical tiles defined at the STeP level are partitioned into smaller physical tiles that match the fabric's compute tile size. Figure 18 shows an example graph transformation for hierarchical tiling. As shown in the graph, STeP operators and the shape semantics can also be used to express hierarchical tiling.

B.3 Dataset

To create batch data from the AzureLLMInference dataset [32], we batch multiple requests within a 5,000-request time window and compute the standard deviation of KV cache lengths for each batch. We experiment with batches whose prompt length standard deviation matches that of the full 5,000-requests, and batches with the top 10% highest and lowest variability.

For the expert routing data in the MoE layers, we run Qwen3-30B-A3B [52] and Mixtral8x7B [17] using the real-world serving trace HH-RLHF [10]. To select representative cases, we measure the standard deviation of expert bin counts across iterations and layers, and choose the one whose deviation is closest to the overall average.

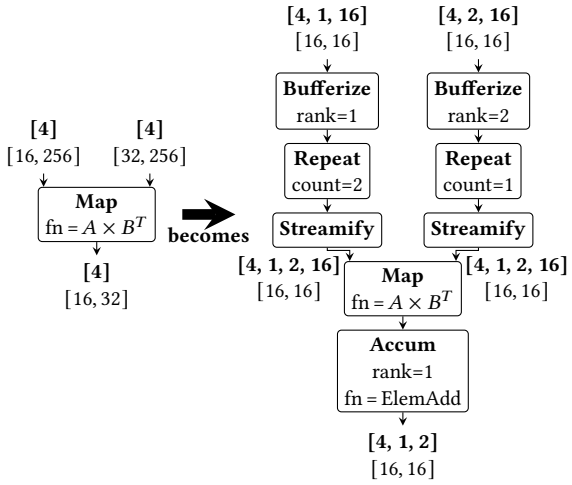

Figure 18. Conversion of STeP $A \times B^T$ map node of large tile size to smaller tile size.

B.4 Dynamic Tiling

In Section 5.2, we only show the Pareto curve for performance and on-chip memory requirement, as the performance and off-chip memory traffic show the same trend due to the application being memory-bound in the simulated hardware configuration. The pareto curve for the off-chip memory traffic and on-chip memory usages is in Figures 19 and 20.

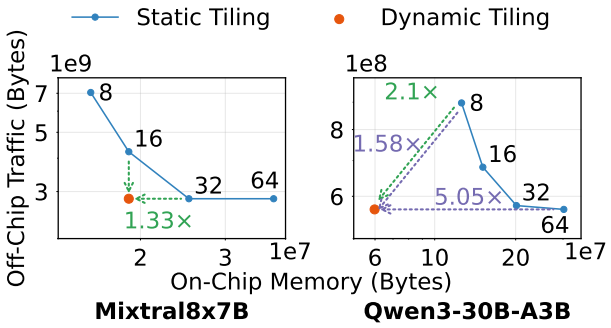

Figure 19. Off-chip traffic and memory requirements of tiling strategies for the batch dimension of each expert (batch = 64). The numbers on the static tiling curve denote tile size. Purple dotted arrows indicate the on-chip memory savings and green dotted arrows indicate speedup.

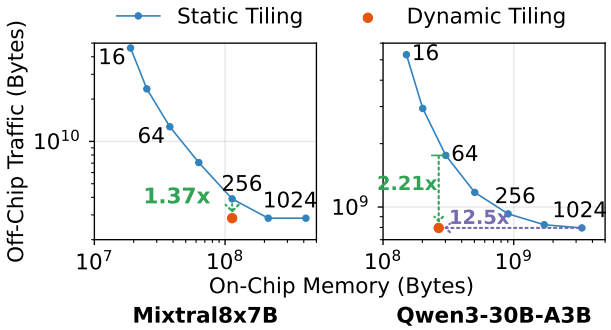

Figure 20. Off-chip traffic and memory requirements of tiling strategies for the batch dimension (batch = 1024). Purple dotted arrows indicate the on-chip memory savings and green dotted arrows indicate speedup.

Operator Signature	In Stream Shape	Out Stream Shape
LinearOffChipLoad<S,R,a,b> (ref: Strm<R,b>, base_addr: int, tiled_in_shape: [int], stride: [int], tiled_out_shape: [int]) → Strm<S,a+b>	$[D_b, \cdots, D_0]$	$[D_b, \cdots, D_0, D'_{a-1}, \cdots, D'_0]$ (a=|tiled_in_shape|)
LinearOffChipStore<S,a> (in: Strm<S,a>, base_addr: int)	$[D_{a-1}, \cdots, D_0]$	
RandomOffChipLoad<I,S,a,b> (raddr: Strm<I,a>, base_addr: int, tiled_in_shape: [int]) → Stream<S,a>	$[D_a, \cdots, D_0]$	$[D_a, \cdots, D_0]$
RandomOffChipStore<I,S,a,b> (waddr: Strm<I,b>, wdata: Strm<S,b>, base_addr: int, tiled_in_shape: [int]) → Stream<bool,a>	$[D_a, \cdots, D_0]$ (waddr) $[D'_b, \cdots, D'_0]$ (wdata)	$[D_a, \cdots, D_0]$

Table 3. STeP off-chip memory operators. The square brackets in the operator signature express a list type.

Operator Signature	In Stream Shape	Out Stream Shape
Bufferize<S,a,b> (in: Strm<S,a>) → Strm<Buffer<S,b>,a-b>	$[D_a, \cdots, D_b, D_{b-1}, \cdots, D_0]$	$[D_a, \cdots, D_b]$ (buffer: $[D_{b-1}, \cdots, D_0]$)
Streamify<S,R,a,b,c> (in: Strm<Buffer<S,a>,b>, ref: Strm<R,b+c>, stride: [int], out_shape: [int]) → Strm<S,|out_shape|+b+c>	$[D_b, \cdots, D_0]$ (data) $[D_b, \cdots, D_0, D'_{c-1}, \cdots, D'_0]$ (ref)	$[D_b, \cdots, D_0, D'_{c-1}, \cdots, D'_0, D''_{\|out\_shape\|-1}, \cdots, D''_0]$

Table 4. STeP on-chip memory operators. For Streamify, if the buffer is dynamically-sized, |out_shape| is replaced with a.

Operator Signature	In Stream Shape	Out Stream Shape
Map<A,B,a> (in: Strm<A,a>, fn: Fn(A)→ B) → Strm<B,a>	$[D_a, \cdots, D_0]$	$[D_a, \cdots, D_0]$
Accum<A,R,a,b> (in: Strm<A,a>, update_fn: Fn(A,R)→R, init_fn: Fn()→R) → Strm<R,a-b>	$[D_a, \cdots, D_b, D_{b-1}, \cdots, D_0]$	$[D_a, \cdots, D_b]$
Scan<A,B,a,b> (in: Strm<A,a>, update_fn: Fn(A,B) → B, init_fn: Fn() → B) → Strm<B,a>	$[D_a, \cdots, D_b, D_{b-1}, \cdots, D_0]$	$[D_a, \cdots, D_b, D_{b-1}, \cdots, D_0]$
FlatMap<A,B,a,b> (in: Strm<A,a>, fn: Fn(A)→Strm<B,b>) → Strm<B,a+b>	$[D_a, \cdots, D_1, D_0]$	$[D_a, \cdots, D_1, D'_b, \cdots, D'_0]$

Table 5. STeP higher-order operators.

Operator Signature	In Stream Shape	Out Stream Shape
Partition<R,SEL,a,b> (in: Strm<R,a>, sel: Strm<SEL,b>, num_consumers: int) → [Strm<R,a-b>]	$[D_a, \cdots, D_0]$ (in) $[D_a, \cdots, D_{a-b}]$ (sel)	$[D^i_{a-b}, D^i_{a-b-1}, \cdots, D^i_0]_i$
Reassemble<R,SEL,a,b> (in: [Strm<R,a>], sel: Strm<SEL,b>) → Strm<R,a+b+1>	$[D^s_b, \cdots, D^s_0]$ (sel) $[D^i_a, D^i_{a-1}, \cdots, D^i_0]_i$ (in)	$[D^s_b, \cdots, D^s_0, D^{sel}_a, D_{a-1}, \cdots, D_0]$
EagerMerge<R,SEL,a> (in: [Strm<R,a>]) → Strm<R,a>, Strm<SEL,0>	$[D^i_a, D^i_{a-1}, \cdots, D^i_0]_i$	$[\sum_i D^i_a, D_{a-1}, \cdots, D_0]$ (data) $[\sum_i D^i_a]$ (sel)

Table 6. STeP routing and merging operators.

Operator Signature	In Stream Shape	Out Stream Shape
Flatten<R,a,min,max> (in: Strm<R,a>) → Strm<R,a-(max-min)>	$[D_a, \cdots, D_{max}, \cdots, D_{min}, \cdots, D_0]$	$[D_a, \cdots, D_{new}, \cdots, D_0]$ $(D_{new} = \Pi_{i=min}^{max} D_i)$
Reshape<A,a,b> (in: Strm<A,a>, chunk_size: int, pad: Option<A>) → Strm<A,a+1>, Strm<bool,a+1>	$[D_a, \cdots, D_b, D_{b-1} \cdots, D_0]$	$[D_a \cdots, \lfloor \frac{(D_b+S-1)}{S} \rfloor, S, D_{b-1} \cdots, D_0]$ (data, padding) (S = chunk_size)
Promote<R,a> (in: Strm<R,a>) → Strm<R,a+1>	$[D_a, \cdots, D_0]$	$[D_{a+1}, D_a, \cdots, D_0]$ $(D_{a+1} = (1\ if\ (D_a > 0)\ else\ 0))$
Expand<R',R,a> (in: Strm<R',a>, ref: Str<R,a>,) b: int) → Strm<R',a>	$[D_a, \cdots, 1_b, \cdots, 1_0]$ (data) $[D_a, \cdots, D_b, \cdots, D_0]$ (ref)	$[D_a, \cdots, D_b, \cdots, D_0]$
Zip<R,R',a> (in1: Strm<R,a>, in2: Str<R',a>)→Strm<(R,R'),a>	$[D_a, \cdots, D_0]$ (in1,in2)	$[D_a, \cdots, D_0]$

Table 7. STeP shape operators.

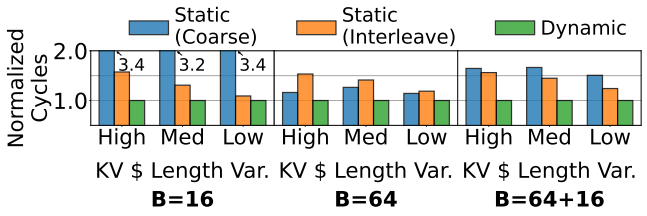

Figure 21. Normalized performance of parallelization strategies relative to dynamic parallelization. For each class, we sample three batches and report the geometric mean performance. KV $ is used as shorthand for KV cache.

To show that dynamic tiling extends the Pareto frontier beyond what is attainable with static tiling, we use the Pareto Improvement Distance metric (PID). The PID measures the distance from a new design point p to a reference Pareto frontier F_B as the smallest worst-objective multiplicative improvement required for some $q \in F_B$ to match p on all objectives. This can be seen as similar to the single-point specialization of the Average Distance from Reference Set (ADRS) metric [9, 26, 51]. We treat both objectives as minimization: cycle count and on-chip memory. Let F_B denote the Pareto-optimal subset of baseline (static) points after removing dominated configurations. For a new point p (e.g., `tile=dynamic`), we measure its distance to the baseline frontier by comparing it to every $q \in F_B$ and computing the smallest multiplicative factor by which a baseline point would need to improve to match p in *both* objectives simultaneously:

$$\mathrm{PID}(p) = \min_{q \in F_B} \max\left(\frac{\mathrm{cycles}(q)}{\mathrm{cycles}(p)}, \frac{\mathrm{mem}(q)}{\mathrm{mem}(p)}\right). \quad (2)$$

Intuitively, for each baseline frontier point q, the inner $\max(\cdot)$ selects the harder objective to match (cycles or memory), and the outer $\min(\cdot)$ picks the baseline point closest to p under this worst-case ratio. This yields a single interpretable number: $\mathrm{PID}(p) > 1$ means p lies strictly beyond the baseline frontier, $\mathrm{PID}(p) = 1$ means p lies on the frontier, and $\mathrm{PID}(p) < 1$ means p is dominated by the baseline frontier.

B.5 Dynamic Parallelization

As shown in Figure 21, static interleave parallelization performs better for smaller batch sizes because the coarse-grained static parallelization can only utilize a portion of the allocated resource when receiving smaller batch sizes. However, for larger batch sizes, static coarse-grained parallelization performs better as it avoids workload distribution being blocked by a single request with a long KV cache. To avoid this blocking, the interleaving static parallelization requires large buffers in front of each parallel region. However, static coarse-grained parallelization will still suffer from the load imbalance across parallel regions. We also simulate the case where a batch with size 64 and 16 is pipelined as micro batches to see the aggregate effect under different batch sizes. Dynamic parallelization consistently outperforms static parallelization across different batch sizes and KV cache length distributions by dispatching work to parallel regions as soon as they become free. Overall, static interleaved parallelization and static coarse-grained parallelization achieve geometric mean speedups of 1.36× and 1.85×, respectively.