Streaming Tensor Program: A streaming abstraction for dynamic parallelism

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

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

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

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

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

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

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

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

Abstract

Dynamic behaviors are becoming prevalent in many tensor applications. In machine learning, for example, the input tensors are dynamically shaped or ragged, and data-dependent control flow is widely used in many models. However, the limited expressiveness of prior programming abstractions for spatial dataflow accelerators forces the dynamic behaviors to be implemented statically or lacks the visibility for performance-critical decisions. To address these challenges, we present the Streaming Tensor Program (STeP), a new streaming abstraction that enables dynamic tensor workloads to run efficiently on spatial dataflow accelerators. 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—dynamic tiling, dynamic parallelization, and configuration time-multiplexing—that adapt to dynamic behaviors while preserving dataflow efficiency. Using a cycle-approximate simulator on representative LLM layers with real-world traces, dynamic tiling reduces on-chip memory requirement by 2.18×, dynamic parallelization improves latency by 1.5×, and configuration time-multiplexing improves compute utilization by 2.57× over implementations available in prior abstractions.

Keywords: Streaming Abstraction, Dataflow Programming Model, Spatial Dataflow Accelerator, Machine Learning

Abstraction	Data Flow	Explicit Data Rate	Explicit Memory Hierarchy	Dynamic Routing & Merging	Dynamic Tiling
Spatial [14]	✗	✗	✓	✗	✗
Revet [26]	✗	✗	✓	✓ (limited)	✗
StreamIt [33]	✓	✓	✗	✗	✗
SAM [12]	✓	✗	✗	✓ (limited)	✓ (limited)
Ripple [8]	✓	✗	✗	✓	✓
STeP	✓	✓	✓	✓	✓

Table 1. Landscape of programming abstractions for SDAs

1 Introduction

Spatial dataflow accelerators (SDAs) [9, 15, 22–24, 36, 37] are reconfigurable architectures with spatially distributed compute and memory units. Due to their high performance and energy efficiency, SDAs have gained popularity as an alternative to general-purpose processors and GPUs across several applications, mainly focusing on tensor applications [9, 15, 22, 23, 26]. Prior work on the compiler and the architecture of SDAs demonstrates that several static tensor applications can be effectively mapped to SDAs [9, 14, 24, 41].

However, the need to support dynamic tensor applications is rapidly growing. As an example, the input data has dynamic or ragged shapes in machine learning (ML) applications due to varying batch sizes, image resolutions, and prompt lengths [20, 21, 28, 38]. This makes efficient parallelization and spatial mapping challenging. Moreover, dynamic control flow is also heavily used in applications with the advent of dataflow applications that sparsely activate parts of the computation graph [4, 18, 39].

Many such dynamic workloads can be characterized as asynchronously running blocks communicating with each other. This characteristic aligns well with the execution model of SDAs, where compute and memory units run asynchronously and communicate via hardware FIFOs. However, unlike the ample software support for expressing static workloads on SDAs, current programming abstractions for SDAs have limited support for accelerating dynamic workloads, as shown in Table 1, leaving performance on the table.

Many prior SDAs use an imperative language [9, 14, 26, 41] as the programming abstraction. While this approach is general, it requires transforming the imperative code to a dataflow representation to match the hardware's execution model. This makes it challenging to exploit the inherent parallelism in the application, as imperative code enforces a sequential order between instructions [41]. Furthermore, imperative languages lack explicit primitives to express asynchronous execution or queueing, which are crucial for optimizing dynamic workloads. The lack of explicit support for asynchrony and queueing forces their implementation with complex control and memory instructions, even though they can be directly mapped to SDA hardware FIFOs with minimal overhead [8].

On the other hand, dataflow abstractions, such as SAM [12] Ripple [8], and StreamIt [33], expose queues at the abstraction level and enable exploiting the inherent parallelism in the application. However, none of them 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 of asynchronous blocks that can contain any imperative code. However, Ripple makes the memory hierarchy implicit in the abstraction. This makes it difficult to express and discover efficient implementations of many widely used dynamic tensor applications, such as ML, where the performance is dictated by the data movement across the memory hierarchy. Furthermore, the opaque data rate at the abstraction level requires lifting the imperative code in each asynchronous block to analyze the program using the data rates.

To address the limitations of prior SDA dataflow abstractions in expressing and optimizing dynamism, we propose the Streaming Tensor Program (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 were unavailable in prior abstractions for SDAs. First, it enables capturing 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-intensive tensor applications and validate the captured metrics with a cycle-accurate simulator (Section 4). STeP also enables expressing optimizations such as dynamic parallelization, configuration time-multiplexing, and dynamic tiling (Section 5), which were not expressible in prior abstractions for SDAs. We evaluate each optimization on representative layers from open-source large language models (LLMs) with real-world traces using a cycle-approximate simulator. Our evaluations show a geometric mean of 2.1× less on-chip memory requirement with dynamic tiling, 2.57× improved compute utilization with configuration time-multiplexing, and 1.5× speedup with dynamic parallelization when compared to the implementations available in prior abstractions. Lastly, we discuss various ways and trade-offs for supporting dynamic features of STeP on SDAs (Section 6).

Overall, our contributions are:

- An asynchronous dataflow abstraction for SDAs (STeP) with full 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 and Related Work

This section provides background on the application, hardware, and related programming abstractions discussed in this paper. First, we explain representative dynamic behaviors in tensor applications, using ML workloads as an example. We then explain the architecture and execution model of SDAs. Lastly, we discuss the limitations in expressing efficient implementations for dynamic tensor applications in prior programming abstractions for SDAs.

2.1 Dynamism in 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 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, is activated for each input activation. With every top-ranked open-source model now adopting the MoE architecture [3, 10, 16, 16, 29, 31, 32, 39][1], efficiently handling the divergence introduced by such control flow has become increasingly important.

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

Dynamic tensor shapes are also common in ML workloads, which arise from external parameters specified at runtime, such as the number of concurrent user requests, image resolution, and sequence length [28, 38]. The control flow described above further amplifies this effect by making the input shape to each expert determined only during execution.

In addition, efficiently parallelizing unevenly sized workloads has become essential. Most state-of-the-art ML models are autoregressive, generating each new output based on the previously produced context. To avoid redundant computation, activations from prior context are stored separately (referred to as *KV caches* in autoregressive models). However, the length of this context can vary widely across users [21, 42], leading to substantial imbalance in both memory footprint and the computation required per request.

2.2 Spatial Dataflow Accelerators

Spatial dataflow accelerators [9, 15, 24, 26, 36, 37] 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. Therefore, imperative abstractions also ultimately lower the code into a dataflow representation to match the hardware's execution model [14, 26, 41]. 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 schedule when data is loaded from one tier of storage to another and when results are written back [11, 14].

2.3 Programming abstractions for SDAs

SDAs can be programmed with either an imperative [9, 14, 26] or a dataflow programming abstraction [8, 12, 33] as listed in Table 1. While the imperative abstractions provide high generality, they lack explicit support for asynchronous execution and queues. The absence of these features in the imperative abstractions incurs overhead when supporting dynamic applications.

Spatial [14] is an imperative programming abstraction for FPGAs and SDAs that uses nested loops and provides explicit control over the memory hierarchy. However, control flow can only occur in restricted places in the program, and all memory constructs must be statically sized, making dynamic behaviors hard to capture. Furthermore, transforming imperative loops into dataflow graphs that can be mapped to the hardware introduces complexity in the compiler [41], and potentially results in suboptimal schedules.
Revet [26] extends Spatial with a dataflow thread abstraction and constructs for dynamic routing and merging, improving support for irregular applications. However, the dataflow thread abstraction forces the data passed to the dynamic constructs to be scalars. This severely limits available parallelism, making it impractical to optimize many dynamic applications that operate on large tensors.

Dataflow abstractions address these limitations with built-in support for dataflow and queueing. However, prior work either focuses only on a specific domain or lacks visibility and control over performance-critical decisions in many dynamic tensor applications.

StreamIt [33] 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 fixed, compile-time–known rates for consuming and producing data in the stream. While this design enables several optimizations based on known data rates, this limits its ability to capture dynamic applications.
SAM [12] 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 [8] is an asynchronous dataflow abstraction, where asynchronous blocks of imperative code communicate via queues. While Ripple can express general asynchronous execution and communication, this design makes the data rates of each asynchronous block opaque, requiring the compiler to deduce them from the imperative code. In addition, the memory hierarchy is implicit in the abstraction. While this 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. In these settings, having visibility and control over the data movement across the memory hierarchy becomes essential for performance.

3 Streaming Tensor Program

Streaming Tensor Program (STeP) is a streaming abstraction for dynamic applications running on SDAs. In this section, we describe the stream representation and operators of STeP. We explain how STeP's features enable efficient implementations that exploit dynamic parallelism and allow the analysis of performance-critical properties of the program.

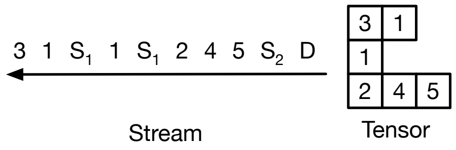

Figure 1. A tensor represented as a rank-2 STeP stream. The *done token* (D) denotes the end of the stream.

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 the 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).

Stream Structure. STeP streams are logically equivalent to zero or more tensors. We draw this connection because higher-order tensors are the basic data structure of many computational workloads. STeP streams embed the logical structure of the corresponding tensor into the data stream using *stop tokens* as shown in Figure 1. We use a similar stop token design to that of SAM [12] because it fits well with the asynchronous dataflow model and allows the size of each dimension 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). At the end of multiple dimensions, we only emit the highest-level stop token.

The logical correspondence between a tensor and an STeP stream provides a foundation for defining shape semantics for streams. These semantics enable analyses and optimizations, and also improve debuggability by exposing dataflow block behaviors at the tensor level and ensuring operator composability. Unlike the shape semantics of streams in synchronous dataflow models [33], which are straightforward due to the fixed data rates, the shape semantics in asynchronous dataflow models require a more 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 $[D_N, \cdots, D_1, D_0]$. To build shape semantics while also expressing dynamic behaviors, we allow each D_i to be either a static-regular, a dynamic-regular, or a ragged dimension. We express the shape of dynamic-regular and ragged dimensions with equations and symbols, such as B_i and C_i in Figure 2. The ragged dimensions have an absorbing property in the equations. If a dimension's shape equation contains the shape of a ragged dimension, that dimension will be treated as a new ragged dimension.

3.2 STeP operators

In this section, we describe STeP operators and their shape semantics. Each operator takes in input streams, operator-specific arguments, and outputs streams.[2]

3.2.1 Off-chip Memory Operators.

Off-chip memory operators express the interface between on-chip and off-chip memory. One of the unique design decisions of STeP compared to other asynchronous dataflow models [8, 12] is making the memory hierarchy explicit. Coupled with the shape semantics, the off-chip memory operators enable capturing metrics that provide performance insights, such as off-chip memory traffic and operational intensity. Furthermore, they expose performance-critical decisions to the user or compiler at the abstraction level.

LinearOffChipLoad loads the input tensor from off-chip memory to on-chip memory in tiles. It supports affine reads over the stored tensor based on the underlying shape, stride, and output shape arguments. The operator has a reference stream that enables the viewed tensor to be dynamically broadcast along the reference stream by triggering an affine read over the tiled tensor on every element in the reference stream. In Figure 2, the first LinearOffChipLoad takes an input stream of shape $[C_i]$ and linearly reads the underlying tensor of shape $[4096, 14336]$, resulting in a $[C_i, 1, 224]$ shaped stream of $[4096, 64]$ shape tiles.

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

RandomOffChipLoad supports random-access reads into tensors stored in off-chip memory. RandomOffChipLoad takes the base address, tile shape, and the underlying tensor shape as arguments. The operator has an input stream that specifies the tile addresses, which are an offset from the given base address.

RandomOffChipStore enables random-access stores to off-chip memory. The operator takes the same arguments

[2]A supplementary syntax and shape semantics are provided in the appendix.

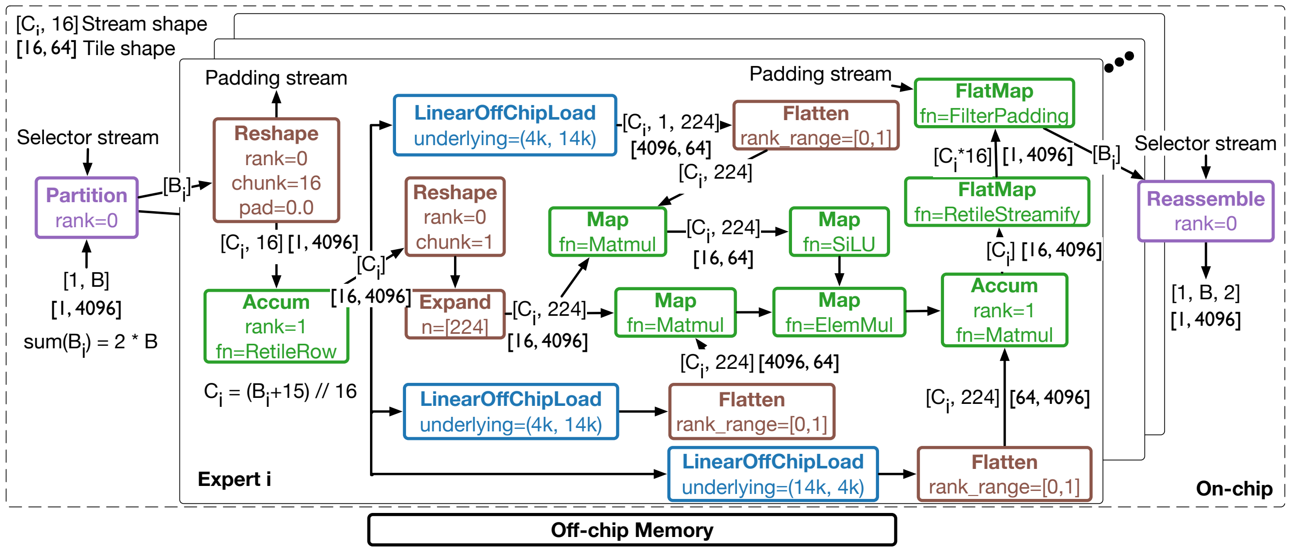

Figure 2. Example STeP graph for an MoE layer in Mixtral8x7B. A token routes to 2 experts, and an expert computes $(SiLU(xW_1) * (xW_3))W_2$. A stream is described with its shape and the shape of tiles on it. Different colors represent different kinds of STeP operators. Some operators have arguments omitted for simplicity. Expand with `n=[224]` is a syntax sugar for static repeating.

as RandomOffChipLoad. The operator takes a write address stream and a write data stream, and outputs a stream specifying that the write has finished.

3.2.2 On-chip Memory Operators.

Based on STeP's design decision to make the memory hierarchy explicit, we introduce on-chip memory operators that convert between streams and on-chip memory. These operators are used to express buffering (parts of) streams on-chip to avoid off-chip memory accesses or recomputation. This expressiveness reveals a vast design space of implementations that trade off on-chip memory requirements and off-chip traffic. Furthermore, the shape semantics of STeP enable capturing this trade-off, providing insight into the vast design space it expresses.

Bufferize reads in data from the input stream and stores it in an on-chip memory in linear order. The operator requires a *bufferize rank* argument. Once the operator finishes creating on-chip memory with the bufferize rank, a reference is emitted to the output stream, and the operator starts accumulating into a new on-chip memory. Given a bufferize rank of b and an input stream shape of $[D_a, \cdots, D_b, \cdots, D_0]$, the output stream shape is $[D_a, \cdots, D_b]$ and the allocated on-chip memory shape is $[D_{b-1}, \cdots D_0]$. STeP allows storing a dynamically-sized regular tensor in on-chip memory. This enables maximizing data reuse with minimal on-chip memory when the application introduces dynamically sized tensors.

Streamify performs an affine read over the tensor stored in on-chip memory based on the stride and the output shape argument. The operator also takes in a *repeat rank* to support dynamic broadcasting of the accessed tensor. Assume a repeat rank of `c(c≥1)`. Let the reference stream have shape $[D_a, \dots, D_0, D'_{c-1}, \dots, D'_0]$ and the data stream have shape $[D_a, \dots, D_0]$, where each element in the data stream is a reference to an on-chip buffer produced by Bufferize. For every element in the reference stream, an affine read is issued repeatedly over the same buffer until all of the inner `c` dimensions of the reference stream are consumed. As a result, the output stream has shape of $[D_a, \cdots, D_0, D'_{c-1}, \cdots, D'_0, D''_{b-1}, \cdots, D''_0]$, where the last `b` dimensions $(D''_{b-1} - D''_0)$ represent the shape of the viewed tensor and the middle `c` dimensions $(D'_{c-1} - D'_0)$ are added due to the broadcasting along the reference stream. STeP supports reading data from on-chip memory with dynamic shapes. In this case, we ignore the stride and output shape arguments, and the operator does a linear read. Bufferize and Streamify together provide full round-trip access to statically- and dynamically-sized on-chip memory.

3.2.3 Dynamic routing and merging operators.

To efficiently support data-dependent control flow and dynamic parallelism, the routing and merging capabilities presented in this section are crucial. Many prior SDA abstractions [14, 26, 33] either lack support for dynamic routing and merging or only support it under specific restrictions that significantly limit the available parallelism. Therefore, STeP introduces dynamic routing and merging operators that can operate on flexible data types.

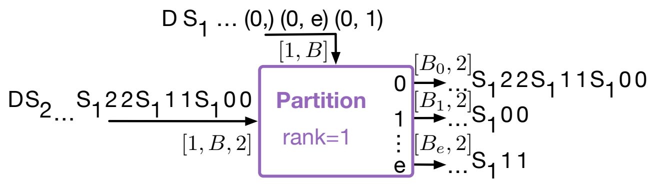

Figure 3. An example of a Partition operator. B_i in each output stream is a newly created dynamic regular dimension.

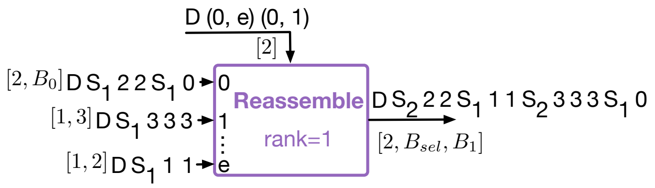

Figure 4. An example of a Reassemble operator. The multihot vector is expressed as tuples. Unless the selector is a k-hot, B_{sel} is a ragged dimension as the selector is multihot.

Partition routes chunks of data from the input stream to multiple output streams based on the data in the selector stream. The operator takes a *partition rank* argument, which is the number of innermost dimensions (i.e., granularity) that are routed at a time to the selected output stream on each selector. Outer-level dimensions of the partition rank are flattened together as shown in Figure 3.

Reassemble takes a list of input streams and merges data from each stream based on the selector stream, as shown in Figure 4. The input streams must all have the same rank b. On every element in the selector stream, the inner b dimensions from the chosen input streams are concatenated, starting from the input stream that arrives first.

EagerMerge is similar to Reassemble but does not take in a selector stream. The inner b dimensions from each input stream are instead concatenated 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. Figure 10 shows its usage in dynamic parallelization.

3.2.4 Higher-order Operators.

These operators take a function supported by the hardware as an argument. They are primarily used to perform arithmetic computations on the stream data or to pack or unpack between data types.

Map expresses element-wise functions (e.g., SiLU in Figure 2) and does not change the shape of the stream.

Accum reduces over multiple inner dimensions of a stream. The operator takes the number of inner dimensions to reduce over, an initialization function, and the update function as arguments. The accumulator is initialized at the beginning of every reduction and gets updated on every element in the input stream with the update function. The operator can also be used to express packing multiple elements in the stream into a single data type, as shown in the Accum block with the RetileRow function in Figure 2. This improves data reuse with more efficient tiling and leveraging tiled compute units such as systolic arrays.

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.

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. This can also express unpacking a larger data type into a stream of smaller data types, as shown in the FlatMap with the RetileStreamify function in Figure 2.

3.2.5 Shape Operators.

Shape operators do not change the content of each element in the stream, but rather change the logical structure (or shape) of the stream by manipulating stop tokens.

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 is padded.

Promote adds a new outermost dimension to the input stream. Given the outermost dimension of the input stream is D_a, the new outermost dimension can be expressed as an equation $(1\ if\ (D_a \geq 1)\ else\ 0)$ if D_a is a dynamic dimension and or 1 otherwise.

Expand expands the consecutive innermost dimensions with shape 1 in the input data stream along the reference stream. Given an input stream shape of $[D_a \cdots, 1_b \cdots, 1_0]$ and a reference stream shape of $[D_a, \cdots, D_b, \cdots, D_0]$, 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. This enables higher-order operators to take functions with multiple arguments.

4 Symbolic STeP Frontend and Performance Model

In this section, we describe the symbolic frontend that implements the STeP shape semantics discussed in Section 3. We then present our performance model, which captures the behavior of STeP graphs on SDAs, along with a cycle-approximate simulator that implements it. We validate the performance model against a cycle-accurate hardware description language (HDL) simulation.

4.1 Symbolic Frontend

We implement a symbolic frontend for STeP in Python. The explicit memory hierarchy and shape semantics in STeP enable capturing metrics such as the on-chip memory requirements, off-chip traffic, and operational intensity of a given STeP graph.

Off-chip Traffic. Every STeP operator implements a function that returns the symbolic expression for the off-chip traffic in bytes made by that operator. For off-chip memory operators (Section 3.2.1), this equation is:

$$(output\ stream\ cardinality)\times(ouput\ stream\ data\ type\ size)$$

To simplify the logic for calculating the cardinality of the output stream in our implementation, we restrict the output stream of off-chip memory operators to contain a ragged dimension only in the two outermost dimensions. In this case, we introduce a new symbol representing the sum of the ragged dimension, which is then multiplied by the size of the inner dimensions. Other operators return zero as they do not interact with off-chip memory.

We obtain the total off-chip traffic by summing up the returned expressions for every operator in the program graph. If we assume that no off-chip memory spilling will occur in other STeP operators, this becomes the amount of off-chip accesses made for the given program implementation, and can be used to derive the operational intensity. If we assume operators can spill to off-chip memory, this metric serves as a lower bound for the off-chip traffic, and derives an upper bound for operational intensity.

On-chip Memory Requirement. Every STeP operator implements a function that returns the symbolic expression for the on-chip memory required in bytes for the operator. The equation for the on-chip memory requirement can be programmed accordingly to reflect the assumptions in the hardware or the requirements of the functions supplied to higher-order operators (Section 3.2.4). In our simulator, we use the following equations for each operator. We use $dtype$ for data type, $||X||$ to denote the cardinality of a Buffer X, and $|x|$ to denote the size of a data type x. Other operators return zero as they do not interact with on-chip memory.

Off-chip memory operators: $|output\ dtype| \times 2$
Bufferize: $|input\ dtype| + ||buffer|| \times |input\ dtype| \times 2$
We multiply by 2, assuming double buffering. Since regular dimensions are only allowed in the buffer shape, the buffer cardinality is the product of the buffer shape's dimensions.
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 only added if the operator is Accum. We count the space to store parts of the input tile and the whole weight tile since matrix multiplication requires repeated access over dimensions in both operands. The value 16 is used to reflect the hardware internally splitting tiles described in the STeP level into smaller tiles that can map to the hardware's compute unit tiles.

The total on-chip memory requirement is obtained by summing up the returned expressions for every operator in the program graph. For SDAs that statically allocate/deallocate on-chip memory, this represents the amount of on-chip memory required to avoid additional off-chip traffic due to limited capacity. For SDAs that dynamically allocate/deallocate on-chip memory, this serves as an upper bound.

STeP's symbolic expression of shapes and operators agnostic to the concrete shape of the input data enables describing the program once and substituting the symbols in the equations for off-chip traffic and on-chip memory requirement with various input shapes or control flow decisions.

4.2 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 [40] 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 [17], a cycle-accurate DRAM simulator. The latency of accessing on-chip memory is factored into the higher-order operators that execute arithmetic functions, using a roofline model. 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:

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

The total FLOPs for a given set of inputs are 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 without passing through an on-chip memory unit.

For routing and merging operators with an input selector stream (Section 3.2.3), data in the selector stream is first dequeued. In the next cycle, data in the chosen input stream is dequeued and enqueued to the output stream with a single cycle latency. The selection stream is blocked until the routing for the current selection is finished. For the remaining operators, we assume a single cycle latency.

4.3 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 [27] layer as the workload since it contains representative computations in ML models such as matrix multiplication, activation function, and row-wise reduction. The STeP graph for the SwiGLU layer corresponds to a single expert in Figure 2. 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. Off-chip memory is modeled as an HBM2 subsystem with 8 stacks and is simulated in Ramulator2 [17]. To match the hardware configurations, we set the on-chip memory bandwidth to 256 (bytes/cycle) and use the same HBM configuration in the STeP simulator.

Validation Methodology To match the granularity of the hardware tiles, we apply a graph transformation to the input STeP graph. The input STeP graph's data, described in larger logical tiles corresponding to the off-chip/on-chip load size, is hierarchically partitioned into smaller physical tiles that match the fabric's compute tile size[3]. 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 HDL model is implemented in Bluespec SystemVerilog and executed in a cycle-accurate BlueSim simulator [2, 19], with off-chip access delays integrated using Ramulator2 library calls. We sweep different tile sizes to cover schedules with varying operational intensities. We measure the total execution time from the first off-chip read to the last off-chip write.

Results As shown in Figure 5, the STeP simulator cycle-count closely matches that of the HDL simulator, with a Pearson correlation of 0.99. 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 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 STeP 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: dynamic tiling, configuration time-multiplexing, and dynamic parallelization. We implement these newly enabled optimizations on representative layers from dynamic ML models and simulate them with real-world traces. The workloads expose a variety of dynamic behaviors, such as runtime-determined tensor sizes, parallelizing unevenly sized workloads, and data-dependent control flow. We investigate the performance and resource utilization of these optimizations using different batch sizes, models, and user requests.

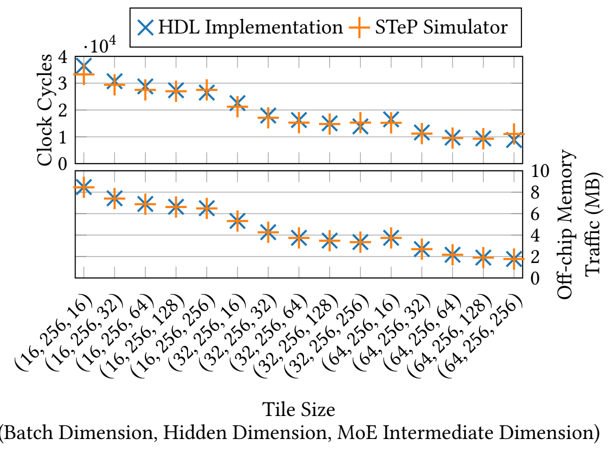

Figure 5. Cycle-count and memory traffic comparison of a SwiGLU Layer with different tile sizes. The full sizes of the batch dimension, hidden dimension, and MoE intermediate dimension are 64, 256, and 512, respectively.

5.1 Methodology

Workload We use two representative layers as our workload: grouped query attention (GQA) [1] and MoE with SwiGLU [27] experts. We choose these two layers because the majority of recent models [3, 4, 6, 10, 13, 16, 16, 18, 29–32, 34, 39] use them as their backbone and only vary parameters such as number of heads, hidden dimensions, number of total experts, and selected experts per token.

In the evaluation, we configure the GQA to be the same as Qwen3-30B-A3B [39] and use the FlashAttention implementation [5, 25]. The KV cache length for each batch is sampled from the AzureLLMInference dataset [21]. We analyze 5,000 requests within a time window, forming batches with varying prompt length distributions. We then measure the standard deviation of prompt lengths in each batch and conduct experiments using batches with average, highest, and lowest variability.

For the MoE layer, we configure it to match Qwen3-30B-A3B [39] and Mixtral8x7B [13] and use the expert routing data collected by running these models using a real-world request traces from HH-RLHF [7]. 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.

We write each design point using STeP's Python frontend described in Section 4.1. Although we present STeP as a

[3]An example transformation is shown in Figure 12 in the Appendix.

programming abstraction in this paper, it can also function as an intermediate representation for compilers. The described optimizations could be automated as compiler passes in the future.

Simulator Setup We set the bandwidth of each on-chip memory unit as 64 (bytes/cycles), which matches the configuration in existing reconfigurable dataflow accelerators [22, 24], which are a class of SDAs. The off-chip memory bandwidth is set to 1024 (bytes/cycle), assuming a HBM configuration of 32 channels. The HBM configuration is set to match the memory system in the latest reconfigurable dataflow accelerator [22]. The amount of compute bandwidth (FLOPs/cycle) allocated is identical across parallel regions and experts.

Baseline Design We chose Revet [26] as our baseline since it has the most extensive support for control flow and flexible parallelism among programming abstractions for SDAs with explicit control over data transfers. We use STeP to implement schedules expressible in Revet as a baseline for each optimization. The parallelization and tiling strategies for statically-sized dimensions are identical, and the allocated compute bandwidth per node is identical across the baseline and optimized implementations.

Evaluation Metrics We compare the newly enabled optimizations against the baseline by simulating them using the cycle-approximate simulator described in Section 4. Along with performance, we also compare metrics such as off-chip traffic and on-chip memory requirements using the symbolic equation described in Section 4.1 for each metric in the STeP operators.

5.2 Dynamic Tiling

Dynamic tiling is a scheduling strategy where the size of a tile in a data stream is determined at runtime, rather than fixed at compile time. In the context of mapping MoEs to SDAs, it means grouping tokens routed to each expert into tiles whose size adapts to the actual number of tokens per expert in each batch. Unlike static tiling, which fixes tile sizes in advance, dynamic tiling lets each tile exactly match the active workload. This flexibility allows for high data reuse with minimal memory overhead, whereas static tiling forces a trade-off. Small tiles lead to frequent off-chip reloads, while large tiles waste on-chip memory by padding unused space. Furthermore, the optimal tile size varies depending on the model and user prompts. Dynamic tiling enables maximizing data reuse consistently under such dynamic changes.

STeP supports dynamic tiling for MoE workloads with two key capabilities: (i) tiles in a stream may have symbolic shapes determined at runtime, (ii) its dynamic routing and merging operators (Section 3.2.3) can operate on tiled streams, and (iii) data movement across the memory hierarchy is made explicit in the abstraction. Leveraging these features, tokens routed to each expert are accumulated into dynamically sized tiles using the Accum operator,

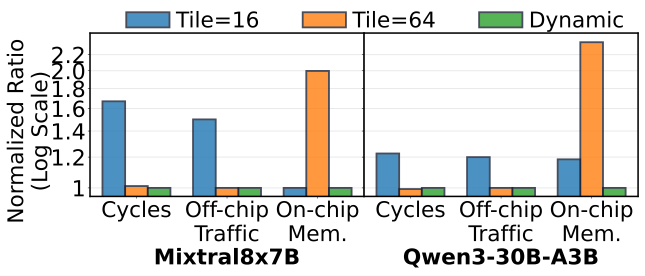

Figure 6. Performance and memory requirements of tiling strategies for gathering tokens per expert. (batch size = 64)

where each resulting tile stream feeds a STeP subgraph implementing the expert's computation. We obtain the dynamic tiling STeP graph by replacing the first Reshape operator in Figure 2 with a Promote. This design maintains SIMD parallelism across the hidden dimension of tokens and maps naturally onto tiled compute units (e.g. systolic arrays). It also accommodates per-batch variations in token-to-expert assignments without padding or static-shape constraints. In contrast, MoE implementations in Revet rely on static tiling, since enabling dynamic tiling would require processing each token as a scalar operation, significantly limiting performance by restricting the ability to exploit parallelism across larger dimensions.

In Figure 6 and Figure 7, Revet corresponds to the design points with static tile sizes. Under static tiling, choosing a small tile size increases the off-chip traffic, as the expert weight matrix has to be loaded several times if the expert has many tokens routed to it. Maximizing weight reuse requires the tile size to match the batch size, which increases on-chip memory usage due to excessive padding. Due to this trade-off, static tiling requires sweeping across tile sizes to find a balanced design point. However, in environments where the batch size varies, this approach is not applicable anymore, since the optimal tile size shifts with the batch size. For example, for Qwen3-30B-A3B in Figure 6, when the batch size is 64, a tile size of 16 will achieve less than 25% overhead with minimal on-chip memory usage. However, when the batch size increases to 1024, the tile size must be at least 256 to have less than 25% performance overhead. Dynamic tiling avoids this trade-off by adapting tile sizes at runtime, achieving higher performance with lower on-chip memory usage. In summary, dynamic tiling shows a geometric mean of 1.45× speedup and 2.18× less on-chip memory requirement across different models and batch sizes.

5.3 Configuration Time-multiplexing

Configuration time-multiplexing is an optimization that time-multiplexes a configuration across different branches with identical computation structure in applications with data-dependent control flow. In the context of executing MoE layers on SDAs, a configuration is time-multiplexed across

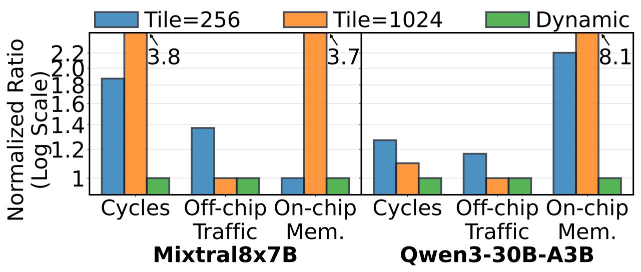

Figure 7. Performance and memory requirements of tiling strategies for gathering tokens per expert. (batch size = 1024)

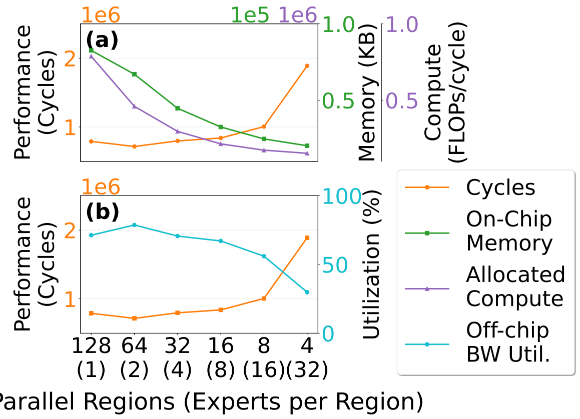

Figure 8. Resource usage and performance for the MoE layer in Qwen3-30B-A3B with different degree of time-multiplexing. (batch size = 64)

experts dynamically by routing inputs and weights accordingly. We observe that when running batched decoding on large-expert models (128+ experts), several experts are not selected within a batch. When experimenting with Qwen3-30B-A3B with a batch size of 64, approximately half of the experts receive no tokens in most of the layers and iterations. Configuring a dedicated region for each expert results in several compute resources remaining idle due to their sparse activation.

STeP enables configuration time-multiplexing with two key capabilities: (i) a dynamic merging operator (EagerMerge) that merges multiple streams and sends them to the downstream pipelines as soon as they are available, and (ii) the explicit data movement across the memory hierarchy. For the MoE layer shown in Figure 2, configuration time-multiplexing is realized by connecting the first Accum operators of multiple experts to an EagerMerge. Each EagerMerge has its own configured subgraph for expert computation, and the three LinearOffChipLoad is replaced by a RandomOffChipLoad, which uses the Selector from EagerMerge to fetch the appropriate tiled weight for the chosen expert.

Leveraging these features, fewer parallel expert regions than the total number of experts can be configured, which enables saving on-chip resources in MoE models with a large number of experts. The flexibility of EagerMerge and stream data type in STeP allows exploiting parallelism along the hidden dimension and the batch dimension in each expert, and the computation can be mapped to tiled compute units without complicated lifting. On the contrary, Revet resorts to the design with dedicated regions for each expert, as it can only be hacked to support the behaviors of EagerMerge by mapping the collected token stream for each expert to a single scalar stream. This severely limits performance by preventing the use of vectorized or tiled compute units.

To evaluate the resource-saving benefits of configuration time-multiplexing, we conduct an ablation study by varying the number of experts sharing the same configured region. We first apply the optimization to the MoE layer in Qwen3-30B-A3B using static tiling of tile size 32 for the batch dimension of each expert, as this is the tile size that balances the off-chip traffic and on-chip memory for static tiling in Qwen3-30B-A3B with batch size 64. As shown in Figure 8(a), the configuration time-multiplexing enables similar performance while using less on-chip memory and compute resources. This frees up the compute resource allocated for idle experts to be used for other computation. However, the compute utilization starts to drop as the number of parallel regions becomes too small. As shown in Figure 8(b), this is due to the decrease in off-chip memory bandwidth utilization, which directly results in a slowdown in performance as this application is memory-bound.

We also evaluate the effect of configuration time-multiplexing with different tiling strategies. For static tiling, the optimization can improve the compute utilization by 2.64× with less than 1% performance overhead (Figure 9(a)). Configuration time-multiplexing can also be applied with dynamic tiling. When dynamically tiling the batch dimension of experts according to the number of tokens routed to each expert, it can further improve performance beyond what is achievable with static tiling. As shown in Figure 9(b), configuration time-multiplexing improves the compute utilization by 2.51× with less than 5% performance overhead for dynamic tiling.

5.4 Dynamic Parallelization

Dynamic parallelization is an optimization that can improve performance by balancing the load across spatially parallel regions when parallelizing a dimension with unevenly-sized workloads. In ML workloads, unevenly-sized workloads appear in the attention [35] computation in Transformer layers during decoding due to the variation in the KV cache length of each request. Static parallelization may hurt performance because unevenly sized workloads can leave some compute resources idle. Dynamic parallelization dispatches workload to idle parallel regions in a greedy manner. This keeps resources busy by dispatching work as soon as downstream

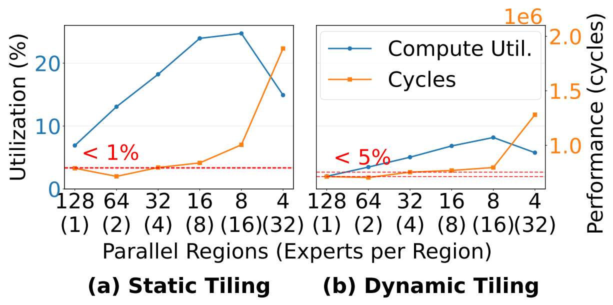

Figure 9. Resource utilization for the MoE layer in Qwen3-30B-A3B with different tiling strategy for the batch dimension (batch size = 64). Dynamic tiling has lower compute utilization compared to static tiling because static tiling has 3.81× higher total FLOPs due to padding. The tile size used for static tiling is 32.

pipelines become available, thereby balancing the load across parallel regions, even when workload sizes are non-uniform.

STeP enables dynamic parallelism through its dynamic routing and merging operators (Section 3.2.3) that can operate on various stream data types. For GQA, STeP can implement dynamic parallelization as shown in Figure 10. Each request is routed to parallel regions using Partition. The selector stream for the Partition is constructed by merging a stream from FlatMap for the initial iteration's round-robin distribution and a stream from EagerMerge that signals which parallel region is available. A Partition is added after EagerMerge to filter out the signals from the last iteration.

As dynamic parallelization also requires the behavior of EagerMerge, implementing dynamic parallelization in Revet suffers from a similar problem as in configuration time-multiplexing (Section 5.3). Therefore, the parallelized GQA implementation in Revet would adopt one of the static parallelization strategies.

To evaluate how dynamic parallelization can improve performance, we compare three different parallelization strategies with varying batch sizes and KV cache length distribution. Within a head, the batch dimension is parallelized by four. Coarse-grained static parallelization statically sets the number of requests processed in each parallel region (in our implementation, we use 16). Interleaving static parallelization distributes requests in a round-robin manner.

As shown in Figure 11, interleaving static parallelization performs better for smaller batch sizes (B=16) 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 (B=64), coarse-grained static 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, it 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. On average, it delivers a 1.5× geometric mean speedup compared to the designs available in Revet.

6 Discussion on supporting dynamic STeP features in SDAs

This section discusses possible approaches to support the dynamic features in STeP on SDAs. Prior work on SDA architecture design [9, 15, 26, 36, 41] demonstrates various implementations of STeP's dynamic features, providing a way to realize the benefits of the optimizations that STeP enables. We leave an optimal hardware SDA design for STeP as future work.

6.1 Stop Tokens

Embedding stop tokens in data streams Prior SDAs [15, 26] demonstrate two different ways of implementing the control-embedded streams. Revet [26] encodes the control tokens using an out-of-band encoding that adds a few additional bits per on-chip vector to encode the stop token level. The vectors are 512-bit (16 × 32-bit) large and four bits are used to encode the stop token level, assuming the maximum levels are less than 15. Onyx [15] uses a 16-bit data path with an additional bit that encodes whether it is a stop token. While Revet couples the stop tokens with the data, Onyx separates data and stop tokens, where the 16-bit data path is used to encode either data or a stop token level.

Architectural support for processing stop-tokens Processing control-embedded streams can be done by either re-purposing existing hardware units or designing a new dedicated state machine to process stop tokens. Revet is mapped to the Aurochs SDA [36], which has compute and memory units with counter chains that serve as programmable loop controllers. Revet re-purposes the counter chains to inject stop tokens into the stream. Onyx, on the other hand, is a clean-slate design, which adds new dedicated state machines to process both the data and stop tokens.

6.2 Dynamic Routing and Merging

STeP's routing and merging operators describe control flow divergence and convergence. SDAs usually implement these by physically laying out all possible datapaths and data-dependently activating the right ones at runtime. This can be implemented either within the reconfigurable compute units or directly in the reconfigurable interconnect fabric. The SARA compiler [41] exemplifies the first approach for branches in imperative code. The branch predicate is turned

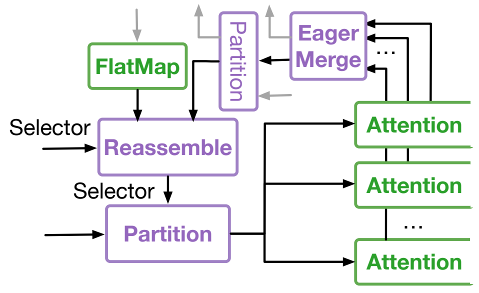

Figure 10. The STeP graph for dynamic parallelization. Shape operators omitted for simplicity.

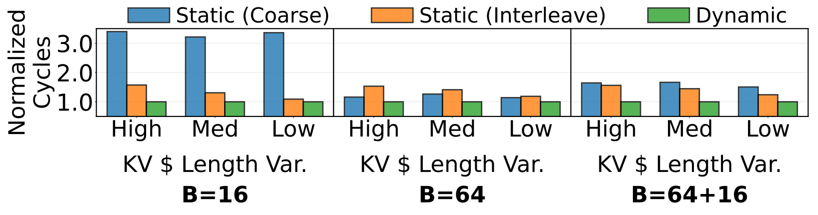

Figure 11. Normalized performance of parallelization strategies relative to dynamic parallelization. For each class, we sample three batches and report the geometric mean performance. KV $ is used as shorthand for KV cache.

into a router that can selectively enqueue data into two datapaths, which represent the true and false conditions. This design isolates dynamic routing decisions within the compute units. Alternatively, RipTide [9] embeds its router into the network-on-chip interconnect. Its routing resources are used for both static and dynamic routing decisions, reusing resources and freeing up the compute units for other work. An accelerator targetable by STeP could use either approach.

6.3 Dynamic memory and tiling

Runtime-determined tensor sizes To handle tensor sizes that are unknown at compile time, the memory system must allocate space at a fixed granularity independent of stream length and maintain mappings between stream references and their memory addresses. To support arbitrary sizes without an upper bound, the memory system should also have a spill mechanism to handle streams that exceed the local memory capacity. If multiple streams are allocated and deallocated concurrently, noncontiguous allocation is required to avoid fragmentation.

Architectural support for dynamic tensor sizes One possible approach is a hardware-managed mapping cache (e.g. a linked list) that translates stream references to a sequence of noncontiguous physical addresses. With 512 KB of local memory per unit [22], the mapping cache requires less than 30 KB of metadata (≈6% overhead), comparable to the tag overhead in conventional caches. Spilling can be handled by the hardware using a similar mechanism to Ripple [8], where both data and the corresponding next pointer are automatically spilled. Since pointers are relatively small compared to a tile, this overhead is minimal. Software-managed scratchpads, as used in GPUs, can reduce hardware complexity but risk fragmentation. Each approach trades off flexibility, metadata overhead, and hardware complexity.

7 Conclusion

We introduced the Streaming Tensor Program, a streaming abstraction designed for dynamic tensor applications on spatial dataflow accelerators. STeP enables optimizations for dynamic workloads that were not expressible in prior abstractions for spatial dataflow accelerators. The flexible stream structure and shape semantics also enable capturing performance-critical metrics at the abstraction level, opening opportunities for further optimizations. As a dataflow abstraction that treats dynamism as a first-class principle, we envision STeP will allow exploring richer forms of dynamism in both applications and hardware architectures.

8 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. Olivia Hsu was supported in part by an 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] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv:2305.13245 [cs.CL] https://arxiv.org/abs/2305.13245

[2] 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.

[3] 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).

[4] Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. 2024. DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models. arXiv:2401.06066 [cs.CL] https://arxiv.org/abs/2401.06066

[5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages.

[6] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[7] Deep Ganguli, Liane Lovitt, Jackson Kernion, Amanda Askell, Yuntao Bai, Saurav Kadavath, Ben Mann, Ethan Perez, Nicholas Schiefer, Kamal Ndousse, Andy Jones, Sam Bowman, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Nelson Elhage, Sheer El-Showk, Stanislav Fort, Zac Hatfield-Dodds, Tom Henighan, Danny Hernandez, Tristan Hume, Josh Jacobson, Scott Johnston, Shauna Kravec, Catherine Olsson, Sam Ringer, Eli Tran-Johnson, Dario Amodei, Tom Brown, Nicholas Joseph, Sam McCandlish, Chris Olah, Jared Kaplan, and Jack Clark. 2022. Red Teaming Language Models to Reduce Harms: Methods, Scaling Behaviors, and Lessons Learned. arXiv:2209.07858 [cs.CL] https://arxiv.org/abs/2209.07858

[8] 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

[9] 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

[10] 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).

[11] 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.

[12] 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

[13] 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, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] https://arxiv.org/abs/2401.04088

[14] 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. *SIGPLAN Not.* 53, 4 (June 2018), 296–311. doi:10.1145/3296979.3192379

[15] 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

[16] 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).

[17] 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.

[18] 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/

[19] 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.

[20] 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

[21] 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.

[22] 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

[23] 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

[24] 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

[25] Markus N. Rabe and Charles Staats. 2022. Self-attention Does Not Need $O(n^2)$ Memory. arXiv:2112.05682 [cs.LG] https://arxiv.org/abs/2112.05682

[26] 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

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

[28] 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

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

[30] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. 2025. Gemma 3 Technical Report. arXiv:2503.19786 [cs.CL] https://arxiv.org/abs/2503.19786

[31] 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).

[32] 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).

[33] 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.

[34] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[35] 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).

[36] 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

[37] 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

[38] 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

[39] 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).

[40] 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

[41] 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

[42] 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 Appendix

A.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 5), 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.

A.2 Hierarchical Tiling

When mapping to the HDL simulator described in Section 4.3, 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 12 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.

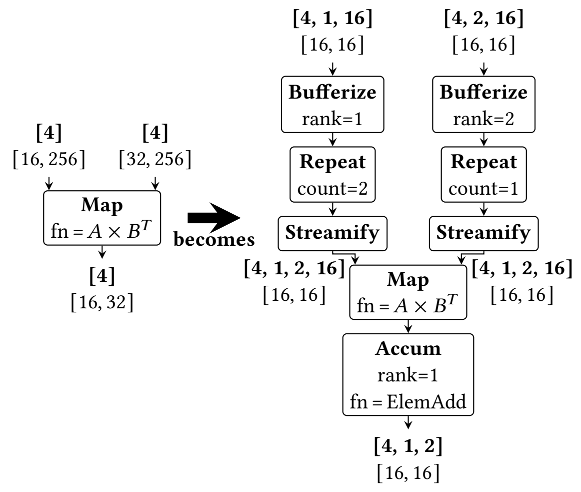

Figure 12. Conversion of STeP $A \times B^T$ map node of large tile size to smaller tile size.

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 2. 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 3. 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 4. 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 5. 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)>`	$[D_a, \cdots, D_{max}, \cdots,$ $D_{min}, \cdots, D_0]$	$[D_a, \cdots, D_{new}, \cdots, D_0]$ $(D_{new} = \Pi^{max}_{i=min} D_i)$
Reshape`<R,a,b> (in: Strm<R,a>, chunk_size: int, pad: Option<R>) → Strm<R,a+1>, Strm<bool,a+1>`	$[D_a, \cdots, D_b,$ $D_{b-1} \cdots, D_0]$	$[D_a \cdots, \left\lceil \frac{(D_b+S-1)}{S} \right\rceil, S,$ $D_{b-1} \cdots, D_0]$ (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) 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 6. STeP shape operators.

A.3 Configuration Time-multiplexing

To apply the configuration time-multiplexing optimization, the EagerMerge operator will send the tokens collected for each expert to the consumer as soon as they arrive. The graph conversion to apply this optimization can be found in Figure 13. The Expert node in the graph can be seen as a STeP subgraph for expert computation. The LinearOffChipLoads used in the expert subgraph to load expert weights will be replaced with a RandomOffChipLoad which uses the selector from EagerMerge as the expert weights to load are determined in runtime.

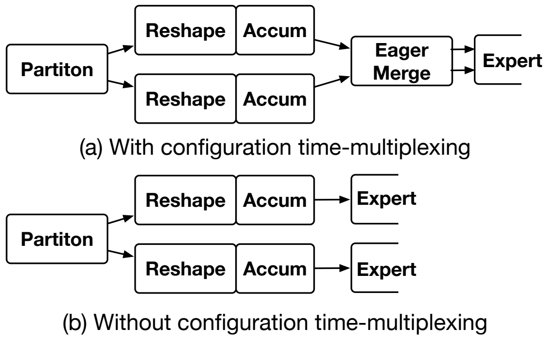

Figure 13. STeP graph with (a) and without (b) configuration time-multiplexing.