USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

TileLoom: Automatic Dataflow Planning for Tile-Based Languages on Spatial Dataflow Accelerators

Wei Li, Zhenyu Bai, Heru Wang, Pranav Dangi, and Zhiqiang Zhang, *National University of Singapore;* Cheng Tan, *Arizona State University and Google;* Huiying Lan, *Lumai Ltd.;* Weng-Fai Wong and Tulika Mitra, *National University of Singapore*

https://www.usenix.org/conference/osdi26/presentation/li-wei

This paper is included in the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation.

July 13–15, 2026 • Seattle, WA, USA

ISBN 978-1-939133-55-7

Open access to the Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

TileLoom: Automatic Dataflow Planning for Tile-Based Languages on Spatial Dataflow Accelerators

Wei Li[1,†], Zhenyu Bai[1,†,✉], Heru Wang[1,†], Pranav Dangi[1,†],
Zhiqiang Zhang[1], Cheng Tan[2], Huiying Lan[3], Weng-Fai Wong[1], Tulika Mitra[1]

[1]School of Computing, National University of Singapore
[2]Arizona State University and Google, [3]Lumai Ltd.

{liwei01, zhenyu.bai, heru.wang, dangi}@nus.edu.sg
t0937444@u.nus.edu, chengtan@asu.edu, huiying.lan93@gmail.com dcswwf,dcstm@nus.edu.sg,

Abstract

Spatial dataflow accelerators are a promising direction for next-generation computer systems because they can reduce the memory bottlenecks of traditional von Neumann machines such as CPUs and GPUs. They organize computation around explicit, compiler-managed data movement over on-chip networks, allowing operands to be forwarded directly between processing elements and reducing reliance on high-latency, bandwidth-limited global shared memory. However, their performance depends strongly on how workloads are mapped to hardware. Naive mappings can perform poorly, and most users rely on hand-tuned vendor libraries. Thus, despite their potential for high performance, energy efficiency, and cost efficiency, limited programmability remains a major barrier to wider adoption.

This paper presents *TileLoom*, an MLIR-based end-to-end framework that compiles tile-based programs, such as Triton kernels, onto spatial dataflow architectures. Unlike compiler frameworks that focus on optimizing code generation *within* a single tile, *TileLoom* distributes tile instances across spatially distributed cores and exploits the on-chip network and distributed memories to increase data reuse and reduce communication. *TileLoom* introduces a hardware representation that captures interconnect topology, memory hierarchy, and compute capabilities, enabling both architecture-specific optimizations and support for diverse spatial dataflow targets. In experiments on two generations of Tenstorrent systems, *TileLoom* achieves performance comparable to vendor libraries on various kernels.

†These authors contributed equally.
✉Zhenyu Bai is the corresponding author (zhenyu.bai@nus.edu.sg).

1 Introduction

Modern high-performance workloads, especially deep learning workloads, are highly data-intensive [5, 15]. For many of these workloads, the main bottleneck is not compute, but memory bandwidth [12, 27, 43, 70]. As process technology scales, we can place more arithmetic units on a chip, but off-chip memory bandwidth and capacity do not scale at the same rate [7, 46]. Each DRAM access costs much more energy than an arithmetic operation [22]. On-chip SRAM also becomes relatively more expensive in area and power as we move to smaller technology nodes [18, 22]. Together, these trends make it hard for conventional, memory-centric von Neumann architectures such as CPUs and GPUs to keep their compute units busy.

Spatial dataflow architectures are emerging as a strong alternative. Systems such as Tenstorrent [3], Cerebras [8], Graphcore [26], SambaNova [55, 58], Groq [1, 2], Meta's MTIA [14], AWS's Trainium [6], and Tesla's Dojo [19] organize computation around explicit, often software-controlled data movements over on-chip networks and buffers, reducing reliance on large shared caches and high-latency off-chip memories. When data is passed from core to core over short on-chip wires, the energy per bit and latency can be much lower than going back and forth to a shared cache or off-chip memory [4, 21, 22]. Figure 1 shows a representative spatial dataflow accelerator architecture from Tenstorrent: a 2D grid of cores, each typically a SIMD or vector engine, often with a matrix unit, plus a local scratchpad, connected by a packet-switched mesh NoC. 64 cores can concurrently access their local scratchpads, yielding an aggregate peak bandwidth of roughly 24.5 TB/s—substantially higher than the 6 TB/s L2 bandwidth of the NVIDIA H100 [60]. This abundant per-core bandwidth significantly alleviates memory bottlenecks for bandwidth-bound operators.

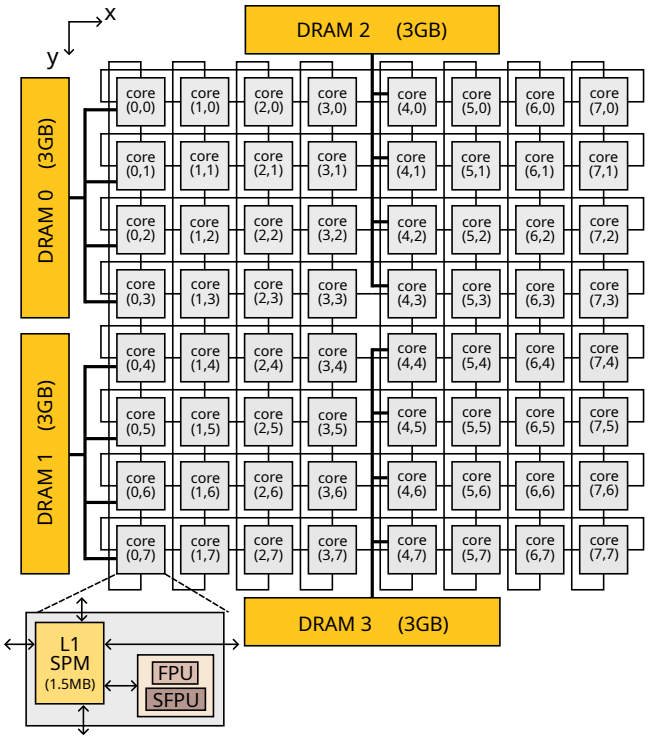

Figure 1: An example 2D-mesh spatial dataflow architecture, modeling Tenstorrent-Wormhole system.

While the structure of spatial dataflow accelerators can provide high efficiency, it also creates a serious programmability problem [16,48,56]. Performance depends heavily on how the workload is mapped: which cores execute which parts of the computation, how data is partitioned across local memories, and how traffic is scheduled on the network [17,38,56,71,73]. A naive mapping can cause severe load imbalance, network congestion, underutilized cores, or excessive off-chip memory traffic, resulting in poor performance and energy efficiency [17,38,71,73]. As a result, compiling high-level programs onto dataflow architectures is challenging, and non-experts struggle to write efficient low-level programs for them. Instead, users rely on vendor-provided, hand-optimized libraries by experts that implement a small set of popular kernels, such as matrix multiplication, convolutions, and attention, with carefully engineered mappings [11,28,34,36,45]. This dependence limits both portability across architectures and the ability to experiment with new kernels or model structures [11,34,45].

A common abstraction for parallel accelerators is the grid–block–thread model popularized by CUDA's grid-of-thread-blocks interface [30,49]. The programmer decomposes the problem into blocks and launches a grid of block instances across the device; each block, such as a CUDA thread block or OpenCL work-group, performs the same computation on a different region of the input or output tensor, and the grid covers the full problem domain [29,49]. Similar grid-based tiling models appear in high-level systems such as Halide and TVM [10,57], which treat tile shapes and launch configurations as schedule parameters and search over them to improve locality and parallelism.

On conventional, memory-centric architectures such as GPUs, the grid level is managed by hardware [30,47,49]: a hardware scheduler dynamically assigns blocks to Streaming Multiprocessors (SMs), and a shared cache hierarchy implicitly captures most data reuse across blocks [25,47]. This leaves little room for compilers to control grid-level placement, execution order, or communication. By contrast, the block–thread level is software-defined: the compiler or programmer decides how work within a block is mapped onto threads, warps, and intra-SM resources. CUDA exposes fine-grain control over thread organization, shared-memory layout, and synchronization, but exploiting it well requires careful tuning and detailed knowledge of GPU microarchitecture. To make this block-level programming more accessible, languages such as Triton [61], TileLang [66], CuTile [50], Tilus [13], and Taichi [23] let users express kernels in terms of high-level tile operators that define the work of a single block, while their compilers lower these operators onto intra-core or intra-SM resources.

Spatial dataflow accelerators change this picture. These architectures distribute compute and scratchpads across large arrays of processing elements connected by a high-bandwidth on-chip network, often without a large, unified cache that automatically exploits cross-core reuse [33,54,63]. Instead of offloading communication and placement decisions to hardware caches and schedulers, they expose richer inter-core programmability to software. Achieving good performance therefore requires a compiler or programmer to decide not only how to implement the tile program on a single core, but also how to place tile instances across cores and schedule them in time so that data can be forwarded or multicast efficiently over the NoC [33,51,63] and memory system. This inter-core programmability creates an enormous mapping design space: valid mappings are combinatorially large, and different mappings expose different reuse patterns and communication costs [51,54]. In today's systems, these decisions are baked into vendor-specific compilers and libraries [9, 54, 58, 63, 64, 75], which encode architecture-specific strategies for placement, routing, and pipelining, but are time-consuming to develop and do not generalize easily to new kernels, models, or hardware generations.

By giving more control to software, spatial dataflow architectures are harder to program but can simplify hardware and improve efficiency. Crucially, **as the hardware becomes more transparent to software by exposing explicit cores, networks, and memories instead of opaque caches and schedulers, dataflow architectures become more predictable than memory-centric architectures.** With sufficient hardware information, **a compiler can more easily**

reason about the performance of different schedules and deduce good static mappings. Building on this insight, we propose ***TileLoom*, a compiler framework that supports end-to-end execution of tile-based programs on spatial dataflow systems.** Our goal is to let users write kernels in a tile-wise language, such as Triton, while the compiler maps the logical grid onto the physical array of cores, schedules tile execution in time, and orchestrates data movement and reuse over the NoC and distributed memories. In other words, *TileLoom* takes on responsibilities typically handled by hardware schedulers and runtimes on GPUs, moving these decisions to compile time for spatial dataflow systems. To support a broad range of dataflow architectures, *TileLoom* introduces a hardware description that models interconnect topology, memory hierarchy, and compute resources. Given this description, the compiler searches for mappings that balance load, improve data reuse, and respect network and memory constraints across architectures. We evaluate our approach on Tenstorrent systems and show that it can match or outperform vendor-provided handwritten libraries on key kernels.

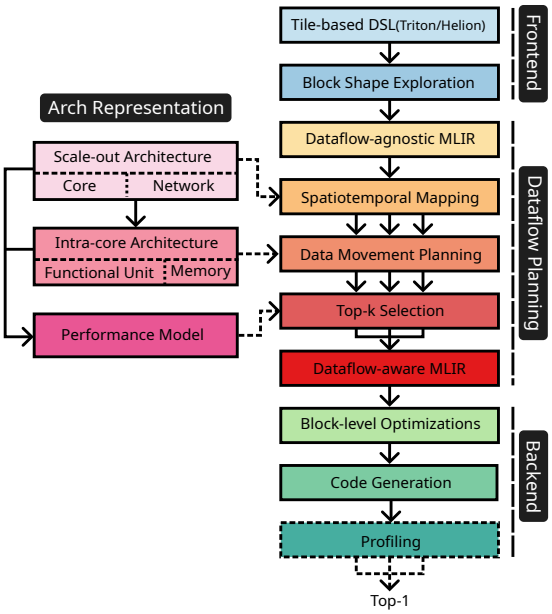

Figure 2: *TileLoom* framework overview.

2 Framework

2.1 Overview

TileLoom compiles a kernel written in a tile-based DSL (currently supporting Triton and Helion) into an executable for a target spatial dataflow architecture.[1] As shown in Figure 2, the compiler stack is organized around three main components: a front-end that lower tile-level kernels into a standard dataflow-agnostic MLIR representation that we propose; a dataflow planning stage that decides spatiotemporal mappings, data movements and generate candidates in a standard dataflow-aware MLIR representation; a back-end that generates hardware-specific executables for each core, all guided by the multi-level architecture representation and performance model.

The **front-end** takes as input a tile-level kernel and a description of how that kernel is scaled out over the full problem (the launch grid). It explores candidate *block shapes*: tile sizes and layouts, similar to the conventional auto-tunning process, and constructs corresponding programs. These candidate programs are then lowered into an MLIR-based intermediate representation and passed through normalization passes so that they share a common structure suitable for the dataflow planning pipeline. At this point, the computation is represented in a uniform, dataflow-agnostic IR affine + linalg + scf + arith): it encodes the tile and grid structure, but does not yet commit to any particular mapping onto the hardware.

The **dataflow planning** stage determines how this logical grid of tiles is realized on the target architecture. Guided by the hardware description, *TileLoom* explores spatiotemporal mappings of tile instances onto cores and time. For each candidate mapping, it performs data reuse analysis to identify which tiles can share data over the NoC or be reused over time via buffering, and it derives concrete data-movement plans: where each tensor tile is allocated, when it is copied, and how it is broadcasted or shared over the buffers. Together, the spatiotemporal mapping and the data-movement plan generate a design space of potential optimal dataflow planning candidates.

To model arbitrary dataflow architectures, the **architecture representation** is proposed, which provides the key inputs to both the dataflow planning search and the performance model. The scale-out (inter-core) description captures the spatial structure of the core array and interconnect, and guides spatiotemporal mapping decisions. The intra-core description captures the local memory hierarchy and compute resources, and guides decisions about where data is stored and how it is staged. *TileLoom* combines both levels of abstraction to generate **a performance model** that estimates the cost of different data-movement plans, taking into account memory bandwidths, NoC capabilities, and per-core compute throughput. The performance model is used to select the top candidates from the dataflow design space.

After dataflow planning stage, the IR is *dataflow-aware*: memory allocations, copies, and communication endpoints are concretized according to the chosen mapping. After chosing the top candidates from the design space, the **block-level optimization and code generation** stage compiles the dataflow-aware program that runs on each core into the vendor's existing back-end to generate an executable for each core. This corresponds to the existing compilers that compile tile-based

[1]The source code is available at https://github.com/ecolab-nus/loom-dataflow.

DSLs to block-level program.

TileLoom uses an optional two-step selection strategy to provide better support for various architectures: the automatically generated performance model (using the architectural information) first ranks candidate dataflow plans and selects the top-k mappings statically, and these k candidates are then profiled on the real hardware to choose the final mapping (which bridge the small details not covered by the architecture representation). This combination of model-guided search and hardware validation enables *TileLoom* to produce high-quality mappings while remaining portable across different spatial dataflow systems, especially when micro-architectural details are missing and hence causing inaccuracy in the hardware modeling.

2.2 Spatiotemporal Mapping

Input representation. A unified dataflow-agnostic MLIR representation is required before the dataflow planning pipeline. We assume this representation produced by the front-end, we will cover the implementation details of the Triton front-end in Section 3.1. An example of this MLIR for a matrix–multiplication kernel is shown in Listing 1. The block size and the strides for the input and output matrices are fixed by the front-end. The 2D output space is partitioned over two grid dimensions, `x` and `y`, which scale over the m and n dimensions of an $m \times k \times n$ matrix multiplication (an output-stationary tiling).

Scaling out across tiles is represented by an `affine.parallel` loop over `%block_id_x` and `%block_id_y`. Inside this loop, an `scf.for` loop iterates over the k dimension and accumulates into the same output tile, representing the sequential execution within one block over one output tile. The front-end is required to "affinize" the address arithmetic for memory operations, so every load and store address is an affine function of the tile indices and intra-tile indices. The tile-wise computation itself is expressed with `linalg` operations; this portion of the program is left unchanged during dataflow planning and is later lowered by the back-end.

The purpose of spatial–temporal mapping is to decide how the iteration space of the `affine.parallel` loop—the logical multidimensional tile grid—is assigned to physical cores and to time. To preserve locality, *TileLoom* uses *tiling-based* mappings: contiguous regions of the iteration space are mapped to contiguous spatial regions of the core array or to contiguous temporal regions in the execution schedule.

Schedule representation. On a 2D-mesh architecture like the one in Figure 1, spatial–temporal mapping produces the loop structure shown in Listing 2. The outermost `affine.parallel` loop now iterates over hardware spatial dimensions x and y, each of size 8. These indices correspond directly to the cores in an 8×8 2D mesh. After mapping, this loop represents code that actually runs in parallel across cores; its semantics have changed from a logical grid of parallelizable work-items to the physical parallel core indices.

The next `affine.for` loops iterate over `%tx` and `%ty`. These loops enumerate *waves* of tiles assigned to the same hardware array, i.e., temporal dimensions across blocks. Each wave assigns a batch of logical tiles to the available spatial cores, and the order of these waves determines the temporal schedule with which tiles traverse the array. The innermost `scf.for` remains a purely sequential loop within each core's program.[2]

Design space. The mapping from the original parallel dimensions to spatial and temporal dimensions defines the dataflow of the kernel. Under our tiling-based scheme, the design space is characterized by three coupled choices.

First, each original parallel dimension can be mapped to zero or more spatial dimensions. Mapping a parallel dimension to a spatial dimension corresponds to tiling the loop by the size of that spatial dimension and introducing a new outer `affine.parallel` loop over the hardware index.

Second, when a parallel dimension is tiled by multiple spatial dimensions (for example, by both x and y), the order in which tiling is applied matters. Different tiling orders induce different spatial layouts of tiles on the mesh and different execution schedules, and therefore expose different opportunities for spatial reuse and different communication costs.

Third, once all available spatial dimensions have been used, the remaining parallel dimensions become temporal dimensions implemented as loops over waves, (i.e. changing the rest of `affine.parallel` to `affine.for`), for instance the `%tx`↩ and `%ty` loops in Listing 2. These temporal loops and their order determine how tiles are batched onto the array over time and in what order they revisit each region of the global iteration space, which in turn affects temporal reuse and the shape of NoC traffic.

TileLoom enumerates candidate spatial–temporal mappings by exploring combinations of these choices. Each mapping fixes a concrete loop nest structure, which the subsequent analyses use to reason about data placement, reuse, and communication.

2.3 Data Reuse and Memory Operations

Different spatial–temporal mappings expose different opportunities to reuse data across time and across cores. *TileLoom* first analyzes these opportunities and then decides how to allocate data to memories and when and where to issue copies as broadcasts over the NoC or as loading from global memories.

Reuse analysis on affine accesses. For a fixed spatial–temporal mapping (Section 2.2), the loop nest contains: spatial loops (`affine.parallel`) over hardware core indices; temporal loops (`affine.for`) over waves of tiles, and sequential loops (`scf.for`) inside each core, as shown in Listing 2.

[2]This sequential loop is also temporal, but we use "sequential" to distinguish intra-core sequencing from inter-tile temporal dimensions.

Listing 1: MLIR representation before dataflow planning.

```
func.func @matmul(%A: memref<*xf32>, %B: memref<*xf32>, %C: memref<*xf32>, %grid_dim_x: index, %grid_dim_y: index, %grid_dim_z: index) {
    affine.parallel (%block_id_x, %block_id_y) = (0, 0) to (%grid_dim_x, %grid_dim_y) {
        %cst = arith.constant 0.000000e+00 : f32
        %0 = tensor.empty() : tensor<64x64xf32>
        %1 = linalg.fill ins(%cst : f32) outs(%0 : tensor<64x64xf32>) -> tensor<64x64xf32>
        %2 = scf.for %arg8 = 0 to 8 step 1 iter_args(%arg9 = %1) -> (tensor<64x64xf32>) {
            %4 = affine.apply affine_map<(d0, d1) -> (d1 * 32768 + d0 * 64)>(%arg8, %block_id_x)
            %reinterpret_cast_0 = memref.reinterpret_cast %arg0 to offset: [%4], sizes: [64, 64], strides: [512, 1] : memref<*xf32> to memref<64x64xf32, strided<[512,↩
        1], offset: ?>>
            %alloc = memref.alloc() : memref<64x64xf32>
            memref.copy %reinterpret_cast_0, %alloc : memref<64x64xf32, strided<[512, 1], offset: ?>> to memref<64x64xf32>
            %5 = bufferization.to_tensor %alloc restrict writable : memref<64x64xf32> to tensor<64x64xf32>
            %6 = affine.apply affine_map<(d0, d1) -> (d1 * 32768 + d0 * 64)>(%block_id_y, %arg8)
            %reinterpret_cast_1 = memref.reinterpret_cast %arg1 to offset: [%6], sizes: [64, 64], strides: [512, 1] : memref<*xf32> to memref<64x64xf32, strided<[512,↩
        1], offset: ?>>
            %alloc_2 = memref.alloc() : memref<64x64xf32>
            memref.copy %reinterpret_cast_1, %alloc_2 : memref<64x64xf32, strided<[512, 1], offset: ?>> to memref<64x64xf32>
            %7 = bufferization.to_tensor %alloc_2 restrict writable : memref<64x64xf32> to tensor<64x64xf32>
            // Tile-wise computation, omitted
            linalg.xxx;
            linalg.yyy;
            scf.yield %9 : tensor<64x64xf32>
        }
    }
}
```

Listing 2: Loop structure after spatial–temporal mapping.

```
affine.parallel (%x, %y) = (0, 0) to (8, 8) {
  affine.for %tx = 0 to %grid_dim_x ceildiv 8 {
    affine.for %ty = 0 to %grid_dim_y ceildiv 8 {
      scf.for %k {
        // tile-wise computation
      }
    }
  }
}
```

The front-end expresses all memory accesses as affine functions of these loop indices. For each access, *TileLoom* inspects which induction variables appear in its affine expression. If an access does not depend on a spatial index such as `%x`, then the accessed tile is identical for all cores along that dimension and is spatially reusable there. If an access does not depend on a temporal loop variable such as `%tx`, then the same tile is used across all iterations of that temporal loop and is temporally reusable there. If the access depends only on sequential indices, then reuse is purely intra-core. *TileLoom* records this information as reuse annotations on the memory operations.

Spatial reuse and broadcasts. We begin from a conservative baseline in which every core loads its tiles directly from global memory (for example, an L2 cache or DRAM) in the innermost loop, with no explicit sharing across cores.

If a load has no spatial reuse then the tile is unique to each core, so the load must remain a per-core global memory operation. If a load is spatially reusable along one or more spatial dimensions, *TileLoom* can reduce global traffic by replacing many per-core loads with a smaller number of global loads, followed by broadcasts over the NoC.

In the simplest case, if a tile is reusable only along a single spatial dimension, a designated producer core (or a small group of producers) loads it once from global memory and forwards it along that dimension (for example, along each row of the mesh), while receiving cores buffer their local copies. When a tile is reusable along multiple spatial dimensions, there are several concrete ways to realize that reuse. One option is to first duplicate the tile across all rows (or columns) and then perform independent one-dimensional broadcasts along the parallel dimensions; another is to propagate the tile in a wavefront-style pattern that sweeps across the array. These choices expose different tradeoffs between NoC traffic, latency, and local buffer usage. Listin 3 shows one example candidate of the matrix multiplication mapped to the 2D-mesh example architecture (Figure 1 with a 2D dataflow where the A tiles are broadcasted for each row of cores through the horizontal links of the NoC and the B tiles for each column of cores through the vertical links (we will present the notion of network resources later in Section 2.4). The broadcast information is associated to the load instructions as annotations.

TileLoom does not fix a single strategy: it uses the network description from the hardware representation to enumerate the broadcast patterns that are legal, for each spatially reusable load, a small set of candidate implementations ranging from direct per-core global loads to one-dimensional and multi-dimensional broadcasts. *TileLoom* enumerate all the possibile combinations of all memory operations that creates a design space. Their different hardware costs will be taken into account later by the performance model to select the best ones.

Listing 3: Spatial Reuse

```
affine.parallel (x,y) = (0,0) to (8,8) {
  affine.for tm = 0 to M_waves {
    affine.for tn = 0 to N_waves {
      scf.for tk = 0 to K_tiles {
        load A[tm*8+x, tk]  {type="broadcast", resource={%noc_h}}
        load B[tk, tn*8+y]  {type="broadcast", resource={%noc_v}}
        // ...
      }
    }
  }
}
```

Temporal reuse and loop hoisting. Temporal reuse is realized by choosing the loop level at which a load (or broadcast) is issued. At this point, the loop order is fixed by the spatial–temporal mapping; but we can hoist loads outward so that the same tile is reused across more iterations, at the cost of retaining it longer in a local buffer.

Consider the simplified GEMM-like loop nest below in Listing 4 left, treated as one candidate loop order, `tm` → `tn` → `tk`: the access `A[tm, tk]` depends on `tm` and `tk`, but not on `tn`, so tiles of `A` are temporally reusable across the `tn`↩ loop. If we hoist the loads of `A` outside the `tk` loop, we must buffer all tiles `A[tm, *]` for the current `tm`. Because the address depends on `tk`, hoisting across `tk` enlarges the buffered region from a single tile `A[tm, tk]` to the entire strip `A[tm, 0..K_tiles-1]`. Hoisting further outward, above `tn`, keeps the same buffered strip but reuses it across all values of `tn`, as shown in Listing 4 right:

```
affine.for tm = 0 to M_tiles {
  affine.for tn = 0 to N_tiles {
    scf.for tk = 0 to K_tiles {
      load A[tm, tk]
      load B[tk, tn]
      // ...
    }
  }
}
```

```
affine.for tm = 0 to M_tiles {
  load A[tm, *]
  affine.for tn = 0 to N_tiles {
    scf.for tk = 0 to K_tiles {
      load B[tk, tn]
      // ...
    }
  }
}
```

Listing 4: Loop structures before and after hoisting.

In general, hoisting a load across a loop that the access does not depend on increases reuse without increasing the size of the buffered region, because the accessed tile is the same for all iterations of that loop. Hoisting across a loop that the access does depend on expands the buffered region in proportion to the extent of that loop, because more distinct tiles must be kept live simultaneously. *TileLoom* applies these rules to enumerate, for each load or broadcast, all legal hoisting levels. For each level it computes the required buffer footprint and discards options whose footprint exceeds the capacity of the hardware model.

Temporal vs. Spatial. Temporal reuse and spatial reuse are orthogonal. A tile can be reused only temporally (loaded once per core and reused across iterations), only spatially (broadcast once and immediately consumed), or in both ways (broadcast once and then reused across several temporal iterations). In all cases, the decision can be viewed as picking when a tile is first loaded or received and how long it remains live in local storage under a fixed loop order.

Combining these choices for all loads yields a concrete allocation and copy mapping: a description of which memory each tile resides in at each point in time and which NoC transfers occur. This schedule can be represented by a loop sctructure with annotations as shown a simplified example in Listing 5. Each memory load is annotated with the target buffer, the type of load (using NoC for broadcasting or simple global load), and the NoC resources required. *TileLoom* prunes mappings that violate memory-capacity constraints and passes the remaining candidates to the performance model, which evaluates their compute, memory, and network costs and selects the top-k mappings for back-end profiling.

Listing 5: Example dataflow-friendly MLIR snippet before selection for matrix multiplication kernel on the example 2D-mesh architecture.

```
affine.for tm = 0 to M_tiles {
  alloc A {target_buffer=%L1, size=K_tiles*block_M*block_K}
  load A[tm, *] {type="global", resources={%noc_h, %noc_v}}
  affine.for tn = 0 to N_tiles {
    alloc C {target_buffer=%L1, size=block_M*block_N}
    scf.for tk = 0 to K_tiles {
      alloc B {target_buffer=%L1, size=block_K*block_N}
      load B[tk, tn] {type="broadcast", resources={%noc_h, %noc_v}}
      // tile-wise computations
      load C
      linalg.matmul ...
      linalg.exp ...
      linalg.sqrt ...
    }
    store C {type="global". resources={%noc_h, %noc_v}}
  }
}
```

2.4 Hardware Representation

TileLoom is designed to target multiple dataflow architectures. To make mapping decisions, the compiler needs a structured description of the hardware: how cores are arranged in space, where memories are placed, how components are connected, and what compute resources are available at each location. *TileLoom* captures this information in a multi-layer hardware representation stack. Different layers of this stack are consumed by different stages of the compiler passes.

We encode this representation in a custom MLIR dialect, `df`. The dialect provides operators that describe the scale-out structure of the machine (cores and interconnects), the memory hierarchy and its connectivity, and the intra-core compute units. The performance model and the mapping passes process the program and the hardware description written this dialect together, rather than hard-coding any particular architecture.

Scale-out architecture. At the top level, the `df` dialect describes the spatial layout of cores and the on-chip interconnect. The following operators are needed:

`df.spatial_dim(size)` declares an abstract spatial dimension, used to index and replicate hardware components. Spatial dimensions naturally represent arrays of parallel resources such as cores or memories.

`df.core(scaleout, scalein)` declares a set of cores indexed by the dimensions listed in `scaleout`; the `scalein` of the operation contains the compute components that live inside each core (optional argument, used at a lower abstraction level, described later).

`df.interconnects(components, map, bandwidth)` declares a network (set of links) that connects a set of components according to an affine map `map` and `bandwidth` specifies the bandwidth per link.

These operators are used to describe the scale-out architecture used in the spatial–temporal mapping. For our example 2D-mesh architecture (Figure 1), we can describe it with:

This fragment describes an 8×8 array of cores connected by horizontal and vertical rings (or a torus). The modulo in the affine maps encodes wrap-around links. The spatial–temporal

Listing 6: 2D mesh with abstract scale-out.

```
%x = df.spatial_dim 8
%y = df.spatial_dim 8
%cores = df.core { scaleout = (%x, %y) }
%noc_h = df.interconnects %cores, %cores { map = affine_map<(d0, d1) -> ((d0 + 1) mod 8, d1)>,bandwidth = 28 }
%noc_v = df.interconnects %cores, %cores { map = affine_map<(d0, d1) -> (d0, (d1 + 1) mod 8)>, bandwidth = 28 }
```

mapping pass uses this scale-out description to map the logical tile grid onto physical cores (Section 2.2), and the performance model uses the `df.interconnects` operators to estimate communication cost and traffic congestion of the memory operations (Section 2.5).

Memories and data movement. When planning data movements (Section 2.3), *TileLoom* must reason about concrete buffers and how the network feeds data into them. To do so, the scale-out description is refined with explicit memories.

`df.memory(scaleout, size, bandwidth)` declares a set of memories indexed by the `scaleout` dimensions. Each instance has the given capacity and per-port bandwidth.

`df.mux(dst, bandwidth, srcs, map)` declares a 1-to-N connectivity between `dst` components and `srcs`, with topology specified by an affine `map`. This operator captures fan-out connections such as "each core can access its local scratchpad" or "groups of cores share a DRAM channel".

A lowered version of the 2D-mesh description that includes L1 memories and DRAM is shown in Listing 7:

Listing 7: 2D mesh with scale-out cores, scratchpads, and DRAM.

```

%x = df.spatial_dim 8
%y = df.spatial_dim 8
%cores = df.core {scaleout=(%x, %y)}
// Per-core scratchpad memories.
%L1 = df.memory {scaleout=(%x, %y), size = 1499136, bandwidth = 60}
// Connect each core (x, y) to its local L1(x, y).
%core_to_L1 = df.mux %cores, %L1, {map = affine_map<(d0, d1) -> (d0, d1)>}
// On-chip NoC now connects L1 memories.
%noc_h = df.interconnects %L1, %L1, {map = affine_map<(d0, d1) -> ((d0 + 1) mod 8, d1)>, bandwidth = 28}
%noc_v = df.interconnects %L1, %L1, {map = affine_map<(d0, d1) -> (d0, (d1 + 1) mod 8)>, bandwidth = 28}
// Off-chip DRAM channels, indexed by a 1D spatial dimension.
%dram_idx = df.spatial_dim 4
%drams = df.memory {scaleout = %dram_idx, size = 12884901888, bandwidth = 267}
// Map each group of 4x4 edge cores to a DRAM channel.
%to_dram = df.interconnects %L1, %drams {map = affine_map<(d0, d1) -> (d0 ceildiv 4 + 2 * (d1 ceildiv 4))>, bandwidth = 30}
```

This representation now distinguishes the physical buffers that can hold tiles. For example, it specifies that each core `(x, y)` has an L1 scratchpad of around 1.5 MB, and that inter-core traffic flows between L1s rather than directly between cores. It also encodes DRAM channels are connected: every group of four adjacent cores along each edge shares the same DRAM bank, as in Figure 1. When *TileLoom* chooses where to buffer a tile and the costs of memory operations, it uses these `df.memory`, `df.mux`, and `df.interconnects` operators. As shown earlier in Listing 5, load instructions are annotated with bindings to these physical resources.

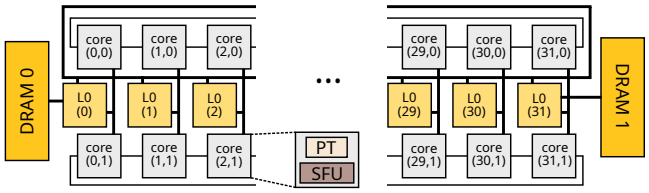

Figure 3: Example 1D triple-ring architecture modeled with the `df` dialect.

Intra-core compute model. To drive the performance model down to the level of individual cores, *TileLoom* needs a coarse description of the microarchitecture within each core. The `df` dialect provides operators for this purpose:

`df.mat(shape, throughput)` declares a matrix unit (for example, a tensor core) with a given input shape and sustained throughput.

`df.vec(shape, throughput)` declares a vector unit with the given vector width and throughput.

`df.scalar(latency)` declares a scalar unit with a given latency for scalar operations.

Each unit is assumed to accept operands of the specified shape and to produce results at the given throughput. These units are then attached to cores via the `scalein` argument of `df.core`, which describes the internal composition of each core. The lowest-level version of the 2D-mesh description therefore extends the previous listing with intra-core units:

Listing 8: Extra specifications of the intra-core architecture

```
%FPU = df.mat {shape=[32, 32, 32], throughput=98}
%SFPU = df.vec  {shape=[32], throughput=3}
%cores = df.core {scaleout = (%x, %y), scalein=(%FPU, %SFPU, [8, 1])}
```

With this information, the performance model can reason about the timing of both compute and memory. It can, for example, estimate how many cycles a particular tile-level matmul consumes on the matrix units and whether the NoC bandwidth and the L1 buffer are sufficient to keep them fed.

Expressiveness beyond 2D meshes. Although we have used the Tenstorrent-2D-mesh architecture as example, `df` can describe other spatial dataflow architectures. For instance, a 1D triple-ring topology similar to the IBM-Spyre accelerator as shown in Figure 3 can be described using `df` program as shown in Listing 9 .

Discussion We structure the hardware representation as a stack of layers of abstractions because each compiler pass should depend only on the level of detail it actually needs. Spatial–temporal mapping requires only the scale-out structure of cores and the topology and bandwidth of the interconnect. Data-movement planning additionally needs to know where memories are placed and how they are wired to compute and to DRAM. The fine-grain performance model, in

Listing 9: Example of the df dialect describing a 1D triple-ring architecture.

```
module {
    // functional units
    %PT = df.mat {shape = [128, 128, 128], throughput=16384}
    %SFP = df.vec {shape = [128], throughput=128}
    // scale-out
    %x = df.spatial_dim 32
    %y = df.spatial_dim 2
    %cores = df.core "cores" {scaleout=(%x, %y) , scalein=(%PT, %SFP, [1,1])}
    %L1 = df.memory {scaleout=(%x) , size = 2097152, bandwidth = 128}
    %core_to_mem = df.mux %cores, %L1, {map = affine_map<(d0, d1) -> (d0)>}
    %small_rings = df.interconnects %cores, %cores, {map = affine_map<(d0, d1)
        -> ((d0 + 1) mod 8, d1)>, bandwidth = 32}
    %big_ring = df.interconnects %L1, %L1, {map = affine_map<(d0) -> ((d0 + 1)
        mod 8)>, bandwidth = 258}
    // Global (DRAM/L2)
    %d = df.spatial_dim 2
    %dram = df.memory {scaleout=(%d) , size = 34359738368, bandwidth = 512}
    %to_dram = df.interconnects %dram, %L1, {map = affine_map<(d0)->(d0*31)>}
}
```

turn, needs a high-level view of intra-core compute units and their throughputs. This separation improves the reusability of the compiler. Changing the on-chip network, the memory hierarchy, or the per-core microarchitecture amounts to modifying the `df` description, without rewriting the optimization passes and the rest of hardware description. It also creates a bridge from software-level mapping decisions to hardware-level design trade-offs. Starting from the lowest abstraction level, the same representation can be refined further to include implementation-specific costs such as area and power, enabling combined design space exploration over both mappings and hardware configurations—an important capability for spatial dataflow architectures, where the architecture itself varies widely across generations and vendors.

2.5 Performance Modeling

After spatial–temporal mapping and data reuse / allocation decisions, *TileLoom* has a set of candidate dataflow schedules. Each candidate is represented as an MLIR program in which loop nests, memory operations, and data movements (global loads, broadcasts, buffered loads) are fully specified, and every memory or network operation is bound to concrete hardware resources described in the `df` dialect (Section 2.4). A simplified example is shown in Listing 5. The role of the performance model is to estimate the execution time of each candidate, using the compute units, memories, and interconnects defined in the `df` description, and then select the top-k candidates for downstream code generation and profiling.

Figure 4 illustrates how the performance model evaluates the overall execution time of the example in Listing 5. It evaluates from the innermost loop outward, aggregating compute, memory, and network costs hierarchically.

Compute cost per loop body. We first estimate the execution time of the innermost loop body, treating it as a block-level program running on a single core. For a given tile shape, every high-level operator (for example, a `linalg.matmul`) is decomposed into the core's low-level compute intrinsics. The available matrix, vector, and scalar units and their throughputs come from the `df.mat`, `df.vec`, and `df.scalar` units attached to that core via `df.core` (Section 2.4).

For each operator, *TileLoom* uses its `linalg` semantics to recover the parallel iteration space. This tells us, for each functional-unit type, how many intrinsic invocations of that type are independent and can, in principle, be issued in parallel. We then conceptually schedule these intrinsics onto the available parallel units of the same type: if there are N independent instances mapped to a unit type with U identical units, each capable of issuing r intrinsics per cycle, we approximate the operator's time contribution on that unit type as $N/(U \cdot r)$ cycles.

TileLoom then accounts for data dependencies and resource sharing among different unit types. Operators that are independent and target different unit types (for example, a matrix multiply on a matrix unit and a pointwise activation on a vector unit) can execute in parallel, whereas dependent operators or operators that compete for the same unit type must execute in sequence. The loop-body compute time is approximated as the sum over sequential segments, where each segment's time is the maximum over all operators that can run in parallel within that segment.

The potential parallelism exposed by this model is not necessarily fully achievable on a concrete microarchitecture, but the model does not attempt to exactly simulate the core's instruction scheduler. Instead, it is calibrated to be accurate enough to distinguish compute-bound from memory-bound mappings and to reason about overlap between compute and data movement. In our experiments, this coarse-grain modeling of compute is sufficient to discriminate between different dataflow schedules.

Compute–memory Overlap. Once we have an estimate for the loop body, we incorporate data movement. Let T_{load} be the time spent on all loads in one iteration of the loop body, T_{compute} the compute time (from the previous step), and T_{store} the time spent on all stores. We assume that each iteration executes as a pipelined load–compute–store sequence with double buffering: while iteration i is computing, the stores for iteration $i-1$ and the loads for iteration $i+1$ proceed in parallel whenever possible.

For an innermost loop with I iterations, the total execution time is approximated as:

$$T_{\text{loop}} \approx (I-2)\cdot \max\big(T_{\text{load}}+T_{\text{store}},\, T_{\text{compute}}\big) + \max\big(T_{\text{load}},\, T_{\text{compute}}\big) + \max\big(T_{\text{store}},\, T_{\text{compute}}\big) + T_{\text{load}} + T_{\text{store}}.$$

The first term accounts for the $I-2$ steady-state iterations, where load, compute, and store can overlap and the throughput is limited by the slower of T_{compute} and $T_{\text{load}}+T_{\text{store}}$. The remaining terms account for filling and draining the pipeline. This behavior is illustrated in Figure 4, where the k-body is executed in parallel with loading the next tile of B and storing the result tile of C from the previous iteration.

Concurrent data transfers and network traffic. Several

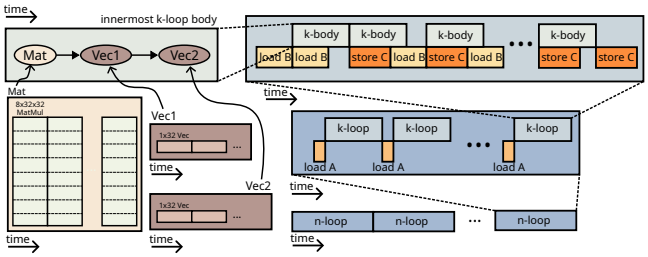

Figure 4: Example Evaluation of Pipelined execution.

memory operations may occur simultaneously and create traffic contention over the NoC. *TileLoom* estimates the effective bandwidth of each memory operation under this contention, using both the interconnect structure described in the hardware representation and the mapping annotations attached during memory operation mapping (Section 2.3). At that mapping step, each load or store is lowered either to a global load/-store or to a broadcast pattern, and the compiler records which subsets of the network it uses. For global loads, we assume accesses are sufficiently random that traffic is spread across the NoC links. For broadcasts, the resources used depend on the chosen pattern: on the 2D-mesh example, a broadcast that is performed independently along each row uses only the horizontal ring links (such as `%noc_h`), whereas a broadcast over the entire mesh may exercise both horizontal and vertical rings (both `%noc_h` and `%noc_v`). These choices are fixed during memory operation mapping and appear as annotations, as in Listings 3 and 5. Given these annotations, the performance model groups memory operations according to the network links and memory interfaces they occupy. For each group of operations that share a particular subset of links, it aggregates their offered traffic and derives an effective bandwidth per transfer by partitioning the nominal link bandwidth among them. Equivalently, the bandwidth seen by any one operation is reduced in proportion to the number of concurrent transfers using the same links or banks. The transfer time for each load or store is then computed from its tile size and effective bandwidth. The per-iteration load and store times, T_{load} and T_{store}, are obtained by combining these transfer times across operations, treating transfers on disjoint link sets as running in parallel and transfers on overlapping link sets as time-sharing the same resources. These T_{load} and T_{store} values are then plugged into the pipelined overlap model described above.

Candidate ranking through auto-profiling. *TileLoom*'s optimizations are highly dependent on the accuracy of the architectural models. When the architectural and micro-architecral information is not accurate enough, *TileLoom* supports a profiling-based autotuning step to maintain its performance. Concretely, for each candidate schedule, *TileLoom* combines the compute and data-movement estimates with the architectural model to obtain an approximate end-to-end execution time. This time reflects the balance between compute and communication, the benefit of spatial and temporal reuse, and the impact of NoC and memory contention on the concrete hardware described by `df`. *TileLoom* then ranks all candidates by this estimated time and keeps only the top-k dataflow mappings. Only these top-k candidates are handed to the back-end for full code generation and on-hardware profiling, where the final best-performing configuration (top-1) is selected. The choice of k controls the trade-off between compile time and the likelihood of including the true optimum: a larger k explores more mappings but costs more compilation and profiling time. We study this trade-off in Section 3.3.4.

3 Evaluation

3.1 Experimental Setup

Hardware platform. We conduct our experiments on Tenstorrent Wormhole and Blackhole cards. Table 1 summarizes the relevant hardware specifications. We compile *TileLoom* programs on a host machine equipped with dual 16-core Intel Xeon Gold 6326 CPUs and 512 GB of DRAM.

Table 1: Specifications of the Tenstorrent Wormhole and Blackhole used in our evaluation.

	TT-Wormhole	TT-Blackhole
Topology	8×8	12×10
On-chip SRAM	108 MB	180 MB
DRAM	12 GB GDDR6	32 GB GDDR6
Off-chip bw.	288 GB/s	512 GB/s
TFLOPS (FP16)	64	162
TBP	160 W	300 W
Technology Node	GF 12nm [52]	TSMC 6nm [53]

Architecture targets and modeling. Each Wormhole and Blackhole chip contains a 2D array of cores, with 8×8 cores on Wormhole and 12×10 cores on Blackhole, as shown in Figure 1. We model the Tenstorrent architecture in the `df` dialect, as described in Figure 1 and Section 2.4. Because the complete proprietary hardware specification is unavailable, we recover key parameters through isolated microbenchmarks, including matrix/vector-unit throughput and effective NoC and DRAM bandwidths. We instantiate these measured values in the `df` hardware representation used by *TileLoom*'s performance model.

Frontend. *TileLoom* currently supports Triton and Helion as kernel-development frontends. Although the core compiler is frontend-agnostic, our experiments use either Triton or Helion depending on kernel availability; the source code for all input kernels is available in our GitHub repository. For both frontends, we tune the tile, or block, shape using their existing Python-based autotuning stacks. For Triton, we use *triton-shared* [44] to lower kernels into MLIR, then apply a custom affinization pass that rewrites index arithmetic into affine expressions, followed by normalization into our dataflow-

agnostic MLIR format, as shown in Figure 2. For Helion, after Python-level tuning, we lower Helion Device IR into the same standard MLIR format using our custom lowering tool.

Backend. We lower our dataflow-aware MLIR to TT-Metalium, Tenstorrent's low-level C API, to generate the final executable, as shown in Figure 2. TT-Metalium exposes coarse-grained primitives for computation, synchronization, buffer allocation, and data movement, and handles most block- and core-level optimizations. *TileLoom* bridges the dataflow-level program and TT-Metalium by performing lifetime analysis over the block-level compute graph, using the resulting lifetimes to determine buffer allocation and synchronization among memory, compute, and data-movement operations. TT-Metalium then lowers these coarse-grained operations to hardware instructions.

3.2 End-to-End Performance

Table 2 reports the geometric mean of the relative performance of *TileLoom* over TTNN on four representative kernels: GEMM, FlashAttention, Flash Decode, and Mamba Chunk Scan linear attention. For each kernel, we evaluate a range of input shapes. The detailed shape-by-shape results are discussed in the corresponding subsections below. In this section, we report the performance of the top-ranked candidate selected by *TileLoom*'s performance model, without the final profiling-based tuning step. Therefore, the reported speedups come solely from *TileLoom*'s architectural modeling and dataflow-planning strategy.

Table 2: Relative performance of *TileLoom* over TTNN on GEMM, FlashAttention, Flash Decode, and Mamba Chunk Scan linear attention.

TileLoom vs. TTNN	Wormhole	Blackhole
vs. library implementation		
FlashAttention	1.94x	1.98x
Flash Decode	0.84x	0.87x
GEMM	0.95x	1.10x
vs. unfused implementation		
Mamba Chunk Scan	27.23x	16.27x

3.2.1 FlashAttention

We evaluate *TileLoom* on non-causal FlashAttention against the native TTNN implementation. The non-causal variant exposes more dataflow-optimization opportunities than the causal variant, making it a useful stress test for *TileLoom*'s spatial mapping strategy. We vary the number of attention heads between 64 and 128, sweep sequence length from 1024 to 16384, and adjust batch size to fit within DRAM capacity. As shown in Figure 5, *TileLoom* consistently achieves substantial speedups over TTNN, with 1.88–2.06× improvement in nearly every configuration. The gains come from exploiting reuse in the attention operands: *TileLoom* places tiles so that key tiles are reused on chip across multiple query and value tiles, reducing DRAM traffic compared with TTNN's default mapping, which repeatedly reloads these operands from DRAM.

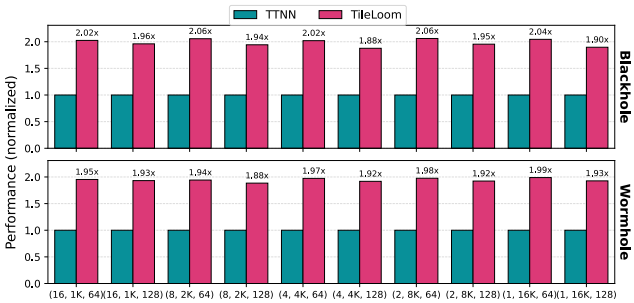

Figure 5: Relative Performance with *TileLoom* vs. TTNN on FlashAttention with parameters (batch size B, sequence length L, number of heads N).

3.2.2 FlashDecode

Flash Decode is algorithmically a special case of FlashAttention with query length one, but this changes the hardware mapping problem significantly. In full FlashAttention, *TileLoom* can exploit parallelism across query blocks and heads, while choosing mappings that improve key/value tile reuse. In Flash Decode, the query dimension no longer offers spatial parallelism, so parallelism mainly comes from the batch dimension and from splitting the key/value reduction across cores.

This yields a much smaller dataflow-planning space: there are fewer ways to distribute independent output tiles, and the main optimization is partitioning the key/value sequence and orchestrating the cross-core gather-reduce. Thus, this benchmark stresses *TileLoom*'s generated block-level code and reduction orchestration more than its ability to choose among diverse dataflow mappings.

As shown in Figure 6, *TileLoom* does not outperform the TTNN Flash Decode baseline, which is expected because TTNN uses scheduling and reduction optimizations specifically tuned for this operator. In contrast, *TileLoom* generates its implementation through a general compiler stack starting from tile-level kernel code. Even so, *TileLoom* achieves about 85% of TTNN performance on average, showing that it can produce competitive code even with limited dataflow optimization opportunity and a highly specialized vendor baseline.

3.2.3 Mamba Chunk Scan

Mamba Chunk Scan is a linear-attention kernel with substantial intra-block computation and nontrivial data movement.

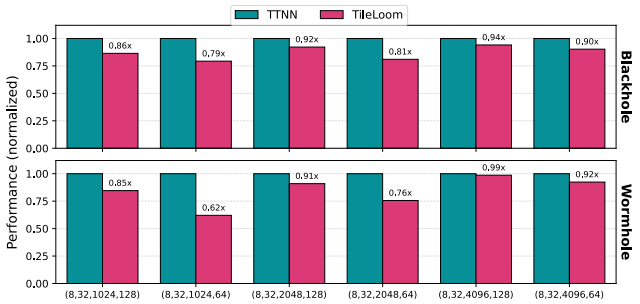

Figure 6: Relative performance of *TileLoom* compared with TTNN on Flash Decode with parameters (batch size B, head dimension H, sequence length L, hidden dimension D).

Since TTNN does not provide a fused implementation, we build the TTNN baseline by composing existing TTNN operations, yielding an unfused implementation. In contrast, *TileLoom* uses a fused tile-level Helion kernel.

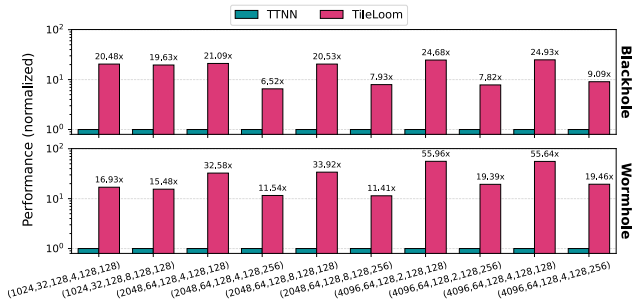

Figure 7: Relative performance of *TileLoom* vs. an unfused TTNN implementation on Mamba Chunk Scan with parameters sequence length L, number of heads N, head dimension H, number of groups G, hidden dimension D, and chunk size C.

This comparison shows a practical advantage of supporting tile-level kernel languages: fused kernels are often easier to obtain than highly optimized vendor-library implementations. Thus, *TileLoom* can generate efficient fused code even when the vendor library only exposes lower-level building blocks. As shown in Figure 7, *TileLoom* achieves 10x–55x speedup over the unfused TTNN baseline, reflecting both kernel fusion and *TileLoom*'s dataflow planning.

This large gap is typical between fused and unfused kernels, and similar trends appear in Tenstorrent's official technical reports. It is especially pronounced on dataflow architectures, which rely on on-chip NoC bandwidth for reuse and often have less off-chip bandwidth than HBM-based GPUs. Fusion and dataflow planning are therefore critical: without them, intermediate tensors are repeatedly written to and read from off-chip memory, causing severe slowdowns.

3.2.4 GEMM

We evaluate GEMM on over M, K, and N, ranging from 256 to 16384. GEMM is a key primitive, and the TTNN vendor implementation is already highly optimized, achieving about 70% of peak hardware throughput on average. Against this strong baseline, Table 2 shows that *TileLoom* delivers comparable performance using tile-level kernels while automatically generating low-level implementations.

For a more interpretable comparison, Figure 8 also reports two TTNN dataflow templates, TT-1D and TT-2D. TT-1D uses a 1D broadcast pattern: each core loads the smaller input matrix from global memory, while the other input is broadcast across the entire array. TT-2D uses a 2D broadcast pattern, streaming the two inputs across the mesh from the top and left. TTNN selects between these templates using a shape-dependent heuristic, with block size chosen by a separate strategy.

Figure 8 shows that *TileLoom* remains competitive with TTNN across the full GEMM sweep. This is notable because TTNN uses manually optimized library kernels, whereas *TileLoom* starts from tile-level kernels and searches the mapping space automatically. The TT-1D and TT-2D results show that fixed dataflows work well for regular GEMM shapes, where M, K, and N are similar, but can degrade on irregular shapes where the best mapping depends more sensitively on dimensions and hardware balance.

3.2.5 Shape Sensitivity on Irregular GEMM

To study shape sensitivity, we evaluate two irregular GEMM families. First, we fix $M = N = 32768$ and vary K from 256 to 2048. Second, we fix $M = K = 32768$ and vary N over the same range. Figure 9 shows the results.

When varying K (Figure 9a), *TileLoom* and TTNN follow trends similar to the 1D and 2D baselines. This is expected because K is mapped sequentially within each core, so changing it mainly affects intra-core compute cost and leaves limited room for inter-core dataflow optimization.

In contrast, varying N (Figure 9b) significantly changes the preferred dataflow. As N approaches M, the workload becomes more balanced across rows and columns, making 2D-like broadcast more attractive due to reuse along both mesh dimensions. When N is much smaller than M, 1D-like strategies are more effective, as reflected by the TT-1D and TT-2D results.

The transition between 1D- and 2D-favorable regimes depends on block size, compute-to-memory ratio, NoC overhead, and other hardware factors. This sensitivity motivates *TileLoom*'s cost-model-guided search, which accounts for spatial reuse, communication volume, and architectural constraints instead of relying on fixed heuristics.

These irregular-shape results also reveal a limitation of TTNN's heuristic selection: TTNN does not always choose

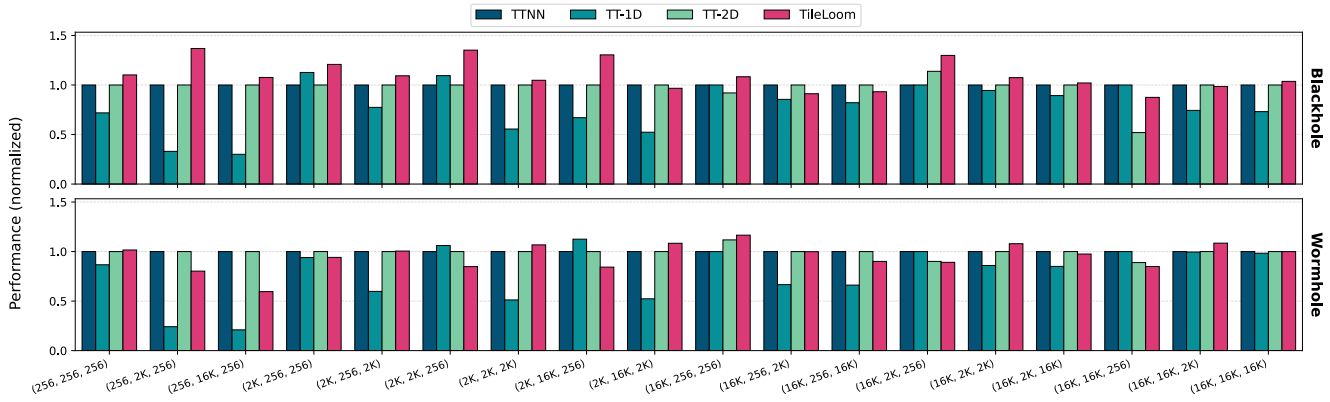

Figure 8: GEMM performance of *TileLoom* compared with TTNN. TT-1D and TT-2D are included as reference dataflow templates.

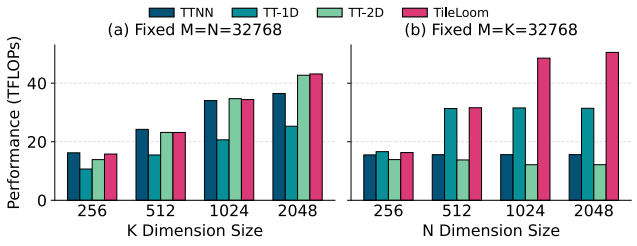

Figure 9: Performance comparison for GEMM under irregular input shapes.

the faster of its two templates. For example, in small-N, large-M cases, TTNN selects TT-2D even though TT-1D is faster, such as $M,K,N = 32K,32K,512/1024$. Our inspection suggests this comes from fixed thresholds over ratios such as M/N. Near these thresholds, the best strategy can also depend on tile size, bandwidth, and NoC cost, but the heuristic still makes a hard ratio-based decision.

In contrast, *TileLoom* searches beyond the two TTNN templates, exploring how logical dimensions map to spatial dimensions, where memory operations are hoisted in the loop nest, and which block sizes to use. This broader search allows *TileLoom* to find mappings that outperform both TTNN templates, as shown in Figure 8.

3.2.6 Wormhole vs. Blackhole

TileLoom generally achieves larger relative gains on Blackhole than on Wormhole. This is because Blackhole increases compute throughput more than off-chip bandwidth: peak FP16 throughput improves by 2.53×, while off-chip bandwidth improves by only 1.78× (Table 1). This shift makes kernels more likely to become memory-bound, increasing the value of *TileLoom*'s dataflow optimizations, which improve spatial reuse and reduce off-chip traffic.

GEMM shows this trend most clearly. On Wormhole, *TileLoom* reaches 0.95× TTNN performance. Profiling shows that GEMM is often compute-bound on Wormhole, so reducing off-chip traffic has limited end-to-end benefit. In addition, *TileLoom* starts from tile-level kernels and does not yet include all microarchitecture-specific optimizations used in TTNN's hand-tuned GEMM library, explaining the remaining 5% gap.

On Blackhole, the same GEMM evaluation improves to 1.10× over TTNN. Blackhole's higher compute-to-bandwidth ratio makes GEMM more sensitive to memory traffic and placement, allowing *TileLoom*'s dataflow planning to provide greater benefit. This shows that while *TileLoom* may not always beat a vendor GEMM library on a compute-limited device, its architectural modeling and spatial-reuse optimizations become more valuable as hardware shifts toward higher compute density.

3.3 Ablation Studies

3.3.1 Effect of Spatial Reuse

We quantify how much of *TileLoom*'s GEMM performance comes from spatial reuse by disabling the spatial-reuse pass and forcing all operands to be loaded from DRAM. Table 3 reports absolute performance with and without this optimization. Spatial reuse gives the largest gains on smaller GEMMs, where DRAM traffic is a major bottleneck. As size grows, the benefit decreases, consistent with the roofline model [69]: GEMM arithmetic intensity increases with matrix size, making larger problems more compute-bound. Once performance is limited by peak compute, reducing DRAM traffic has diminishing runtime impact.[3] Still, spatial reuse substantially reduces memory pressure: across these GEMM configurations, it cuts DRAM accesses by 70% on average. This does not

[3] Across all GEMM measurements, including configurations not shown here, effective throughput stabilizes around 45 TOP/s, indicating that many configurations operate near the compute roof.

always translate to proportional speedup in compute-bound regimes, but can improve memory headroom, reduce power, and leave more bandwidth for concurrent workloads.

Table 3: GEMM performance (TFLOP/s) on Wormhole with (*TileLoom*) and without (DRAM only) spatial reuse.

Configuration	DRAM only	*TileLoom*	Speedup
M=K=N=1024	11.15	23.70	2.12×
M=K=N=2048	16.77	33.96	2.03×
M=K=N=4096	22.96	40.44	1.76×
M=K=N=5120	29.19	41.61	1.42×
M=K=N=6144	28.22	42.47	1.51×

3.3.2 Effect of Temporal Reuse

Figure 10 shows the impact of temporal reuse across GEMM shapes. Temporal reuse buffers tiles locally so the same A or B tiles are reused across iterations instead of repeatedly loaded from DRAM.

We compare *TileLoom* with and without temporal reuse over different M and N values. As with spatial reuse, this optimization is most useful in memory-bound regimes, so we decrease K as M and N grow to keep the configurations memory-bound. In this setting, temporal reuse yields speedups up to 1.12×. The benefit increases with M and N, since larger dimensions create more opportunities to reuse the same A or B tiles. Thus, temporal reuse is most helpful when M or N is large and K is small. When it provides little benefit, *TileLoom*'s performance model deprioritizes those mappings, resulting in the same selected mapping and performance as the baseline without temporal reuse.

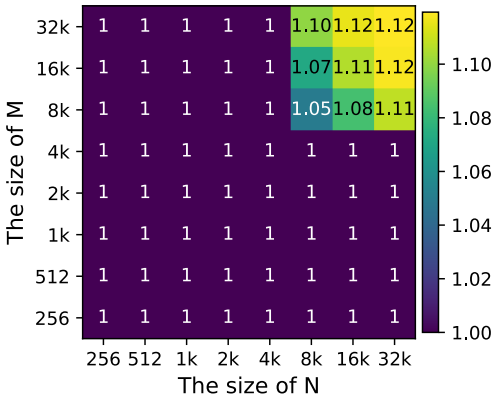

Figure 10: Normalized GEMM performance on Wormhole with and without temporal reuse.

3.3.3 Accuracy of *TileLoom*'s Performance Model Alone

We validate *TileLoom*'s performance model on Wormhole by comparing predicted throughput with measured GEMM performance across a wide range of (M,N,K) configurations. Figure 11 plots both estimates and measurements.

The predictions differ from measurements by 17% in geometric mean. However, the goal is not cycle accuracy, but reliable ranking and trend prediction, especially for transitions between memory- and compute-bound regimes as described by the roofline model [69]. As shown in Figure 11, the model captures these transitions well: it identifies when configurations become compute-bound and reflects the relative performance changes across shapes.

The end-to-end results in Table 2 and 4 further show that this error has limited impact on mapping selection. The model is accurate enough to rank candidates effectively, leaving the final optional profiling stage to choose among a small set of high-quality mappings.

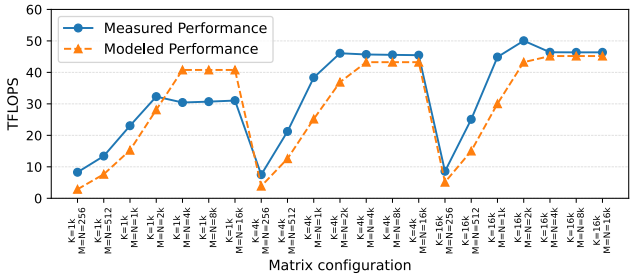

Figure 11: Validation of *TileLoom*'s performance model against measured GEMM performance.

3.3.4 Effect of top-k tuning and different topologies

As discussed in Section 2.5, *TileLoom* can optionally profile the top-k candidates generated by the compiler and select the best one. Thus, k trades compilation cost for final performance. We vary k from 1 to 5 and report geometric-mean normalized performance on Wormhole, along with compile time across hardware topologies, in Table 4. Here, top-1 means fully static compilation with only the best-predicted mapping, while larger k values add more profiled candidates.

We evaluate three topologies to test sensitivity to available parallelism and communication structure. The 8×8 mesh uses the full Wormhole core array, while the 4×8 mesh and 1×8 ring model smaller subsets. These reduced topologies change the balance between compute, NoC communication, and off-chip bandwidth, stressing whether mappings predicted for the full mesh remain effective under more constrained layouts.

Overall, k has a modest effect on performance. On the 8×8 mesh, top-5 outperforms top-1 by 7%, with most of the gain already achieved at top-2, which improves performance by 4.7%. The remaining gap mainly comes from occa-

sional model misrankings, where the predicted-best mapping is slightly worse than another candidate. In practice, a small k, such as 2 or 3, is usually enough to include the best mapping while keeping compile time moderate.

Similar trends hold on smaller topologies. The 1×8 ring is least sensitive to k because its restricted layout leaves fewer strong mappings to distinguish. The 4×8 and 8×8 meshes expose larger search spaces, so profiling a few extra candidates helps more. However, gains saturate quickly: beyond top-3, performance improves only marginally while compile time grows nearly linearly. This suggests that *TileLoom*'s static model effectively narrows the search to a small set of promising mappings, and a small profiling budget recovers most available performance across topology scales.

Table 4: Geometric-mean normalized performance (relative to TTNN) and compile time (seconds) of *TileLoom* under different top-k profiling settings. Top-1 corresponds to fully static compilation without additional profiling.

	1×8 Ring	4×8 Mesh	8×8 Mesh
top-1	-1.3% (6.36)	-5.4% (5.53)	-6.1% (5.75)
top-2	-1.3% (12.70)	-0.8% (11.29)	-1.4% (11.17)
top-3	-1.2% (19.07)	+0.5% (17.16)	-0.5% (16.65)
top-4	-0.4% (25.47)	+1.1% (23.03)	+0.4% (22.17)
top-5	-0.3% (31.82)	+1.3% (28.91)	+0.9% (27.66)

4 Related Works

Hardware modeling and co-design. There exist works on modeling and co-design of spatial accelerators: CGRAs, spatial FPGAs, and systolic arrays, where the computes and control are at a finer granularity than *TileLoom*. Languages and frameworks such as Spatial [31], Plasticine [56], T2S [59], Halide(-to-Hardware) [57], and HeteroCL [35] compile loop nests or functional pipelines into arrays of processing elements (PEs) and local memories, often co-designing the overlay itself. Polyhedral and tensor-centric systems such as AutoSA [65], TensorLib [24], and Rubick [39, 41] similarly start from affine loop nests or tensor expressions and derive space–time mappings that synthesize systolic or tensor arrays, buffers, and controllers. Co-design tools like Timeloop/Accelergy [51] and AMOS [74] explore dataflows, tilings, and memory hierarchies for DNN accelerators, while MAESTRO/-MAERI [32, 33] analytically model buffer and network usage for specialized systolic-style designs. In all of these systems, the modeling unit is a PE, buffer, or loop level in the memory hierarchy, and the goal is to search or synthesize hardware at that granularity. *TileLoom*, in contrast, assumes intra-core microarchitecture and local mapping are fixed, treats each core as the atomic unit, and models only the multi-level memory system and NoC *between* cores to decide how tile instances are distributed across space and time.

Software mapping and compilation for spatial architectures. On the software side, classical CGRA and FPGA flows treat mapping as a place-and-route problem on a PE-level dataflow graph: frameworks such as DSAGEN [67], ML-oriented CGRA compilers like ML-CGRA and MLIR-to-CGRA [40, 72], and architecture-agnostic mappers such as Morpher [68] and CaSMap [42] perform placement, routing, and modulo scheduling onto a fixed fabric. Loop- and polyhedral-based tools including Timeloop/Accelergy [51], and MAESTRO/MAERI [32, 33] reuse loop schedules or space–time mappings as the scheduling representation but are primarily design-space exploration tools: they evaluate mappings analytically. Research compilers for specific spatial architecture, such as AMOS [74], LISA [37], and system-level compilers for wafer-scale fabrics [20, 62], do generate per-core programs and communication schedules, but are typically tailored to a particular architecture family with baked-in mapping heuristics.

5 Conclusion

TileLoom demonstrates that compiler-driven mapping can deliver competitive performance on spatial dataflow accelerators while substantially reducing the need for hand-written, hardware-specific kernels. Starting from high-level tile-centric kernels, TileLoom automatically selects spatial and temporal mappings, data-reuse strategies, and communication patterns using a dataflow dialect that uniformly represents cores, memories, and interconnect.

Across Tenstorrent Wormhole and Blackhole, TileLoom achieves strong results on FlashAttention, Mamba Chunk Scan, GEMM, and Flash Decode. It outperforms TTNN when dataflow planning and fusion expose substantial opportunities, and remains competitive with highly optimized vendor implementations when the optimization space is more constrained. These results show that many decisions traditionally embedded in hand-engineered kernels can instead be handled by a reusable compiler stack, making optimized kernel development more accessible and providing a foundation for future spatial dataflow architectures.

Acknowledgment

This work is partially supported by the Advanced Research and Technology Innovation Centre (ARTIC), the National University of Singapore under Grant AFP-RP6, and by National Research Foundation, Singapore, under its Competitive Research Program Award NRF-CRP23-2019-0003 and the Ministry of Education, Singapore, under Tier 3 grant MOE-MOET32024-0003.

References

[1] Dennis Abts, Garrin Kimmell, Andrew C. Ling, John Kim, et al. A software-defined tensor streaming multiprocessor for large-scale machine learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA 2022)*, pages 567–580, 2022.

[2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, et al. Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA 2020)*, pages 145–158, 2020.

[3] Jenny Lynn Almerol, Elisabetta Boella, Mario Spera, and Daniele Gregori. Accelerating gravitational *N*-body simulations using the RISC-V-based tenstorrent wormhole™. *arXiv preprint*, arXiv:2509.19294, 2025.

[4] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *Computer*, 35(1):70–78, 2002.

[5] Randal E Bryant. Data-intensive supercomputing: The case for disc. 2007.

[6] Nafea Bshara. Aws trainium: the journey for designing and optimization full stack ml hardware. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 4–4, 2024.

[7] Doug Burger, James R Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. *ACM SIGARCH Computer Architecture News*, 24(2):78–89, 1996.

[8] Cerebras Systems. Cerebras systems: Achieving industry best AI performance through a systems approach. Technical report, Cerebras Systems, 2021. Whitepaper 03.

[9] Cerebras Systems. The cerebras software development kit: A technical overview. Whitepaper, 2023.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[12] Zeshan Chishti and Berkin Akin. Memory system characterization of deep learning workloads. In *Proceedings of the International Symposium on Memory Systems*, pages 497–505, 2019.

[13] Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Hao Yu, Yida Wang, and Gennady Pekhimenko. Tilus: A tile-level GPGPU programming language for low-precision computation. *arXiv preprint arXiv:2504.12984*, 2025.

[14] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, et al. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[15] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

[16] Graphcore Ltd. Poplar graph framework software. https://www.graphcore.ai/products/poplar, 2022. Accessed: 2024-03-19.

[17] Sumanth Gudaparthi, Sarabjeet Singh, Surya Narayanan, Rajeev Balasubramonian, and Visvesh Sathe. CANDLES: Channel-aware novel dataflow-microarchitecture co-design for low energy sparse neural network acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 876–891, 2022.

[18] Waqas Gul, Maitham Shams, and Dhamin Al-Khalili. Sram cell design challenges in modern deep sub-micron technologies: An overview. *Micromachines*, 13(8):1332, 2022.

[19] James Hamilton. Tesla project dojo overview. https://perspectives.mvdirona.com/2021/08/tesla-project-dojo-overview/, 2021. Blog post.

[20] Congjie He, Yeqi Huang, Pei Mu, Mike Wang, Ziming Miao, Jilong Xue, Lingxiao Ma, Fan Yang, and Luo Mai. Wafer-scale ai compute: A system software perspective.

[21] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[22] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pages 10–14. IEEE, 2014.

[23] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics*, 38(6), 2019.

[24] Liancheng Jia, Zizhang Luo, Liqiang Lu, and Yun Liang. Tensorlib: A spatial accelerator generation framework for tensor algebra. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 865–870. IEEE, 2021.

[25] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint*, arXiv:1804.06826, 2018.

[26] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore IPU architecture via microbenchmarking. *arXiv preprint*, arXiv:1912.03413, 2019.

[27] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[28] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, Vasilii Filippov, Jing Zhang, Jing Zhou, Bragadeesh Natarajan, and Mayank Daga. MIOpen: An open source library for deep learning primitives. *CEUR Workshop Proceedings*, 2744, 2020.

[29] Khronos Group. *The OpenCL Specification, Version 3.0*, 2020. Available from the Khronos OpenCL Registry.

[30] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[31] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

[32] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

[33] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. A communication-centric approach for designing flexible DNN accelerators. *IEEE Micro*, 38(6):25–35, 2018.

[34] Yongin Kwon, JooHyoung Cha, Sehyeon Oh, Misun Yu, Jeman Park, and Jemin Lee. Luthier: Bridging autotuning and vendor libraries for efficient deep learning inference. *ACM Transactions on Embedded Computing Systems*, 24(5s), 2025.

[35] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 242–251, 2019.

[36] Jonathan S. Lew, Deval A. Shah, Suchita Pati, Shaylin Cattell, Mengchi Zhang, Amruth Sandhupatla, Christopher Ng, Negar Goli, Matthew D. Sinclair, Timothy G. Rogers, and Tor M. Aamodt. Analyzing machine learning workloads using a detailed GPU simulator. *CoRR*, abs/1811.08933, 2018.

[37] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. Lisa: Graph neural network based portable mapping on spatial accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 444–459. IEEE, 2022.

[38] Andrea Lottarini, João P. Cerqueira, Thomas J. Repetti, Stephen A. Edwards, Kenneth A. Ross, Mingoo Seok, and Martha A. Kim. Master of none acceleration: A comparison of accelerator architectures for analytical query processing. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 762–773, 2019.

[39] Liqiang Lu, Zizhang Luo, Size Zheng, Jieming Yin, Jason Cong, Yun Liang, and Jianwei Yin. Rubick: A unified infrastructure for analyzing, exploring, and implementing spatial architectures via dataflow decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(4):1177–1190, 2023.

[40] Yixuan Luo, Cheng Tan, Nicolas Bohm Agostini, Ang Li, Antonino Tumeo, Nirav Dave, and Tong Geng. Ml-cgra: An integrated compilation framework to enable efficient machine learning acceleration on cgras. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[41] Zizhang Luo, Liqiang Lu, Size Zheng, Jieming Yin, Jason Cong, Jianwei Yin, and Yun Liang. Rubick: A synthesis framework for spatial architectures via dataflow

decomposition. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[42] Xingchen Man, Jianfeng Zhu, Guihuan Song, Shouyi Yin, Shaojun Wei, and Leibo Liu. Casmap: agile mapper for reconfigurable spatial architectures by automatically c lustering intermediate representations a nd s cattering mapping process. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 259–273, 2022.

[43] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[44] Microsoft. triton-shared: A shared middle-layer for the triton compiler. https://github.com/microsoft/triton-shared, 2025.

[45] Pengyu Mu, Yi Liu, Rui Wang, Guoxiang Liu, Hangcheng An, Qianhe Zhao, Hailong Yang, Chenhao Xie, Zhongzhi Luan, Chunye Gong, and Depei Qian. Deep learning operators performance tuning for changeable sized input data on tensor accelerate hardware. *IEEE Transactions on Computers*, 74(6):2101–2113, 2025.

[46] Onur Mutlu. Memory scaling: A systems architecture perspective. In *2013 5th IEEE International Memory Workshop*, pages 21–25. IEEE, 2013.

[47] NASA Advanced Supercomputing Division. Basics on NVIDIA GPU hardware architecture. https://www.nas.nasa.gov/hecc/support/kb/basics-on-nvidia-gpu-hardware-architecture_704.html, 2025. HECC Knowledge Base Article 704.

[48] Tim Noack, Louis Krüger, and Andreas Koch. Accelerating sparse linear solvers on intelligence processing units. In *Proceedings of the 39th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1023–1035, 2025.

[49] NVIDIA Corporation. *CUDA C Programming Guide*, 2017. PG-02829-001_v8.0.

[50] Nvidia Corporation. Nvidia cuda tile. https://developer.nvidia.com/cuda/tile, 2025. Accessed: 2025-12-6.

[51] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. Timeloop: A systematic approach to DNN accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[52] Dylan Patel. Tenstorrent wormhole analysis - a scale out architecture for machine learning that could put nvidia on their back foot, June 2021.

[53] Dylan Patel. Tenstorrent blackhole, grendel, and buda - a scale out architecture for sparsity, conditional execution, and dynamic routing, April 2022.

[54] Hongwu Peng, Caiwen Ding, Tong Geng, Sutanay Choudhury, Kevin Barker, and Ang Li. Evaluating emerging AI/ML accelerators: IPU, RDU, and NVIDIA/AMD GPUs. *arXiv preprint arXiv:2311.04417*, 2024.

[55] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. Sambanova sn10 RDU: A 7nm dataflow architecture to accelerate software 2.0. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 350–352, 2022.

[56] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402, 2017.

[57] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, 2013.

[58] SambaNova Systems. Accelerated computing with a reconfigurable dataflow architecture. Technical report, SambaNova Systems, 2021. Whitepaper.

[59] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, et al. T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–189. IEEE, 2019.

[60] Moritz Thüning. Attention in sram on tenstorrent grayskull. *arXiv preprint arXiv:2407.13885*, 2024.

[61] Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2019.

[62] Dirk Van Essendelft, Patrick Wingo, Terry Jordan, Ryan Smith, and Wissam Saidi. A system level compiler for massively-parallel, spatial, dataflow architectures. *arXiv preprint arXiv:2506.15875*, 2025.

[63] Dirk Van Essendelft, Patrick Wingo, Terry Jordan, Ryan Smith, and Wissam A. Saidi. A system level compiler for massively-parallel, spatial, dataflow architectures. *arXiv preprint arXiv:2506.15875*, 2025.

[64] Erwei Wang, Samuel Bayliss, Andra Bisca, Zachary Blair, Sangeeta Chowdhary, Kristof Denolf, Jeff Fifield, Brandon Freiberger, Erika Hunhoff, Phil James-Roxby, et al. From loop nests to silicon: Mapping ai workloads onto amd npus with mlir-air. *arXiv preprint arXiv:2510.14871*, 2025.

[65] Jie Wang, Licheng Guo, and Jason Cong. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 93–104, 2021.

[66] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. Tilelang: A composable tiled programming model for AI systems. *arXiv preprint arXiv:2504.17577*, 2025.

[67] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

[68] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunaratne, Li-Shiuan Peh, and Tulika Mitra. Morpher: An open-source integrated compilation and simulation framework for cgra. In *Fifth Workshop on Open-Source EDA Technology (WOSET)*, 2022.

[69] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[70] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[71] Jiaqi Yang, Hao Zheng, and Ahmed Louri. DiTile-DGNN: An efficient accelerator for distributed dynamic graph neural network inference. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1240–1253, 2025.

[72] Tianyi Yu, Omar Ragheb, Stephen Wicklund, and Jason Anderson. Mlir-to-cgra: A versatile mlir-based compileir framework for cgras. In *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–192. IEEE, 2024.

[73] Jinming Zhang, Xi Fan, Yaoyao Ye, Xuyan Wang, Guojie Xiong, Xianglun Leng, Ningyi Xu, Yong Lian, and Guanghui He. INDM: Chiplet-based interconnect network and dataflow mapping for DNN accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(4):1107–1120, 2024.

[74] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 874–887, 2022.

[75] Jinming Zhuang, Shaojie Xiang, Hongzheng Chen, Niansong Zhang, Zhuoping Yang, Tony Mao, Zhiru Zhang, and Peipei Zhou. Aries: An agile mlir-based compilation flow for reconfigurable devices with ai engines. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 92–102, 2025.