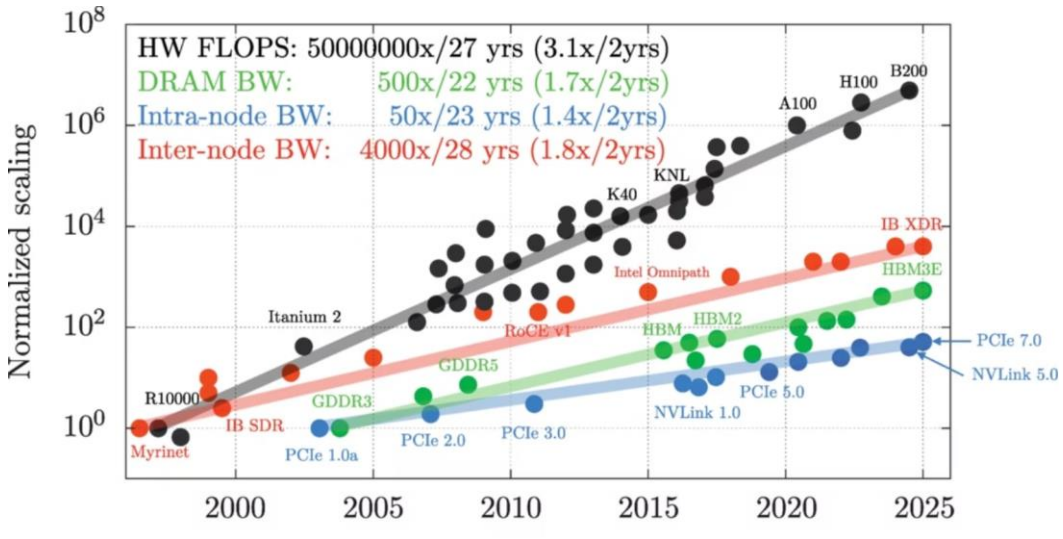
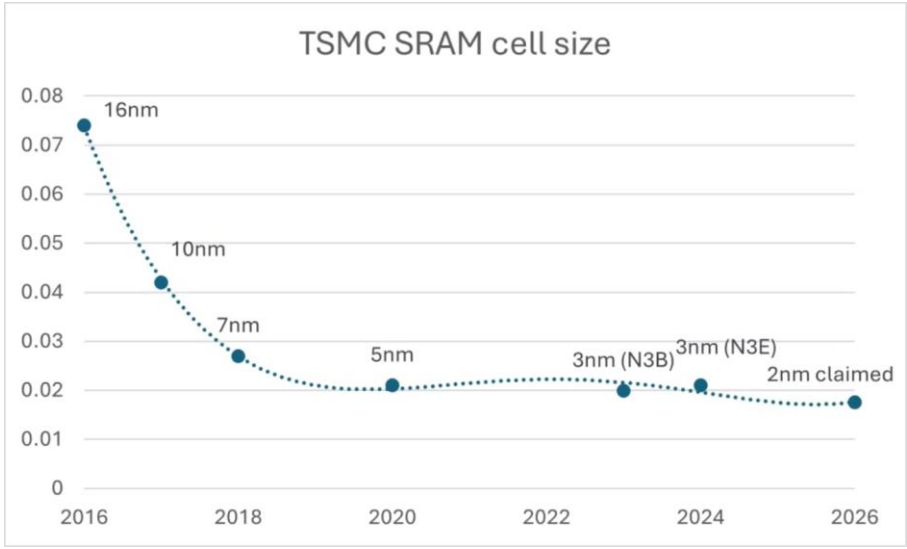
TileLoom: An end-to-end Compiler from Tile-Based DSLs to Spatial Dataflow Accelerators

Wei Li, **Zhenyu Bai**, Heru Wang, Pranav Dangi, Zhiqiang Zhang, Cheng Tan, Huiying Lan, Weng-Fai Wong, Tulika Mitra
National University of Singapore

The Memory Wall



Bandwidth Gap

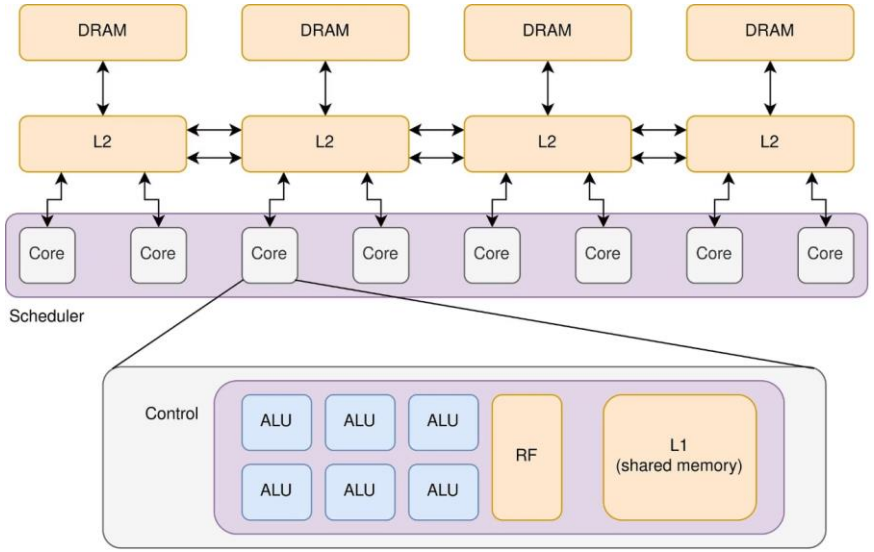


SRAM Capacity

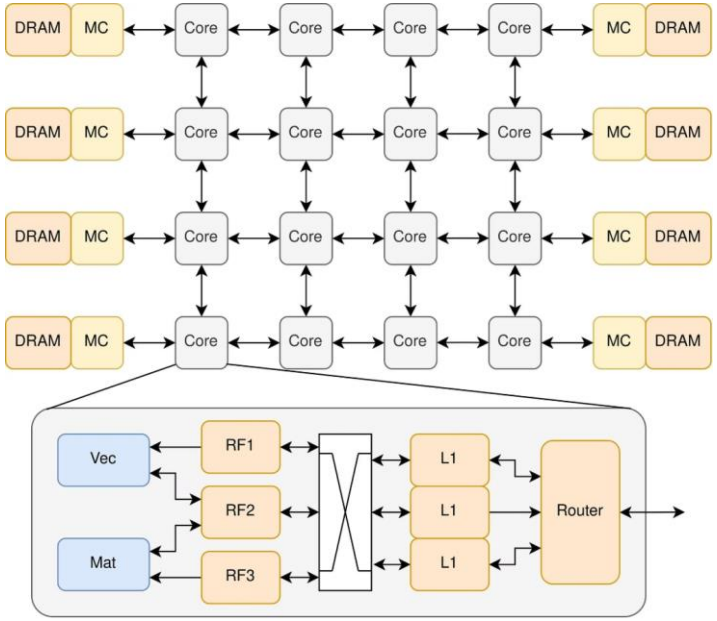
Roccon, A., et al. "GPU-accelerated simulations of turbulence: Review of current applications and future perspectives." Physical Review Fluids 11.3 (2026): 034905.

semiengineering.com/memory-wall-gets-higher/

Spatial Dataflow Accelerators:



- Hardware-managed inter-core data movement through L2 cache
- Hardware-managed intra-core task scheduling
- Hardware-managed intra-core data movements
- *Simpler programming model*
- Modern GPUs are becoming dataflow



- Software-managed inter-core data movements through the Network on Chip, often no L2.
- Software-managed intra-core data movements (among the buffers)
- Software-managed explicit intra-core synchronizations
- *Significantly harder to program*

Tradeoffs & Challenges

- Lesser hardware managements ~= higher peak TOPS (per area/power/price/...)
- More regular/predictable workloads (AI workloads) = room for improvements with compilers
- Higher performance volatility: good potential TOPS, bad achieved TOPS without dedicated optimizations
- In practice, the performance relies on vendor provided, expert tuned libraries for specific kernels
- Programmer needs deep expertises of the hardware, needs to manage almost everything through software.

Is this manageable by compiler to easily get a relatively good performance?

Tile-based DSLs

- Grid-Block hierarchy matches two levels of dataflow: inter-core and intra-core
- Easy to program: abstraction to the block level details

```
for tile_m, tile_n in hl.tile([m, n]):
    acc = hl.zeros([tile_m, tile_n], dtype=torch.float32)
    for tile_k in hl.tile(k):
        acc = torch.addmm(acc, x[tile_m, tile_k], y[tile_k, tile_n])
    out[tile_m, tile_n] = epilogue(acc, (tile_m, tile_n))
return out
```

TileLoom's Objectives

- End-to-end compilation from tile-based languages to executable
- Compiler managed explicit inter- and intra-core data movements
- Supporting various dataflow architectures

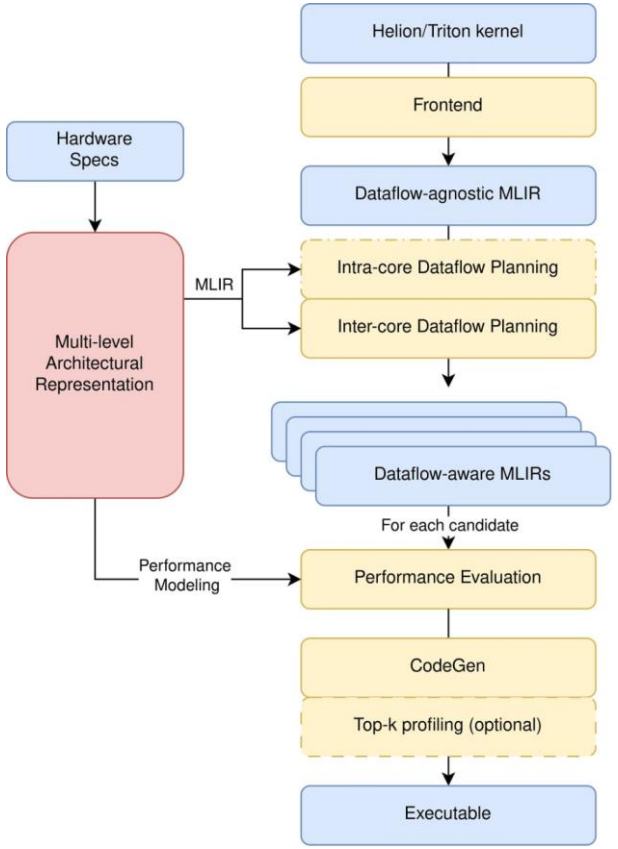
TileLoom Framework Overview

Frontend

Tile-based programs to unified dataflow-agnostic MLIR

Multi-level Architecture Representation

Describes hardware at intra-core, memory, and NoC levels. Drives Dataflow Planning and Block-level Optimizations.



Dataflow Planning

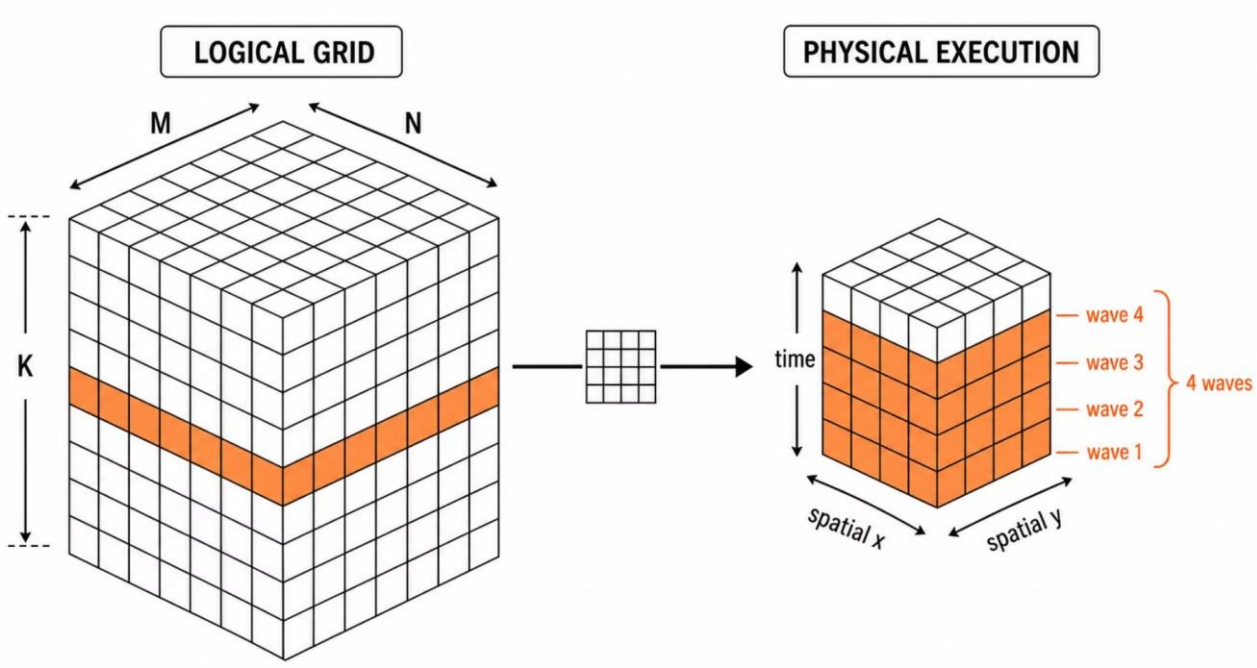
Spatiaotemporal mapping of the grid over the target architecture (inter-core dataflow).

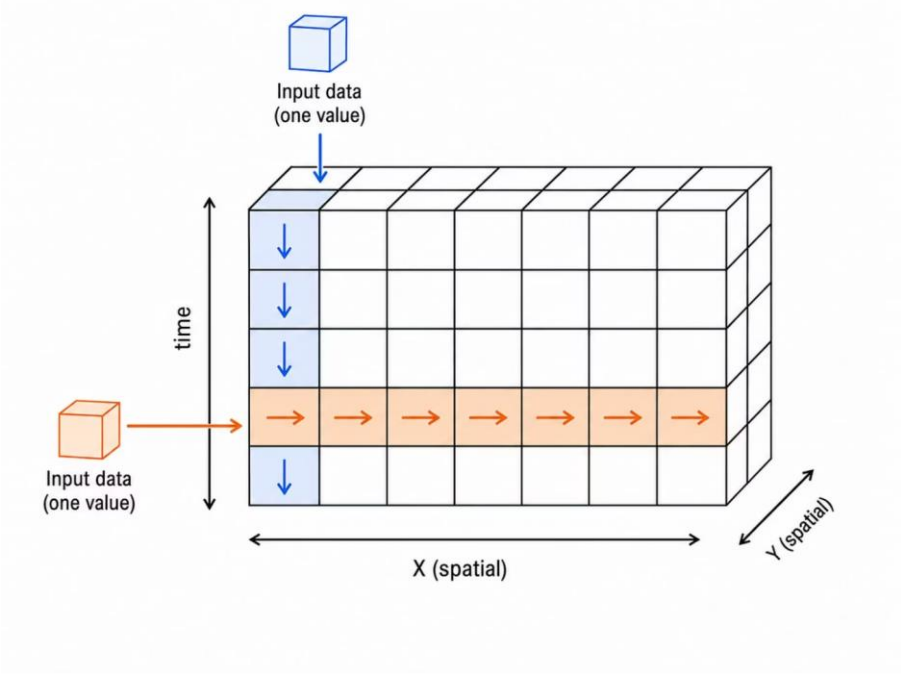
Block-level Optimizations & Code Gen

Maps the block-level program to cores with back-end optimizations.

Emit block-level executable for data movements, computations, and synchronizations (intra-core dataflow).

Spatiotemporal Mapping (Grid Mapping) & Data Reuse





Data Reuse Analysis

Spatial Reuse

If an access does not depend on a spatial core index, that tile can be **shared across cores** via broadcast rather than loaded repeatedly from DRAM.

Temporal Reuse

If an access does not depend on a temporal loop variable, that tile can be **reused across time** by hoisting the load outside the loop at the cost of (potentially) extra memory usage.

```
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}}
        // ...
      }
    }
  }
}
```

```
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]
      // ...
    }
  }
}
```

Dataflow Candidates

```
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}}
  }
}
```

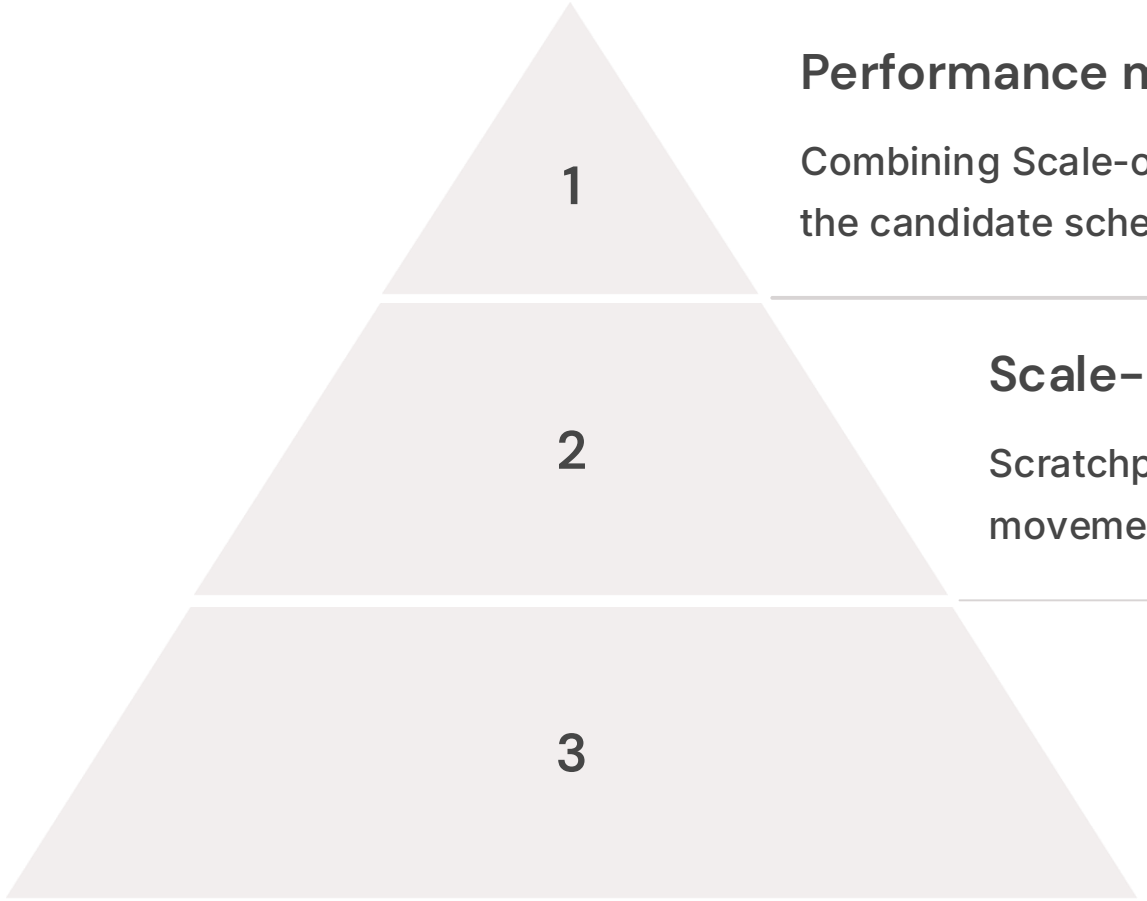
One schedule is defined by:

- tiling decisions (which loops, by which factor)
- loop ordering
- memory oprations' position w.r.t the loops
- memory operations' type (e.g. spatial broadcast, DMA, global loads, ...)

Different such schedules will be evaluated by the performance model later.

Hardware Dialect

Hardware Representation: The `df` Dialect



Performance model

Combining Scale-out + Intra-Core architecture, auto generated to evaluate the candidate schedules

Scale-out Architecture

Scratchpads, DRAM channels, connectivity — used during data movement planning

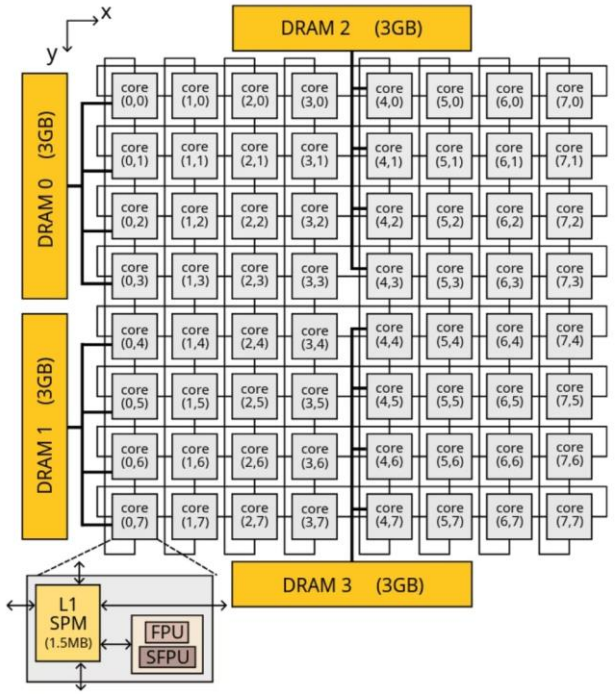
Intra-Core Architecture

Matrix, vector, scalar units, and their compute throughput

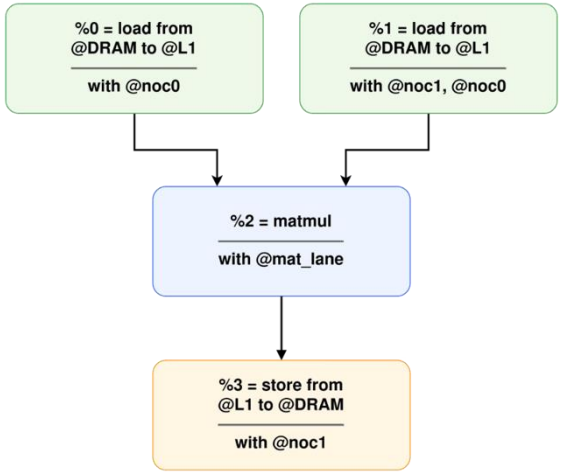
Example MLIR description

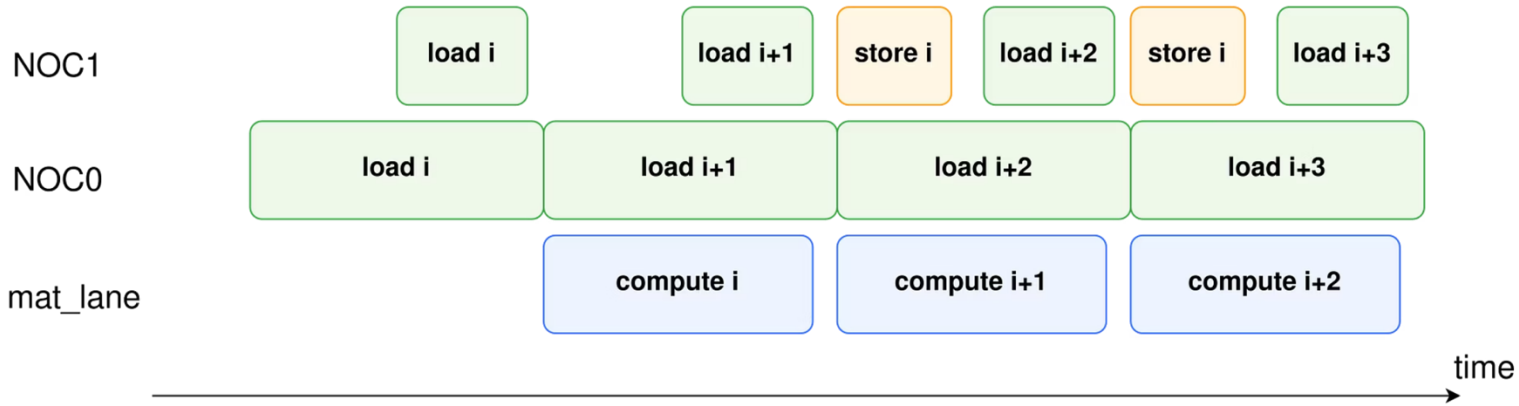
```
module {
    // functional untis description
    %mat_unit = df.mat "FPU" {shape = [32, 32, 32], throughput = 128}
    %vec_unit = df.vec "SFPU" {shape = [32], throughput = 64}
    // scale-out description
    %x = df.spatial_dim "x", 8
    %y = df.spatial_dim "y", 8
    %cores = df.core "core" {scaleout=(%x, %y) , scalein=(%mat_unit, %vec_unit, [8,1])}
    %L1 = df.memory "L1" {scaleout=(%x, %y) , size = 32768, bandwidth = 64}
    %core_to_mem = df.mux %cores, %L1, {map = affine_map<(d0, d1) -> (d0, d1)>}
    %noc_h = df.interconnects "horizontal_links" %L1 : !df.memory, %L1 : !df.memory, {map = affine_map<(d0, d1) -> ((d0 + 1) mod 8, d1)>, bandwidth = 128}
    %noc_v = df.interconnects "vertical_links" %L1 : !df.memory, %L1 : !df.memory, {map = affine_map<(d0, d1) -> (d0, (d1 + 1) mod 8)>, bandwidth = 128}
    // dram
    %dram_idx = df.spatial_dim "d", 4
    %drams = df.memory "DRAM" {scaleout=(%dram_idx) , size = 34359738368, bandwidth = 512}
    %to_dram = df.interconnects %L1: !df.memory, %drams : !df.memory, {map = affine_map<(d0, d1) -> (d0 ceildiv 4 + 2 * (d1 ceildiv 4))>}
}
```

- spatial dimensions for grid mapping
- interconnect descriptions for matching the spatial reuse to netowork broadcast
- bandwidth/throughput for performance modeling



Evaluating Dataflow Candidates





Evaluations

Experimental Setup

Hardware Platforms

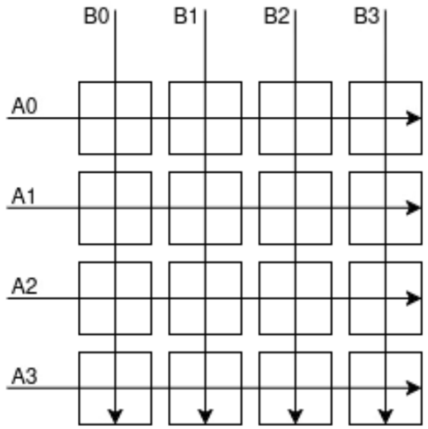
Spec	Wormhole	Blackhole
Topology	8×8 mesh	12×10 mesh
SRAM	108 MB	180 MB
DRAM	12 GB GDDR6	32 GB GDDR6
Off-chip BW	288 GB/s	512 GB/s
FP16 Compute	64 TFLOPS	162 TFLOPS
Price	~1000 USD	~1000 USD

Software Stack

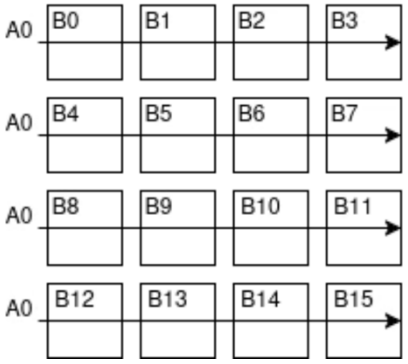
- **Frontends:** Triton or Helion (Depends on kernel's public availability)
- **Backend:** TT-Metalium
- Hardware parameters recovered via microbenchmarks and instantiated in the `df` model
- We only use the top-1 (fully relying on the hardware model, without profiling)
- **Baseline**: TT-NN library

GeMM

TTNN GeMM



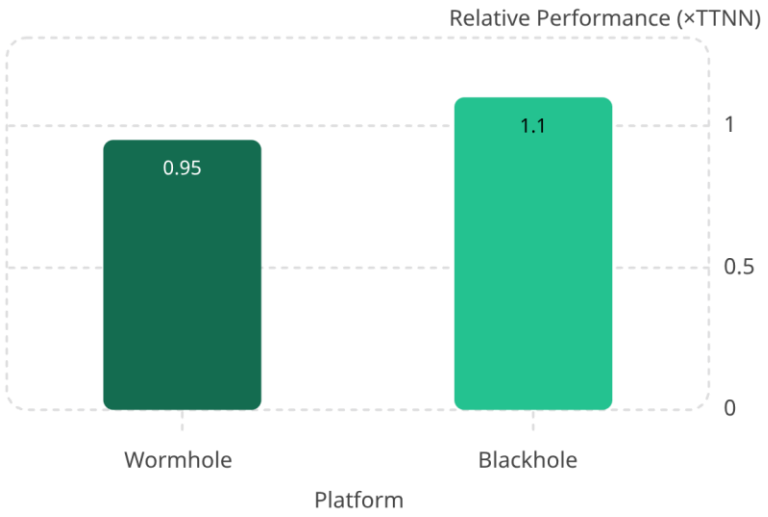
TTNN 2D broadcast



TTNN 1D broadcast

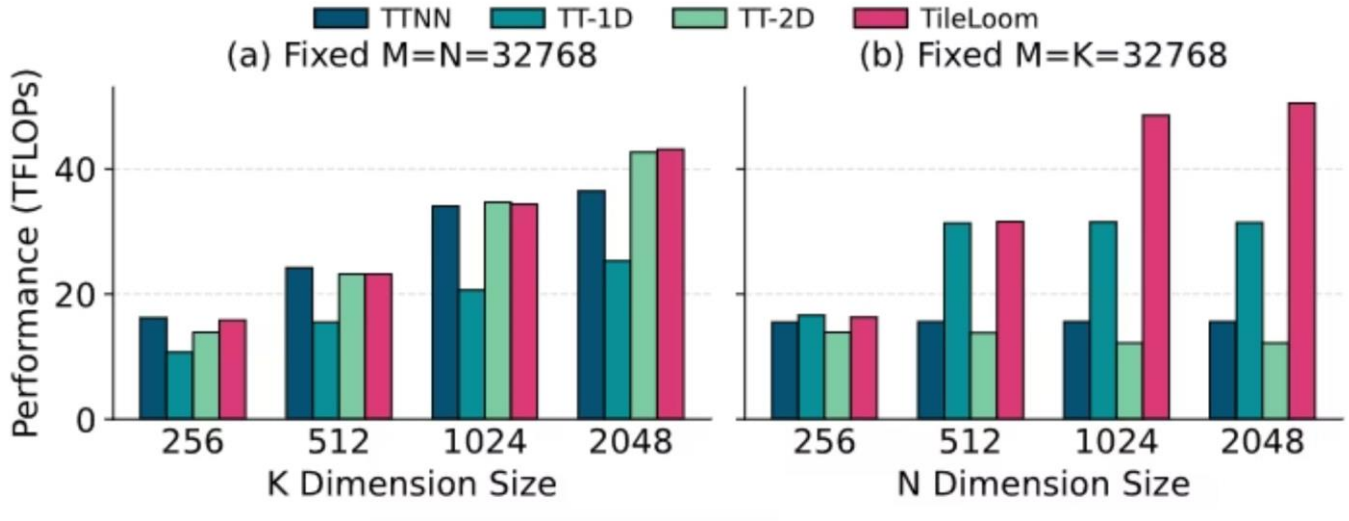
- TTNN uses two fixed broadcast patterns
  - 2D: two inputs are streamed independently vertically and horizontally using the NoC
  - 1D: the big input is streamed across the entire chip, the small input is loaded w/o broadcast
- TTNN chooses one of the two templates based on the ratio of M/K/N, prefer 1D when big K, small M/N.
- TTNN has separated strategy for choosing the block sizes

TileLoom GeMM



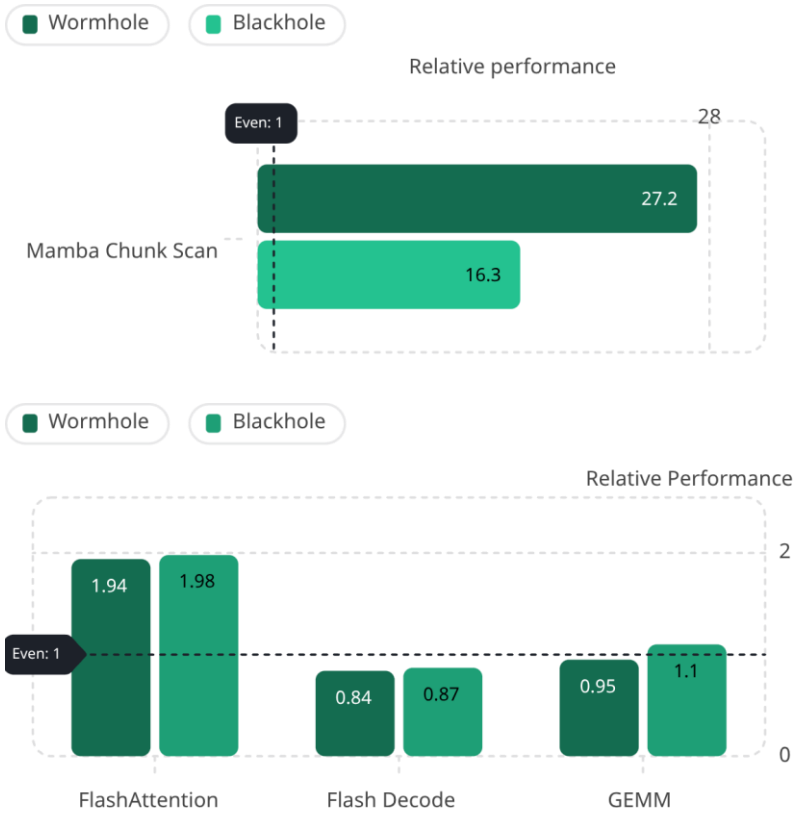
- TileLoom has no fixed spatial/temporal reuse pattern, nor fixed block sizes, resulting in larger dataflow design space.
- TileLoom achieved comparable performance vs. highly optimized TTTN library.
- Still a small gap vs. TTNN due to backend optimizations.

Irregular GEMM



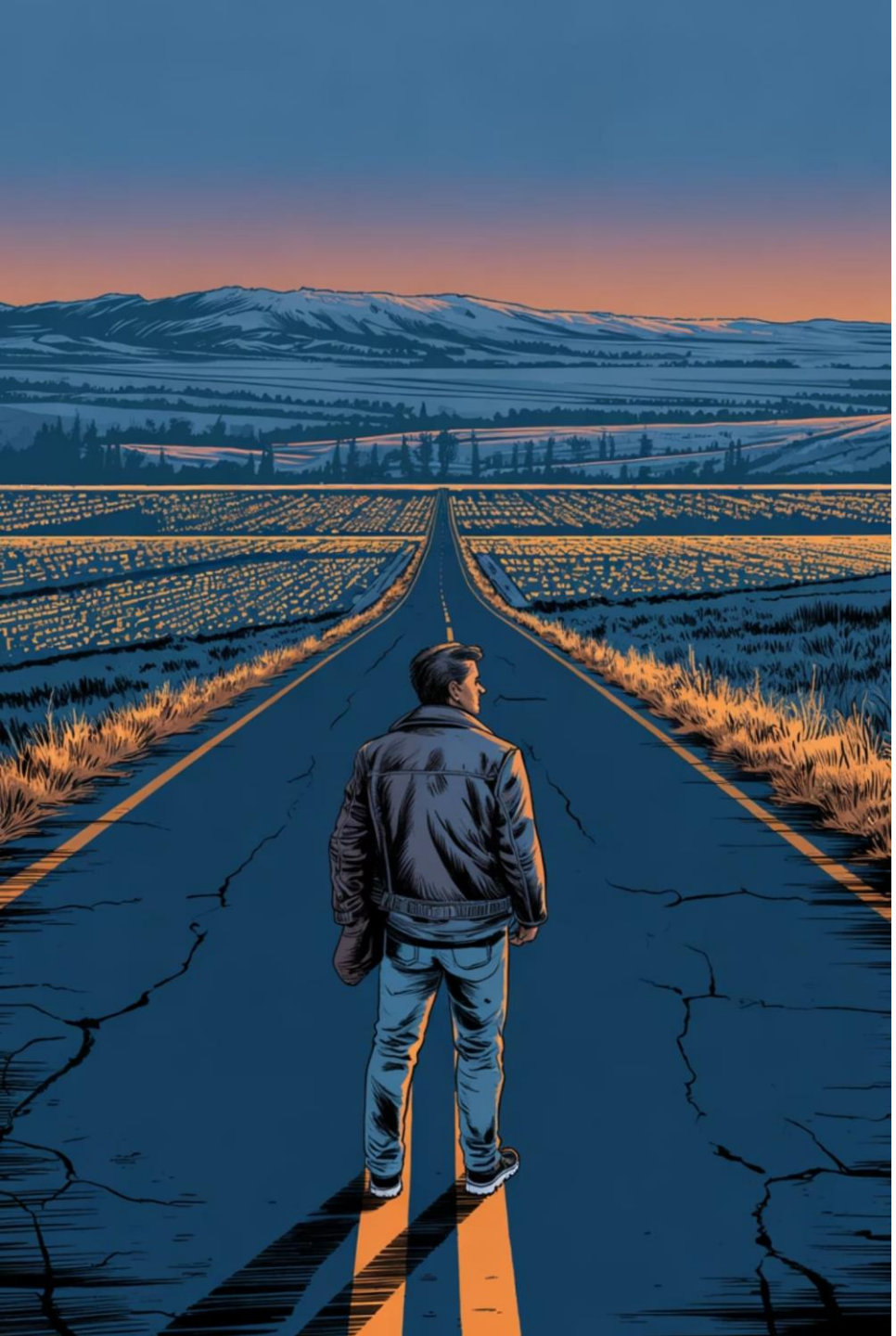
- TTNN could lose against its own broadcast patterns, meaning the selection strategy is not perfect.
- TileLoom performs better on these irregular shapes for relying on hardware model and performance estimation rather than fixed dataflow selection strategies.
- TileLoom can beat both TT-1D & TT-2D

Results Summary



TileLoom is compared against **TTNN**, Tenstorrent's vendor library.

- **Mamba Chunk Scan:** massive win — TTNN lacks a fused kernel
- **FlashAttention:** strong win from data reuse using NoC broadcast
- **Flash Decode:** near-competitive; limited mapping freedom, and split-k style spatial reuse. The gap mostly comes from specialized block-level optimizations.
- **GEMM:** competitive, beats TTNN on Blackhole for better reuse pattern and block sizes on irrugelar shapes
- Blackhole improves compute throughput by **2.53×** but off-chip bandwidth by only **1.78×**. This makes Blackhole **more memory-bound**, increasing the value of dataflow planning.



Takeaways

- Architectures are evolving with more "dataflow features", improving the hardware efficiency, but much harder to program
- Tile-based DSLs are good abstraction to free the programmers from the hardware details, let the compiler to take the decisions
- Dataflow features improve the predictibility of the hardware, making static scheduling more viable, but requires hardware details
- Dataflow + compiler + hardware info could be a direction for both hardware efficiency and sustainable performance