ETH zürich

MemPool meets Systolic

Mid-term Report

Samuel Riedel
Matheus Cavalcante
Prof. Luca Benini

Content

1. Refresher on MemPool
   - MemPool's architecture
   - Challenging physical implementation
2. Completing MemPool's architecture
   - Full instruction path
   - Improved L2 bandwidth
   - DMA
3. Adding a systolic mode
   - Software emulation
   - Hardware extensions
   - Halide support

MemPool: A 256-core Many-core Cluster

- Goal: **An efficient scaled-up many-core system with low-latency shared L1 memory**
  - 256 cores
  - 1 MiB of shared L1 data memory
  - ≤ 5 cycles latency (without contention)
- Physical-aware GF 22FDX design
  - 700 MHz at typical conditions
  - 480 MHz at worst-case conditions

- First presented in DATE 2021

MemPool: A Shared-L1 Memory Many-Core Cluster with a Low-Latency Interconnect

Matheus Cavalcante
ETH Zürich
Zürich, Switzerland
matheusd *at* iis.ee.ethz.ch

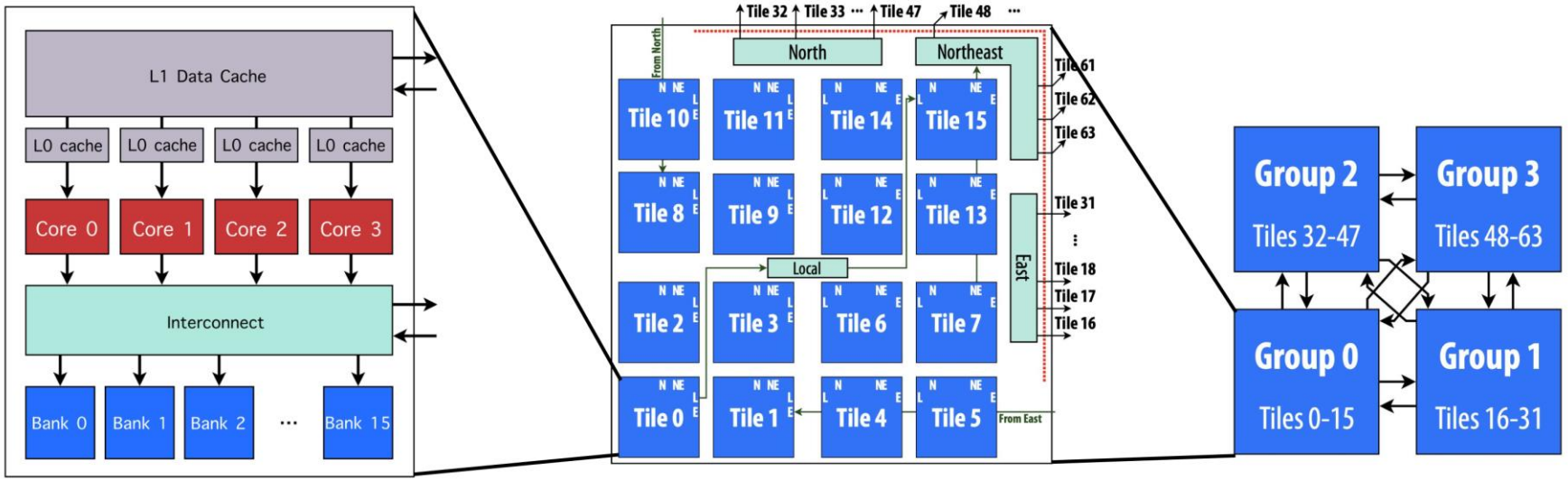
Samuel Riedel
ETH Zürich
Zürich, Switzerland
sriedel *at* iis.ee.ethz.ch

Antonio Pullini
GreenWaves Technologies
Grenoble, France
pullinia *at* iis.ee.ethz.ch

Luca Benini
ETH Zürich
Zürich, Switzerland
Università di Bologna
Bologna, Italy
lbenini *at* iis.ee.ethz.ch

MemPool's Hierarchy

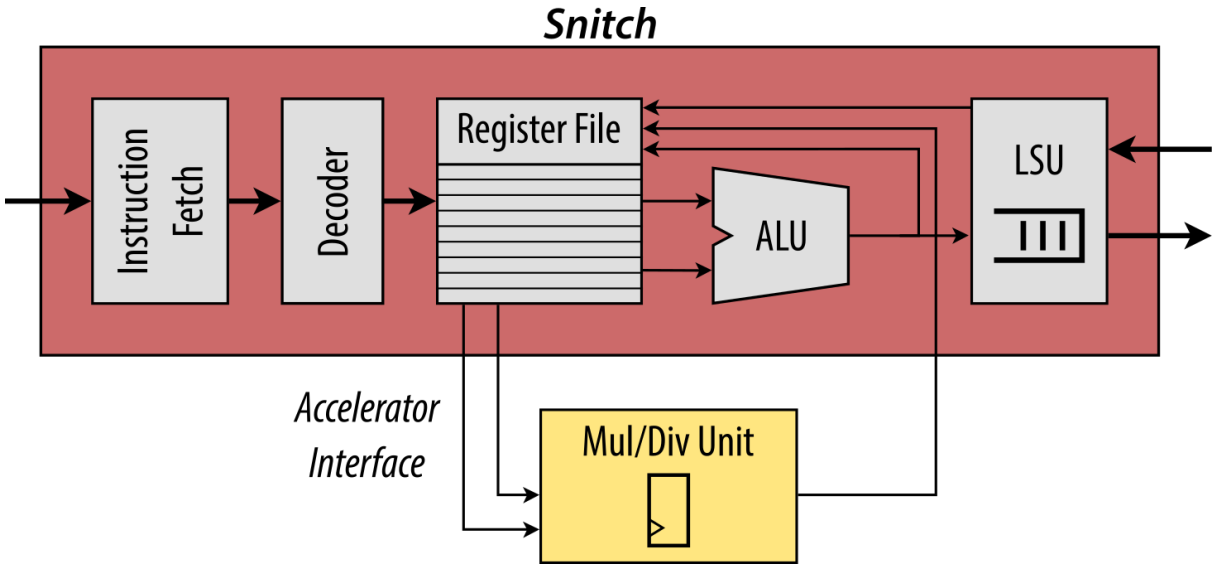
- Tile:
  - 4 32-bit cores
  - 16 banks
  - Single cycle memory access
- Group:
  - 64 cores
  - 256 banks
  - 3 cycles latency
- Cluster
  - 256 cores
  - 1 MiB of memory (1024 banks)
  - 5 cycles of latency



MemPool Tile Group Cluster

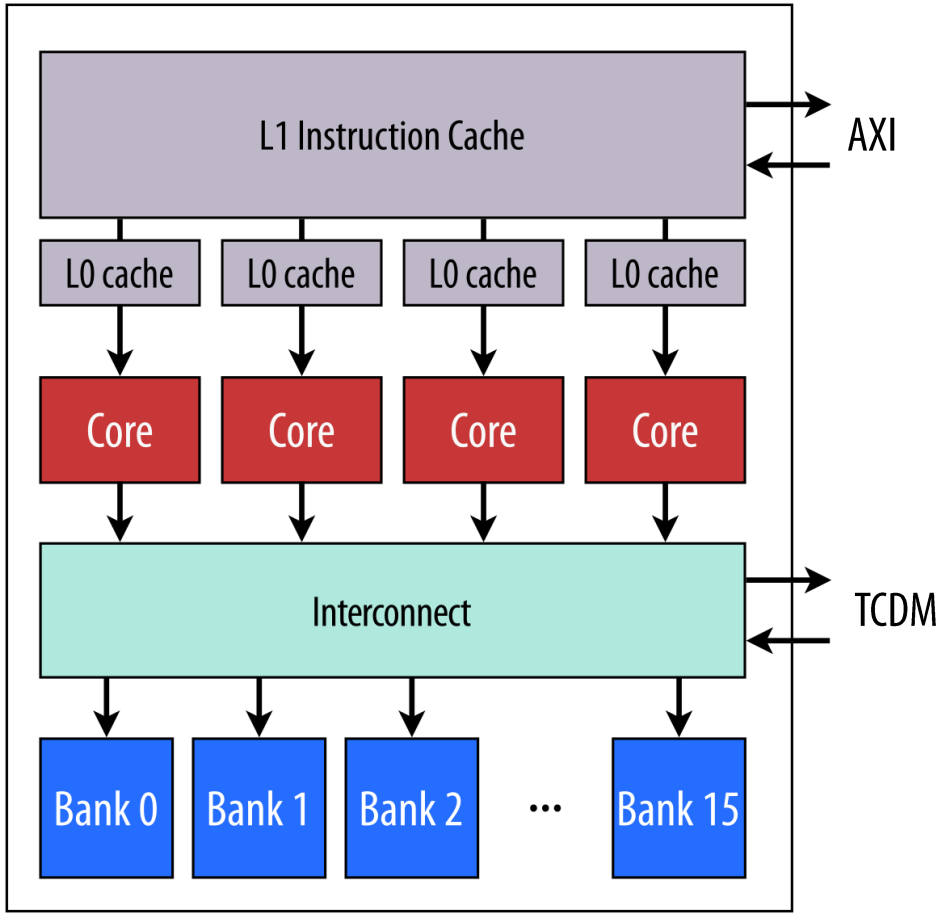
Processor: **Snitch**

- RISC-V 32-bit IMAXpulpimg
- Single-stage processor
  - Small area
- Accelerator interface
  - Pipelined
  - Complex instructions
- Tolerates multiple outstanding transactions



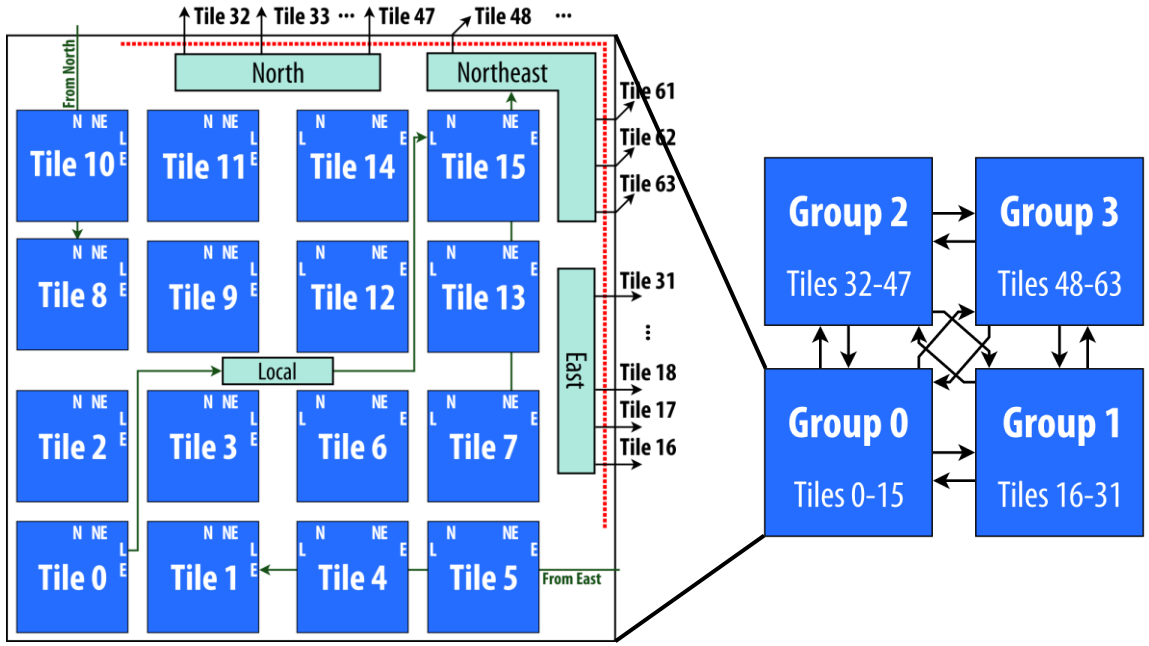
The Base Unit of MemPool's Hierarchy: **Tiles**

- Four Snitch cores
- 2 KiB shared L1 instruction cache
  - 2-way set associative
  - Refill port connected to AXI
- 32 instructions of L0 cache
  - Private for each core
  - Implement prefetching
- 16 L1 SPM Banks
  - 16 KiB
  - Accessible from local banks within one cycle
- Four ports to connect to remote tiles' TCDM



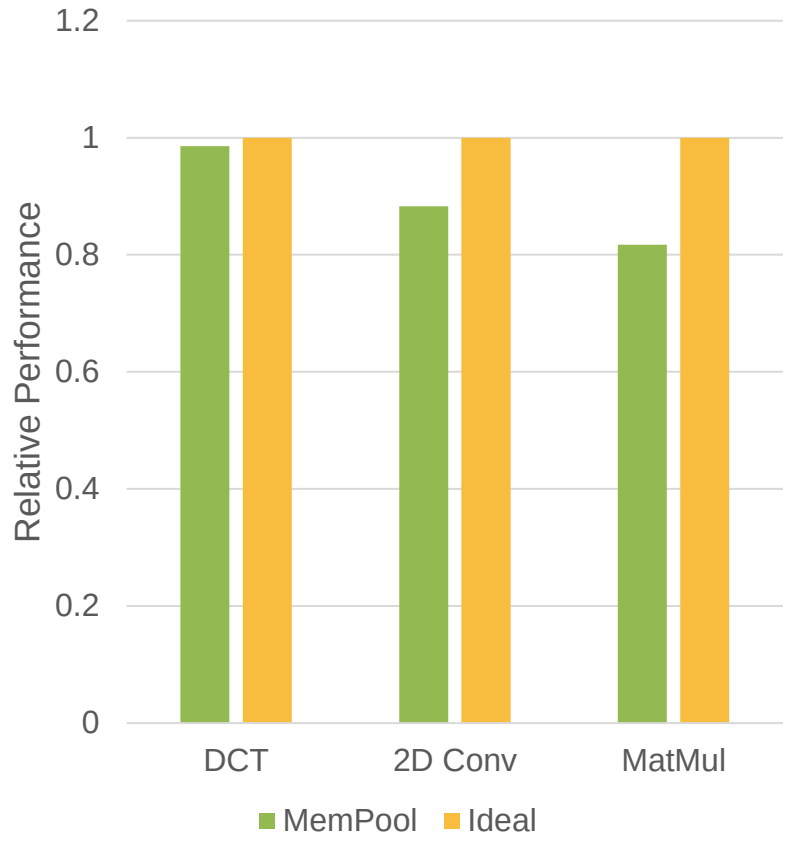
Intermediate hierarchy level: **Groups**

- 16 tiles compose a *group*
- Tiles in the same group can be accessed within 3 cycles
  - "Local" interconnect is realized as a 16x16 fully connected crossbar
- Each group has three other 16x16 crossbars to access remote groups
  - "North," "Northeast," and "East" interconnects
  - Register boundary implies remote groups can be accessed within 5 cycles
- Four groups make up the MemPool cluster



Can we compete with the impossible?

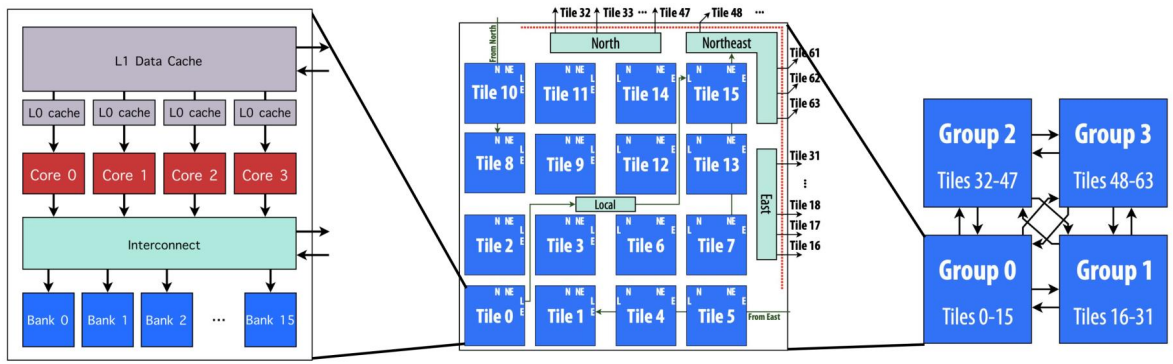
- Ideal system
  - 256 cores in one cluster
  - 256x1024 fully-connected logarithmic crossbar
  - Single cycles access latency
  - Physically infeasible
- Only local accesses
  - Almost ideal performance
- Mostly global accesses
  - Still achieve 80% of the baseline's performance



MemPool as a **platform**

- Important to distinguish between the MemPool **design** and the MemPool **platform**

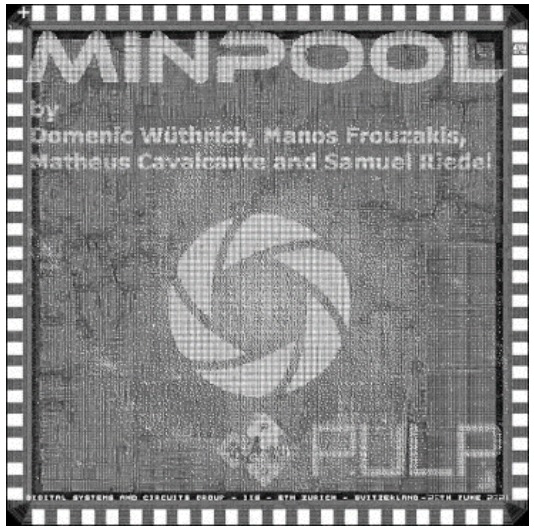
- Highly flexible platform
  - We can tune the number of cores per tile, and tiles per group, and groups per cluster to generate several MemPool configurations

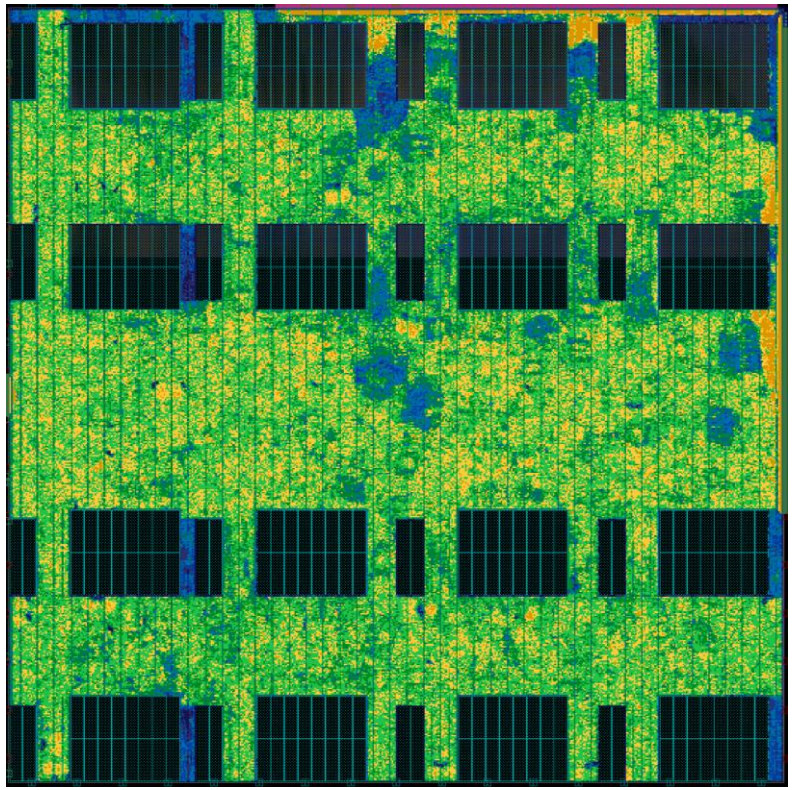


The MemPool Family: MinPool

- **MinPool**
  - Our first MemPool tape-out!
  - The little 16-core version of the **MemPool** design

The MemPool Family: MemPool





- **MemPool**
  - A peculiar design in terms of size, connectivity, routing
  - DZ uses it as a **benchmark** for EDA tools
  - GlobalFoundries 22FDX implementation, Synopsys Fusion Compiler 2021.06
    - 500 MHz (wc), 3.6 mm × 3.6 mm macro

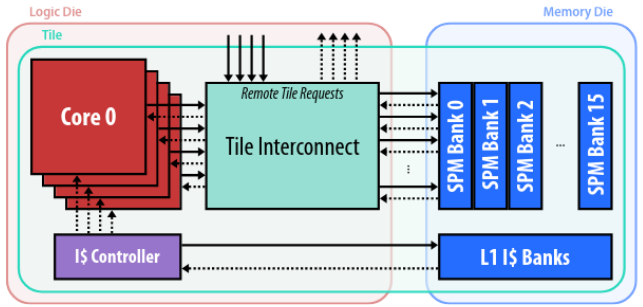
MemPool-3D

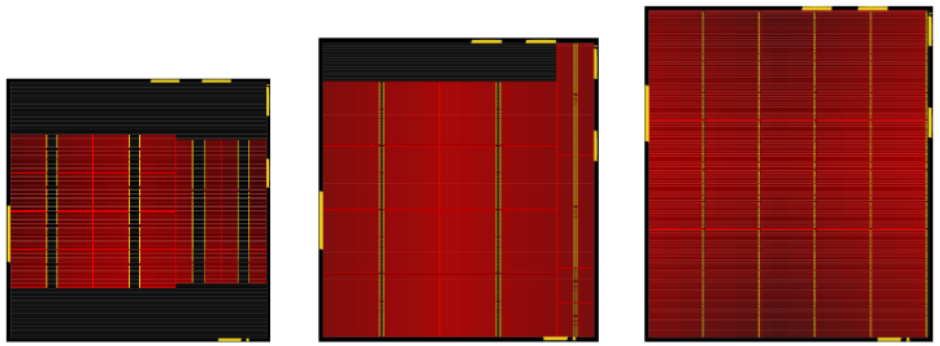
- **Memory-on-Logic** implementation of MemPool
- Accepted for publication in **DATE2022** (arxiv.org/abs/2112.01168)

MemPool-3D: Boosting Performance and Efficiency of Shared-L1 Memory Many-Core Clusters with 3D Integration

Matheus Cavalcante*, Anthony Agnesina†, Samuel Riedel*, Moritz Brunion‡,
Alberto García-Ortiz‡, Dragomir Milojevic§, Francky Catthoor§, Sung Kyu Lim†, and Luca Benini*¶

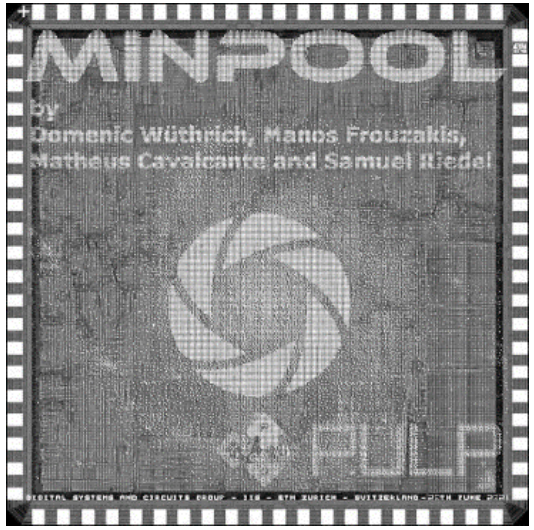
- Implementation of MemPool with 1, 2, 4, 8 MiB of L1
- Leading to a higher utilization of the memory die

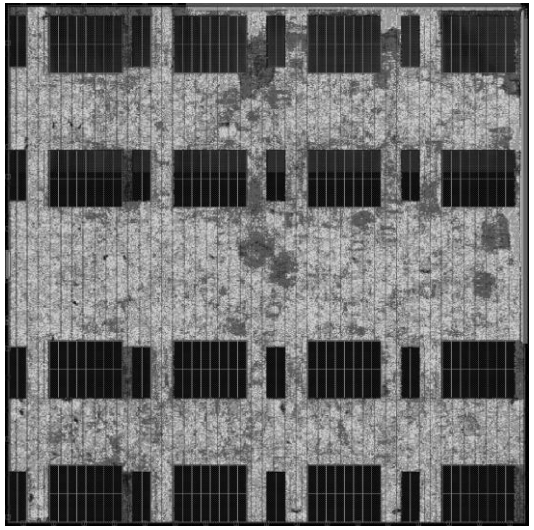


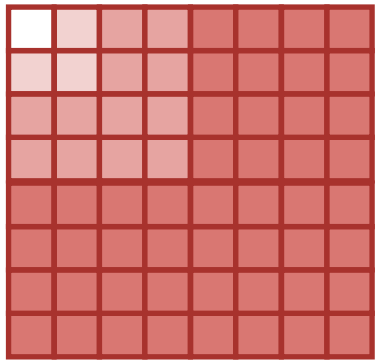


(a) MemPool-3D$_{1\,\text{MiB}}$. (b) MemPool-3D$_{4\,\text{MiB}}$. (c) MemPool-3D$_{8\,\text{MiB}}$.

The MemPool Family: TeraPool







- **TeraPool**
  - Using MemPool to break the **TOPS** barrier
  - We need **1024 cores**, and we need them running **fast**
    - **Generalized MemPool**
      - Any number of groups, any number of tiles per group
    - **Multiple MemPool clusters?**
  - How do we program this?

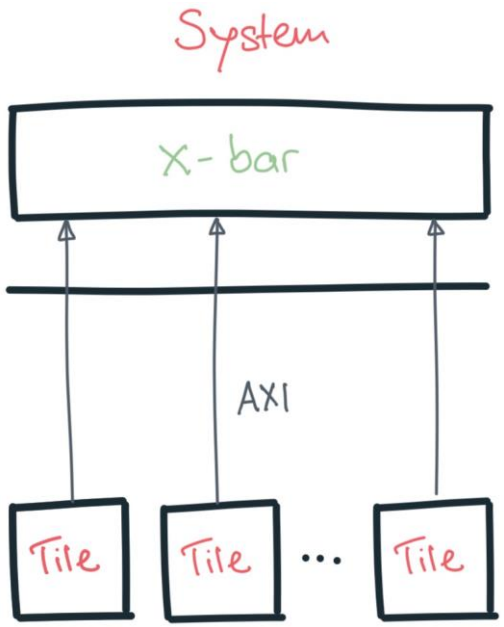
Content

1. Refresher on MemPool
2. **Completing MemPool's architecture**
   - Full instruction path
   - Improved L2 bandwidth
   - DMA
3. Adding a systolic mode

How to build the instruction path?

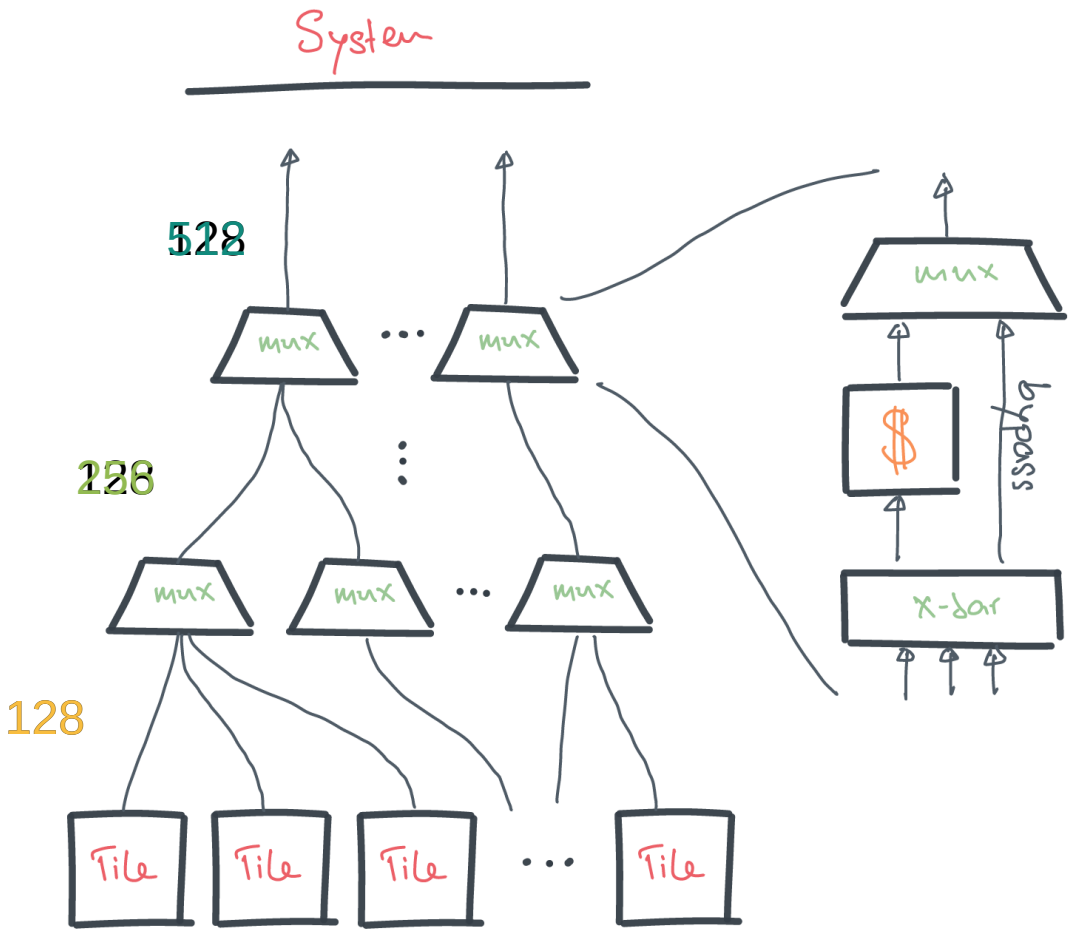
- We optimized the tile's instruction cache
- How to get to the instruction memory?
  - AXI interconnect
- Previously:
  - Single AXI bus per tile
  - 64 masters in system crossbar
  - Infeasible

→ We need hierarchy and caches



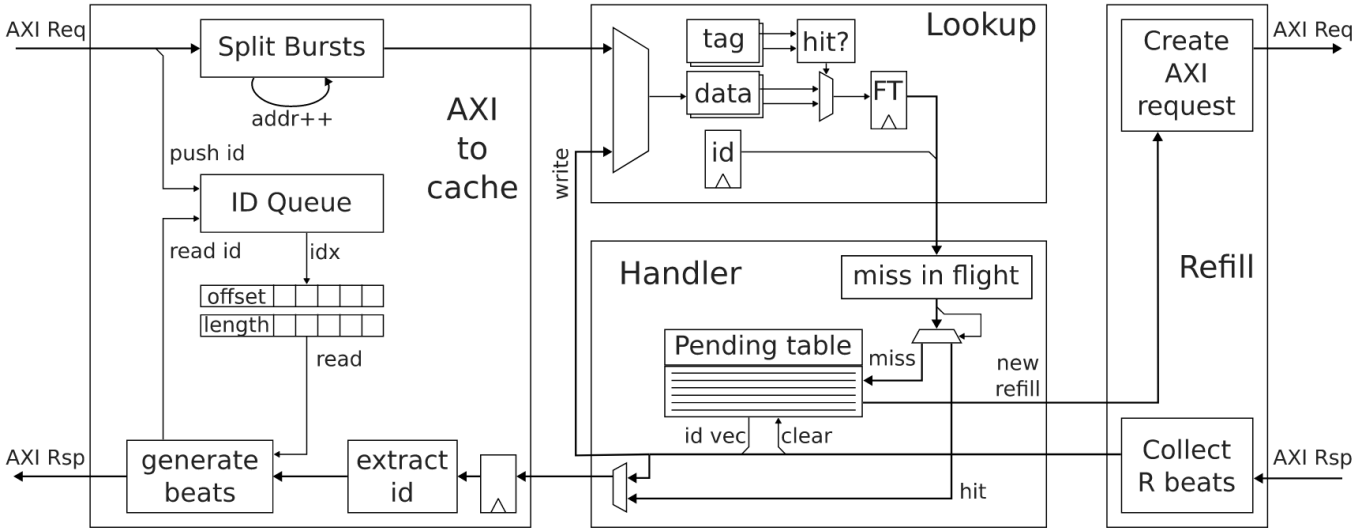
Hierarchical AXI interconnect

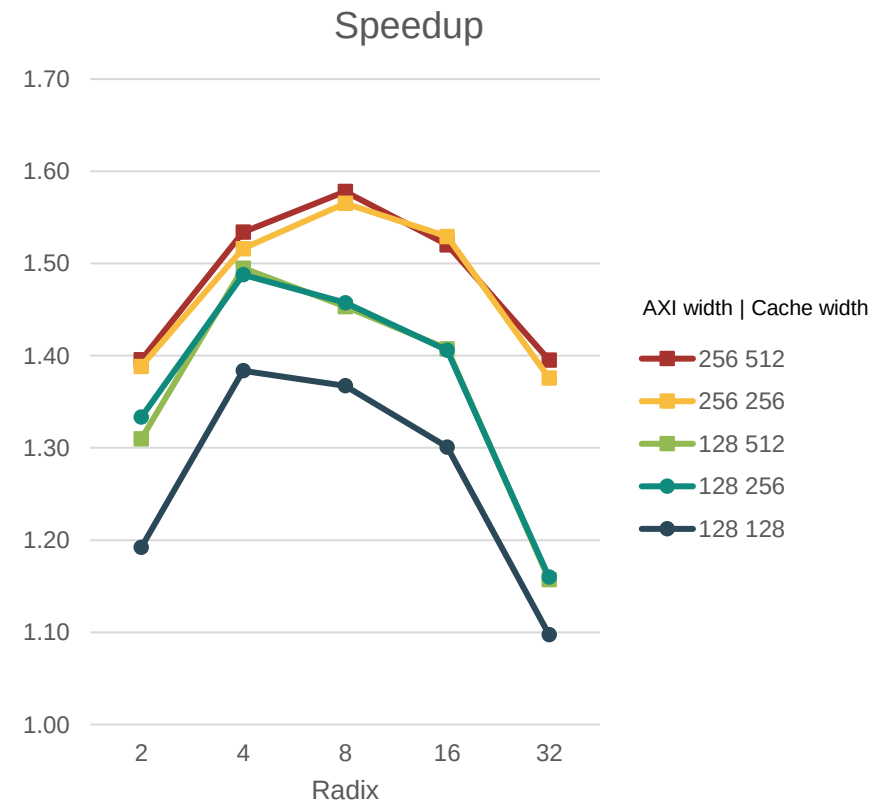
- Combine AXI buses as soon as possible in a tree-like fashion
  - Minimal routing
  - Same bandwidth as a single crossbar in the end
  - Interconnect becomes feasible
- Improve bandwidth with read-only caches
  - Configurable cache at each node in the tree
- Improve bandwidth by going wider on top
  - Not explored yet, but only requires parametrization changes



Read-only cache to keep bandwidth high

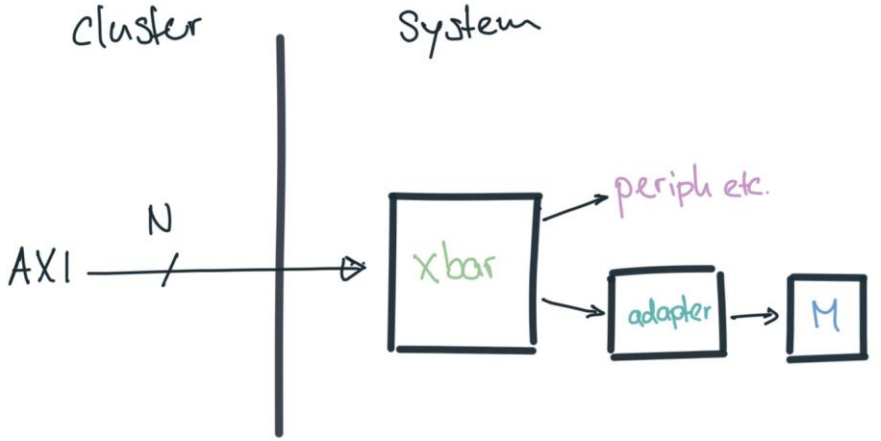
- Read-only cache to keep things simple
- Coalesces requests from multiple tiles thereby reducing bandwidth requirements higher up
- Programmable: Can also be used for data



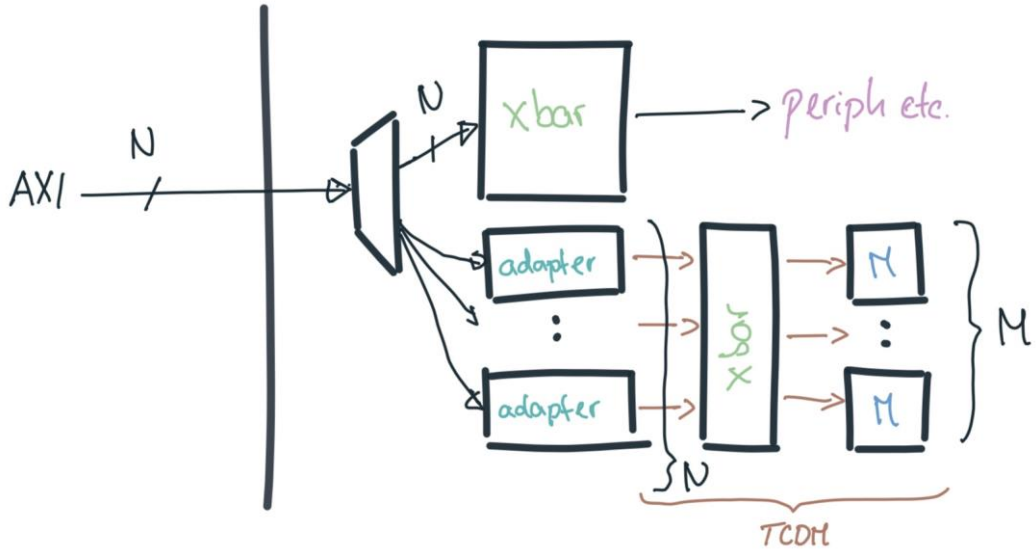


Multi-banked L2 interface

- Previously:
  - Single crossbar
  - Single L2 `adapter` (AXI to memory slave)
  - L2 bandwidth limited to AXI width
  - Bottleneck at L2 memory

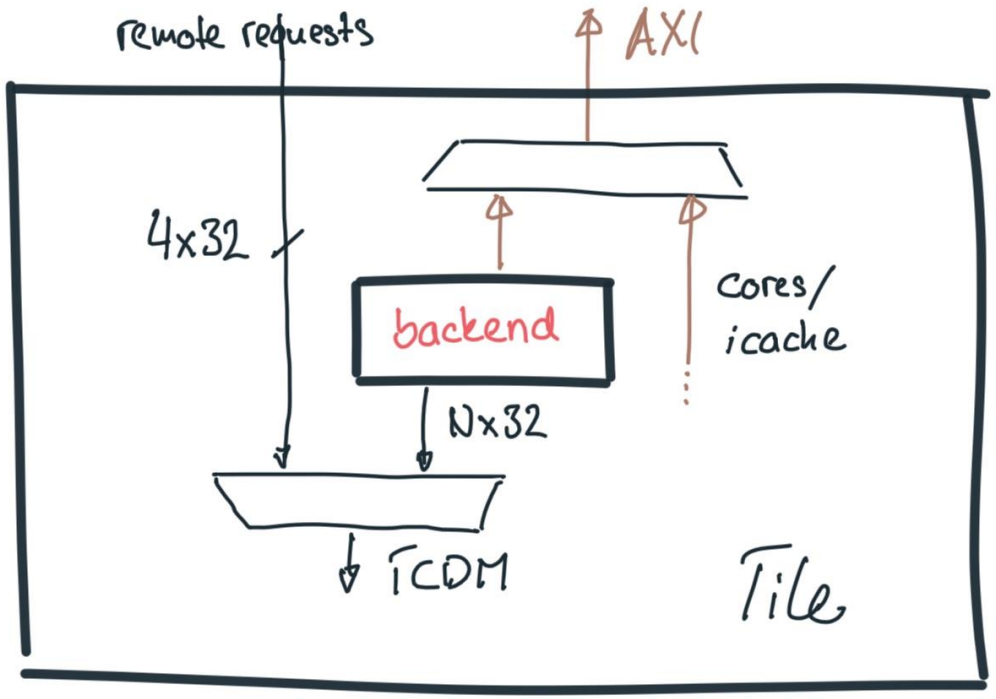


- Now:
  - Dedicated adapter for each group/AXI channel
  - Crossbar after adapter allows for interleaved memory map like TCDM
  - Easy to have parallel accesses
  - More challenging backend
  - ×1.8 speedup



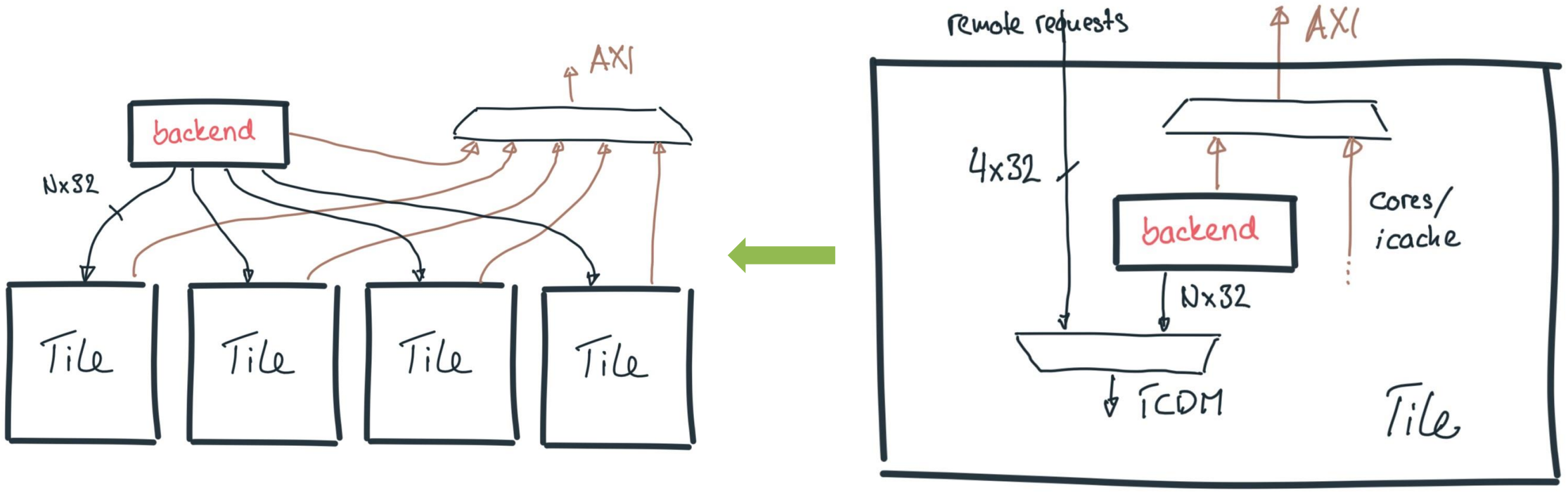
DMA: Get data in and out of the cluster

- MemPool is a single cluster: We only want one DMA
  - But: access to all tiles without adding another interconnect
- Idea:
  - Have one DMA frontend
  - Have one DMA backend **per tile**
- Each backend is responsible for its memory region
  - 64 backends will choke at the L2
  - Max TCDM ports might also be limiting at full AXI bandwidth
  - But we don’t need both interfaces to work at full speed
  - Can we reduce the number of backends?



DMA: Balance the bandwidths

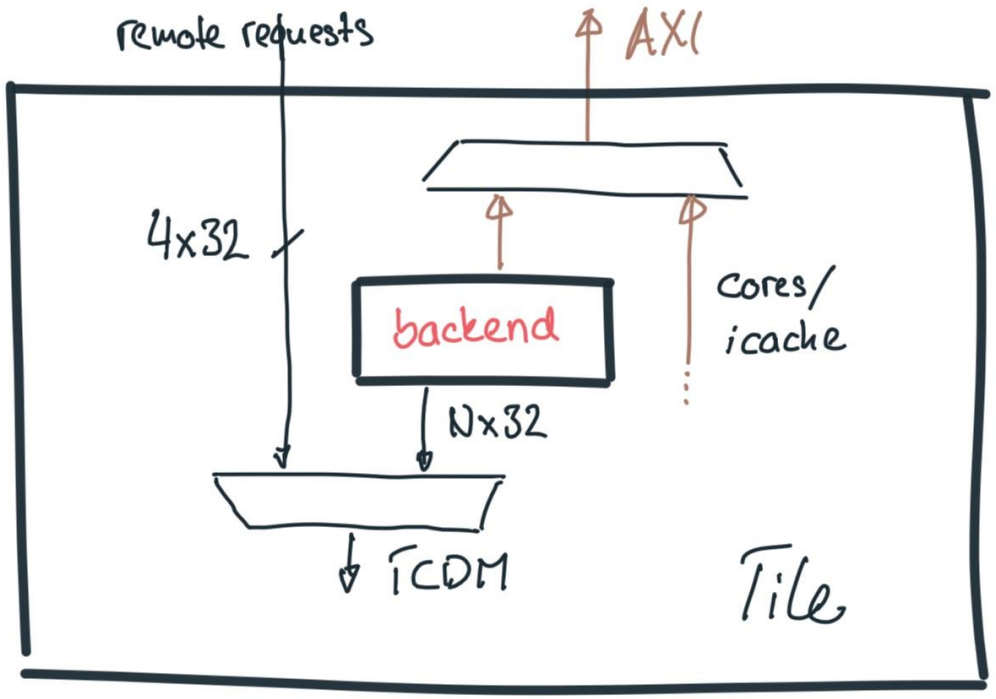
- Have one DMA backend **per N tiles**
  - Similar interfaces
  - Balance the bandwidths



DMA: Current progress

- Front-end integrated
- Back-end integrated to get first working transfer

- Next step:
  - Implement mid-end
  - Find ideal number of back-ends



Summary

- Implemented a feasible AXI interconnect
  - Hierarchical
  - With read-only caches
  - Application speedup of ×1.5
- Eliminated bottleneck in L2 memory
  - Multi-banked and interleaved
  - Memcpy benchmark improved by ×1.8
- Outline a DMA implementation
  - Front-end and back-end implemented
  - Mid-end is work-in-progress

→ The MemPool cluster is becoming ready for full benchmarking

→ We are writing a Journal about the complete MemPool design

Content

1. Refresher on MemPool
2. Completing MemPool’s architecture
3. **Adding a systolic mode**
   - Software emulation
   - Hardware extensions
   - Halide support

Can we accelerate systolic workloads?

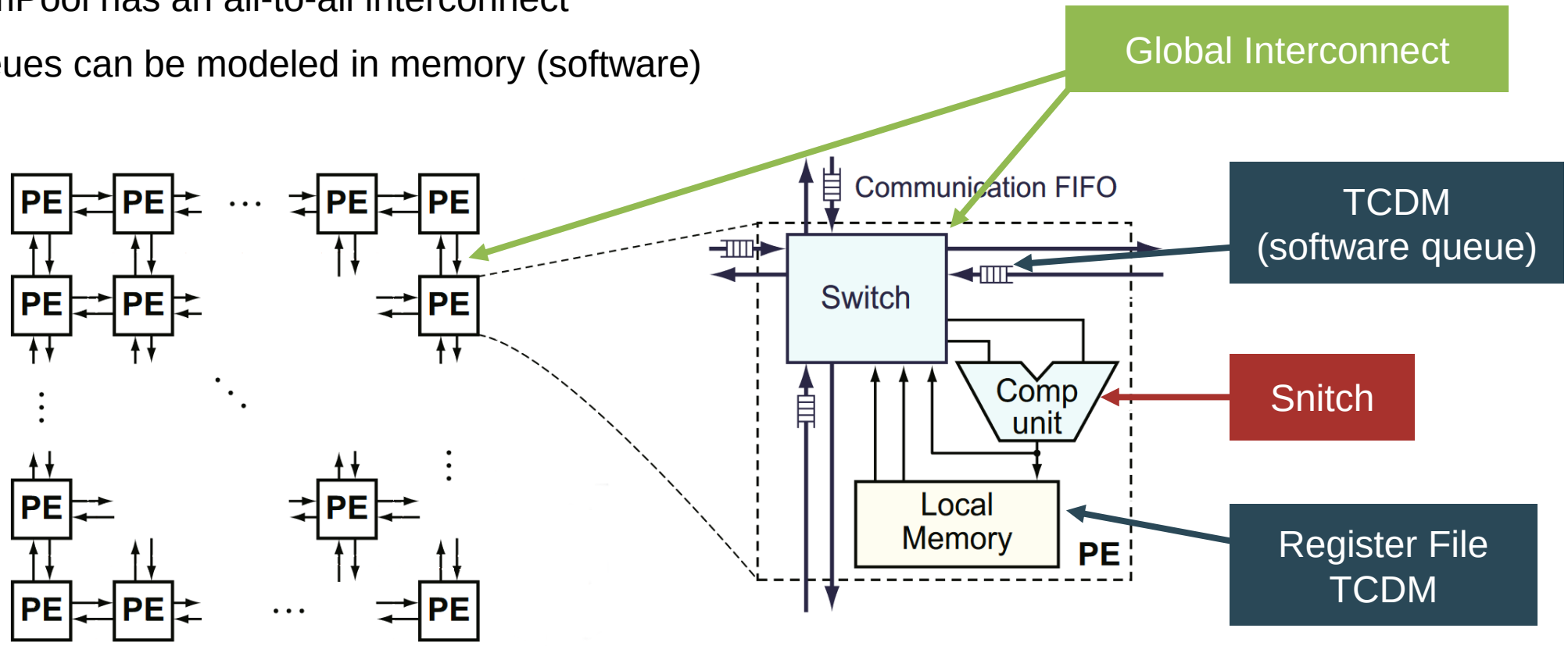
- Flexible
- Shared L1 memory
  - Long paths to traverse

- Combine the two worlds
- Remain flexible
- High efficiency for systolic workloads

- Rigid architecture
- High performance in supported systolic workloads

Emulate Systolic Workloads on MemPool

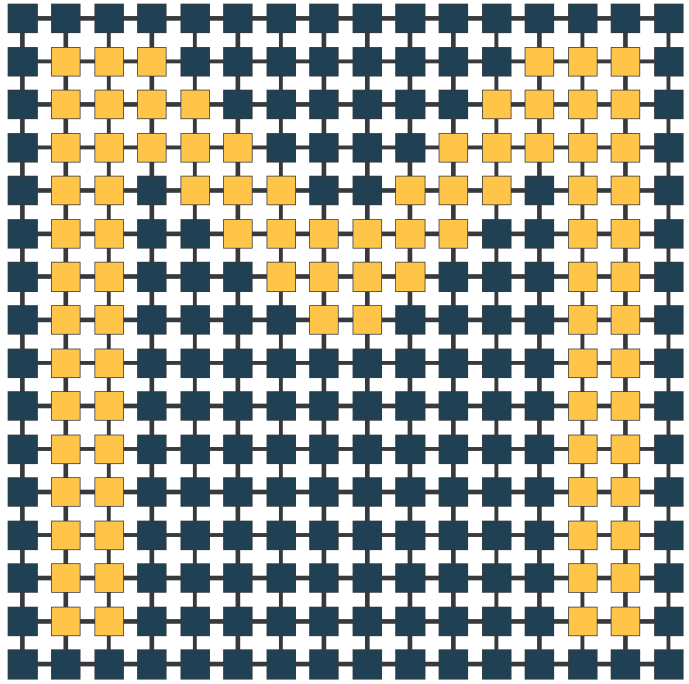
- 256 cores can be thought of as a 16×16 grid
- MemPool has an all-to-all interconnect
- Queues can be modeled in memory (software)



Source: Domain-Specific Language and Compiler for Stencil Computation on FPGA-Based Systolic Computational-Memory Array - https://link.springer.com/chapter/10.1007/978-3-642-28365-9_3

How to approach this topic?

- Start with a software emulation layer
- Add ISA extensions to accelerate message-passing
- Student project investigated the topic
  - We need to clean up the code and verify the results
- What do we currently have:
  - Started with a software emulation layer
  - Added the necessary runtime infrastructure
  - Get a feeling for systolic workloads
- Next steps:
  - ISA extensions to accelerate queue accesses
  - Get more benchmarks



Software Queues

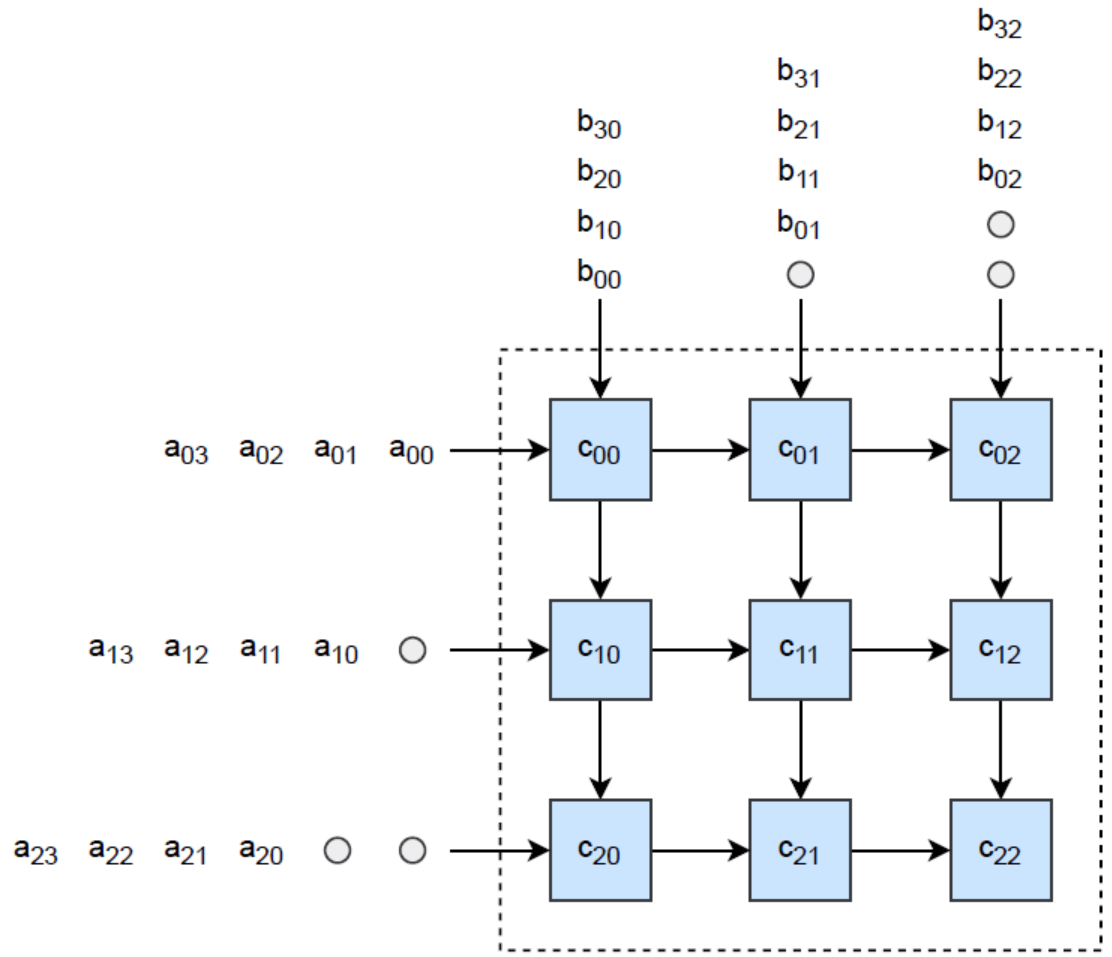
- Emulate all communication queues in software
- Allows for:
  - Arbitrary number of queues
  - Arbitrary interconnect topology
- At the cost of performance
  - Software queue push and pop take tens to hundreds of cycles
  - Synchronization issues lead to polling overhead

```
// Baseline
c = 0;
for (i=0; i<N; i++) {
  a = queue_pop(qa_in);
  b = queue_pop(qb_in);
  c += a * b;
  queue_push(a, qa_out);
  queue_push(b, qb_out);
}
```

→ Function calls take up to hundreds of cycles

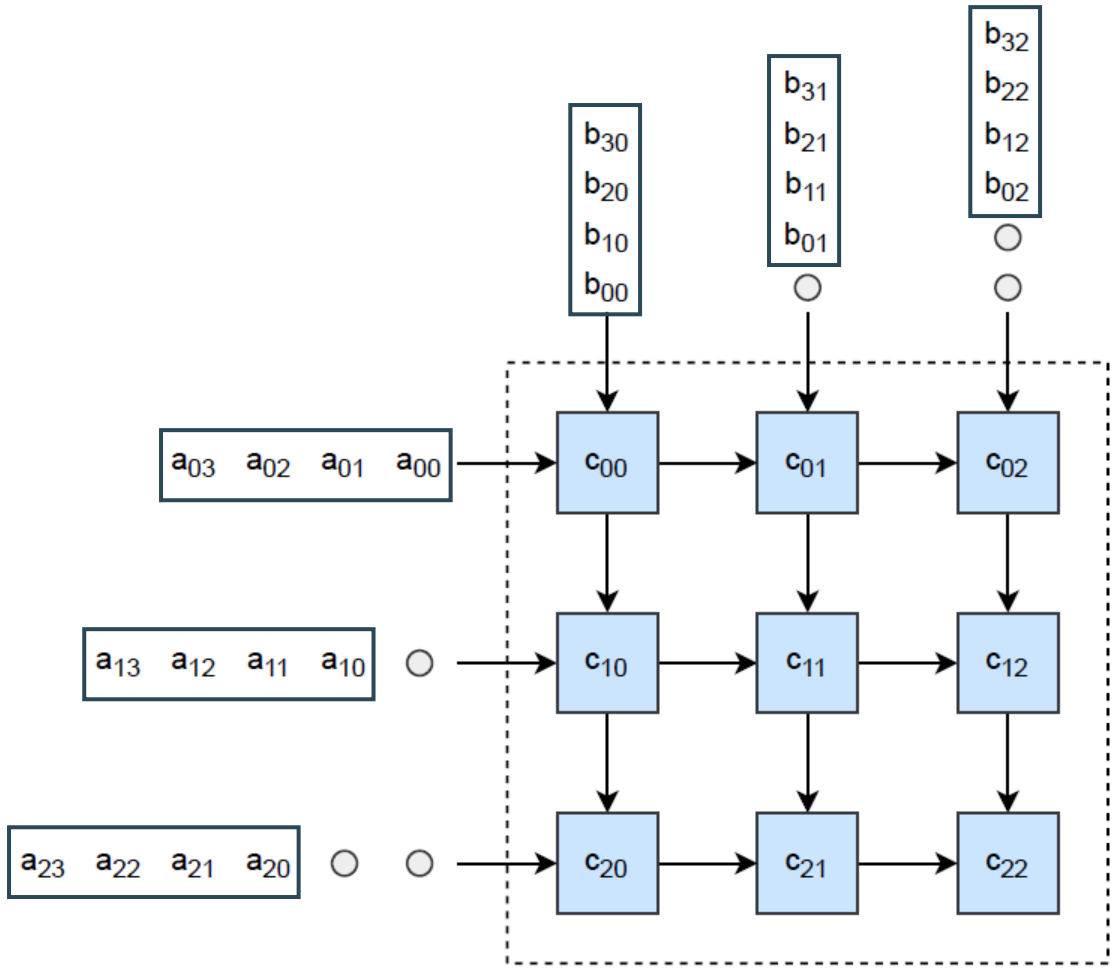
Baseline systolic matrix multiplication

- Implement allocator and software queue library
- Per element, each core does:
  - 2 pop
  - 1 MAC
  - 2 push
- More queue operations than MACs
- → Software queue could handle multiple matrix elements per queue element



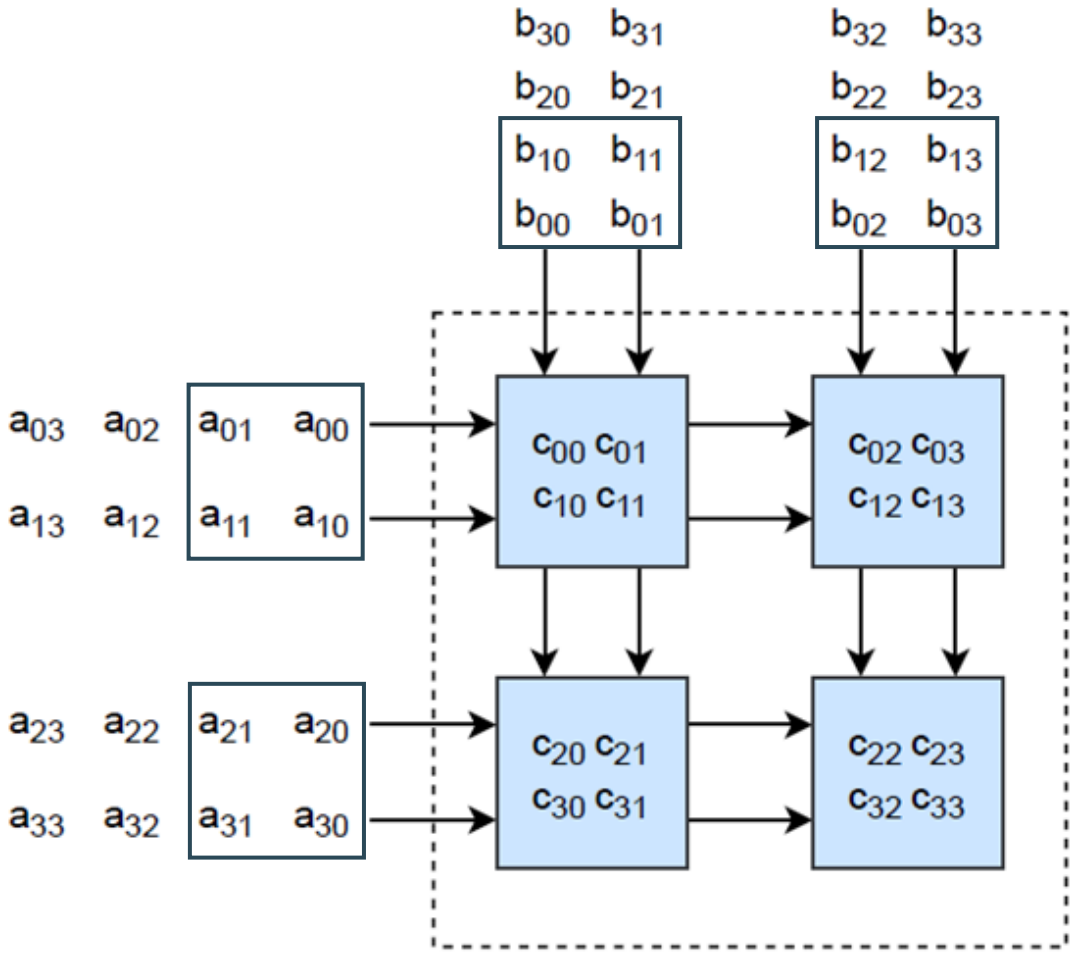
Improved systolic matrix multiplication

- Pop/push multiple elements at once
- Per four elements, each core does:
  - 2 pop
  - 4 MAC
  - 2 push
- Just as many MACs as queue operations
- ➔ 2.78 times faster than previously



Improved computational intensity

- Pop/push multiple elements at once
- Per four element, each core does:
  - 2 pop
  - 8 MAC
  - 2 push
- Half the number of queue operations
- ➔ 1.86 times faster than previously
- ➔ 5.17 overall speedup

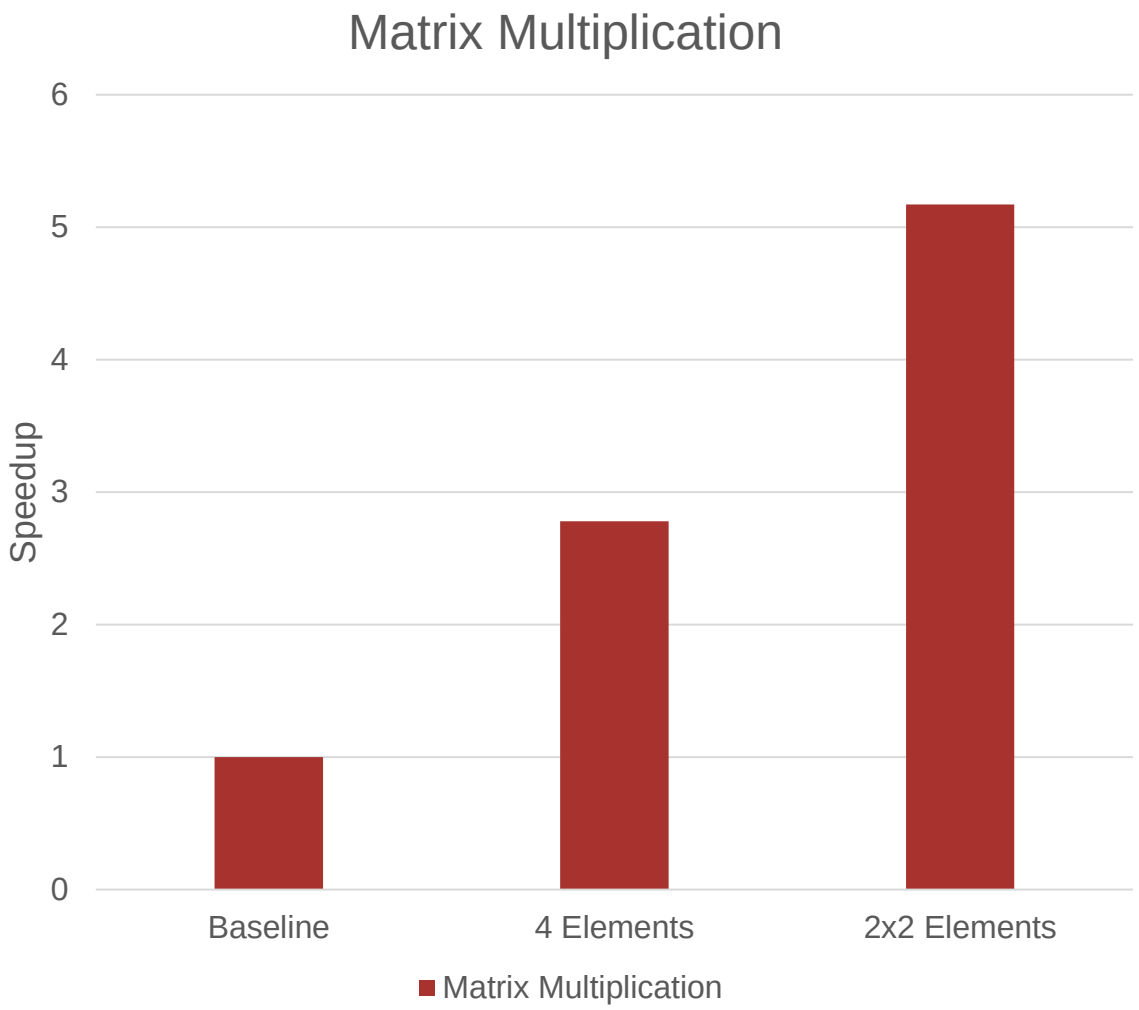


What is next?

- We need to combine systolic and conventional programming techniques to achieve optimal performance.
- The queue push/pop routines are the bottleneck

- What is next?
  - ISA extension to accelerate queue access
  - Add 2D convolution benchmark

- Additional Ph.D. student will help us out until the Journal is finished



Content

1. Refresher on MemPool
2. Completing MemPool’s architecture
3. Adding a systolic mode
   - Software emulation
   - **Hardware extensions**
   - **Halide support**

ISA Extension: Queue pop and push

- Reduce the complex pop and push functions to a single instructions
- Keep the benefits of queues in the TCDM

- Similar implementation as atomics
- Extension in Snitch and memory controller
  - Needs checking and cleanup

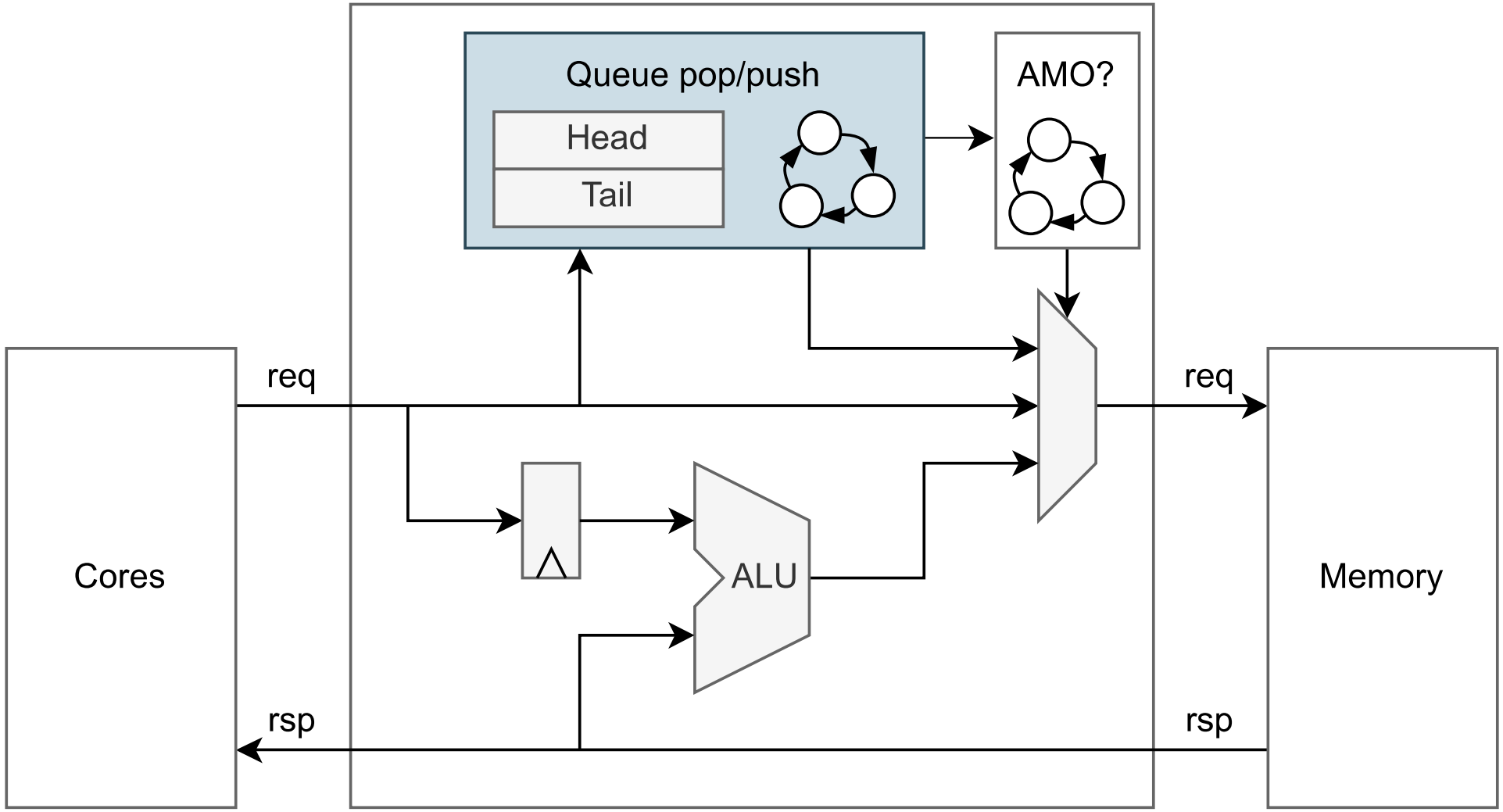
Eliminate tens of instructions

```
// Baseline
c = 0;
for (i=0; i<N; i++) {
  a = queue_pop(qa_in);
  b = queue_pop(qb_in);
  c += a * b;
  queue_push(a, qa_out);
  queue_push(b, qb_out);
}
```

```
// +queue pop/push extension
c = 0;
for (i=0; i<N; i++) {
  a = __builtin_pop(qa_in);
  b = __builtin_pop(qb_in);
  c += a * b;
  __builtin_push(a, qa_out);
  __builtin_push(b, qb_out);
}
```

Memory latency → RAW hazard

Queue pop and push in memory controller



Automatically push and pop

- Eliminate the explicit push/pop instructions
  - SSR like behavior
  - Do communication in parallel
- Extension to Snitch

- Allows for latency hiding
- Snitch focuses on computation only

Eliminate explicit communication

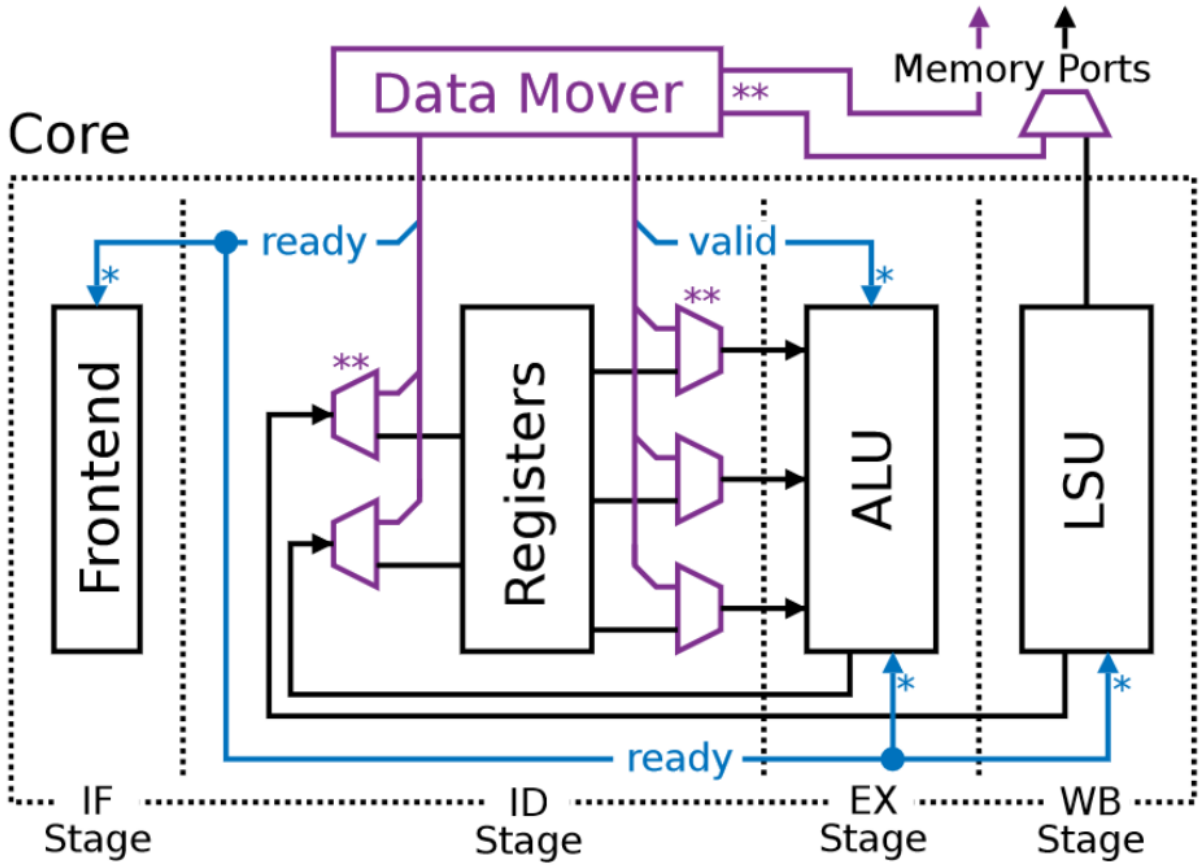
```
// Baseline
c = 0;
for (i=0; i<N; i++) {
  a = queue_pop(qa_in);
  b = queue_pop(qb_in);
  c += a * b;
  queue_push(a, qa_out);
  queue_push(b, qb_out);
}
```

```
// +queue pop/push extension
c = 0;
for (i=0; i<N; i++) {
  a = __builtin_pop(qa_in);
  b = __builtin_pop(qb_in);
  c += a * b;
  __builtin_push(a, qa_out);
  __builtin_push(b, qb_out);
}
```

```
// +SSR like extension
c = 0;
setup_ssr(a, qa);
setup_ssr(b, qb);
for (i=0; i<N; i++) {
  c += a * b;
}
```

SSR extension

- ‘Data Mover’ can be configured to read/write data streams
- Registers are refilled automatically
- Data mover performs queue pop/push
- Could increase memory ports

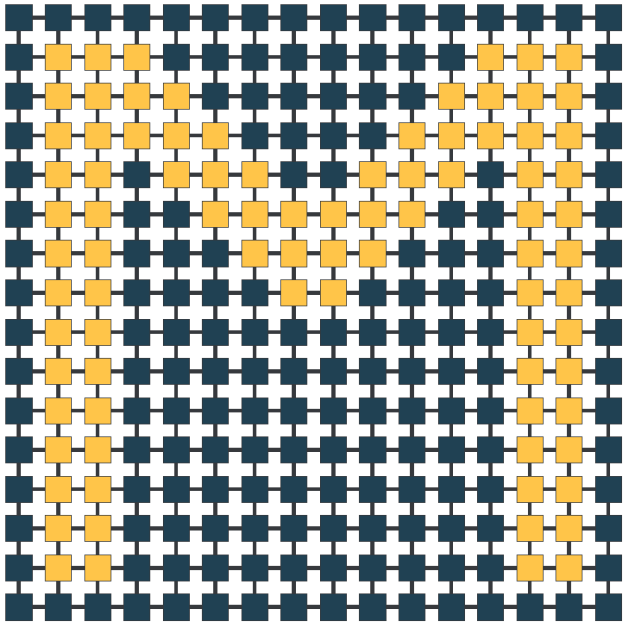


Source: SCHUIKI et al.: STREAM SEMANTIC REGISTERS - http://htor.inf.ethz.ch/publications/img/schuiki-ssr.pdf

Summary

- Implemented a software layer for systolic applications
  - Get an understanding of systolic execution on MemPool
  - Benchmarked and optimized matrix multiplication

- Next steps and milestones
  - Implement and evaluate queue push and pop atomics
  - Implement and evaluate SSR-like extension
  - Extend MemPool's Halide runtime for systolic execution
  - Final evaluation of finalized hardware and software extensions



Deliverables

- Release hardware and software code on GitHub
  - https://github.com/pulp-platform/mempool
  - RTL code of ISA extensions integrated with MemPool
  - Systolic runtimes (barebone C runtime and Halide)
  - Benchmark code

- Publications
  - *MemPool Journal*: Summarizing and evaluating the full MemPool architecture (no systolic yet)
    → In progress
  - *MemPool meets Systolic:* How to build an efficient hybrid architecture with a systolic layer
    → Not concrete yet