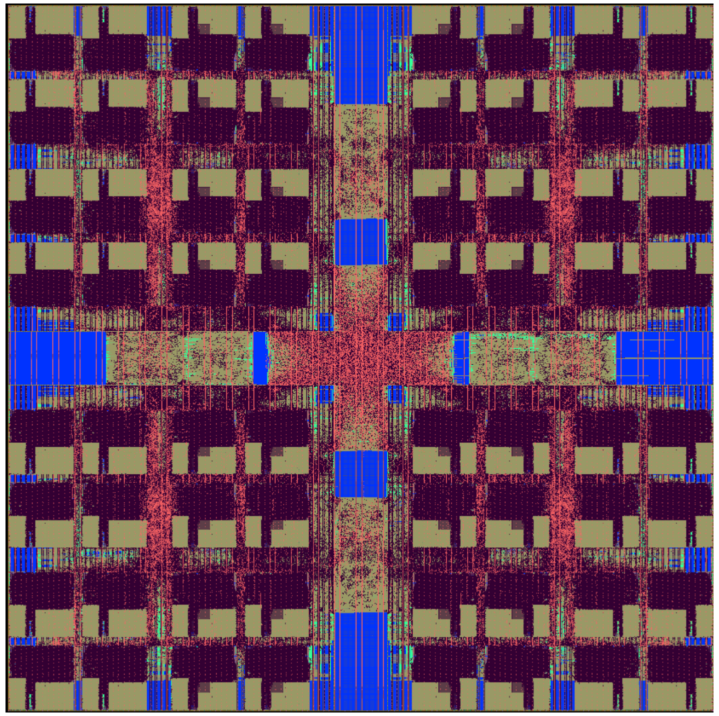
# The road to TeraPool

**Matheus Cavalcante**
**Samuel Riedel**
**Prof. Luca Benini**
Zürich, 27 September 2021

# MemPool: a challenging design to place-and-route

- MemPool is a challenge for the PnR tools
  - **Large design**: 256 cores in a 4.6 mm x 4.6 mm core area
  - **Highly-connected**: the cores communicate through fully-connected crossbars
  - **Low latency**: all cores can access all the shared L1 memory into at most 5 cycles of zero-load latency
  - **High frequency**: In 22FDX, the design can be clocked up to ~700 MHz in typical conditions

*The design is limited by the wire propagation delay. MemPool's large footprint limits the operation frequency.*



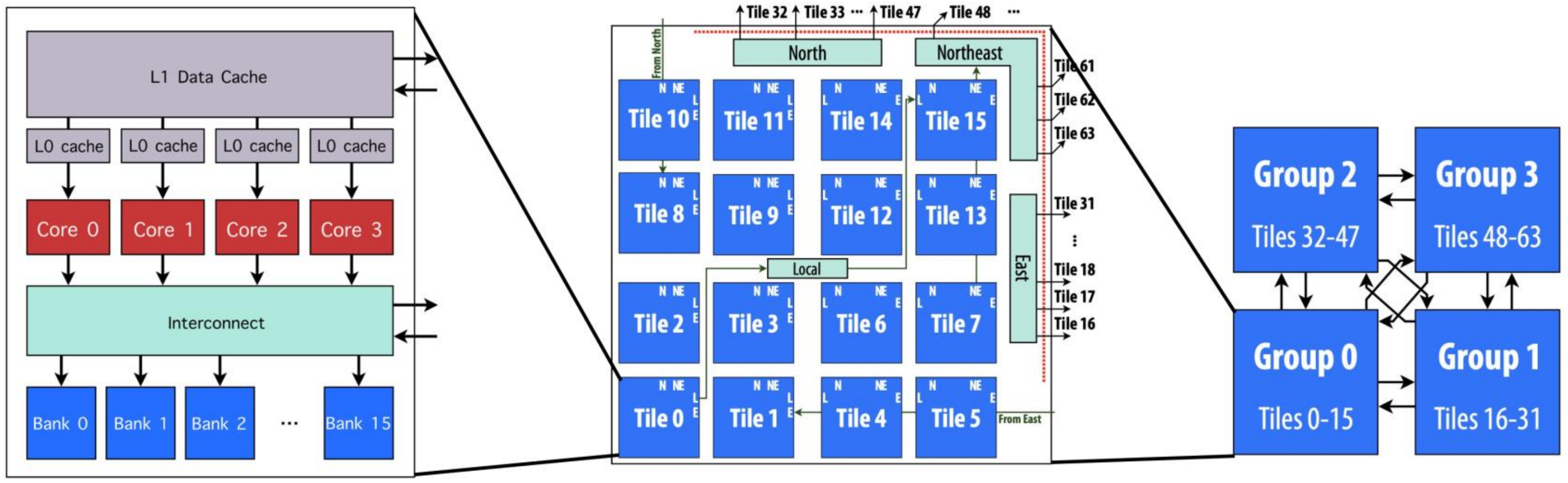
# TeraPool: scaling MemPool to the teraops region

- **TeraPool**: Using MemPool to break the **teraops** barrier
- We need:
  - 1024 cores, *four times* the current core count
  - Support for fused multiply-accumulate instructions
    - 2 OP/cycle
  - A high operating frequency, of *at least 500 MHz*
- We also want:
  - To keep MemPool’s low latency and high-connectivity characteristics
  - To keep a shared view of memory

## TeraPool: some implementation challenges

- A small footprint is **essential**
  - Worst-case scenario:
    - *Four times the core count* → Four times the core area → Twice as long MemPool diagonal length → Twice as long wire delay → *Half of the operating frequency*
    - 3D Integration for footprint and interconnect length reduction?
- We also have a very large design to implement
  - MemPool already has runtime issues with the EDA tools (particularly the group)
  - Our bottom-up implementation flow would need a lot of tweaking at the interfaces' constraints

# Quick refresher on MemPool’s hierarchy



MemPool Tile

Group

Cluster

## TeraPool’s microarchitecture

- MemPool’s hierarchy is a basis for how TeraPool should look like
- How to reach 1024 cores?
  - **Approach #1**: tiles with 16 cores, everything else the same – 16C16T4G
    - Huge 16 x 64 crossbar at each tile!
    - Traffic bottleneck at the tile boundary
    - Each “TeraPool group” now has as many cores as the whole MemPool cluster
  - **Approach #2**: tiles with 8 cores, groups with 16 tiles (128 cores), cluster with 8 groups (1024 cores) – 8C16T8G
    - How do the interconnection between the eight groups look like?
    - Some early approaches led to a lot of interconnection

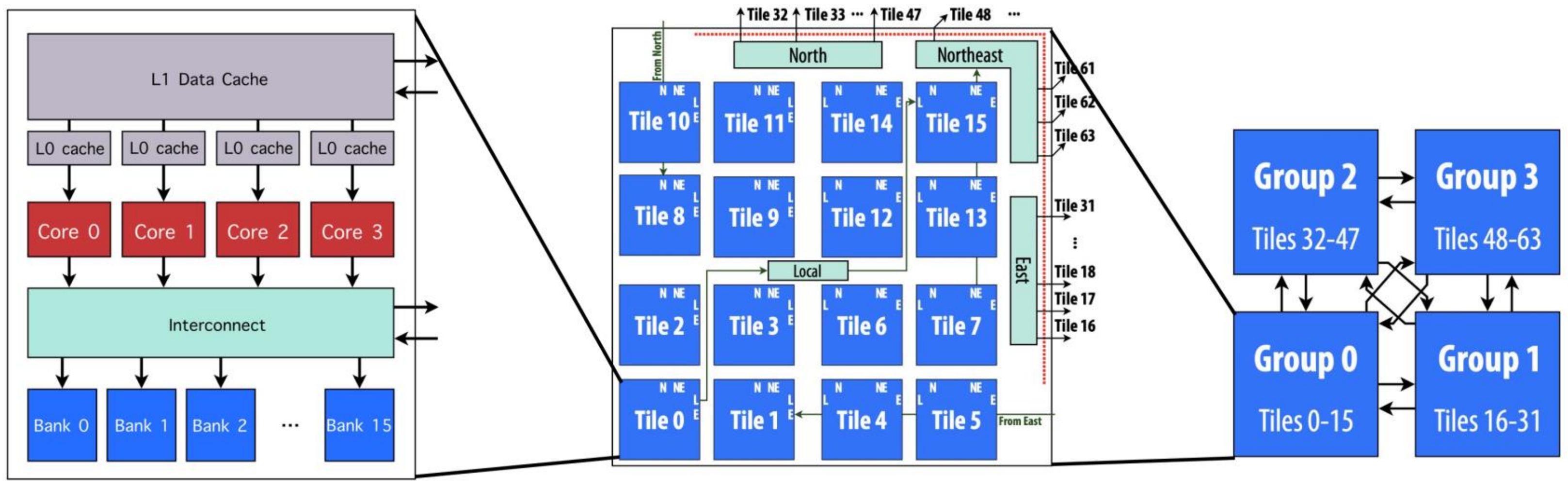
ETH zürich

# MemPool meets Systolic

Samuel Riedel
Matheus Cavalcante
Prof. Luca Benini

## Quick refresher on MemPool’s hierarchy

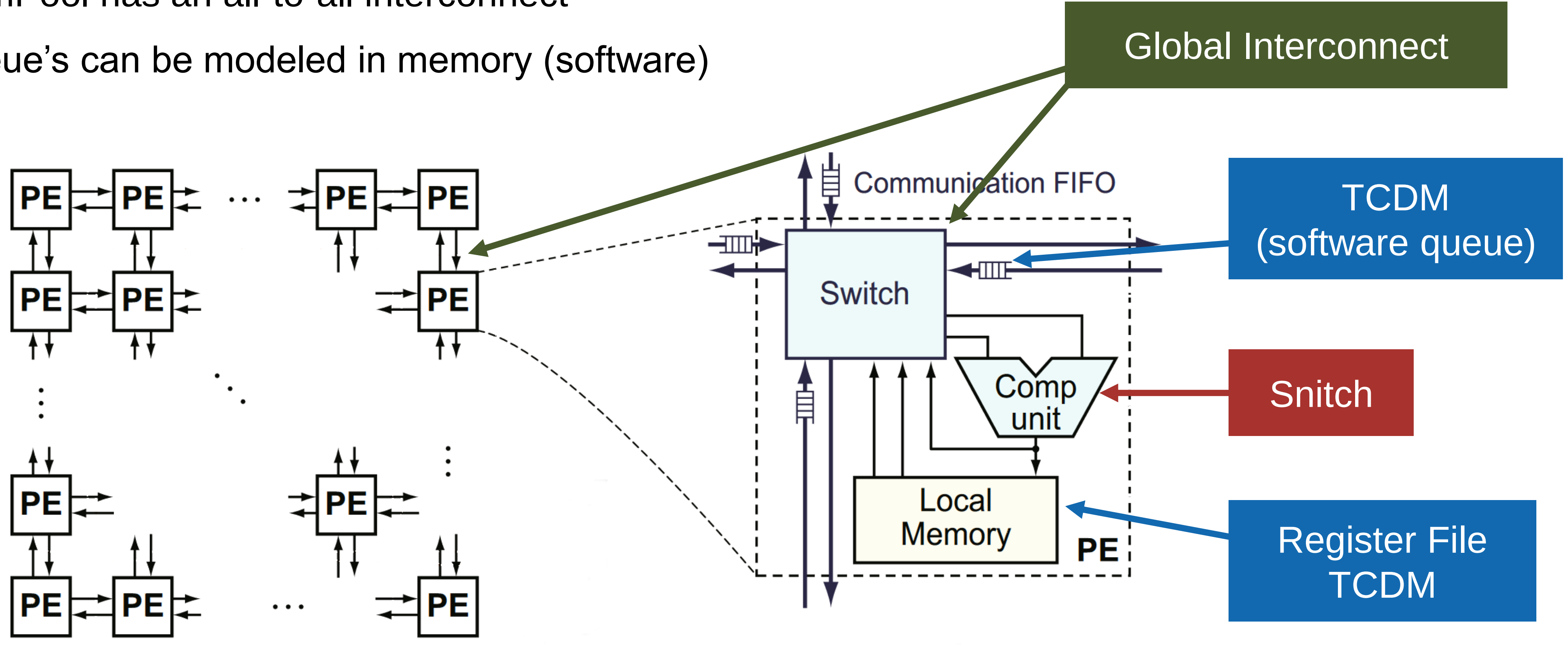
- Tile:
  - 4 32-bit cores
  - 16 banks
  - Single cycle memory access
- Group:
  - 64 cores
  - 256 banks
  - 3 cycles latency
- Cluster
  - 256 cores
  - 1 MiB of memory (1024 banks)
  - 5 cycles of latency



MemPool Tile Group Cluster

# Emulate Systolic Workloads on MemPool

- 256 cores can be thought of as a 16×16 grid
- MemPool has an all-to-all interconnect
- Queue’s 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*

## 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);
}
```

$\longrightarrow$ Function calls take up to hundreds of cycles

# 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

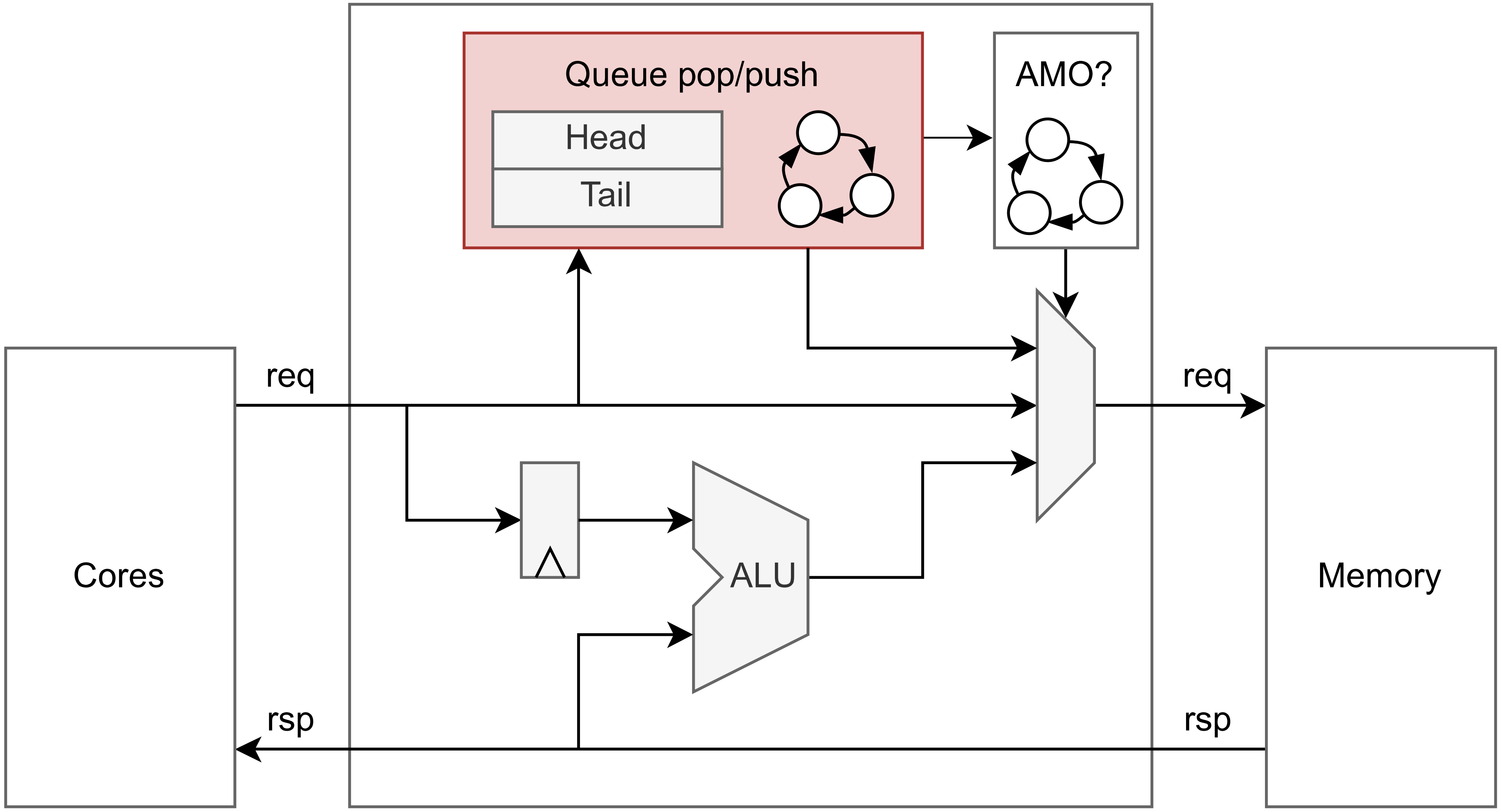
Eliminate tens of instructions

```
// 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

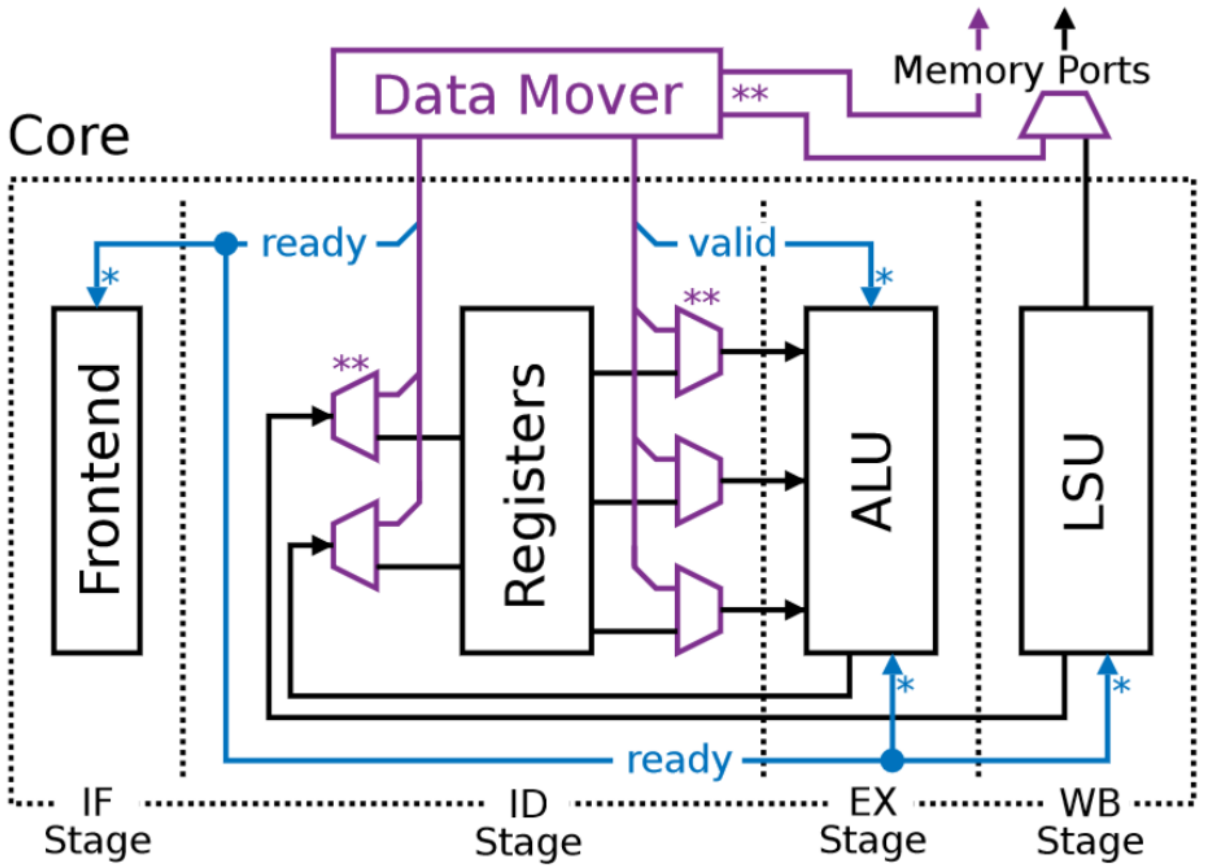
```
// Baseline
c = 0;
for (i=0; i<N; i++) {
  a = queue_pop(qa_in);
  b = queue_pop(qb_in);
  c += a * b;
  queue_push(a, qa_out);
  queue_push(b, qb_out);
}
```

```
// +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

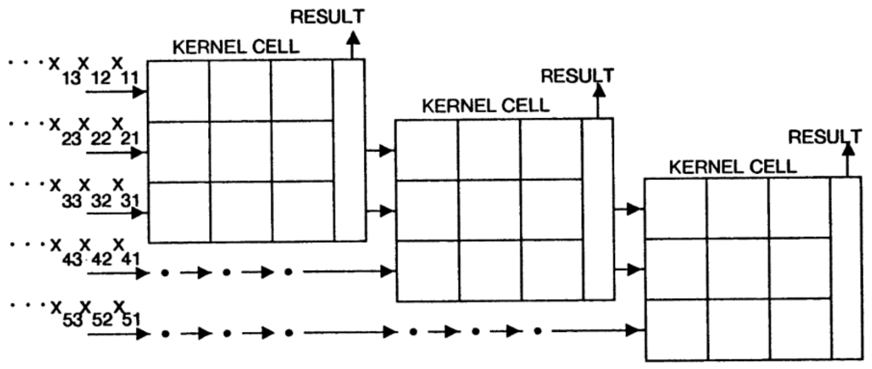
- ‘Data Mover’ can be configured to read/write data streams
- Registers are refilled automatically
- Data mover performs queue pop/push
- Could increase memory ports



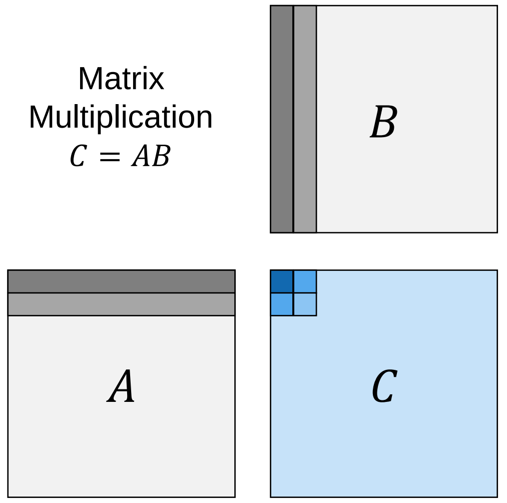
*Source: SCHUIKI et al.: STREAM SEMANTIC REGISTERS - http://htor.inf.ethz.ch/publications/img/schuiki-ssr.pdf*

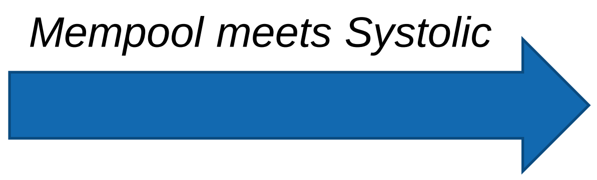
# Benchmarking

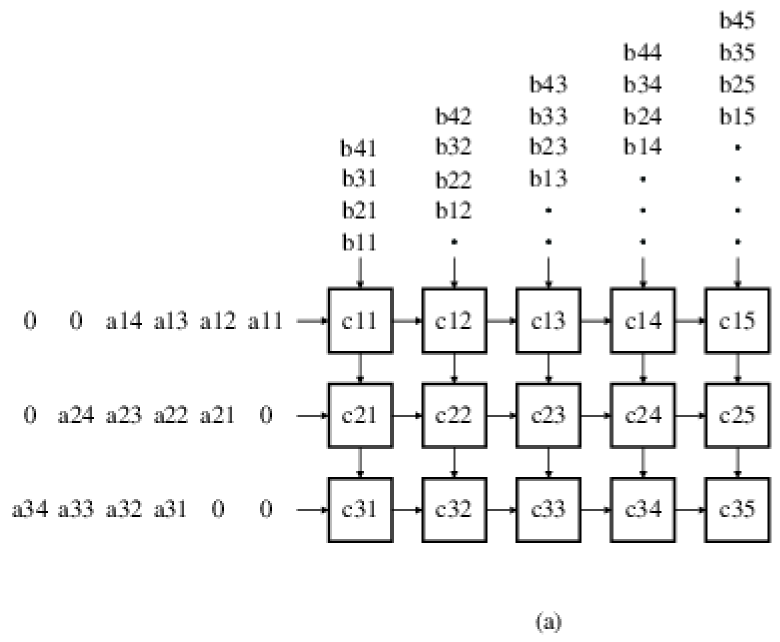
- Implement kernels at each step
  - Matrix Multiplication
  - 2D Convolution
- Evaluate performance gain of each ISA extension
- Compare to data-parallel implementation



Source: H. T. Kung and S. W. Song “A Systolic 2-D Convolution Chip”







Source: https://gyires.inf.unideb.hu/KMITT/a52/ch04.html