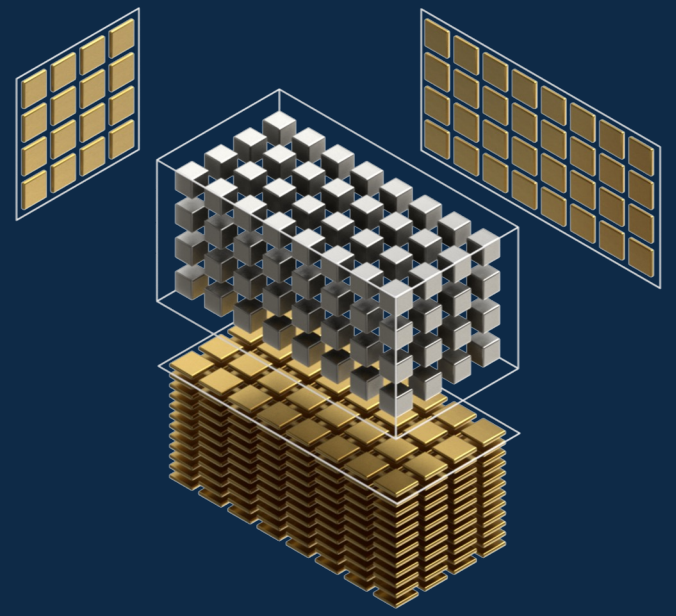
OSCAR 2026, Raleigh, NC

RTL Design and Analysis of Sparse Tensor Cores for Unstructured Sparsity in RISC-V GPUs

A synthesizable, dual-side unstructured sparse Tensor Core for the Vortex GPGPU



Authors

Giorgos Triantafyllou*, Panagiotis-Eleftherios Eleftherakis*, Konstantinos Iliakis*, Blaise Tine§ and Sotirios Xydis*

*National Technical University of Athens
§University of California, Los Angeles

June 2026

Outline

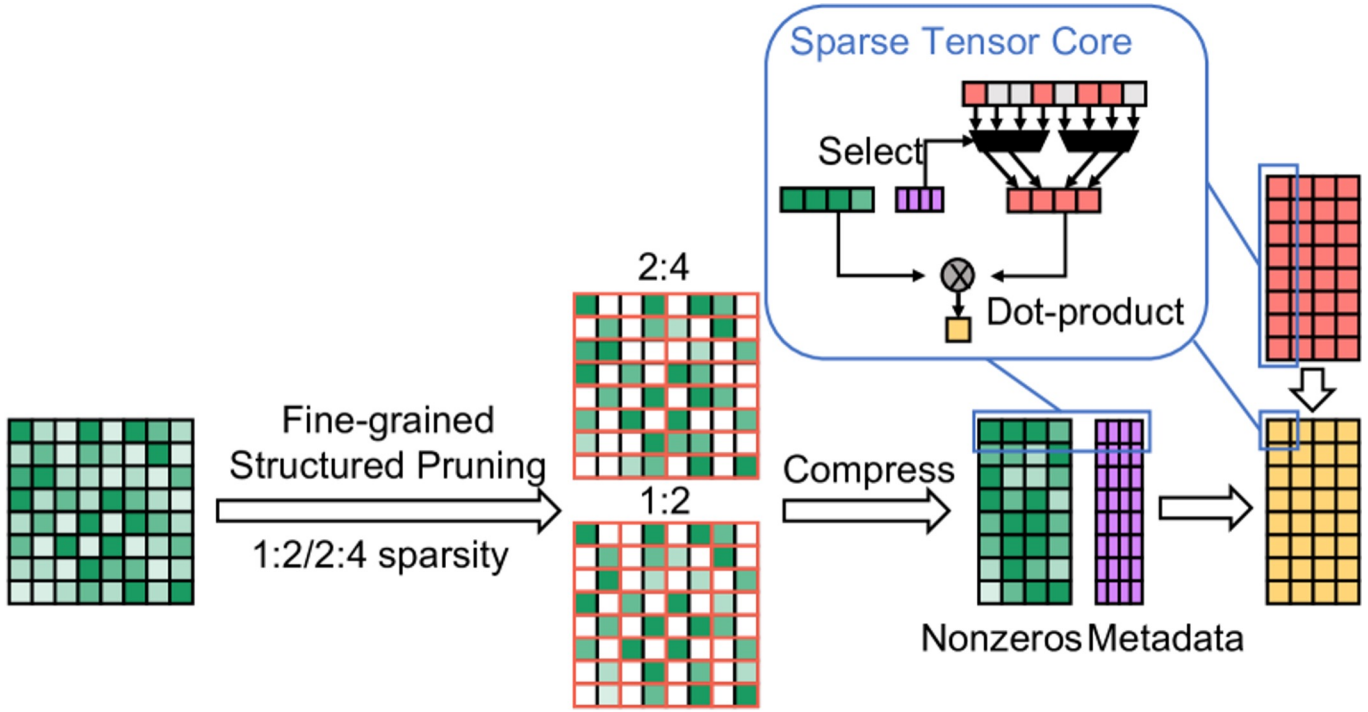
Matrix Multiplication in Modern AI

- AI / ML / HPC workloads are growing explosively and are dominated by **General Matrix-Matrix Multiplication (GEMM)**, expressed as **D = A × B + C**:
  - **MLPs**: activations × weight matrices
  - **CNNs**: convolutions lowered to GEMM via im2col
  - **Transformers**: Q / K / V projections, attention scores $(QK^\top)$, value aggregation $(\mathrm{softmax}(QK^\top)V)$, and MLP blocks

- GPUs accelerate GEMM with **Tensor Cores** (since NVIDIA Volta) replacing scalar/vector instructions
- In practice, real performance ≠ peak FLOPS. Tensor Cores are limited by **data movement**, **RF pressure** and **instruction overhead**
- Therefore, improving GEMM requires optimizing both computation and operand delivery

Leveraging Sparsity and Limitations of Existing Hardware Support

- **Leveraging Sparsity** can reduce both computation and memory traffic.
- Existing Tensor Core support targets **structured 2:4 sparsity**, on **one operand only**
  - fixed pattern, ≤ 2× theoretical gain, possible **accuracy loss** from pruning
- **Research gap:** no RTL (synthesizable) Tensor Core that accelerates GEMM where **both** operands have **unstructured** sparsity
- **Opportunity:** Dual-side, unstructured sparsity in RTL.



NVIDIA 2:4 Structured Sparse Tensor Core

Proposed design: TCU-OP - Contributions

An RTL outer-product TCU - contributions w.r.t. SOTA:

Sparse TCU in Synthesizable RTL

Implementation in synthesizable RTL allowing PPA post-synthesis evaluation

Hardware optimizations

Hardware step (sub-tile) skipping,
Adders after the crossbar

Direct shared-memory access

operands read straight from SHMEM
- register-file bypass

Three execution modes

including uncompressed × compressed for dynamic activations

Asynchronous execution

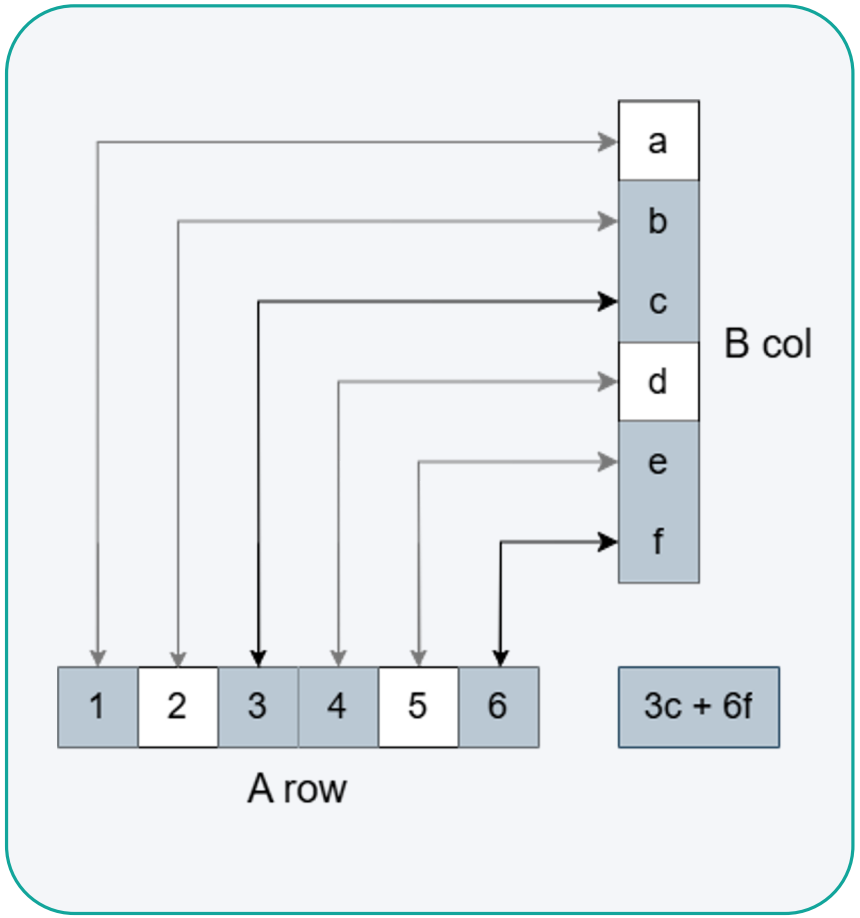
transactional barriers -
warp keeps working

HW/SW co-design

new ISA instruction +
SHMEM double buffering kernel design

Tensor Cores & the inner-product model

- A **Tensor Core** is a hardware engine for tiled multiply-accumulate: **D = A × B + C**
- Conventional Tensor Cores use the **inner-product** formulation:
  - implemented with Fused-Element **Dot-Product (FEDP)** units
- Efficient for **dense** GEMM
- **Problem for dual-side sparsity:** the hardware must dynamically **match non-zero pairs**
- This motivates a **different computation primitive** for sparse execution

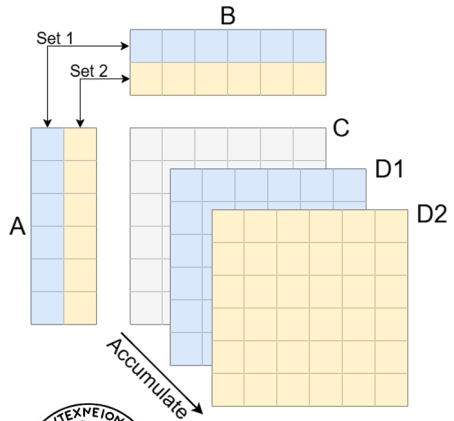


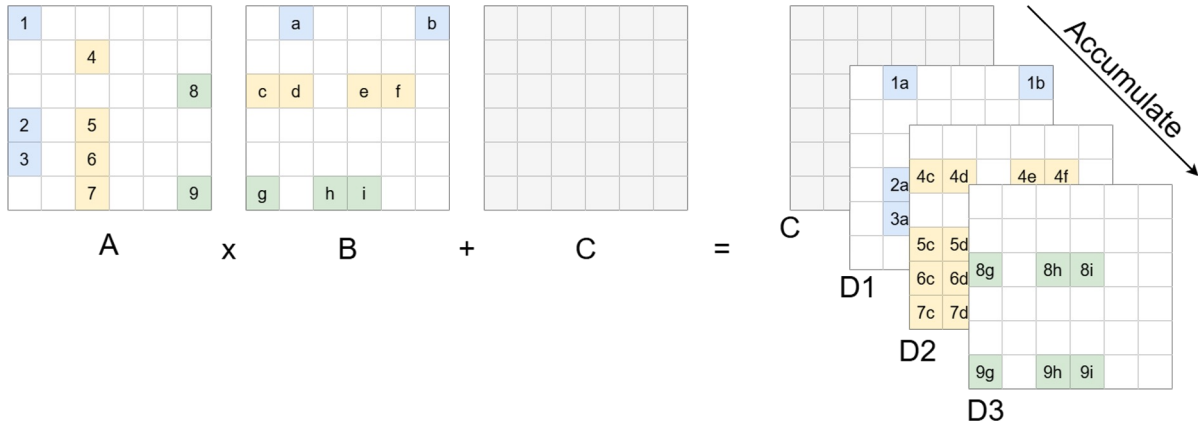
Inner-product computation

The outer-product reformulation

- **Outer-product:** each step multiplies one **column of A** by one **row of B** → a rank-1 partial output tile
- Accumulate partial tiles across the **K dimension** → final output
- High **input reuse:** each nonzero of A multiplies every nonzero of B
  Low **output reuse:** the output tile is updated for every new partial sum → alleviated by an accumulation buffer.

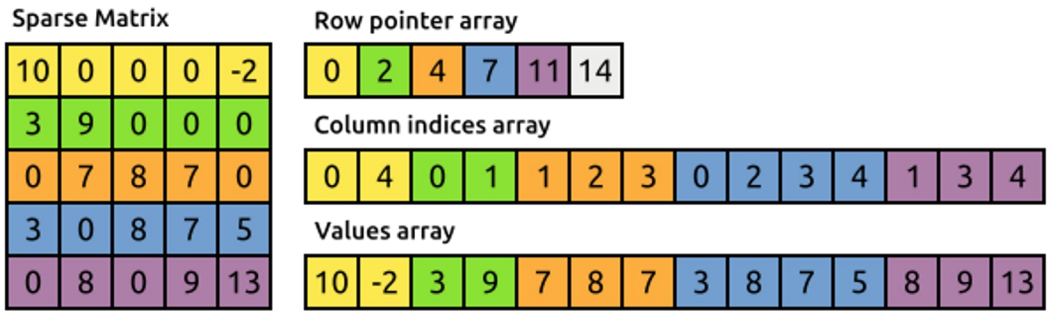
- Why it fits sparsity:
  - **compact** the non-zeros into short dense vectors, then **skip zero-only steps**
  - matching is replaced by a regular **cross-product** of remaining non-zeros





Bitmap-encoded sparsity

- Each sparse tile is stored as a **non-zero value array + a bitmap**:
  - bit = 1 → nonzero is present in the value array
  - bit = 0 → an omitted zero
- Layout matches the outer product: **A column-major, B row-major**
- **Bitmap**: low metadata overhead and easy recovery of original (dense) coordinates
- **CSR**: stores values + column indices + row pointers; double metadata accesses and greater memory footprint than bitmap for DNN sparsity



compression → bitmap + non-zero values

dense vector

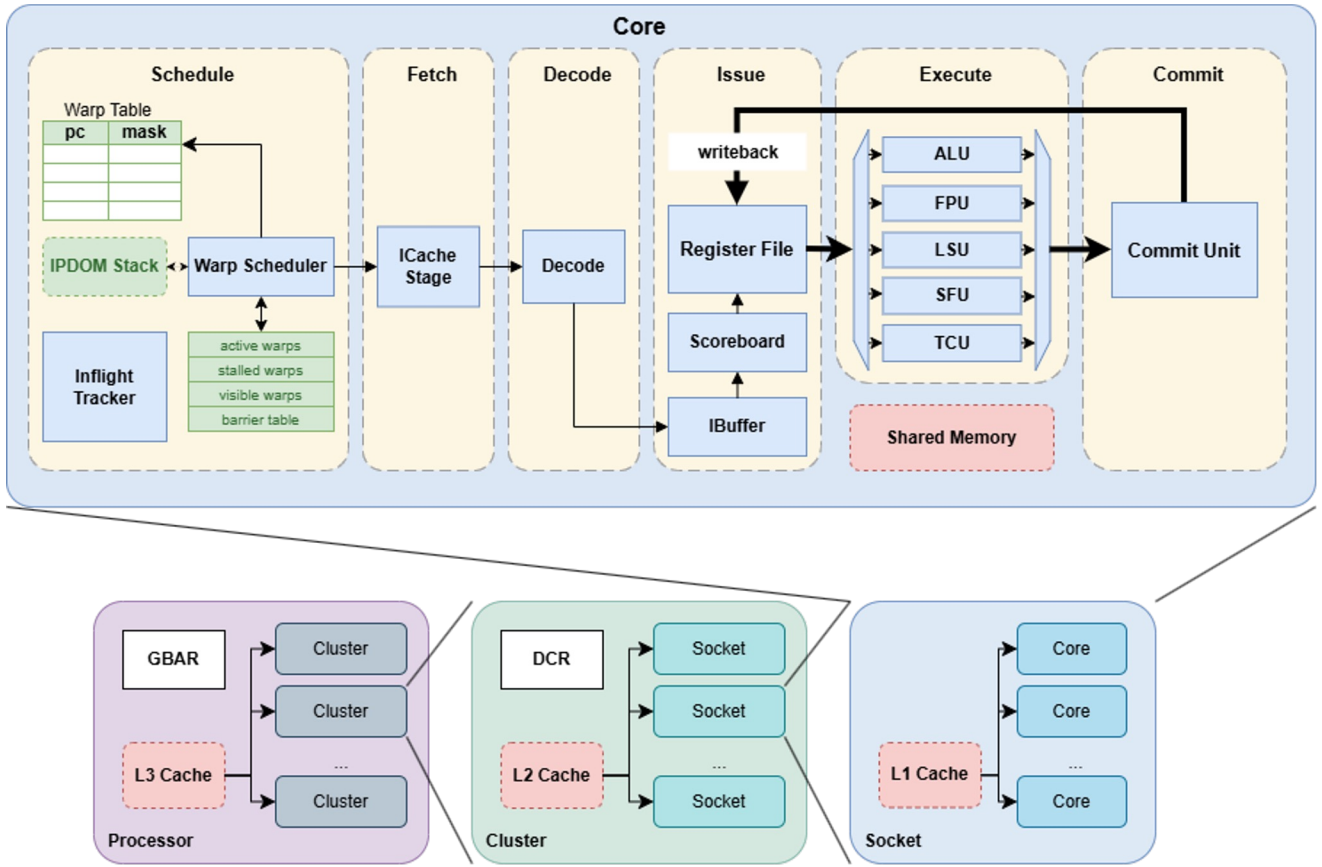
bitmap

compacted values

4 non-zeros stored instead of 8 values

The Vortex RISC-V GPU & its Tensor Core

- **Vortex** — open-source, synthesizable RISC-V GPGPU in Verilog
- Enables **cycle-accurate RTL** simulation exposing parameters hidden by high-level simulators
- Ships an **NVIDIA-style, warp-level Tensor Core**, plus a 2:4 structured-sparse mode
- Baseline dataflow:
  GMEM → RF → TCU → RF → GMEM
- Equipped with a **DMA** mechanism for asynchronous global → shared-memory transfers

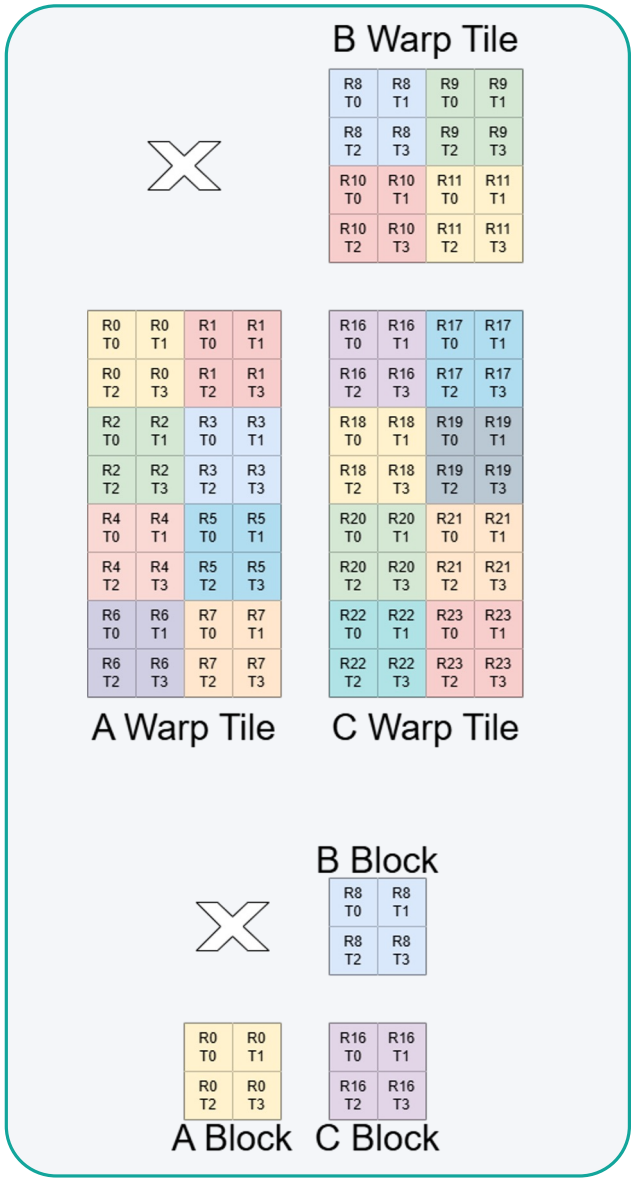


Current Limitations of Vortex Tensor Core

- Each warp carries an output tile, thus tile size is **bounded by available registers**
- **Register-file staging** - 24 registers / thread for fragments
  → RF pressure, serialization in register loading
- Heavy **address-indexing overhead** for the load / store of blocks
- Each mma_sync **expands into 16/32 micro-operations**

```
- mma_sync c_block, a_block, b_block, c_block
```

- **2:4 sparsity** is a fixed pattern, only for one operand, with **max speedup = 2x.**
- **Net effect:** TCU multiplier utilization < 4% - the design is dominated by data movement and instruction-issue overhead

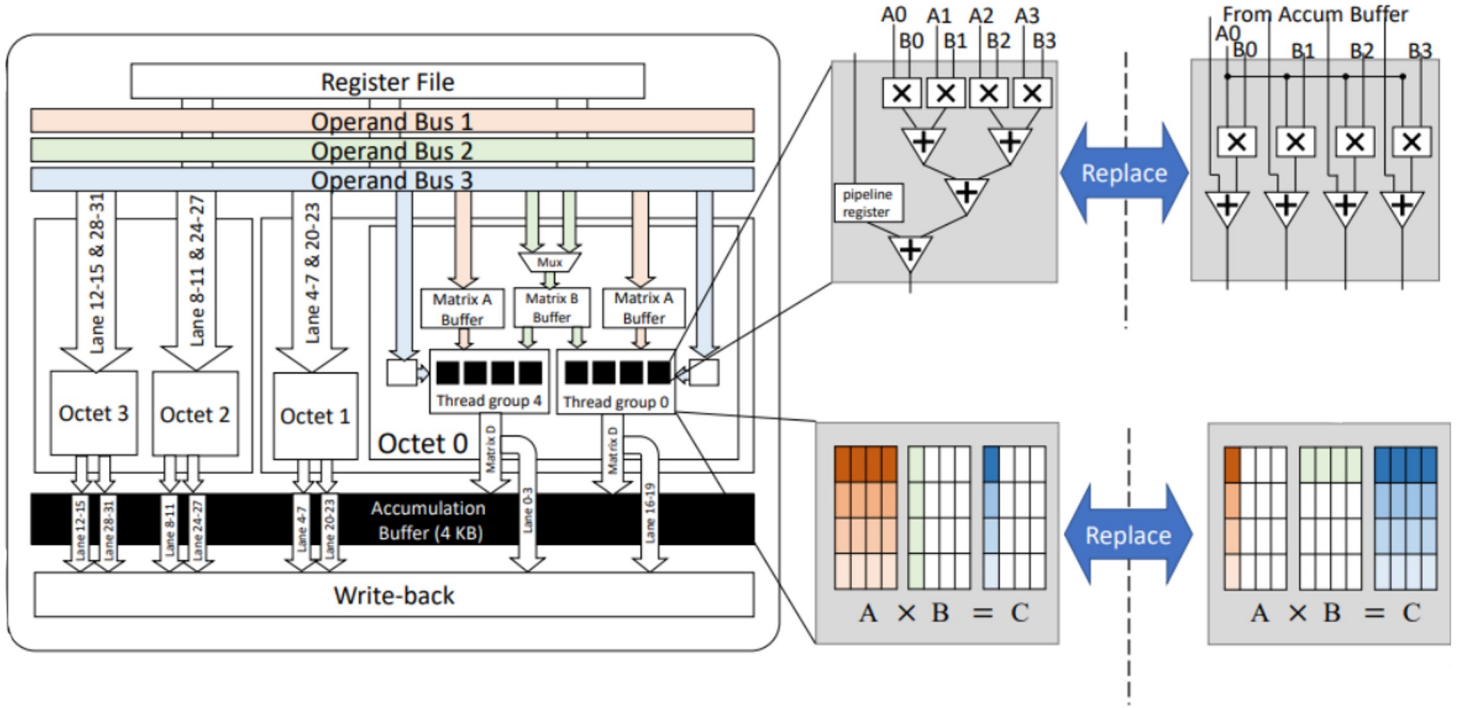


Dual-side Sparse Tensor Core (DSTC)

- Key ideas the DSTC work builds on:
  - **outer-product** primitive **+ bitmap encoding** → avoids the sparse inner-join
  - FEDP replaced by **FEOP** (Fused-Element **Outer-Product**) units
  - **Tile Index Aligner (TIA)** - Unit processing bitmaps and recovering nonzero addresses
  - **Banked Accumulation Buffer + crossbar + queues** (gather-accumulate-scatter)

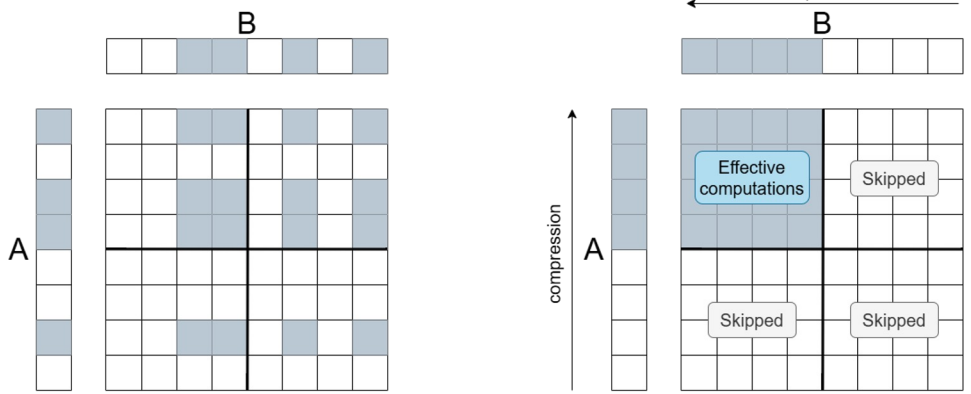
- Proposed & evaluated in **Accel-Sim** (high-level C++ model, not synthesizable)

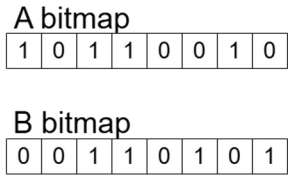
- **This work:** Implement DSTC in RTL within the Vortex framework and resolve key design ambiguities absent from prior simulation studies

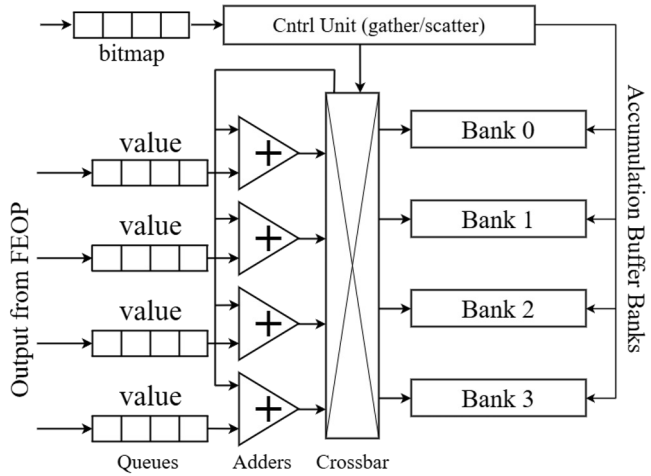


Dual-side Sparse Tensor Core (DSTC)

- **Step**: The sub-region of the tile calculated using the FEOP units in a single cycle.
- **Step-skipping**: After bitmap decoding and compaction, steps with no useful non-zero pairs are not executed. Step-skipping is controled by *software instruction predication*, not by hardware scheduling.
- **Accumulation**: Partial products are fed into the Accumulation Buffer along with their target addresses as calculated from the Tile Index Aligner, to locally update the output tile
- **Gather - Accumulate - Scatter**: FEOP outputs are gathered, accumulated and routed through the crossbar to the target bank

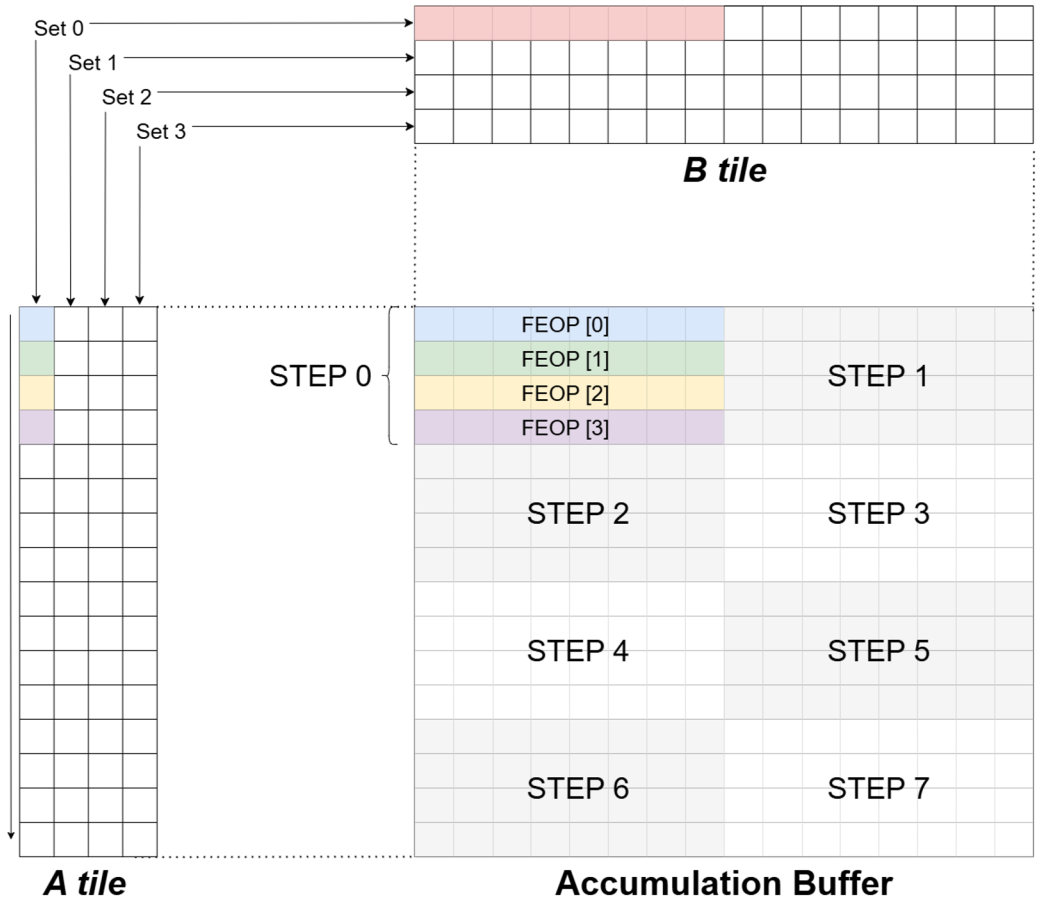






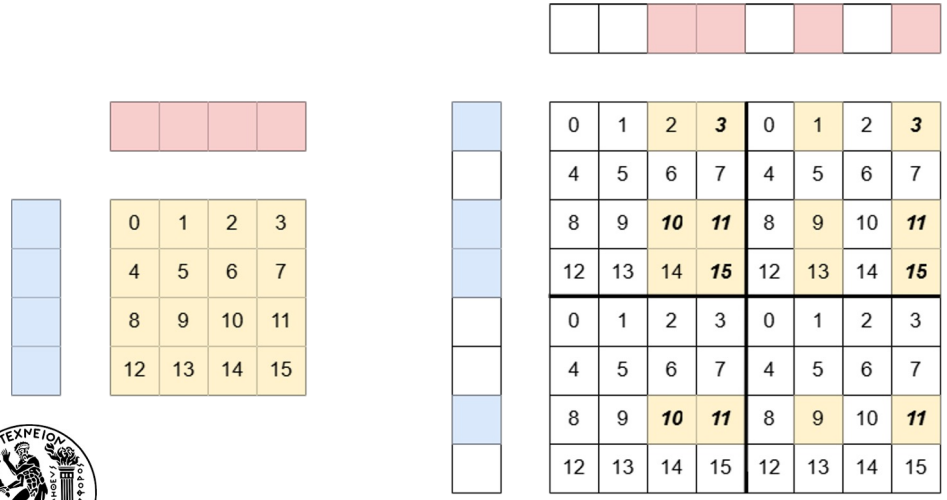
FEOP datapath - sets & steps

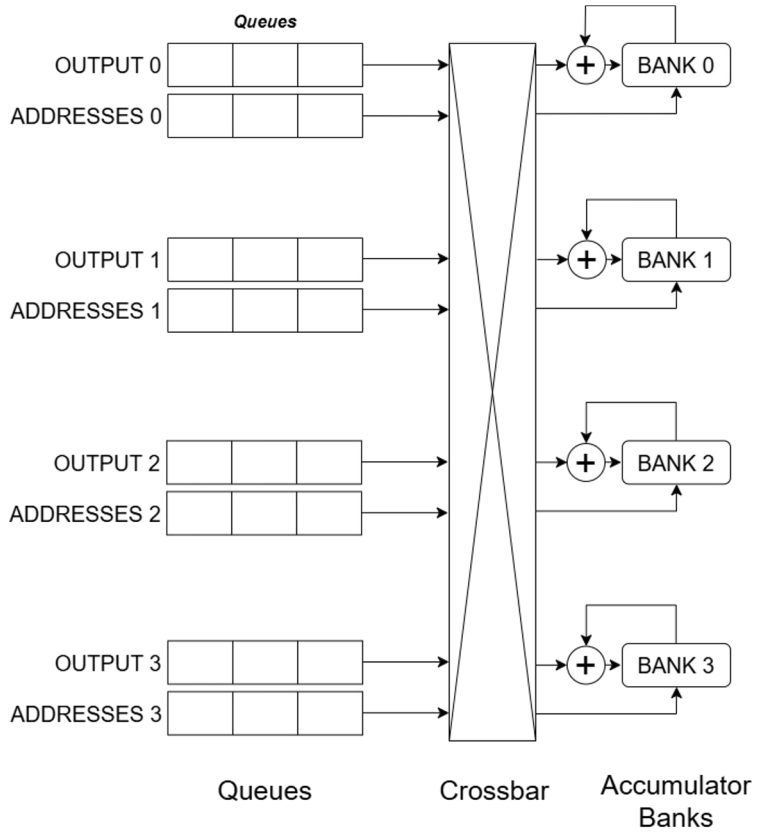
- **FEOP** replaces FEDP: each unit takes *1 element of A* × *n elements of B* → **1 × n** partial products
- Work is organized as **sets** and **steps**:
  - **set** = one column of A × one row of B (a rank-1 contribution)
  - **step** = one m × n sub-region of that set's output, produced by **m FEOPs**
- **Parameterizable m × n** step shape trades compute throughput vs. sparsity granularity
- Supports **FP8** / **FP16** / **FP32** / **INT8** operands



Accumulation Buffer

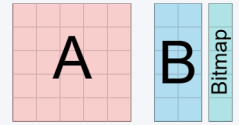
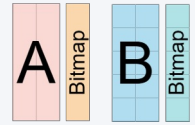
- **32 × 32 output tile** kept resident on-chip (4 KB), **banked** into m × n banks
- Each update = **gather → accumulate → scatter** into the right bank
- Crossbar placed before the bank-local adders → routes each partial product once, less crossbar traffic, simpler write-back
- **Queues** before the crossbar absorb **bank conflicts** caused by irregular sparse addresses





TCU Core Design

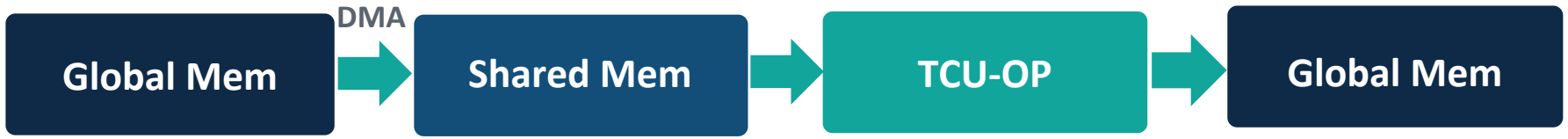
- **TCU core** orchestrates set / step counters, operand dispatch and backpressure
- **Two TIAs** (A → row indices, B → column indices) decode bitmaps and count non-zeros
  - step-skipping decided **in hardware** - not software predication.

Uncompressed × Uncompressed	Dense execution - no bitmaps, no preprocessing	
Uncompressed × Compressed	B = static weights compressed offline; A = activations compressed on-the-fly inside the TCU	
Compressed × Compressed	Both operands pre-compressed → maximum compute & memory savings, at the cost of offline preprocessing.	

Direct shared-memory access & asynchronous execution

Baseline Vortex TCU - per-thread register-file staging

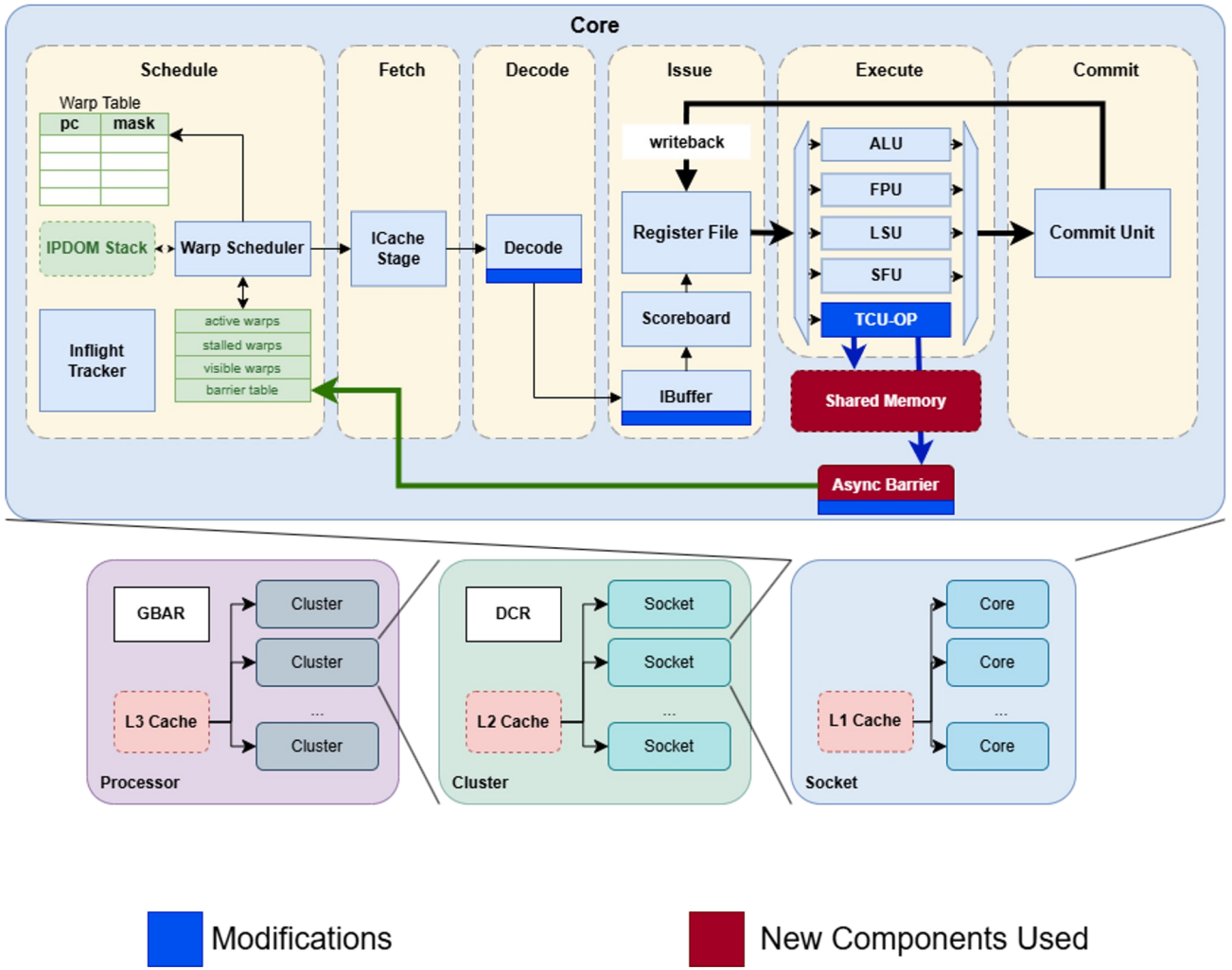
TCU-OP : direct SMEM access (register-file bypass)



- Operand tiles + bitmaps moved to shared memory by **DMA**; TCU reads contiguous blocks directly
- **Asynchronous** execution — the issuing warp keeps doing independent work
- Alleviates **register-file pressure** and **instruction count**
- Enables larger **32 × 32** output tiles, increasing reuse of A / B operands and reducing redundant **memory traffic**

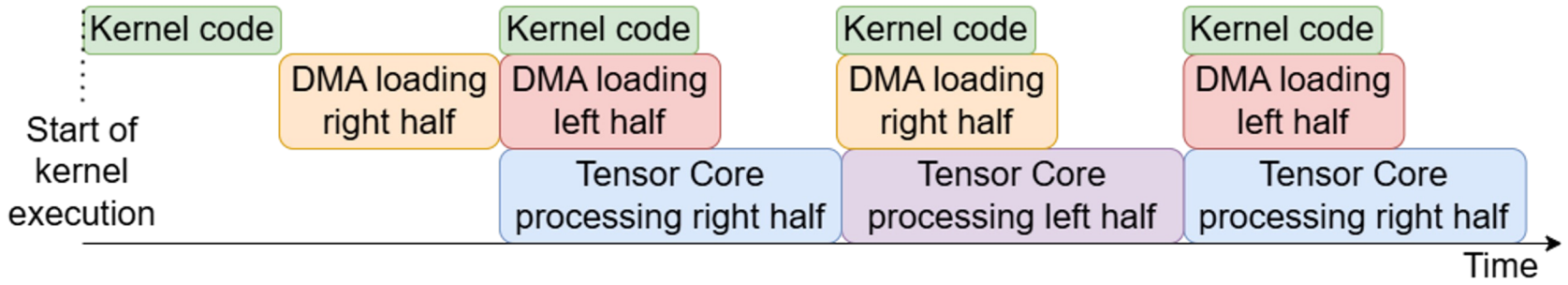
Proposed Microarchitectural changes

- New TCU-OP execution unit added to the Vortex core
- ISA extension launches complete tile TCU-OP operations with a single instruction
- Ibuffer no longer expands each MMA instruction into multiple micro-ops
- Direct shared-memory path for A, B, C and Bitmap blocks
- Matrix operands bypass the Register File
- Asynchronous (transactional) barrier used to synchronize TCU completion



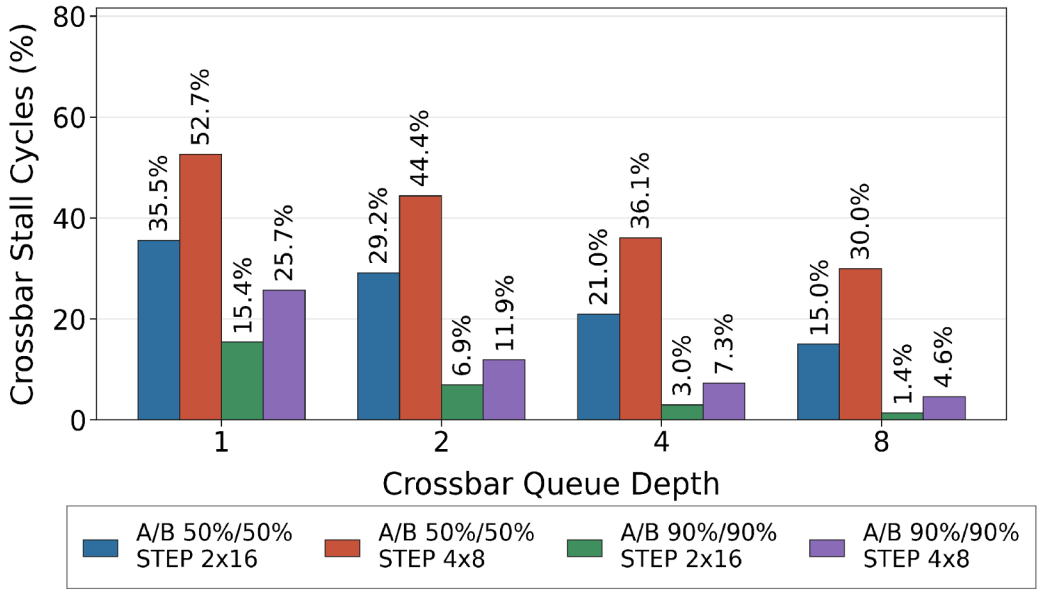
HW/SW co-design & ISA extension

- Optimized kernel **overlaps three phases** via shared-memory **double buffering**:
  - software **metadata generation** · **DMA transfers** · **asynchronous TCU computation**
- One warp **orchestrates**; the remaining warps run independent work
- Output tile **stays resident across the K dimension** → initialize & flush **once per output tile**
- **Single new ISA instruction** encodes everything the TCU-OP needs



Micro-architecture Right-Sizing & Design-Space Exploration

- **Right-sizing computation:** m × n = **32** partial products/cycle — ¼ of the baseline's 128 multipliers
  - workloads are **memory-bound** → negligible performance loss, large area/power/crossbar savings
- **Step shape: 2×16 vs 4×8:**
  - 2×16 → fewer crossbar stalls + lower flushing overhead → **selected**
- **Queue depth** swept 1 → 8: **depth 4** captures most of the stall reduction at low cost
- These choices determine the **default configuration** for the evaluation

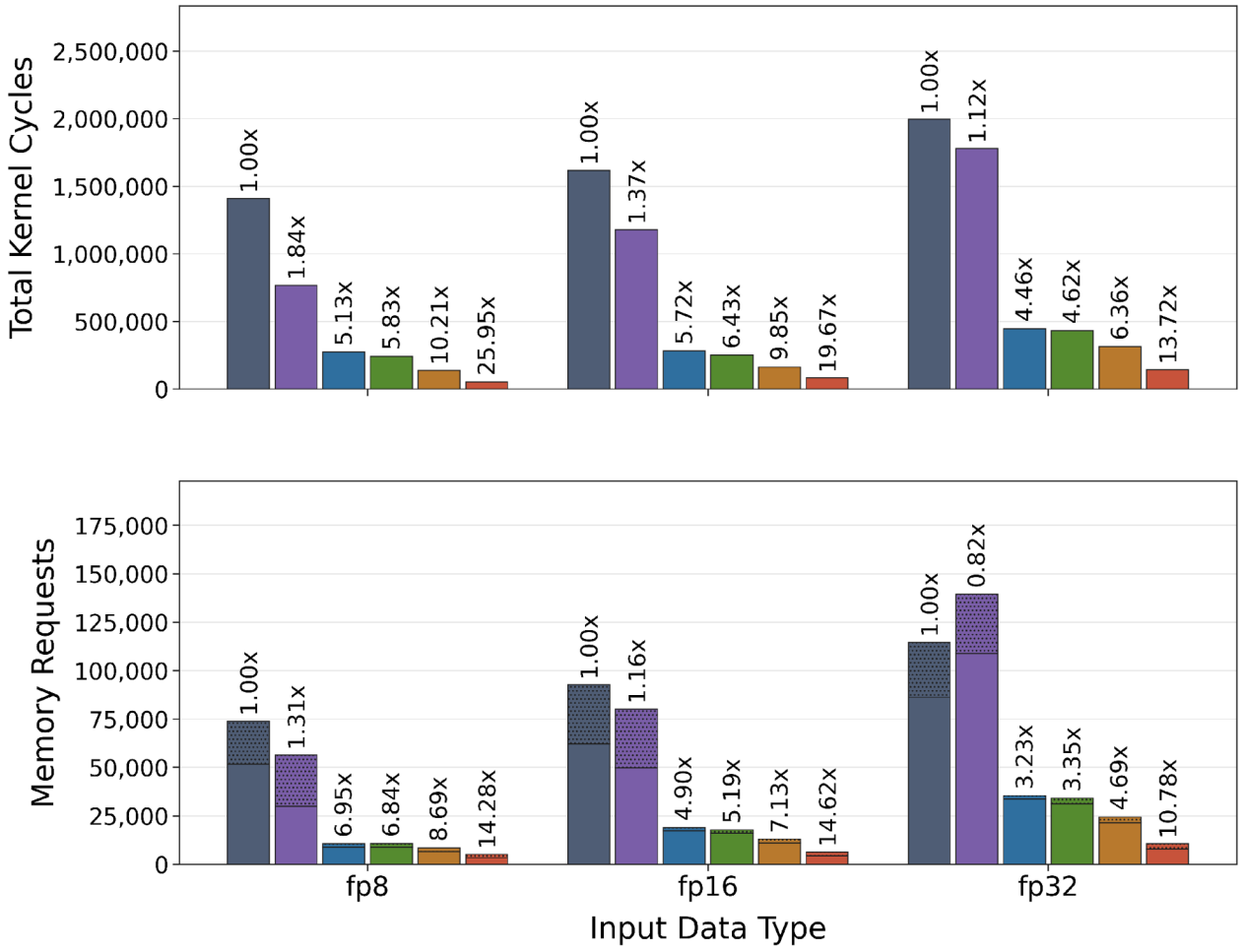


Experimental setup

- **Cycle-accurate RTL** simulation (Vortex + Verilator) - single GPU core, 32-thread warps
- Workload: **GEMM D = A × B + C**, main size (**M × N × K) = (128 × 128 × 512)**; FP8 / FP16 / FP32
- Baselines:
  - **SGEMM** - dense Vortex Tensor Core
  - **SGEMM_SP** - 2:4 structured sparse (theoretical max 2×)
  - **SGEMM_OP** - proposed TCU-OP (unstructured sparse)
- Metrics: **total kernel cycles, instruction count, DRAM traffic, DMA transactions, TCU utilization**

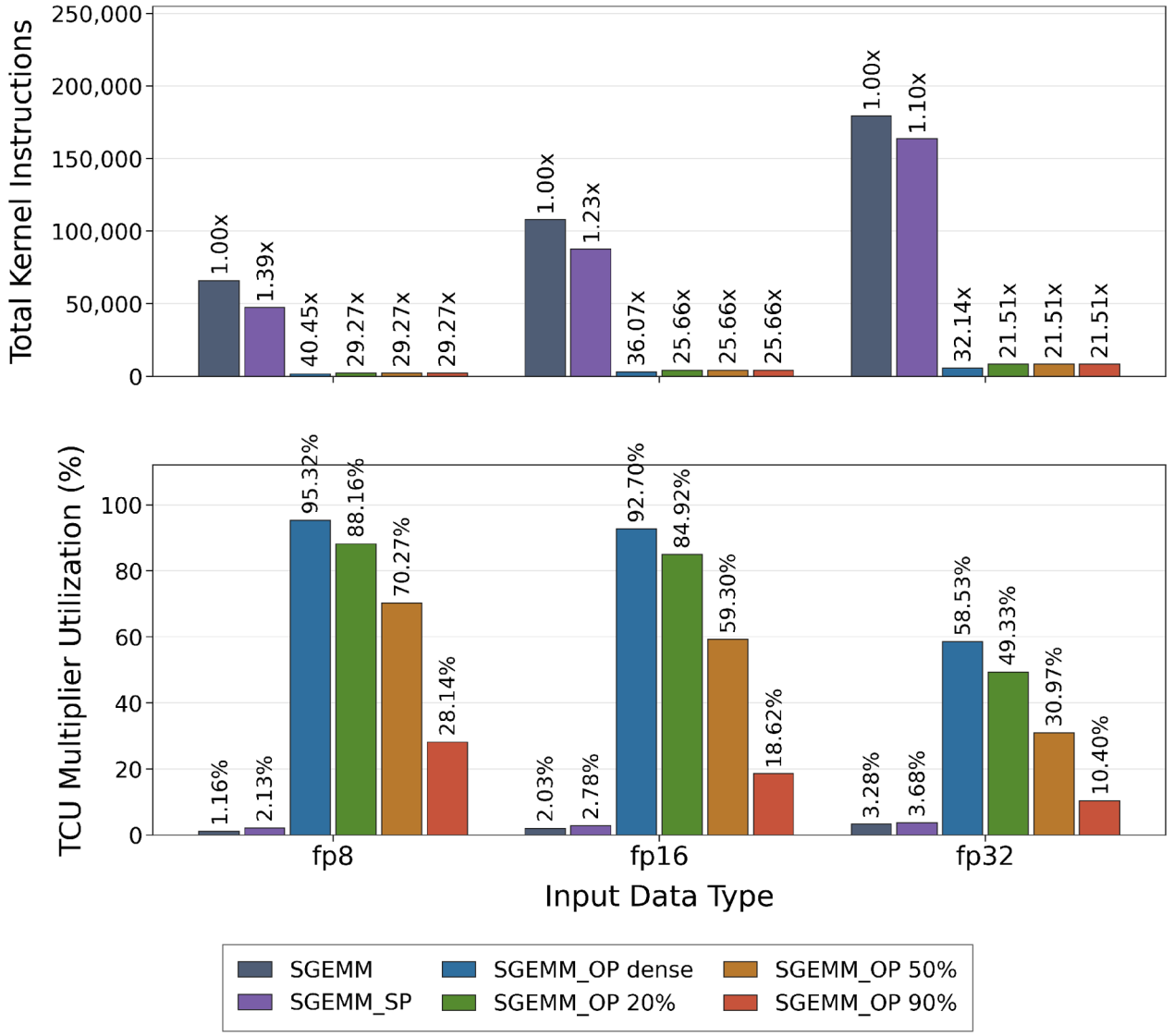
Results - speedup over the dense GEMM baseline

- **Dense TCU-OP alone:** 4.5 - 5.7×
  from the memory interface + tile reuse
  Already **2.8 - 4.2×** faster than the 2:4 baseline
- **50% / 50%**: 6.4 - 10.2×
  90% / 90%: 13.7 - 25.95×
- Vs the 2:4 baseline at 90% / 90%: **≈ 12 - 14×**
- **Low precision (FP8) benefits most** - more values per memory request
- Total speedup ranging from 4.46× to 25.95×
- **Off-chip memory traffic** drops up to **14.62×**



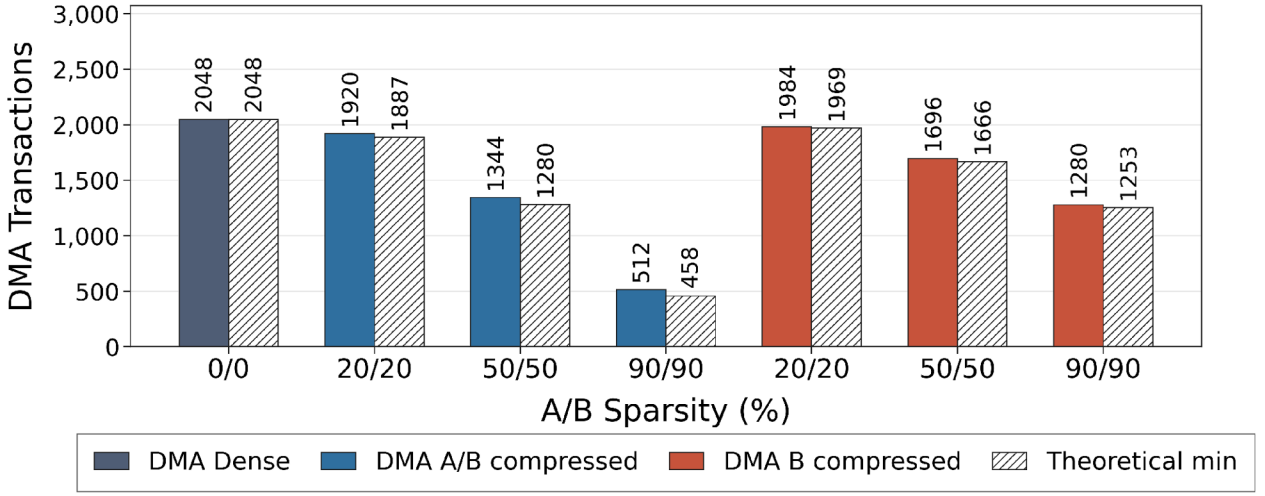
Results - instructions & utilization

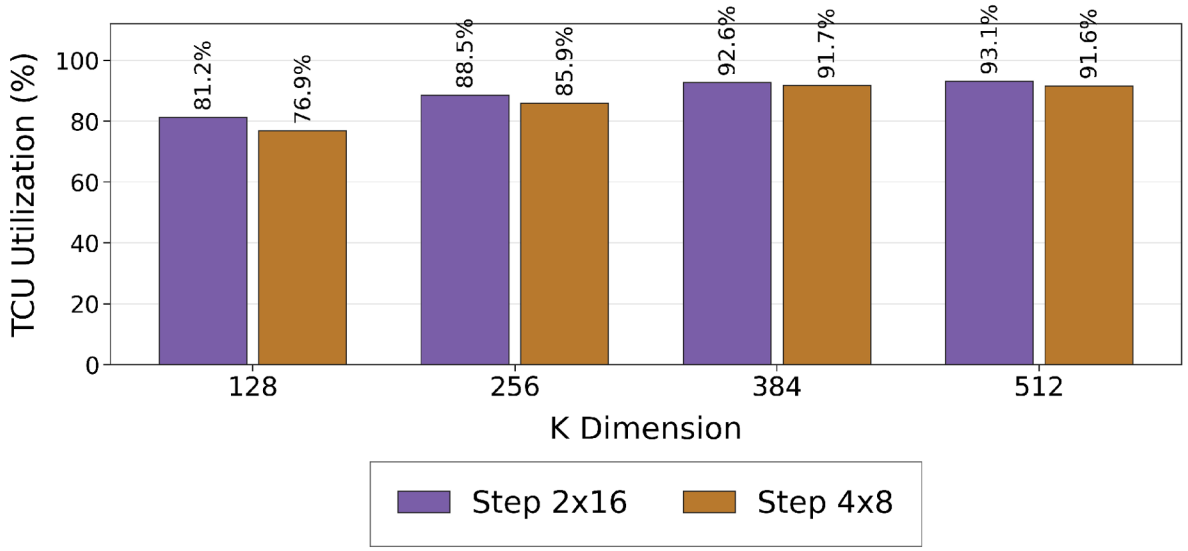
- **Instruction count ↓ ~25×** - one async instruction vs. RF staging + micro-ops
- Baselines **below 4%** (3.28% dense, 3.68% for 2:4) - memory/issue bound
- **TCU-OP reaches up to 95.3%** (dense FP8) - double buffering + RF bypass keep the datapath active
- Utilization **reduced at high sparsity** - highly sparse kernels are memory-bound



Results - DMA traffic & K-sensitivity

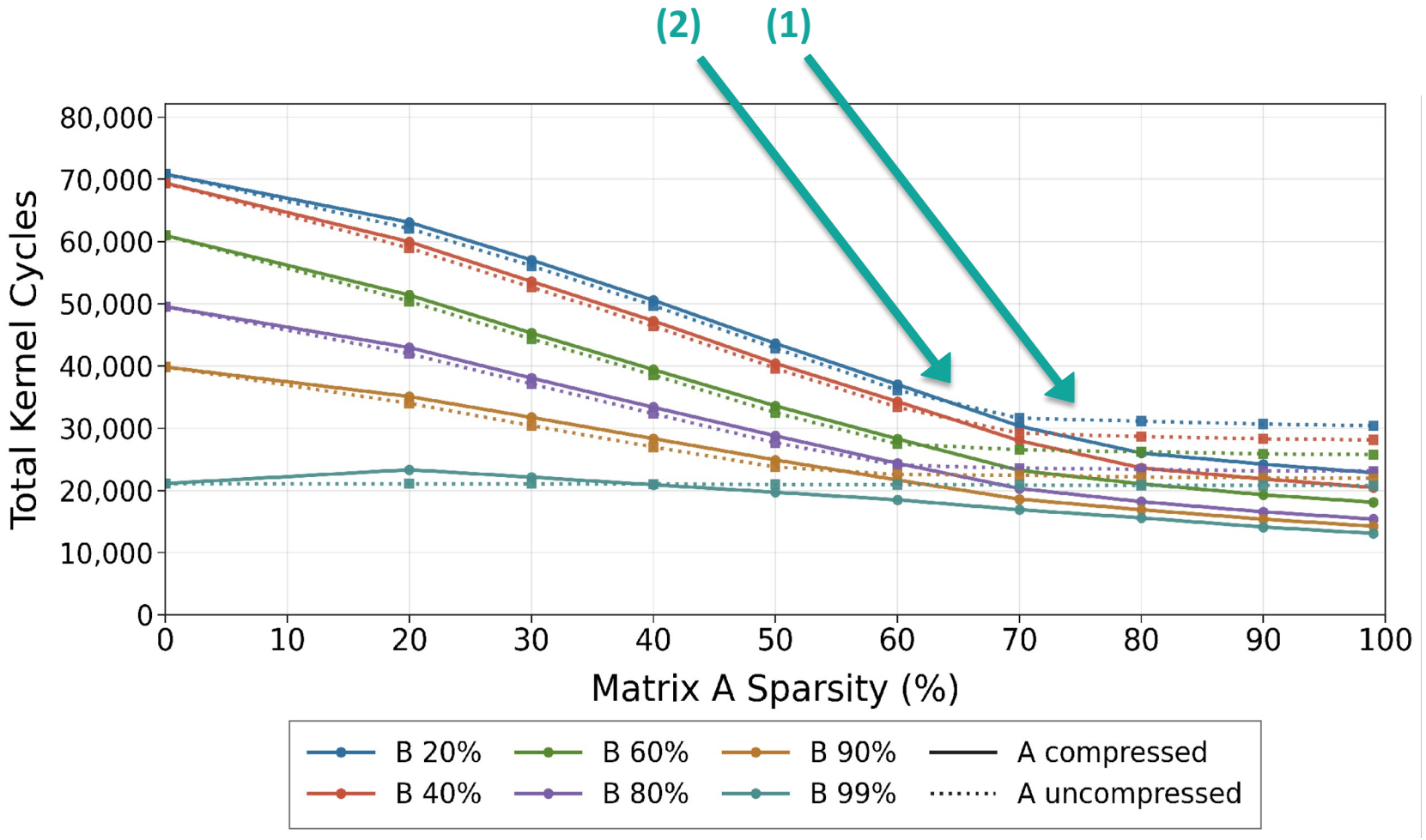
- **DMA traffic** drops by **75%** at 90%
  measured ≈ theoretical minimum
- Red bars show the
  uncompressed × compressed mode:
  A is still transferred full,
  so DMA traffic is higher than
  compressed × compressed
- **K-dimension:** utilization **81%** at **K=128,**
  ~93% at **K=512** as fixed overheads amortize





Results - sparsity scaling

- **End-to-end sparse speedup** (2-D A/B sweep)
- Compressed × compressed: **4.58×** at **90% / 90%**, **5.38×** at **99% / 99%**
- Uncompressed × compressed: **3.20×** at **90% / 90%**, **3.39×** at **99% / 99%** - no A preprocessing
- **Compute-bound up to ~70 - 80%** sparsity, then gradually memory-bound (1)
- Uncompressed × compressed plateaus near **60 - 70%**, since A traffic is not reduced (2)



Related Work - where TCU-OP stands

Design	Sparsity support	Dual-side sparsity	Unstructured sparsity	Direct SMEM	RF bypass	Synthesizable
VEGETA (CPU)	✓	—	—	—	—	—
Coop. Warp Exec.	—	—	—	—	—	✓
Virgo	—	—	—	✓	✓	✓
Vortex 2:4 TC	✓	—	—	—	—	✓
DSTC	✓	✓	✓	—	—	—
TCU-OP (ours)	✓	✓	✓	✓	✓	✓

Conclusions & future work

25.95×	5×	95.3%	25×	14.62×
Peak Speedup	Dense Speedup	TCU Utilization	Fewer Instructions	Less DRAM Traffic

- Effective sparse matrix multiplication acceleration requires a **dataflow re-design & SW/HW co-design.**
- **TCU-OP** : The first **RTL, dual-side, unstructured** sparse Tensor Core

- **Future work:**
  - Include compression / preprocessing overheads; alternative encodings (COO, CSR, hybrid)
  - DNN kernel + independent work overlap evaluation
  - Physical design - clock / power gating, SRAM, layout-aware crossbar

Thank you!

Code: github.com/GiorgosTriantafyllou324/vortex

Georgios Triantafyllou · NTUA Microlab · 2026

Appendix 1 - Stall Frequency Analysis

$$\text{bank}(r, c) = (r \bmod m) \cdot n + (c \bmod n).$$

$$\text{unique\_banks} = R \cdot C.$$

$$\begin{aligned}\mathbb{E}[\text{conflicts}] &= 32 - \mathbb{E}[\text{unique\_banks}] \\ &= 32 - \mathbb{E}[R] \cdot \mathbb{E}[C].\end{aligned}$$

$$\mathbb{E}[\text{occupied}(d, k)] = d\left(1 - \frac{\binom{32-32/d}{k}}{\binom{32}{k}}\right)$$

For the 2×16 configuration, the expected number of occupied row and column residue classes is

$$\mathbb{E}[R] = 2\left(1 - \frac{\binom{16}{2}}{\binom{32}{2}}\right) = 1.516,$$

and

$$\mathbb{E}[C] = 16\left(1 - \frac{\binom{30}{16}}{\binom{32}{16}}\right) = 12.129.$$

Thus, the expected number of bank conflicts is

$$\mathbb{E}[\text{conflicts}] = 32 - 1.516 \cdot 12.129 \approx 13.6.$$

For the 4×8 configuration, the corresponding values are

$$\mathbb{E}[R] = 4\left(1 - \frac{\binom{24}{4}}{\binom{32}{4}}\right) = 2.818,$$

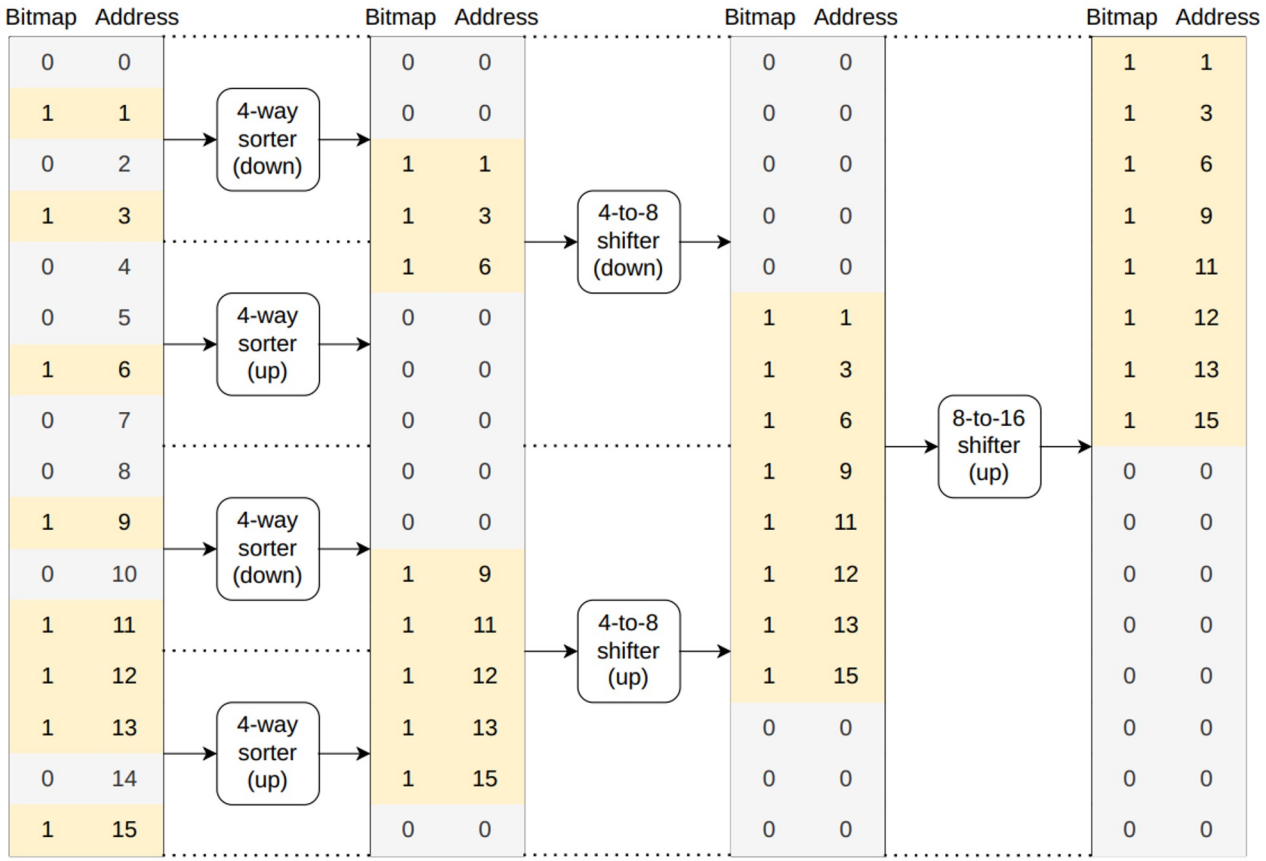
and

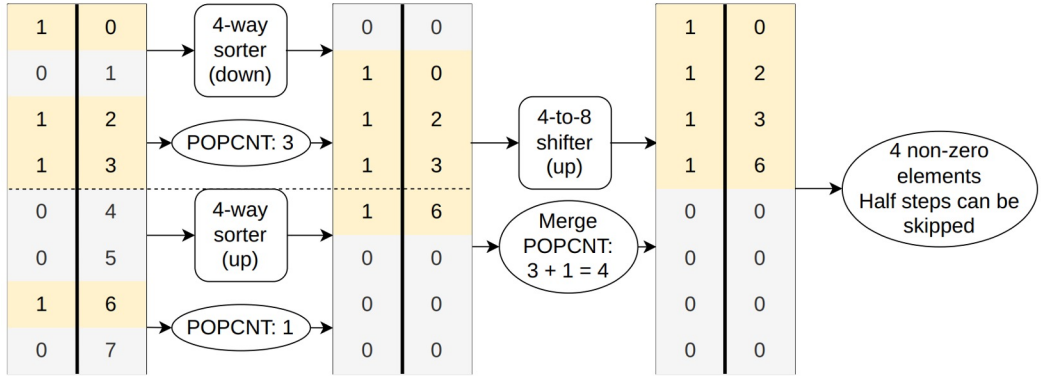
$$\mathbb{E}[C] = 8\left(1 - \frac{\binom{28}{8}}{\binom{32}{8}}\right) = 5.636.$$

Therefore,

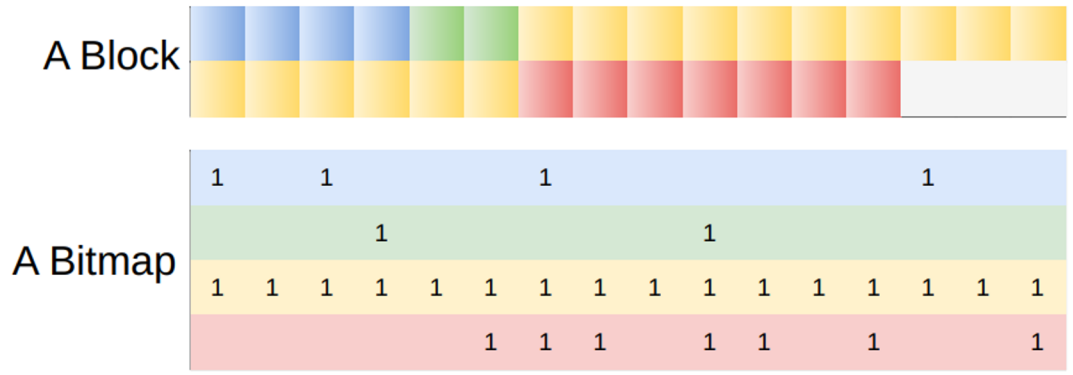
$$\mathbb{E}[\text{conflicts}] = 32 - 2.818 \cdot 5.636 \approx 16.1.$$

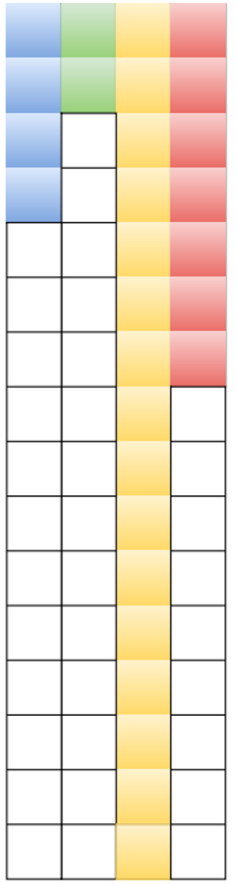
Appendix 2 - Tile Index Aligner



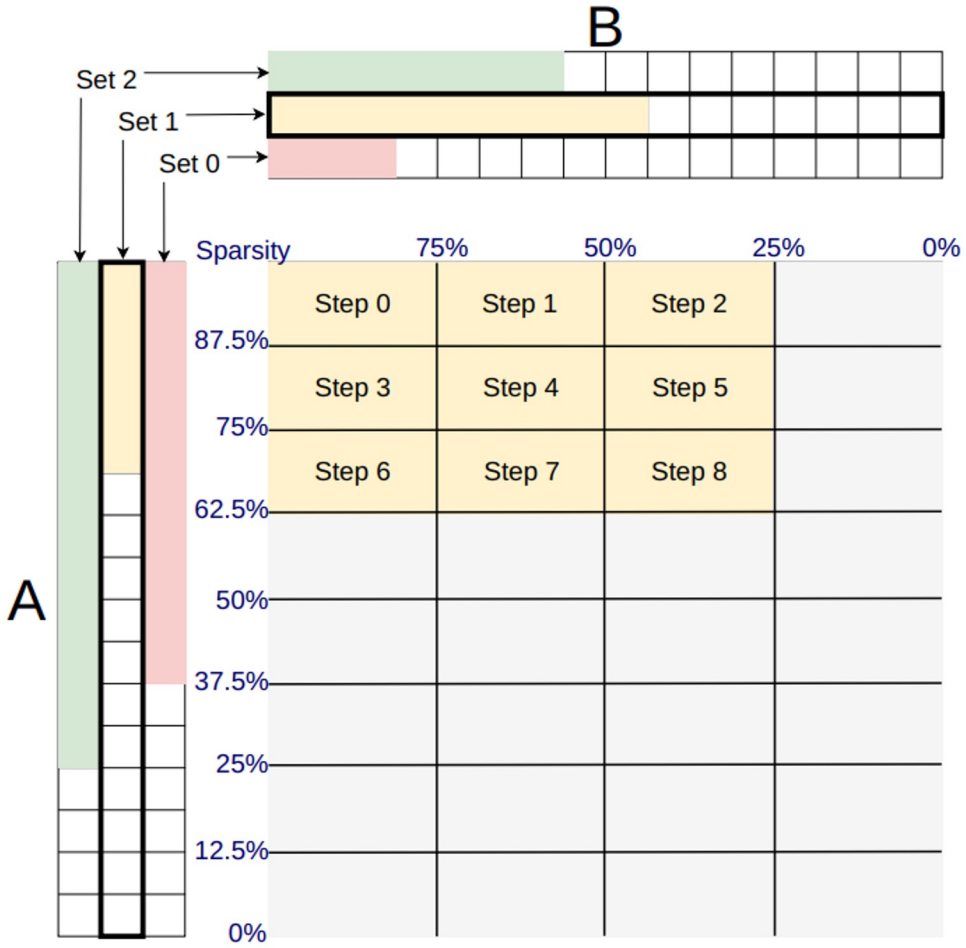


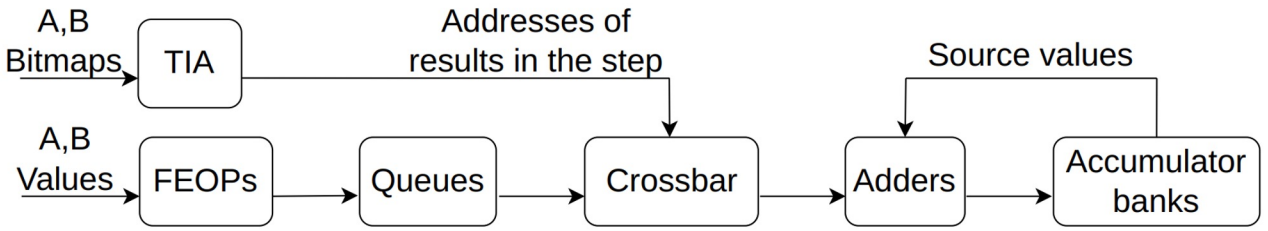
Appendix 3 - Compressed Data Interpretation



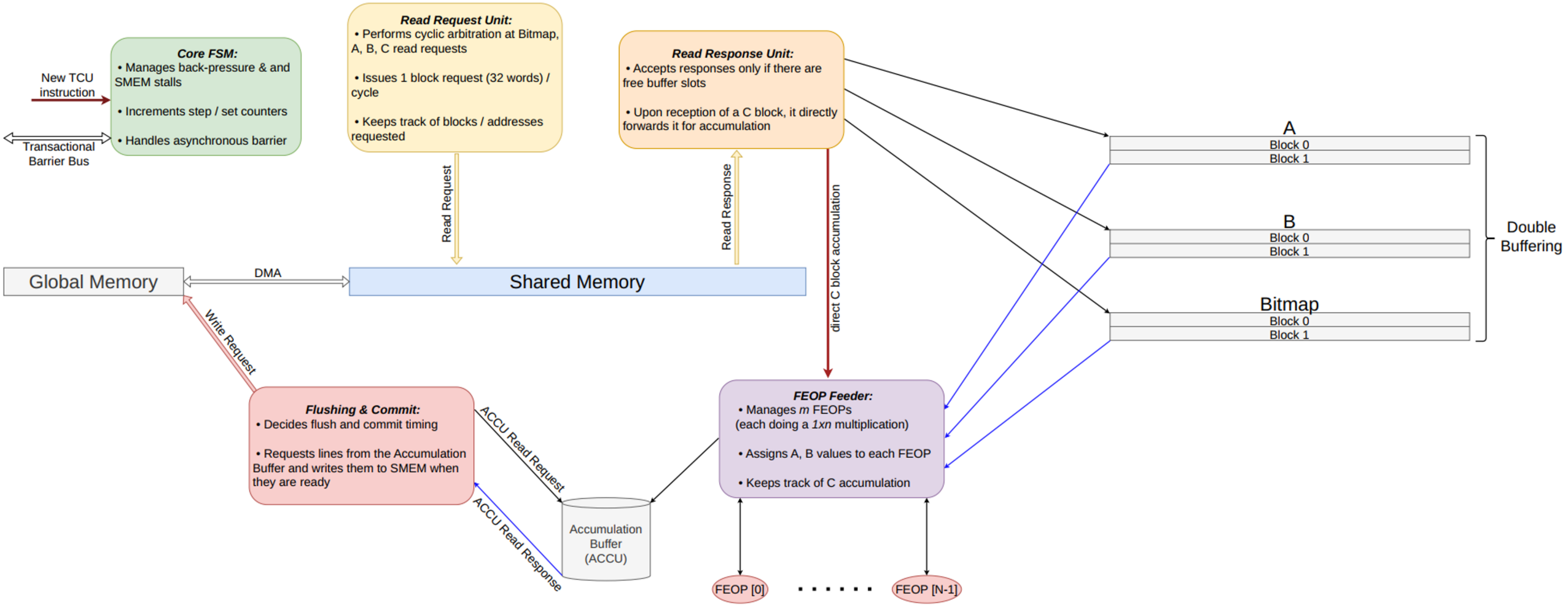


Appendix 4 - Step-skipping Example & TCU Pipeline

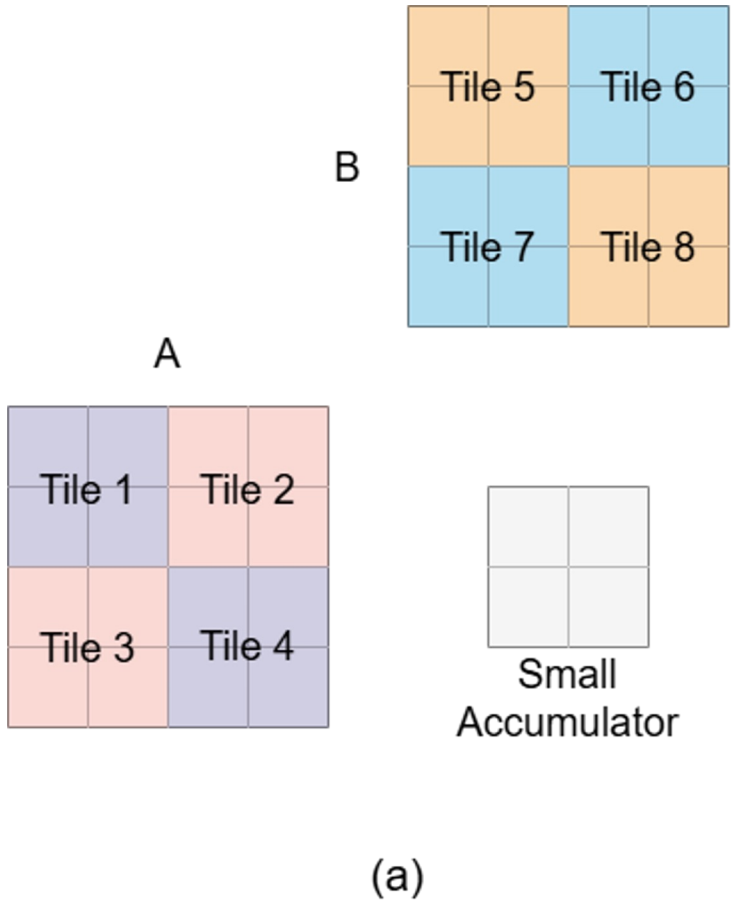


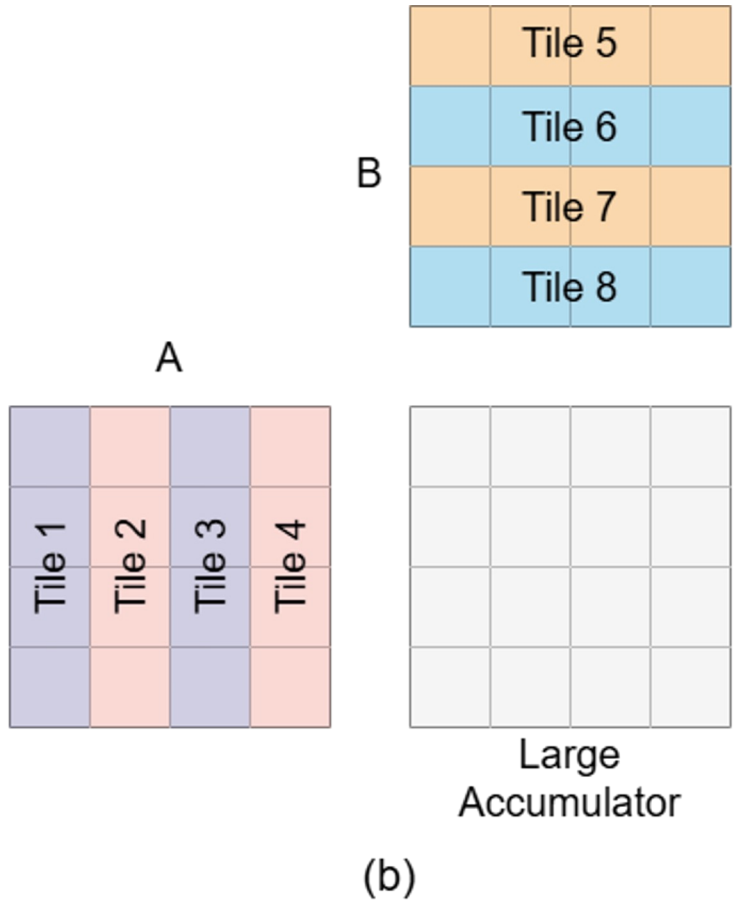


Appendix 5 - TCU Overview



Appendix 6 - Tile Reuse





Code: github.com/GiorgosTriantafyllou324/vortex

Georgios Triantafyllou · NTUA Microlab · 2026