TMEM: Implementing Blackwell-Inspired Tensor Memory in the Vortex RISC-V GPGPU

Eliot Yoon
Department of Computer Science
University of California, Los Angeles
Los Angeles, USA
eyoon1131@ucla.edu

***Abstract*—Modern GPU tensor cores increasingly move the MMA accumulator off the register file to eliminate register pressure as a bottleneck on tile size. NVIDIA Blackwell introduced Tensor Memory (TMEM), a dedicated on-chip memory for the tensor core, along with the UMMA instruction that sources and writes MMA data directly from TMEM rather than the register file.**

This work implements TMEM and UMMA in Vortex, an open-source RISC-V GPGPU. The implementation includes a five-instruction ISA extension, a configurable TMEM storage array, integration of the UMMA operation with the existing WGMMA pipeline, and a programmer-facing kernel API. A key microarchitectural difference from the existing WGMMA operation is the k-outer uop loop ordering, which eliminates TMEM RAW hazards without additional stall logic.

RTL simulation results demonstrate that UMMA nearly matches WGMMA performance at the WGMMA-supported NRC configurations, while improving on the performance with support for larger NRC that are inaccessible for WGMMA due to register pressure. UMMA achieves up to a $3\times$ cycle reduction for large matrices compared to WGMMA. The results also reveal that WGMMA IPC degrades non-monotonically with NRC due to accumulator register pressure, while UMMA IPC increases monotonically, demonstrating the architectural benefit of moving the accumulator off the register file.

I. Introduction

As deep learning and artificial intelligence has grown rapidly in the past decade, so has the demand for matrix multiplication throughput. Floating point matrix operations are the backbone of modern neural network training and inference, particularly the multiply-accumulate chains that appear in attention layers, convolutional filters, and fully connected layers. General-purpose SIMD units are not optimized for this workload - they are designed for per-thread scalar operations, not the large, fused tile computations that are characteristic of GEMM. This was the primary motivation for the development of dedicated hardware tensor cores.

NVIDIA introduced the first generation of tensor cores with the Volta V100 in 2017 [1]. The Volta WMMA (Warp Matrix Multiply-Accumulate) instruction allowed an entire warp to cooperate on a small $M \times N \times K$ matrix multiply, with results accumulated into register-file fragments distributed across the warp's threads. Successive generations refined this interface. Turing added support for more data types [2]. Ampere introduced asynchronous data movement and the warpgroup abstraction, allowing multiple warps to coordinate on larger output tiles [3]. Hopper extended the warpgroup interface with TMA (Tensor Memory Accelerator) and WGMMA, enabling operands to be sourced directly from shared memory via descriptors rather than loaded into registers first [4].

In 2024, Blackwell took the next logical step in this evolution by introducing Tensor Memory (TMEM), a dedicated on-chip per-SM SRAM array that the tensor core can directly read or write to [5]. The corresponding UMMA (Unified Matrix Multiply-Accumulate) instruction sources its accumulator C from TMEM and writes its result D back to TMEM, keeping the register file free.

This report describes the design and implementation of TMEM and UMMA in Vortex, an open-source RISC-V GPGPU [6]. Section II provides background on the WGMMA and TMEM programming models in NVIDIA hardware. Section III describes the Vortex architecture, the limitations of its existing WGMMA implementation, and the full implementation of TMEM and UMMA. Section IV presents RTL simulation results measuring the performance across varying tile and matrix sizes. Section V discusses limitations and future work.

II. Background

A. Warpgroup Matrix Multiply-Accumulate (WGMMA)

Beginning with Hopper, NVIDIA's PTX programming model exposes warpgroup-level MMA instructions that let multiple warps cooperate on a single output tile [7]. In WGMMA (`wgmma.mma_async`), a warpgroup of four warps collectively computes an $M \times N \times K$ tile, where M is fixed at 64, N is configurable, and K depends on the data precision. The A operand can come from registers or shared memory via a descriptor; B always comes from shared memory. The output C/D accumulator tile is distributed across the warpgroup's floating-point registers.

Due to the accumulator occupying the register file, a large WGMMA tile can consume a substantial portion of the available registers per warpgroup. This register pressure limits the tile size a programmer can choose and constrains occupancy, as fewer registers available for other computation means fewer warps can be resident simultaneously, reducing the GPU's ability to hide memory latency via warp switching.

B. NVIDIA Tensor Memory (TMEM) and UMMA

Blackwell introduces TMEM, a dedicated on-chip SRAM array designed specifically for use by the tensor core [5]. It is not accessible to ordinary load/store instructions; instead it is managed by a small set of dedicated instructions (`tcgen05.alloc`, `tcgen05.dealloc`, `tcgen05.st`, `tcgen05.ld`) and consumed directly by the UMMA instruction (`tcgen05.mma`).

In UMMA, the output accumulator D and input accumulator C both reside in TMEM. The instruction fetches C from TMEM at the start of each tile computation and writes D back to TMEM when the pipeline completes. The A and B operands continue to come from shared memory. Blackwell does provide support for storing A directly in TMEM and sourcing it from TMEM rather than through shared memory, but this is outside the scope of this work. To allow for a fair comparison with WGMMA, future sections only explore the case where A and B both come from shared memory.

The key architectural benefit of TMEM is that accumulator register pressure is eliminated regardless of tile size. On large NVIDIA GPUs the impact is significant at the largest tile sizes, but on the RISC-V based Vortex, where each thread is limited to 32 floating-point registers, the constraint is far more severe. This is discussed further in Section III.

III. Implementation

A. Vortex GPGPU Overview

Vortex is an open-source, fully parameterizable RISC-V GPGPU designed for research and education. It implements the SIMT execution model, where each warp contains a configurable number of threads and each SM contains a configurable number of warps. The threads in a warp execute in lockstep, while the warp scheduler interleaves warps to hide latency. The baseline ISA is RV32I/F, and architectural extensions are enabled via preprocessor flags.

The tensor core extension (enabled by `-DEXT_TCU_ENABLE`) provides a dedicated execution path for matrix multiply-accumulate operations. MMA instructions are decoded into the `EX_TCU` execution type and dispatched as a series of uops to the tensor core unit, which distributes them across tensor core instances. A single tensor core is able to perform a $tcM \times tcN \times tcK$ MMA operation with its FEDP units. The WGMMA extension (`-DTCU_WGMMA_ENABLE`) adds the tile buffer infrastructure, in which A and B tile data is fetched from local shared memory asynchronously and presented to the FEDP pipeline in the tensor core.

The FEDP pipeline is the actual compute engine of the tensor core. For each output element (i, j) of the $tcM \times tcN$ output tile, an FEDP unit computes the inner product of A row i with B column j, adding the accumulator C. A new uop can enter the pipeline every cycle, allowing for high throughput.

There are two key limitations of the WGMMA implementation that motivate this work. First, since accumulator registers are consumed throughout the entire WGMMA instruction, the maximum N dimension of the warpgroup tile is heavily constrained. N is determined at kernel runtime based on the chosen number of C accumulator registers per thread (`NRC`) and the number of threads. With the default Vortex configuration of 4 threads per warp, $N =$ `NRC`, meaning an `NRC` of 32 exhausts the entire FP register file for accumulation. Second, since the accumulator lives in the register file it is tracked by the scoreboard, creating RAW dependencies between uops that access the same accumulator element. These hazards must be managed by the uop loop ordering to avoid stalls in the pipeline.

B. ISA Design

All TMEM instructions share the RISC-V CUSTOM0 opcode (`0x0b`) with `funct7=2`, distinguished by `funct3`. The full instruction set is listed in Table I.

The TMEM address encoding packs (`lane_base`, `col`) into a single 32-bit word: bits[31:16] = `lane_base`, bits[15:0] = `col`. This allows a single integer register to specify both dimensions of the TMEM address. The exact layout of TMEM will be explained in detail in the next section.

The UMMA instruction reuses the same opcode fields as WGMMA but with a different `funct3`. In WGMMA, the `rs2` field encodes a set of flags in the 5-bit register number: bit 0 is `is_sparse`, bits [2:1] encode `cd_nregs` (selecting `NRC` = 8, 16, or 32), and bit 3 encodes `a_from_smem` (whether A is sourced from shared memory or registers). For UMMA, `is_sparse` is always 0 (no sparsity support yet) and `a_from_smem` is always 1 (UMMA only supports SS-mode), so bits [3:1] are repurposed as a 3-bit `NRC` encoding: {`a_from_smem`, `cd_nregs`} maps 0→`NRC`=8, 1→16, 2→32, 3→64, 4→128. This encoding is computed at runtime and embedded as an immediate register number in the inline assembly.

The kernel API in `vx_tensor.h` exposes TMEM through the `umma_context<NT, InputType, OutputType, NRC>` template to abstract the low-level TMEM address computation from the programmer. The key operations are:

- `fill_tmem(value)` — Initialize the accumulator tile for a given warpgroup with a scalar value.
- `umma_sync(desc_a, desc_b)` — Perform one UMMA tile multiply with both operands sourced from shared memory, accumulating into TMEM.
- `store_output(C_global, tile_row, tile_col, N)` — Read the accumulated result from TMEM and write to the global output matrix.

It is important to note that `desc_a` and `desc_b` are not passed via the `rs` fields in the UMMA instruction encoding, but rather through dedicated argument registers `a0` and `a1` (`x10` and `x11` in the RISC-V ABI). This is because the `rs1` and `rs2` fields in the `.insn r` encoding are used to carry the input format identifier (`It::id`) and the `NRC` flags respectively, leaving no `rs` field available for the descriptors.

funct3	Instruction	Operands	Description
0x2	TMEM_ALLOC	rs2=ncols	Allocate TMEM, zero-initialize
0x3	TMEM_DEALLOC	—	Deallocate TMEM
0x4	TMEM_ST	rs1=addr, rs2=value	Store float to TMEM per thread
0x5	TMEM_LD	rs1=addr, rd=result	Load float from TMEM per thread
0x6	UMMA	rd=Ot, rs1=It, rs2=flags	Tile MMA with TMEM accumulator

TABLE I
TMEM INSTRUCTION ENCODING (OPCODE CUSTOM0, FUNCT7=2).

C. Architectural Design

1) TMEM Storage Array: TMEM is implemented as a two-dimensional register array in `VX_tcu_unit.sv`:

```
logic [31:0] tmem_data
[TCU_TMEM_LANES][TCU_TMEM_COLS]
```

where `TCU_TMEM_LANES` = `NUM_WARPS`×`NUM_THREADS` and `TCU_TMEM_COLS` = 256. The maximum TMEM capacity is thus configured at compile time. With the standard Vortex configuration of 4 warps and 4 threads per warp, the total capacity is $16 \times 256 \times 32$ bits = 16 KB per core.

Each TMEM lane corresponds to one row of one warp's UMMA output tile. The lane and column address for output element (i, j) of warp w at step indices (m, n) are:

$$\texttt{lane} = w \times \texttt{xtileM} + m \times tcM + i$$
$$\texttt{col} = n \times tcN + j$$

where `xtileM` is the per-warp M tile dimension, tcM and tcN are the hardware tensor core tile dimensions, and m (`step_m`) and n (`step_n`) count the number of tcM- and tcN-sized steps needed to cover the full per-warp tile.

While the primary method of accessing TMEM is through the UMMA instruction, the `TMEM_ST` and `TMEM_LD` instructions provide direct per-element access for initializing and reading back the accumulator. These instructions take a 32-bit TMEM address that is uniform across the warp: bits [31:16] encode `lane_base` and bits [15:0] encode `col`. Each thread t independently accesses lane `lane_base` + t, allowing all threads in a warp to read or write to consecutive lanes in a single instruction. This addressing scheme is illustrated in Figure 1. The figure shows `NUM_WARPS` × `NUM_THREADS` = 16 lanes, with `xtileN` = 8 columns allocated out of the available 256. `xtileN` is a function of `NUM_THREADS`, a hardware configuration parameter, and `NRC`, a kernel configuration parameter that determines the `umma_context` template instantiation. The actual number of TMEM columns used is set at runtime when the `TMEM_ALLOC` instruction executes with `rs1=xtileN`.

In a production implementation, TMEM would be implemented as SRAM rather than flip-flops. The current FF-based implementation is functionally correct and produces accurate cycle counts in RTL simulation, but is not accurate in terms of area and power overhead.

2) Uop Sequencer: The UMMA instruction is expanded into `NRC`×`k_steps` micro-ops in `VX_tcu_uops.sv`. Each uop carries (`step_m`, `step_n`, `step_k`) indices that select the specific A, B, and accumulator tiles for that computation.

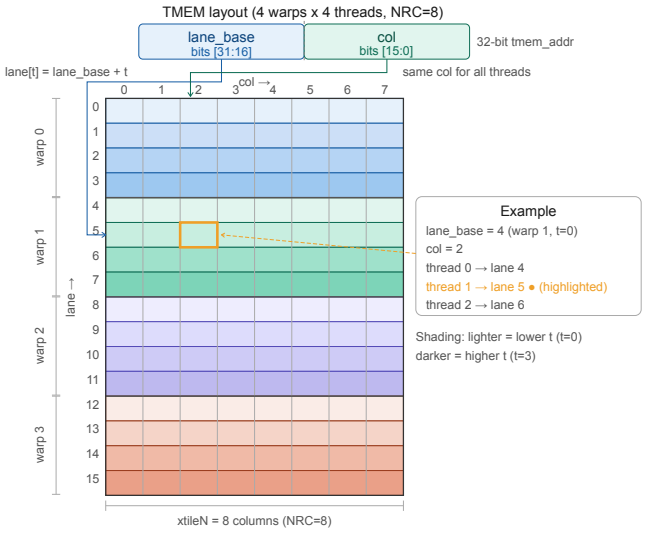

Fig. 1. TMEM array layout for the default Vortex configuration (`NUM_WARPS=4`, `NUM_THREADS=4`, `NRC=8`), showing 16 lanes × 8 columns. Each warp occupies a contiguous block of 4 lanes. The highlighted cell illustrates how the address `lane_base=4`, `col=2` with thread $t = 1$ resolves to lane 5, column 2 — the second element of warp 1's region.

Figure 2 demonstrates the breakdown of a single warp's UMMA instruction into a uop sequence. `xtileM`, `xtileN`, and `xtileK` are the per-warp tile dimensions. With the default Vortex configuration of 4 warps, 4 threads per warp, and NRC of 8, these dimensions are set to `xtileM` = 4, `xtileN` = 8, and `xtileK` = 4. The tensor core tile dimensions are $tcM = 2$, $tcN = 2$, and $tcK = 2$. Thus, the UMMA instruction decomposes into `m_steps * n_steps * k_steps` $= 2 \times 4 \times 2 = 16$ uops, where the steps represent the number of tensor-core sized tiles along each dimension of the per-warp tile.

The figure only illustrates the slices of the A and B operands that a single warp uses per uop. While the A slice is unique per warp, the B slice is shared amongst all warps in the warpgroup. Each uop reads a $tcM \times tcK$ slice of A and a $tcK \times tcN$ slice of B from shared memory, and accumulates the result into the corresponding $tcM \times tcN$ tile of the warp's TMEM region. Thus each TMEM tile is updated twice, once per k-step.

A key design decision is the order in which uops are issued. In WGMMA, the uop sequence uses n-outer ordering

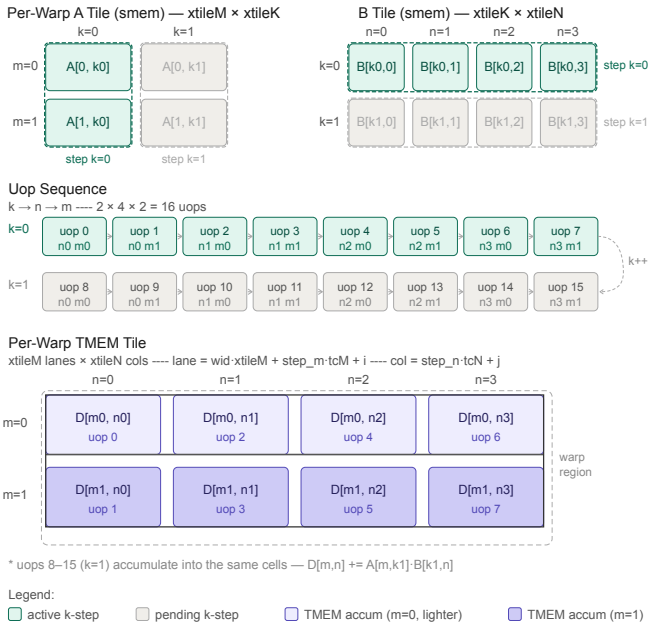

Fig. 2. Uop breakdown of a single warp's UMMA instruction with `xtileM=4`, `xtileN=8`, `xtileK=4` ($tcM = tcN = tcK = 2$). The 16 uops are issued in k-outer order ($k \to n \to m$). Each uop computes one tensor-core-sized tile of the per-warp TMEM accumulator using a $tcM \times tcK$ slice of A and a $tcK \times tcN$ slice of B from shared memory. The A slice is unique per warp; the B slice is shared across all warps in the warpgroup.

($n \to k \to m$), which is compatible with register scoreboard tracking because consecutive uops with the same (n, m) but different k are spaced far enough apart that the FEDP pipeline completes before the next read of the same accumulator register. However, TMEM has no scoreboard, meaning the hardware has no mechanism to detect when a TMEM location has a pending write.

For UMMA, we instead use k-outer ordering ($k \to n \to m$): all uops with $k = 0$ fire first, iterating over all (n, m) combinations before any $k = 1$ uop is issued. This is illustrated in Figure 2. The first uop to revisit a previously written TMEM location (i.e. the same m and n with different k), is therefore separated from the original write by `n_steps` × `m_steps` cycles. This separation must be at least as large as the FEDP pipeline latency to guarantee the $k = 0$ write has completed before the $k = 1$ read. For any reasonable configuration, this separation is comfortably larger than the FEDP pipe latency, eliminating the RAW hazard without any stall logic.

3) FEDP Pipeline Integration: In `VX_tcu_core.sv`, UMMA modifies the `c_val` read within the FEDP grid. For output element (i, j), the accumulator is sourced from TMEM rather than the register file:

```
wire [31:0] c_val = is_umma
  ? tmem_data[wid*xtileM + step_m*tcM + i]
             [step_n*tcN + j]
  : rs3_data[i * TCU_TC_N + j];
```

The `d_val` writeback uses a delay pipeline (`tmem_addr_pipe`) of depth `PIPE_LATENCY` to carry the write address through the FEDP stages alongside the data. When `tmem_wr_en` fires after `PIPE_LATENCY` cycles, `VX_tcu_unit` applies the write to `tmem_data`.

TMEM management instructions (`ALLOC`, `DEALLOC`, `ST`, `LD`) are handled entirely in `VX_tcu_unit` via a bypass path that routes them around `VX_tcu_core`. This avoids polluting the FEDP pipeline with non-compute instructions and allows these operations to complete in a single cycle.

4) RAW Hazard Analysis: The fundamental challenge in UMMA is that TMEM, unlike the register file, is not tracked by the scoreboard. A naive implementation might stall all UMMA uops for `PIPE_LATENCY` cycles after each write, effectively allowing a throughput of one uop per `PIPE_LATENCY` cycles. This would be extremely inefficient.

The k-outer loop ordering resolves this without any stall hardware. Since all $k = 0$ uops fire before any $k = 1$ uops, and the number of $k = 0$ uops (`n_steps` × `m_steps`) exceeds `PIPE_LATENCY`, every $k = 0$ write is guaranteed to complete before the corresponding $k = 1$ read.

IV. Evaluation

A. Experimental Setup

All experiments use Vortex RTL simulation (Verilator-based `rtlsim`), which provides cycle-accurate performance measurements. The configuration is a single Vortex core with 4 warps × 4 threads, `XLEN` = 32, using the DPI FEDP implementation.

The benchmark is single-precision GEMM (`sgemm_tcu_tmem` for UMMA, `sgemm_tcu_wg` for WGMMA), with fp16 input and fp32 accumulation. Three matrix size configurations are evaluated, as shown in Table II.

Size	M	N	K
Small	32	32	16
Medium	64	64	32
Large	128	128	64

TABLE II
MATRIX SIZE CONFIGURATIONS.

UMMA is evaluated at `NRC` $\in \{8, 16, 32, 64, 128\}$ and WGMMA at `NRC` $\in \{8, 16, 32\}$. WGMMA is limited to `NRC` ≤ 32 by architectural constraints. The N dimension constraint also limits small matrices to `NRC` ≤ 32 and medium matrices to `NRC` ≤ 64.

Reported metrics are instruction count, cycle count, and IPC. All tests pass correctness verification against a reference CPU implementation with a tolerance of 12 ULP.

B. Effect of Increasing NRC

1) Cycles: Figures 3, 4, and 5 show the performance results for small, medium, and large matrices respectively. For all matrix sizes, UMMA and WGMMA both show consistent cycle reduction as `NRC` increases up to `NRC` = 32. The cycle reduction pattern is approximately the same for both, although the relative gap grows as matrix size decreases, with WGMMA achieving lower cycle counts than UMMA at smaller sizes. This confirms that the overhead of the additional TMEM

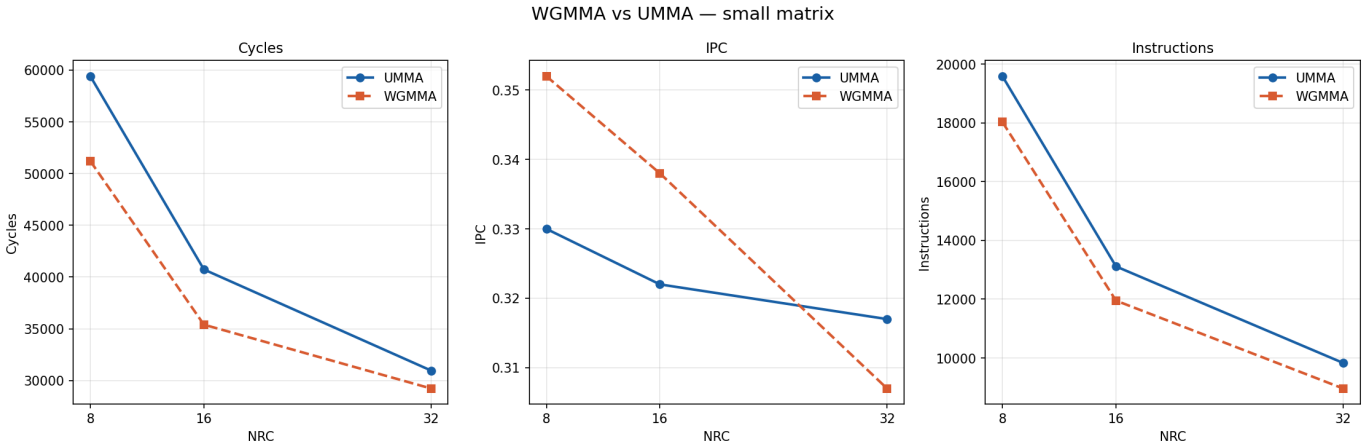

Fig. 3. Performance comparison of WGMMA and UMMA across NRC values for small matrices ($M = 32, N = 32, K = 16$). Left to right: cycle count, IPC, and instruction count.

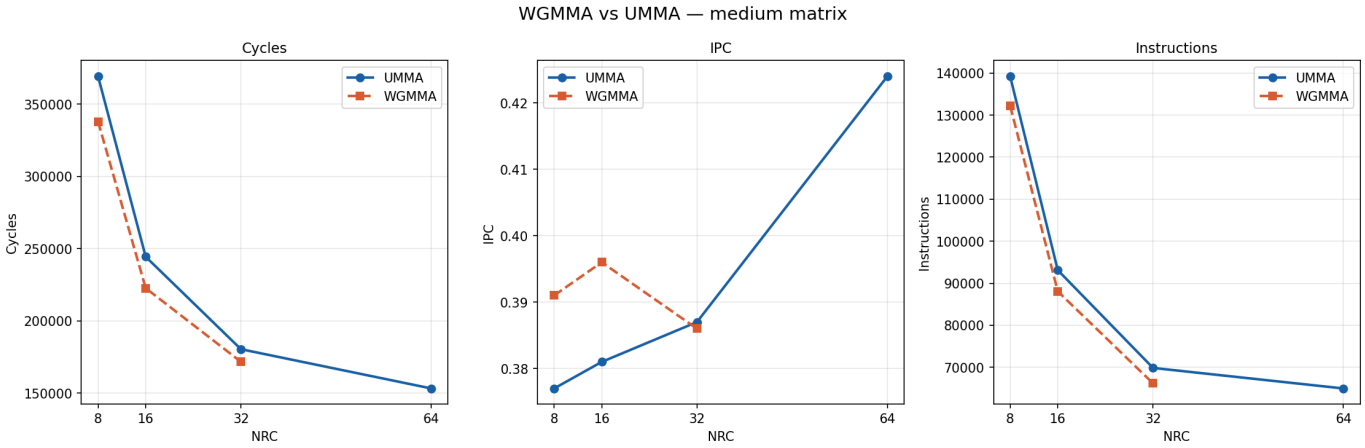

Fig. 4. Performance comparison for medium matrices ($M = 64, N = 64, K = 32$).

management instructions in UMMA is amortized as matrix size increases.

The primary advantage of UMMA is demonstrated at medium and large matrix sizes (Figures 4 and 5). Since UMMA can scale NRC ≥ 32, it achieves lower cycle counts simply by increasing NRC. Each doubling of NRC roughly halves the number of `umma_sync` calls needed to cover the same output tile, proportionally reducing tile buffer fetch overhead and instruction issue overhead. At NRC $= 128$ with large matrices, UMMA achieves an approximately $3\times$ cycle reduction relative to NRC $= 8$, a speedup inaccessible to WGMMA.

2) Instruction Count: UMMA consistently executes more instructions than WGMMA at the same NRC, reflecting the TMEM management overhead. However, as seen in Figures 3–5, the relative gap shrinks as matrix size increases. As NRC grows, the instruction count decreases across all configurations since fewer `umma_sync` calls are needed to cover the same output tile, further demonstrating how larger NRC amortizes per-call overhead.

3) IPC: The IPC plots in Figures 3–5 reveal an interesting matrix-size dependence. For small matrices (Figure 3), IPC decreases as NRC increases. At small sizes the dominant cost is fixed overhead from TMEM management instructions and tile buffer fetch latency. As NRC grows, the B tile fetch covers more data, increasing fetch latency without proportionately increasing useful compute cycles, so the FEDP pipeline spends more time stalled waiting for tile data. The rate of IPC decrease is larger for WGMMA than UMMA at small sizes. One possible explanation is that WGMMA's accumulator register dependencies create additional stall cycles as NRC increases, since larger NRC means more accumulator registers in flight and a higher likelihood of scoreboard conflicts. UMMA, which doesn't have register dependencies, is less sensitive to this effect.

For large matrices (Figure 5), IPC increases with NRC for both approaches. With many tiles to compute, fixed overhead is amortized and the FEDP pipeline stays utilized for longer stretches. The IPC growth rate is greater for UMMA than WGMMA, consistent with WGMMA's register dependencies

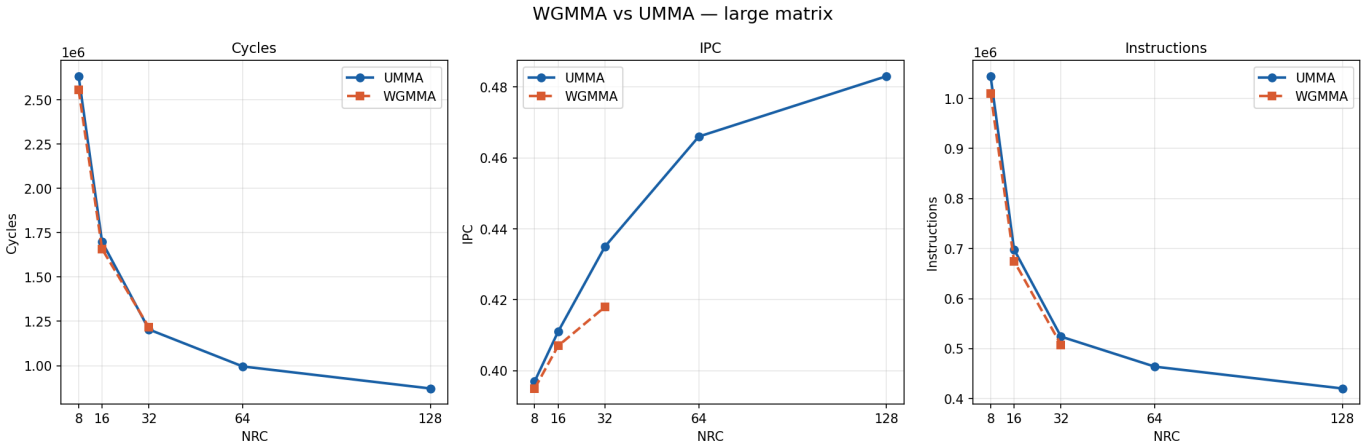

Fig. 5. Performance comparison for large matrices ($M = 128, N = 128, K = 64$).

becoming more constraining at larger NRC. Extrapolating this curve, the relative IPC gap would grow further if WGMMA could support NRC values beyond 32.

For medium matrices (Figure 4), WGMMA shows a non-monotonic IPC trend, rising from NRC $= 8$ to NRC $= 16$ before falling at NRC $= 32$. One possible explanation is that at NRC $= 8$ the tile is small enough that the pipeline frequently idles between tiles. At NRC $= 16$ the tile is large enough to sustain good utilization. At NRC $= 32$, however, the accumulator occupies all available floating-point registers, increasing the frequency of scoreboard stalls and causing IPC to fall despite the larger tile size. UMMA does not exhibit this behavior, with IPC increasing consistently.

C. Scalability Analysis

Figure 6 compares WGMMA and UMMA at NRC $= 32$ (the largest value that both can support) across the matrix sizes. Both operations show roughly the same pattern for cycle and instruction counts, with both increasing at approximately the same rate as matrix size increases. IPC also improves with matrix size for both operations, although UMMA achieves consistently greater IPC, with the gap between UMMA and WGMMA being the lowest at medium matrix size. The gap widens at large and small matrix sizes. This shows that with NRC $= 32$, the workload hits a sweet spot at medium matrix size where the matrix is large enough that fixed overhead is well amortized, but not so large that the sustained compute pressure exposes the difference in register pressure between UMMA and WGMMA.

V. Conclusion

This work implements Tensor Memory (TMEM) and UMMA in the Vortex open-source GPGPU, following the architectural design introduced by NVIDIA Blackwell. The source code is available on Github [8]. The key contributions are:

- A five-instruction RISC-V ISA extension encoded under the RISC-V CUSTOM0 opcode
- A configurable TMEM storage array in RTL, with per-warp addressing and single-cycle read/write access
- A corresponding TMEM implementation in SimX, the cycle-approximate C++ functional simulator for Vortex
- k-outer uop sequencing that eliminates TMEM RAW hazards without stall logic, exploiting the natural separation between k-steps to guarantee hazard-free execution
- Tile buffer modifications to support UMMA's larger B tiles across NRC values up to 128
- A kernel API (`vx_tensor.h`) that abstracts TMEM management from the programmer

RTL simulation demonstrates that UMMA roughly matches WGMMA performance with NRC up to 32, with greater benefits as matrix size increases. UMMA performance further improves as NRC increases to 128, achieving an approximately $3\times$ cycle reduction for large matrices from NRC $= 8$ to NRC $= 128$. The highest pipeline utilization (IPC) is achieved by UMMA with NRC $= 128$ and large matrix sizes, where the FEDP pipeline remains consistently occupied and TMEM overhead is fully amortized.

A. Limitations and Future Work

The current TMEM implementation in RTL uses flip-flops rather than SRAM. A production implementation would replace the FF array with an SRAM module, introducing some cycles of SRAM read latency. Additionally, Blackwell's tensor memory supports storing A operand tiles directly in TMEM, as well as the ability for multiple warpgroups to share a single TMEM instance for pipelined execution. This enables multiple warpgroups on a single SM to issue UMMA instructions concurrently, hiding latency across warpgroups. This is not currently possible on Vortex, as its current implementation requires that only one warpgroup is active at a time per SM. Vortex does not pipeline execution across multiple CTAs on a single SM, and its shared memory has restrictions on how many CTAs can be active at a time.

1) Sparse UMMA: Vortex's existing WGMMA implementation supports sparse A operands via the

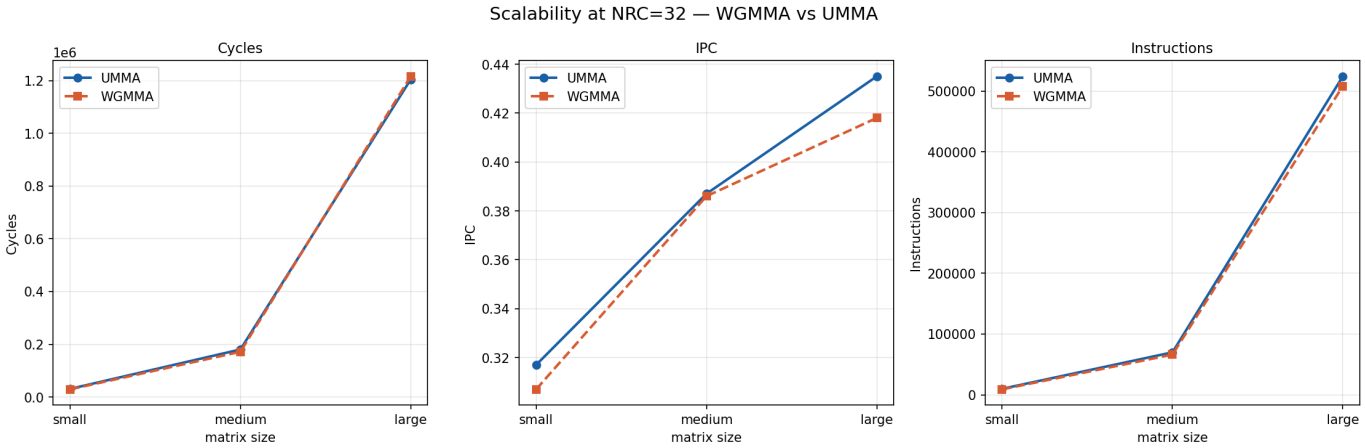

Fig. 6. Scalability comparison of WGMMA and UMMA at `NRC=32` across matrix sizes.

`TCU_SPARSE_ENABLE` extension. In sparse WGMMA, A is stored in a compressed $2:4$ format where only two out of every four elements are non-zero, and a metadata register encodes which elements are non-zero. The tile buffer gather and FEDP pipeline are modified to select the appropriate B elements based on the sparsity pattern, effectively halving the number of multiply-accumulate operations.

The natural next step is to extend sparsity support to UMMA. The core FEDP pipeline would not require significant modification; the primary work would be integrating sparse metadata handling into the UMMA uop path and ensuring the tile buffer gather correctly handles sparse B column selection for UMMA's larger tile sizes. Since UMMA's accumulator lives in TMEM rather than registers, sparse UMMA would preserve the register pressure advantage of TMEM while also benefiting from the $2\times$ throughput improvement of sparsity.

2) Increased K-Steps: In the current implementation, `k_steps` is fixed at 2 for both WGMMA and UMMA. For WGMMA this is a deliberate choice, as each additional k-step consumes more accumulator registers, and `k_steps` $= 2$ balances register pressure against the number of uops per `wgmma_sync` call. For UMMA, however, there is no such constraint, since the accumulator lives in TMEM and additional k-steps consume no additional registers.

Increasing `k_steps` beyond 2 for UMMA would allow each `umma_sync` call to cover more of the K dimension in a single instruction, further amortizing the fixed cost of tile buffer fetches and TMEM management instructions. An interesting trade-off to evaluate is whether larger `k_steps` provides meaningful cycle reduction beyond what increasing `NRC` already achieves.

3) Power and Area Analysis: A complete hardware cost analysis of the TMEM implementation was not performed due to tool availability constraints preventing Yosys synthesis from completing. The dominant area contributor is the TMEM storage array, which in the current RTL implementation uses a substantial number of flip-flops that would be replaced by SRAM in a production design.

Future work includes running Yosys synthesis on the full Vortex core with and without `TCU_TMEM_ENABLE` to quantify the logic area delta, and estimating the SRAM area of the TMEM array. SAIF-annotated power analysis from RTL simulation would allow a further comparison of energy per GEMM operation between WGMMA and UMMA, which is interesting given that UMMA reduces cycle counts significantly for large matrices, potentially offsetting the additional power of the TMEM array.

References

[1] NVIDIA Corporation, "NVIDIA Tesla V100 GPU Architecture," NVIDIA, Tech. Rep., 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[2] ——, "NVIDIA Turing GPU Architecture," NVIDIA, Tech. Rep., 2018. [Online]. Available: https://images.nvidia.com/akamai/technology/turing/nvidia-turing-architecture-whitepaper.pdf

[3] ——, "NVIDIA Ampere GA102 GPU Architecture," NVIDIA, Tech. Rep., 2021. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga102-gpu-architecture-whitepaper-v2.pdf

[4] ——, "NVIDIA H100 Tensor Core GPU Architecture," NVIDIA, Tech. Rep., 2022. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

[5] ——, "NVIDIA Blackwell Architecture Technical Brief," NVIDIA, Tech. Rep., 2024. [Online]. Available: https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/

[6] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and H. Kim, "Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021.

[7] NVIDIA Corporation, *PTX ISA Reference Manual*, 2024. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/

[8] E. Yoon, https://github.com/eyoon1131/vortex/tree/tensor_mem, 2026.