ACCELERATING FUSED WMMA OPERATIONS VIA IN-REGISTER DATA PERMUTATION

Tommy Morales
ENSIIE
Évry, France
tommy.morales@ensiie.eu

ABSTRACT

The performance of GPU kernels for AI, such as FlashAttention, is increasingly limited by on-chip data movement after achieving I/O-awareness. A critical bottleneck in fused operations using the WMMA (Warp-Level Matrix-Multiply-Accumulate) API is the costly round-trip to shared memory required for post-matmul reductions, which breaks data locality and introduces synchronization overhead. To address this, we propose a novel technique that performs reductions entirely within the GPU's register file. By reverse-engineering the swizzled data layout of WMMA accumulator fragments, we devised a set of in-register permutations that realign data for efficient intra-warp reduction using shfl_sync instructions. A proof-of-concept implementation on a fused matrix-multiply-max operation demonstrates up to a 7x speedup across modern NVIDIA architectures (Blackwell and Hopper) over the standard shared memory-based approach, and a 1.90x speed-up for an end-to-end implementation on a Flash-Attention-1 style kernel compared to a conventional shared memory-based implementation. These results indicate that direct manipulation of fragment data layouts is a powerful new vector for micro-architectural optimization beyond the current state-of-the-art.

1 Introduction and Motivation

Attention-based Transformer architectures remain the dominant choice for Large Language Models (LLMs). While variations like Grouped-Query Attention (GQA) have emerged, most designs stem from the Multi-Head Attention (MHA) mechanism introduced in the seminal "Attention Is All You Need" paper[1]. Optimizing the computational cost of the attention forward pass is critical for both training and inference. To address this, I/O-aware algorithms have become the standard since the introduction of FlashAttention, which minimizes expensive data transfers between GPU high-bandwidth memory (HBM) and on-chip SRAM.

With state-of-the-art implementations like FlashAttention achieving near-optimal occupancy and effectively eliminating the I/O bottleneck, performance improvements now depend on on-chip compute efficiency. This shifts focus to optimizing calculations performed by Tensor Cores via the WMMA (Warp-Level Matrix-Multiply-Accumulate) API [2], a set of CUDA primitives for executing efficient matrix operations directly on Tensor Core hardware. However, a common pattern in fused kernels-performing a matrix multiplication followed by a reduction or element-wise operation (e.g., softmax)-introduces a new bottleneck: the data round-trip through shared memory. Current practice requires storing the accumulator fragment from registers to shared memory using `wmma::store_matrix_sync`, synchronizing via `__syncthreads()`, and then reloading the data into registers for further processing. This approach incurs significant latency (see Section 4.3) and breaks data locality, forcing subsequent operations to re-fetch data from the slower shared memory cache instead of keeping it within the register file.

In this paper, we propose a technique to bypass this shared memory round-trip by performing post-matmul operations directly in registers. We first detail our analysis of the WMMA fragment's internal data layout, revealing the mapping between matrix elements and specific thread registers. We then exploit this knowledge to develop an in-register data permutation algorithm. This allows us to realign data for efficient parallel reduction using `shfl_sync` instructions. Our benchmarks, conducted on a fused matrix-multiply-max kernel, a core operation in algorithms like FlashAttention,

demonstrate a 7x speedup compared to the standard shared memory-based approach as well as a **1.90x end-to-end speedup** on a Flash Attention 1-style kernel compared to a conventional shared memory-based implementation, validating the significant real-world impact of our approach.

Specifically, this paper makes four key contributions. First, we present an experimental methodology for reverse-engineering the WMMA data layout. Second, we introduce a novel in-register permutation algorithm to eliminate shared memory round-trips. Third, we validate this technique with a measured 1.90x end-to-end speedup on a complete FlashAttention-1 style kernel. Finally, we provide a full open-source implementation to ensure reproducibility and provide a foundation for future work.

The primary limitation of this method is its inherent dependency on architecture-specific data layouts, sacrificing portability for maximum performance. Nevertheless, this work establishes a new, generalizable optimization pattern for fused GPU kernels: **Matmul** $\rightarrow$ **In-Register Row-wise Operation**.

This pattern is not limited to attention. It could be applied to other critical AI operations such as max-pooling, layer normalization, or certain loss function calculations that follow a similar structure of a matrix multiplication followed by a row-wise reduction or element-wise operation. Exploring these applications constitutes a promising direction for future work and could yield significant performance improvements across a wider range of AI models. This paper, along with a full open-source implementation for reproducibility, provides the foundation for this exploration.

2 Related Work

The paradigm for sequence modeling was fundamentally altered by [1] with the introduction of the Transformer architecture. By replacing recurrent neural networks (RNNs) with a purely attention-based mechanism, their work overcame the limitations of sequential processing and enabled massive parallelization on modern hardware like GPUs. The introduction of Multi-Head Attention (MHA) allowed the model to jointly attend to information from different representation subspaces at different positions, forming the computational core of most modern Large Language Models (LLMs).

While the Transformer architecture is highly parallelizable, its performance on physical hardware is overwhelmingly dominated by data movement, not arithmetic complexity. This reflects a broader shift in high-performance computing, where the cost of fetching data from off-chip DRAM is orders of magnitude higher than a floating-point operation, rendering classical complexity analysis (e.g., [3]) less relevant for practical performance. This performance characteristic is well-described by the Roofline model, which shows that for many modern workloads, performance is limited by memory bandwidth ('memory-bound') rather than raw computational throughput ('compute-bound'). This has led to the development of **I/O-aware algorithms**. The seminal work in this area for Transformers is FlashAttention by [4], which redesigns the attention algorithm to minimize data transfers between the GPU's High-Bandwidth Memory (HBM), its on-chip SRAM, and the register file. By using techniques such as kernel fusion and tiling, FlashAttention computes the exact attention output with significantly fewer memory reads/writes. Subsequent versions [5] [6] have further refined this approach

With the I/O bottleneck largely addressed by these state-of-the-art methods [5, 6], the new performance frontier lies in optimizing computation and data movement *on-chip*. Modern NVIDIA GPUs provide specialized hardware units, Tensor Cores, to accelerate matrix multiplications. These are typically accessed via high-level abstractions like the WMMA (Warp-Level Matrix-Multiply-Accumulate) API [2]. However, while powerful, the WMMA API treats the physical layout of data within the GPU's register file as an opaque "black box". This abstraction, designed for ease of use, can introduce subtle but significant on-chip bottlenecks, such as the costly round-trip to shared memory for post-matmul operations. This paper addresses this specific on-chip bottleneck, exploring optimizations that lie beyond the current SOTA by manipulating the fragment data layout directly.

It is important to note that I/O-aware attention is not a monolithic field. While FlashAttention and its derivatives optimize the standard Multi-Head/Grouped-Query Attention (MHA/GQA) mechanism, a parallel and architecturally distinct approach has emerged with Multi-Head Latent Attention (MLA), introduced by [7]. Instead of optimizing the computation order, MLA fundamentally alters the architecture by compressing the Key-Value cache itself into a low-rank latent representation. Recent work such as TransMLA [8] has even demonstrated methods for converting pre-trained GQA models to the MLA format. While our direct application focuses on the FlashAttention-style softmax, the underlying technique of in-register data manipulation is a kernel-level optimization that addresses a bottleneck fundamental to any algorithm leveraging the WMMA API, including potential future MLA implementations.

The remainder of this paper is structured as follows. Section 2 reviews the state-of-the-art in attention optimization. Section 3 details our reverse-engineering methodology for the WMMA fragment layout and presents our novel in-register permutation algorithm. Section 4 provides a comprehensive experimental validation of our technique across

two GPU architectures. Finally, Section 5 demonstrates how our method can be integrated as a core component within a full FlashAttention kernel, and tests one such integration on a Flash Attention 1-style kernel.

3 WMMA Fragments Manipulation

Fragments are the form in which the WMMA API stores matrices to perform operations - (mainly Matrix Multiplications via `mma_synch()` - on tensor cores. In order to manipulate and reorganize the data layout, reverse engineering this black box was an essential first step.

3.1 Extracting the data layout

To manipulate data within a WMMA fragment, it is imperative to understand how the 256 elements of a 16x16 matrix are distributed across the 8 registers of each of the 32 threads in a warp. As this mapping is undocumented, we developed a micro-kernel to chart it experimentally. The principle is to create a "tracer matrix" where the value of each element encodes its own (row, col) coordinate. This matrix is then passed through the WMMA pipeline via an identity multiplication, preserving the tracer values. The resulting fragment is dumped to global memory, allowing a host-side script to reconstruct the mapping. The overall process is described in Algorithm 1.

The principle, described in Algorithm 1, is to initialize a 16x16 'tracer matrix' in shared memory where each element's value encodes its own coordinates. After passing this matrix through an identity WMMA multiplication, the resulting fragment is dumped to global memory. A host-side script can then reconstruct the mapping.

Algorithm 1 WMMA Fragment Layout Reverse-Engineering

Input: None
Output: A 16x16 table Mapping[row][col] = (lane, frag_idx)
procedure DEVICEKERNEL(output_buffer)
    tracer_matrix ← 16x16 shared memory array
    lane ← thread's lane index (0-31)
    // Each thread in the block populates one element of the tracer matrix
    tracer_matrix[threadIdx.y][threadIdx.x] ← threadIdx.y * 100 + threadIdx.x
    `__syncthreads()`
    if lane < 32 **then** ▷ Code executed by a single warp
        $a\_frag \leftarrow$ `wmma::load_matrix_sync(tracer_matrix)`
        $id\_frag \leftarrow$ Load Identity Matrix into a fragment
        $acc\_frag \leftarrow$ `wmma::mma_sync(a_frag, id_frag)` ▷ Fragment now holds tracer values
        for $i \leftarrow 0$ to 7 **do** ▷ Dump fragment registers to global memory
            output_buffer[lane * 8 + i] ← acc_frag.x[i]
        end for
    end if
end procedure
procedure HOSTANALYSIS(output_buffer)
    for $lane \leftarrow 0$ **to** 31 **do**
        for $frag\_idx \leftarrow 0$ **to** 7 **do**
            $value \leftarrow output\_buffer[lane * 8 + frag\_idx]$
            $row \leftarrow \lfloor value/100 \rfloor$
            $col \leftarrow value\%100$
            $Mapping[row][col] \leftarrow (lane, frag\_idx)$
        end for
    end for
    return $Mapping$
end procedure

The mapping we get is detailed in Figure 1.

```
      C00  C01  C02  C03  C04  C05  C06  C07 | C08  C09  C10  C11  C12  C13  C14  C15
      ------------------------------------+------------------------------------
R00 |   0    0    1    1    2    2    3    3 |   0    0    1    1    2    2    3    3
R01 |   4    4    5    5    6    6    7    7 |   4    4    5    5    6    6    7    7
R02 |   8    8    9    9   10   10   11   11 |   8    8    9    9   10   10   11   11
R03 |  12   12   13   13   14   14   15   15 |  12   12   13   13   14   14   15   15
R04 |  16   16   17   17   18   18   19   19 |  16   16   17   17   18   18   19   19
R05 |  20   20   21   21   22   22   23   23 |  20   20   21   21   22   22   23   23
R06 |  24   24   25   25   26   26   27   27 |  24   24   25   25   26   26   27   27
R07 |  28   28   29   29   30   30   31   31 |  28   28   29   29   30   30   31   31
----+------------------------------------+------------------------------------
R08 |   0    0    1    1    2    2    3    3 |   0    0    1    1    2    2    3    3
R09 |   4    4    5    5    6    6    7    7 |   4    4    5    5    6    6    7    7
... |  ...                                   |  ...
```

Figure 1: Experimentally discovered data layout on an RTX 5080 GPU. The mapping of $lane$ to matrix coordinates exhibits a repeating pattern every 8 rows and 8 columns. Each cell indicates the thread $lane$ ID that holds the corresponding matrix element. For example, the element at (R00, C02) is stored in a register of thread lane 1

3.2 Reorganising the data

While a full de-swizzle to a linear memory layout might seem intuitive, our analysis shows it is suboptimal for two key reasons. First, it would require a complex sequence of multiple `shfl_sync` instructions to move data across distant threads within the warp, incurring significant instruction overhead. Second, the goal is not to linearize the data, but to consolidate elements from the same matrix row for reduction. A more cycle-efficient solution consists of a partial in-register permutation combined with carefully constructed masks for the reduction phase.

As conceptually illustrated in Figure 2, a minimal swap operation de-interleaves the data within each thread's registers. This consolidates all elements belonging to a single logical matrix row into contiguous registers within a thread sub-group, making the data ready for an efficient parallel reduction. This permutation, requiring only a few clock cycles, is detailed in Section 3.3.

However, in a complete kernel implementation like FlashAttention, it is often necessary to store thread-local accumulators (e.g., the O_accum matrix) back to shared memory and later on HBM. For this store operation to be correct and efficient, we must use the forward mapping formulas derived from our reverse-engineering analysis. These formulas allow us to calculate the correct global memory address for each element stored in a thread's registers.

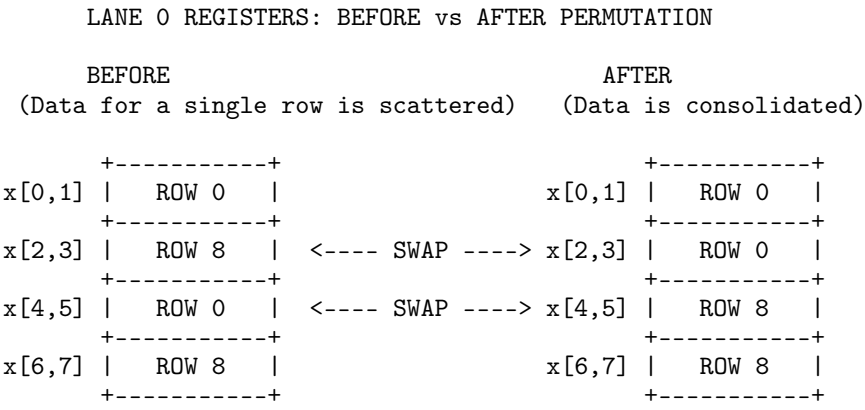

Figure 2: Conceptual view of the in-register permutation for a single thread (e.g., lane=0). The swap operation reconsolidates scattered data segments from a single logical matrix row (e.g., ROW 0) into contiguous registers, making subsequent parallel reduction feasible.

The mapping from a thread's lane ID ($lane$, where $lane \in [0, 31]$) and its register index (i, where $i \in [0, 7]$) to a matrix coordinate (row, col) can be described by the following formulas:

$$\text{row} = (lane/4) + (((i \gg 1)\&1) \times 8) \tag{1}$$

$$\text{col} = ((lane\%4) \times 2) + (i\%2) + ((i \gg 2) \times 8) \tag{2}$$

Algorithm 2 describes how these formulas are used to write a thread-local array ($local\_data$) to its correct, swizzled position in a 2D global output matrix.

Algorithm 2 Storing Thread-Local Data using Forward Mapping

Input: A thread-local array local_data[0..7], global output matrix Out
Context: Executed by each thread $lane$ in a warp
$block\_row\_start \leftarrow$ Starting row index of the output tile in Out
$block\_col\_start \leftarrow$ Starting col index of the output tile in Out
for $i \leftarrow 0$ to 7 **do**
    // Calculate target coordinate using forward mapping formulas
    $target\_row \leftarrow block\_row\_start + (lane/4) + (((i \gg 1)\&1) \times 8)$
    $target\_col \leftarrow block\_col\_start + ((lane\%4) \times 2) + (i\%2) + ((i \gg 2) \times 8)$
    // Write the local data to the correct global memory address
    $Out[target\_row][target\_col] \leftarrow local\_data[i]$
end for

3.3 In-Register Parallel Reduction

Following the in-register permutation, the data is logically aligned for efficient parallel processing. All data elements for two distinct matrix rows are now held within a single 4-thread sub-group (a "quad"). This enables any associative row-wise reduction (e.g., max, sum) to be performed entirely in registers using `shfl_xor_sync` instructions.

The process involves two stages, as described in Algorithm 3. First, each thread computes a local reduction on the data held in its registers. For example, after permutation, a thread's registers x[0]-x[3] contain all its data for the first set of 8 rows, and x[4]-x[7] for the second. The second stage is an intra-quad parallel reduction, where the 4 threads of a sub-group cooperatively reduce their local values. This is achieved using `shfl_xor_sync` with a dynamic mask to constrain the communication within the quad.

Algorithm 3 Generic In-Register Parallel Reduction

Input: A permuted accumulator fragment `acc_frag`, an associative binary operator $\oplus$
Output: For each quad, two reduced values `val_A` and `val_B`
Context: Executed by each thread $lane$ in a warp
// Stage 1: Local reduction within each thread's registers
$local\_A \leftarrow$ identity_element
for $k \leftarrow 0$ to 3 **do**
    $local\_A \leftarrow local\_A \oplus acc\_frag.x[k]$
end for
$local\_B \leftarrow$ identity_element
for $k \leftarrow 4$ to 7 **do**
    $local\_B \leftarrow local\_B \oplus acc\_frag.x[k]$
end for
// Stage 2: Parallel reduction within each 4-thread quad
$mask \leftarrow 0xF << ((lane/4) * 4)$ ▷ e.g., 0xF for quad 0, 0xF0 for quad 1
$partner\_A \leftarrow$ `__shfl_xor_sync(mask, local_A, 1)`
$val\_A \leftarrow local\_A \oplus partner\_A$
$partner\_A \leftarrow$ `__shfl_xor_sync(mask, val_A, 2)`
$val\_A \leftarrow val\_A \oplus partner\_A$
// Repeat for B
$partner\_B \leftarrow$ `__shfl_xor_sync(mask, local_B, 1)`
$val\_B \leftarrow local\_B \oplus partner\_B$
$partner\_B \leftarrow$ `__shfl_xor_sync(mask, val_B, 2)`
$val\_B \leftarrow val\_B \oplus partner\_B$

if $lane\%4 = 0$ **then** ▷ Lead thread of the quad now holds the final reduced values
    return val_A, val_B
end if

The mask $0xF << ((lane/4) * 4)$ dynamically creates a 4-bit window corresponding to the thread's sub-group (e.g., $0x0000000F$ for threads 0-3, $0x000000F0$ for threads 4-7). This allows all 8 sub-groups to perform their reductions in parallel without interfering with one another, achieving maximum data-level parallelism.

4 Experimental Validation

To quantify the performance benefits of our in-register technique, we conducted a micro-benchmark comparing it against the standard shared memory-based approach.

4.1 Setup

All experiments were performed on an NVIDIA RTX 5080 GPU (Blackwell architecture) and an NVIDIA H100 (Hopper architecture). The key technical specifications and software versions are detailed in Table 1.

Table 1: Experimental Setup Details.

Component	Configuration
Operating System	Ubuntu 24.04.3 LTS
CUDA Toolkit	12.9
NVIDIA Driver	575.57.08

Initial attempts looping over kernel launches and using an internal loop within the CUDA kernel were found to be unreliable due to launch overhead and aggressive compiler optimizations (Dead Code Elimination), respectively.

To mitigate these issues, we adopted a script-driven methodology. For runtime measurements, an external `bash` script launches the executable, which performs a single kernel execution hundreds of times. The script discards the initial warm-up runs and averages the subsequent results, providing a stable measurement free from both compiler interference and cold-start anomalies. A similar approach was used for power measurement, where a sustained load was generated by the script to allow for stable readings using `nvidia-smi`. All benchmarking scripts are provided in this public repository to ensure full reproducibility[1].

4.2 Kernels Under Test

- **Baseline Kernel (Shared Memory):** This kernel represents the standard, textbook approach. It performs the matrix multiplication using `wmma::mma_sync`, stores the full accumulator fragment to a `__shared__` memory array via `wmma::store_matrix_sync`, synchronizes the warp with `__syncthreads()`, and finally has each thread in the warp read its assigned row from shared memory to compute the maximum value.
- **Proposed Kernel (In-Register):** This kernel implements our technique. After the `wmma::mma_sync`, it performs the in-register permutation and subsequent intra-quad parallel reduction as described in Algorithm 3. The final result is written directly to global memory, completely bypassing the shared memory round-trip.

4.3 Results

Our experiments reveal a significant and robust advantage for the proposed in-register technique, demonstrating consistency across two distinct state-of-the-art GPU architectures.

The runtime measurements, presented in Table 2, show a stable and substantial speedup of approximately 7x on both NVIDIA's Hopper and Blackwell architectures. This consistency validates the fundamental nature of our optimization, proving its effectiveness beyond a single hardware generation. The shared memory round-trip constitutes a major bottleneck on both platforms, and its elimination yields a massive performance gain in latency-bound scenarios.

To understand the source of this acceleration, a micro-architectural analysis was conducted under a high-occupancy workload to simulate a realistic, throughput-bound scenario (Table 3). This analysis confirms the speedup is not due to a change in occupancy, which is nearly identical for both kernels as it is limited by register count, but is instead rooted in the complete elimination of on-chip data movement and synchronization overhead. Our kernel reduces the shared memory footprint and the number of synchronization barriers to zero. It is noteworthy that while runtime gains are

[1] `https://github.com/Z6BQw2/in-register-wmma-poc`

Table 2: Performance Comparison Across GPU Architectures.

Architecture	Kernel	Runtime (ms)	Speedup
NVIDIA H100 (Hopper)	Baseline (Shared Mem)	0.216	1.0x
	In-Register (Ours)	**0.03**	**7x**
NVIDIA RTX 5080 (Blackwell)	Baseline (Shared Mem)	0.0825	1.0x
	In-Register (Ours)	**0.0127**	**6.5x**

dramatic, throughput under high occupancy is comparable. This illustrates a core principle of GPU architecture: latency hiding. While the GPU's scheduler can effectively hide the baseline kernel's shared memory latency by switching to other active warps, our technique fundamentally removes this latency source. This provides greater scheduling flexibility and reduces on-chip power draw, which directly translates to the significant performance improvements observed in our latency-bound benchmarks.

Table 3: Micro-architectural Analysis under High Occupancy. Occupancy data from Nsight Compute, other metrics from code analysis and `ptxas` compiler output.

Metric	Architecture	Baseline Kernel	In-Register Kernel
Achieved Occupancy (%)	H100	24.50	25.07
	RTX 5080	44.10	43.77
Registers per Thread	H100 & 5080	22	**22**
Shared Memory Footprint	H100 & 5080	1024 bytes	**0 bytes (-100%)**
Synchronization Barriers	H100 & 5080	1	**0 (-100%)**

Table 4: Average Power Consumption Under Sustained Load, measured with `nvidia-smi`.

Architecture	Baseline (W)	In-Register (W)
NVIDIA H100 (Hopper)	85.44	85.57
NVIDIA RTX 5080 (Blackwell)	61.06	61.55

Finally, the energy efficiency gains detailed in Table 5 directly mirror the performance improvements. As shown in Table 4, the average power draw under a sustained load is nearly identical for both kernel versions. By completing the operation significantly faster at this similar power draw, our in-register method consumes approximately 7x less energy per operation. This demonstrates that the technique not only accelerates computation but does so with a massive improvement in performance-per-watt, a critical consideration for large-scale AI workloads.

Table 5: Energy Efficiency Comparison. Energy/Op = Avg. Power (W) $\times$ Runtime (s).

Architecture	Kernel	Energy/Op (µJ)	Efficiency Gain
NVIDIA H100	Baseline (Shared Mem)	18.45	1.0x
	In-Register (Ours)	**2.57**	**7.0x**
NVIDIA RTX 5080	Baseline (Shared Mem)	5.04	1.0x
	In-Register (Ours)	**0.78**	**6.5x**

5 Application to a Full FlashAttention Kernel

The true potential of our technique is realized when it is integrated as the core softmax engine within a full FlashAttention kernel. The standard implementation of the softmax step within FlashAttention's main loop relies on the same shared memory round-trip that our micro-benchmark identified as a bottleneck. Our in-register method can replace this entire sequence.

Algorithm 4 details the complete, fused softmax calculation performed directly on the accumulator fragment `work_frag` resulting from the $S_{ij} = Q_i K_j^T$ matrix multiplication. The algorithm first applies the necessary scaling and permutation,

then calculates the row-wise max, subtracts it, computes the exponentials, calculates the row-wise sum, and finally normalizes to get the probability matrix P_{ij}. All intermediate steps, including the update of running statistics, are performed entirely within registers, maximizing data locality and eliminating all `__syncthreads()` stalls from the softmax computation.

Algorithm 4 In-Register Fused Softmax for FlashAttention Tile

Input: Accumulator fragment `work_frag` (containing S_{ij}), previous stats m_{old}, l_{old}
Output: Probability fragment `p_frag` (containing P_{ij}), updated stats m_{new}, l_{new}
Context: Executed by each thread $lane$ in a warp after the $Q_i K_j^T$ `mma_sync`
// 1. Scale and Permute
`work_frag` $\leftarrow$ `work_frag` / $\sqrt{d_k}$
Apply in-register permutation to `work_frag` (as per Section 2.2)
// 2. Compute current tile's row-wise max (m_{ij})
$m_{local\_A}, m_{local\_B} \leftarrow$ Perform in-register Max reduction (Algorithm 3) on `work_frag`
// 3. Update global row-wise max
$m_{new} \leftarrow \max(m_{old}, m_{ij})$
$l_{old\_rescaled} \leftarrow l_{old} \times \exp(m_{old} - m_{new})$
// 4. Compute exponentials and current tile's row-wise sum (l_{ij})
`work_frag` $\leftarrow \exp($`work_frag` $- m_{new})$
$l_{local\_A}, l_{local\_B} \leftarrow$ Perform in-register Sum reduction (Algorithm 3) on `work_frag`
// 5. Update global sum normalizer
$l_{new} \leftarrow l_{old\_rescaled} + l_{ij}$
// 6. Final normalization to get P_ij
`p_frag` $\leftarrow$ `work_frag` / l_new
Apply inverse permutation to `p_frag` to prepare for $P_{ij}V_j$ multiplication.

5.1 Validation on a FlashAttention-1 Style Kernel

To validate the end-to-end impact of our technique, we integrated it as the softmax engine within a complete, custom FlashAttention kernel. Our implementation is modeled after the original FlashAttention (FA-1) architecture, featuring the core principles of tiling and kernel fusion, including an optimized K-matrix pipeline. This provides a realistic and well-understood testbed. We then created a baseline version of this same kernel, replacing our in-register method with a standard, optimized shared memory approach.

The end-to-end benchmark results, presented in Table 6, demonstrate a measured speedup of **1.90x** for our in-register technique over the conventional shared memory implementation.

Table 6: End-to-End Performance on a FlashAttention-1 Style Kernel (RTX 5080).

Kernel Version	**Runtime (ms)**
Baseline (Shared Memory)	4.7785
In-Register (Ours)	**2.5129**
Speedup	**1.90x**

This result confirms that our technique provides a significant performance improvement on a complex, fused kernel architecture. While this proof-of-concept does not include the more advanced work scheduling and partitioning optimizations of later publications like FlashAttention-2/3, it serves as a robust validation of the in-register permutation technique's fundamental impact.
The kernels are available on GitHub[2].

6 Conclusion and Future Work

In this work, we identified and addressed a critical on-chip bottleneck in fused WMMA operations: the costly data round-trip through shared memory for post-matmul reductions. We have demonstrated that by reverse-engineering

[2] `https://github.com/Z6BQw2/in-register-wmma-poc`

the opaque data layout of WMMA fragments, it is possible to design a novel in-register permutation algorithm that completely bypasses this bottleneck. Our experimental results are conclusive. On a key micro-benchmark, our technique yields a speedup of up to 7x. When integrated into a full FlashAttention-1 style kernel, this translates to a **measured 1.90x end-to-end performance gain**, validating its significant real-world impact.

References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, 2017.

[2] NVIDIA Corporation. CUDA C++ Programming Guide: Warp-Level Matrix Operations. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[3] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 1969.

[4] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

[5] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

[6] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[7] DeepSeek-AI. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[8] Fanxu Meng, Pingzhi Tang, Xiaojuan Tang, Zengwei Yao, Xing Sun, and Muhan Zhang. TransMLA: MLA Is All You Need. *arXiv preprint arXiv:2502.07864*, 2025.