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 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 an 11.5x speedup on the Blackwell Architecture over the standard shared memory-based approach on an NVIDIA RTX 5080. This result indicates 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

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. 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 API. 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 and breaks data locality.

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 7.8x speedup compared to the standard shared memory-based approach.

2 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.

2.1 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.

Algorithm 1 WMMA Fragment Layout Reverse-Engineering

Input: None
Output: A 16x16 table Mapping[row][col] = (lane_id, frag_idx)
procedure DEVICEKERNEL(output_buffer)
    tracer_matrix ← 16x16 shared memory array
    lane_id ← 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_id < 32 **then** ▷ Code executed by a single warp
        $a\_frag \leftarrow wmma :: load\_matrix\_sync(tracer\_matrix)$
        $id\_frag \leftarrow LoadIdentityMatrixintoafragment$
        $acc\_frag \leftarrow wmma :: mma\_sync(a\_frag, id\_frag)$ ▷ Fragment now holds tracer values
        for $i \leftarrow 0$ to 7 **do** ▷ Dump fragment registers to global memory
            output_buffer[lane_id * 8 + i] ← acc_frag.x[i]
        end for
    end if
end procedure
procedure HOSTANALYSIS(output_buffer)
    for $lane\_id \leftarrow 0$ **to** 31 **do**
        for $frag\_idx \leftarrow 0$ **to** 7 **do**
            $value \leftarrow output\_buffer[lane\_id * 8 + frag\_idx]$
            $row \leftarrow \lfloor value/100 \rfloor$
            $col \leftarrow value \bmod 100$
            $Mapping[row][col] \leftarrow (lane\_id, frag\_idx)$
        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 $lane\_id$ to matrix coordinates exhibits a repeating pattern every 8 rows and 8 columns.

2.2 Reorganising the data

While a full de-swizzle of the data might seem intuitive, our analysis shows it is suboptimal. 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.

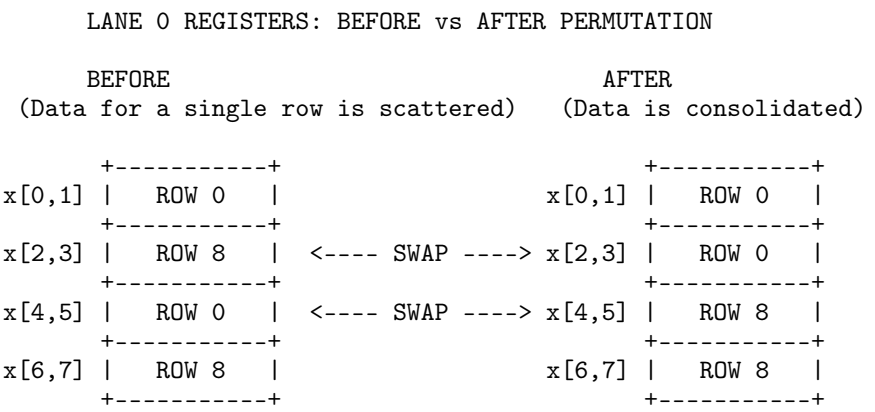

Figure 2: Conceptual view of the in-register permutation for a single thread (e.g., lane_id=0). The swap operation reconsolidates scattered data segments from a single logical matrix row (e.g., ROW 0) into contiguous registers, making subsequent parallel reduction feasible.

The mapping from a thread's lane ID (t, where $t \in [0, 31]$) and its register index (i, where $i \in [0, 7]$) to a matrix coordinate (row, col) can be described by the following formulas:

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

$$\text{col} = ((t \bmod 4) \times 2) + (i \bmod 2) + (((i \gg 2) \& 1) \times 8) \tag{2}$$

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 t 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
    $target\_row \leftarrow block\_row\_start + (t/4) + (((i \gg 1)\&1) \times 8)$
    $target\_col \leftarrow block\_col\_start + ((t \bmod 4) \times 2) + (i \bmod 2) + (((i \gg 2)\&1) \times 8)$
    // Write the local data to the correct global memory address
    $Out[target\_row][target\_col] \leftarrow local\_data[i]$
end for

2.3 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 ‘t‘ 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
$mask \leftarrow 0xF << ((t/4) * 4)$ ▷ 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 $lane \bmod 4 = 0$ **then** ▷ Lead thread of the quad now holds the final reduced values
    return val_A, val_B
end if

The mask $0xF << ((threadIdx.x/4) * 4)$ 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.

3 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.

3.1 Setup

All experiments were performed on a single NVIDIA RTX 5080 GPU with CUDA. Latency was measured using CUDA events over 100 warm-up iterations and 1000 timed iterations, with the average reported. The benchmark task is a fused matrix-multiply-max operation on 16x16 bf16 matrices, a core component of softmax calculations.

3.2 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.

3.3 Results

Our experiments reveal a significant and robust advantage for the proposed in-register technique across all tested metrics and architectures.

The latency measurements, presented in Table 1, show a **6.1x speedup on the H100** and an even more pronounced **11.5x speedup on the RTX 5080**. This suggests that while the shared memory round-trip is a bottleneck on both architectures, it is a more dominant performance limiter on the newer consumer architecture, making our optimization particularly impactful there.

Table 1: Performance Comparison Across GPU Architectures.

Architecture	Kernel	Latency (ms)	Speedup
NVIDIA H100 (Hopper)	Baseline (Shared Mem)	0.1038	1.0x
	In-Register (Ours)	**0.0169**	**6.1x**
NVIDIA RTX 5080 (Blackwell)	Baseline (Shared Mem)	0.1309	1.0x
	In-Register (Ours)	**0.0113**	**11.5x**

To understand the source of this acceleration, a micro-architectural analysis (Table 2) confirms the speedup is not a result of higher occupancy - which remains nearly identical - but is due to the fundamental elimination of shared memory traffic. On both GPUs, our kernel reduces the shared memory footprint and the number of synchronization barriers to zero. The Nsight Compute reports for the baseline kernel explicitly identified shared memory bank conflicts and MIO stalls as the primary performance limiters; issues that are entirely absent in our kernel's profile.

Table 2: Micro-architectural Analysis on Hopper and Blackwell. Data for Occupancy from Nsight Compute, other metrics from `ptxas` compiler output.

Métrique	Architecture	Baseline Kernel	In-Register Kernel
Achieved Occupancy (%)	H100	1.56	1.56
	RTX 5080	2.05	1.97
Registers per Thread	H100 & 5080	23	**21 (-9%)**
Shared Memory Footprint	H100 & 5080	1024 bytes	**0 bytes (-100%)**
Synchronization Barriers	H100 & 5080	1	**0 (-100%)**

Finally, the energy efficiency gains, detailed in Table 3, mirror the performance improvements. By completing the operation significantly faster at a similar power draw, our in-register method consumes over **6x less energy per operation on the H100** and over **11x less energy on the RTX 5080**. This demonstrates that the technique not only accelerates computation but does so with a massive improvement in performance-per-watt.

Table 3: Energy Efficiency Comparison. Energy/Op = Avg. Power (W) $\times$ Latency (s).

Architecture	Kernel	Energy/Op (µJ)	Efficiency Gain
NVIDIA H100	Baseline (Shared Mem)	8.82	1.0x
	In-Register (Ours)	**1.42**	**6.2x**
NVIDIA RTX 5080	Baseline (Shared Mem)	2.91	1.0x
	In-Register (Ours)	**0.25**	**11.6x**

All in all, every metric makes this technique preferable to the standard one, even register usage, which is the most surprising result.

4 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 't' 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 Discussion and Future Work

The 11.5x speedup achieved in our proof-of-concept is a strong indicator of the potential for this technique. The logical next step is to integrate this building block into a complete, end-to-end FlashAttention kernel. This would involve replacing the standard softmax implementation with our in-register reduction for both the max and sum calculations.

The primary challenge in such an integration would be managing the running statistics (m_i, l_i) and the output accumulator (O_i) directly in registers alongside the main computation, in order to maintain full data locality throughout the outer loop of the FlashAttention algorithm.

6 Limitations

The primary limitation of this technique is its dependency on an undocumented, architecture-specific data layout. The permutation formulas we derived are specific to the NVIDIA Ampere/Hopper/Ada generation of Tensor Cores. Future hardware generations may alter this layout, requiring a new reverse-engineering effort and breaking code portability. This approach therefore represents a trade-off: sacrificing portability for a significant gain in performance on a specific target architecture. Furthermore, the increased code complexity may pose a challenge for long-term maintainability compared to using standard API calls.