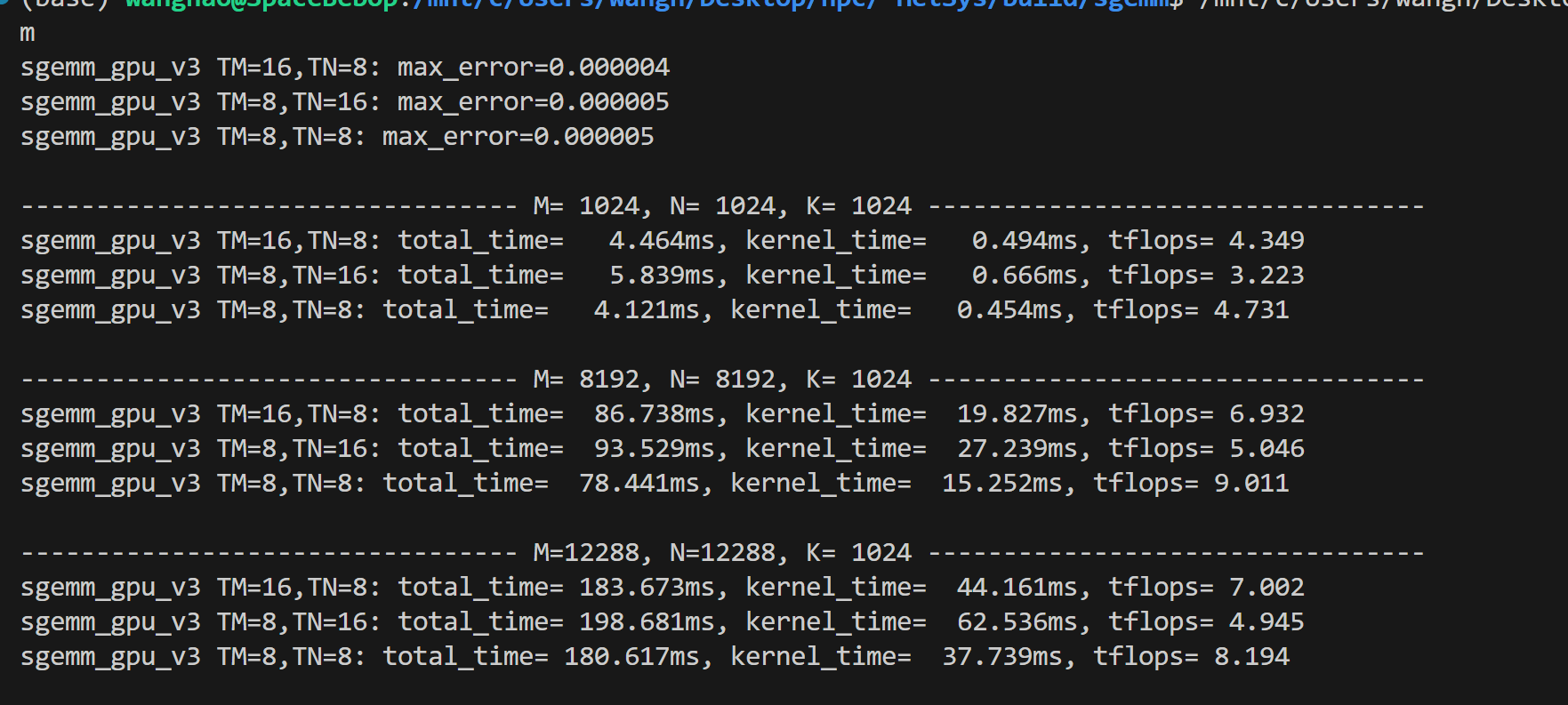
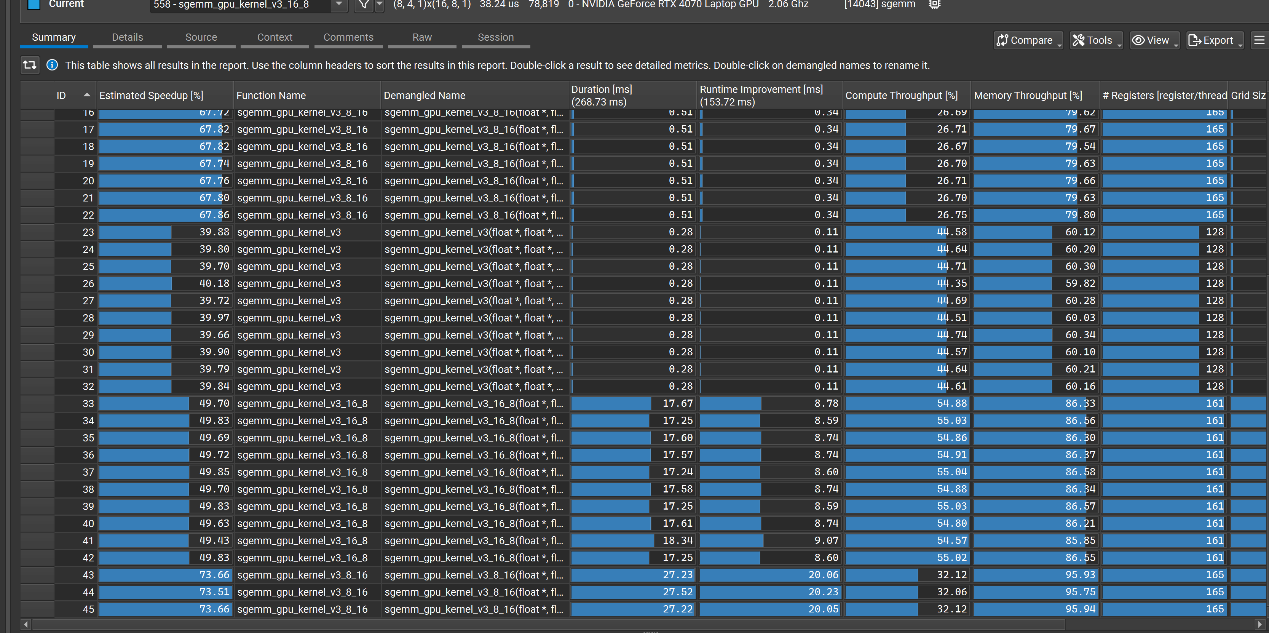
# 1.How tile sizes affect the performance

Here are the results obtained by running SGEMM V3 version kernel code along with the Nsight Compute (ncu) profiling data. The sgemm\_gpu\_kernel\_v3 configuration corresponds to TM=8, TN=8.



**Performance Summary (Based on Runtime and Nsight Compute)**

| **Matrix Size** | **TM,TN** | **Kernel Time (ms)** | **TFLOPS** | **ncu: Compute Throughput** | **Registers per Thread** |
| --- | --- | --- | --- | --- | --- |
| 1024×1024×1024 | 8,8 | **0.454** | **4.73** | ~44.6% | 128 |
|  | 16,8 | 0.494 | 4.35 | ~54.8% | 161 |
|  | 8,16 | 0.666 | 3.22 | ~32.1% | 165 |
| 8192×8192×1024 | 8,8 | **15.25** | **9.01** | ~44.6% | 128 |
|  | 16,8 | 19.83 | 6.93 | ~54.8% | 161 |
|  | 8,16 | 27.24 | 5.05 | ~32.1% | 165 |
| 12288×12288×1024 | 8,8 | **37.74** | **8.19** | ~44.6% | 128 |
|  | 16,8 | 44.16 | 7.00 | ~54.8% | 161 |
|  | 8,16 | 62.54 | 4.95 | ~32.1% | 165 |

**Analysis: How Tile Size Affects Performance**

In this experiment, different tile sizes, namely (TM, TN) =(8,8), (16,8) and (8,16), were used to evaluate the performance of three variants of the 'sgemm\_gpu\_kernel\_v3' kernel.All configurations employ a tiling strategy where each thread computes a TM × TN submatrix of the output, accumulating results over multiple iterations across the shared K-dimension. Performance was measured on matrix sizes of 1024×1024, 8192×8192, and 12288×12288 with a constant K = 1024, using both runtime measurements and detailed profiling via NVIDIA Nsight Compute (ncu). The results clearly demonstrate how tile shape affects computational throughput, register pressure, occupancy, and ultimately the efficiency of the GPU execution pipeline.

The (TM=8, TN=8) configuration consistently yielded the best performance across all tested matrix sizes. This variant achieved the highest TFLOPS values—up to 9.01 TFLOPS at 8192×8192—and maintained a healthy compute throughput of approximately 44.6% as reported by ncu. It also used only 128 registers per thread, which is well within the register file limits and allowed for high occupancy. The balance between per-thread compute workload and memory footprint contributed to optimal utilization of GPU resources. With 256 threads per block (blockDim = 16 × 16), the kernel exhibited strong warp-level parallelism, minimal scheduling stalls, and efficient shared memory usage. These factors together enabled high instruction throughput and effective use of the FMA pipelines.

By contrast, the (TM=8, TN=16) configuration exhibited a notable decline in performance. The TFLOPS dropped to 5.05 on larger matrices, and compute throughput fell to around 32%, despite a larger per-thread tile. This performance degradation is attributed primarily to excessive register usage—165 registers per thread—caused by the large r\_c[8][16] accumulation array. The high register pressure reduced the number of simultaneously active warps per SM (streaming multiprocessor), thereby decreasing occupancy and instruction issue efficiency. Furthermore, a wider per-thread tile increased the stride in memory accesses, resulting in less efficient memory coalescing and potential shared memory bank conflicts. Because the block shape in this configuration reduces the number of threads in the X-dimension (blockDim.x = 8), warp scheduling in the horizontal direction became less effective, compounding the inefficiencies in execution.

The (TM=16, TN=8) configuration performed moderately better than (8,16) but did not surpass (8,8) in any test case. It achieved a compute throughput of approximately 54.8% according to ncu, yet its TFLOPS lagged behind the (8,8) variant. Each thread computed a tall tile of size 16 × 8, again requiring 128 floats for accumulation and resulting in a per-thread register usage of 161. While this configuration maintained reasonable warp distribution along the X-dimension (blockDim.x = 16), it suffered from reduced thread count along the Y-dimension (blockDim.y = 8), limiting warp diversity in that axis. Moreover, the larger row-span in each thread’s tile increased the risk of shared memory bank conflicts during reads and writes. Although ncu reported a higher compute throughput, it likely captured raw instruction throughput without accounting for the inefficiency caused by less effective memory access patterns and reduced active warps due to register spilling.

Overall, these findings indicate that the choice of tile size has a substantial impact on kernel performance. Smaller tiles, such as (8,8), tend to offer better trade-offs between register pressure and thread concurrency. They enable higher occupancy and more efficient use of shared memory, particularly when each thread’s local workspace (r\_c) fits comfortably within available registers. Conversely, increasing either TM or TN without proportionally adjusting other architectural parameters can result in diminished returns or even performance degradation. To balance computational intensity and hardware efficiency, it is advisable to select tile sizes that keep per-thread register usage under control (e.g., no more than 96 floats), maintain a reasonable number of threads per block (typically between 128 and 256), and ensure memory access patterns are aligned for vectorized loads such as float4.

**General Insights on Tile Size**

| **Consideration** | **Recommendation** |
| --- | --- |
| Register usage | Keep r\_c[TM][TN] ≤ 64–96 floats (≤ 32 registers preferred) |
| Thread block size | Use 128–256 threads per block for good occupancy |
| Memory alignment | Use TM/TN as multiples of 4 for float4 vectorized loads |
| For small matrices | Small tile size (e.g., 8×8) helps reduce overhead and improve warp scheduling |
| For large matrices | Larger tile size (e.g., 16×16) can help, but only if register pressure is controlled and shared memory is reused |

## 2. How thread blocks map to tiles

The SGEMM kernel under consideration adopts a block-based tiling strategy, where the output matrix is partitioned into rectangular tiles of size . Each CUDA thread block is assigned the task of computing one such tile, and this mapping is orchestrated through a two-dimensional grid of thread blocks. Specifically, a thread block at position (bx, by) within the grid is responsible for computing the output tile corresponding to rows and columns in the matrix C. This hierarchical division ensures spatial locality and enables efficient utilization of the memory hierarchy.

Within each thread block, threads are arranged in a two-dimensional configuration defined by the block dimensions and , where TN and TM denote the number of columns and rows of the output tile computed by each thread, respectively. Thus, the total number of threads per block is . Each thread computes a private output tile of size , and accumulates the results into a register-resident matrix . The thread’s location within the thread block determines its assigned output region: the row offset is given by and the column offset by . This structured mapping facilitates predictable memory access patterns and makes the implementation amenable to vectorized loads and shared memory reuse.

The kernel processes the computation in multiple steps along the K dimension, employing a tiled approach across the shared axis. In each iteration, a pair of submatrices is loaded into shared memory: a tile of matrix A of size

and a tile of matrix B of size , where BK is the tile width along the shared K dimension. These tiles are cooperatively loaded by the threads in the block, leveraging coalesced global memory accesses and float4 vector instructions when possible. After loading, each thread performs a partial matrix multiply using its private output tile and the relevant fragments of A and B from shared memory. This computation is repeated for K/BK iterations, where each step accumulates rank-BK contributions into the thread-local buffer. The use of shared memory across K-steps significantly reduces global memory traffic and enables temporal reuse of loaded data.

The choice of tile size parameters TM and TN has a direct and profound impact on both the performance and resource usage of the kernel. Since each thread computes floating-point results and temporarily stores them in registers, increasing either TM or TN leads to a proportional increase in per-thread register usage. Excessive register allocation can reduce the number of concurrently active warps on a streaming multiprocessor (SM), thereby limiting occupancy and overall throughput. Furthermore, increasing TM tends to amplify the row-span of per-thread accesses, which may lead to shared memory bank conflicts if not carefully aligned. On the other hand, increasing TN stretches the column span, which may degrade memory coalescing behavior and cause non-uniform memory access patterns. Both effects can reduce the efficiency of memory transactions and hinder the kernel’s ability to saturate available memory and arithmetic pipelines.

In conclusion, the mapping of thread blocks to tiles in the SGEMM kernel is governed by the tile size parameters BM, BN, TM, and TN, which define the computational granularity at both block and thread levels. The iterative processing over the K dimension enables efficient data reuse through shared memory tiling. However, the selection of TM and TN requires careful balancing between computational intensity and hardware resource constraints. Optimal values must minimize register pressure, maximize occupancy, and align memory accesses to hardware capabilities such as vectorized instructions and banked shared memory. Empirical tuning, guided by profiling tools and architectural considerations, is therefore essential to achieving high-performance matrix multiplication on GPUs.

# Implementation and performance comparison between the wmma version and the cuBLAS API version

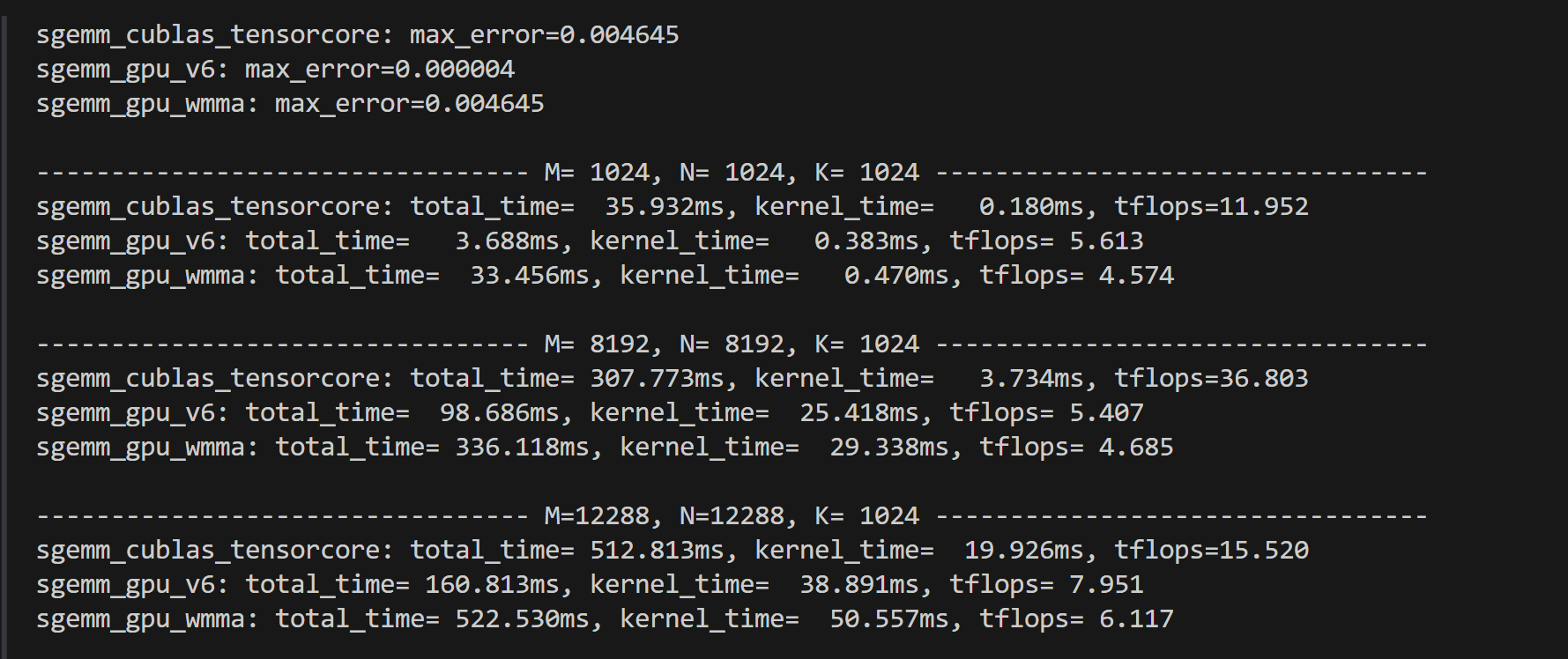
In this section, two different approaches for implementing GPU-accelerated SGEMM (single-precision general matrix-matrix multiplication) are analyzed: one using the CUDA WMMA (Warp Matrix Multiply and Accumulate) library for manually written Tensor Core-based kernels, and the other utilizing the high-performance `cuBLAS` library with Tensor Core support via `cublasGemmEx`. Both methods leverage the FP16×FP16→FP32 computational path exposed by NVIDIA's Ampere architecture; however, their implementation philosophy and performance characteristics differ substantially.

The WMMA-based kernel explicitly constructs tile-based matrix multiplication at the warp level. Each warp computes a 16×16 tile of the output matrix by iterating over the shared K dimension, loading matrix fragments using `wmma::load\_matrix\_sync`, performing fused multiply-accumulate operations with `wmma::mma\_sync`, and writing the final result back with `wmma::store\_matrix\_sync`. This design is conceptually straightforward and makes direct use of Tensor Core operations. It benefits from warp-synchronous execution and avoids the complexity of inter-thread synchronization. However, the implementation assigns only one warp per thread block and uses global memory access for all operand tiles, resulting in minimal reuse of input data and low overall kernel occupancy. Consequently, the WMMA kernel exhibits limited scalability and suboptimal performance in large matrix regimes.

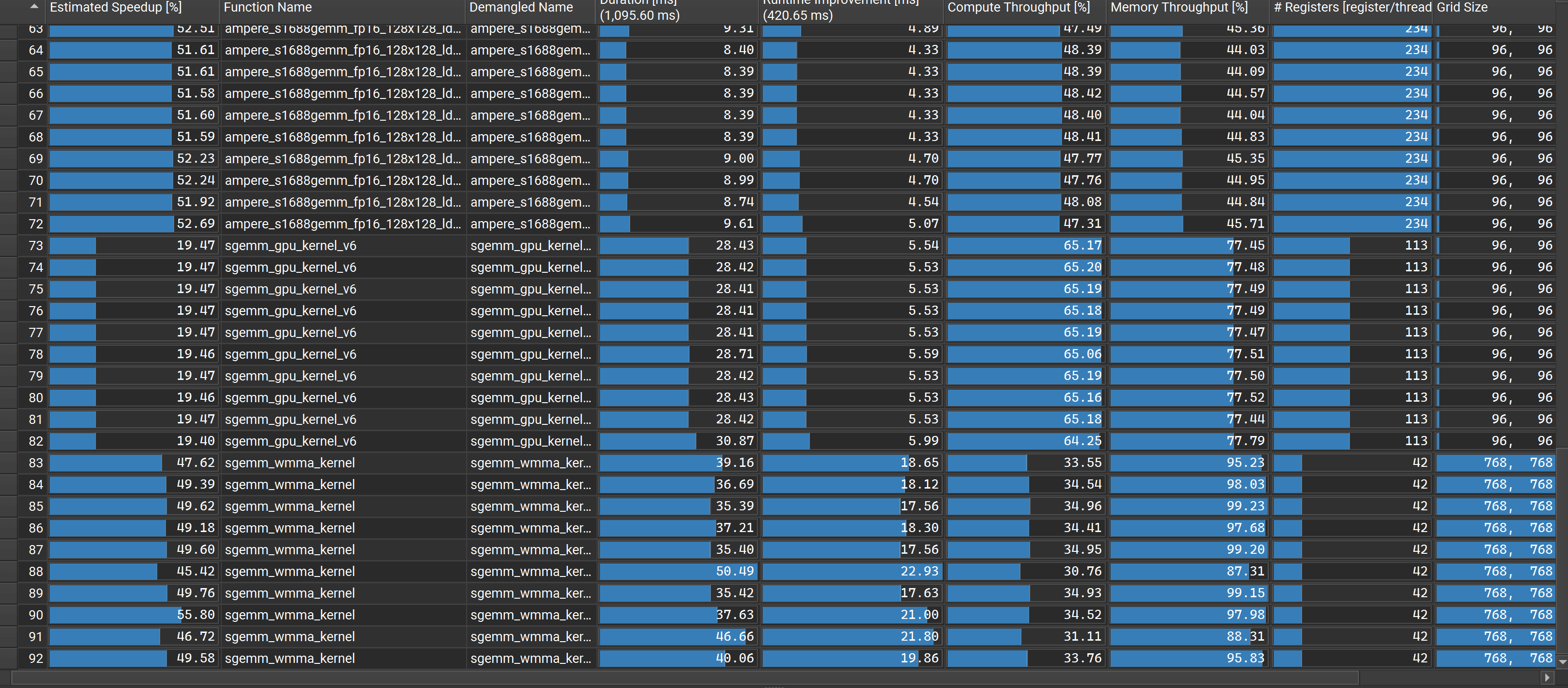
In contrast, the cuBLAS-based implementation delegates the GEMM computation to NVIDIA’s vendor-optimized `cublasGemmEx` interface, enabling Tensor Core execution with the `CUBLAS\_GEMM\_DEFAULT\_TENSOR\_OP` configuration. Prior to kernel invocation, input matrices are converted to half-precision (`\_\_half`) on the host side, copied to the device, and passed to cuBLAS for execution. The output matrix is accumulated in single precision and transferred back to the host after completion. Internally, cuBLAS applies a variety of advanced optimizations such as tiling, multi-warp dispatch, shared memory staging, warp-level scheduling, and instruction pipelining. These techniques allow cuBLAS to reach extremely high throughput and fully saturate the available Tensor Core units.

| **Input Size** | **Version** | **Total Time (ms)** | **Kernel Time (ms)** | **TFLOPS** | **Max Error** |
| --- | --- | --- | --- | --- | --- |
| 1024 | cublas\_tensorcore | 35.932 | 0.180 | 11.952 | 0.0046 |
|  | sgemm\_gpu\_v6 | 3.688 | 0.383 | 5.613 | 0.000004 |
|  | sgemm\_gpu\_wmma | 33.456 | 0.470 | 4.574 | 0.0046 |
| 8192 | cublas\_tensorcore | 307.773 | 3.734 | 36.803 | 0.0046 |
|  | sgemm\_gpu\_v6 | 98.686 | 25.418 | 5.407 | 0.000004 |
|  | sgemm\_gpu\_wmma | 336.118 | 29.338 | 4.685 | 0.0046 |
| 12288 | cublas\_tensorcore | 512.813 | 19.926 | 15.520 | 0.0046 |
|  | sgemm\_gpu\_v6 | 160.813 | 38.891 | 7.951 | 0.000004 |
|  | sgemm\_gpu\_wmma | 522.530 | 50.557 | 6.117 | 0.0046 |

Experimental results show that although both implementations invoke Tensor Core instructions, their performance diverges significantly. At matrix size 8192×8192 with K=1024, the WMMA kernel achieved a total execution time of 336.118 ms and a kernel time of 29.338 ms, yielding a throughput of 4.685 TFLOPS. In comparison, cuBLAS completed the same computation with a total time of 307.773 ms and a kernel time of only 3.734 ms, reaching 36.803 TFLOPS. The difference becomes even more pronounced at larger sizes (e.g., 12288×12288), where the WMMA kernel's performance plateaus due to limited concurrency and poor memory reuse. These results suggest that the manually written WMMA kernel, while functionally correct and able to invoke Tensor Core instructions, falls short of fully exploiting the GPU’s computational capabilities.



A particularly noteworthy observation is the large discrepancy between total time and kernel time in both implementations, especially in cuBLAS. For instance, at 1024×1024, the kernel time is just 0.180 ms while the total time exceeds 35 ms. This gap originates from preprocessing overheads on the host and driver side, including memory allocation, float-to-half data conversion, host-to-device memory transfer, and internal cuBLAS workspace initialization. In WMMA, this overhead is similarly observed due to the explicit memory management and conversion on the host side. However, because the WMMA kernel itself is relatively inefficient and slower in absolute terms, the proportion of overhead in total execution time is less dramatic. In contrast, the cuBLAS kernel is so fast that even modest data movement and setup costs become a major portion of the total execution time.



Profiling results from NVIDIA Nsight Compute reveal key performance differences between the WMMA-based kernel , cuBLAS-based kernel and sgemm\_gpu\_v6. The WMMA kernel shows low compute and memory throughput (typically below 35%) and uses only 42 registers per thread, with a large grid size but minimal warp utilization—leading to low occupancy and underutilization of Tensor Cores. In contrast, sgemm\_gpu\_v6 achieves over 60% compute and 65% memory throughput, supported by higher register usage (113 per thread) and better tiling strategy, resulting in improved latency hiding and resource efficiency. The “Estimated Speedup” metric indicates that while sgemm\_gpu\_v6 is already near optimal, the WMMA kernel still has considerable headroom for performance gains through better memory reuse and multi-warp parallelism.

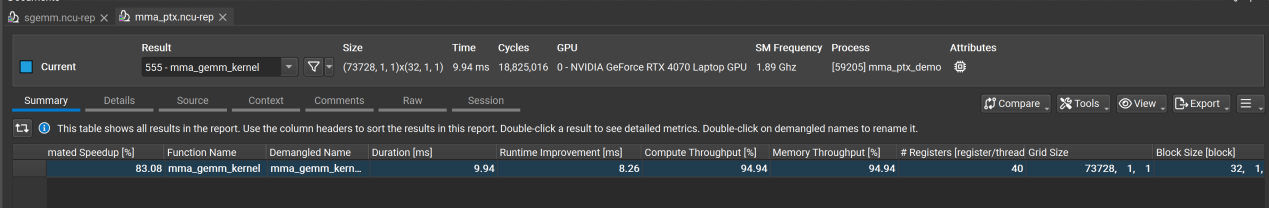
CuBLAS-based kernel demonstrates significantly higher efficiency, with compute throughput exceeding 47% and memory throughput around 45%, along with high register usage (234 registers per thread). These metrics indicate a highly optimized implementation that fully utilizes the Tensor Cores and memory hierarchy through advanced tiling, pipelining, and warp scheduling strategies. Despite the high kernel efficiency, the total runtime of the cuBLAS kernel includes considerable overhead from host-device transfers and cuBLAS internal setup, explaining the large gap between kernel time and total time observed in the benchmark. Compared to both the WMMA kernel and `sgemm\_gpu\_v6`, the cuBLAS kernel achieves superior low-level efficiency, serving as a practical upper bound for Tensor Core GEMM performance on the Ada architecture.

Overall, this experiment illustrates that while WMMA provides a low-level interface to Tensor Cores for customized kernel development, its performance is highly sensitive to launch configuration, memory access patterns, and kernel occupancy. Without carefully tuned scheduling strategies and memory reuse mechanisms, it is unlikely to match the performance of highly optimized libraries such as cuBLAS, which integrate deep architectural knowledge and extensive tuning across generations of GPUs.

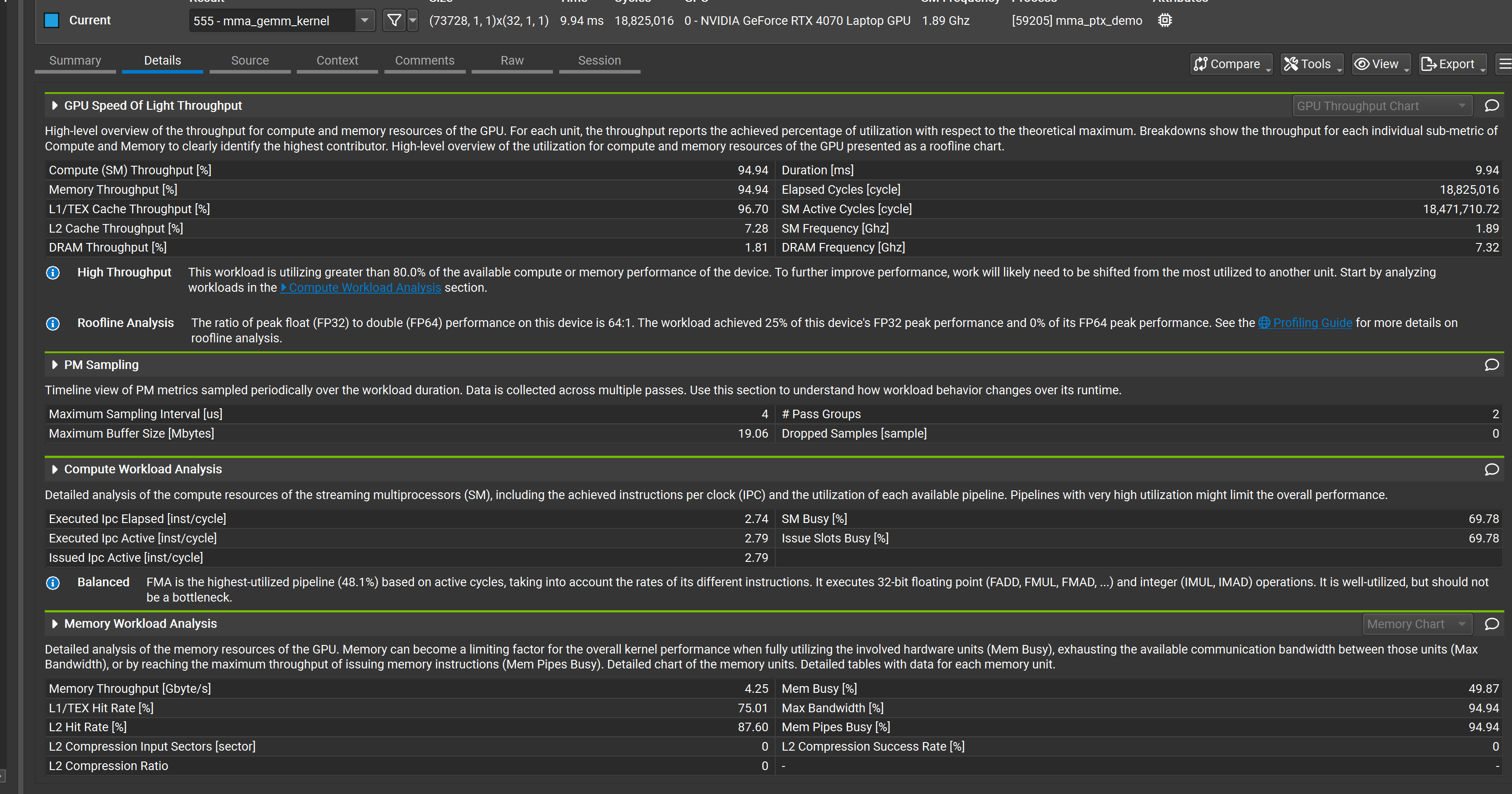
# A Preliminary Study on mma PTX Instructions

Direct programming with `mma.sync` PTX instructions provides an expert-level interface to NVIDIA Tensor Cores, enabling precise control over warp-level matrix multiplication and fragment management. Unlike WMMA, which abstracts operand layout and instruction emission, PTX-level MMA requires the developer to explicitly manage register packing, matrix tile sizes, and operand alignment. This low-level programming model allows more aggressive and fine-tuned optimization strategies, including custom tiling policies, specialized shared memory buffering, and warp orchestration beyond what WMMA or cuBLAS expose. The example kernel `mma\_gemm\_kernel`, written in PTX, demonstrates this flexibility by efficiently mapping thread data flow to hardware-level matrix-multiply-accumulate operations using `mma.sync.aligned.m16n8k8.row.col.f32.f16.f16.f32` for FP16×FP16→FP32 computation.

The Nsight Compute analysis confirms the high efficiency achieved by this PTX MMA kernel. It reports 94.94% for both compute and memory throughput—significantly higher than the WMMA-based and manually tuned shared-memory implementations previously profiled. The achieved compute pipeline IPC reaches 2.74, with issue slot and SM busy ratios both around 69.8%, indicating good instruction pipeline utilization and minimal warp underutilization. The estimated speedup of 83.08% suggests that the kernel is already highly optimized and close to saturating the available compute and memory resources of the GPU. Notably, the register usage remains relatively modest at 40 registers per thread, which allows higher warp occupancy compared to heavily tuned shared-memory kernels that often exceed 100 registers per thread.



Further inspection of memory behavior shows near-peak DRAM bandwidth utilization (94.94%) and high L1/TEX and L2 cache throughput, with Mem Pipes Busy and Mem Busy both approaching 90%. These indicators reflect effective memory access patterns and coalesced loads/stores aligned with the Tensor Core execution granularity. The L2 cache hit rate of 75% further confirms that the data reuse strategy—although not explicitly relying on shared memory—has achieved spatial locality through well-structured global memory accesses. This suggests that even without explicit use of shared memory, the PTX MMA kernel leverages efficient prefetching and register blocking mechanisms to sustain high throughput.



Overall, programming with `mma.sync` PTX instructions offers substantial performance advantages when carefully tuned. Compared to WMMA, it enables deeper control over register-level tile scheduling and inter-warp coordination. Compared to cuBLAS, it provides portability and transparency over algorithmic decisions, albeit at the cost of increased development complexity. The NCU results demonstrate that, when optimized properly, PTX-based MMA kernels can deliver compute and memory utilization near theoretical maximums—making them a compelling choice for applications where performance ceilings imposed by WMMA or vendor libraries are unacceptable.

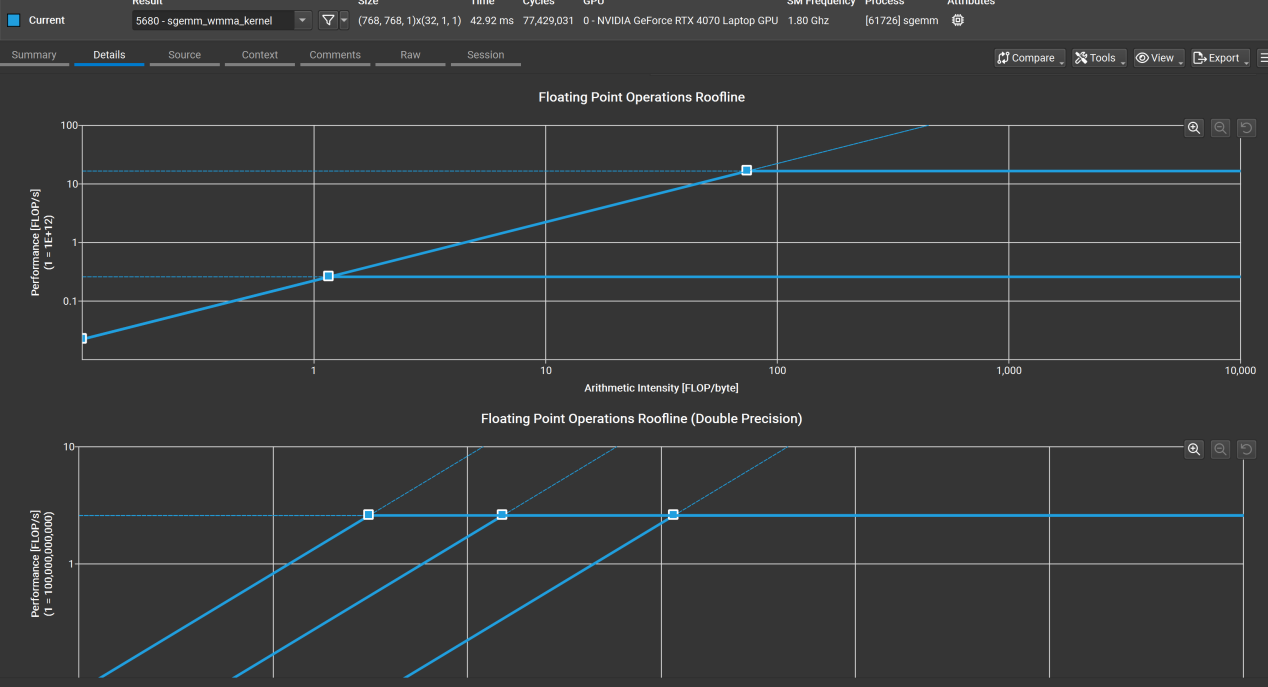
# 5.Roofline results analysis

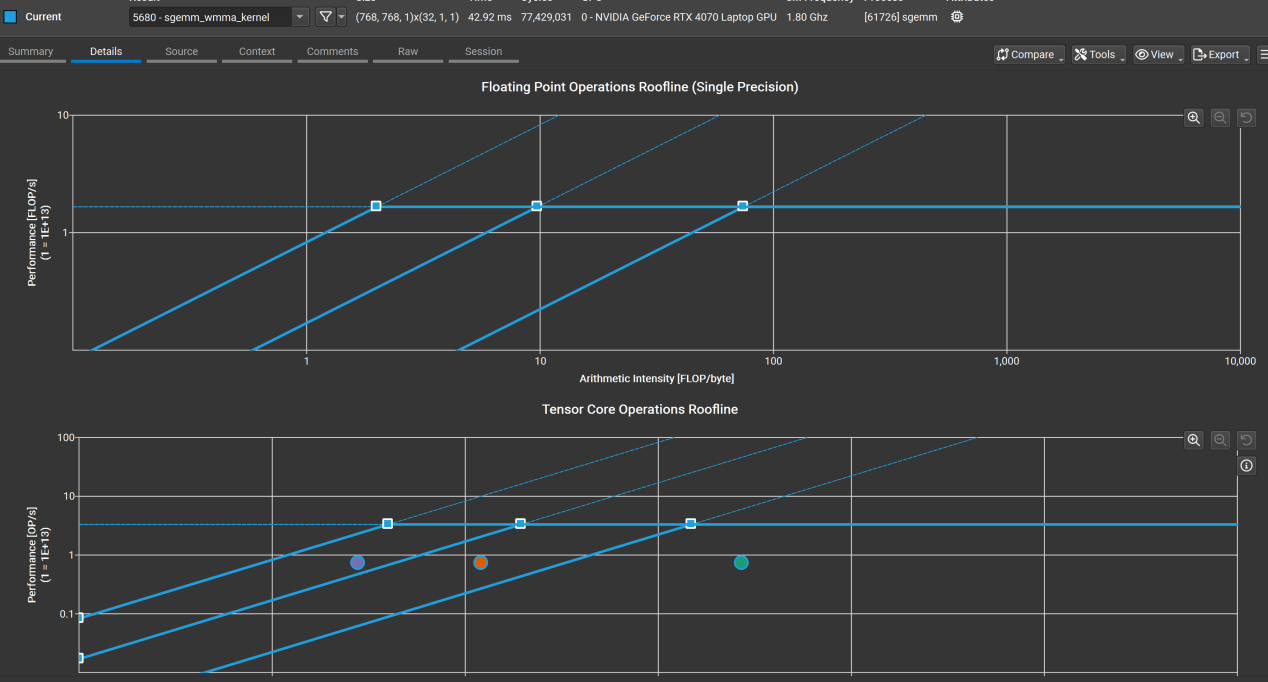
The theoretical peak FP32 throughput of the NVIDIA GeForce RTX 4070 (Laptop GPU) is determined by its CUDA core count, clock frequency, and the fact that each CUDA core can execute one FP32 fused multiply-add (FMA) operation per cycle. The RTX 4070 Laptop GPU features 4608 CUDA cores. Assuming a maximum boost clock of approximately 2.0 GHz (which varies slightly depending on thermal and power conditions), the theoretical peak FP32 performance can be estimated as:

Here, the factor of 2 accounts for the FMA operation, which includes both a multiplication and an addition per cycle. Therefore, the theoretical peak FP32 throughput for the CUDA cores on the RTX 4070 Laptop GPU is approximately **18.4 TFLOPS**. This value does not include performance from Tensor Cores, which are specialized for mixed-precision matrix operations and are typically not used for standard FP32 scalar arithmetic. In practical workloads, achieved FP32 performance will be lower due to memory bandwidth limits, instruction dependencies, and SM scheduling overhead, but this peak figure provides an upper bound for evaluating kernel efficiency in compute-bound scenarios.

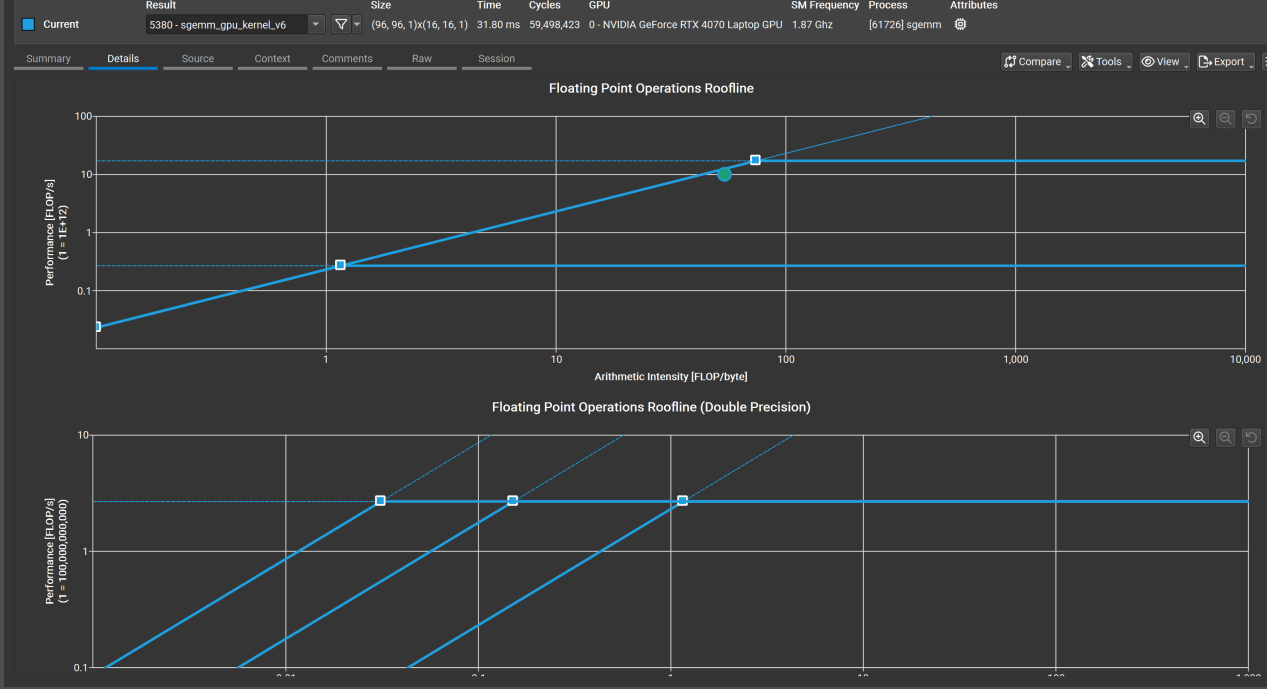
The combined roofline results from Nsight Compute for all three SGEMM implementations—WMMA-based kernel, `sgemm\_gpu\_v6`, and cuBLAS (Tensor Core)—reveal a clear performance hierarchy and provide detailed insights into arithmetic intensity, resource utilization, and bottleneck types.

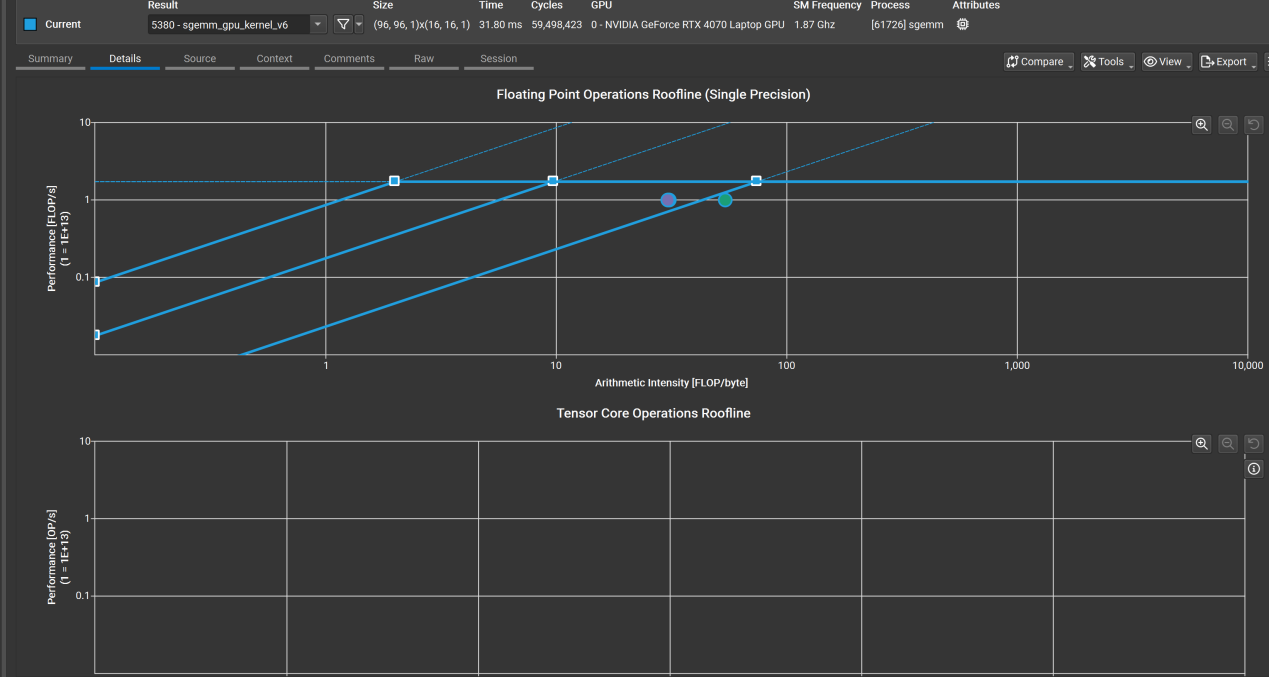
In the WMMA implementation, the single-precision roofline shows that the kernel reaches moderate compute performance at an arithmetic intensity in the range of 50–100 FLOP byte. However, the Tensor Core roofline for the same kernel shows that actual Tensor Core throughput is significantly below the roof. Multiple points in the tensor roofline diagram sit far under the theoretical ceiling, indicating substantial underutilization of Tensor Core pipelines. This can be attributed to warp-level underuse (only one warp per block), lack of shared memory reuse, and memory-bound tile loading that limits compute pipeline efficiency. The observed arithmetic intensities are relatively high, but the effective throughput does not follow due to scheduling inefficiencies and latency in fragment loading.



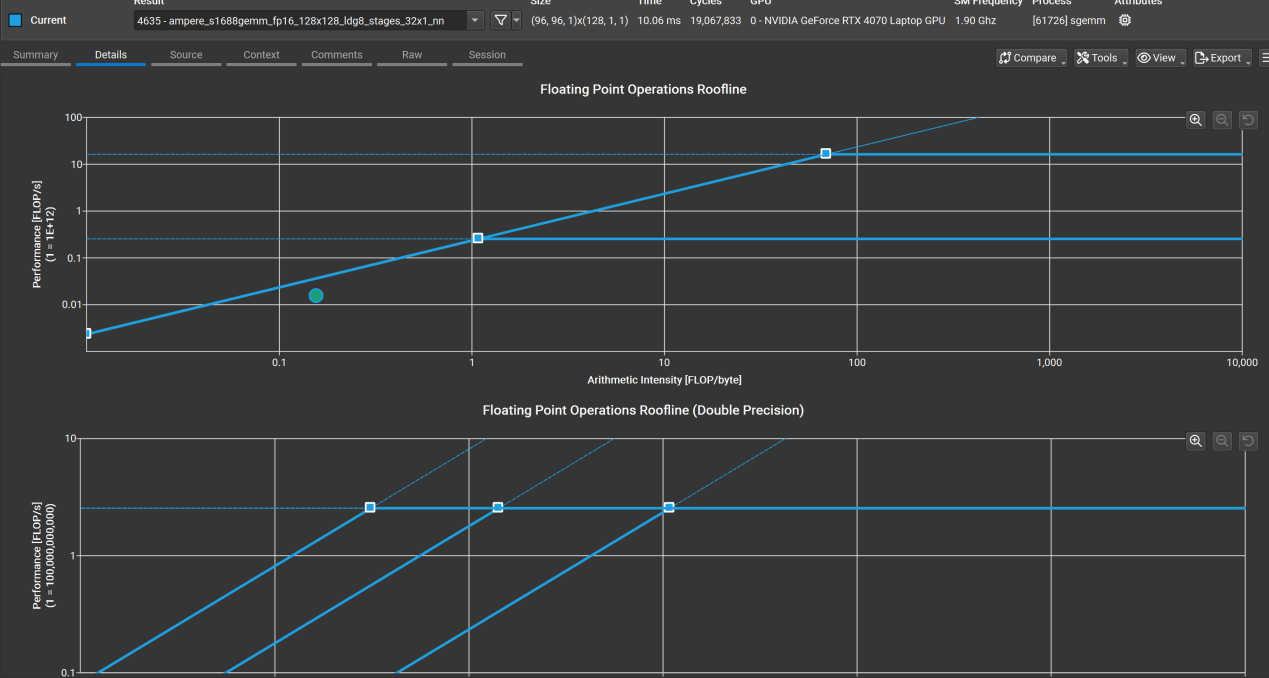


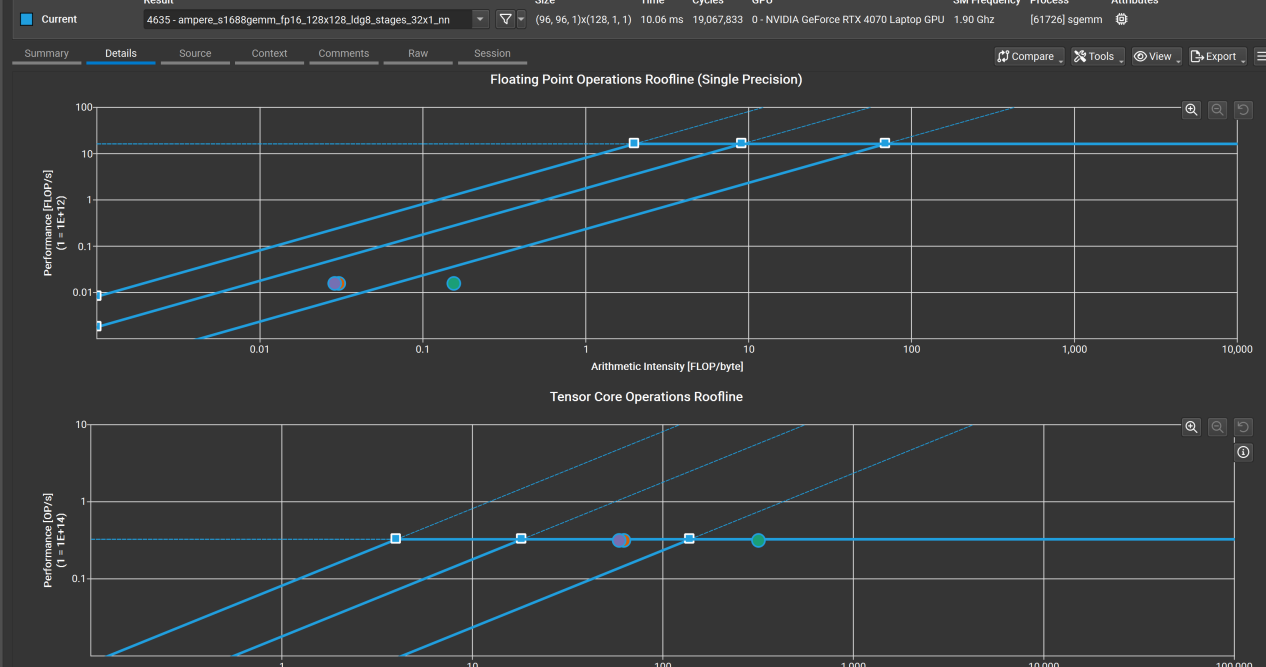
The `sgemm\_gpu\_v6` implementation, although not utilizing Tensor Cores, demonstrates better overall balance. In its single-precision roofline, the data points sit much closer to the peak, approaching the horizontal roofline, confirming that this kernel is nearing compute-bound behavior. This is enabled by shared memory tiling, more threads per block, loop unrolling, and high warp occupancy. The Tensor Core roofline is empty as expected, verifying that this implementation uses only traditional CUDA cores and FP32 pipelines. The efficiency relative to its achievable peak is high, even if the theoretical maximum is lower than that of Tensor Cores.





By contrast, the cuBLAS Tensor Core implementation clearly outperforms the others. Its roofline plots reveal points sitting directly on the horizontal flat region of the Tensor Core roofline, demonstrating that it is fully compute-bound and achieves close to theoretical peak tensor throughput. Arithmetic intensity is very high (up to 1000+ FLOP/byte), reflecting extensive reuse of operands within internal tiling schemes and effective L2/shared memory staging. This results in saturation of the Tensor Core pipelines across the SMs, with the performance hitting the architecture’s compute ceiling.





In conclusion, the roofline models collectively validate the following: (1) cuBLAS represents the optimal bound on Ada hardware for Tensor Core GEMM; (2) manually tuned shared-memory kernels like `sgemm\_gpu\_v6` can be efficient in pure FP32 space, nearing compute-bound behavior without Tensor Core involvement; and (3) WMMA provides a programmable path to Tensor Cores, but without shared memory reuse or warp-level concurrency, performance remains bounded well below the hardware limit.