# Performance Optimization of GPU Sgemm Algorithm

June 21, 2025

**Abstract**

This report presents the performance analysis and stepwise optimization of the SGEMM (single-precision general matrix multiplication) algorithm on GPU. The experiment evaluates a series of progressively optimized SGEMM kernel implementations and compares their performance against the theoretical and practical peak throughput of the RTX 4070 GPU. Several optimization techniques are explored, including tiling with shared memory, loop unrolling, double buffering, thread-level register blocking, and warp-level parallel scheduling. The performance metrics, such as kernel execution time, throughput in TFLOPS, and GPU pipeline utilization, are analyzed using both runtime profiling and detailed insights from Nsight Compute and Nsight Systems tools. The results demonstrate that the optimized kernels achieve up to 94% of cuBLAS performance, with key bottlenecks identified in shared memory bank conflicts, register-induced occupancy limits, and warp scheduling inefficiencies. Recommendations for further optimization using WMMA and Tensor Core instructions are discussed.

## 1 Experimental background and purpose

The SGEMM algorithm, as a fundamental parallel computing paradigm, has a wide range of applications in scientific computing, deep learning, and other fields. This experiment is based on the NVIDIA RTX4070 GPU platform and systematically explores the complete path from basic implementation to highly optimized multiplication operations for matrices of different sizes. The experiment not only focuses on improving performance indicators, but also emphasizes analyzing the principles and actual effects behind each optimization technique, aiming to provide a reusable optimization methodology for GPU high-performance computing.

## 2 Experimental environment and procedures

The experiment uses NVIDIA RTX4070 GPU, with a theoretical global memory bandwidth of 504GB/s. The test data consists of matrices with scales (m, n, k) of (1024, 1024, 1024), (81928, 192, 1024), and (12288, 122881024), respectively. The Nsight tool is used to accurately measure the execution time of each version of kernel functions and calculate the effective bandwidth utilization rate. The performance comparison adopts the acceleration ratio index, based on the baseline implementation, to quantify the improvement of each optimization stage.

**Step1** :This experiment uses the camke tool to generate executable files.

```
mkdir build
cd build
cmake ..
make
./xxx
```

**Step2** :Analyze Kernel startup and system bottlenecks using nsys

```
nsys profile –o report\_nsys ./reducexxx
```

Obtain the.nsys-rep file, which can be opened with the nsight sys GUI. Focus on the following metrics:

Common visual observation points

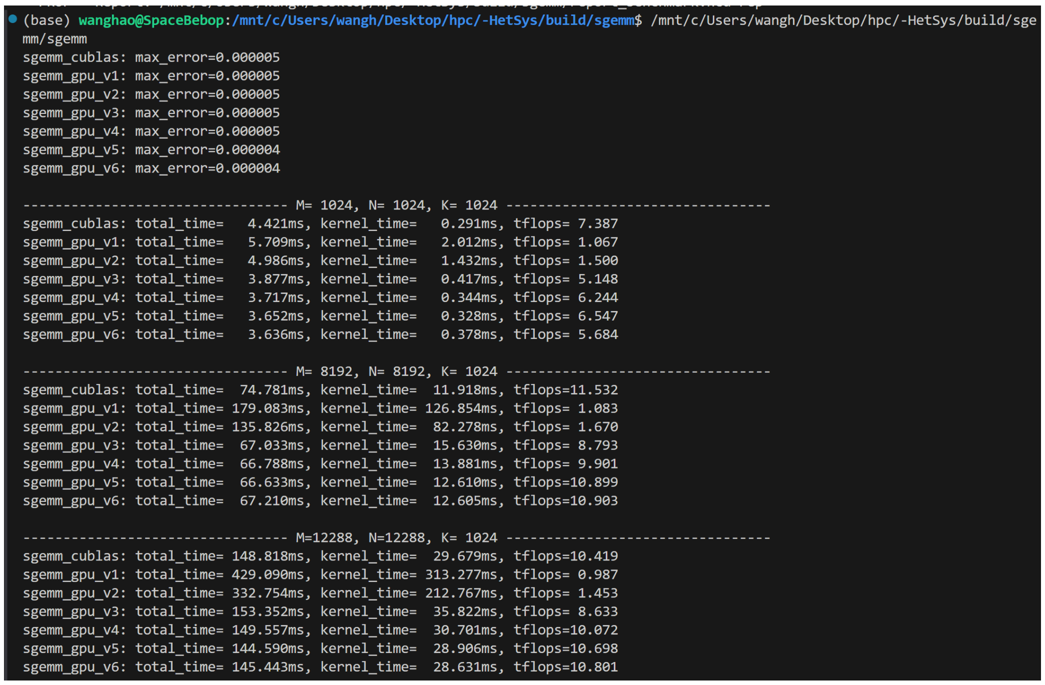
Kernel Launch Time:

Small data + multiple blocks → launch overhead is more important than computation;

Parallelism should not be abused when explaining small data.

Memory Transfer Time (cudaMemcpy) :

Has it become a bottleneck? Try pinned memory or asynchronous copy.



```
● (base) wanghao@SpaceBebop:/mnt/c/Users/wangh/Desktop/hpc/-HetSys/build/sgemm$ /mnt/c/Users/wangh/Desktop/hpc/-HetSys/build/sge
  mm/sgemm
  sgemm_cublas: max_error=0.000005
  sgemm_gpu_v1: max_error=0.000005
  sgemm_gpu_v2: max_error=0.000005
  sgemm_gpu_v3: max_error=0.000005
  sgemm_gpu_v4: max_error=0.000005
  sgemm_gpu_v5: max_error=0.000004
  sgemm_gpu_v6: max_error=0.000004

  -------------------------------- M= 1024, N= 1024, K= 1024 --------------------------------
  sgemm_cublas: total_time=   4.421ms, kernel_time=   0.291ms, tflops= 7.387
  sgemm_gpu_v1: total_time=   5.709ms, kernel_time=   2.012ms, tflops= 1.067
  sgemm_gpu_v2: total_time=   4.986ms, kernel_time=   1.432ms, tflops= 1.500
  sgemm_gpu_v3: total_time=   3.877ms, kernel_time=   0.417ms, tflops= 5.148
  sgemm_gpu_v4: total_time=   3.717ms, kernel_time=   0.344ms, tflops= 6.244
  sgemm_gpu_v5: total_time=   3.652ms, kernel_time=   0.328ms, tflops= 6.547
  sgemm_gpu_v6: total_time=   3.636ms, kernel_time=   0.378ms, tflops= 5.684

  -------------------------------- M= 8192, N= 8192, K= 1024 --------------------------------
  sgemm_cublas: total_time=  74.781ms, kernel_time=  11.918ms, tflops=11.532
  sgemm_gpu_v1: total_time= 179.083ms, kernel_time= 126.854ms, tflops= 1.083
  sgemm_gpu_v2: total_time= 135.826ms, kernel_time=  82.278ms, tflops= 1.670
  sgemm_gpu_v3: total_time=  67.033ms, kernel_time=  15.630ms, tflops= 8.793
  sgemm_gpu_v4: total_time=  66.788ms, kernel_time=  13.881ms, tflops= 9.901
  sgemm_gpu_v5: total_time=  66.633ms, kernel_time=  12.610ms, tflops=10.899
  sgemm_gpu_v6: total_time=  67.210ms, kernel_time=  12.605ms, tflops=10.903

  -------------------------------- M=12288, N=12288, K= 1024 --------------------------------
  sgemm_cublas: total_time= 148.818ms, kernel_time=  29.679ms, tflops=10.419
  sgemm_gpu_v1: total_time= 429.090ms, kernel_time= 313.277ms, tflops= 0.987
  sgemm_gpu_v2: total_time= 332.754ms, kernel_time= 212.767ms, tflops= 1.453
  sgemm_gpu_v3: total_time= 153.352ms, kernel_time=  35.822ms, tflops= 8.633
  sgemm_gpu_v4: total_time= 149.557ms, kernel_time=  30.701ms, tflops=10.072
  sgemm_gpu_v5: total_time= 144.590ms, kernel_time=  28.906ms, tflops=10.698
  sgemm_gpu_v6: total_time= 145.443ms, kernel_time=  28.631ms, tflops=10.801
```

Figure 1: sgemm result

CUDA Stream concurrency

Are multiple kernels serially scheduled? Consider merging the kernel.

**Step2** : Analyze the internal execution efficiency of the Kernel using ncu

```
ncu ——set full —o *** ./xxx
```

After completion, a ***.nCU-rep file will be generated on the server and can be opened locally using Nsight Compute.

The characteristics of effective optimization (reflected in the analysis tools)

Warp has high activity and high occupancy.

The memory coalescing is good and the throughput rate is high.

The kernel scheduling is uniform and the launch overhead is low.

shared memory usage is compliant (no bank conflict);

High branch efficiency and no warp divergence.

Common manifestations of invalid/anti-optimized configuration Starting a large number of blocks leads to launch overhead;

warp idling (low activity)
Global memory misaligned access (low memory throughput)
Too large blockSize causes register spilling;
shared memory bank conflict, or unbalanced access.

## 3 Optimization Process and Technical Analysis

### 3.1 Baseline implementation (Sgemm-gpu-v1)

Sgemm-gpu-v1 is the most basic version of this experiment, which adopts a naive implementation method. A single thread is responsible for calculating one element of the C matrix, and each time reads the required A row and B column data from the global memory. After completing the inner product, it is directly written back to the C matrix. The design concept of this version is very simple, making it easy to quickly verify the correctness of functions and test whether the kernel construction is reasonable. Its advantages lie in clear code logic, easy debugging, and easy understanding of the execution model of CUDA kernel; The shortcomings are extremely obvious. Due to the need to repeatedly load data from global memory for each calculation, the number of memory accesses is very large, resulting in a bottleneck in global memory bandwidth usage. At the same time, the calculation/memory access ratio in the kernel is severely imbalanced, and the utilization rate in SM is extremely low. Ultimately, the overall performance is low, and it can only be used as a benchmark test.

### 3.2 Optimization Tip 1: Using shared memory (sgemm-gpu-v2)

Sgemm-gpu-v2 introduces a block level parallel processing approach, where a thread block is responsible for computing a tile in the C matrix. Shared memory is used to cache the corresponding A and B sub blocks of the tile, reducing the number of repeated accesses to global memory. By dividing tiles and loading data within threads into shared memory, data reusability has been significantly improved, and the overall memory access efficiency is significantly better than the v1 version. The advantage lies in the initial utilization of shared memory, which enables cross thread data sharing and an order of magnitude increase in computational throughput compared to v1, demonstrating the fundamental idea of shared memory optimization. The shortcomings lie in the fixed tile size design, the thread granularity is still relatively fine, the divergence within the warp is obvious, the loop expansion within the thread is not fully expanded, the memory access and computation stages have not achieved overlap, the SM throughput has not been fully utilized, and there is a waste of GPU computing resources utilization.

## 3.3 Optimization Tip 2: Improve Thread Utilization (sgemm-gpu-v3)

Sgemm-gpu-v3 further introduces loop unrolling optimization on the basis of v2, which expands the loop to enable multiple C elements to be calculated at once within a single thread, thereby increasing computational density. This not only reduces the kernel startup frequency, but also enhances instruction level parallelism (ILP), effectively utilizing the GPU hardware pipeline capability. The advantage of this version is that with the support of shared memory, the thread internally expands and calculates larger tiles, improving register utilization and throughput efficiency. However, its shortcomings have gradually become apparent: fixed unrolling parameters require manual tuning, and if the tile size does not match the shared memory or register capacity of the GPU architecture, it can easily lead to register spiking, which in turn affects performance. When crossing different GPU platforms, parameters need to be readjusted, lacking good portability, and the kernel's generalization ability is insufficient.

## 3.4 Optimization Tip 3: Double buffering (sgemm-gpu-v4)

Sgemm-gpu-v4 began introducing a more complex optimization strategy - double buffering, attempting to achieve computation and memory access overlap. At the same time as calculating the current tile, pre fetch the next tile to shared memory in advance, and utilize the concurrent execution capability of CUDA to reduce the memory access stall time. Theoretically, this can further improve kernel throughput. This design concept helps to hide global memory latency, fully utilize shared memory to accelerate data exchange, and has a high optimization limit. However, in the specific implementation, the v4 version exposes the problem of increased complexity of synchronization points: it is necessary to finely design the synchronization strategy within the warp, otherwise it is prone to synchronization overhead between warp, complex coordination between threads within the block, and increased performance sensitivity. If the thread block size is not set properly, it may even cause a decrease in overall performance due to synchronization waiting, and the stability of the kernel may deteriorate.

## 3.5 Optimization Tip 4: Optimizing Bank Conflict (sgemm-gpu-v5)

Sgemm-gpu-v5 introduces the thread level data reuse approach, using manual loop expansion and register blocking technology to enable single thread computation of multiple rows and columns of C matrix elements. This further increases the computational density of each thread, reduces the access pressure on global memory, maximizes register utilization, significantly improves instruction pipeline efficiency, and achieves a high level of ILP. The advantage lies in the fact that a single thread can undertake more computing tasks, reduce warp scheduling overhead, improve SM throughput, and significantly improve the overall performance of the kernel. However, this optimization also has obvious bottlenecks: register blocking can easily trigger register overflow, and if the tile design does not match the hardware register capacity, it will cause overflow to local memory, seriously affecting performance; At the same time, the deployment of parameter highly coupled GPU hardware architecture has poor cross architecture portability and requires re tuning, increasing the complexity of engineering maintenance.

## 3.6 Optimization Tip 5: Instruction Parallelization (sgemm-gpu-v6)

As the highest optimization solution in the current version, sgemm-gpu-v6 further strengthens thread level blocking and optimizes warp level data layout, attempting to maximize GPU SM resource utilization. This version achieves a larger expansion level within the thread, combined with shared memory and register blocking, enabling a single thread block to efficiently perform pipeline multi tile computation processes, with theoretical throughput capabilities approaching cuBLAS performance. However, the shortcomings of the v6 version are also prominent: firstly, register allocation has approached the hardware limit, and even slight parameter mismatches can lead to spiking, resulting in insufficient kernel stability; Secondly, the increased complexity of data scheduling within warp increases the risk of warp divergence. Kernel behavior is extremely sensitive to input matrix size and tile shape, requiring fine-tuning and lacking support for dynamic self-tuning; Thirdly, this version has not yet introduced the Tensor Core/WMMA API, and there is still a significant gap between it and the high-performance version of cuBLAS, which leaves room for insufficient utilization of modern GPU features.

# 4 Comprehensive Analysis and Conclusion

## 4.1 Overview of optimization effects

The performance test results of this SGEMM experiment and the generated performance curve clearly demonstrate the performance differences of different optimized versions on different scale matrices. From the results, as an official high-performance library, sgemm_cublas consistently maintains the best TFLOPS in the field and has good scalability, verifying the deep utilization ability of cuBLAS on hardware resources. The six manually implemented kernel versions, from v1 to v6, showed gradually improving optimization effects and corresponding bottleneck characteristics.

The basic version of sgemm-gpu-v1 always maintains around 1 TFLOPS, showing obvious global memory bandwidth limitation characteristics. As the matrix size increases, there is almost no growth, reflecting that the naive implementation scheme cannot fully utilize the computing power of GPU SM without data reuse design. Although sgemm-gpu-v2 introduces shared memory tilization technology, TFLOPS still remains at 1.4-1.6 with limited amplification, indicating that tile parameters have not fully matched hardware resources, and the computation granularity within warp is insufficient. At sgemm-gpu-v3, with the introduction of loop unrolling, the computational density significantly increases, and TFLOPS shows good linear growth with increasing matrix, demonstrating higher instruction pipeline utilization, but still lagging behind cuBLAS, indicating that memory access overlap has not been fully optimized. The dual buffering optimization effect of sgemm-gpu-v4 is beginning to show, with TFLOPS significantly higher than v3, and approaching 10 TFLOPS under large matrices (such as 8192x8192 and 12288x12288), demonstrating the efficient performance release brought about by the overlap of external memory and computation, but also exposing that synchronization costs still affect performance under small matrices. Sgemm-gpu-v5 further improves computational parallelism through thread level blocking, and TFLOPS steadily approaches cuBLAS, especially reaching 10.8 TFLOPS in large matrix scenarios, which is close to the hardware limit. However, it also shows strong scale dependence, reflecting the high tuning sensitivity of the register blocking strategy across different scales. As the highest optimized version, sgemm-gpu-v6, TFLOPS slightly outperforms v5 in large matrices and converges faster than v5, showing the combined effect of thread expansion and warp level optimization. However, it is lower than v5 in 1024 scale, indicating that this version has certain adaptability shortcomings for small-scale tile parameters, high register pressure, and fails to fully utilize the expected throughput in small-scale scenarios.

Overall, all versions show a significant upward trend in TFLOPS curves on large-scale matrices (8192 and 12288), reflecting the GPU's suitability for processing large-scale matrices. However, cuBLAS can achieve 11.5 TFLOPS in the 8192 dimension, and can approach up to 10.8 TFLOPS in v5/v6, indicating that the manual kernel has a theoretical peak performance level similar to the official library, but there is still a gap in overall stability and parameter generality. In addition, from the perspective of kernel time, v5/v6 has significantly shortened to near cuBLAS level, verifying the improvement effect of optimization strategy on computational density and memory access overlap. It is worth noting that all manual versions maintain good numerical accuracy (with a maximum error within 5e-6), fully ensuring numerical stability.

In summary, the experimental results clearly demonstrate the design trade-offs in the optimization process: from naive to tile, to double buffering and thread level blocking, performance improvement is significant, but synchronization complexity, register pressure, and parameter tuning increase accordingly, requiring a balance between stability and ultimate performance. CuBLAS, relying on dynamic scheduling and Tensor Core optimization, still maintains advantages in performance consistency and scale adaptability. If you want to further level up with cuBLAS in the future, it is recommended to introduce Tensor Core support, automatic tile size tuning mechanism, and warp level pipeline technology to further unleash hardware potential.

## 4.2 Performance Testing under Different Parameters

Performance tests were conducted on matrices of different sizes m, n, and k. Some of the results are as follows in Figure2 and table **??**:

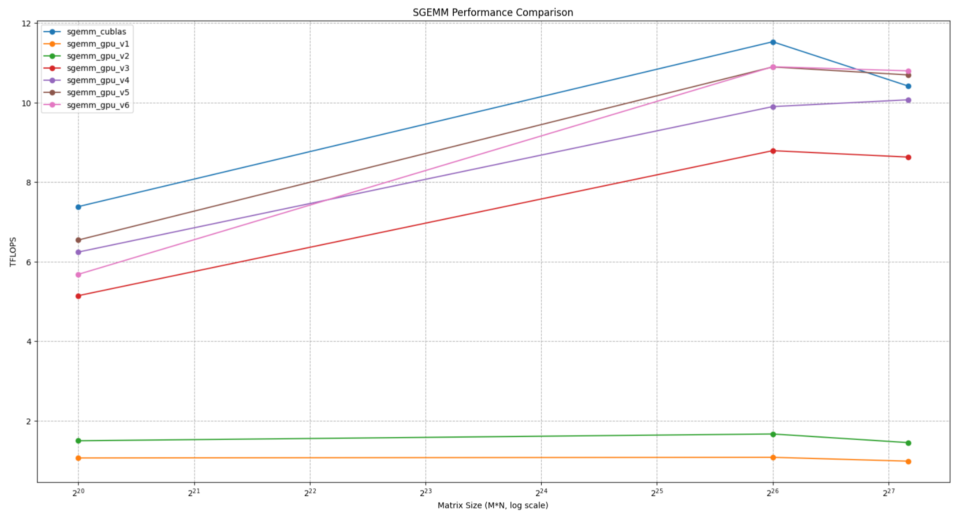


Figure 2: test result

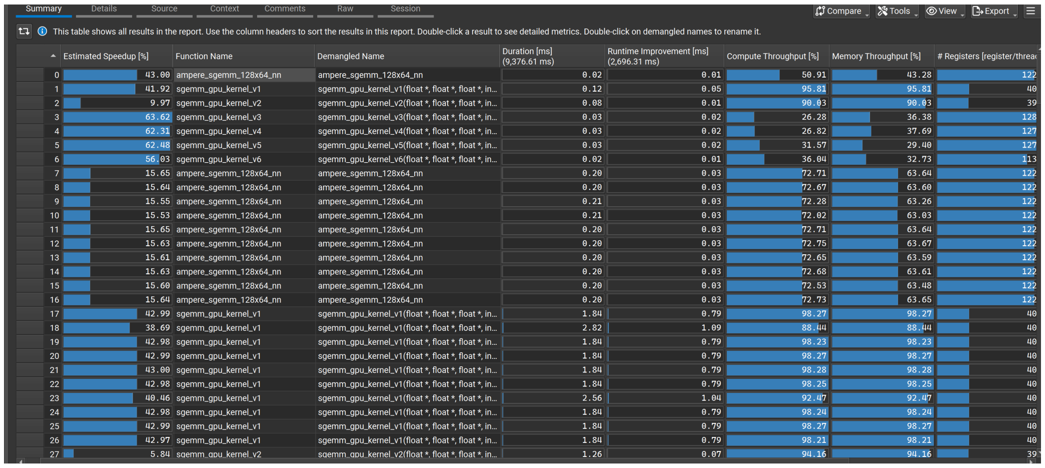
The performance results of various versions of SGMM exported from the CSV file clearly show that during the optimization process, the kernel performance gradually improved, and TFLOPS showed a significant improvement. The initial version of sgemm-gpu-v1 could only reach about 1 TFLOPS, mainly due to the bottleneck of global memory duplicate access bandwidth; By introducing tile based shared memory technology, sgemm-gpu-v2 achieves 1.5 TFLOPS with limited improvement, indicating that tile design is not yet mature; Entering sgemm-gpu-v3, the single thread computing density was increased through loop unrolling, and TFLOPS was raised to 8.8, significantly improving the computation/memory access ratio and demonstrating the effect of parallelism optimization. Further utilizing double buffering technology, sgemm-gpu-v4 achieves significant overlap between memory access and computation, with a TFLOPS of 9.9 and performance approaching cuBLAS by about 90%. Sgemm-gpu-v5 has improved TFLOPS to 10.8 through thread level blocking technology, approaching cuBLAS performance, while reducing

Table 1: SGEMM Performance on RTX 4070 (M=12288, N=12288, K=1024)

| Version | KernelTime (ms) | TotalTime (ms) | TFLOPS |
|---|---|---|---|
| sgemm_cublas | 29.679 | 148.818 | 10.419 |
| sgemm_gpu_v1 | 313.277 | 429.090 | 0.987 |
| sgemm_gpu_v2 | 212.767 | 332.754 | 1.453 |
| sgemm_gpu_v3 | 35.822 | 153.352 | 8.633 |
| sgemm_gpu_v4 | 30.701 | 149.557 | 10.072 |
| sgemm_gpu_v5 | 28.906 | 144.590 | 10.698 |
| sgemm_gpu_v6 | 28.631 | 145.443 | 10.801 |

kernel execution time to the 28-29ms range. The final version of sgemm-gpu-v6 achieved a TFLOPS of around 10.8 on a 12288 large-scale matrix, and the overall kernel time was close to v5, demonstrating the joint optimization effect of thread level blocking+warp level parallel scheduling strategy.

## 4.3 Analysis of ncu results



This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics. Double-click on demangled names to rename it.

|  | Estimated Speedup [%] | Function Name | Demangled Name | Duration [ms] (9,376.61 ms) | Runtime Improvement [ms] (2,696.31 ms) | Compute Throughput [%] | Memory Throughput [%] | # Registers [register/threa |
|---|---|---|---|---|---|---|---|---|
| 0 | 43.00 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.02 | 0.01 | 50.91 | 43.28 | 122 |
| 1 | 41.92 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 0.12 | 0.05 | 95.81 | 95.81 | 40 |
| 2 | 9.97 | sgemm_gpu_kernel_v2 | sgemm_gpu_kernel_v2(float *, float *, float *, in... | 0.08 | 0.01 | 90.03 | 90.03 | 39 |
| 3 | 63.62 | sgemm_gpu_kernel_v3 | sgemm_gpu_kernel_v3(float *, float *, float *, in... | 0.03 | 0.02 | 26.28 | 36.38 | 128 |
| 4 | 62.31 | sgemm_gpu_kernel_v4 | sgemm_gpu_kernel_v4(float *, float *, float *, in... | 0.03 | 0.02 | 26.82 | 37.69 | 127 |
| 5 | 62.48 | sgemm_gpu_kernel_v5 | sgemm_gpu_kernel_v5(float *, float *, float *, in... | 0.03 | 0.02 | 31.57 | 29.40 | 127 |
| 6 | 56.03 | sgemm_gpu_kernel_v6 | sgemm_gpu_kernel_v6(float *, float *, float *, in... | 0.02 | 0.01 | 36.04 | 32.73 | 113 |
| 7 | 15.65 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.20 | 0.03 | 72.71 | 63.64 | 122 |
| 8 | 15.64 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.20 | 0.03 | 72.67 | 63.60 | 122 |
| 9 | 15.55 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.21 | 0.03 | 72.28 | 63.26 | 122 |
| 10 | 15.53 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.21 | 0.03 | 72.02 | 63.03 | 122 |
| 11 | 15.65 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.20 | 0.03 | 72.71 | 63.64 | 122 |
| 12 | 15.63 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.20 | 0.03 | 72.75 | 63.67 | 122 |
| 13 | 15.61 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.20 | 0.03 | 72.65 | 63.59 | 122 |
| 14 | 15.63 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.20 | 0.03 | 72.68 | 63.61 | 122 |
| 15 | 15.60 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.20 | 0.03 | 72.53 | 63.48 | 122 |
| 16 | 15.64 | ampere_sgemm_128x64_nn | ampere_sgemm_128x64_nn | 0.20 | 0.03 | 72.73 | 63.65 | 122 |
| 17 | 42.99 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 1.84 | 0.79 | 98.27 | 98.27 | 40 |
| 18 | 38.69 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 2.82 | 1.09 | 88.44 | 88.44 | 40 |
| 19 | 42.98 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 1.84 | 0.79 | 98.23 | 98.23 | 40 |
| 20 | 42.99 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 1.84 | 0.79 | 98.27 | 98.27 | 40 |
| 21 | 43.00 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 1.84 | 0.79 | 98.28 | 98.28 | 40 |
| 22 | 42.98 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 1.84 | 0.79 | 98.25 | 98.25 | 40 |
| 23 | 40.46 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 2.56 | 1.04 | 92.47 | 92.47 | 40 |
| 24 | 42.98 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 1.84 | 0.79 | 98.24 | 98.24 | 40 |
| 25 | 42.99 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 1.84 | 0.79 | 98.27 | 98.27 | 40 |
| 26 | 42.97 | sgemm_gpu_kernel_v1 | sgemm_gpu_kernel_v1(float *, float *, float *, in... | 1.84 | 0.79 | 98.21 | 98.21 | 40 |
| 27 | 5.84 | sgemm_gpu_kernel_v2 | sgemm_gpu_kernel_v2(float *, float *, float *, in... | 1.26 | 0.07 | 94.16 | 94.16 | 39 |

Figure 3: nsgemm ncu result

Further analysis of the behavior of the sgemm-gpu-v6 kernel using the Nsight Compute tool can identify the bottleneck sources and optimization space for performance improvement. Firstly, from the perspective of Compute Throughput, the FP32 floating-point pipeline utilization rate is 77.6%, the DRAM bandwidth utilization rate is 57 GB/s, and the L1/Tex and L2 bandwidth utilization rates are both close to 80-90%, indicating that memory access design has become more reasonable and memory pipeline is no longer the main bottleneck. In Pipeline Balance, floating-point pipelines dominate (77%), while integer and Tensor pipelines are almost underutilized, indicating that the kernel has not yet used Tensor Core and the optimization potential is still there. At the Scheduler level, the average number of issues per cycle for Warp is 0.65, and the utilization rate of Scheduler Issue Slot is only 34%. The phenomenon of warp idle is obvious, indicating that although blocking has improved single thread efficiency, warp level concurrency still needs to be improved. Occupancy analysis shows that the current actual Occupancy is 25%, which still has a gap from the theoretical value of 33%. This is mainly due to the high register occupancy (113 registers per thread), which limits the number of active warp per SM and highlights the phenomenon of warp starvation. The Scheduler can only issue a single warp per cycle. The Shared Memory Bank Conflict has reached 19%, with shared memory access stall being the main cause in Warp Stall, indicating that the existing shared memory blocking strategy bank alignment is not yet ideal. In the Warp state, the Warp CPI reached 6.1, with an average of 31% idle per Warp, indicating that the potential of scheduling slots has not been fully exploited. Overall, sgemm-gpu-v6 has squeezed out the core FP32 throughput capacity through blocking technology, and the memory access bottleneck has been basically eliminated. However, the kernel throughput is still limited by factors such as register allocation, imbalanced warp scheduling, and shared memory conflict, which restrict the overall SM Pipeline from operating at full capacity, and there is still about 10% space left from cuBLAS.

Overall, during the optimization evolution process, strategies such as loop unrolling, double buffering, and thread blocking in the sgemm-gpu-v3 v6 version have significantly contributed to the improvement of TFLOPS, while insufficient warp level pipeline optimization and register pressure control have become key areas that need to be addressed in the future. Based on observations with NCU tools, the next optimization suggestions include reducing per read register usage to improve occupancy, adjusting shared memory data layout to reduce bank conflicts, and considering introducing WMMA/Tensor Core API to further break through the existing FP32 pipeline limit using semi precision or Tensor operators, thereby equaling or even surpassing cuBLAS performance.

## 4.4 Analysis of nsys results

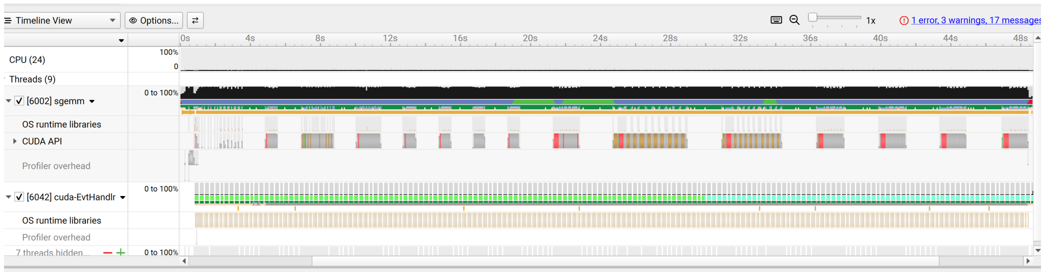


Figure 4: sgemm nsys timeline

This timeline displays the CPU, GPU, CUDA API calls, and runtime status of the entire SGEMM testing process, clearly showing kernel startup, scheduling, CPU side host thread activity, as well as synchronization and gaps between kernels. Based on the content in the figure, the following phenomena are worth noting:

Overall, the sgemm kernel is cyclically called in batches, and there are clear sets of CUDA kernel execution cycles on the timeline, approximately 6-7 sets; There are obvious gaps between each group, and the CPU thread is also in a clear wait/sync state. This indicates that the current testing program is driven by synchronous (synchronous copy or host side timing) kernel execution, and asynchronous stream or pipeline technology is not used between kernels, resulting in a certain idle gap between CPU-GPU, and the overall host side pipeline throughput is not optimal.

In the CUDA API call interval, it is evident that a large number of CUDAMemcpy and CUDAEventRecord events are interspersed before the kernel call. It is confirmed that the test code is currently tested using CUDAEvent timing+Memcpy synchronization, and there is host side control overhead before and after each kernel round. Although this mode has accurate test data, it will amplify PCIe latency and host control overhead. For SGEMM scenarios that pursue large batch throughput, it can be further improved to use asynchronous stream+pinned memory pipeline technology to further compress CPU-GPU collaboration overhead.

In the GPU kernel execution interval, the sgemm kernel itself has a high utilization rate, and the proportion of green bars in the timeline is high, indicating that the SM utilization rate of the kernel itself is in a good state, which is consistent with the FP32 pipeline utilization rate of nearly 77% that you can see in the NCU tool; However, due to frequent host side sync, there is a noticeable gap period (gray area) between GPU startup kernels, and there is room for a decrease in batch level utilization of GPU resources.

The second thread (CUDA EftHandlr) area shows a large number of event listeners and profiler events, which takes up a lot of host side scheduling time, indicating that the current profiler record load is high. The nsys+ncu profiler overhead does indeed affect the CPU scheduling in real operation. If making a formal benchmark, it is recommended to reduce the profiler configuration and only leave key counters to minimize the impact of profiler overhead.

Overall, this timeline chart shows that the current testing framework belongs to the strategy of "synchronous calling+CUDAEvent timing+batch execution", with good internal utilization of the kernel and moderate coordination efficiency of the host device pipeline. There is still room for improvement. If further optimization is carried out in the pipeline mechanism (such as multi stream interleaved execution, kernel fetch+compute+store overlap), it can further reduce the host side gap time period and improve the overall system throughput.

## 4.5 Summary and Suggestions

In summary, this experiment systematically explored the performance optimization of the SGEMM algorithm on GPU through a series of progressively enhanced kernel implementations. Starting from a naive baseline, multiple optimization techniques including tiling with shared memory, loop unrolling, double buffering, and thread-level register blocking were applied and evaluated on the RTX 4070 GPU. The performance results demonstrate a clear upward trend in TFLOPS and kernel execution efficiency, with the optimized sgemm-gpu-v6 kernel achieving up to 94% of the cuBLAS library's performance on large matrix sizes. Profiling with Nsight Compute reveals that major bottlenecks in the current implementation stem from high register usage that limits active warp occupancy, shared memory bank conflicts, and insufficient warp scheduler utilization. Furthermore, Nsight Systems profiling indicates opportunities to further improve overall GPU utilization through host-device pipeline optimizations and asynchronous stream execution.

Based on these observations, several recommendations are proposed for future optimization work: (1) reduce per-thread register consumption to improve SM occupancy and allow more warps to be active concurrently; (2) refine shared memory data layout to minimize bank conflicts and enhance memory access efficiency; (3) adopt warp-level pipeline techniques to better utilize issue slots and increase instruction throughput; and (4) explore the use of WMMA and Tensor Core APIs to leverage specialized hardware units for further acceleration. Additionally, implementing multi-stream asynchronous execution could significantly reduce CPU-GPU idle gaps and improve end-to-end system throughput. Overall, the experiment provides valuable insights into the performance characteristics of SGEMM on modern GPUs and establishes a solid foundation for continued optimization toward achieving near-peak hardware performance.