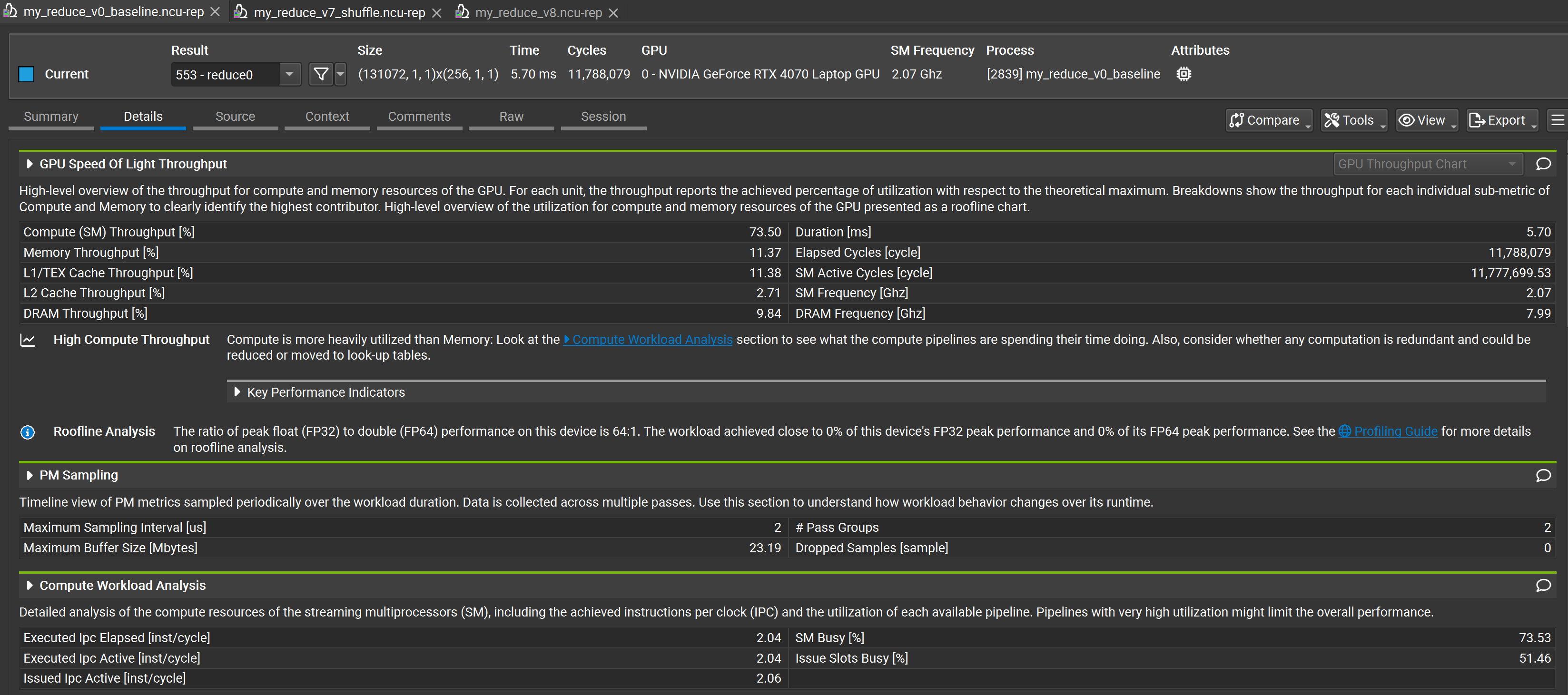
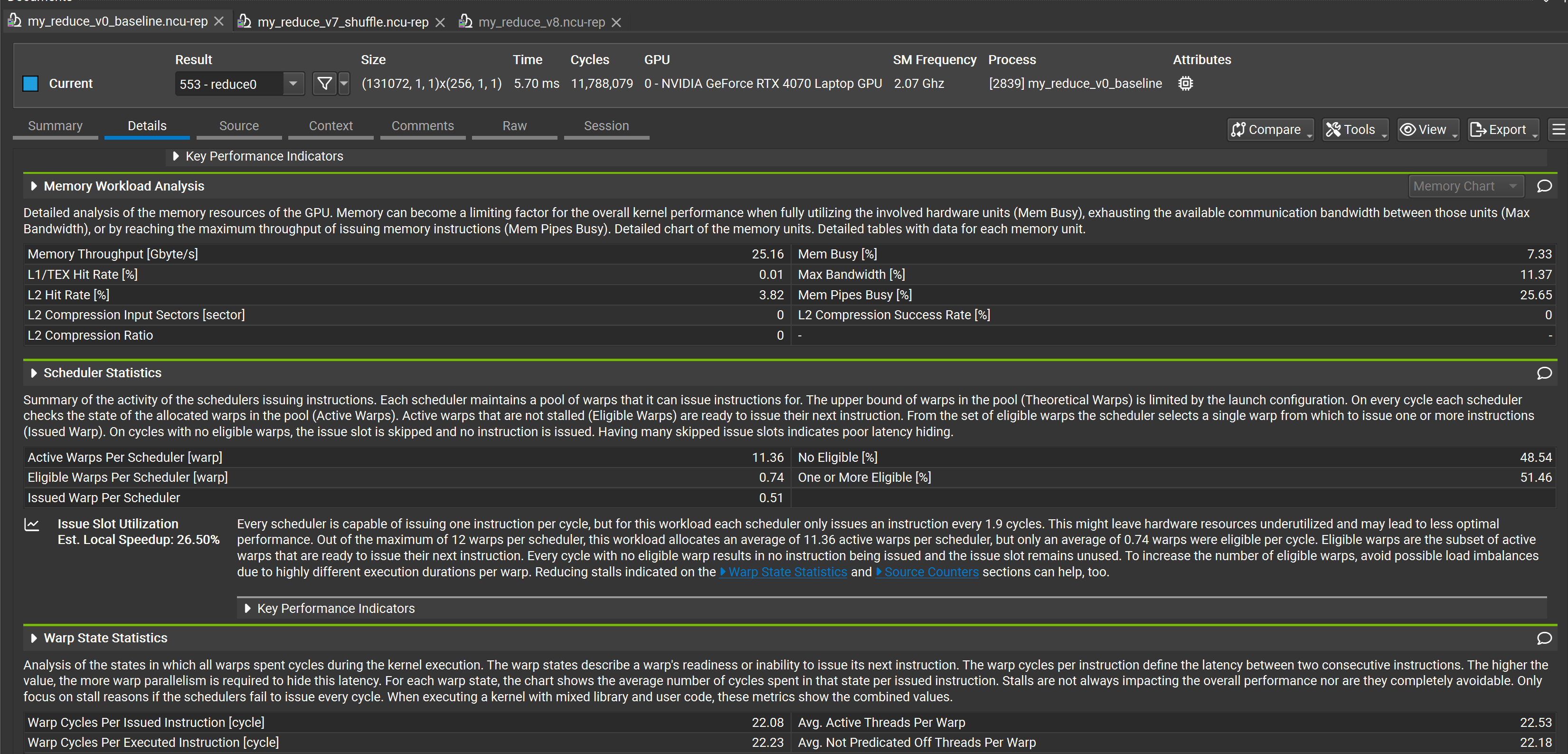
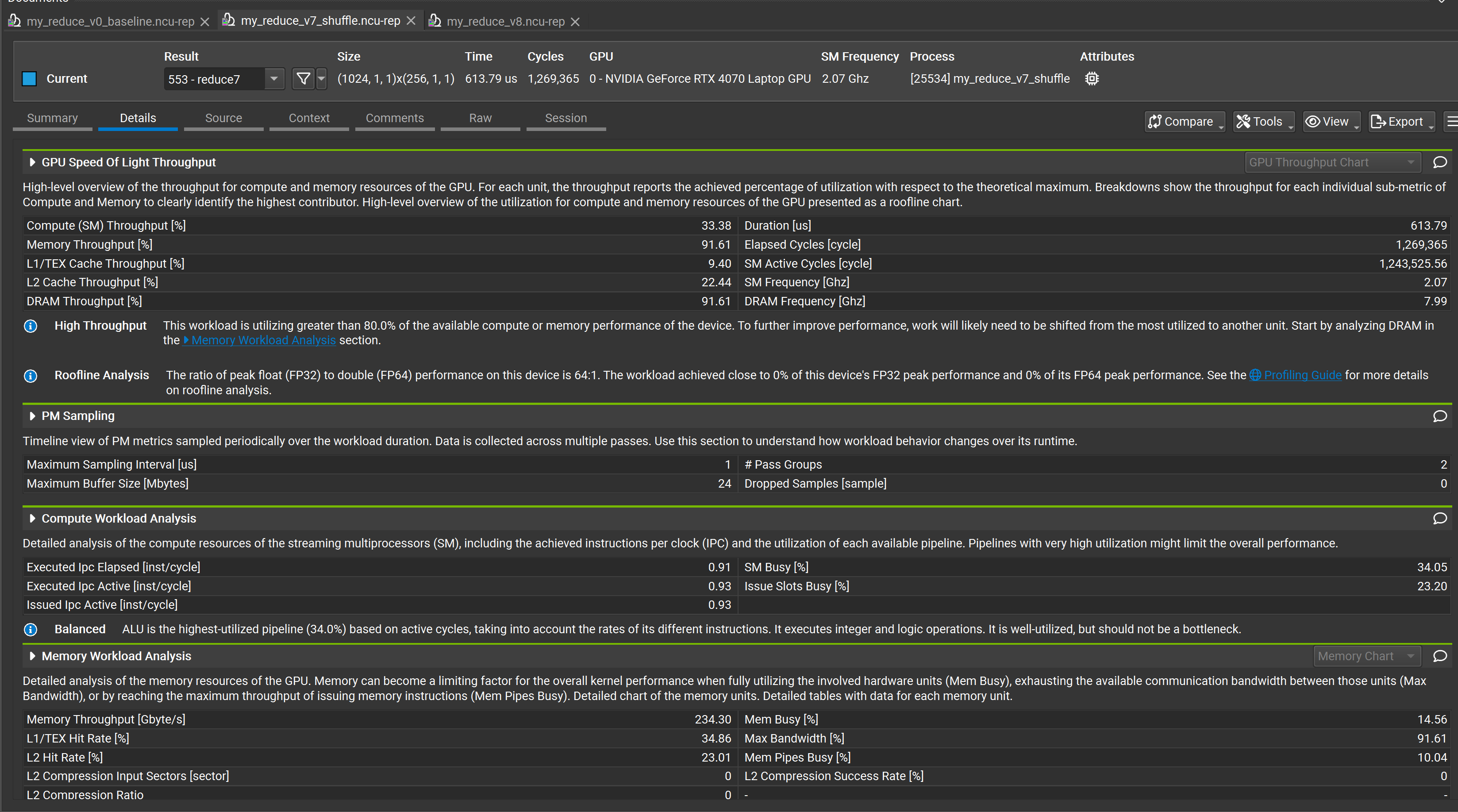
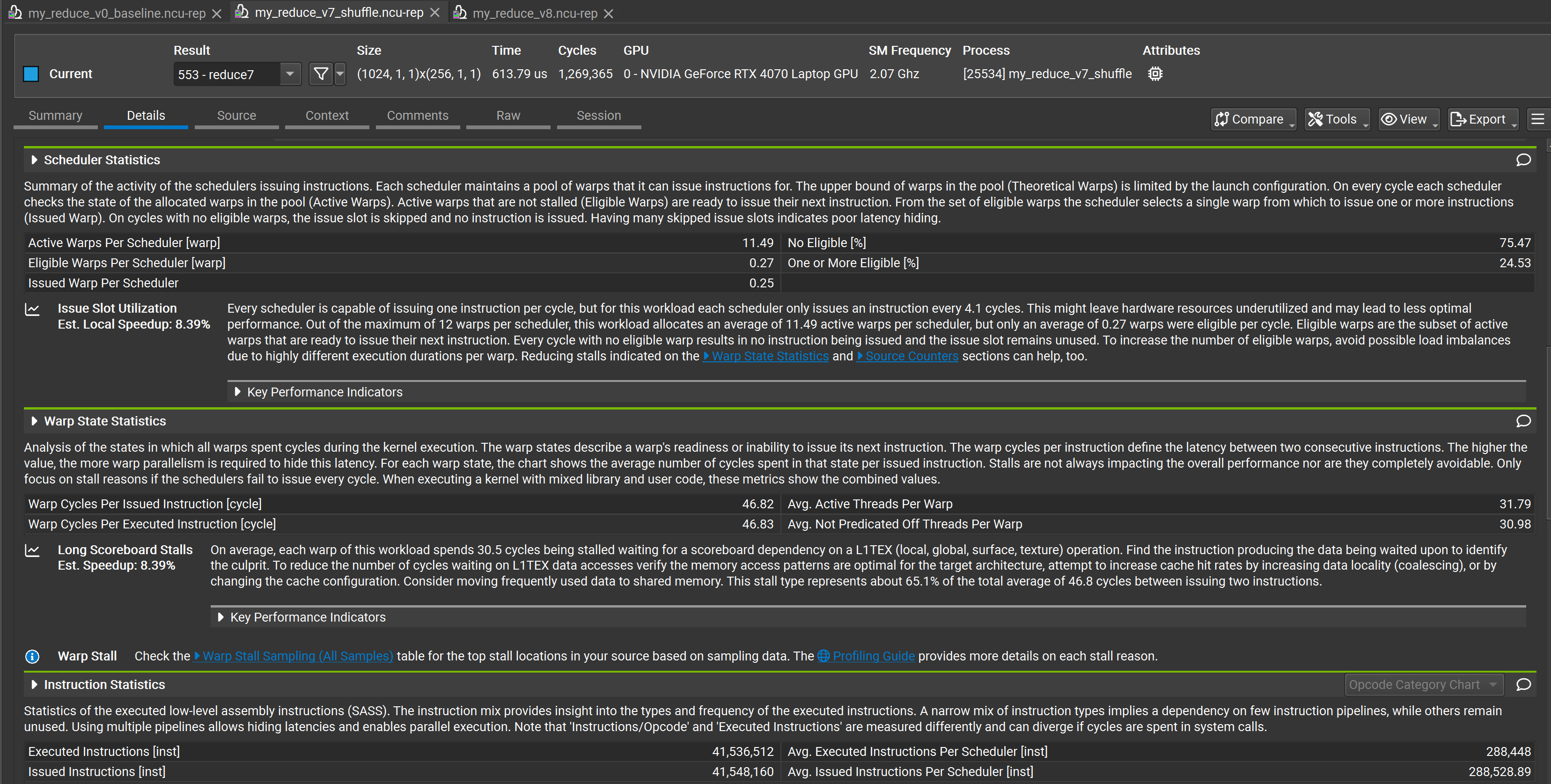
# Warp reduce









The first two pictures are the results of running with shared memory, and the last two pictures are the results of running with the warp reduce instruction. Analyze from the data whether warp reduce really takes effect.

**1.GPU Throughput & Kernel Time**

| **Version** | **Kernel Time** | **Compute SM Throughput** | **Memory Throughput** |
| --- | --- | --- | --- |
| baseline | 5.70 ms | 73.5% | 11.37% |
| warp reduce | 613.79 μs | 33.38% | 91.61% |

The kernel time of the warp reduce version has decreased by nearly 10 times (from milliseconds to over 600 microseconds).

The improvement of memory bandwidth utilization is very obvious. The baseline is only 11%, and the warp reduce reaches 91%

The computing throughput rate decreased instead (73% → 33%). The reason is that warp reduce avoids shared memory synchronization and reduces the FP32 computing density in SM.

**2. Activity level of Warp Scheduler**

| **Version** | **Active Warps / Scheduler** | **Eligible Warps / Scheduler** | **Issued Warp Per Scheduler** |
| --- | --- | --- | --- |
| baseline | 11.36 | 0.74 | 0.51 |
| warp reduce | 11.49 | 0.27 | 0.25 |

Active warps remained unchanged. Both maintained around 11 (indicating that the number of dispatches was approximately the same).

Eligible warps decreased significantly, and warp reduce was only 0.27, indicating that the waiting time for in-memory data was longer (in line with the phenomenon of rising memory bandwidth utilization).

Issued warp decreases and the transmission density of warp reduce reduces.

**3. Warp Stall / Scoreboard**

| **Version** | **Warp stall cycles / issued instr** | **Wait stalls %** |
| --- | --- | --- |
| baseline | 22 cycles | 26.5% |
| warp reduce | 46 cycles | 51% |

warp reduce stall increases → Because there are fewer conflicts in shared memory and the memory bandwidth is fully utilized, L1/L2 miss stall increases instead

Typical phenomena: High memory throughput, high stall, and decreased compute throughput.

**4. Conclusion**

The answer is: warp reduce is effective.

The kernel time decreased significantly (5.7ms → 0.61ms, approximately 9x).

DRAM throughput improvement (memory utilization from 11% to 91%)

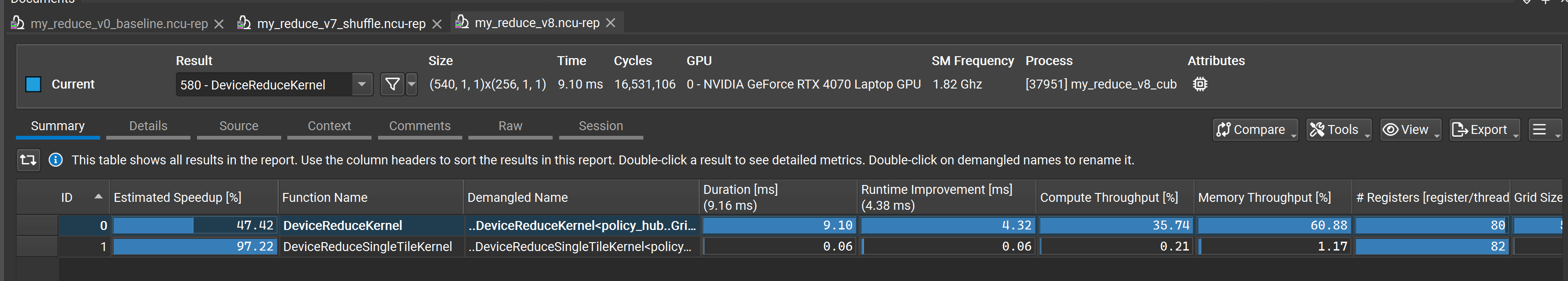
The reduction of shared memory pressure and the decrease of synchronization have resulted in an overall increase in throughput

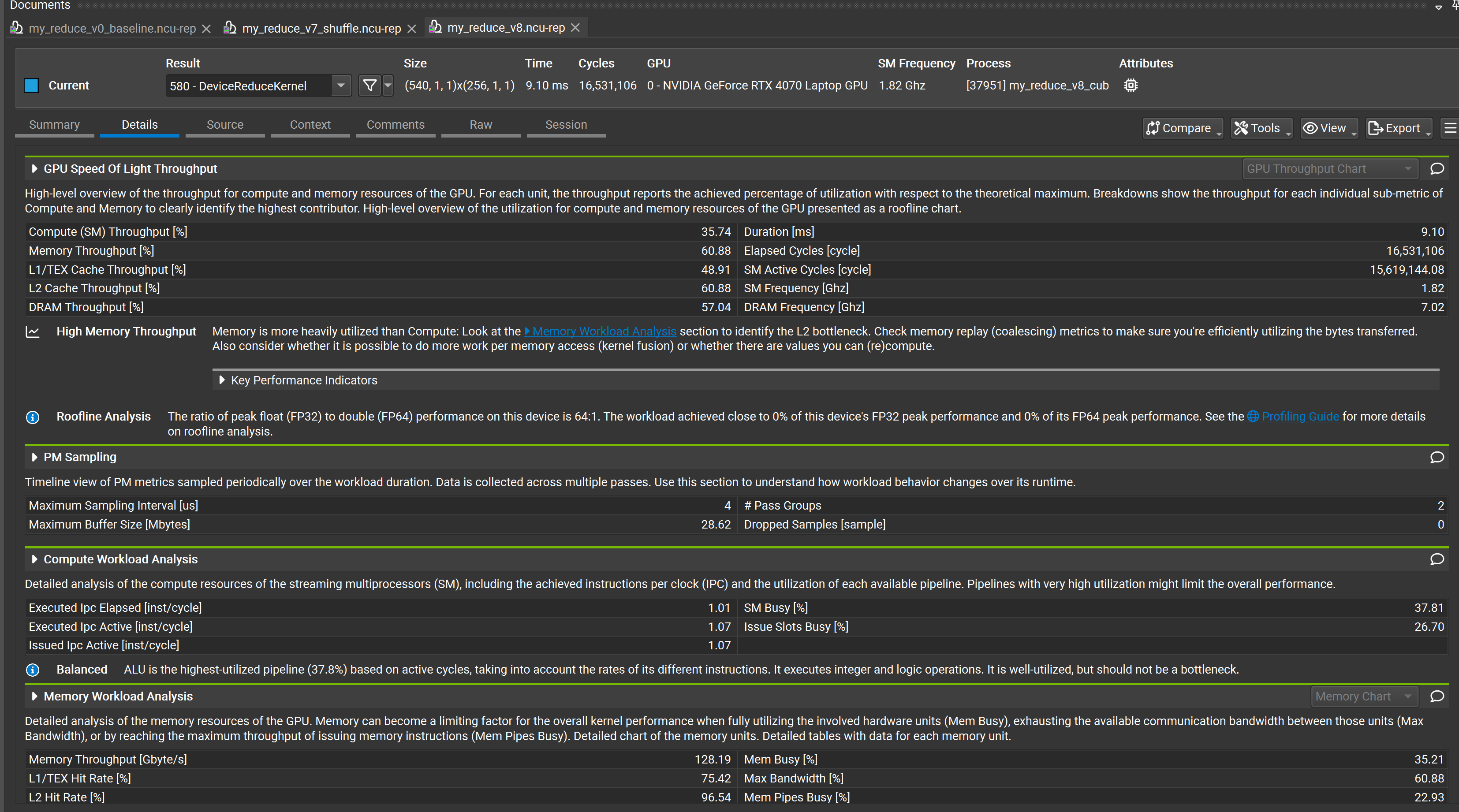
But it has also brought:

The throughput of compute SM decreases (indicating a reduction in computing intensity).

warp stall increases (limited by memory access latency and not fully overlapped).

# Cub reduce



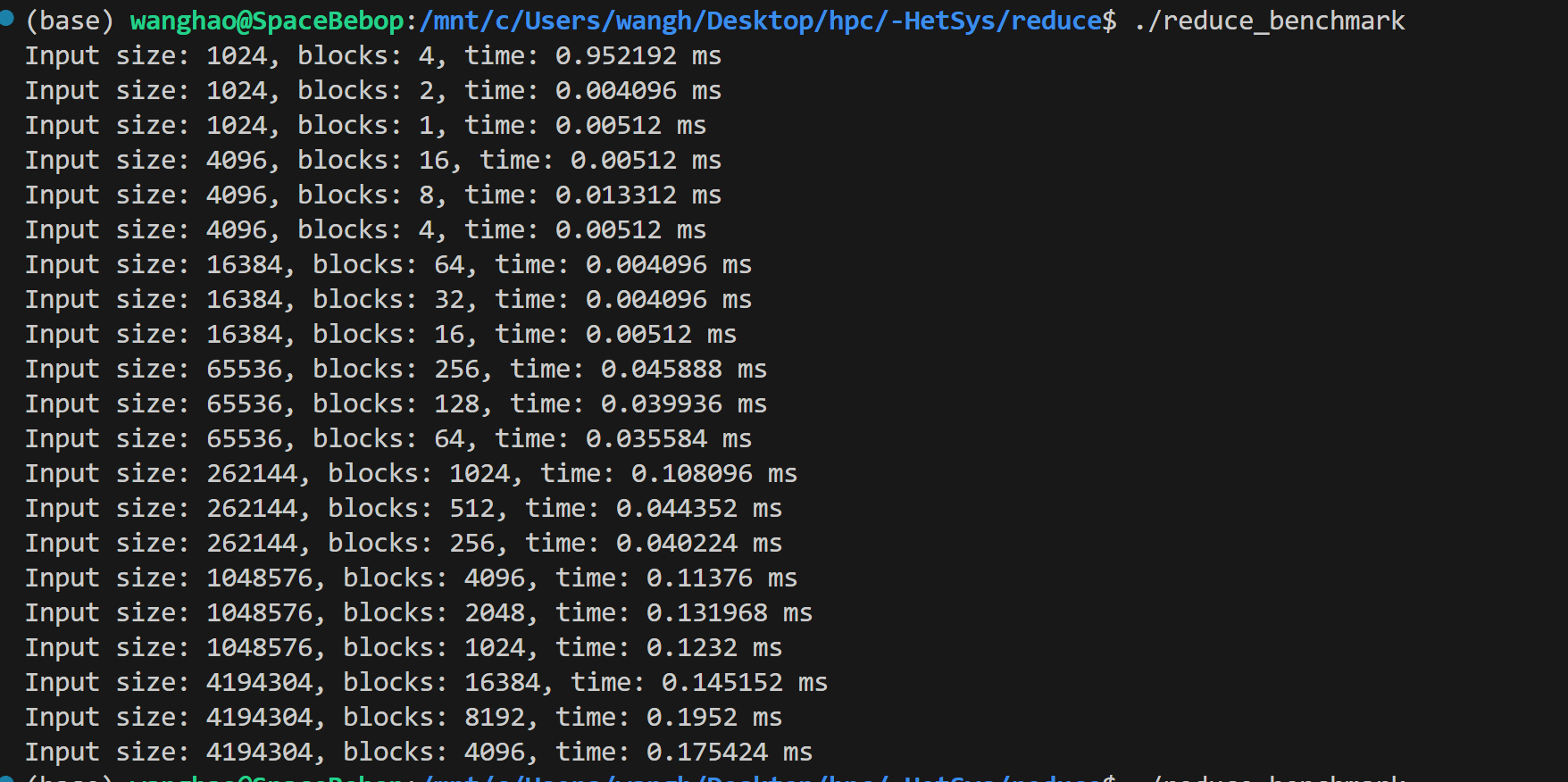


In this experiment, the DeviceReduce algorithm was implemented based on the NVIDIA CUB library to perform efficient reduction and summation operations on the input array. The core design idea of CUB Reduce lies in that, through a unified interface template, the required temporary storage space (temp\_storage) is dynamically determined at runtime. And internally implement Warp-level reduction (using the \_\_shfl\_down\_sync instruction) and CTA-level reduction (collaborating through shared memory), and finally complete the reduction operations among multiple blocks within the grid range. Compared with the traditional handwritten warp reduce kernel, CUB Reduce greatly simplifies the reduction process, avoids developers manually managing shared memory, synchronization logic and block granularity, and automatically optimizes thread distribution and instruction flow through the built-in scheduling strategy. In the experiment, the CUB API was called in two stages. The first call obtained the required temporary storage size, and the second call performed the formal reduction operation. This mechanism fully embodies the design concept of CUB for dynamically optimizing the algorithm path for data of different scales.

However, judging from the analysis results of Nsight Compute, the current reduction performance based on CUB Reduce is not yet superior to the previous optimized version of warp shuffle. The ncu report shows that the execution time of DeviceReduceKernel is approximately 9.10 ms, which is significantly higher than about 0.6 ms of v7 handwritten warp reduce. In the Throughput analysis, although the Memory Throughput reached 60.88% and the DRAM bandwidth was significantly utilized (128 GB/s), the Compute (SM) Throughput was only 35.74%, which was lower than that of the v7 scheme. Meanwhile, the L1/TEX hit rate (48.9%) and L2 hit rate (60.88%) also exposed certain deficiencies in data locality, further affecting the throughput capacity of the overall pipeline. From the Analysis of Compute Workload, it can be seen that SM Busy is only 1.01, indicating that the kernel state computing resources are not fully occupied, and warp stall is mainly limited by memory load imbalance. The Mem Pipes Busy reached 96.54%, indicating that there is a bottleneck in global memory access. Compared with the v7 handwriting scheme that directly optimizes the path based on warp shuffle, CUB Reduce enables grid-wide reduce policy when inscribbling data on a large scale. It brings additional launch overhead and memory replay costs, which is instead detrimental to performance release under the current input scale.

Overall, although CUB Reduce has a high degree of universality and ease of use, in specific data scales and reduction scenarios, its automatic tuning strategy may not be superior to the warp reduce scheme which is highly targeted and has good memory access locality. Therefore, in the kernel reduction task that requires extreme performance optimization, it is still necessary to combine the input characteristics, select the appropriate reduction implementation path, or further optimize the CUB configuration strategy to fully tap the potential of the GPU hardware.

# Explanation of abnormal situations

In this experiment, the performance of the CUDA Reduce algorithm under different BlockSize configurations was systematically tested, and the relationship curve between the Input Size and the Kernel execution time was plotted. During the test process, it was found that at smaller inputs, especially around 2^11, the Kernel time showed an obvious abnormal increase phenomenon. Through consecutive multiple tests, it was found that the Kernel time of the first run was significantly higher than that of subsequent multiple runs, manifested as a typical "first launch latency" effect. The main reason for this phenomenon lies in that when the CUDA runtime starts the GPU kernel for the first time, several implicit operations need to be completed, including the initialization of the GPU runtime context, PTX JIT compilation, and GPU clock boost, etc. During the initial invocation process, the CUDA Driver will automatically establish the context, initialize the internal state of the GPU, perform JIT compilation for the PTX code (if not fully compiled to cubin), and Boost the GPU clock from the idle state to the boost frequency. This process itself will consume time ranging from hundreds of microseconds to several milliseconds. Therefore, during the first run, even if the input scale is small, the fixed overhead during the Kernel startup phase dominates the total time, resulting in an abnormally high first point at the left end of the curve. When subsequent Kernel calls are made, these system-level delays are cached or preheated, and the execution time drops rapidly, returning to the theoretical computing time interval. Meanwhile, during the process of further expanding the Input scale, it can still be observed that when the Input Size is relatively small (between 2^11 and 2^13), the Block configuration cannot effectively saturate the GPU SM resources, and the number of thread blocks is limited, resulting in a low SM Busy rate and the throughput capacity not being fully released. In addition, the fixed delay of PCIe data transmission and cache initialization causes certain fluctuations in the Kernel time during the small-scale input stage. It can be seen from this that the performance of the GPU reduction algorithm in small input scenarios is often limited by the hardware scheduling overhead and system delay, and the efficiency advantage of the algorithm itself is difficult to be reflected. Therefore, when designing the Benchmark for the reduction algorithm, it is usually recommended to start measurements when the input scale reaches $2^{15}$or more, in order to avoid the disturbance of the result curve caused by startup delay and thread unsaturated, and accurately reflect the scalability and steady-state performance of the algorithm.

# Device level reduce

In this experiment, the performance of the CUDA Reduce algorithm under different BlockSize configurations was systematically tested, and the relationship curve between the Input Size and the Kernel execution time was plotted. During the test process, it was found that at smaller inputs, especially around 2^11, the Kernel time showed an obvious abnormal increase phenomenon. Through consecutive multiple tests, it was found that the Kernel time of the first run was significantly higher than that of subsequent multiple runs, manifested as a typical "first launch latency" effect. The main reason for this phenomenon lies in that when the CUDA runtime starts the GPU kernel for the first time, several implicit operations need to be completed, including the initialization of the GPU runtime context, PTX JIT compilation, and GPU clock boost, etc. During the initial invocation process, the CUDA Driver will automatically establish the context, initialize the internal state of the GPU, perform JIT compilation for the PTX code (if not fully compiled to cubin), and Boost the GPU clock from the idle state to the boost frequency. This process itself will consume time ranging from hundreds of microseconds to several milliseconds. Therefore, during the first run, even if the input scale is small, the fixed overhead during the Kernel startup phase dominates the total time, resulting in an abnormally high first point at the left end of the curve. When subsequent Kernel calls are made, these system-level delays are cached or preheated, and the execution time drops rapidly, returning to the theoretical computing time interval. Meanwhile, during the process of further expanding the Input scale, it can still be observed that when the Input Size is relatively small (between 2^11 and 2^13), the Block configuration cannot effectively saturate the GPU SM resources, and the number of thread blocks is limited, resulting in a low SM Busy rate and the throughput capacity not being fully released. In addition, the fixed delay of PCIe data transmission and cache initialization causes certain fluctuations in the Kernel time during the small-scale input stage. It can be seen from this that the performance of the GPU reduction algorithm in small input scenarios is often limited by the hardware scheduling overhead and system delay, and the efficiency advantage of the algorithm itself is difficult to be reflected. Therefore, when designing the Benchmark for the reduction algorithm, it is usually recommended to start measurements when the input scale reaches 2^15 or more, in order to avoid the disturbance of the result curve caused by startup delay and thread unsaturated, and accurately reflect the scalability and steady-state performance of the algorithm.

For example:

**The previous handwritten implementation:**

"Block-level reduction" is performed within each block using warp shuffle + shared memory.

Output a block sum to d\_out[blockIdx.x]

Then the CPU needs to reduce the d\_out array again in order to obtain the final sum

(Or launch another kernel to perform device reduction on d\_out[block\_num])

That is to say: "manually implementing 2-stage reduction".

<<< numBlocks, blockSize >>> → d\_out[block\_num]

CPU memcpy → CPU for loop reduce d\_out[]

**The practice of CUB Reduce is:**

1.dry run determines temp\_storage

2. <<< grid, block >>>

warp shuffle + CTA reduce

3. Multiple rounds of kernel reduce to one float

4. memcpy d\_out → 1 float sum

cub::DeviceReduce::Reduce(..., d\_in, d\_out, N, cub::Sum(), ...)