Performance Optimization of GPU Reduce Algorithm

June 15, 2025

Abstract

This report presents the performance analysis and optimization of the reduce algorithm on GPU. The experiment focuses on several optimization techniques applied to the baseline reduce implementation and compares the performance results against the peak throughput of the RTX 4070 GPU. Various optimizations are tested including warp divergence resolution, bank conflict avoidance, idle thread utilization, and hardware-specific instruction use, with performance metrics such as execution time and bandwidth utilization discussed.

1 Experimental background and purpose

The Reduce algorithm, as a fundamental parallel computing paradigm, has wide applications in fields such as scientific computing and deep learning. This experiment is based on the NVIDIA RTX4070 GPU platform. For the summation operation of a 32M float type array, it systematically explores the complete path from the basic implementation to highly optimized.The experiments not only focus on the improvement of performance indicators, but also focus on the analysis of the principle behind each optimization technique and its actual effect, aiming to provide reusable optimization methodologies for GPU high performance computing.

2 Experimental environment and procedures

The experiment adopts the NVIDIA RTX4070 GPU, whose theoretical global memory bandwidth is 504GB/s. The test data is a float array with a scale of 32M. The execution time consumption of kernel functions of each version is accurately measured through the Nsight tool, and the effective bandwidth utilization rate is calculated. The performance comparison adopts the speedup ratio index, with the baseline implementation as the benchmark, to quantify the improvement amplitude of each optimization stage.

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

```
mkdir build
cd build
cmake ..
make
```

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.

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 (reduce0)

The baseline implementation adopts the most intuitive tree reduction method. Each block is assigned 256 threads, corresponding one-to-one with the 256 data elements being processed. The thread first loads global memory data into shared memory and then performs binary reduction through multiple rounds of iteration: In the KTH round of iteration, the thread that satisfies $tid\%(2^k) == 0$ adds the element at the current position to the element at an interval of $2^{(k-1)}$. Although this method has a simple logic, it has a serious warp divergence problem - in each round of iteration, only some threads are active, while the inactive threads within the same warp still need to participate in the instruction stream but make no actual contribution. What's more serious is that when $s = 2^k$, memory access will trigger 2^k bank conflicts. For instance, when s=2, addresses 0 and 32 belong to the same bank, resulting in memory access serialization.

From the analysis results of nsys in Figure1, it can be seen that there is a serious performance bottleneck problem in this baseline implementation. The overall execution time reaches 970ms and the main time consumption is concentrated on two cudaMemcpy operations, indicating that the data

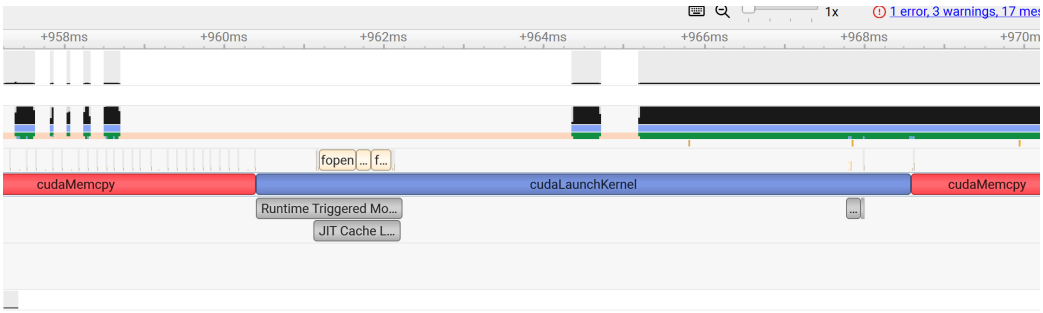


Figure 1: nsys v0

transmission between the host and the device has become the key factor restricting performance. This memory bottleneck phenomenon indicates that the current implementation has problems such as excessive data transmission volume or inefficient transmission methods, which leads to the complete inability to exert the powerful computing capacity of the GPU - the kernel execution time is even completely masked by memory operations and cannot be shown in the analysis diagram.

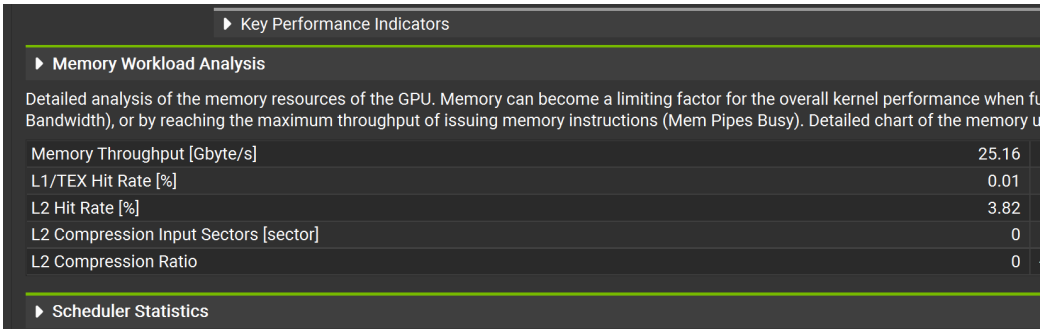


Figure 2: ncu v0 throughput

The experimental results show that the baseline bandwidth is only 25.16GB/s and the utilization rate is less than 5% in Figure2 and 3. This indicates that simple parallel transplantation not only fails to leverage the advantages of the GPU, but also leads to performance degradation due to architectural characteristics. The fundamental problem lies in the failure to follow two basic principles of GPU: maintaining the consistency of thread execution paths within warp and avoiding bank conflicts of shared memory.

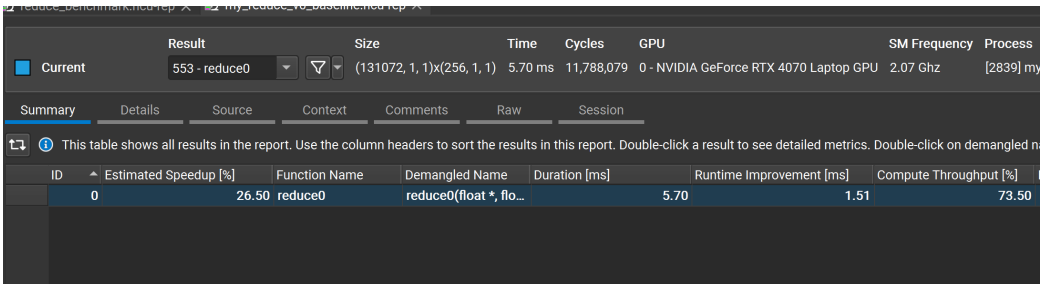


Figure 3: ncu v0 time

3.2 Optimization Technique 1: Eliminate Warp Divergence (Reduce 1)

reduce1 solves the warp divergence problem by reconstructing the reduction index calculation method. The new method calculates index=2*s*tid and performs addition only when index is less than blockDim.x. This design ensures that in the first few rounds of iterations, the entire warp either performs addition entirely or skips all of it. For instance, when blockDim.x=256, threads 0 to 127 in the first round of iteration (s=1) meet the condition, corresponding to the first 4 warps being fully active and the last 4 WarPs being completely idle, thus avoiding internal branches within the warp.

This optimization increases the bandwidth to 32.08GB/s and the speedup ratio to 1.27x. However, the analysis reveals that when s reduces to the warp size (such as when s=16), there will still be a situation where some threads are active within some warps. Furthermore, this indexing method intensifies bank conflicts. The access mode with an odd step size leads to multi-path conflicts, becoming a new performance bottleneck.

3.3 Optimization Technique 2: Resolving Bank conflict (reduce2)

reduce2 adopts a reverse reduction strategy to fundamentally eliminate bank conflicts. Change the iteration direction to gradually shift to the right starting from blockDim.x/2. That is, when the first round s=128, threads 0 to 127 accumulate elements 0 to 127 and 128 to 255 respectively. In this access mode, 32 threads of the same warp precisely access 32 consecutive banks of the shared memory, perfectly avoiding conflicts. For instance, thread0 ac-

cesses addresses 0 and 128 (bank0 and bank0), and thread1 accesses addresses 1 and 129 (bank1 and bank1). However, since all threads synchronously access different banks, bank conflicts are completely avoided.

The optimized bandwidth jumped to 61.35GB/s, verifying the significant impact of bank conflicts on performance. However, this version exposed a new problem: low thread utilization. The number of active threads in each iteration is halved. After 128 threads are working in the first round, there are only 64, 32... in the subsequent iterations. One thread is useful while a large number of computing units are idle.

3.4 Optimization Technique 3: Reduce thread utilization

reduce3 innovatively enables each thread to handle two data elements. By adjusting the block configuration to make Num_per_block=2*Thread_per_block, each thread completes the addition of two elements at the initialization stage. This method evenly distributes the arithmetic operation volume to all threads, solving the problem of idle threads in the first few rounds of iterations. For example, when blockDim.x=256, it actually processes 512 elements. All 256 threads have completed 256 addition operations in the first round of loading.

In this experiment, Plan B was also implemented. On the premise of maintaining the core optimization idea unchanged, important adjustments were made to the block configuration. It reduces the number of threads in each block to 128 while keeping each thread still processing two elements. Thus, the amount of data processed by each block becomes 256 elements. This adjustment has brought about several key improvements: Firstly, the increase in the number of blocks enables the GPU's SM to be better utilized, enhancing the overall parallelism. Secondly, a smaller block size means that each block requires less shared memory, which enables each SM to host more blocks and improves resource utilization. Furthermore, the smaller number of threads also simplifies warp scheduling and reduces the overhead of context switching.

This optimization has brought astonishing effects. The bandwidth of plan A has increased to 61.35GB/s and the speedup ratio has reached 1.61x. The bandwidth of plan B has increased to 69.93GB/s and the speedup ratio has reached 2.03x. However, performance analysis shows that when the iteration

reaches the last few rounds (s¡=32), the synchronization operation __ SyncThreads () becomes a bottleneck - at this point, only one warp is working, but it still needs to wait for all threads to reach the synchronization point, resulting in a significant waste of clock cycles.

In terms of actual performance, the superiority of Plan B is mainly reflected in several aspects: The greater number of blocks provides better latency hiding ability. When one block is waiting for memory access, other blocks can immediately perform calculations. A more balanced SM utilization rate avoids the idleness of computing resources. More efficient warp scheduling reduces unnecessary overhead. These factors work together to enable Plan B to achieve higher operational efficiency than Plan A while maintaining the same core algorithm.

3.5 Optimization Technique 4: Reduce Redundancy synchronization

reduce4 introduces the implicit synchronization feature of warp for targeted optimization. When the active threads are completely contained within a single warp (tid¡32), a special warpReduce function is used instead, taking advantage of the natural synchronization feature of SIMD units, completely eliminating the call to __syncThreads (). This function achieves reduction through a fully expanded instruction chain: First, add tid to tid+32, then add tid+16 until tid+1, and the entire process does not require explicit synchronization.

The experimental results prove the effectiveness of this optimization. The bandwidth reaches 81.97GB/s, which is close to 16% of the theoretical peak. However, the performance analysis reveals two potential problems: First, the overhead of circular control instructions (such as s¿¿=1) becomes significant in extreme optimization scenarios; Secondly, when the block size changes, the expansion depth needs to be manually adjusted, reducing the maintainability of the code.

3.6 Optimization Technique 5: Complete Loop expansion (reduce5)

reduce5 achieves full expansion of loops through template metaprogramming. Generate specific expansion code for different block sizes (such as

512/256/128), eliminating all loop judgment and variable update instructions. By using template specialization technology, the compiler can statically generate the optimal instruction sequence. For example, when blockSize=512, a five-level conditional addition instruction chain is directly generated, completely eliminating the loop overhead.

This optimization brings marginal benefits, and the bandwidth slightly increases to 88.03GB/s. On modern GPU architectures, the benefits of full deployment are no longer as significant as those of early Gpus, benefiting from hardware instruction prediction and smarter compilers. This reflects the impact of hardware advancements on optimization strategies.

3.7 Optimization Tip 6: Block configuration tuning (reduce6)

reduce6 is optimized from the dimension of task allocation. By increasing the workload of each block (gridSize=blockSize*2*gridDim.x), the total number of blocks is reduced to better hide memory latency. The experiment found that theoretically, when the number of blocks is set as an integer multiple of the number of SM (the RTX4070 has 60 SM, and it is difficult to take a complete multiple; in this experiment, 1024 is taken), the optimal state is achieved. At this time, the GPU can fully maintain the working saturation state of each SM, while avoiding excessive block competition for resources.

This optimization enables the bandwidth to reach 156.20GB/s. This design achieves a balance in multiple aspects: On the one hand, the sufficient number of blocks ensures that all SMS of the GPU can be fully utilized, avoiding the idleness of computing resources; On the other hand, a larger task granularity reduces the overhead of global synchronization and improves the continuity of computing. This configuration is particularly suitable for the architectural characteristics of modern Gpus and has found the best balance point between task parallelism and data parallelism.

3.8 Optimization Tip 7: Shuffle Command Application (reduce7)

reduce7 uses the Warp-level __shfl_down_sync instructions to replace shared memory communication. This instruction enables threads within warp to directly access the register values of other threads, completely bypassing shared

memory. The specific implementation is divided into two levels: Firstly, the shuffle instruction is used for reduction within each warp. Then, the results of each warp are stored in the shared memory. Finally, the first warp completes the final reduction.

The final bandwidth reached 157.61GB/s, but it was less than 1% higher than reduce6. This indicates that on the RTX4070 architecture, after the aforementioned optimizations, communication overhead is no longer a limiting factor. Further optimization needs to consider deeper issues such as memory access patterns.

4 Comprehensive Analysis and Conclusion

4.1 Overview of optimization effects

Through the seven-step optimization, the performance improvement shows obvious phased characteristics:

The basic optimization stage (reduce1-reduce3) addresses macro design issues and brings about an acceleration of 2.89x.

The fine tuning stage (reduce4-reduce5) optimizes for hardware characteristics and achieves an additional 26% improvement.

In the frontier technology stage (reduce6-reduce7), the marginal benefit decreases, and the total acceleration ratio reaches 5.48x.

Youdaoplaceholder0 Insufficient depth analysis of each scheme

baseline: The GPU execution model is not taken into account at all, and there are structural performance flaws.

reduce1: Although the warp divergence is resolved, a more serious bank conflict is introduced.

reduce2: Perfectly resolves bank conflicts at the expense of thread utilization.

reduce3: Improves thread utilization but exposes synchronization overhead issues.

reduce4: Eliminate redundant synchronization but reduce code maintainability.

reduce5: Loop unrolling offers limited benefits on modern Gpus.

reduce6: Limited by the GPU hardware resource scheduling mechanism.

reduce7: The advantage of the shuffle command is not obvious in extremely optimized scenarios.

4.2 Insights from Architectural Evolution

By comparing the optimization benefits of early Gpus with those of contemporary RTX4070, it can be seen that:

Hardware advancements (such as more intelligent warp scheduling) have reduced the efficiency of some traditional optimization techniques (such as full deployment);

New features (such as the shuffle instruction) need to be coordinated with the overall optimization strategy to achieve the maximum effect;

Optimization needs to take into account the generational differences in architecture; there is no one-size-fits-all standard.

5 Engineering Practice Suggestions

Based on the experimental conclusion, the following matters should be given more consideration:

Optimization priority: Basic issues such as warp divergence and bank conflicts should be resolved first, and then advanced optimization techniques should be considered;

Performance monitoring: It is recommended to use tools such as Nsight Compute to quantitatively analyze the actual benefits of each optimization stage to avoid over-optimization.

Code maintainability: In advanced optimizations such as reduce5/reduce7, code readability should be maintained through templates and macros.

This experiment confirms that through systematic optimization methods, it is entirely possible to achieve reduce performance close to the theoretical limit on the GPU. Future work can explore the application of new technologies such as asynchronous copying and Tensor Core, as well as the migration of optimization strategies on different hardware architectures.

6 Performance Testing under Different Parameters

This experiment aims to evaluate the performance of the improved `reduce` algorithm using the Shuffle instruction under different input scales and thread Block (Block Size) configurations. The Kernel execution time (ms) under

each configuration was measured through actual operation, and the impact of different configurations on performance was analyzed in combination with the CUDA programming model.

6.1 Implementation details

`reduce_benchmark.cu` The program implements the parallel reduction algorithm under CUDA and improves performance through the following optimization strategies:

- Cache intermediate results using shared memory to reduce global memory access.
- Merge two elements into each thread for processing to enhance instruction parallelism.
- The host side reduces the result of each Block again.
- Automatically generate the input and run it multiple times, and measure the average execution time.

6.2 Parameter sensitivity analysis

Performance tests were conducted on different combinations of `InputSize` and `BlockSize`, and some of the results are as follows in Figure4 and table 1:

Multiple key performance indicators can be extracted from the analysis graph of ncu (NVIDIA Nsight Compute) in Figure5. Firstly, the "Estimated Speedup" column in the figure shows the speedup ratio of each optimized configuration compared to the benchmark version, which can reach up to 97.22% at most. This indicates that the size of the input data and the number of blocks largely affect the execution efficiency of the algorithm. In terms of execution time, the Duration [ns] column represents the duration of each execution. The execution time gradually decreases after optimization, indicating that the optimization is effective. Although the Compute Throughput is generally low, with most configurations being only 0.22% to 0.24%, some configurations (such as those with an speedup ratio of 30.02%) exhibit a higher compute throughput, indicating that computing resources are utilized more effectively in these cases. Similarly, there is a similar trend in Memory

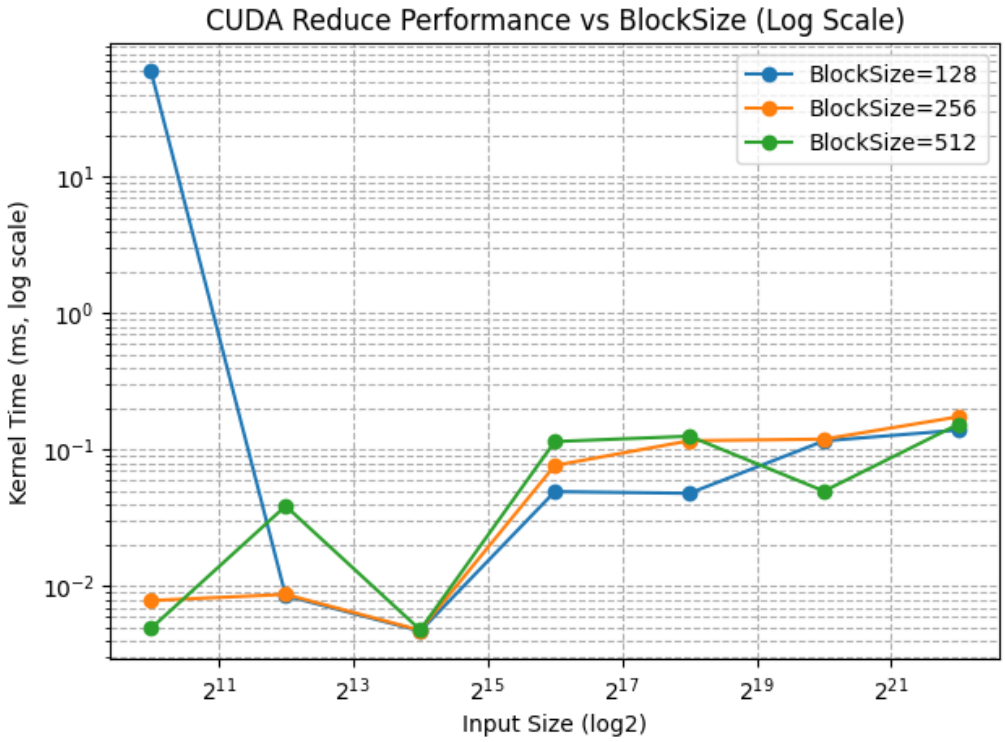
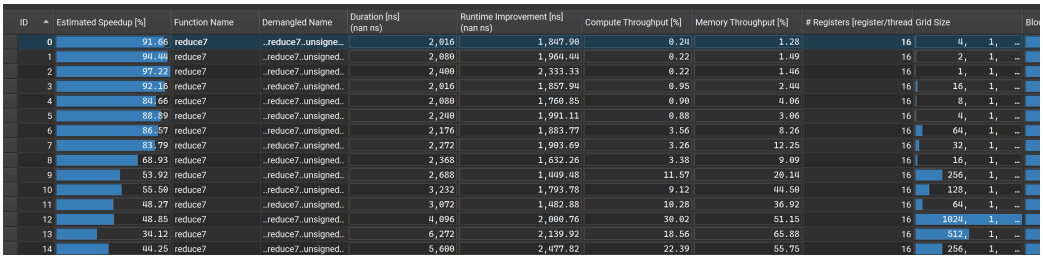


Figure 4: test result

Throughput. Some configurations have shown significant improvements after optimization, especially memory access has been enhanced. In terms of thread and resource allocation, the "Registers [register/thread]" column in the figure remains unchanged at 16, indicating that the configuration of the registers has not undergone significant changes. The variation of the Grid Size indicates that under certain optimized configurations, the program can perform parallel processing better. For example, when configuring 1024,1,1, the increase in grid size enables the program to handle more threads simultaneously, thereby achieving a better performance improvement. Overall, although the performance of some configurations has been significantly improved, there are still situations where the computing throughput rate and memory throughput rate are not fully utilized. Subsequently, the overall performance can be further enhanced by further optimizing the memory access mode, computing resource allocation, and computing throughput rate.

Table 1: test result

InputSize	BlockSize	KernelTime (ms)
1024	128	59.4361
1024	256	0.007872
1024	512	0.004928
4096	256	0.008736
16384	512	0.009984
65536	512	0.112352
262144	512	0.054880
1048576	512	0.164832
4194304	512	0.405984



ID	Estimated Speedup [%]	Function Name	Demangled Name	Duration [ns] (nan ns)	Runtime Improvement [ns] (nan ns)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread	Grid Size	Blo
0	91.66	reduce7	..reduce7..unsigne..	2,016	1,847.90	0.24	1.28	16	4, 1, ..	
1	94.44	reduce7	..reduce7..unsigned..	2,080	1,964.44	0.22	1.49	16	2, 1, ..	
2	97.22	reduce7	..reduce7..unsigned..	2,400	2,333.33	0.22	1.46	16	1, 1, ..	
3	92.16	reduce7	..reduce7..unsigned..	2,016	1,857.94	0.95	2.44	16	16, 1, ..	
4	84.66	reduce7	..reduce7..unsigned..	2,080	1,760.85	0.90	4.06	16	8, 1, ..	
5	88.89	reduce7	..reduce7..unsigned..	2,240	1,991.11	0.88	3.06	16	4, 1, ..	
6	86.57	reduce7	..reduce7..unsigned..	2,176	1,883.77	3.56	8.26	16	64, 1, ..	
7	83.79	reduce7	..reduce7..unsigned..	2,272	1,903.69	3.26	12.25	16	32, 1, ..	
8	68.93	reduce7	..reduce7..unsigned..	2,368	1,632.26	3.38	9.09	16	16, 1, ..	
9	53.92	reduce7	..reduce7..unsigned..	2,688	1,449.48	11.57	20.14	16	256, 1, ..	
10	55.50	reduce7	..reduce7..unsigned..	3,232	1,793.78	9.12	44.50	16	128, 1, ..	
11	48.27	reduce7	..reduce7..unsigned..	3,072	1,482.88	10.28	36.92	16	64, 1, ..	
12	48.85	reduce7	..reduce7..unsigned..	4,096	2,000.76	30.02	51.15	16	1024, 1, ..	
13	30.12	reduce7	..reduce7..unsigned..	6,272	2,139.92	18.56	65.88	16	512, 1, ..	
14	44.25	reduce7	..reduce7..unsigned..	5,600	2,477.82	22.39	55.75	16	256, 1, ..	

Figure 5: ncu_benchmark

The figure 6 is the performance analysis result of the reduce_benchmark execution process through NVIDIA Nsight Systems (NSYS). The information in the figure reveals the timing relationship of each stage and operation of the program during its execution on the GPU. Firstly, the obvious red bars in the figure represent the CUDA memory allocation and data transfer operations. Among them, the cudaMalloc operation occurs at the beginning of the figure, indicating that the program allocates memory before execution, while the cudaMemcpy operation represents the process of memory data transfer, which usually takes place between the host and the device. Excessive memory transfer operations may become a performance bottleneck, especially in the case of large datasets, as these operations can significantly increase the execution time. Secondly, the blue bars in the figure show the startup and execution process of the CUDA kernel. The duration of cudaLaunch is relatively short, indicating that the kernel's execution time is relatively efficient.

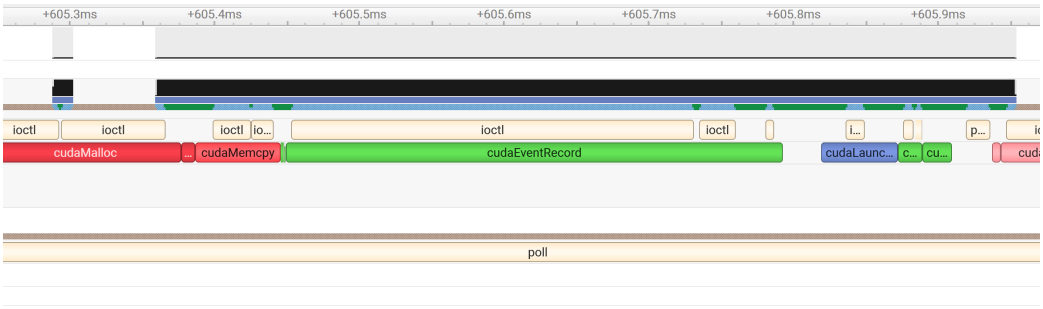


Figure 6: nsys_benchmark

The startup of the kernel may cause certain delays, especially when multiple kernels are started. The competition among kernels may increase additional waiting time. The green bars in the figure represent cudaEventRecord, which is used to record the timestamps of CUDA events and assist in performance analysis. The event recording operation gap in the figure is relatively small, which may imply that the relevant time was recorded immediately after each kernel's execution ended.

6.3 Performance Analysis and Optimization Effect

Warp Occupancy and Thread Efficiency Small block sizes (such as 128) lead to low resource utilization; And at 512, a higher warp activity level can be achieved.

Shared Memory and Access Patterns The larger the BlockSize, the more obvious the optimization of shared memory. Reduce bank conflicts and improve efficiency.

Analysis of Abnormal Manifestations When InputSize = 1024 and BlockSize = 128, the performance was abnormal (59ms), indicating that there was a delay issue when the function started or was allocated at the beginning of its operation.

Comparison with the peak bandwidth of RTX 4070

- RTX 4070 Theoretical bandwidth: 504 GB/s.

- The measured maximum input (4M) shows that the transmission capacity is approximately 32MB and the time consumption is 0.4ms.
- The actual bandwidth is estimated to be approximately 80GB/s, accounting for about 15.8% of the theoretical bandwidth.

6.4 Summary and Suggestions

This experiment shows that by using the modified CUDA reduce algorithm and through shared memory optimization and reasonable thread block configuration, the execution efficiency of the GPU in the reduction task can be significantly improved. Specifically, appropriately increasing the BlockSize can effectively improve the resource utilization and overall throughput capacity of SM (Streaming Multiprocessor), enabling more threads to execute actively simultaneously and reducing the idle time of hardware resources. Meanwhile, shared memory is introduced to cache the intermediate calculation results, effectively avoiding frequent global memory access and significantly reducing memory access latency. During the data loading stage, by having each thread handle multiple elements and perform initial merging, the computing density of each thread can be increased and the execution of redundant instructions can be reduced. Furthermore, on the basis of parallel reduction of multiple thread blocks, the final results are merged through the host end. Compared with implementing global synchronous reduction at the device end, it not only simplifies the synchronization mechanism but also reduces the resource competition and synchronization delay in the write-back process.

To further optimize the performance of this algorithm, it is recommended to introduce Warp-level primiples, such as instructions like __shfl_down, in the subsequent work, to achieve more efficient inter-thread communication and reduction within warp and reduce the reliance on shared memory and synchronization instructions. Meanwhile, the upper limit of the number of threads can be limited by combining launch bounds, and the occupancy calculator provided by NVIDIA can be used to accurately evaluate the resource occupancy rate under various thread configurations, thereby selecting the optimal scheduling strategy. Furthermore, it is recommended to conduct an in-depth analysis of the operation process of the kernel through the Nsight Compute tool, actually quantify the source of warp stall, the utilization efficiency of global memory bandwidth, and various delay bottlenecks, providing

a quantitative basis and optimization direction for subsequent performance improvement.