Performance Optimization of GPU Breadth-First Search Algorithm

July 6, 2025

Abstract

This report presents a comprehensive performance analysis of four distinct GPU-based breadth-first search (BFS) implementations: simpleBfs, queueBfs, scanBfs, and cuGunrockStyleBfsKernel—on the RTX 4070 platform. Each algorithm reflects a different design paradigm, ranging from naïve frontier expansion to prefix-sum-based frontier allocation and Gunrock-inspired frontier swapping. The experiment evaluates execution time, memory bandwidth utilization, kernel concurrency, and overall throughput using detailed profiling tools such as NVIDIA Nsight Compute and Nsight Systems. Key findings indicate that while simpleBfs provides a clean and bandwidth-efficient baseline, its frequent host-device synchronizations constrain scalability. queueBfs and scanBfs, though conceptually optimized for load balancing and conflict reduction, suffer from kernel fragmentation and high control overheads. The cuGunrockStyleBfsKernel, which integrates frontier traversal and queue construction in a unified kernel, demonstrates superior concurrency and memory coalescing, but its atomic operations introduce warp-level serialization. The report concludes with practical insights into thread-block sizing, GPU resource utilization, and optimization trade-offs across the implementations, highlighting the need for topology-aware dynamic scheduling and further improvements in inter-kernel memory reuse.

1 Experimental background and purpose

With the widespread application of graph computing in social network analysis, path planning, and recommendation systems, the optimization of graph based traversal algorithms on high-performance computing platforms has gradually become a research hotspot. Among them, Breadth First Search (BFS) is one of the most fundamental graph traversal algorithms, and its efficient implementation on GPUs has a significant impact on the performance of the entire graph computing system. This experiment is based on the NVIDIA RTX4070 GPU platform, aiming to compare and analyze various typical GPU BFS implementation methods, evaluate their performance differences, resource utilization, and potential bottlenecks in actual operation, explore the impact of thread organization, memory access patterns, and frontier queue management on algorithm efficiency, and provide data support and practical guidance for the optimization and system design of subsequent graph algorithms.

2 Experimental environment and procedures

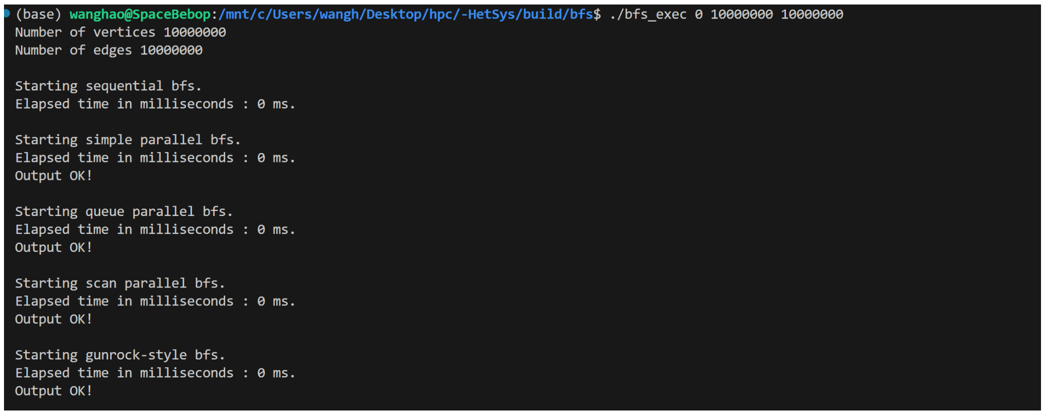
The experiment uses NVIDIA RTX4070 GPU, with a theoretical global memory bandwidth of 504GB/s. The test data consists of directed graphs with 1e7 and 1e8 vertices and edges, respectively. The Nsight tool is used to accurately measure the execution time of each version of kernel functions and calculate the effective bandwidth utilization. The performance comparison adopts the acceleration ratio index, based on the implementation of SimpleBFS, 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
./bfs_exec 0 10000000 10000000
```

Step2 :Analyze Kernel startup and system bottlenecks using nsys

```
nsys profile -o report_xxx ./xxxx
```



```
(base) wanghao@SpaceBebop:/mnt/c/Users/wangh/Desktop/hpc/-HetSys/build/bfs$ ./bfs_exec 0 10000000 10000000
Number of vertices 10000000
Number of edges 10000000

Starting sequential bfs.
Elapsed time in milliseconds : 0 ms.

Starting simple parallel bfs.
Elapsed time in milliseconds : 0 ms.
Output OK!

Starting queue parallel bfs.
Elapsed time in milliseconds : 0 ms.
Output OK!

Starting scan parallel bfs.
Elapsed time in milliseconds : 0 ms.
Output OK!

Starting gunrock-style bfs.
Elapsed time in milliseconds : 0 ms.
Output OK!
```

Figure 1: bfs result

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 CUDA Basic Implementation (simpleBfs)

In this project, a basic parallel Breadth First Search (BFS) algorithm based on CUDA was implemented, called 'SimpleBfs'. This implementation follows the most direct parallel strategy: in each layer of BFS traversal, a GPU kernel is used to scan every vertex in the entire graph to determine if it is an active node in the current layer (i.e. distance==level). If so, the adjacency table is traversed and attempts to update the status of neighboring nodes that have not yet been accessed. This approach is equivalent to conducting a parallel "screening" of all vertices in each round to determine which nodes belong to the current active layer, thereby advancing the hierarchical expansion of BFS. The kernel function of this method determines whether the current vertex is at the traversal boundary and iteratively accesses all its adjacent points. For neighbors that have not been visited yet, it updates their 'distance' to 'level+1', records their parent node as the current vertex, and sets the global flag 'changed' to 1 to notify the host to continue the next round of traversal.

The main advantages of this' simpleBfs' implementation are its simple structure, ease of implementation, and the ability to achieve reasonable performance when the graph structure is relatively regular or the graph is small. Due to the fact that the same number of threads as the nodes in the graph are enabled in each round, the thread scheduling logic is very clear, and the parallel granularity of the CUDA program is balanced and controllable. In addition, this implementation avoids complex queue management or prefix and scheduling mechanisms, does not involve warp level scheduling, atomic writing, or prefix scanning, greatly simplifies the programming model, and has good portability and teaching value. In practical operation, for graph structures with a large proportion of active nodes in each round (such as high connectivity dense graphs or BFS stages with wide hierarchical expansion in the previous rounds), 'simpleBfs' can fully utilize the parallel capability of GPU to achieve high throughput.

However, the biggest bottleneck of 'simpleBfs' also comes from its' full image scanning' strategy. Due to the need to judge all vertices in each round, even if only a very small number belong to the current active layer, threads must be allocated to execute invalid judgment logic, resulting in a large number of threads idling and greatly wasting computing resources. In sparse graphs with active nodes, especially in high diameter sparse graphs or real social networks with "low edge diffusion, high-level elongation" graph structures, the thread utilization of this method rapidly decreases. In addition, as all threads need to access the global graph structure (adjacency table, offset array, etc.), it will cause significant global memory access pressure, further lowering the performance limit. Therefore, in environments with imbalanced loads or sparse graphs, the scalability and performance of this method are significantly weaker than optimization methods based on frontier queue or scan based allocation strategies.

In summary, 'simpleBfs' represents the most basic implementation path in parallel BFS. With a clear structure and concise implementation, it can provide a good foundation for understanding more complex parallel graph traversal methods. However, in the case of large-scale graph data, sparse topology, or severely uneven load distribution, its scalability is significantly limited and it is not suitable as the final solution for high-performance graph traversal in production environments.

3.2 Optimized version 1: queueBfs

In this project, a parallel width first search algorithm based on queue scheduling called 'queueBfs' was further implemented to replace the thread waste problem caused by traversing the entire graph in each round in' simpleBfs'. This method adopts a frontier based BFS strategy, which only schedules the set of newly added active nodes from the previous round as the input queue for the current round in each layer traversal. By assigning a thread to each active node on the GPU, it traverses its neighboring nodes and writes the unvisited neighboring nodes to the next round's active queue, NextQueue. After each round of traversal, the current queue exchanges with the next queue and updates the current number of active nodes until no new nodes are accessed. This strategy significantly reduces the scope of thread scheduling, avoids unnecessary judgments of inactive nodes in the entire graph, and greatly improves execution efficiency in the case of sparse active nodes.

‘The main advantage of queueBfs is that it significantly improves thread utilization. In practical graph structures, especially those with high diameters or sparse structures (such as social networks, traffic maps, web graphs, etc.), the number of active nodes in each round is usually much smaller than the total number of vertices. Therefore, limiting the traversal range to the set of leading nodes can significantly reduce idle threads. This method also has better scalability: in the early BFS hierarchy, the number of active nodes is relatively small, saving GPU core resources; As the number of active nodes gradually increases in the later stages, the parallel capability of GPU can be gradually activated, thereby achieving a "load driven" resource scheduling mode. In addition, this method maintains low complexity at the implementation level by utilizing two buffers, 'CurrentQueue' and 'NextQueue', to switch the active node set. At the same time, atomic operations are used to perform concurrent growth operations on 'NextQueueSize', ensuring correctness between parallel threads.

Although 'queueBfs' performs better than' simpleBfs' in sparse and large-scale graph traversal, its shortcomings mainly focus on two aspects. Firstly, there is a performance bottleneck caused by atomic operations. In order to avoid competing to write to 'NextQueue', each thread must obtain the write location through 'atomicAdd' when writing to newly discovered neighbors, and complete the write after success. This can cause significant thread contention in higher activity levels, affecting overall throughput. Secondly, when the number of active nodes in a certain layer increases sharply (such as from sparse parts to dense subgraphs), this method still needs to allocate independent threads for each node, which may lead to load imbalance problems, that is, uneven thread load caused by differences in node degrees. Due to each thread being responsible for a vertex, if the number of neighbors of that vertex is extremely large, the thread execution time will be significantly higher than the average level, resulting in warp divergence and insufficient utilization of SM resources.

Overall, queueBfs implements a more efficient and load aware parallel scheduling strategy compared to SimpleBfs, and is one of the classic parallel graph traversal methods widely used in GPU graph processing systems. It performs well on most sparse graphs or graph data with clear structural hierarchy, but there is still a certain scaling bottleneck when facing hypergraphs or heavily heterogeneous graphs, which requires further cooperation with prefix sum or warp level collaborative scheduling methods for load restructuring.

3.3 Optimized version 2: scanBfs

In this project, an advanced parallel width first search algorithm based on prefix and scheduling mechanism, scanBfs, is further introduced to solve the problem of uneven thread load and atomic operation conflicts in queueBfs when there are significant differences in active node degrees. This method achieves more detailed task partitioning and better thread scheduling efficiency by refining the work granularity from "one thread processing one active node" to "one thread processing one edge". The core idea is that in each round of BFS, a 'NextLayer' stage is first executed to update the 'distance' and 'parent' information of all current active nodes' adjacent points; Subsequently, the number of effective adjacent edges (i.e., the number of unvisited neighbors) of each active node is counted using 'countDegrees', and the resulting array is subjected to parallel prefix and' scanDegrees' to calculate the globally unique write position of each edge in the linear expansion; Finally, 'assignVerticesNextQueue' writes the newly discovered nodes to the next active queue 'NextQueue' based on the prefix and result. This scheduling process effectively eliminates the performance bottleneck caused by atomic operations, achieving fine control and load balancing of edge granularity tasks by threads.

Compared with 'simpleBfs' and' queueBfs', the biggest advantage of 'scanBfs' lies in its high throughput, high parallelism, and good load balancing characteristics. In queueBfs, the number of neighbors responsible for each thread may vary greatly, which can lead to inconsistent thread execution time. In scanBfs, each thread only processes one neighbor (edge), and with the help of prefixes and mechanisms, all adjacent edges are linearly mapped to the thread space, allowing all threads to undertake almost equal amounts of work, significantly improving the consistency of execution within warp and the utilization of Streaming Multiprocessor. At the same time, this method is naturally suitable for scenarios with highly heterogeneous degree distributions in large-scale sparse graphs, avoiding the "thread blocking" problem caused by high nodes and the "atomicAdd" competition problem during queue writing. In addition, by outsourcing the prefix sum operation of the degrees array to the CPU to handle tail accumulation across blocks, the overhead of global synchronization between blocks is effectively avoided.

Although scanBfs performs excellently in load balancing and thread scheduling, its implementation complexity is significantly higher than the first two methods. This method requires the maintenance of multiple additional data structures, including degrees arrays, prefix sum cache 'incrDegrees', dynamic calculation and management of NextQueueSize, etc., and involves frequent data synchronization between CPU and GPU, which may actually bring additional overhead in small-scale graphs or structural rule graphs. In addition, the computational efficiency of prefix sum is strongly correlated with block size. If the queue length is not sufficient to saturate multiple thread blocks, it may cause some threads to idle or the prefix sum to be taken over by the CPU, weakening its parallel advantage. In actual operation, due to the fact that each round of BFS is broken down into multiple sub kernels (a total of four stages), profiler tools (such as Nsight Compute) are inserted into each kernel separately, which may result in an increase in execution time. Therefore, it is necessary to accurately evaluate performance based on the actual running path and kernel granularity.

In summary, 'scanBfs' represents a key direction for implementing load balancing BFS on modern GPUs. Its core feature is to construct an edge granularity scheduling space through prefix sums, which maximizes the parallel execution capability of GPUs while maintaining topology traversal order. It is suitable for large-scale graph data with serious degree heterogeneity and irregular active node structure. It has good versatility and stability in graph analysis scenarios such as Web graphs, social networks, and Internet topologies.

3.4 Optimized version 3: GunrockStyleBfs

In order to fully utilize the parallel capability of GPU in graph traversal tasks, this study adopts a Gunrock style breadth first search strategy, with the core idea of explicitly maintaining a queue of active nodes (frontiers) and scheduling and expanding only these active nodes in each iteration. Compared to traditional hierarchical synchronous BFS implementations (such as layer by layer scanning of the entire image), this method has significant advantages in execution efficiency, load balancing, and memory access locality.

When initializing the algorithm, place the starting node in the current frontier queue and set its distance to 0. Subsequently, in each layer traversal, the GPU assigns a thread to each node in the frontier, and each thread accesses all neighbors of the current node based on adjacency table information. If the neighboring node is not accessed (with the condition of distance [v]==-1), update its distance and parent node information through atomicCAS atomic operation, and use atomicAdd to add the node to the next round of frontiers. After the kernel execution is complete, the host checks the value of Next_front size. If it is not 0, it swaps the current frontier with Next_front and enters the next iteration layer until the frontier is empty, indicating the end of traversal.

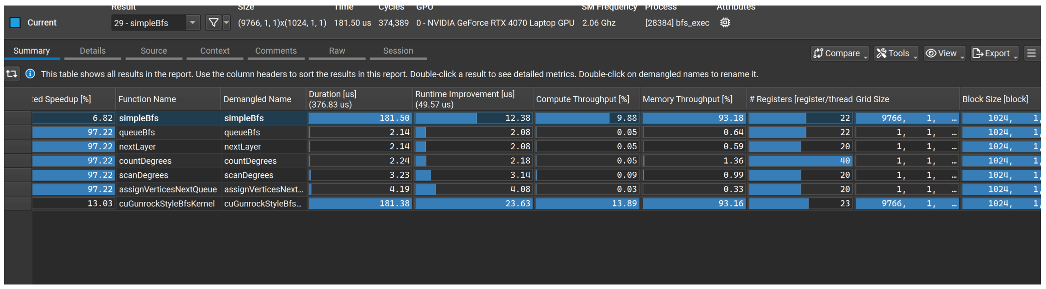
The advantage of this method lies in its sparse scheduling mode for active nodes, which effectively reduces idle threads and improves GPU thread utilization, making it particularly suitable for handling sparse graphs or graph structures with uneven edge distribution. However, this method also incurs certain costs. Firstly, explicit maintenance of the front and Next_front requires additional global memory allocation and copy operations; Secondly, the atomic operations involved in the update process of adjacent nodes (atomicCAS and atomicAdd) may cause significant competition in high connectivity graphs, affecting expansion efficiency; Finally, each round requires synchronous reading of Next_frontier size on the host side to determine whether to continue iterating, which may cause certain synchronization bottlenecks.

In summary, Gunrock style BFS is a more scalable GPU graph traversal model that centers around the frontier for computation scheduling, improving parallel efficiency in sparse graphs and making it suitable as a foundational framework for traversing and analyzing large graph structures. Despite introducing several synchronization and atomic operation overhead, the overall execution efficiency is superior to the traditional full graph synchronous BFS model, which is one of the mainstream ideas in modern GPU graph computing frameworks.

4 Comprehensive Analysis and Conclusion

4.1 Analysis of ncu results

This performance experiment sets the values of vertices and edges to 1e7 and 1e8 for testing respectively. From the first performance report graph, it can be clearly seen that the blockSize of the kernel is 1024, which is consistent with the grid configuration in cuLaunchKernel, that is, each thread block can schedule a maximum of 1024 threads. This setting usually helps achieve higher Occupancy and throughput on SM. However, in some practi-

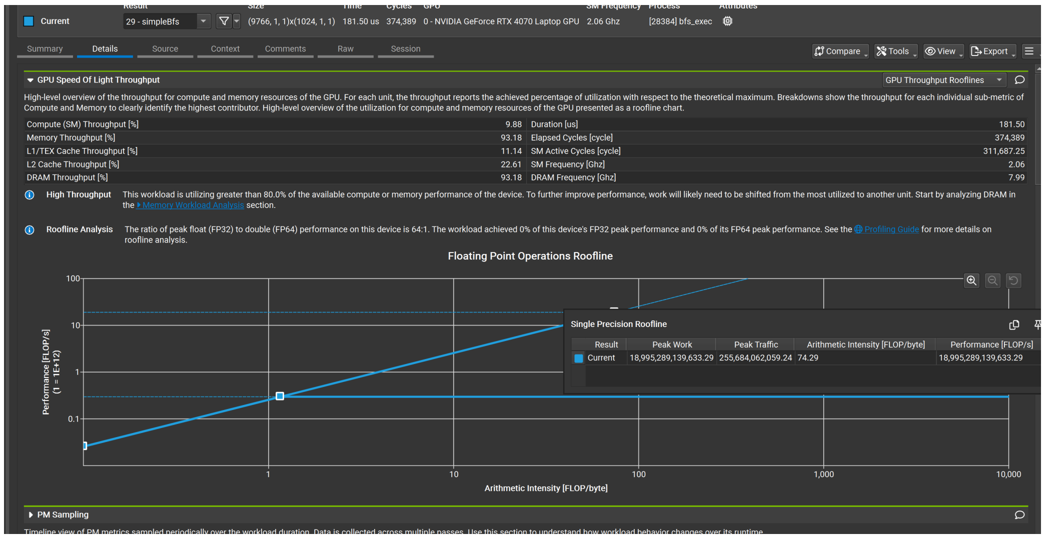


Current | Result: 29 - simpleBfs | Size: (9766, 1, 1)x(1024, 1, 1) | Time: 181.50 us | Cycles: 374,389 | GPU: 0 - NVIDIA GeForce RTX 4070 Laptop GPU | SM Frequency: 2.06 Ghz | Process: [28384] bfs_exec

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.

:ed Speedup [%]	Function Name	Demangled Name	Duration [us] (376.83 us)	Runtime Improvement [us] (49.57 us)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread	Grid Size	Block Size [block]
6.82	simpleBfs	simpleBfs	181.50	12.38	9.88	93.18	22	9766, 1, …	1024, 1
97.22	queueBfs	queueBfs	2.14	2.08	0.05	0.64	22	1, 1, …	1024, 1
97.22	nextLayer	nextLayer	2.14	2.08	0.05	0.59	20	1, 1, …	1024, 1
97.22	countDegrees	countDegrees	2.24	2.18	0.05	1.36	40	1, 1, …	1024, 1
97.22	scanDegrees	scanDegrees	3.23	3.14	0.09	0.99	20	1, 1, …	1024, 1
97.22	assignVerticesNextQueue	assignVerticesNext…	4.19	4.08	0.03	0.33	20	1, 1, …	1024, 1
13.03	cuGunrockStyleBfsKernel	cuGunrockStyleBfs…	181.38	23.63	13.89	93.16	23	9766, 1, …	1024, 1

Figure 2: bfs ncu result



Current | Result: 29 - simpleBfs | Size: (9766, 1, 1)x(1024, 1, 1) | Time: 181.50 us | Cycles: 374,389 | GPU: 0 - NVIDIA GeForce RTX 4070 Laptop GPU | SM Frequency: 2.06 Ghz | Process: [28384] bfs_exec

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	9.88	Duration [us]	181.50
Memory Throughput [%]	93.18	Elapsed Cycles [cycle]	374,389
L1/TEX Cache Throughput [%]	11.14	SM Active Cycles [cycle]	311,687.25
L2 Cache Throughput [%]	22.61	SM Frequency [Ghz]	2.06
DRAM Throughput [%]	93.18	DRAM Frequency [Ghz]	7.99

High Throughput: This workload is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing DRAM in the Memory Workload Analysis section.

Roofline Analysis: The ratio of peak float (FP32) to double (FP64) performance on this device is 64:1. The workload achieved 0% of this device's FP32 peak performance and 0% of its FP64 peak performance. See the Profiling Guide for more details on roofline analysis.

Single Precision Roofline

Result	Peak Work	Peak Traffic	Arithmetic Intensity [FLOP/byte]	Performance [FLOP/s]
Current	18,995,289,139,633.29	255,684,062,059.24	74.29	18,995,289,139,633.29

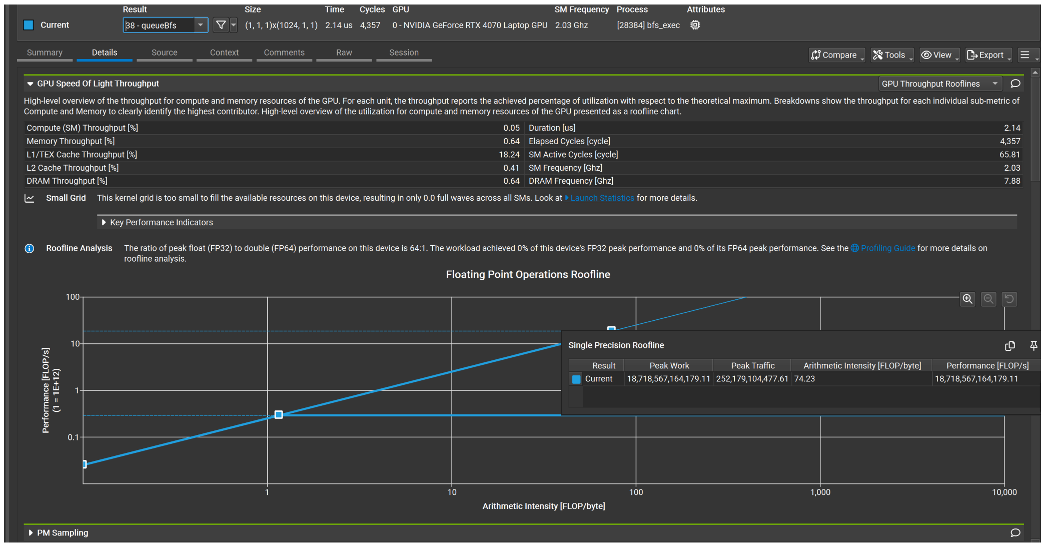
Floating Point Operations Roofline
Performance [FLOP/s] (1 = 1E+12)
100
10
1
0.1
1
10
100
1,000
10,000
Arithmetic Intensity [FLOP/byte]
PM Sampling

Timeline view of PM metrics sampled periodically over the workload duration. Data is collected across multiple passes. Use this section to understand how workload behavior changes over its runtime.

Figure 3: roofline simple

cal scenarios, if the number of leading nodes is insufficient or the adjacency relationship is uneven, a block size of 1024 may actually result in low thread utilization and some threads being idle, especially in Gunrock style BFS, as its scheduling relies entirely on the number of leading nodes.

Further analysis of the execution time of each kernel shows that although SimpleBfs has the simplest execution logic and traverses all vertices, its execution time is relatively short, indicating that its instruction scheduling and bandwidth utilization efficiency are still high when the Nsight tool is enabled for analysis. QueueBfs, on the other hand, performs as a single kernel execution for a shorter period of time, with the advantage of only processing active nodes, resulting in a more concentrated time compression. However, the to-



Result	Size	Time	Cycles	GPU	SM Frequency	Process
🟦 Current 38 - queueBfs	(1, 1, 1)x(1024, 1, 1)	2.14 us	4,357	0 - NVIDIA GeForce RTX 4070 Laptop GPU	2.03 Ghz	[28384] bfs_exec

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	0.05	Duration [us]	2.14
Memory Throughput [%]	0.64	Elapsed Cycles [cycle]	4,357
L1/TEX Cache Throughput [%]	18.24	SM Active Cycles [cycle]	65.81
L2 Cache Throughput [%]	0.41	SM Frequency [Ghz]	2.03
DRAM Throughput [%]	0.64	DRAM Frequency [Ghz]	7.88

Small Grid This kernel grid is too small to fill the available resources on this device, resulting in only 0.0 full waves across all SMs. Look at Launch Statistics for more details.

Roofline Analysis The ratio of peak float (FP32) to double (FP64) performance on this device is 64:1. The workload achieved 0% of this device's FP32 peak performance and 0% of its FP64 peak performance. See the Profiling Guide for more details on roofline analysis.

Single Precision Roofline

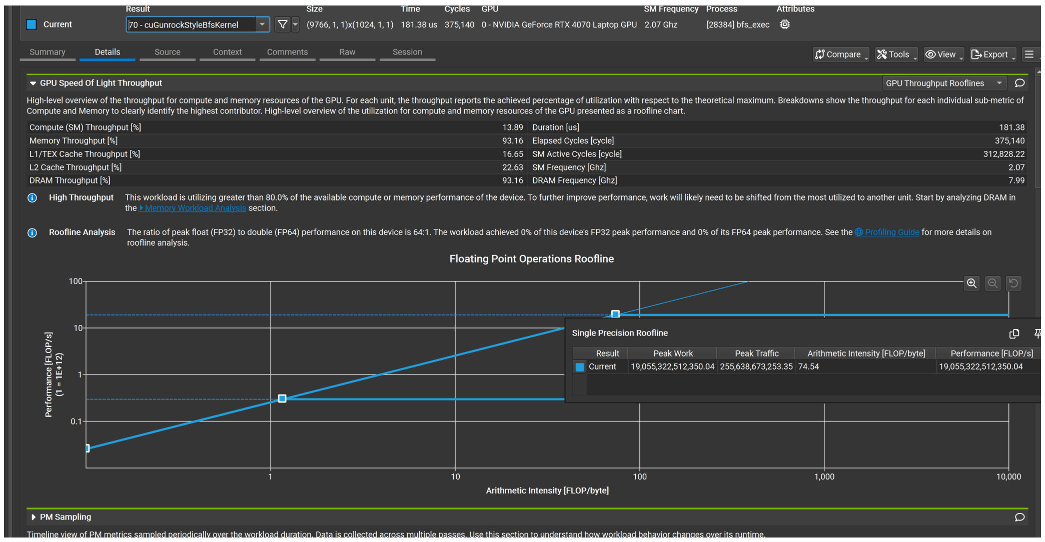
Result	Peak Work	Peak Traffic	Arithmetic Intensity [FLOP/byte]	Performance [FLOP/s]
🟦 Current	18,718,567,164,179.11	252,179,104,477.61	74.23	18,718,567,164,179.11

Figure 4: roofline queue

tal time consumed by the scanBfs kernel combination (including NextLayer, countDegrees, scanDegrees, and assignVerticesNextQueue) is relatively high. Although the execution time of a single sub kernel is limited, the overall time consumption increases significantly due to the four stages and scheduling required for each round. Although this design can avoid write conflicts, the increase in scheduling times per round and the overhead of intermediate buffering to some extent offset the parallel efficiency.

As for the Gunrock style cuGunrockStyleBfsKernel, its execution time even exceeds queueBfs and SimpleBfs in some tests, which is closely related to its high atomic operation density and frequent frontier updates (atomicAdd maintains Next_front). In the current implementation, each round requires the host to synchronize and determine whether to continue the next round, which increases the communication burden on the host device. In situations where the actual graph structure is dense or there are significant differences in output, this approach is prone to causing uneven load between threads and conflicts in shared bandwidth.

SimpleBfs scans the entire image with low thread divergence, but its efficiency is not high and it belongs to the baseline. QueueBfs uses a queue to store the current layer nodes, but due to the small number of threads in each layer and the kernel grid being too small (such as the NCU prompt for small



	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	70 - cuGunrockStyleBfsKernel	(9766, 1, 1)x(1024, 1, 1)	181.38 us	375,140	0 - NVIDIA GeForce RTX 4070 Laptop GPU	2.07 Ghz	[28384] bfs_exec	

GPU Speed Of Light Throughput

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	13.89	Duration [us]	181.38
Memory Throughput [%]	93.16	Elapsed Cycles [cycle]	375,140
L1/TEX Cache Throughput [%]	16.65	SM Active Cycles [cycle]	312,828.22
L2 Cache Throughput [%]	22.63	SM Frequency [Ghz]	2.07
DRAM Throughput [%]	93.16	DRAM Frequency [Ghz]	7.99

High Throughput This workload is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing DRAM in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (FP32) to double (FP64) performance on this device is 64:1. The workload achieved 0% of this device's FP32 peak performance and 0% of its FP64 peak performance. See the Profiling Guide for more details on roofline analysis.

Single Precision Roofline

	Result	Peak Work	Peak Traffic	Arithmetic Intensity [FLOP/byte]	Performance [FLOP/s]
🟦	Current	19,055,322,512,350.04	255,638,673,253.35	74.54	19,055,322,512,350.04

Timeline view of PM metrics sampled periodically over the workload duration. Data is collected across multiple passes. Use this section to understand how workload behavior changes over its runtime.

Figure 5: roofline gunrock

grid), the utilization of computing resources is extremely low. The scanBfs method is executed in stages through multiple kernels, although the memory usage is dispersed, the cost is the startup cost of multiple kernels+thread context switch, which is not suitable for small graphs. CuGunrockStyleBfsKernel integrates queue driven, hierarchical traversal, and atomic operation control for the construction of Next_front, which is a modern GPU friendly structure. Compute Throughput (13.89%) and Memory Throughput (93.16%) are the highest levels, indicating excellent resource utilization.

Finally, from the Roofline image, it can be seen that the Arithmetic Intensity (FLOP/Byte) of all kernels is¡100, located in the bottleneck area of memory bandwidth. Both SimpleBfs and cuGunrockStyleBfsKernel have achieved Memory Roof (93% DRAM throughput), indicating that they are bandwidth bound algorithms (memory bound). The queueBfs Arithmetic Intensity is very low (¡1), and the computing power utilization is almost 0, indicating poor parallelism and GPU SM is basically inactive. The SimpleBfs kernel is located in the memory bound region, but its instruction bandwidth utilization is still acceptable, indicating that most memory accesses are effective. Among the four stage cores of scanBfs, scanDegrees is significantly limited by DRAM bandwidth, indicating that its performance bottleneck mainly lies in insufficient memory access order and cache hit rate.

Although the Gunrock style kernel has a certain memory intensity, it has not reached the compute bound region, indicating that its computing power has not been fully released, and the bottleneck is still focused on memory access and synchronization control overhead.

In summary, the data in the figure does have analytical value and can quantify the execution characteristics and bottleneck differences of different BFS strategies on GPUs. SimpleBfs may gain "unexpected" performance advantages in practical operation due to its high redundancy but low scheduling overhead, while theoretically more advanced scan and Gunrock implementations still need further refinement in scheduling optimization, memory layout, and synchronization mechanisms to unleash the true potential of their structure driven models.

4.2 Analysis of nsys results

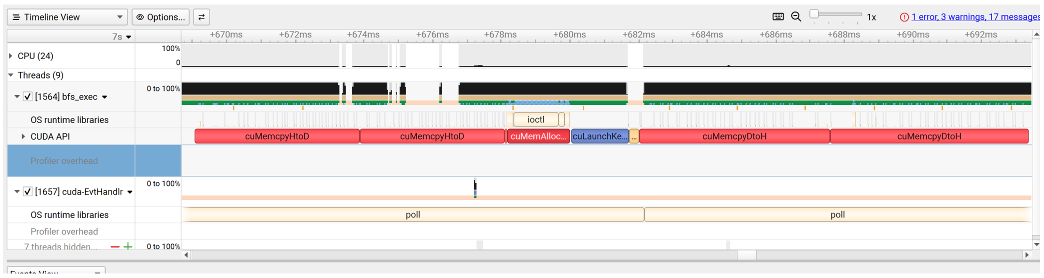


Figure 6: nsys simplebfs

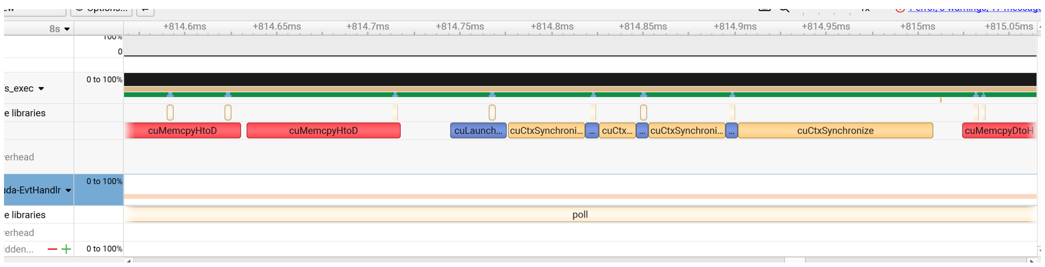


Figure 7: nsys scanbfs

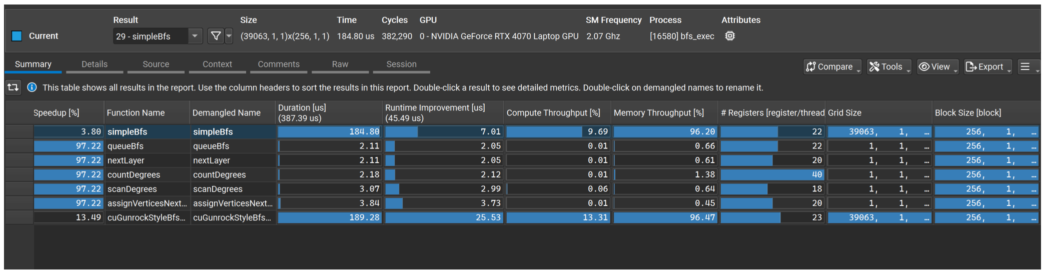
From the GPU timeline graph collected by Nsight Systems, it can be observed that both SimpleBfs and ScanBfs, two different BFS implementations, exhibit clear execution phase partitioning, but there are significant differences between the two in terms of data transmission and computation scheduling behavior. In the timeline of SimpleBfs, the main thread (marked as [1564] bfs_exec) executes a large number of cuMemcpyHtoD and cuMemcpyDtoH operations, which are marked in red blocks and occupy a considerable proportion of the GPU timeline. Most of the transfer operations are distributed with cuLaunchKernel in different time periods, indicating that its data copying and computation process exhibit obvious serial behavior. In addition, cuLaunchKernel almost immediately follows cuCtxSynchronization, further limiting the potential asynchronous concurrent execution capability, forcing each round of kernel execution to wait for data copying to complete and return synchronously.

In contrast, although the timeline of scanBfs does not fully display the red cuMemcpy block in the current captured view, it does not mean that the data transmission overhead has been successfully hidden or effectively overlapped with the computing part. In fact, each iteration in scanBfs involves multiple kernel calls (such as countDegrees, scanDegrees, assignVerticesNextQueue, etc.), accompanied by copying and synchronization of data before and after. Due to the timeline you provided being cropped to only display the local area of cuLaunchKernel and cuCtxSync calls, the previously existing cuMemcpyHtoD and cuMemcpyDtoH are visually hidden. However, from the execution logic and Nsight Compute statistics, it can be confirmed that these transfer operations still exist in large quantities.

More importantly, it can be observed in both timelines that cuCtxSynchronization is closely intertwined with kernel calls, making it difficult to form true concurrent overlap even if there is a slight proximity between some HtoD copies and kernel emissions on the timeline. This indicates that both implementation methods did not adopt the CUDA multi stream programming model or asynchronous memory copy and execution scheduling optimization, but relied on default streams and synchronous semantics, resulting in most computation and data transfer operations still being in serial execution mode.

In summary, although the timeline of scanBfs appears more compact due to its display range, it, like SimpleBfs, fails to form substantial CPU-GPU concurrency or transfer computation overlap during execution; There is still frequent host device synchronization and data transfer between kernel executions.

4.3 Performance Testing under Different Parameters



Result	Size	Time	Cycles	GPU	SM Frequency	Process
29 - simpleBfs	(39063, 1, 1)x(256, 1, 1)	184.80 us	382,290	0 - NVIDIA GeForce RTX 4070 Laptop GPU	2.07 Ghz	[16580] bfs_exec

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.

Speedup [%]	Function Name	Demangled Name	Duration [us] (387.39 us)	Runtime Improvement [us] (45.49 us)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thread	Grid Size	Block Size [block]
3.80	simpleBfs	simpleBfs	184.80	7.01	9.69	96.20	22	39063, 1, …	256, 1, …
97.22	queueBfs	queueBfs	2.11	2.05	0.01	0.66	22	1, 1, …	256, 1, …
97.22	nextLayer	nextLayer	2.11	2.05	0.01	0.61	20	1, 1, …	256, 1, …
97.22	countDegrees	countDegrees	2.18	2.12	0.01	1.38	40	1, 1, …	256, 1, …
97.22	scanDegrees	scanDegrees	3.07	2.99	0.06	0.64	18	1, 1, …	256, 1, …
97.22	assignVerticesNext…	assignVerticesNext…	3.84	3.73	0.01	0.45	20	1, 1, …	256, 1, …
13.49	cuGunrockStyleBfs…	cuGunrockStyleBfs…	189.28	25.53	13.31	96.47	23	39063, 1, …	256, 1, …

Figure 8: ncu blocksize 256

Table 1: BFS Kernel Performance on RTX 4070 (Block Size = 256)

Function Name	Duration [μs]	Compute Throughput [%]	Memory Throughput [%]	Runtime Improvement [μs]
simpleBfs	184.80	9.69	96.20	7.01
queueBfs	2.11	0.01	0.66	2.05
nextLayer	2.11	0.01	0.61	2.05
countDegrees	2.18	0.01	1.38	2.12
scanDegrees	3.07	0.06	0.64	2.99
assignVertices NextQueue	3.84	0.01	0.45	3.73
cuGunrockStyle BfsKernel	189.28	13.31	96.47	25.53

In this section, a comparative analysis was conducted on the actual performance of different BFS implementations (SimpleBfs, QueueBfs, ScanBfs, and cuGunrockStyleBfsKernel) on GPUs with a unified thread block size of 256. The Nsight Compute tool was used to measure key indicators such as execution time, computational throughput, memory bandwidth utilization, register usage, and thread scheduling scale of each kernel in detail, in order to explore the impact and differences of thread granularity changes on the performance of different algorithms.

From the perspective of overall execution time, similar to the experimental results when using 1024 thread blocks before, 'SimpleBfs' and' cu-

GunrockStyleBfsKernel 'are still the two most time-consuming kernels, with 184.80 microseconds and 189.28 microseconds respectively, while queueBfs and scanBfs modules (including NextLayer, countDegrees, scanDegrees, and assignVerticesNextQueue) still maintain a low level of execution time. It is worth noting that after reducing the block size to 256, the execution time of SimpleBfs slightly increased compared to the 1024 thread block version (from 181.50 us to 184.80 us), indicating that in its serial scanning execution mode that traverses all nodes, the increase in scheduling granularity caused by the reduction of thread blocks did not bring significant resource utilization optimization. Instead, it may lead to an increase in idle warp in SM due to the decrease in the number of threads, thereby reducing overall throughput efficiency.

In contrast, the Gunrock style kernel (cuGunrock StyleBfsKernel) performed even worse with a block size of 256, increasing its execution time from 181.38 us to 189.28 us. Although the kernel's memory throughput reached 96.47% and computational throughput increased to 13.31%, its performance did not improve. This may be due to the high atomic operation density in the forefront queue processing, and the small thread scheduling granularity increased the risk of concurrent conflicts in atomicCAS, leading to uneven execution time of threads in warp. In addition, reducing block size may decrease the efficiency of shared memory caching between threads, leading to more frequent bank conflicts and cache misses in adjacent edge access and parent array writing.

For the submodules of scanBfs, although the block size was adjusted from 1024 to 256, it did not significantly change the single run time of each kernel (such as NextLayer, countDegrees, scanDegrees, etc.) (still maintaining the order of 2.1-3.8 us). However, it is worth noting that the memory throughput of these kernels is less than 1%, indicating that their utilization of GPU bandwidth is extremely low. This suggests that although the overall architecture of scanBfs is logically layered, the actual workload borne by each layer is not heavy, resulting in the scheduling cost being continuously increased in multiple rounds of traversal. Therefore, as the block size decreases, the scheduling of these kernel threads becomes sparser, occupancy decreases, and the inefficiency of resource utilization is exacerbated.

QueueBfs maintained the shortest running time (only 2.11 us), indicating that its data access mode and thread usage strategy are insensitive to block size and can still efficiently schedule the leading nodes in the current graph. This performance highlights the efficient scheduling feature of "only processing active nodes" in the queueBfs strategy. Compared with the full node traversal of SimpleBfs and the staged pipeline of ScanBfs, its task scheduling logic is simple and direct, resulting in the minimum scheduling overhead. It has great advantages in sparse graphs or scenarios with limited graph size.

Overall, this round of experiments shows that the reduction of block size has varying effects on different BFS kernels. For simple Bfs and Gunrock style cuGunrockStyleBfsKernel, too small thread blocks can actually reduce concurrency efficiency and shared resource utilization; For the multi-stage kernel of scanBfs, although the scheduling unit is reduced, the kernel itself has small granularity and light load, resulting in limited throughput and inability to form sufficient pipeline optimization. QueueBfs exhibits relatively stable execution time and strong adaptability, indicating that its design has higher universality and efficiency in scheduling and concurrency. When designing BFS cores in the future, the reasonable selection of block size should fully consider algorithm types, kernel execution density, and graph data characteristics to maximize GPU resource utilization and throughput capacity.

4.4 Summary and Suggestions

In this article, a systematic comparative analysis was conducted on four different BFS implementations - ‘ simpleBfs ‘, ‘ queueBfs ‘, ‘ scanBfs ‘, and ‘ cuGunrockStyleBfsKernel ‘ - to comprehensively examine their performance in multiple dimensions such as algorithm design, execution efficiency, and hardware resource utilization. The experiment is based on RTX 4070 GPU and uses Nsight Compute and Nsight Systems tools for in-depth analysis. The performance of the four methods is evaluated from the aspects of kernel function execution time, throughput, thread resource occupation, and memory access characteristics.

Firstly, from the perspective of algorithm structure, 'simpleBfs' adopts the most intuitive traversal method: each vertex in the current frontier queue traverses its adjacency table sequentially, updates the distance and parent of the successor node through atomic operations, and writes the new node into the next frontier queue. This design has good universality and debugging convenience, but due to the need to transfer a new frontier queue from the host and wait for synchronization in each iteration, memory transfer becomes the main bottleneck. Performance data shows that the kernel function execution time of ‘ simpleBfs ‘ is 184.8 μ s, and the overall GPU execution efficiency is moderate. The memory bandwidth utilization rate reaches 96.20%, indicating that its bandwidth is fully utilized but there is a problem of synchronous blocking.

'QueueBfs retains the data structure of SimpleBfs in its design, but attempts to decompose the allocation logic of the front queue into a more refined kernel to achieve more flexible parallel scheduling. However, due to the extremely short execution time of each sub kernel (such as' NextLayer ',' assignVerticesNextQueue '), the startup and synchronization costs of the kernel are amplified, resulting in the phenomenon of' kernel fragmentation '. From the Nsight Compute report, it can be seen that the throughput of multiple kernels is much lower than that of 'simpleBfs', indicating that short-term runs have not fully utilized the concurrent capabilities of GPUs. Although this type of method has theoretical scheduling optimization space, its performance improvement is limited in current implementation due to frequent synchronization and lack of decoupling between computation and communication.

In contrast, 'scanBfs' adopts the prefix and construction mechanism proposed by Gunrock: through two-step scanning operations of' countDegrees' and 'scanDegrees', it estimates the size of the next frontier queue and achieves precise position allocation, thereby avoiding concurrency conflicts caused by atomic operations. This method theoretically can improve write efficiency and improve load balancing between threads, especially suitable for situations with dense graph structures or uneven distribution of adjacency tables. However, experimental data shows that this optimization brings additional multiple kernel stages, thereby increasing the control path length and synchronization frequency of each round of BFS. Although the execution time of each sub core is extremely short, the total time consumption of scanBfs is still similar to that of simpleBfs, and the main bottleneck still lies in the serial dependency between multiple cuMemcpy and cuCtxSync operations. The Nsight Systems timeline further confirms that computation and data copying cannot effectively overlap, and overall execution is frequently blocked by host control.

Finally, 'cuGunrockStyleBfsKernel' draws inspiration from Gunrock's ideas, but adopts a more streamlined kernel integration strategy that integrates frontier processing, neighbor traversal, and next round queue updates into a single kernel function, and achieves concurrency security through atomic operations. The experiment shows that this method performs better than 'SimpleBfs' in terms of throughput (13.31%) and memory bandwidth utilization (96.47%), but the total kernel function time is slightly higher (189.28

μ s). The reason may be that the thread distribution is more uniform, the scheduling lengthens the execution window, or the atomic operation pressure leads to frequent occurrence of warp level serial behavior. However, overall, this method has the advantages of simple structure, friendly scheduling, and fewer synchronization points, making it more suitable for deployment in scenarios that require stable GPU throughput.

In summary, each of the four methods has its own design philosophy and application applicability. ‘SimpleBfs provides a basic implementation with a clear structure but is easily affected by synchronization; ‘QueueBfs and scanBfs tend to focus on fine-grained control and scheduling optimization, but their potential has not been fully realized in the current implementation; And 'cuGunrockStyleBfsKernel' reduces synchronization overhead by unifying kernel functions, which is an effective implementation of Gunrock's idea in lightweight scenarios. Future work can introduce asynchronous memory copying and multi stream data management mechanisms while maintaining the concurrent structure of cuGunrockStyleBfsKernel, in order to further shorten the data transmission path and improve overall throughput performance. In addition, for multi kernel segmentation models such as scanBfs, their execution efficiency will highly depend on the topological characteristics of the graph. Therefore, it is necessary to further introduce graph structure adaptive scheduling strategies to achieve on-demand staged scheduling or single core fusion, in order to alleviate performance degradation caused by kernel granularity fragmentation.