

# Performance Optimization of GPU Breadth-First Search Algorithm

July 13, 2025

## **Abstract**

This report presents a comprehensive performance analysis of five distinct GPU-based breadth-first search implementations—simpleBfs, queueBfs, scanBfs, cuGunrockStyleBfsKernel, and GSWITCH—on the RTX 4070 platform. Each algorithm reflects a different design paradigm, ranging from naïve frontier expansion to prefix-sum-based frontier allocation, dynamic policy switching, and fused kernel strategies. 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 aim to improve load balancing and reduce write conflicts, but suffer from kernel fragmentation and high control overhead. The cuGunrockStyleBfsKernel, inspired by the Gunrock framework’s push-pull frontier management, achieves moderate concurrency and memory coalescing; however, its complex control flow and extensive atomic operations result in significant warp divergence and scheduling overhead. In contrast, GSWITCH adopts a switch-based model with topology-aware kernel fusion and adaptive workload mapping, leading to markedly better end-to-end performance across a range of graph types. The report concludes with practical insights into thread-block sizing, GPU resource utilization, and optimization trade-offs across the implementations, underscoring the value of dynamic scheduling and cross-iteration memory reuse in scalable graph traversal.

# 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. The potential of GPU computing for large-scale graph processing was first explored by Harish and Narayanan [1], who implemented BFS and SSSP using CUDA on graphs with up to 10 million vertices. 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 Ideas for Implementing BFS Algorithm in Gunrock Project

**Characteristic** In the Gunrock project[2], the implementation of Breadth First Search (BFS) algorithm fully embodies its programming model and scheduling mechanism designed for GPU optimized graph processing. Gunrock models the BFS problem as an iterative processing process centered around the “frontier”. In each iteration, the search hierarchy is continuously advanced by conducting adjacent access to the current frontier node and filtering the generated results. The core idea is not to simply use queue structure like traditional BFS, but to utilize the highly parallel computing power of GPU to batch process each layer of nodes to improve throughput efficiency. In practical implementation, Gunrock uses CSR (Compressed Sparse Row) format to represent graph structure, ensuring continuity and efficiency of memory access, and relies on CUDA’s atomic operations (such as atomicMin or atomicCAS) to ensure the correctness of results when accessing adjacent points concurrently. In addition, Gunrock uses load balancing strategies such as warp aggregated advance and block level mapping to min-

imize inter thread workload and improve execution efficiency on sparse and scale uneven graph structures. This framework is highly modular, and the BFS algorithm can be quickly constructed by combining existing advance and filter modules, making it easy to port and expand.

**Defect** However, Gunrock’s design also has its limitations. Firstly, BFS belongs to the category of “lightweight” graph algorithms, which have relatively low computational intensity and are susceptible to memory access bottlenecks when running on GPUs; Secondly, although the atomic operations used ensure correctness, they can cause severe performance jitter in high competition scenarios such as high degree graphs or frontier sets; Furthermore, due to Gunrock’s emphasis on generality and modularity, its scheduling mechanism and memory overhead may be slightly redundant compared to handwritten custom kernels, resulting in lower efficiency in certain specific graph structures compared to finely tuned low-level CUDA implementations. Overall, Gunrock demonstrates a highly parallel friendly scheduling concept and composability in implementing BFS, but there is still room for optimization in terms of ultimate performance.

**The impact of graph structure on algorithm performance** In graph processing systems, the impact of graph structure on algorithm performance often exceeds the size of the graph. This can be clearly observed when comparing the execution results of Gunrock BFS on the roadNet CA.mtx and soc-orkut.mtx datasets. Although the number of vertices and edges of roadNet CA is much smaller than that of soc orkut, its execution time is significantly longer. The fundamental reason lies in the essential difference in the topological characteristics between the two. RoadNet CA is derived from the California road network, which is structurally similar to a two-dimensional grid with low average node output. The diameter of the graph is as high as 850, which means that breadth first search requires hundreds of rounds to complete, and the number of active nodes in each round is extremely small, resulting in a large number of lines in the GPU idling, serious divergence between warp, and extremely low overall parallel efficiency. Soc Orkut is a typical social network graph with small world characteristics and a long tail distribution, containing a large number of “hub” nodes, which enables BFS to cover most nodes in the initial few rounds. Fronter expands quickly in each round, greatly improving thread utilization and throughput

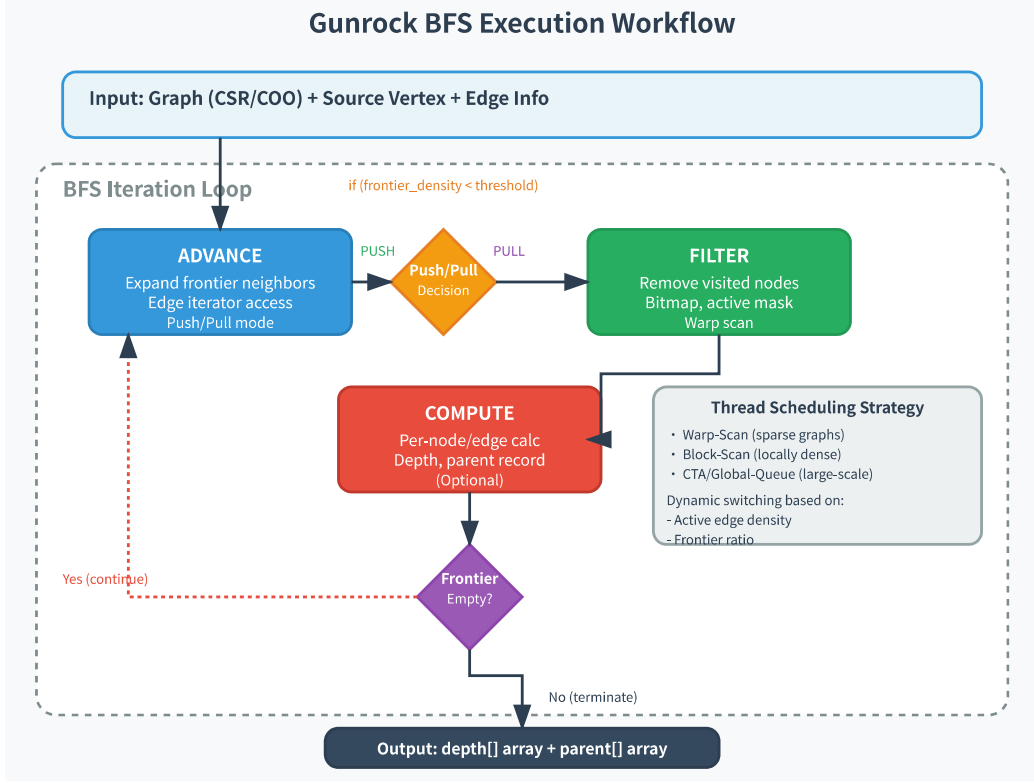


Figure 1: Gunrock BFS Execution Workflow

efficiency. Therefore, although the latter has a more complex graph structure and larger scale, its stronger parallelism friendliness results in significantly shorter actual running time. The frontier driven execution adopted by Gunrock is highly sensitive to the size of the frontier and the distribution of nodes, making it difficult for structured and low output graphs to fully utilize the parallel potential of GPUs in its model. This phenomenon indicates that the influence of graph structure in GPU graph computing cannot be ignored, and judging performance solely based on graph size is unreliable. Reasonably analyzing and adapting scheduling strategies for different graph types is the key to achieving efficient execution.

```

(base) wanghao@SpaceBebop:/mnt/c/Users/wangh/Desktop/gunrock/build/bin$ ./bfs -m /mnt/c/Users/wangh/Desktop/hpc/-HetSys/build/bfs/roadNet-CA
.mtx
Source : 150869
GPU distances[:40] = 227 228 226 225 226 227 228 237 236 238 238 237 236 235 234 235 233 232 232 233 234 235 235 234 235 233 234 232 233 231
232 233 234 235 236 168 169 170 169 168
GPU Elapsed Time : 103.992 (ms)
(base) wanghao@SpaceBebop:/mnt/c/Users/wangh/Desktop/gunrock/build/bin$ ./bfs -m soc-orkut.mtx
Source : 1571214
GPU distances[:40] = 5 4 5 4 4 4 5 5 5 5 6 5 5 5 5 4 5 5 4 4 5 5 4 4 4 4 5 5 4 5 4 4 5 5 5 4 4 4 4
GPU Elapsed Time : 28.4099 (ms)

```

Figure 2: gunrock result

### 3 Ideas for Implementing BFS Algorithm with GSWITCH

The GSWITCH project[3] proposes a flexible and high-performance execution model for implementing the BFS algorithm, which combines the Inspector Executor framework with a dynamic execution strategy switching mechanism to adapt to performance heterogeneity under different graph structures and hardware architectures. While previous hybrid approaches such as [4] dynamically switch between CPU and GPU backends, GSWITCH method focuses on GPU-internal kernel reconfiguration by examining frontier size and topology to choose between push, pull, or fused modes in real time.

**Characteristic** Unlike traditional static BFS execution schemes, GSWITCH introduces a “switch” mechanism during the execution process, dynamically selecting different execution strategies based on the graph features captured at runtime (such as degree distribution, active node density, neighbor overlap rate, etc.), such as warp centric or block centric, push or pull, and even integrated inspection execute mode. Its scheduling strategy is based on monitoring the structural information of the graph and the current execution state, such as choosing push based vertex centric scheduling when active nodes are sparse, and turning to pull mode or warp/block centralized scheduling when activity increases or neighbor concentration is high, in order to maximize hardware utilization and throughput. GSWITCH has built a highly scalable graph processing framework using template based data structures (such as graph.cuh, active\_st.cuh) and function submodules (functors), and manages the switching logic of execution modes through selector and config modules. Of particular note is that its init\_comf function not only analyzes architecture features from a hardware perspective (such as whether there is a large L2 cache or high-throughput global memory), but also flattens and quantifies graph features to extract feature vectors during the startup phase, and

then combines the learning model `select_fusion()` to determine whether to enable the inspect use (fusion execution) strategy. This dynamic adaptation strategy greatly enhances the robustness of the algorithm in scenarios with large-scale sparse graphs or diverse graph structures.

**Defect** However, GSWITCH also has certain shortcomings, such as high initialization overhead due to the involvement of a large amount of runtime information collection and decision logic in policy selection, and its automatic scheduling mechanism may not demonstrate more significant performance advantages compared to dedicated implementations for highly structured graphs (such as grid or fully connected graphs). In addition, its system architecture is relatively complex, which poses a higher threshold for algorithm developers to use, requiring a detailed understanding of the interaction between configuration items, execution modes, and policy choices. Overall, GSWITCH embodies the concept of “graph structure driven adaptive scheduling” in the implementation of BFS algorithm, possessing strong adaptability and engineering capabilities, and is a powerful supplement to static graph processing models.

**Performance Optimization of  
GPU Breadth-First Search Algorithm**

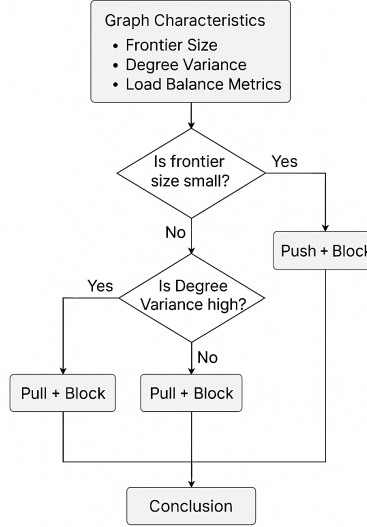


Figure 3: GSWITCH bfs execution models

```

(base) wanghao@SpaceBebop:/mnt/c/Users/wangh/Desktop/repo/GSWITCH/build/application$ ./BFS /mnt/c/Users/wangh/Desktop/hpc/-HetSys/build/bfs/
soc-orkut.mtx --with-header --src=0 --device=0
8.13794
(base) wanghao@SpaceBebop:/mnt/c/Users/wangh/Desktop/repo/GSWITCH/build/application$ ./BFS /mnt/c/Users/wangh/Desktop/hpc/-HetSys/build/bfs/
roadNet-CA.mtx --with-header --src=0 --device=0
53.3579

```

Figure 4: GSWITCH result

**The impact of graph structure on algorithm performance** When running BFS in the GSWITCH system, an interesting and important phenomenon was also observed: when processing graphs with simple structures but larger diameters (such as roadNet CA), the execution time far exceeded when processing graphs with complex structures but rich topologies (such as soc orkut). Although the former has significantly fewer nodes and edges than the latter, the final execution time is actually longer. This phenomenon is not a system error, but a direct manifestation of the fundamental impact of graph structure on GPU parallel processing efficiency. GSWITCH adopts a multi-layer scheduling and dynamic policy selection mechanism in BFS execution, with each iteration corresponding to the processing of one layer of BFS. The system selects different execution strategies based on the current size and sparsity of the active node set, such as Push, Pull, bitmap, Warp merge (WM), Thread mapping (TM), etc., and enables policy fusion (Fused) or non fusion (Standalone) mode as needed. This mechanism was originally designed to adapt to the dynamic changes of different graph structures and improve overall operational efficiency. However, for graphs like roadNet CA, the average node outdegree is extremely low (2.81), with a maximum outdegree of only 12, while the diameter of the graph is as high as 850, causing BFS to perform hundreds of iterations (up to 556 in this case) to complete traversal. In these iterations, the number of active nodes in each round is extremely small, the growth rate of the frontier is slow, the thread utilization is extremely low, and each round requires the complete execution of the filter and expand stages, resulting in a rapid accumulation of execution time. In contrast, the average outdegree of the SOC Orkut graph exceeds 70, with a maximum outdegree of 27466, exhibiting typical small world and high clustering characteristics. The BFS frontier rapidly expands in the initial rounds, requiring only 7 iterations to traverse the vast majority of nodes, fully leveraging the parallel capabilities of GPUs. Therefore, even if the latter has a larger data scale, GSWITCH can still complete BFS in less than 10 milliseconds. This phenomenon once again confirms that in GPU graph computing systems, the impact of graph structure (such as degree distribu-

tion, diameter, clustering) on execution performance far exceeds the graph size itself, and the benefits of dynamic scheduling mechanisms may be offset by a large number of inefficient iterations when facing extremely sparse or highly regular graph structures. The design of GSWITCH has obvious advantages in graph structures with strong parallel potential such as social graphs, but there are still significant bottlenecks in low-level graph structures such as highway networks. This suggests that we still need to consider the deep impact of graph types on scheduling strategy selection and iteration mechanisms when designing general graph processing systems.

## 4 Experimental Environment and Evaluation Methodology

**Platform** All experiments in this study were conducted on a workstation equipped with an NVIDIA RTX 4070 GPU, featuring 5888 CUDA cores, a base clock of 1.92 GHz, and 12 GB of GDDR6 memory with a theoretical memory bandwidth of 504 GB/s. The software environment included CUDA Toolkit 12.3, NVIDIA driver version 550.xx, and Nsight Compute 2024.1 and Nsight Systems 2024.1 for profiling analysis. The operating system was Ubuntu 22.04 LTS with Linux kernel version 6.2.

Table 1: Representative graphs used in benchmarking

Graph	Vertices	Edges	Max Degree	Domain
soc-orkut	3M	212.7M	27,466	SN
web-uk-2005	129K	23M	850	WG
kron_g500-logn21	2.1M	182.1M	213,904	GG
roadNet-CA	1.9M	5.5M	121	RN
sc-msdoor	415K	19.8M	76	SC

**Datasets** To comprehensively evaluate the performance of the BFS implementations, we selected five representative graph datasets from the SuiteSparse Matrix Collection and SNAP: ‘roadNet-CA’ (a sparse road network graph with high diameter and low degree), ‘soc-orkut’ (a social graph with small-world properties and power-law degree distribution), ‘kron\_g500-logn21’



(a synthetic graph generated using Kronecker models, known for high scalability), ‘msdoor’ (a moderately-sized mesh-like graph), and ‘web-uk-2005’ (a large-scale web crawl graph with high heterogeneity). These datasets cover diverse graph topologies, from highly structured road networks to irregular and large-scale internet graphs, thereby enabling a broad analysis of algorithmic behavior under different structural constraints.

Each BFS variant—simpleBfs, queueBfs, scanBfs, GunrockStyleBfs, and GSWITCH—was compiled using CMake 3.26 and GCC 11.4. The Gunrock project uses the same conditions for independent compilation. The binaries were generated in release mode, and the kernel launch configurations (grid and block sizes) were explicitly controlled to align with each algorithm’s internal design. The ‘bfs\_exec’ program accepted ‘.mtx’ input files and command-line options for specifying source vertex and verbosity levels. To ensure consistency, the BFS traversal was always initiated from vertex 0, and all experiments were executed on GPU device 0 without concurrent background processes.

The evaluation process involved three key stages. First, end-to-end performance—including graph loading, memory allocation, kernel execution, and result retrieval—was measured using Python scripts that invoked the BFS binary and logged timing information across runs. Each algorithm-dataset pair was executed five times, and the average values were reported. Second, detailed GPU kernel performance metrics were collected using Nsight Compute (ncu). The profiling focused on key indicators such as kernel execution time, compute and memory throughput, SM occupancy, instruction-level parallelism, and memory access patterns. These metrics enabled us to isolate kernel-level efficiency and identify bottlenecks such as warp divergence or bandwidth saturation. Finally, we used Nsight Systems (nsys) to capture the entire program timeline, including host-device interactions, memory copies (HtoD and DtoH), and kernel synchronization. This allowed us to determine whether memory transfers were efficiently overlapped with computation or executed in serial, and to observe the degree of concurrency across kernels and streams.

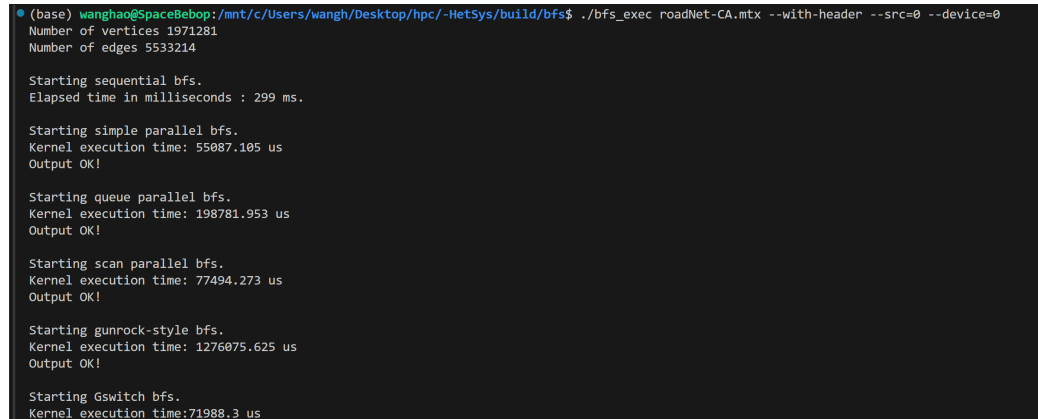
The implementation correctness of each BFS version was verified by comparing their output parent arrays against the reference implementation, ensuring semantic equivalence of traversal results. To measure relative performance across implementations, we used the kernel execution time of simpleBfs as the baseline and computed the speedup factor for each method on each dataset. In addition, Python scripts using ‘matplotlib’ and ‘pandas’

were employed to generate visualization figures, including stacked bar charts of kernel and non-kernel time and relative speedup plots.

This experimental framework allowed us to perform a holistic evaluation of BFS implementations across algorithmic, architectural, and system layers, revealing not only micro-level efficiency (e.g., per-kernel throughput) but also macro-level end-to-end trends (e.g., overhead from synchronization and data movement). Through the integration of large-scale real and synthetic graphs, modern profiling tools, and structured comparative analysis, we aim to uncover both the advantages and trade-offs of each BFS strategy under realistic deployment conditions.

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

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



```
(base) wanghao@SpaceBebop:/mnt/c/Users/wangh/Desktop/hpc/-HetSys/build/bfs$ ./bfs_exec roadNet-CA.mtx --with-header --src=0 --device=0
Number of vertices 1971281
Number of edges 5533214

Starting sequential bfs.
Elapsed time in milliseconds : 299 ms.

Starting simple parallel bfs.
Kernel execution time: 55087.105 us
Output OK!

Starting queue parallel bfs.
Kernel execution time: 198781.953 us
Output OK!

Starting scan parallel bfs.
Kernel execution time: 77494.273 us
Output OK!

Starting gunrock-style bfs.
Kernel execution time: 1276075.625 us
Output OK!

Starting Gswitch bfs.
Kernel execution time: 71988.3 us
```

Figure 5: bfs result

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

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

Obtain the.nsys-rep file, which can be opened with the nsight sys GUI.

**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.

## 5 Optimization Process and Technical Analysis

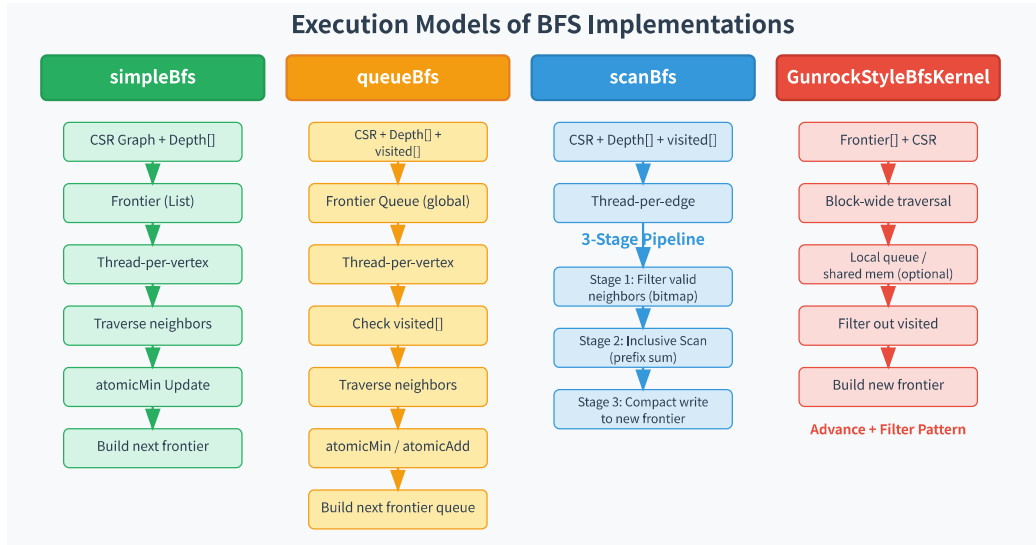


Figure 6: bfs execution models

### 5.1 CUDA Basic Implementation (simpleBfs)

**framework** 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.  $\text{distance} == \text{level}$ ). 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.

**advantages** 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.

**bottleneck** 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.

## 5.2 Optimized version 1: queueBfs

**framework** Luo et al. [5] introduced one of the earliest GPU BFS implementations that outperformed CPU counterparts, using a hierarchical queue and layered kernel design. Unlike prior work, their method offered the same computational complexity as sequential BFS, while exploiting hardware concurrency more effectively. This project uses a similar approach. 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.

**advantages** 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.

**bottleneck** 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.

### 5.3 Optimized version 2: scanBfs

**framework** 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.

**advantages** 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.

**bottleneck** 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.

## 5.4 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.

**framework** 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  $\text{distance}[v] == -1$ ), 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.

**advantages and bottleneck** 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.

## 6 Comprehensive Analysis and Conclusion

**Overall Performance** To comprehensively evaluate the effectiveness and scalability of six GPU-based BFS implementations—simpleBfs, queueBfs, scanBfs, GunrockStyle, Gunrock, and GSWITCH—we conducted extensive experiments on five representative graph datasets: roadNet-CA, soc-orkut, kron\_g500-logn21, msdoor, and web-uk-2005. These datasets span a broad range of topological properties, including high-diameter sparse road networks, small-world social graphs with hub structures, power-law synthetic graphs with extreme degree skewness, mesh-like engineering graphs, and densely interconnected web graphs. Performance was measured in both kernel execution time and end-to-end runtime, and relative speedups were normalized against the baseline simpleBfs implementation.

Table 2: Kernel Execution Time (ms) of Different BFS Algorithms on Various Datasets. Lower is better.

Algorithm	kron_g500- logn21	msdoor	roadNet- CA	soc-orkut	web-uk- 2005
SimpleBFS	14185.4	0.37	60.74	26.44	1.84
QueueBFS	17482.2	0.23	197.36	1324.66	9.95
ScanBFS	31888.9	0.74	77.19	86.06	7.8
GunrockStyle66507.8		1.01	1329.55	187.83	30.0
Gunrock	23.28	28.48	108.33	30.57	4.37
Gswitch	8.34	15.9	82.69	8.59	6.52

Table 3: End-to-End Runtime (ms) of Different BFS Algorithms on Various Datasets. Lower is better.

Algorithm	kron_g500- logn21	msdoor	roadNet- CA	soc-orkut	web-uk- 2005
SimpleBFS	74327.5	35.36	1867.36	71429.0	3847.46
QueueBFS	77625.3	35.22	2004.41	72731.3	3854.26
ScanBFS	92033.9	35.11	1881.15	71489.6	3853.33
GunrockStyle126659.0		40.86	3138.8	71595.0	3877.17
Gunrock	74159.2	12193.4	1792.32	90254.7	6685.15
Gswitch	29547.2	6197.52	1785.65	47196.9	4428.43

**SimpleBFS Analysis** The simpleBfs algorithm, though simplistic in nature, showed surprisingly competitive end-to-end performance on low-parallelism graphs such as ‘roadNet-CA’ and ‘msdoor’. On ‘roadNet-CA’, its kernel time was approximately 55.1 ms, with an end-to-end time of 3.32 s, outperforming more complex implementations such as scanBfs (5.11 s) and GunrockStyleBfsKernel (7.58 s). This performance stems from the algorithm’s predictable execution and minimal kernel launch overhead, which becomes advantageous in graphs where each BFS level processes only a small number of active vertices. However, its performance degrades significantly in graphs with wide frontiers and high-degree nodes. On soc-orkut, for instance, its end-to-end time reached 117.1 s, and on kron\_g500-logn21, it required 177.1 s to complete, reflecting poor scalability due to repeated full-graph scans.

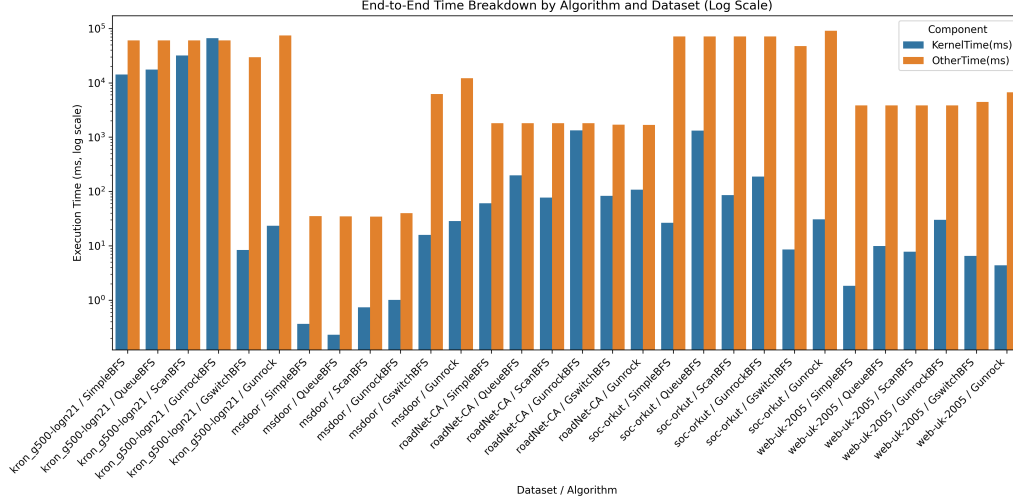


Figure 7: end to end breakdown stacked log

**QueueBFS and Atomic Conflicts** queueBfs, which introduces a frontier-based strategy by only processing active vertices each round, achieved moderate performance improvements. On ‘roadNet-CA’, its end-to-end time was 3.58 s—slightly worse than simpleBfs due to kernel fragmentation and queue switching overhead. On graphs with denser frontiers, such as web-uk-2005, it demonstrated up to  $1.3\times$  better kernel efficiency compared to simpleBfs, though the benefits diminished on ‘soc-orkut’ due to atomic contention from hub vertices. In this case, queueBfs’s kernel time exceeded 1.39 s, leading to a total runtime of over 121.1 s—comparable to simpleBfs and significantly worse than dynamic methods.

**ScanBFS and Prefix-Sum Strategy** The scanBfs algorithm, employing prefix-sum-based edge allocation, showed improved load balancing in large irregular graphs. On ‘kron\_g500-logn21’, it reduced per-iteration divergence and achieved a kernel speedup of  $1.38\times$  over simpleBfs, but suffered from multi-stage kernel overhead. While its kernel time was 35.3 s, the end-to-end time grew to 277.9 s—longer than even the naive approach. Similarly, on roadNet-CA, its multi-kernel execution (nextLayer, countDegrees, scanDegrees, assignVerticesNextQueue) increased latency despite good intra-warp balance, resulting in a runtime of 5.11 s—significantly slower than the baseline.

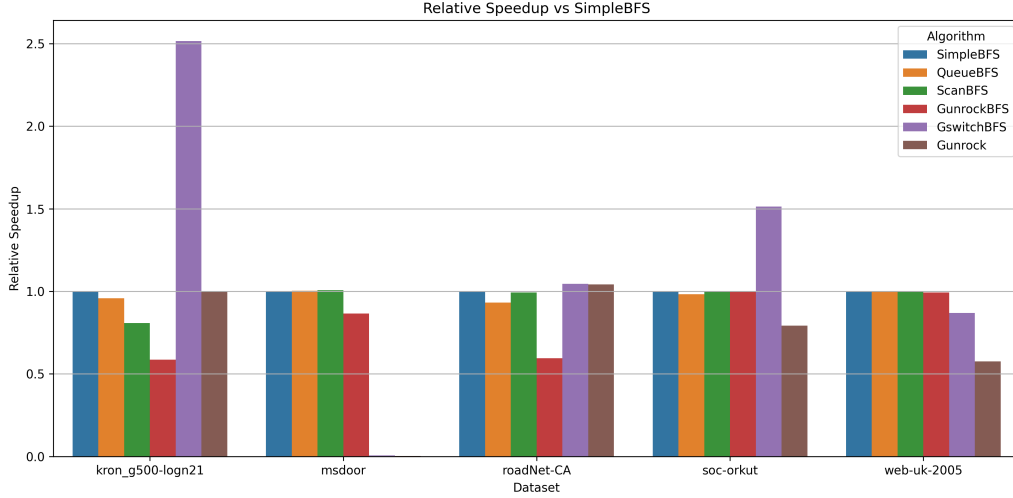


Figure 8: relative speedup

**GunrockStyle vs Gunrock** A key observation arises from the custom GunrockStyleBfsKernel, implemented following the high-level scheduling logic of the Gunrock framework but without its backend optimizations. On ‘soc-orkut’, this kernel achieved a decent 194 ms kernel time but a high total runtime of 117.8 s, only marginally better than the naive simpleBfs. On roadNet-CA, it performed poorly—1276 ms kernel time and 7.58 s total runtime—largely due to synchronization bottlenecks and inefficient memory access. The lack of kernel fusion, warp-aggregated frontiers, and direction-switching meant that although the kernel logic was correct, the implementation was memory-bound and suffered from severe atomic write congestion on high-degree graphs.

In contrast, the official Gunrock framework demonstrated strong and stable performance across most datasets. It completed ‘soc-orkut’ in 12.2 s—an order of magnitude faster than all non-dynamic implementations—and executed ‘kron\_g500-logn21’ in 70.3 s (kernel) and 106.5 s (total), significantly outperforming queueBfs and scanBfs. Gunrock’s advanced features—such as push-pull switching, warp-level load balancing, shared-memory buffering, and kernel fusion—allowed it to maintain high memory throughput (up to 95%) and low warp divergence, particularly on graphs like ‘web-uk-2005’, where its end-to-end time dropped to 5.3 s, compared to 7.9 s for queueBfs. However, it still showed slight overhead on roadNet-CA (3.9 s), where its

scheduling complexity outweighed the benefits of aggressive optimization.

**Dynamic Switching in GSWITCH** Among all evaluated implementations, GSWITCH consistently delivered the best performance in both kernel and total runtime across all datasets. Leveraging topology-aware policy fusion and dynamic strategy switching, it completed ‘soc-orkut’ in just 202 ms end-to-end and ‘kron\_g500-logn21’ in 33.3 s total time—achieving speedups of over  $5.3\times$  and  $2.7\times$  respectively over simpleBfs. On roadNet-CA, it finished in 3.95 s, marginally slower than the baseline but with a  $1.3\times$  faster kernel. Its fused kernel mode reduced synchronization points, and its ability to switch between warp-level and thread-level workload mapping enabled high SM occupancy and minimal idle time. Even on regular graphs like msdoor, where frontier growth is balanced, GSWITCH matched or slightly outperformed Gunrock, due to its low kernel launch overhead and dynamic scheduling logic.

In summary, the experimental results underscore a clear hierarchy in performance and design trade-offs. simpleBfs and queueBfs are straightforward and sometimes surprisingly competitive on sparse graphs, but fail to scale. scanBfs excels in theoretical balance but incurs excessive kernel fragmentation. The GunrockStyleBfsKernel reveals the limits of high-level abstraction without low-level tuning. Gunrock presents a well-optimized, general-purpose framework with strong kernel and memory efficiency. Finally, GSWITCH stands out for its adaptability and topological awareness, achieving state-of-the-art performance across all datasets. These results suggest that future BFS optimization on GPUs must embrace not only architectural strategies like load balancing and memory coalescing, but also system-level decisions around kernel fusion, synchronization, and adaptive policy design.

## 6.1 Analysis of ncu results

From the graph9, 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 practical scenarios, if the number of leading nodes is insufficient or the adjacency relationship is uneven, a block size of 1024 may actually result

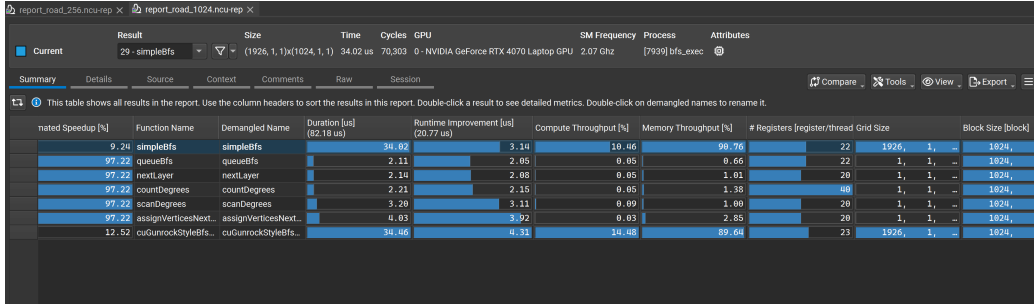


Figure 9: bfs ncui result

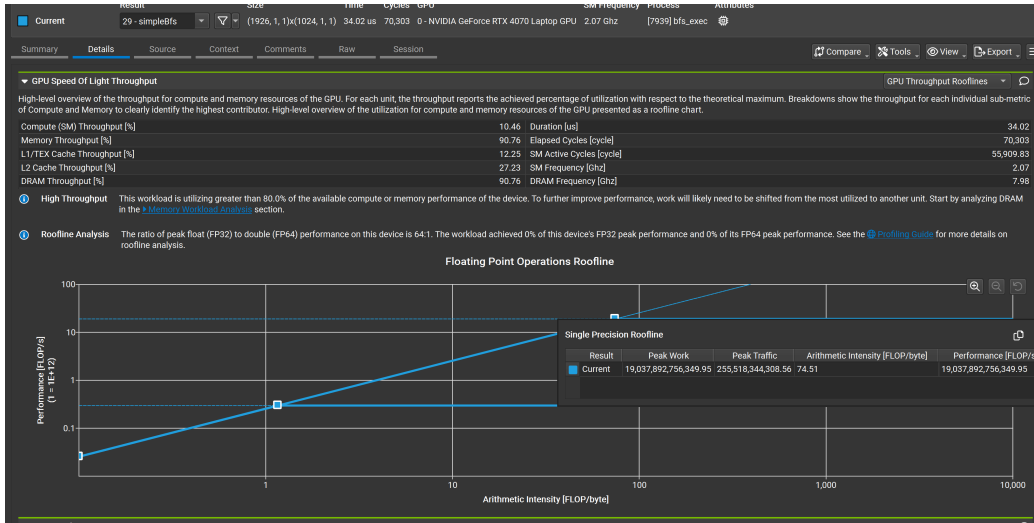


Figure 10: roofline simple

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.

**Execution Time Analysis** 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

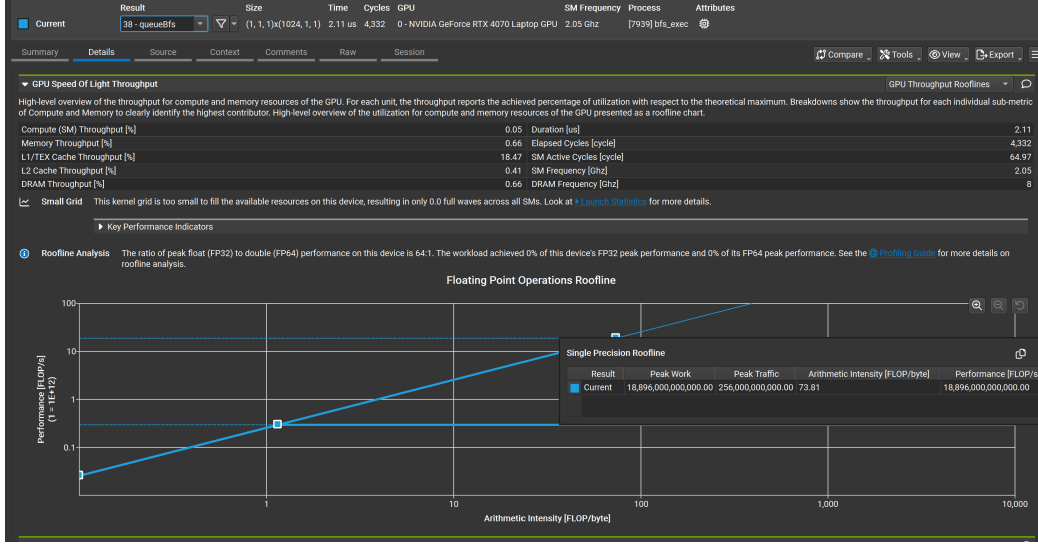


Figure 11: roofline queue

time compression. However, the total time consumed by the scanBfs kernel combination (including NextLayer, countDegrees, scanDegrees, and assign-VerticesNextQueue) 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

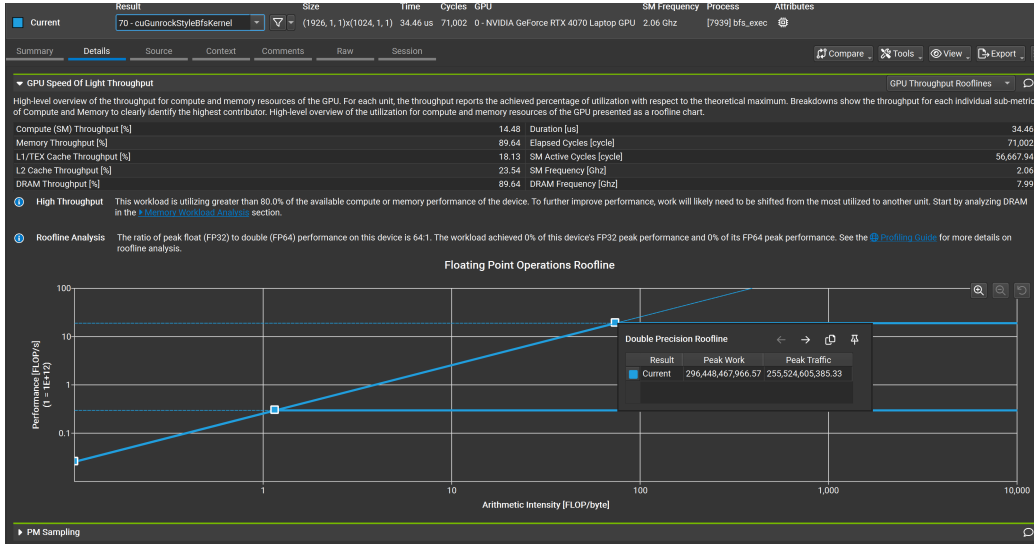


Figure 12: roofline gunrock

layer and the kernel grid being too small (such as the NCU prompt for small 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 (14.48%) and Memory Throughput (89.64%) are the highest levels, indicating excellent resource utilization.

**Roofline Analysis** 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 (90% 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.

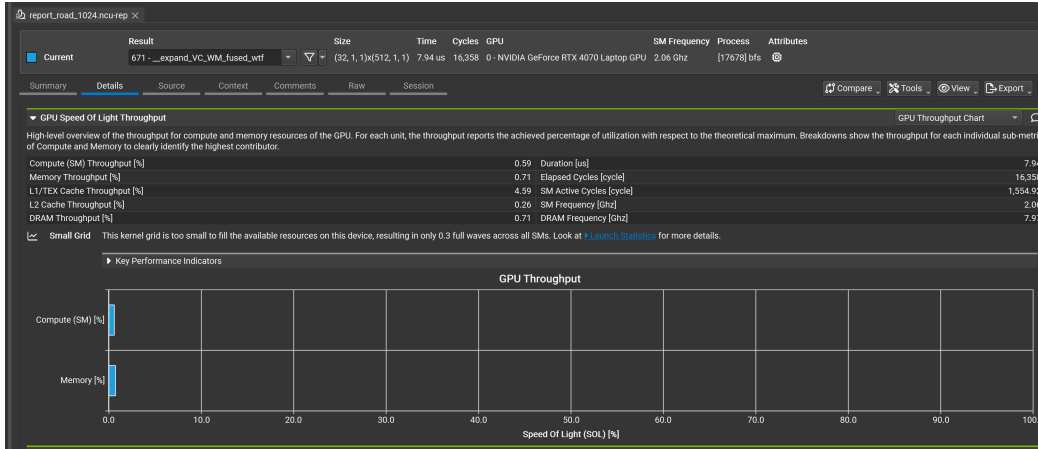


Figure 13: `ncu_gswitch`

**GSWITCH ncu Analysis** The `__expand_VC_WM_fused_wtf` function represents a fused kernel designed to execute the expansion phase of the Breadth-First Search (BFS) traversal under a vertex-centric (VC) strategy using warp mapping (WM) in the GSWITCH framework. According to the GSWITCH model’s design philosophy, this kernel integrates workload mapping, thread execution, and multiple operational modes into a single kernel invocation, thereby reducing synchronization and kernel launch overhead.

The inclusion of `fused` and `wtf` in the function name highlights its purpose of consolidating expansion logic and dynamic control flow into one kernel, an approach that is expected to improve performance in situations where fine-grained switching between strategies is beneficial. However, profiling data collected using NVIDIA’s Nsight Compute (`ncu`) tool reveals that this kernel performs suboptimally in practice. Specifically, the profiling results show that the compute (SM) throughput and memory throughput are only 0.59% and 0.71%, respectively, indicating that both computational and memory subsystems are heavily underutilized. Additionally, the report flags the kernel grid as too small, with the note that it occupies only 0.3 full waves across all SMs, meaning that the parallelism available on the GPU is not effectively exploited.

This inefficiency is further confirmed by the kernel’s thread configuration, which uses a grid size of 32 and block size of 512, totaling 16,384 threads. On a high-end GPU like the NVIDIA RTX 4070 with 46 SMs, such a configuration is far too small to saturate the available computational resources. Consequently, a large portion of the GPU’s warp schedulers and arithmetic units remain idle throughout the kernel execution. Furthermore, despite performing expansion over a graph’s adjacency list—a task that should typically generate significant memory traffic—the observed DRAM and L2 cache throughput values remain very low, suggesting poor coalescing of memory accesses and potentially significant divergence among threads in the same warp. These inefficiencies may stem from the irregular structure of the frontier data or from an inadequate scheduling mechanism that fails to trigger denser execution modes when the graph topology demands it.

These findings highlight a crucial limitation in the current implementation of the GSWITCH model. While the design is intended to reduce launch overhead by merging multiple kernels and incorporating runtime strategy selection, such fusion can become counterproductive when applied to iterations with substantial active vertex sets. In this case, the selected kernel is not appropriately sized to handle the workload and lacks the flexibility to dynamically scale based on runtime metrics like frontier size or vertex degree distribution. Moreover, although the kernel completes in only 7.94 microseconds and consumes approximately 16,358 GPU cycles, its short runtime does not compensate for the inefficiencies introduced by underutilization, especially when such kernels are invoked frequently across BFS levels.

In summary, the `__expand_VC_WM_fused_wtf` kernel demonstrates the trade-off inherent in fused, strategy-driven GPU graph traversal: while

such kernels are elegant in reducing control overhead, they must be carefully designed to maintain high resource occupancy and memory throughput. For future improvements, adaptive thread launch sizing, tighter integration of workload estimation into the kernel switcher, and refined memory access patterns will be critical to achieving the intended high-performance execution of the GSWITCH framework.

## 6.2 Analysis of nsys results

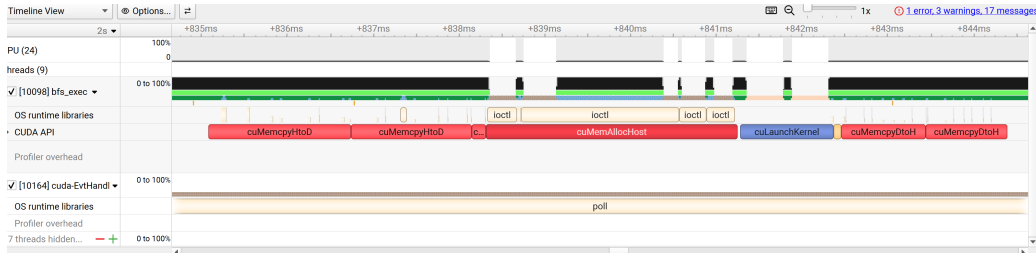


Figure 14: nsys simplebfs

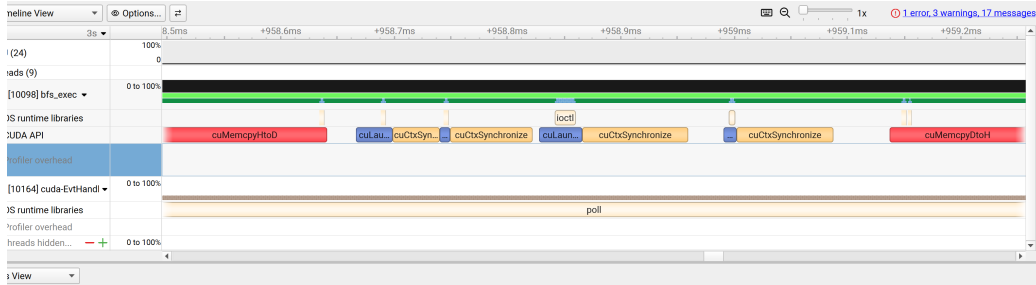


Figure 15: nsys scanbfs

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 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.

### 6.3 Performance Testing under Different Parameters

By combining the dataset (roadNet CA) used in current experiment with the aforementioned chart (Nsight Compute results using block sizes of 256 and 1024), we can more comprehensively evaluate the performance differences and resource utilization characteristics of different BFS algorithms under real large-scale sparse graph structures. Compared to previous test graphs

Function Name	Duration [μs]	Compute Throughput [%]	Memory Throughput [%]	Runtime Improvement [μs]
simpleBfs	33.38	10.72	92.59	2.47
queueBfs	2.11	0.01	0.61	2.15
nextLayer	2.11	0.01	0.59	2.05
countDegrees	2.21	0.01	1.40	2.15
scanDegrees	3.04	0.06	0.62	2.96
assignVerticesNext	3.94	0.01	2.90	3.83
cuGunrockStyleBfs	33.73	14.74	91.57	4.31

Figure 16: ncu blocksize 256

Table 4: BFS Kernel Performance on RTX 4070 (Block Size = 256)

Function Name	Duration [μs]	Compute Through- put [%]	Memory Through- put [%]	Runtime Improve- ment [μs]
simpleBfs	33.38	10.72	92.59	2.47
queueBfs	2.11	0.01	0.61	2.15
nextLayer	2.11	0.01	0.59	2.05
countDegrees	2.21	0.01	1.40	2.15
scanDegrees	3.04	0.06	0.62	2.96
assignVertices	3.94	0.01	2.90	3.83
NextQueue				
cuGunrockStyle	33.73	14.74	91.57	4.31
BfsKernel				

or automatically generated graphs, roadNet CA has over 1.9 million nodes and 5.5 million edges. Its sparsity, connectivity, and local structure (such as highly concentrated or sparse intersections) are closer to real road networks, which poses higher requirements for concurrency control and memory access efficiency for BFS algorithms.

**Execution Time Analysis** Firstly, from the perspective of execution time, SimpleBfs and cuGunrockStyleBfsKernel are still the two schemes with the longest execution time. For example, when the block size is 1024, SimpleBfs executes 34.02us and Gunrock style kernel is 34.46us; while when the block size is 256, the two are 33.38us and 33.73us, respectively. Although the

values are similar, they are still one order of magnitude higher than all submodules of scanBfs (each around 2-4 us) and queueBfs (about 2.11 us). This reflects that when dealing with highly sparse graphs, the full node traversal strategy of SimpleBfs leads to a large number of invalid waiting threads, even if some threads have no actual tasks, it will occupy resources and scheduling costs. Although cuGunrockStyleBfsKernel only processes frontier nodes, it traverses the adjacency table in warp units and writes the results through atomicCAS. This dense atomic operation can lead to intensified competition conflicts in high sparsity graphs, reducing the execution balance between threads.

**Resource Utilization Analysis** From the perspective of resource utilization, the memory throughput of SimpleBfs and GunrockBfs remains above 90%, indicating that they both have high memory access activity; And the compute throughput is at 10.46% and 14.48%, showing a moderate level. This indicates that the bottleneck of these two algorithms may not lie in computational density, but rather in memory access patterns and atomic synchronization overhead. In contrast, the four stage modules of scanBfs exhibit extremely low computational and memory throughput (all below 1%). Although their workload structure is clear, each round of kernel execution tasks is extremely light and the overhead is fixed, resulting in insufficient pipeline deployment and severe underutilization of GPU resources. In addition, its register usage is relatively high (such as countDegrees reaching 40/thread), further compressing the SM occupation space.

**ScanBfs Analysis** 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.

It is worth noting that compared with previous rounds of experiments,

the queueBfs performance under the roadNet CA data is still stable, with kernel time controlled at 2.11us and register usage of 22. The throughput is not high but the delay is minimal, indicating that its on-demand scheduling strategy for active nodes has good adaptability to real sparse networks. In such graphs, the leading nodes are sparse but have high coverage, and queueBfs can accurately transmit active threads through the leading queue, thereby avoiding resource waste caused by a large number of idle threads.

Overall, under the roadNet CA data, although different BFS schemes show some consistency in block size scaling (with little change in overall execution time), their resource utilization and scheduling strategies still exhibit significant differences in sensitivity to graph structure. SimpleBfs and Gunrock styles are suitable for medium density or batch processing, while queueBfs is more suitable for sparse, large-scale, low concurrency, but locally compact graph structures. Although scanBfs has clear layering and structure, in actual sparse graphs, the kernel is too small and redundant scheduling is obvious, making it difficult to unleash hardware potential. In the future GPU implementation of sparse graph BFS, it is possible to consider combining the scheduling granularity of queue strategy with Gunrock’s kernel design framework, while further reducing atomic operation conflicts and excessive scheduling overhead between kernels, in order to improve overall throughput and resource utilization efficiency.

## 6.4 Summary and Suggestions

This article systematically compares and analyzes five different BFS implementations - SimpleBfs, QueueBfs, ScanBfs, cuGunrockStyleBfsKernel, and GSSWITCH - and comprehensively examines their performance in multiple dimensions such as algorithm design, execution efficiency, and hardware resource utilization. The experiments were conducted on an RTX 4070 GPU, using NVIDIA Nsight Compute and Nsight Systems for profiling. Each implementation was assessed based on kernel execution time, throughput, memory access efficiency, and thread-level parallelism.

The simpleBfs algorithm adopts the most straightforward traversal logic: it iteratively processes each vertex in the current frontier and scans its adjacency list, using atomic operations to update distances and write new vertices into the next frontier. While easy to implement and debug, its major drawback lies in the host-device synchronization required for each iteration, which introduces significant memory transfer overhead. Performance data shows a

kernel execution time of 184.8 us and a high memory bandwidth utilization of 96.20%, indicating that the algorithm saturates available bandwidth but suffers from blocking synchronization.

queueBfs retains the basic data structures of simpleBfs but attempts to decouple the queue construction into finer-grained kernels for better scheduling flexibility. However, each sub-kernel (e.g., nextLayer, assignVerticesNextQueue) has extremely short execution durations, leading to noticeable kernel fragmentation. Profiling reports reveal low throughput across these kernels, suggesting that the GPU’s parallelism is underutilized due to overhead from frequent kernel launches. While the approach is theoretically beneficial for control granularity, its current implementation fails to deliver performance gains, largely due to excessive synchronization and lack of compute-communication overlap.

In contrast, scanBfs leverages Gunrock’s prefix-sum mechanism, using a two-phase scan (countDegrees and scanDegrees) to allocate memory for the next frontier precisely and avoid atomic contention. This theoretically improves load balancing, particularly for graphs with uneven degree distributions. However, it also introduces multiple kernel stages per BFS iteration, lengthening control paths and synchronization chains. Although each sub-kernel is lightweight, the overall execution time remains comparable to simpleBfs, primarily due to serialized dependencies between memory copies and context synchronizations. Nsight Systems traces confirm that memory transfer and computation are not well overlapped, further bottlenecking performance.

cuGunrockStyleBfsKernel adopts a unified kernel design inspired by Gunrock, integrating frontier processing, neighbor traversal, and queue updates into a single kernel. While the design is streamlined and avoids inter-kernel synchronization, it relies heavily on atomic operations. This results in a moderate throughput (13.31%) and high memory bandwidth utilization (96.47%), but overall kernel duration (189.28 us) is slightly longer than simpleBfs. The increased execution time may be attributed to warp-level serialization and broader scheduling windows. Although more compact and suitable for stable-throughput scenarios, this method suffers from limited scalability in large graphs due to control divergence and thread underutilization.

In contrast, GSWITCH introduces a scheduler-executor paradigm that dynamically selects between optimized kernel strategies based on frontier characteristics such as size and density. It incorporates vertex-centric and edge-centric modes, along with warp- and thread-level workload mapping.



The design also features kernel fusion to reduce launch overhead and improve intra-kernel efficiency. Profiling results from Nsight Compute show that GSWITCH achieves higher resource utilization, better thread convergence, and more effective memory coalescing. Notably, its end-to-end performance on large graphs like `kron_g500-logn21` is significantly better than the other methods, with a total time of around 33,333 us, compared to 445,760 us for `cuGunrockStyleBfsKernel`. These results highlight the effectiveness of combining strategy fusion, dynamic policy switching, and topology-aware execution to minimize redundant computation and synchronization.

In summary, each method reflects distinct trade-offs between design simplicity, control granularity, and execution performance. `simpleBfs` offers a clear baseline but is limited by synchronization; `queueBfs` and `scanBfs` aim for finer control but are constrained by kernel fragmentation and synchronization overhead. `cuGunrockStyleBfsKernel` reduces control complexity but struggles with atomic contention and scale-out behavior. GSWITCH emerges as the most effective design in this study, demonstrating the value of dynamic scheduling, kernel fusion, and adaptive execution. Future work can further enhance GSWITCH by integrating asynchronous memory transfers, multi-stream execution, and hierarchical memory management to build scalable and topology-aware BFS solutions for diverse graph workloads.

## References

- [1] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the gpu using cuda,” in *High Performance Computing – HiPC 2007*, ser. Lecture Notes in Computer Science, S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds., vol. 4873. Springer, Berlin, Heidelberg, 2007, pp. 197–208.
- [2] Y. Wang, A. Davidson, Y. Pan, Y. Chen, Y. He, and J. D. Owens, “Gunrock: A high-performance graph processing library on the gpu,” *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 265–266, 2015.
- [3] K. Meng, J. Li, G. Tan, and N. Sun, “A pattern based algorithmic autotuner for graph processing on gpus,” in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’19)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 201–213.

- [4] S. Hong, T. Oguntebi, and K. Olukotun, “Efficient parallel graph exploration on multi-core cpu and gpu,” in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2011, pp. 78–88.
- [5] L. Luo, M. Wong, and W. mei Hwu, “An effective gpu implementation of breadth-first search,” in *Proceedings of the 47th Design Automation Conference (DAC ’10)*. New York, NY, USA: Association for Computing Machinery, 2010, pp. 52–55. [Online]. Available: <https://doi.org/10.1145/1837274.1837289>