Establish the basis for Breadth-First Search on Frontier System: XBFS on AMD GPUs

Haoshen Yang
Rutgers University
hy482@scarletmail.rutgers.edu

Hao Lu
Oak Ridge National Laboratory
luh1@ornl.gov

Naw Safrin Sattar
Oak Ridge National Laboratory
sattarn@ornl.gov

Hang Liu
Rutgers University
hl1097@soe.rutgers.edu

Feiyi Wang
Oak Ridge National Laboratory
fwang2@ornl.gov

***Abstract*—Graphics Processing Units (GPUs) offer significant potential for accelerating various computational tasks, including Breadth-First Search (BFS). Numerous efforts have been made to deploy BFS on GPUs effectively. To address the dynamic nature of BFS, XBFS, the state-of-the-art work, employs an adaptive strategy that leverages different optimized frontier queue generation designs, accommodating the varying characteristics of levels in BFS. While XBFS demonstrates excellent performance on NVIDIA Quadro P6000 GPUs, it faces challenges when deployed on AMD GPUs. In this work, we present our efforts to implement XBFS's adaptive approach on Frontier, the most powerful supercomputer system, by porting XBFS to AMD MI250X GPUs. Through targeted optimizations tailored to the unique features of AMD GPUs, our implementation achieves an average performance of 43 Giga-Traversed Edges Per Second (GTEPS) per Graphics Compute Dies (GCD). Based on these results, we observe potential for surpassing the performance of the official Frontier results from the Graph500 benchmark released in June 2024.**

***Index Terms*—Breadth-First Search, AMD GPUs, Frontier SuperComputer.**

I. Introduction

BFS is the building block for many graph algorithms, providing essential functionality for various tasks. For instance, the Strongly Connected Component (SCC) detection algorithm utilizes both forward and backward BFS to identify SCCs within directed graphs [16,28]. Other algorithms, including Betweenness Centrality (BC) and subgraph matching, also rely heavily on BFS [4,14,15,22,24]. Beyond algorithmic applications, BFS is crucial in practical scenarios such as peer-to-peer network routing [30]. BFS's significance is ultimately resonated by its role in the Graph 500 benchmark, where it serves as a key evaluation metric for the world's most powerful supercomputers [9].

AMD GPUs are attractive platforms for accelerating BFS thanks to their exceptional computational and memory throughput capabilities. With continuous hardware advancements and a thriving community support, AMD GPUs offer improved computing and memory features. For example, the AMD MI250X GPU boasts 47.9 Tera-FLOPS of peak single-precision (FP32) performance, 128 GB of memory, and 3.2 TB/s of memory throughput, representing significant improvements over previous GPU generations [1].

Conventional BFS methods typically use the status array, which indicates the "status" (i.e., the visit level) of each vertex, to generate the frontier queue [20,31]. To address the workload imbalance problem, existing approaches assign different numbers of threads to vertices based on their degrees or use the Single Source Shortest Path (SSSP) algorithm to relax synchronization requirements, thereby mitigating the penalties associated with workload imbalance [3,20,25,27,31]. In contrast, XBFS introduces atomic operations and employs optimizations such as adaptive frontier queue generation, which better manage workload balance and improve overall performance [8].

In the June 2024 evaluation of the Graph500 benchmark, the BFS implementation on the Frontier supercomputer achieved a performance of 29,654.6 GTEPS [9] across 9,248 nodes. Since this BFS implementation is CPU-based, the average throughput per GCD is only 0.4 GTEPS (with each node containing 8 GCDs). This highlights the significant potential for accelerating BFS on Frontier by implementing a high-performance BFS on AMD GPUs, which could considerably improve the system's Graph500 benchmark performance.

In this work, we report our work-in-progress BFS implementation which achieves 43 GTEPS throughput on a single GCD, this is around 16.2% of the hardware peak

This work has been supported by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy (DOE) and NSF CRII award 2331536, Medium award 2212370, and CAREER award 2326141. We used resources of the Oak Ridge Leadership Computing Facility located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

throughput. This is notable for sparse graph algorithms on massively parallel GPUs, according to XBFS [8]. We believe this endeavor has established a solid basis for distributed BFS on AMD GPUs. Particularly, our development process began with the careful porting of XBFS to the HIP framework, ensuring compatibility with the Frontier system. We then focused on optimizing the code to exploit the unique architectural features of AMD GPUs. Our optimization process involved the following four key takeaways:

- Parameter Tuning: We conducted extensive experiments to fine-tune various parameters, adapting them to the specifics of the AMD GPU architecture.
- Control Flow Modifications: We implemented strategic changes in the control flow to preserve and enhance the performance benefits originally claimed by XBFS on NVIDIA GPUs.
- Degree-Aware Neighbor Order Re-arrangement: To further boost performance, we introduced a neighbor ordering technique that optimizes memory access patterns and improves cache utilization.
- Performance Profiling: Utilizing rocProfiler, AMD's performance analysis tool, we meticulously examined the code's behavior under various conditions. This allowed us to estimate optimal parameters for peak performance across different graph structures and sizes.

Through these optimizations and careful analysis, our HIP-based XBFS implementation has achieved a remarkable performance of 43 GTEPS on the Frontier system. To the best of our knowledge, this makes our implementation the fastest end-to-end BFS code ever developed for AMD GPUs across all platforms. The significance of this achievement extends beyond raw performance metrics. It demonstrates the potential of HIP as a framework for developing high-performance graph algorithms on AMD GPUs, potentially paving the way for future advancements in graph processing on exascale systems. In the following paper, we will delve deeper into the technical details of our implementation and provide a comprehensive analysis of our performance results compared to existing state-of-the-art BFS implementations.

II. Related works

Various methods were applied to improve BFS performance, but significant challenges remained, particularly in frontier queue generation and workload balancing. Different BFS levels require different approaches to frontier queue generation, and using a single strategy for the entire traversal is often inadequate.

- **Hierarchical Queue Method:** This method, as discussed in [23], performs well at levels with very few frontiers but suffers from enormous space consumption and inefficient strided memory access at levels with substantial frontiers.
- **Edge Frontier Filtering:** Techniques such as those used in B40C [25] and Gunrock [31] also struggle with excessive space consumption and duplicated frontiers at high-frontier levels, explaining Gunrock's difficulties with the FR graph as shown in Table 2.
- **Scan Approach:** Employed by Enterprise [20], this approach is designed for levels with a large number of frontiers but incurs noticeable overhead at levels with fewer frontiers.

While previous work [13,19] has explored adaptive graph traversal concepts, XBFS is distinct in focusing its adaptive approach specifically on frontier queue generation, a crucial step in graph traversal.

In terms of workload balancing, the runtime workload of each vertex in bottom-up BFS is determined dynamically. Most existing methods assume a vertex's degree indicates its associated workload. To accommodate various frontiers, [12] divides thread warps into sub-warps with different numbers of threads. B40C [20,25] extends this idea by assigning a cooperative thread array (CTA) to each frontier, followed by a warp and thread. Further efforts [6,17,18,26] propose pre-calculating the workload of each frontier and dividing them into segments to achieve balanced workloads. However, while the degree-workload association rule holds in top-down BFS, early termination in bottom-up BFS disrupts this association, rendering existing workload balancing optimizations ineffective. GraphGrind [29] also challenges the degree-workload tie in different graph algorithms, suggesting that various edges may yield different amounts of work.

Finally, SSSP-based asynchronous BFS can introduce excessive redundant work [10,27]. By employing the SSSP algorithm [3,5,27] for BFS traversal, synchronization among threads is eliminated. However, this design results in multiple updates of vertices across iterations, leading to redundant vertex revisiting. SIMD-X [21] identifies this redundant checking as a key factor contributing to SSSP's slower performance compared to BFS.

In addition to the graph operation-based approaches, linear algebra-based GraphBLAST [32] focuses on load balancing, memory management, and a simple programming model that performs comparable to Gunrock and other libraries. Authors accelerated GraphBLAST with bit-level optimizations of graph computations suitable for modern GPUs [7]. TurboBFS [2] also uses linear algebra and can achieve up to 40 GTEPs for irregular graphs with a smaller depth. Another recent scalable implementation, FSGraph [33] claims more than a hundred GTEPS on V100 GPU, using a GPU-friendly CSR, but not an open-source library.

Unlike traditional methods, XBFS adaptively applies various optimized and novel frontier queue generation techniques to address dynamic challenges throughout the BFS traversal [8]. Key strategies include scan-free, single-scan, and bottom-up approaches. The scan-free and single-scan methods challenge the conventional use of prefix sums [11] to merge frontiers across threads [20,25,31], showing that atomic operations are faster during the initial BFS levels. To tackle load balancing in the bottom-up phase, XBFS incorporates dynamic workload balancing to handle the unpredictable workloads caused by frequent early terminations. Additionally, during the bottom-up phase, XBFS checks whether the neighbors of unvisited nodes were updated, allowing it to proactively update frontier points for the next layer, further boosting efficiency.

III. Background

In this section, we introduce the algorithm and core concepts of XBFS, which utilizes three distinct frontier generation strategies: scan-free, single-scan, and bottom-up [8]. The choice of strategy is governed by a parameter, α. If the total number of neighbors for all vertices in the frontier queue is sufficiently large—specifically, if the ratio of neighbors to the total number of edges in the graph exceeds α—XBFS selects the bottom-up strategy. Otherwise, it switches between the scan-free and single-scan strategies based on the growth rate of the number of nodes in the frontier queue.

Here we use the graph in Figure 1 as an example to explain these frontier generation strategies.

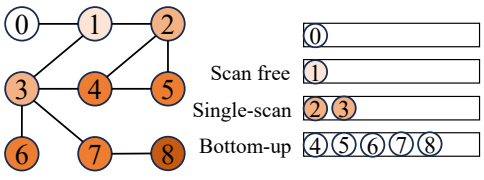

Fig. 1: Example graph and frontier queue.

A. Scan-free

The scan-free frontier queue generation method is mainly based on atomic operations, recognizing that even a single scan through the status array can be costly. This approach generates the frontier queue concurrently with the graph traversal. Specifically, the scan-free method uses atomic operations to update the status of each neighbor. If the status update is successful, indicating that the neighbor is being visited for the first time, an atomic operation is used to enqueue this neighbor into the next frontier queue. In summary, the scan-free approach employs atomic operations in two key phases: status updates and frontier enqueuing.

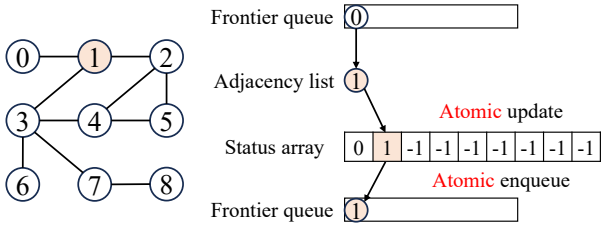

Fig. 2: Scan-free of example graph.

As shown in Figure 2, at the beginning of XBFS, we get vertex v_0 from the frontier queue and find its neighbor v_1. Then we atomic check and update v_1, since its status is this level, we atomic put it into the next frontier queue.

B. Single-scan

The single-scan frontier queue generation strategy also builds upon the aforementioned observations to enhance graph traversal. During traversal, each thread first loads the neighbors of the current frontiers and updates the status array with the first-time visited neighbors. Once the traversal is complete, the algorithm scans the entire status array and uses atomic operations to add those newly updated vertices to the next frontier queue.

It should be noted that the single-scan strategy has a variant that XBFS refers to as the *No Frontier Generation* approach. This variant uses conditional judgment to omit the unnecessary frontier queue generation stage when transitioning from the scan-free and bottom-up strategies to the single-scan strategy. Instead, the existing frontier queue is directly used for inspection and updates. This optimization eliminates the need for one or two scans and reduces the time required to generate the frontier queue, significantly improving performance on graphs with shorter diameters.

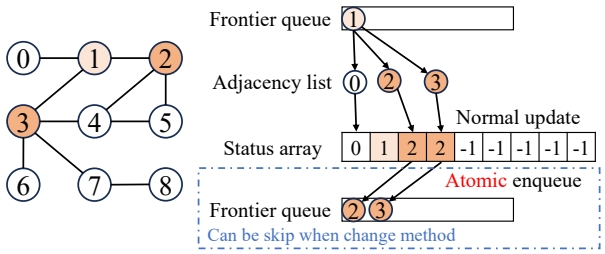

Fig. 3: Single-scan of example graph.

As shown in Figure 3, at the second level of XBFS, we get vertex v_1 from the frontier queue and find its neighbor v_0, v_2 and v_3. Then we normally check and update them. since the status of v_2 and v_3 is this level, we atomic put them into the next frontier queue. And

note that here actually this frontier queue construction can be skipped.

C. Bottom-up

XBFS calls its bottom-up frontier queue generation strategy double-scan frontier queue generation because it involves two scans of the status array. The double-scan method partitions the status array into multiple segments. The length of each segment is made evenly divisible by the size of the warp, which is the number of threads in a warp. In the first scan, an individual thread scans its assigned segment and obtains the number of frontiers vertex in it. Next, it performs a prefix sum to compute the global offsets for each segment to place the frontiers vertex to generate globally sorted frontiers. It then places those frontiers from each segment into the next frontier queue based on the global offsets.

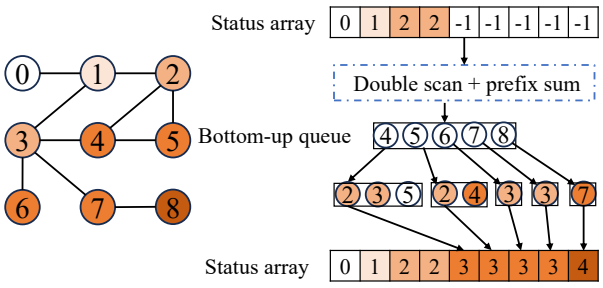

Fig. 4: Bottom-up of example graph.

As shown in Figure 4, when the ratio becomes larger, the XBFS chooses a bottom-up strategy. Here we double scan the status array and get the bottom-up queue, which are all unvisited vertex. Then check the adjacency list for each vertex and update the status. Here we can see almost all vertex find neighbors of the last level at the beginning and early terminated. And note that here since v_7 is updated in this phase, so v_8, which belongs to the next level, can be updated in this bottom-up.

IV. Porting and Optimization

XBFS, initially developed for NVIDIA GPUs, demonstrated excellent performance on systems equipped with NVIDIA Pascal Quadro P6000 GPUs. However, to harness the capabilities of the more powerful Frontier supercomputer and achieve higher performance, porting XBFS to AMD GPUs was imperative. This transition presented several porting challenges and necessitated a series of optimizations to ensure both compatibility with the HIP framework and performance enhancement on the AMD architecture. Additionally, we introduced new optimizations specifically tailored to AMD GPUs to further improve the performance of XBFS on Frontier.

A. Porting challenges

Differences in APIs from CUDA to HIP: To port XBFS from NVIDIA to AMD GPUs, we employed the hipify tool to convert the code from CUDA to HIP. However, certain CUDA APIs, such as `__any_sync` and `__shfl_sync`, which are integral to warp operations, are not directly supported by hipify. In CUDA, these APIs use a warp mask, but on AMD GPUs, this mask is unnecessary for their HIP equivalents, `__any` and `__shfl`. Due to the difference in warp and wavefront sizes between NVIDIA and AMD GPUs (32 threads per warp versus 64 threads per wavefront), we had to modify all warp-based code accordingly. This involved changing the mask type from `unsigned int` to `unsigned long` and ensuring that 64 threads work together to maintain optimal performance. Additionally, functions like `__popc` needed to be replaced with `__popcll` to handle the `unsigned long` mask on AMD GPUs.

Adaptive frontier queue generation on AMD GPUs: As the main contribution of XBFS, adaptive frontier queue generation continues to demonstrate significant performance improvement on AMD GPUs. Adaptive frontier queue generation can select one of three strategies—scan-free, single-scan, or bottom-up—based on the proportion of edges that need to be explored at the current level. At the beginning of the traversal, when the number of edges to be explored is small, the atomic update and construction of the frontier queue using the scan-free strategy are more advantageous. In the middle of the traversal, when a large number of frontiers need to be explored, the bit status check and update of the bottom-up strategy can achieve good results due to its fast termination. We port this optimization to AMD GPUs to achieve its significant benefit.

Warp-centric dynamic workload balancing and status update: After porting to AMD GPUs, the warp-centric approach performs well in both the scan-free and single-scan strategies by dividing vertices based on their degree. This method, combined with warp-centric and block-centric updating, allows for more efficient use of threads. However, in the bottom-up strategy, warp-centric dynamic workload balancing and status updates did not yield further improvements and even degraded overall performance. The reason of such degrade is due to the early termination in the bottom-up phase. High degree vertices that assigned with multiple threads terminate as quickly as the low degree vertices in practice, causing several threads in the warp remain idle. Furthermore, on AMD GPUs, the warp (wavefront) size is increase from 32 to 64, resulting even more wasted computing resources.

Compiler and register spilling: Regarding compilers, we tested both the `clang` and `hipcc` compilers on AMD GPUs. We found that `clang` delivered a better

performance for the bottom-up part of XBFS because it used fewer registers. On the Rmat25 dataset, it can reduce 17% of the runtime of an iteration of bottom-up. Additionally, we noticed that omitting the `-O3` optimization flag caused the code to run up to 10 times slower, primarily due to register spilling.

B. Optimizations

Cost of device synchronization: We found that device synchronization costs on AMD GPUs are significantly higher than on NVIDIA GPUs. To address this, we aimed to minimize device synchronization in the project. In the CUDA-based XBFS, three different streams were used to handle frontier queues with small, medium, and large degree frontier vertices. However, due to the high synchronization cost in HIP and the necessity of synchronizing different streams, we consolidated these streams into a single stream to process all frontier queues together.

Degree-Aware Neighbor Order Re-arrangement: By re-arrangement of the adjacency list of each vertex based on its degree, XBFS can achieve better performance. In the bottom-up strategy, early termination can effectively improve its performance by halting the search once a single neighbor with the target status is found. Ideally, we would like to re-arrange the adjacency list such that the first neighbor always has the lowest level. However, this is unachievable since the level is data and source vertex dependent. Instead, we use a re-arrangement based on neighbor degrees, moving neighbors with larger degrees to the front of the adjacency list. The intuition behind this approach is that high-degree vertices are more likely to be visited before low-degree vertices. A simple yet effective probability model supports this intuition: Given a graph G with m edges, assuming at level k, m_k edges have been visited, the probability of vertex i with degree d_i being visited is $1-[C(m-d_i, m_k)/C(m, m_k)]$. Here, $C(a, b)$ represents the number of ways to choose b items from a items. This formula indicates that, in general, vertices with larger degrees have a higher likelihood of being visited earlier.

TABLE I: Comparison of performance with Not Re-arranged and Re-arranged graph.

Level	Not Re-arranged		Re-arranged	
	FetchSize (KB)	Runtime (ms)	FetchSize (KB)	Runtime (ms)
0	3.31	0.0383	3.31	0.0369
1	6933.38	0.8096	6941.63	1.0970
2	2,572,656.53	8.4693	1,661,800.84	6.0604
3	707,405.69	2.3868	695,144.25	2.3274
4	616,971.94	5.8313	585,538.94	1.5481
5	233,464.75	0.5510	233,398.19	0.5615
6	108.81	0.0184	108.81	0.0182
Sum	**4,137,435.59**	**18.0862**	**3,182,827.16**	**11.6313**

As shown in Table I, we use the same seed for the Rmat25 dataset and compare the performance of re-arranged and not re-arranged datasets. We can observe that after re-arrangement, in the bottom-up strategy, the amount of data read from memory is significantly reduced, and the running time is also decreased accordingly, which is consistent with our hypothesis. As shown in Figure 8, this optimization achieves a 17.9% speedup on the Rmat25 dataset.

Reorganization of code: The ported code initially exhibited performance bottlenecks due to differences in how AMD GPUs handle certain operations compared to NVIDIA GPUs. We used the HIP toolkit to profile the code and analyze the time costs of both CPU operations and GPU kernels. Additionally, we used `rocprofiler` to identify the bottlenecks and determine where optimizations were needed. Based on this analysis, we rewrote kernels and reorganized the CPU part of the code to better align with AMD's performance characteristics, leading to significant improvements in execution time.

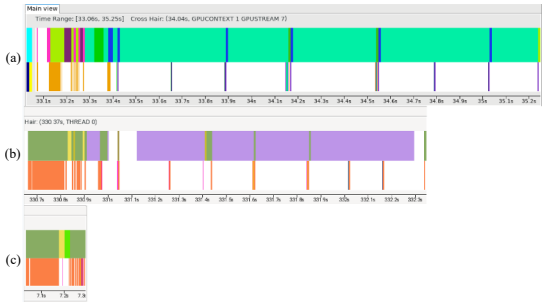

Fig. 5: Toolkit result of XBFS.

Figure 5 shows the runtime situation of each kernel. Figure 5(a) presents the result of the original XBFS on Summit, which is based on the CUDA environment, while Figure 5(b) shows the result on Frontier after simply hipifying the code and fixing all the bugs. Figure 5(c) shows the result after optimizations and kernel rewrites tailored to AMD's performance characteristics. The end-to-end time of XBFS was greatly reduced by rewriting the kernels and improving the code based on the performance of AMD GPUs.

V. Evaluation

A. Graph dataset

TABLE II: Graph datasets.

Graph	Vertices	Edges	Data size
LiveJournal (LJ)	4036538	69362378	478 MB
USpatent (UP)	6009555	33037896	268 MB
Orkut (OR)	3072627	234370166	1.7 GB
Dblp (DB)	425957	2099732	13 MB
Rmat23 (R23)	838809	134214744	1 GB
Rmat25 (R25)	33554432	536866130	4.3 GB

Graph datasets. Table II shows the six graph datasets we used throughout the evaluation to demonstrate the effectiveness of our optimizations.

B. Evaluation Setting

The evaluations were conducted on the Frontier supercomputer, a high-performance computing system optimized for AMD GPUs, features compute nodes that include a single 64-core AMD EPYC 7A53 "Optimized 3rd Gen EPYC" CPU with 512 GB of DDR4 memory. Each node houses 4 AMD MI250X GPUs, each containing 2 Graphics Compute Dies (GCDs), totaling 8 GCDs per node. These GCDs function as independent GPUs, each equipped with 64 GB of high-bandwidth memory (HBM2E).

C. Test of ratio of different dataset in each level

We test the ratio of the number of edges that need to be expanded at the next level to the total number of edges in the graph for different data sets. This shows the changes in the frontier queue of BFS in different data sets. The high ratio level is more suitable for the bottom-up strategy, while the low ratio method is more suitable for scan-free and single–scan.

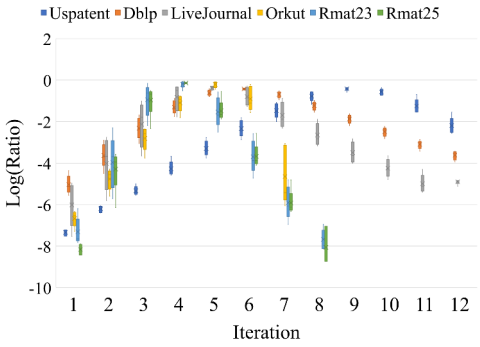

Fig. 6: Ratio of different datasets in each level.

As shown in Figure 6, here the box of each level shows the range the ratio with different initial seed. Here the log ratio means the logarithm of ratio with respect to 2. We can see that the BFS of the USpatent dataset requires more levels, followed by the Dblp dataset, while the sparse Rmat dataset requires fewer levels, which means that fewer cycles are needed to complete the BFS on the Rmat dataset.

D. Test of best α

We tested the performance of Scan-free, Single-scan and Bottom-up strategies on the Rmat25 dataset at different ratios. Because the running time of the Bottom-up strategy depends on the number of edges that have been visited, here we only select the levels from the beginning of BFS to the ratio rising to the maximum value. The running time of these levels can tell us how to set the optimal α value.

As shown in Figure 7, When the ratio is very small, that is, in the beginning stage of XBFS, the Scan-free strategy performs best. This is because there are very few edges that need to be expanded, and atomic operations have advantages in updating and establishing new frontier queues. Then when the ratio increases slightly and is less than 0.1, the Scan-free strategy performs best. However, since the number of visited edges is very small in the first few levels of XBFS, the Bottom-up strategy actually traverses and checks all edges almost linearly, so it is very slow. But when the ratio rises above 0.1, if the top-down strategy (Scan-free or Single-scan) continues to be used, the frontier queue that needs to be checked will be very large. At this time, the Bottom-up strategy has a greater advantage and can check and update all unvisited points in batches.

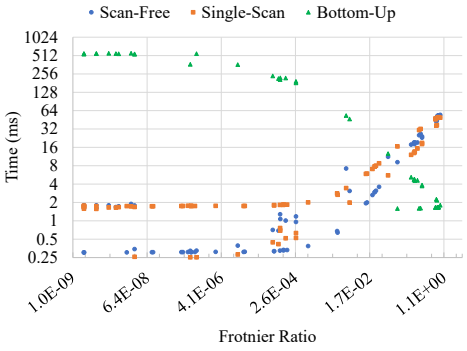

Fig. 7: The runtime of each method with different ratio(ms).

E. Test of rocprofiler

We use rocprofiler to test the detailed operating parameters of the kernel for Scan-free, Single-scan and Bottom-up strategies on the Rmat25 dataset.

In Table III, we show the performance counter for the scan-free strategy. It consists of only one kernel, and the memory access requirement depends linearly on the calculated ratio. Since the scan-free strategy accesses only the status and edge list of vertices in the current and next frontier, we can see it requires only a small amount of memory access at the lower level, $O(|F|)$.

TABLE III: Rocprofiler result of scan-free strategy on Rmat25. Here $L2$, $MBusy$, FS represent $L2CacheHit$, $MemUnitBusy$, and $FetchSize$, respectively.

Strategy	Ratio	Level	Runtime (ms)	L2 (%)	MBusy (%)	FS (KB)
Scan-free	1.86×10^{-9}	0	20.237	96.545	0.426	2.563
	1.02×10^{-6}	1	0.180	39.796	5.975	76.875
	5.44×10^{-3}	2	3.124	40.379	16.458	234139.875
	0.725	3	43.310	27.810	59.312	21699891.063
	0.267	4	24.265	37.327	81.438	9817098.875
	2.40×10^{-3}	5	0.540	5.574	66.119	229095.875
	1.35×10^{-5}	6	0.150	1.866	16.118	1453.438
	8.38×10^{-8}	7	0.140	50.685	0.189	12.938

In Table IV, we show the performance counter for the single-scan strategy. It consists of two kernels. The first kernel is for new frontier queue generation, which

has a memory requirement of $O(|V|)$, and the second kernel is for status array updates, which has a memory requirement of $O(|F|)$. Even though the second kernel requires a similar amount of memory as the scan-free strategy, the single-scan strategy eliminates the need for synchronization and duplicate writes to the frontier queue. We observe that the single-scan strategy has a higher L2 cache hit rate and higher memory read bandwidth compared to the scan-free strategy for frontier queue generation.

TABLE IV: Rocprofiler result of scan-free strategy on Rmat25. Here $L2$, $MBusy$, FS represent $L2CacheHit$, $MemUnitBusy$, and $FetchSize$, respectively.

Strategy	Ratio	Level	Runtime (ms)	L2 (%)	MBusy (%)	FS (KB)
Single-scan	1.86×10^{-9}	0	23.032	0.043	27.124	131073.875
	1.86×10^{-9}	0	0.299	97.668	0.188	1.750
	1.02×10^{-6}	1	0.477	0.001	26.216	131073.750
	1.02×10^{-6}	1	0.289	51.706	3.411	35.563
	5.44×10^{-3}	2	0.396	0.120	27.184	131112.438
	5.44×10^{-3}	2	1.744	48.287	16.441	139846.563
	0.725	3	0.876	37.822	51.138	205496.563
	0.725	3	37.788	28.582	57.616	20728852.500
	0.267	4	7.851	77.140	76.238	389393.250
	0.267	4	31.609	32.048	59.214	9526954.125
	2.40×10^{-3}	5	1.028	37.566	42.694	200315.563
	2.40×10^{-3}	5	2.711	55.466	33.885	566780.625
	1.35×10^{-5}	6	0.449	0.984	27.308	131582.438
	1.35×10^{-5}	6	1.789	70.360	28.566	341930.500
	8.38×10^{-8}	7	0.433	0.012	25.910	131077.938
	8.38×10^{-8}	7	1.764	70.591	28.653	339272.250

In Table V, we show the performance counter for the bottom-up strategy. Unlike both top-down kernels, it consists of five different kernels. The first kernel checks through the status and counts the number of unvisited vertices, which requires $O(|V|)$ memory access. The second and third kernels implement a prefix sum algorithm that does not require much memory. The fourth kernel generates a bottom-up frontier queue by revisiting the full vertex status, which requires $O(|V|)$ memory access. The last and most important kernel traverses the unvisited edges from all unvisited vertices, and in the worst case, the memory requirement is $O(|M|)$. We observe that the L2 cache hit rate and read bandwidth of the last kernel are not as high as in the top-down strategy. Therefore, the bottom-up strategy should only be used when a significant number of vertices have been visited.

In Table VI, we show the sum of memory read and runtime for all kernels at the same level under three different strategies. In the initial stages of XBFS, specifically at levels 0 and 1, the bottom-up strategy tends to add all unvisited vertices to the bottom-up queue. This process not only checks each vertex but also results in significant memory read overhead. As indicated in Table VI, the memory read for this strategy is substantially higher compared to the other two strategies.

TABLE V: Rocprofiler result of bottom-up strategy on Rmat25. Here $L2$, $MBusy$, and FS represent $L2CacheHit$, $MemUnitBusy$, and $FetchSize$, respectively.

Strategy	Ratio	Level	Runtime (ms)	L2 (%)	MBusy (%)	FS (KB)
Bottom-up	1.86×10^{-9}	0	20.193	3.225	31.228	133090.750
	1.86×10^{-9}	0	0.375	68.219	2.840	31.813
	1.86×10^{-9}	0	0.284	96.965	15.029	0.688
	1.86×10^{-9}	0	1.960	92.632	77.467	131075.750
	1.86×10^{-9}	0	546.222	28.297	14.531	27354527.688
	1.02×10^{-6}	1	0.447	3.176	29.373	133088.438
	1.02×10^{-6}	1	0.359	65.737	2.916	31.625
	1.02×10^{-6}	1	0.281	96.954	15.218	0.688
	1.02×10^{-6}	1	1.958	92.665	72.874	131076.563
	1.02×10^{-6}	1	540.707	28.320	14.702	27228927.688
	5.44×10^{-3}	2	0.464	3.167	29.734	133091.000
	5.44×10^{-3}	2	0.359	66.667	2.888	34.250
	5.44×10^{-3}	2	0.280	99.997	15.369	0.063
	5.44×10^{-3}	2	1.315	92.354	72.243	131076.750
	5.44×10^{-3}	2	46.410	28.880	36.786	7738606.125
	0.725	3	0.434	3.156	30.908	133098.750
	0.725	3	0.356	66.547	2.731	28.969
	0.725	3	0.280	96.955	16.243	0.063
	0.725	3	1.042	89.766	64.280	131076.000
	0.725	3	1.951	17.286	47.357	483963.875
	0.267	4	0.406	3.169	30.968	133095.188
	0.267	4	0.362	65.468	3.001	26.219
	0.267	4	0.290	99.997	16.206	0.938
	0.267	4	0.996	89.368	62.390	131076.375
	0.267	4	1.367	23.093	45.159	339673.781
	2.40×10^{-3}	5	0.401	3.173	30.345	133083.813
	2.40×10^{-3}	5	0.358	66.127	2.882	21.375
	2.40×10^{-3}	5	0.286	96.955	15.611	0.688
	2.40×10^{-3}	5	0.991	89.368	60.767	131076.313
	2.40×10^{-3}	5	1.342	23.287	43.855	338706.406
	1.35×10^{-5}	6	0.405	3.178	29.774	133093.188
	1.35×10^{-5}	6	0.361	65.647	2.907	29.469
	1.35×10^{-5}	6	0.286	99.997	15.469	0.938
	1.35×10^{-5}	6	1.022	89.367	61.956	131076.063
	1.35×10^{-5}	6	1.349	23.281	43.850	338691.406
	8.38×10^{-8}	7	0.406	3.171	29.049	133099.750
	8.38×10^{-8}	7	0.361	65.468	2.759	28.375
	8.38×10^{-8}	7	0.286	99.997	14.924	0.063
	8.38×10^{-8}	7	1.025	89.369	62.979	131076.438
	8.38×10^{-8}	7	1.380	23.283	44.198	338698.063

TABLE VI: Total memory read (MB) comparison for different strategies across levels

Level	Scan Free	Single Scan	Bottom up
0	**0.003** / 20.24	128.004 / 23.43	26971.413 / 569.25
1	**0.075** / 0.18	128.036 / 0.79	26848.755 / 543.93
2	**228.652** / 3.12	264.608 / 2.18	7815.242 / 48.98
3	21191.300 / 43.31	20443.700 / 38.78	**730.632** / 4.20
4	9587.011 / 24.27	9683.933 / 39.59	**589.719** / 3.54
5	**223.726** / 0.54	749.117 / 3.84	588.758 / 3.51
6	**1.419** / 0.15	462.415 / 2.28	588.761 / 3.53
7	**0.013** / 0.14	459.326 / 2.24	588.772 / 3.58

Furthermore, the single-scan strategy incurs higher memory read than the scan-free strategy because it performs a scan while establishing the frontier queue. Consequently, the scan-free strategy demonstrates superior performance during the early stages of XBFS since it doesn't need to

scan the status queue.

At level 2, the status array has not been updated on a large scale, and the majority of vertices are still unvisited. Thus, the bottom-up strategy still has a high memory read. At this point, the scan-free strategy and the single-scan strategy have similar memory read. However, the single-scan strategy has better performance because it doesn't need to use atomic operations to update the status array, despite its higher memory usage.

When the ratio is very large, that is, at levels 3 or 4, the frontier queue becomes extremely large (this is related to the average degree of the graph). At this point, the scan-free strategy and the single-scan strategy have many redundant vertices in the frontier queue, and a large number of redundant edges need to be checked, resulting in very large memory read. Here, the bottom-up strategy, based on the bottom-up generation and high-speed update of the wavefront, can achieve better results. Consequently, although the scan-free strategy and the single-scan strategy have higher MemUnitBusy, the bottom-up strategy still achieves the best performance.

After the bottom-up process is completed, which is at level 5, even though Table VI suggests that the scan-free strategy seems to have better performance, we actually use the single-scan strategy here. This is because when we run the XBFS, the single-scan strategy after the bottom-up can skip the frontier queue generation, often making it faster than scan-free at this level. Finally, in the last few levels, where level is 6 or 7, we can observe that the scan-free strategy has much less memory read, so we choose to use scan-free at these levels.

This careful study helped us ensure that the correct α value is selected for the Frontier system. It's important to note that using the α value allows us to estimate the memory access requirement for each level, theoretically reducing the overall memory access requirement. However, the actual processing time depends on system-specific features, such as the cost of atomic operations and irregular memory access patterns. For example, Level 2 in Table VI shows that the single-scan strategy accesses more memory, yet its total runtime is lower than that of the scan-free strategy. Similar considerations apply to the bottom-up strategy: if the cost of cache misses is higher, then the bottom-up approach may require a higher ratio of early terminations to be beneficial. Our analysis ensured that we take into account these factors for our XBFS implementation on the Frontier system.

F. Test of performance

We tested the performance of XBFS on Frontier, here we set $\alpha = 0.1$ and tested the n to n time and calculated GTEPS.

As shown in Figure 8, choose Gunrock for baseline and the performance of XBFS is much better. The

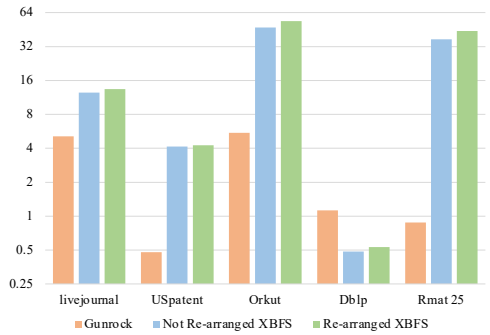

Fig. 8: Performance on Frontier (GTEPS).

performance on USpatent and Dblp is poor because these two datasets are more sparse, with a smaller average degree, and require more levels to complete BFS. The Dblp dataset is a very small dataset, so in the n to n test, most of the time is spent on processing data or completing the interaction between the CPU and the AMD GPU, resulting in poor performance. On datasets with a larger average degree, they only need fewer levels to complete BFS and achieve better performance. Among them, in the random dataset with a scale of 25 generated by the Rmat generator, a performance of 43 GTEPS was achieved.

Theoretically, the BFS algorithm requires visiting all vertices twice and edges once to complete all checks and updates. This means the predicted memory read is $8 * 2|V| + 4 * |M|$ bytes, for 8-byte edge indices and 4-byte vertex indices. For the Rmat25 dataset, we get a memory bandwidth efficiency of $((16|V| + 4|M|)/runtime)/1.6TFLOPS = 13.7\%$. From the results of `rocprofiler`, we can also get the actual hardware bandwidth efficiency of $((3.183GB)/runtime)/1.6TFLOPS = 16.2\%$. This is a rather high memory bandwidth efficiency for an irregular memory access application. Note that the real memory usage is larger than $16|V| + 4|M|$ due to the implementation overhead.

VI. Summary

In order to further explore the potential of the BFS algorithm on the MI250X GPUs on the Frontier system, we ported the XBFS algorithm, which performs well on NVIDIA GPUs, to AMD GPUs and added new optimizations, including parameter tuning, control flow modifications, degree-aware neighborhood re-ordering and performance profiling. This work greatly improved the end to end runtime of XBFS. Through targeted optimizations tailored to the unique features of AMD GPUs, our implementation achieves an average of performance of 43 GTEPS on the Rmat25 dataset, which reaching a predicted efficiency of 13.7% and hardware bandwidth efficiency of 16.2%.

REFERENCES

[1] AMD Instinct™ MI250X Accelerators. https://www.amd.com/en/products/accelerators/instinct/mi200/mi250x.html, 2024.

[2] Oswaldo Artiles and Fahad Saeed. Turbobfs: Gpu based breadth-first search (bfs) algorithms in the language of linear algebra. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 520–528. IEEE, 2021.

[3] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-gpu programming model for irregular computations. *ACM SIGPLAN Notices*, 52(8):235–248, 2017.

[4] Bibek Bhattarai, Hang Liu, and H. Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1447–1462, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. *Journal of the ACM (JACM)*, 67(5):1–27, 2020.

[6] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[7] Jou-An Chen, Hsin-Hsuan Sung, Xipeng Shen, Nathan Tallent, Kevin Barker, and Ang Li. Accelerating matrix-centric graph processing on gpus through bit-level optimizations. *Journal of Parallel and Distributed Computing*, 177:53–67, 2023.

[8] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. Xbfs: exploring runtime optimizations for breadth-first search on gpus. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, page 121–131, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Graph 500 Benchmark. http://www.graph500.org/, 2024.

[10] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a gpu. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 233–245. IEEE, 2017.

[11] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[12] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. *Acm Sigplan Notices*, 46(8):267–276, 2011.

[13] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 78–88. IEEE, 2011.

[14] Yang Hu, Hang Liu, and H. Howie Huang. High-performance triangle counting on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–5, 2018.

[15] Yang Hu, Hang Liu, and H. Howie Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182, 2018.

[16] Yuede Ji, Hang Liu, Yang Hu, and H. Howie Huang. ispan: Parallel identification of strongly connected components with spanning trees. *ACM Trans. Parallel Comput.*, 9(3), aug 2022.

[17] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50. IEEE, 2015.

[18] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252, 2014.

[19] Da Li and Michela Becchi. Deploying graph algorithms on gpus: An adaptive solution. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1013–1024. IEEE, 2013.

[20] Hang Liu and H Howie Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[21] Hang Liu and H Howie Huang. {SIMD-X}: Programming and processing of graph algorithms on {GPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 411–428, 2019.

[22] Hang Liu, H. Howie Huang, and Yang Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 403–416, New York, NY, USA, 2016. Association for Computing Machinery.

[23] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55, 2010.

[24] Adam McLaughlin and David A. Bader. Scalable and high performance betweenness centrality on the gpu. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 572–583, 2014.

[25] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *ACM Sigplan Notices*, 47(8):117–128, 2012.

[26] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices*, 53(2):622–636, 2018.

[27] Roger Pearce, Maya Gokhale, and Nancy M Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[28] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 550–559, 2014.

[29] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, 2017.

[30] Sabu M Thampi et al. Survey of search and replication schemes in unstructured p2p networks. *arXiv preprint arXiv:1008.1629*, 2010.

[31] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 1–12, 2016.

[32] Carl Yang, Aydın Buluç, and John D Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 48(1):1–51, 2022.

[33] Yuan Zhang, Huawei Cao, Yan Liang, Jie Zhang, Junying Huang, Xiaochun Ye, and Xuejun An. Fsgraph: fast and scalable implementation of graph traversal on gpus. *CCF Transactions on High Performance Computing*, 5(3):277–291, 2023.