SwarmGraph: Analyzing Large-Scale In-Memory Graphs on GPUs

Yuede Ji
George Washington University
yuedeji@gwu.edu

Hang Liu
Stevens Institute of Technology
hliu77@stevens.edu

H. Howie Huang
George Washington University
howie@gwu.edu

***Abstract*—Graph computation has attracted a significant amount of attention since many real-world data come in the format of graph. Conventional graph platforms are designed to accommodate a *single large* graph at a time, while simultaneously processing a large number of in-memory graphs whose sizes are small enough to fit into the device memory are ignored. In fact, such a computation framework is needed as the in-memory graphs are ubiquitous in many applications, e.g., code graphs, paper citation networks, chemical compound graphs, and biology graphs. In this paper, we design the first large-scale in-memory graphs computation framework on Graphics Processing Units (GPUs), SwarmGraph. To accelerate single graph computation, SwarmGraph comes with two new techniques, i.e., memory-aware data placement and kernel invocation reduction. These techniques leverage the fast memory components in GPU, remove the expensive global memory synchronization, and reduce the costly kernel invocations. To rapidly compute many graphs, SwarmGraph pipelines the graph loading and computation. The evaluations on large-scale real-world and synthetic graphs show that SwarmGraph significantly outperforms the state-of-the-art CPU- and GPU-based graph frameworks by orders of magnitude.**

1. Introduction

Graph, made up of vertex and edge, is a natural representation for many real-world applications [1], [2], [3], [4]. As graph algorithms can help to mine useful and even hidden knowledge, they are of particular importance. Towards that need, we have witnessed a surge of interests in graph computation in various directions, including shared memory, distributed system, and graphics processing unit (GPU) [5], [6], [7].

1.1. Motivation

While a vast majority of the graph computation framework, by default, assumes to process a single large graph, this paper unveils that rapidly processing a zoo of in-memory graphs is equally important. Here, we regard a graph as in-memory if it is able to reside in the GPU shared memory (up to 96KB in Nvidia Tesla V100 GPU). Such an in-memory graph framework is needed from the following three aspects.

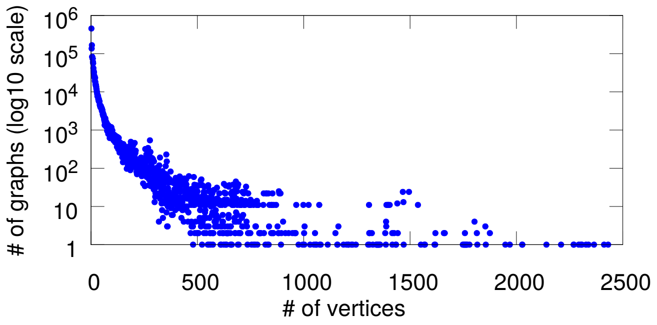


Figure 1: Graph size distribution of the 1.8 million control flow graphs for binary code vulnerability detection from [11], [12].

First, in-memory graphs are popular in real-world applications. Here, we will elaborate on four of them. (1) In the paper citation, the graph sizes (in terms of vertex count) are smaller than 100 for the 19.5 thousand graphs from a DBLP dataset [8]. (2) In the chemical compound, the graph sizes are smaller than 200 for all the 8.2 thousand graphs from NCI1 and NCI109 dataset [8], [9]. (3) In biology, the graph sizes of a protein dataset with 1.1 thousand graphs are smaller than 600 [8], [10]. (4) In binary code vulnerability detection, the control flow graph (CFG) comes with a small number of vertices as shown in Figure 1. Note, each function inside the binary code is represented as a CFG, where a vertex denotes a code block and an edge denotes the control flow relationship between code blocks [11], [12], [13].

In addition, concurrently processing a collection of in-memory graphs is widely needed, while the system dedicated to this direction, to the best of our knowledge, is absent. Still using the binary code vulnerability detection as an example, for each control flow graph, one needs to compute the betweenness centrality (BC) and other algorithms to find the important vertex and edge [11], [12], [13]. However, current graph frameworks, which *run one algorithm on a single graph at a time*, will take an absurdly long time to process a million control flow graphs which is the case in the binary vulnerability detection works [11], [12], [13].

Third, although recent graph computation methods on GPU greatly improve the performance [14], [15], they usually aim at a single large graph, which will cause low resource utilization when applied to a large number of in-memory graphs. To begin with, conventional graph frameworks routinely assign all GPU threads to work on one graph at a time, while this single in-memory graph is usually not able to saturate all the GPU resources, i.e., threads. In addition, due to the limitation of global thread synchronization support, these frameworks have to either terminate the kernel after each iteration or use the expensive GPU device level synchronization, e.g., cooperative groups [16], to synchronize all the threads. Further, existing frameworks place computation metadata[1] in slow global memory because it assumes they are too big to be stored in the fast but small GPU memory components, e.g., shared memory.

1.2. Contribution

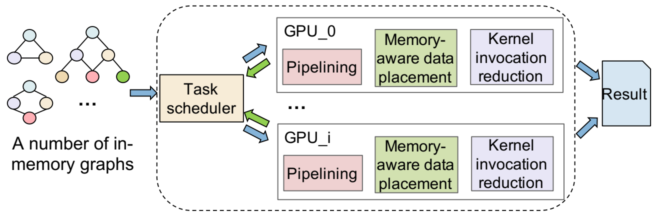
In this work, we have designed a new GPU-based graph computation framework for processing a large number of in-memory graphs, namely SwarmGraph. As shown in Figure 2, given many in-memory graphs, SwarmGraph leverages the task scheduler to distribute the workload to each GPU. On each GPU, SwarmGraph applies our newly designed pipelining technique to pipeline the graph loading and computation for many graphs. For each graph, our memory-aware data placement strategy carefully places different data in different memory components to reduce the memory access latency. Further, our kernel invocation reduction (KIR) strategy carefully schedules the GPU threads to reduce the number of expensive kernel invocations and avoid global memory synchronizations. The contributions are three-fold.

First, observing the conventional GPU platforms are not efficient for computing one in-memory graph, we design two new techniques, memory-aware data placement, and kernel invocation reduction. They are able to leverage the fast memory components in GPU, remove the expensive global memory synchronization, and reduce the number of costly kernel invocations. With these optimizations, SwarmGraph is able to outperform both CPU- and GPU-based graph frameworks by several orders of magnitude.

Second, to accelerate the computation of many graphs, we design a pipelining technique motivated by the fact that the data dependency only exists in the graph loading and computation of one graph, while does not exist between different graphs. The pipelining technique is able to bring $1.9\times$ speedup for the computation of many graphs.

Third, we have implemented SwarmGraph with three frequently used graph algorithms, all-pairs shortest path (APSP), closeness centrality (CC), and betweenness centrality (BC). We tested SwarmGraph with two graph benchmarks, i.e., protein graphs and one million synthetic graphs. For one graph computation on the protein graphs, SwarmGraph achieves $314\times, 93\times, 24\times$, and $19\times$ speedup over the parallel CPU implementation, conventional GPU, Gunrock, and Hybrid BC, respectively. Further, for many graphs

1. We regard the original vertex and edge information as the graph data, the intermediate information of the vertex and edge as metadata [14].


A number of in-memory graphs
Task scheduler
GPU_0
Pipelining
Memory-aware data placement
Kernel invocation reduction
GPU_i
Result
Figure 2: Overview of SwarmGraph.

computation on the one million synthetic graphs, SwarmGraph is able to significantly reduce the total runtime of the three algorithms from 248.5 to 34.4 minutes.

Due to the page limit, we moved out some detailed descriptions, that can be found in the full technical report [17]. The rest of the paper is organized as follows: Section 2 introduces the background. Section 3 presents the design of SwarmGraph. Section 4 evaluates SwarmGraph. Section 5 summarizes the related works. Section 6 discusses and Section 7 concludes the paper.

2. Background

This section discusses the background of the CUDA programming model and GPU architecture.

The Nvidia CUDA programming model includes grid, cooperative thread array (CTA), warp, and thread [18], [19]. A grid consists of multiple CTAs, which are also known as blocks. A grid is mapped as one GPU chip from the hardware point as shown in Figure 3. A GPU chip includes many multiple processors (MPs), and each MP handles the computation of one or more blocks. Each MP is composed of many streaming processors (SMs) that handle the computation of one or more threads.

In each grid, all the threads share the same global memory, texture memory, and constant memory as shown in Figure 3. The texture memory and constant memory are read-only, while global memory can be read and written. In each block, all the threads share the same shared memory, which can be read and written. It can be accessed by the threads in the same block, but cannot be accessed by outside threads. The configurable shared memory and L1 cache share the total memory size (64KB for Tesla K40 GPU).

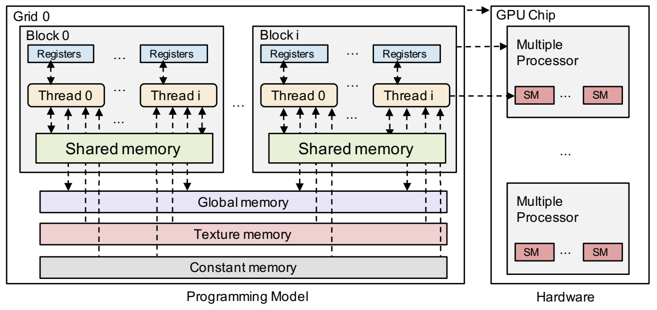


Figure 3: GPU architecture and CUDA programming.

TABLE 1: Data placement comparison with conventional graph platforms. GDDR means GPU global memory, the latency is in the clock cycle.

Memory (Tesla K40)			Conventional graph platforms [14], [5]	SwarmGraph
Name	Size	Latency		
Register	65,536	-	on-demand	Graph, metadata
L1 + Shared	64KB	~20	Hub vertex	Metadata
L2 cache	1.5MB	~80	on-demand	on-demand (graph)
GDDR	12GB	~400	Graph, metadata	Graph, metadata

Each thread is assigned with its own registers for reading and writing.

Among the read and write memory, global memory is off-chip, while the shared memory is on-chip. Therefore, the shared memory has a smaller size, but much higher bandwidth and lower latency. Taking Tesla K40 GPU as an example (Table 1), whose compute capability is 3.5, the global memory size is 12GB, while the shared memory size is 48KB per block. However, the shared memory latency is about 20 times faster than global memory. Thus, moving the computation to the shared memory can significantly reduce the latency.

3. Design of SwarmGraph

This section presents the design of SwarmGraph. Firstly, we will discuss the techniques for computing one single in-memory graph, i.e., memory-aware data placement and kernel invocation reduction. Later, we will discuss our pipelining technique for many in-memory graphs. Finally, we will discuss the currently implemented graph algorithms.

3.1. Memory-Aware Data Placement

Our memory-aware data placement aims to place different types of data on different memory components to fully utilize the resources. In general, we try to put the frequently and randomly accessed metadata in the fast memory components, such as shared memory and register, while the graph data in the global memory.

The graph computation typically operates on two types of data, namely, graph data and computation metadata. Using the all-pairs shortest path algorithm as an example, the graph data is the original graph representation while the metadata is the distance array that records the distance from every vertex to others. The access patterns of these two data types vary dramatically. The graph data is accessed sequentially and consecutively since the neighbors of a vertex are stored adjacently. However, the metadata is accessed randomly since each vertex may connect to random neighbors. Also, the metadata is frequently updated. Therefore, metadata suffers from frequent random access, which will lead to severer cache misses than graph data.

The conventional graph computation frameworks, e.g., Gunrock [5], put both graph data and metadata in the global memory. Differently, SwarmGraph puts the frequently updated metadata into the fast shared memory, while loads the graph data into global memory. Table 1 presents the way

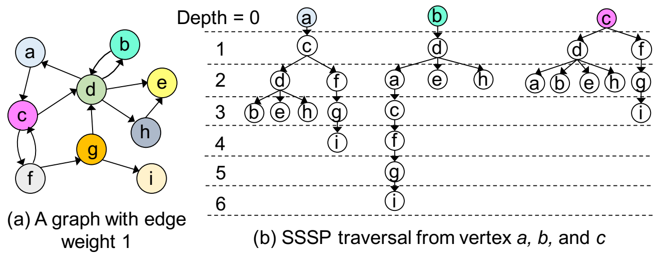

Depth = 0
1
2
3
4
5
6
a
b
c
d
e
f
g
h
i
(a) A graph with edge weight 1
(b) SSSP traversal from vertex a, b, and c

Figure 4: Computing all-pairs shortest path on an example graph shown in (a), the single-source shortest path traversal from vertex a, b, and c are shown in (b).

we organize different data structures in SwarmGraph. As metadata is mostly randomly accessed in graph computation [20], such a design is able to tremendously improve the performance. From the access latency wise, the shared memory is about 20× faster than the global memory as shown in Table 1. Although the L2 cache is on-demand for SwarmGraph, it is worth noting that the whole graph data is likely to reside in it for two reasons. First, algorithmic metadata will not compete with graph data for the L2 cache space, leaving graph data exclusive resident of the L2 cache. Second, the graph of interests is sufficiently small that can fit in the L2 cache.

3.2. Kernel Invocation Reduction

It usually takes several iterations to converge for graph computation. Taking the all-pairs shortest path (APSP) algorithm as an example, one needs to run the number of iterations equaling the length of the longest path. For the example graph in Figure 4(a), one needs to run 6 iterations because the longest path is 6 (from vertex *b*). During computation, one needs to synchronize after each iteration to get the correct result. Prior methods rely on terminating the GPU kernel at the end of each iteration to achieve global synchronization [14]. Such a design needs to invoke the kernel multiple times, e.g., 6 for the example graph. This is low efficiency because, one, kernel invocation itself is costly [21], two, the synchronization will generate extra global memory traffic [22].

Motivated by that, we design the kernel invocation reduction (KIR) technique to significantly reduce the number of kernel invocations. Firstly, KIR tries to divide the entire computation task into a number of independent sub-tasks. For the all-pairs shortest path (APSP) algorithm, one can divide it into many single-source shortest path (SSSP) computations running from every vertex. For the example graph with 9 vertices in Figure 4(a), the APSP can be divided into 9 independent SSSPs. KIR is used in closeness centrality (CC) for computing APSP and betweenness centrality (BC) for computing single-source betweenness centrality starting from every vertex.

Secondly, leveraging the small size property of the in-memory graph, one can assign a single block to compute a sub-task without communicating with others. In other words, after one iteration, one only needs to synchronize the metadata with the threads in the same block. For the example graph in Figure 4(a), the SSSP computation metadata from vertex a in Figure 4(b) does not need to synchronize with the one from vertex b. As the metadata is stored in the shared memory by our memory-aware data placement technique, we can synchronize it with the well supported intra-block synchronization, i.e., `__syncthreads()`. In this way, we are able to replace the synchronization from the expensive global memory with the fast shared memory.

To this end, KIR is able to reduce the number of kernel invocations by over one order of magnitude. Particularly, for one in-memory graph, KIR only invokes the kernel by 1, 2, and 2 times for all-pairs shortest path, closeness centrality, and betweenness centrality, respectively. KIR works for the graph algorithms that can be divided and conquered. Not limited to the three algorithms, many others fall into this discipline, such as node embedding generation algorithms (Node2vec [23], DeepWalk [24], LINE [25], etc.), PageRank, and other centrality algorithms.

3.3. Pipelining

With a GPU, one needs two basic steps to compute a graph, that is, loading a graph from the disk into the memory, and computing on the GPU. With many graphs, this process will be repeated till the end as shown in Figure 5. Particularly, the CPU handles the loading process, and the GPU works on the computation. That means, the CPU is idle while the GPU computes the graph and vice versa. In this process, the data dependency only exists between the loading and computation of the same graph but does not exist between two graphs.

Motivated by that, we design a pipelining technique for many graphs scenario. That is, we split the graph loading and computation into two tasks. In particular, one CPU thread keeps loading the graphs from the disk into the memory, while another CPU thread launches the GPU kernel to compute the graphs. The pipelining design is shown in Figure 5. In our implementation, we maintain a task queue shared by the two threads to accomplish this. The graph loading thread loads the graph and pushes the graph into the queue, while the computation thread reads from the queue, and invokes a GPU kernel to compute. Such a two-way pipelining design is able to get up to 2× speedup which can be reached when the loading and computation take the same time.

We also leverage the asynchronous APIs provided by CUDA, such as memory copy and memory set, to overlap the execution of some host and device code. We also notice that the CUDA stream is designed to concurrently run different kernels on the same GPU device [26]. However, their purpose is different from ours since the computation of one graph needs to exclusively occupy the whole GPU device. In other words, we can only run one kernel at a time.

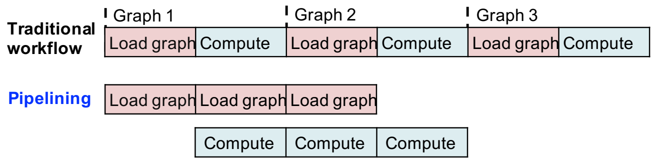


Figure 5: The pipelining in SWARMGRAPH.

3.4. Graph Algorithms

Currently, SWARMGRAPH implements three popular graph algorithms, i.e., all-pairs shortest path (APSP), closeness centrality (CC), and betweenness centrality (BC). These algorithms are vital for graph analytics and are widely used in the analysis of source and binary code [11], social network [27], chemical network [28], and biological network [29]. Not limited, the techniques of SWARMGRAPH can be applied to more graph algorithms, such as (weakly) connected component, strongly connected component, and triangle counting. We will explore these graph algorithms in the future.

All-pairs shortest path (APSP) computes the shortest path from every vertex to all the others. The single-source shortest path (SSSP) algorithm computes the shortest path from a source vertex to all the others. By combining the SSSP from every vertex, we can get the APSP. SWARMGRAPH stores the original graph data (adjacent vertices of each vertex $adj\_v(i)$, and the weight between them $w_{i,j}$) in the global memory. We invoke just one kernel to compute the APSP and assign each block to compute the SSSP from each source vertex to others. After computation, SWARMGRAPH copies the SSSP result from the shared memory to the global memory.

Closeness centrality (CC) measures the importance of every vertex in a graph. The CC value of a vertex is defined as the reciprocal of the sum of the shortest distances from other vertices to it as shown in Equation 1. The more important a vertex, the smaller the sum of the shortest distances from other vertices, and the larger value of CC. If a graph is not strongly connected (a vertex may not be reachable by others), Equation 2 is normally used to calculate the closeness centrality. If a vertex a cannot reach b, the distance is infinite and $\frac{1}{\text{inf}}$ is defined as 0. Its CC is defined as the sum of the reciprocal of the shortest distances.

$$CC(s) = \frac{1}{\sum_{t \in V} d(t,s)} \tag{1}$$

$$CC(s) = \sum_{t \in V} \frac{1}{d(t,s)} \tag{2}$$

CC is implemented with two kernel invocations by SWARMGRAPH. The first kernel computes APSP with multiple blocks, and the second one accumulates CC from every vertex using Equation 2 with one block.

Betweenness centrality (BC) is another centrality measure to understand the importance of the vertices in a graph. It shows the frequency of a vertex acting as an intermediate

TABLE 2: Specifications of protein graphs (Benchmark I).

Group	Graph size	# of graphs	Avg. vertex	Avg. edge
G1	(0, 128)	208	80	292
G2	[128, 256)	452	179	711
G3	[256, 384)	258	316	1270
G4	[384, 512)	130	438	1764
Total	(0, 512)	1048	225	896

TABLE 3: Specifications of one million synthetic graphs (Benchmark II).

Size	32	64	128	256	512	1024
RMAT	57,795	58,881	110,831	104,287	105,780	107,020
Random	42,205	41,119	89,169	95,713	94,220	92,980
Total	100,000	100,000	200,000	200,000	200,000	200,000

vertex along the shortest paths in the whole graph. As defined in Equation 3, the computation of BC includes two steps. First, we compute the single-source BC ($bc_s(t)$), which is the number of shortest paths from vertex s to others passing through vertex t ($\sigma_{s,*}(t)$) over the total number of shortest paths from vertex s to others ($\sigma_{s,*}$). Second, we accumulate the single-source BC value to get the final BC value ($bc(t)$).

$$\begin{aligned} bc_s(t) &= \frac{\sigma_{s,*}(t)}{\sigma_{s,*}}, t \neq s, t, s \in V \\ bc(t) &= \sum_s bc_s(t), t \neq s, t, s \in V \end{aligned} \tag{3}$$

SwarmGraph invokes two kernels, one for single-source BC computation, the other for accumulation. The first kernel utilizes all the blocks and threads to compute the single-source BC starting from all the vertices. The second one recursively accumulates each single-source BC to get the final BC using the Brandes method [27].

4. Experiment

In this experiment, we first compare SwarmGraph with related works on computing one in-memory graph. Later, we scale SwarmGraph to compute one million graphs.

4.1. Graph Benchmarks

We use the following two graph benchmarks for the experiment.

- **Benchmark I: Protein graphs**. We get $1,048$ real-world protein graphs from the protein graph repository [30]. We divide them into four groups based on graph size (vertex count) with 128 as the division step. The specifications are summarized in Table 2. This benchmark is used to evaluate the performance of computing one single graph between SwarmGraph and related works.
- **Benchmark II: One million synthetic graphs**. We generate one million synthetic graphs with both RMAT and random graph generators from X-Stream [31]. The graph sizes (number of vertices) in the synthetic graphs are set to 32, 64, 128, 256, 512, and $1,024$, respectively. For each graph, the average degree is set to a random number between 8 and 16. The specifications are summarized in Table 3. This benchmark is used to evaluate SwarmGraph on scaling to million graphs and multiple GPUs.

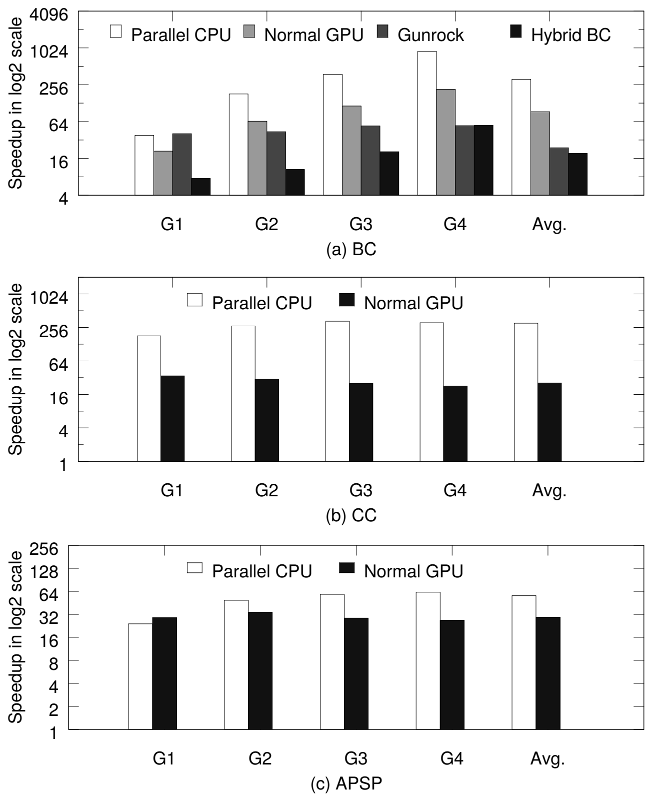


Figure 6: Speedup of SwarmGraph over related works on computing a single graph for (a) BC, (b) CC, and (c) APSP.

4.2. Experiment Setting

The experiments are performed on a server running CentOS 7.6 with two Intel Xeon Gold 6126 CPUs, each of which has 12 cores and enables hyper-threading. The server is equipped with eight Nvidia Titan GPUs, including six Titan V and two Titan XP. The same Nvidia Titan V GPU is used for single GPU test. SwarmGraph is implemented with over 2,000 lines of CUDA and C++ code. SwarmGraph is compiled with GCC 4.8.4 and Nvidia CUDA toolkit 10.0 with optimization level *O3*. The graphs are stored in edge list format, while converted to the required format of compared works later. The result is reported with an average of ten runs.

4.3. Performance Comparison

In this section, we compare SwarmGraph with related works on computing a single graph. We use the protein

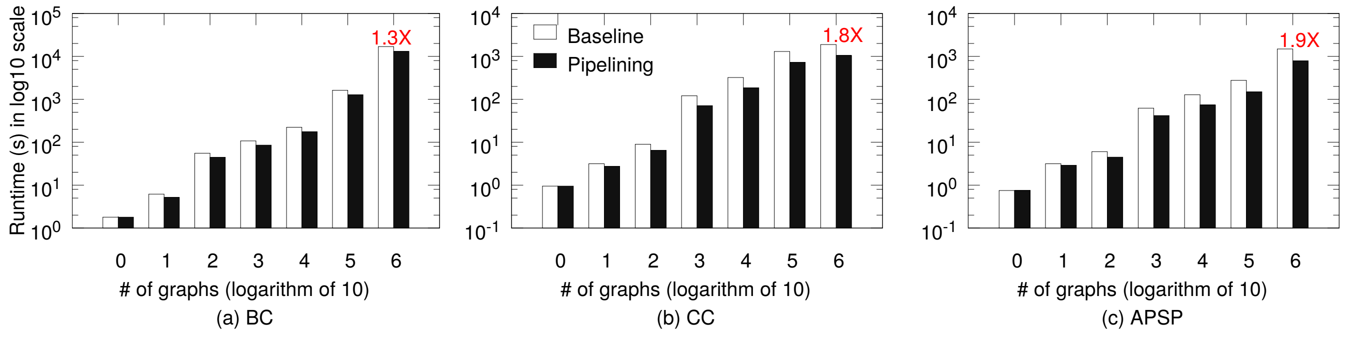


Figure 7: The performance of scaling to one million graphs for (a) BC, (b) CC, and (c) APSP.

graphs (Benchmark I) and report the average computation time of all the graphs in each group to represent the performance.

We compare with four methods, the parallel CPU method, the conventional GPU implementation, a state-of-the-art GPU graph library – Gunrock [5], and a state-of-the-art BC computation on GPU – hybrid BC [32]. Particularly, the parallel CPU method follows the parallel CPU implementation in [33]. During the test, it runs with all the available threads, i.e., 48. The conventional GPU method is implemented on top of a recent breadth-first-search (BFS) work on GPU [14]. For the widely used betweenness centrality computation, we further compare with two state-of-the-art GPU implementations, hybrid BC [32], and Gunrock [5].

Figure 6 presents the speedup of SwarmGraph over related works on computing a single graph, where the techniques for many graphs computation are not used. For betweenness centrality (BC) computation, SwarmGraph is able to achieve $314\times, 93\times, 24\times$, and $19\times$ speedup over the parallel CPU implementation, conventional GPU, Gunrock, and Hybrid BC, respectively, as shown in Figure 6(a). Such a significant speedup is attributed to two reasons, replacing global memory with the fast shared memory, and reducing the number of kernel invocations. SwarmGraph gets the smallest speedup on G1 because hybrid BC benefits from its smart sampling technique. SwarmGraph achieves the largest speedup on graph group G4. With the increase in graph size, the computation time of the related works increases dramatically. That is, the average runtime from G1 to G4 increases $29\times, 13\times, 1.7\times$, and $9\times$, respectively for CPU, conventional GPU, Gunrock, and Hybrid BC. However, SwarmGraph only increases $1.2\times$ because the frequently accessed data are stored in the shared memory which is not obviously affected by the data size.

On average, SwarmGraph is able to achieve $306\times$ and $26\times$ speedup over the parallel CPU and GPU implementation for CC shown in Figure 6(b). Further, SwarmGraph can get $56\times$ and $29\times$ speedup over the parallel CPU and GPU implementation for APSP as shown in Figure 6(c). Interestingly, the parallel CPU implementation is faster than the conventional GPU on graph group G1, because the conventional GPU implementation invokes the kernel multiple times which introduces considerable communication penalties. On the other side, the CPU's large last-level cache (LLC) is able to cache the small graph (less than 128) which boosts the CPU performance.

4.4. Scale to Million Graphs

In this section, we evaluate the performance of SwarmGraph when the graph count reaches one million scales. We use the one million synthetic graphs (Benchmark II).

The baseline method is SwarmGraph without pipelining. We evaluate the performance when the number of graphs scales from one to one million. For each experiment with graph count $|g|$, we randomly select $|g|$ graphs from the one million graphs and follow the graph size distribution shown in Table 3.

Figure 7 presents the total runtime with different number of graphs for the three algorithms. One can see that for the one million graph computation, the pipelining technique is able to bring $1.9\times$, $1.8\times$, and $1.3\times$ speedup over the baseline for APSP, CC, and BC, respectively. Such a pipelining technique is able to get close to the upper bound speedup for two-way pipelining, i.e., $2\times$. The pipelining technique is able to achieve the best speedup on the APSP algorithm because the graph loading time and APSP computation time are close to each other on average. For smaller graphs, the graph loading takes more time than APSP computation, while for larger graphs, the APSP computation takes more time. The speedup drops a little bit (from $1.9\times$ to $1.8\times$) for CC because its computation time increases. While the BC

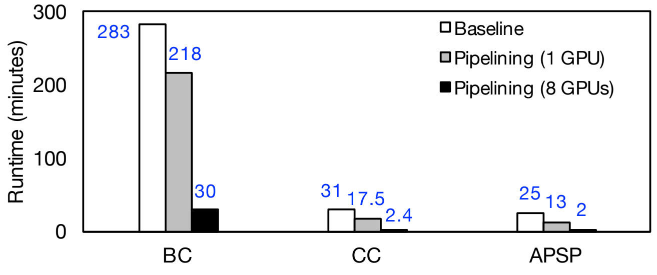


Figure 8: Accumulated runtime (minutes) for one million graphs.

computation takes more time but it is still in the same order of magnitude with graph loading time.

Figure 8 presents the total runtime of one million graphs. The baseline method takes 283 minutes for BC, while the pipelining method takes 218 minutes. For the CC algorithm, the pipelining method is able to reduce the runtime from 25 minutes to 13 minutes. For the APSP algorithm, the pipelining method reduces the time from 31 minutes to 17.5 minutes. SwarmGraph with 8 GPUs is able to significantly reduce the total runtime. For the most time-consuming BC algorithm, we are able to reduce the runtime from more than 3 hours (218 minutes) to just 30 minutes. SwarmGraph also reduces the CC computation time from 13 to 2 minutes, and the APSP time from 17.5 to 2.4 minutes.

5. Related Work

This section discusses the related works on graph computation and the closely related linear algebra and matrix computation.

Matrix computation. Small and medium-size problems are common in matrix computation applications [34], [35], [36], while a graph can be represented as a matrix. Motivated by the fact that current numerical linear algebra libraries are mainly focusing on the large matrix, Anderson *et al.* devise a linear algebra computation framework on GPU specifically designed for the small matrix [34]. They distribute the independent problems into multiple GPU streaming processors and compute them in the streaming processor's register files completely. They have demonstrated on four linear algorithms, Gauss-Jordan elimination, LU factorization, QR factorization, and least squares. Later, motivated by the need for bulk computation on a large number of small matrices in control systems, Tokura *et al.* narrow the matrix computation to eigenvalue computation [35]. They design an efficient GPU method for the small matrices by using GPU-specific optimizations, including assignment strategy of GPU thread to matrices and memory arrangement. Further, Heinecke *et al.* try to improve small matrix multiplications from the point of optimized code generation [36]. Particularly, they design a code generator for the x86 vector instruction set. The generator is able to create code without the auto-tuning phase and only build the required kernel by using a just-in-time compilation technique.

Graph computation. Similar to matrix computation, small and medium-size graphs are common [37], [38]. Targeting continuously generated small graphs, Bleco *et al.* investigates the graph query problem on a large number of small graphs stored in a database [37]. The graph query is a subgraph matching problem. Their major innovations are on the database side, including column-oriented storage, efficient indexing mechanism, and the materialized graph views of different types. Along this line, Pal *et al.* design a new indexing technique for subgraph matching and approximate graph matching queries on a large database of small and medium-sized graphs [38]. The new indexing technique is boosted by building a signature for every graph, which is used for fast searching.

Gunrock is a high-performance graph processing library on GPU [5]. It designs a novel data-centric abstraction focusing on a vertex or edge frontier. It allows a developer to quickly develop new graph primitives with small code size and few GPU programming knowledge. Observing that betweenness centrality computation on GPU is inefficient, Mclaughlin *et al.* design a scalable and efficient implementation [32]. Starting from a sampling technique to approximately predict the structure of the analyzed graph, they will figure out how the size of the workload changes across iterations. Then, they will use a hybrid of vertex and edge parallel techniques to compute. Breadth-first-search is a key graph traversal algorithm used in SwarmGraph. Motivated by the wasted thread scheduling and workload imbalance in current GPU implementation, Liu *et al.* design Enterprise to fast compute BFS on GPU [14]. They design a streamline GPU threads scheduling technique which is achieved by efficiently generating the frontier queue. They balance the workload by dividing the frontiers (based on out degree) into several and conditionally assign them to the thread, warp, thread block, and grid. Further, they optimize the usage of the direction-optimizing technique [39] on GPU. Groute is an asynchronous programming model for graph computation on multi-GPU [40]. SwarmGraph is different from such regular GPU computation frameworks by focusing on computing the large number of small graphs motivated by real applications.

6. Discussion

This section discusses graph size and the extension to more algorithms.

Graph size. SwarmGraph is mainly designed for the massive but small size in-memory graphs. A graph that cannot fit into the shared memory will be computed by the conventional GPU implementation. However, we have seen the increase of shared memory size in recent GPUs, e.g., Nvidia Tesla V100 can be configured to have up to 96KB (doubles the size in K40) shared memory [41]. In the future, we plan to explore the ways to increase the supported graph size by integrating with graph partition methods. Also, the newly introduced cooperative groups provide possibilities of controllable and potential more efficient synchronizations [16], we will explore such opportunities for SwarmGraph in the future.

Extension. SwarmGraph is a framework that can be easily extended to support other graph algorithms and applications. The techniques for many graphs computation, i.e., pipelining and dynamic scheduler, are generic to any graph algorithms. The memory-aware data placement technique can also be generic as long as the metadata of the input graph can fit into the shared memory. The kernel invocation reduction (KIR) technique is limited to the algorithms that can be divided and conquered, i.e., PageRank, node embedding generation algorithm, other centrality algorithms. We will add such algorithms to SwarmGraph in the future.

7. Conclusion

This paper presents SwarmGraph, a fast GPU-based computation framework for million-scale in-memory graphs. SwarmGraph carefully places different types of data in different memory components and reduces the expensive kernel invocations. For many graphs, SwarmGraph leverages a pipelining technique and scales to multi-GPUs. In a nutshell, SwarmGraph is able to outperform the state-of-the-art GPU frameworks $19\times$ for betweenness centrality, $29\times$ for the all-pairs shortest path, and $26\times$ for closeness centrality.

Acknowledgement

The authors would like to thank the anonymous HPCC'20 reviewers for their valuable suggestions. This work was supported in part by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774.

References

[1] Dipanjan Sengupta and Shuaiwen Leon Song. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In *International Supercomputing Conference*, 2017.

[2] Benjamin Bowman, Craig Laprade, Yuede Ji, and H. Howie Huang. Detecting lateral movement in enterprise computer networks with unsupervised graph ai. In *Proceedings of RAID*, 2020.

[3] Yuede Ji, Benjamin Bowman, and H. Howie Huang. Securing malware cognitive systems against adversarial attacks. In *International Conference on Cognitive Computing (ICCC)*. IEEE, 2019.

[4] Yuede Ji, Yukun He, Xinyang Jiang, Jian Cao, and Qiang Li. Combating the evasion mechanisms of social bots. *Computers & Security*, 2016.

[5] Yangzihao Wang and et al. Gunrock: A high-performance graph processing library on the gpu. In *PPoPP*, 2016.

[6] Yuede Ji and H. Howie Huang. Aquila: Adaptive parallel computation of graph connectivity queries. In *Proceedings of HPDC*, 2020.

[7] Yuede Ji, Hang Liu, and H. Howie Huang. ispan: Parallel identification of strongly connected components with spanning trees. In *Proceedings of SC*. IEEE, 2018.

[8] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016.

[9] Nikil Wale and et al. Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowledge and Information Systems*, 2008.

[10] Yunjie Zhao and et al. Improvements of the hierarchical approach for predicting rna tertiary structure. *Journal of Biomolecular Structure and Dynamics*, 2011.

[11] Qian Feng and et al. Scalable graph-based bug search for firmware images. In *Proceedings of CCS*, 2016.

[12] Xiaojun Xu and et al. Neural network-based graph embedding for cross-platform binary code similarity detection. *Proceedings of CCS*, 2017.

[13] Yuede Ji, Lei Cui, and H. Howie Huang. Buggraph: Differentiating source-binary code similarity with graph triplet-loss network. In *16th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2021.

[14] Hang Liu and H Howie Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of SC*, 2015.

[15] et al. Nodehi Sabet. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *Proceedings of ASPLOS*, 2018.

[16] Mark Harris. Cuda 9 features revealed: Volta, cooperative groups and more, 2017.

[17] Yuede Ji, Hang Liu, and H. Howie Huang. Swarmgraph: Analyzing large-scale in-memory graphs on gpus. Technical report, http://howie.seas.gwu.edu/publications/swarmgraph.pdf, 2020.

[18] Nvidia corporation: Cuda c programming guide. 2018.

[19] Ang Li and et al. Warp-consolidation: A novel execution model for gpus. In *Proceedings of ICS*, 2018.

[20] Scott Beamer and et al. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of IISWC*, 2015.

[21] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *Proceedings of SC*, 2014.

[22] Hang Liu and H Howie Huang. Simd-x: Programming and processing of graph algorithms on gpus. In *Proceedings of USENIX ATC*, 2019.

[23] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of KDD*, 2016.

[24] Bryan Perozzi and et al. Deepwalk: Online learning of social representations. In *Proceedings of KDD*, 2014.

[25] Jian Tang and et al. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077, 2015.

[26] Mark Harris. Gpu pro tip: Cuda 7 streams simplify concurrency, 2015.

[27] Ulrik Brandes. A faster algorithm for betweenness centrality. Taylor & Francis, 2001.

[28] Peng Zhao and et al. On the application of betweenness centrality in chemical network analysis: Computational diagnostics and model reduction. *Combustion and Flame*, 2015.

[29] Dirk Koschützki and et al. Centrality analysis methods for biological networks and their application to gene regulatory networks. *Gene regulation and systems biology*, 2008.

[30] Ina Koch and Tim Schfer. Ptgl – the protein topology graph library, 2018.

[31] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of SOSP*, 2013.

[32] Adam McLaughlin and David A Bader. Scalable and high performance betweenness centrality on the gpu. In *Proceedings of SC*, 2014.

[33] Duane Merrill and et al. Scalable gpu graph traversal. In *Proceedings of PPoPP*, 2012.

[34] Michael J. Anderson and et al. A predictive model for solving small linear algebra problems in GPU registers. In *Proceedings of IPDPS*, 2012.

[35] Hiroki Tokura and et al. Gpu-accelerated bulk computation of the eigenvalue problem for many small real non-symmetric matrices. In *CANDAR*, 2016.

[36] Alexander Heinecke and et al. Libxsmm: accelerating small matrix multiplications by runtime code generation. In *Proceedings of SC 2016*.

[37] Dritan Bleco and Yannis Kotidis. Graph analytics on massive collections of small graphs. In *Proceedings of EDBT*, 2014.

[38] Dipali Pal and et al. Fast processing of graph queries on a large database of small and medium-sized data graphs. *JCSS*, 2016.

[39] Scott Beamer and et al. Direction-optimizing breadth-first search. *Scientific Programming*, 2013.

[40] Tal Ben-Nun and et al. Groute: An asynchronous multi-gpu programming model for irregular computations. In *Proceedings of PPoPP*, 2017.

[41] Tesla NVIDIA. V100 gpu architecture, 2017.