CUDA Acceleration for Community Detection using Label Propagation Algorithm

Zhongbo Zhu
Nachuan Wang
Zheyuan Zhang
Xiwei Wang
{zhongbo2,nachuan3,zheyuan5,xiwei2}@illinois.edu
University of Illinois at Urbana-Champaign
Urbana, IL, USA

ABSTRACT

In this report we present several ways to accelerate the Community Detection using Label Propagation (CDLP) Algorithm in CUDA GPU. The main idea is to use hash tables to reduce the complexity of counting neighbors, then based on this idea, we propose different methods to improve the program's performance, including shared memory, dynamical parallelism and warp-level atomic operations. After testing it with some benchmark graphs [3], we found that some methods work while others don't, so we do detailed profiling for each method and discuss the trade-off between them.

1 MOTIVATION

Community detection is a basic problem in data science. It is of special interests for social network. For example, Twitter or Facebook may would like to push the same contents to people who belong to the same community. Also, in these use cases, the network is usually very large so it's important to find a efficient implementation of the algorithm. Since the computation on each node in the graph are independent, it is natural to consider to parallize it on GPU to improve the performance.

2 BACKGROUND

2.1 LDBC Graphalytics Benchmark

LDBC Graphalytics Benchmark is a benchmark platform for multiple graph analysis platform. In our case, the implementation is based on the GraphBLAS platform. The benchmark platform provides a full benchmark workflow from the raw graph files to the final benchmark result. The benchmark driver starts with minimizing and relabeling the raw graph files with DucKDB, then converts the files into binary format and load them into the memory. The algorithm is run on the data in the memory. The original algorithm is replaced with our implementation in CUDA. After the algorithm finishes, the output will be in a `GrB_Vector` and inverse-relabled in C++. Finally the result will be serialized into binary format and validated by DuckDB. Note that the algorithm and the input are deterministic, so the result can be validated by comparing with the pre-computed result.

To fully support LDBC standard benchmark framework for local testing and RAI server, we have been using a self-customized docker image, which contains the software dependencies needed for the benchmark framework, CUDA, and several large datasets. Our team test and debug the algorithm locally, but the test data we gathered in this report is from the RAI server.

We open source the docker building process at https://github.com/zzb66666666x/cuda-ldbc-graph-analytics-docker. The code we use for the rest of the report can be accessed at https://github.com/tomzzy1/ldbc_graphalytics_platforms_graphblas.

2.2 CDLP Algorithm

Algorithm 1 CDLP Algorithm

```
$Input : graph\ G = (V, E), integer\ max\_iterations$
$Output : array\ labels\ storing\ vertex\ communities$
for all $v \in V$ do
    $labels[v] \leftarrow v$
end for
for $1, \ldots, max\_iterations$ do
    for all $v \in V$ do
        $C \leftarrow create\_histogram()$
        for all $u \in N_{in}(v)$ do
            $C.add(labels[u])$
        end for
        for all $u \in N_{out}(v)$ do
            $C.add(labels[u])$
        end for
        $freq \leftarrow C.get\_maximum\_frequency()$ ▷ Find maximum frequency of labels
        $candidates \leftarrow C.get\_labels\_for\_frequency(freq)$ ▷ Find labels with max frequency
        $new\_labels[v] \leftarrow min(candidates)$ ▷ Find labels with max frequency
    end for
    $labels \leftarrow new\_labels$
end for
```

The algorithm is from the benchmark specification of LDBC[4]. To break the tie of maximum frequency, the algorithm choose the candidate with the least label, so that the algorithm is deterministic. For undirected graph, the in and out neighbor set are the same, so only out set needs to be checked.

2.3 Data Structure

In the LDBC competition, the data structure for graph can have several options. We choose the version based on GraphBLAS. GraphBLAS stands for Graph Basic Linear Algebra Subprograms, which is an API specification for representing graph using linear algebra [1].

More specifically, the graph is represented in the adjacency matrix and stored in the Compressed Sparse Column (CSC) or compressed sparse row (CSR) format. Take the CSR format as an example, for a given node, its neighbor's can be found by first finding the row pointer with its index, then iterating the row array to get the index of its neighbor, and finally get the neighbor's label with the index. The GraphBLAS library has provided functions for exporting the `GrB_Matrix` data structure to CSR format C arrays. Then we can use `cudaMemcpy` to pass it into GPU memory for further processing.

2.4 Graph Datasets

In this section, we present our choice of graph datasets. Table 1 shows the set of graohs we use to benchmark the algorithms. The datasets are open to public via link https://ldbcouncil.org/benchmarks/graphalytics/. We only include large enough graphs in this tabel for performance benchmark and ignore those small example graphs.

Table 1: Statistics of Graph Datasets

Name	# vertices	# edges	tar file size (MB)
datagen-7_5-fb	633432	34185747	162
datagen-7_6-fb	754147	42162988	199
datagen-7_7-zf	13180508	32791267	434
datagen-7_8-zf	16521886	41025255	544
datagen-7_9-fb	1387587	85670523	401

3 DESIGN OVERVIEW

Figure 1 shows all the designs we are about to discuss. We mainly designed four types of GPU kernels plus several small optimizations. We will discuss the performance of each design and finally come up with a fused version combination all the best practices. Table 2 summarizes the methods we tried. We use DyPa to represent Dynamic Parallelism.

Table 2: Algorithms Tried

Name	Thread Layout	Usage
GPU Baseline	Fig. 1a	General
GPU Hashing	Fig. 1a	General
DyPa	Fig. 1b	General
Dypa + Shared Mem	Fig. 1c	General
One-warp-per-node	Fig. 1d	General
First Iteration Optim.	Fig. 1a wo. histogram	Only First Iter.
Optimization Fusion	N/A	Fine Tuned

4 BASELINE

4.1 CPU Baseline

In the framework provided by LDBC Benchmark [3], a reference CPU version of CDLP algorithm is implemented using the LAGraph library. The LAGraph library is a project that aims at collecting efficient graph algorithm implementations built on top of GraphBLAS[2]. We choose this as our baseline in CPU. In Figure 2, we can see the relative scale of each problem to solve. In our dataset, the iteration count is 10, so the average time for each iteration will be ranging from 1500ms to 1500ms.

4.2 GPU Baseline

Based on the reference algorithm, we implemented a naive GPU version as the baseline. Figure 1a illustrates our design. For each node in a graph, we simply used one thread to count the label frequencies of its neighbors and find the most frequent label. Counters for all labels are stored in a histogram. Since in the worst case each neighbor has a different label, the histogram size should equal to the number of the node's neighbors.

Since different nodes could have different amounts of neighbors, and the actual number could be potentially very large, we cannot allocate a piece of fixed shared memory for each node to store its histogram, instead, all histograms go to global memory, forming a large histogram array, and each thread independently update its own section.

We then measure its performance with the same benchmark framework. To precisely measure the GPU kernel time over iterations, we collect the time consumption for each iteration of CDLP.

Based on Figure3, the speedup is not evident, which is expected. For `datagen-7_9-fb`, the average CPU time per iteration is 3700ms, while the result for the GPU's first iteration reach to 2600ms. The label-counting is memory-bounded and all memory access is global. More importantly, the algorithm we use to build histogram is a double for loop, which has $O(n^2)$ algorithm complexity, where n is the number of neighbors for a node. The optimization of it will be introduced in the next section.

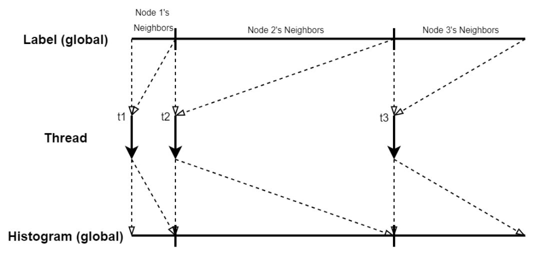
5 OPTIMIZATIONS AND RESULTS

5.1 Algorithm Level Optimizations

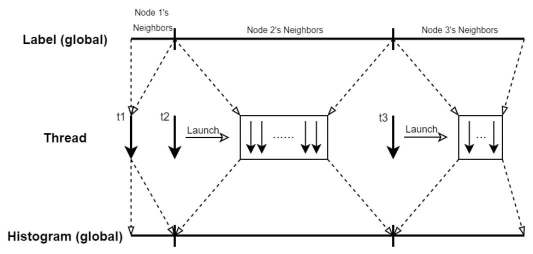
5.1.1 Counting with Hash Table. Since the label is represented as a random integer, in the baseline implementation, we have to search the entire histogram to find the right label to increment, so the complexity of counting one neighbor is $O(n)$, and the complexity of counting all neighbors is hence $O(n^2)$. This is unacceptable especially when we have abundant global memory access. Since the labels of each node's neighbors can be very sparse, it's not acceptable to directly build a direct-indexed array for counting.

To solve this problem, we use a hash function to transform the random integer to the index of the histogram, in the case that two labels are mapped to the same address, we use linear probing to resolve the conflict. In this way, we reduce the complexity of counting one neighbor and all neighbors to $O(1)$ and $O(n)$, separately.

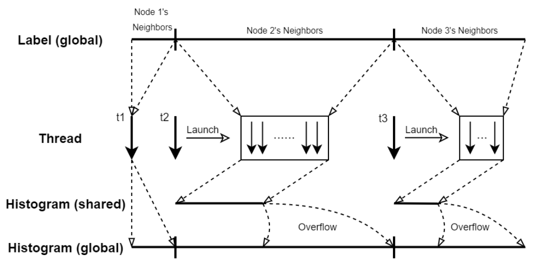
The performance of a hash table could be affected by the rate of hash conflict. One key observation is that for the first iteration, each label only occurs once, so the chance of hash conflicts is at the highest point. However, as the labels converge, the ratio between the set of labels and the hash table size will decrease, leading to a reduction of hash conflicts, which also explains why the time for each iteration drops over iterations in Figure 5. To further reduce hash conflicts, we tuned the size of hash table array to see how it would affect the performance. In our experiments, we found that giving two times the size of neighbor list (the column indices array



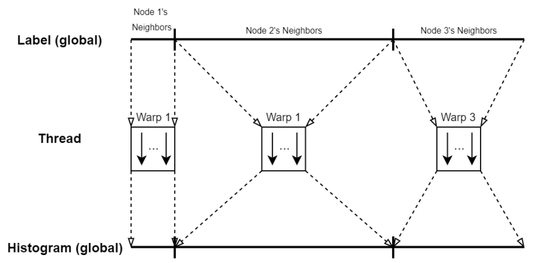
(a) Baseline



(b) Dynamic parallelism



(c) Dynamic parallelism with shared memory



(d) One-warp-per-node

Figure 1: Graphic illustration of different schemes we implemented

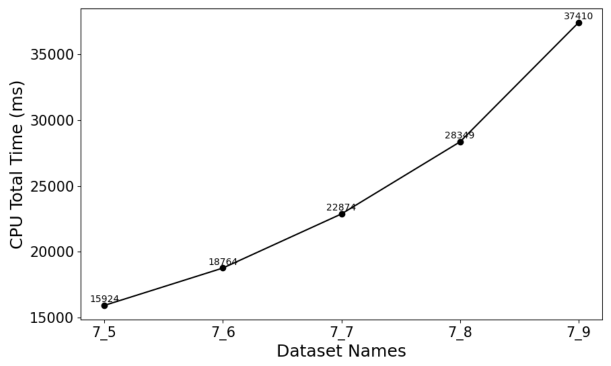


Figure 2: CPU Total Time for Each Dataset

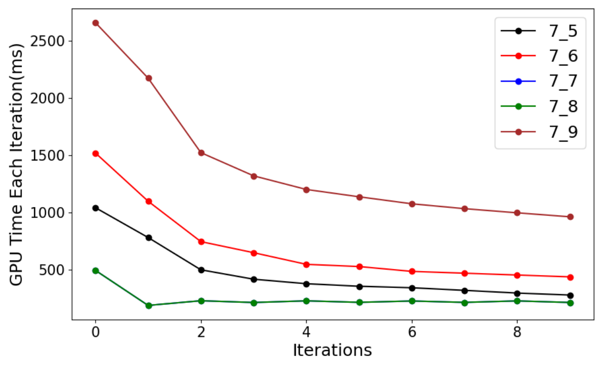


Figure 3: GPU Time for Each Iterations for Baseline

in CSR format) would be a balanced option between performance and memory consumption.

To benchmark this optimization, we first measure the GPU time for each iterations, as shown in Figure 4. Then we disable the maximum iteration count limit and run CDLP until converge for dataset `datagen-7_5-fb` to obserse how GPU time changes over iterations, as shown in Figure 5. The reduction of GPU time in the figure comes from the reduction of hash conflicts like we discussed before. When compared to GPU baseline, the overall performance increase is more than 10 times faster for many iterations and 5 ~10 times faster in terms of total time.

In the following schemes, assume that the hash table is always implemented.

5.2 CUDA-Specific Optimizations

5.2.1 Dynamic Parallelism. Due to the nature of the network community, nodes could have very different amounts of neighbors, so we modify our program to suit this variability. As shown in 1b, we still statically allocate one thread to each node, but now every thread can launch a new kernel whose size is proportional to the amount of neighbors. The new kernel will collaboratively update the histogram.

We use CUDA dynamic parallelism API to achieve this functionality, threads launch kernels by themselves and are only synchronized at the end of each iteration. Considering the overhead of launching dynamic kernels, we set a threshold for the neighbors' number, only nodes with more neighbors than this threshold could launch new kernels.

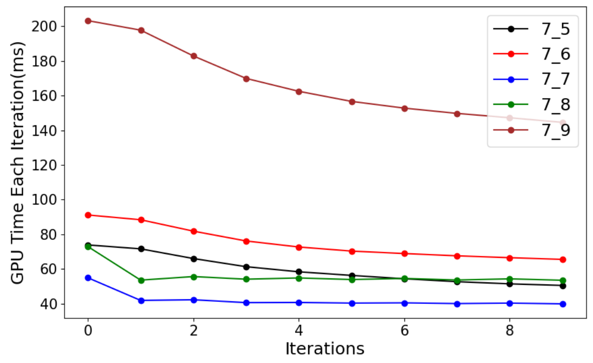


Figure 4: GPU Time for Each Iteration with hash

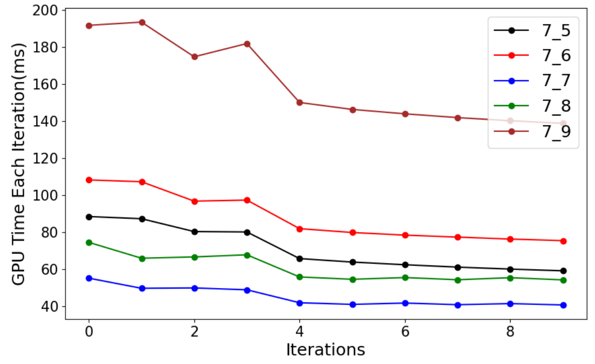


Figure 6: GPU Time for Each Iteration with Dynamic Parallelism

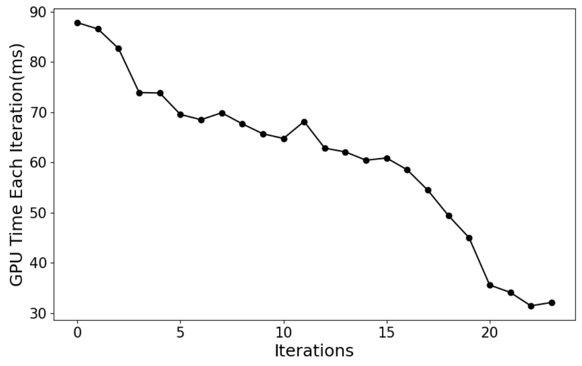


Figure 5: GPU Time for Each Iteration on datagen-7_5-fb

One problem of this method is that now we are using multiple threads to build a single hash table, so it's not possible to achieve lock free hash table operations. The locking strategy we used is that we put a integer called mutex inside of the struct of hash table item, and use this mutex to protect each hash table slot. If labels are hashed to different slots, we can still get a nearly lock free performance. However, the hash conflicts might introduce extra atomic operation overhead. If the set of labels are very small for a node's neighbor list, i.e. the labels converge overtime, all threads will be updating one single slot of hash table, this will also introduce overhead. This trade-off will be further discussed in section 5.3.2. With this problem in mind, we have to carefully tune the parameters to see if the dynamic parallelism method is worthy.

Figure 6 shows the time for each iteration, when compared to Figure 4, the performance increase is not evident. We think this is caused by global memory accesses and the atomic operations performed on shared memory. We will introduce the optimization using shared memory in the next section.

5.2.2 Dynamic Parallelism with Shared Memory. Until now all data still locates in global memory. As we mentioned before, two issues prevent us from using shared memory efficiently: first, different nodes have different amounts of neighbors, which means shared memory should also be dynamically assigned; second, a node could have many neighbors, it is hard to fit all of them into limited shared memory.

The first issue can be solved with dynamic parallelism since we can assign different size of shared memory when launching a new kernel. The second issue is a little tougher since shared memory size is a hardware limitation.

However, recall that the goal of CDLP algorithm is to let nodes with the same community converge to the same label, which indicates that after several iterations (although the actual number depends on specific graphs), a small fraction of labels will occupy a large fraction of nodes. This means a small amount of counters in a histogram will be frequently updated, while other counters are only incremented occasionally. If we can put those counters, theoretically the performance can be improved a lot.

Then the question becomes how to map labels with high frequency to a small piece of shared memory. Since we have implemented the hash table previously, it is intuitive to modify the hash function to concentrate a set of sparse indexes.

Now the solution becomes clear. Based on the design in 5.2.1, we proposed the design shown in Figure 1c. We allocate a piece of shared memory when launching a dynamic kernel, the size of shared memory is proportional to the kernel size. We then use *mod* as hash function to map most of the counters to the shared memory as shown in 7. Again we use linear probing to resolve index conflict, and some counters may overflow to global memory if conflict happens. In addition, a large shared memory can reduce conflict and hence reduce counters that overflow to global memory, so the value X is carefully selected to balance the rate of conflict and limitation in shared memory capacity In practice, we allocate 128 slots of hash tale item per 1200 neighbors and the upper bound is 512 slots.

This design may not demonstrate evident performance improvement at first several iterations since there are still too many labels, but after that, there will only be some hotspots left in the histogram. Most hotspots will be loaded to shared memory by the hash function we choose, so the kernel will primarily work on shared memory and only a few global memory access. However, we also observe

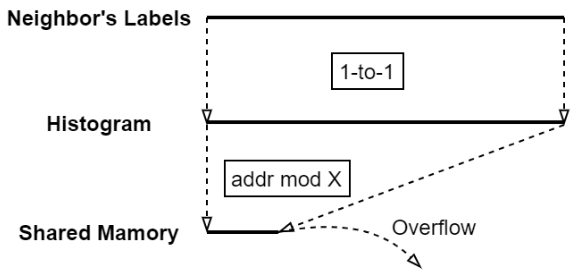


Figure 7: Address mapping

that if the labels converge to only a few, the atomic operations on hash table slots will be serialized. Therefore, the parameter tuning for this method is also a hard problem.

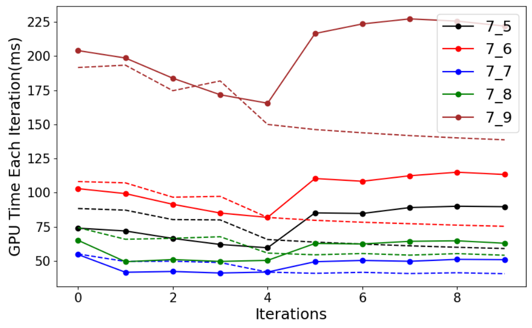


Figure 8: GPU Time for Each Iteration with Dynamic Parallelism Plus Shared Memory

Figure 8 shows the performance of this design with the dotted line represents the dynamic parallelism method without shared memory. We are see the in the first 5 iterations, having a two level hash table is bringing in some performance increase. However, the overall dynamic kernel launches introduce a significant overhead that we didn't expect. When we are doing dynamic parallelism, each child kernel launch comes with a `cudaDeviceSynchronize`, the extra synchronization overhead is also reducing the performance increase we expected.

To conclude the experiments we did on the dynamic parallelism, we have proved that dynamic kernel launches might not be a good idea at least for the datasets we use. However, the two level hash table method can still be utilized. In the next section, we will introduce how we reuse the two level hash table design and use only one warp for each source node.

5.2.3 One-warp-per-node. We didn't observe evident performance improvement from 5.2.2. As discussed before, the main reason is CUDA device has to maintain memory consistency after synchronization, which prevents the kernel from fully pipelining memory access to hide latency. To remove the overhead of launching kernels, we choose to use exactly one warp for building the hash table. The rest of the design follows the section 5.2.2, which means that we still use shared memory to build a two level hash table. Since the atomic operations done by threads in a warp are optimized by hardware, and if these atomic operations are updating a shared memory location, the memory access overhead will also be small. For this design, we want to compare it with our vanilla hash table method to see if we can get performance increase if certain iterations.

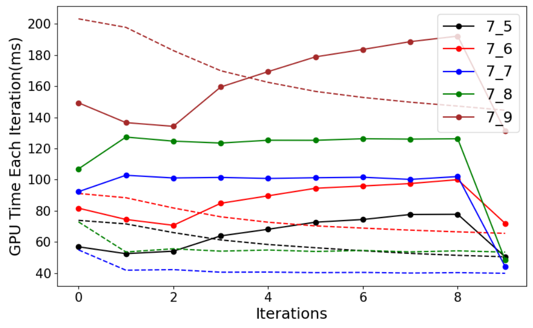


Figure 9: GPU Time for Each Iteration with One Warp Per Node

Figure 9 shows the performance of this design compared with the performance of using vanilla hash table. The dotted lines represent the vanilla hash table method. We can see that for first several iterations, the new method brings some performance increase for some graphs: `datagen-7_5-fb`, `datagen-7_6-fb` and `datagen-7_9-fb`, but it always performs worse for other graphs: `datagen-7_7-zf` and `datagen-7_8-zf`. We believe that the times of those `*zf` graphs can not be improved is because there are too many vertices and the number of edges doesn't increase (as shown in Table 1), so allocating one warp per node has a huge overhead. In addition, for the last iteration, we use the vanilla hash table method, so the time drops. Interestingly, the `datagen-7_9-fb` specifically prefer this optimization. From the difference between the line with label 7_9 and the dotted line with the same color, we can see that the first four iterations receive a big performance increase.

With the performance increase over some of the graphs, we can specifically tune a threshold value for iterations, which means that we can use the *one-warp-per-node* method for the first few iterations and then switch to the vanilla hash table method.

5.3 Fusing Different Schemes

In this section, we combine several best practices together to get the best performance of CDLP.

5.3.1 First Iteration Optimization. For the first iteration of CDLP, since the label count will always be 1, there is no need to use hash table. So each neighbor can simply find its minimum neighbor and set the label for next iteration. This optimization is free meal for all the strategies.

5.3.2 Trade-off between Atomicity and Hash Conflicts. Based on our result in section 5.2.3, the kernel time increase over iterations. That's because of the trade-off between atomicity and hash conflicts. We have discussed that due to the quick convergence of labels, the hash conflicts will be greatly reduced. However, if we want parallel threads to build up the hash table, lower hash conflicts rate mean higher collision for atomic operations on each hash table slot. As label converges, all the parallel threads are trying to update the same slot of hash table, so we tune a threshold value to switch from the kernel in 5.2.3 to vanilla hash table solution in 5.1.1.

5.3.3 Evaluations. With Figure 10, we show the performance comparison between fusion and vanilla hash method (dotted lines). We can see a huge performance increase for the first iteration. With our optimization in 5.2.3, the first few iterations also get optimized. For example, on dataset `datagen-7_5-fb`, the threshold iteration value to switch from *one-warp-per-node* to vanilla hash method is the 3, which means that iteration 1, 2 all benefit from this optimization.

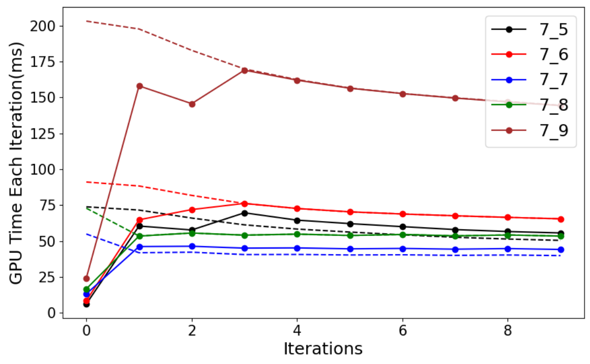


Figure 10: GPU Time for Each Iteration with Optim. Fusion

6 CONCLUSION

In this report, we propose several strategies for accelerating the CDLP algorithm with CUDA. First, we build a general-purpose Docker image containing the LDBC benchmark framework and CUDA dependencies for running benchmarks on RAI and debugging on local machines. We then develop a baseline algorithm using CUDA. Through performance bottleneck analysis, we propose the linear hashing method for executing the counting step in the CDLP algorithm, resulting in a significant performance increase. To optimize further, we explore the dynamic parallelism approach to address the load imbalance issue among threads. However, the performance benchmark did not meet expectations even with our two-level hash table using shared memory. We conducted a trade-off analysis of the dynamic parallelism approach and developed a one-warp-per-node method to eliminate child kernel launch overhead and atomic operation overhead. Finally, we combined multiple optimizations to achieve the best performance improvement. For future work, we can benchmark our implementation using even larger graphs and conduct more fine-grained profiling to identify performance bottlenecks and attempt to surpass our current best version.

REFERENCES

[1] Jeremy Kepner, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, Jose Moreira, Peter Aaltonen, David Bader, Aydin Buluc, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, and Andrew Lumsdaine. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. https://doi.org/10.1109/hpec.2016.7761646

[2] Tim Mattson, Timothy A. Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, Jose Moreira, and Carl Yang. 2019. LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 276–284. https://doi.org/10.1109/IPDPSW.2019.00053

[3] Gabor Szarnyas. 2023. LDBC Graphalytics Benchmark. https://github.com/ldbc/ldbc_graphalytics

[4] Gabor Szarnyas. 2023. LDBC Graphalytics Benchmark Specification. https://arxiv.org/pdf/2011.15028.pdf