Dynamic Graph Data Structure for GPU

Baijia Ye
New York University
New York, USA
by2352@nyu.edu

Changyue Su
New York University
New York, USA
cs7483@nyu.edu

Shidong Zhang
New York University
New York, USA
sz4570@nyu.edu

Ziyue Feng
New York University
New York, USA
zf2182@nyu.edu

Abstract

Graphs serve as a versatile and powerful data structure, offering an expressive representation for modelling relationships and connections in diverse fields such as social networks, simulations, and bioinformatics. The increasing complexity of real-world graph problems together with the inherent connection between graphs and sparse matrices show the importance of building and analyzing graph data structures on GPU. Leveraging sparse representations in parallel patterns can significantly reduce memory and energy requirements for working on graph adjacency matrices. Traditional graph structures relying on CPU-based systems face limitations in the era of massively parallel processors like GPUs. This has also sparked interest in parallel graph analytics research on GPUs to address computational resource and memory bandwidth constraints. This paper introduces a parallelized graph data structure library for GPUs, aiming for exceptional performance through advanced parallel programming models. Key contributions include implementing GPU-specific graph operations, optimizing memory efficiency, load balancing, workload management, and conducting comprehensive performance evaluations against CPU and GPU implementations.

Keywords: Data Structure, Graph, CUDA

ACM Reference Format:
Baijia Ye, Changyue Su, Shidong Zhang, and Ziyue Feng. 2023. Dynamic Graph Data Structure for GPU. In *Proceedings of Data Structure Library for GPUs: Graphs (GPUs Architecture and Programming)*. ACM, New York, NY, USA, 7 pages. https://github.com/fzyyy0601/gpu-final-project-ds

GPUs Architecture and Programming, Dec 2023, New York, NY, USA

https://github.com/fzyyy0601/gpu-final-project-ds

1 Introduction

Graphs is a versatile and powerful data structure that provides an expressive way to represent and model relationships and connections between entities. By representing entities as vertices and relations as edges, many real-world problems can be viewed as graph problems. Some examples are social networks, physical simulations, driving directions services, bioinformatics, etc. [11]. With the development in these areas, there has been a notable increase in the size and computational complexity of challenges in practical graph problems.

In fact, graphs have an inherent connection to sparse matrices. In real world scenarios, the modeled graphs from problems are more likely to be sparsely connected [8]. Many graph tasks can be transformed into sparse matrix operations. Utilizing a sparse representation technique in parallel patterns can result in substantial reductions in memory capacity, memory bandwidth, time, and energy for working on the graph adjacency matrix.

Current graph data structures commonly rely on CPU-based systems and serial programming models, which are not well-suited for a massively parallel processor like the GPU [10]. This has led to a heightened interest in the exploration of parallel graph analytics research on GPU, aiming to address the limitations in computational resources and memory bandwidth inherent in single processors.

In this paper, we present a parallelized graph data structure library for Graphics Processing Units (GPUs), which achieves exceptional performance in the computation of graph analytics through advanced and parallel programming models. We also aim to achieve a balance between performance and programmability. From a performance perspective, we construct lower-level optimizations on GPU-version graphs by leveraging cutting-edge fundamental data parallel primitives. From a programmability perspective, we create a high-level programming model for graph data structure, ensuring flexibility, and aligning with the GPU's parallel execution.

Our main contributions are as follows:
1) We build a library that implements and parallel graph data structure and operations for GPU applications.

2) We illustrate various optimization strategies tailored specifically for GPUs, focusing on enhancing memory efficiency, load balancing, and workload management. These combined strategies contribute to achieving optimal performance.
3) We provide an extensive experimental assessment of our graph library, presenting performance comparisons against multiple implementations on both CPUs and GPUs.

2 Literature Review

Graphs are fundamental data structures, central to numerous algorithms and mathematical problems, and are crucial for abstracting many of today's scientific challenges. Traditionally, graph data structures were optimized for CPUs, tailored to their specific features. However, with the rise of AI and parallel computing, GPUs have become essential in modern computing. While extensive work has been done on adapting structures like lists, arrays, stacks, and queues for GPUs, graph structures have received less attention, presenting a ripe area for exploration.

Graphs can primarily be represented in three forms: adjacency matrices, adjacency lists, and edge lists. Adjacency matrices are preferred for symmetric or dense undirected graphs, while adjacency lists and edge lists are more suited for large, sparse, or directed graphs. The choice of underlying data structures, like two-dimensional arrays for adjacency matrices or lists and arrays for adjacency lists and edge lists, varies based on specific requirements. Additional structures like hash maps are sometimes used for faster retrieval of edges and vertices. For instance, Parallel BGL, a C++ library, supports distributed adjacency lists and external property maps for parallel graph computations[7]. GraphLab, a parallel machine learning framework, excels in large-scale, real-world problems using asynchronous iterative algorithms [9]. Other researchers have focused on message-passing frameworks for GPU-based graph structures, enhancing memory access by leveraging GPUs' coalesced memory feature[12].

Shifting focus to GPUs, these are designed for large-scale, parallel tasks, offering higher bandwidth and efficient scalability. Various graph data structures have been designed to exploit GPU features. For example, Gunrock adopts a data-centric abstraction for operations on vertices or edges, emphasizing bulk synchronous processing [11]. Dynamic graphs on GPUs, such as those utilizing Slab Hash[2] for underlying hash tables, enable faster searches for specific edges and vertices [3]. VertexAPI2 and nvGRAPH are other high-performance graph analytics libraries, the latter viewing graph analytics through the lens of linear algebra and matrix computations [6].

The Coordinate Format (COO), commonly used in sparse matrix storage on GPUs, offers efficient memory access and flexibility. GPU-oriented graph libraries often implement storage using Compressed Sparse Row (CSR) format [11][1]. An example is cuRnet, an R package for graph traversal on GPUs, which utilizes both COO and CSR for storing graphs for computations[5].

3 Proposed Idea

In this section, we introduce the implementation of our graph data structure library and its individual graph operators. We also provide a brief overview of how these implementations facilitate optimizations. Section 3.1 illustrates our graph storage with COO format, and Section 3.2 demonstrates our graph operations algorithms.

3.1 Memory Management

Since many real-world graphs are sparsely connected, we focus mainly on handling graphs with sparse adjacency matrices. For better memory management, sparse matrices are usually maintained in a format or representation that circumvents the storage of zero elements. Here we implement the Coordinate Format (COO).

In COO format, a non-zero element is saved along with its corresponding column and row indices. We use a *col_index* array, a *row_index* array, and a *value* array to store representative column/row positions and its non-zero values. Both the *col_index* and *row_index* arrays are utilized alongside the data array. The COO format enables us to check any element in the storage and find its original position in the sparse matrix. In this project, we use the value array to store the weights of all edges. That means an index i shows that there is an edge from $row\_index[i]$ to $col\_index[i]$ with weight $value[i]$. Figure 1 is an example of the COO format.

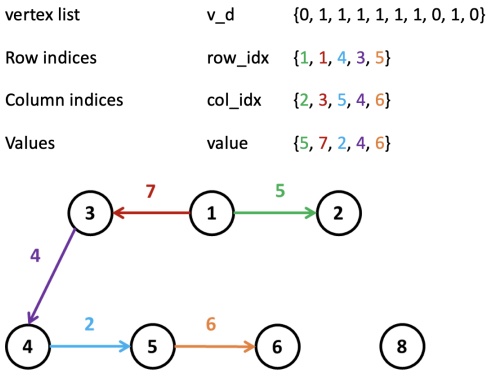

Figure 1. This is one of the graphs we used to test the correctness of our code. Number of vertices: $v\_num = 7$; Number of edges: $e\_num = 5$; Max number of edges: $MAX = 10$.

To keep track of the existence of each vertex, we maintain an array with a length of the maximum number of vertices. Each index in this array corresponds to the unique ID of the vertex. If a vertex exists, the vertex array has the value 1 in this vertex's ID index. Otherwise, we turn the value into 0.

We also use a queue to store the deleted positions. To delete an edge, we mark its position in row and column arrays

as -1, meaning that this position is empty. Next time, if we want to insert a new edge, we can pop a front position from the deleted queue and store the new edge in this position. The realization of the deleted queue on GPU is using an array with two extra locations representing the head and the tail. We keep the queue up-to-date while inserting and deleting vertices or edges.

3.2 Graph Operations

In this section, we show our parallelized graph operations algorithms in detail. These algorithms aim at parallelizing the graph computations on GPU for achieving speedups and saving resources.

A. Initialization

For initialization, two key procedures are necessary. Firstly, to initialize the three arrays of COO format row_idx, col_idx, $value$, this is accomplished by copying the arrays from the CPU to the GPU using cudaMemcpy. Secondly, the v_d array needs to be initialized based on an input vertex list v_list_d. In this process, each vertex specified in v_list_d should be assigned a value of 1 in the v_d array. This is achieved by launching a kernel where, for each vertex value v in v_list_d, the corresponding element in v_d indexed by v is set to 1.

Algorithm 1 Initialize V

```
1: function Initialize_V(v_d, v_list_d, n)
2:     index = blockIdx.x × blockDim.x + threadIdx.x
3:     stride = gridDim.x × blockDim.x
4:     while index < n do
5:         v = v_list_d[index]
6:         v_d[v] = 1
7:         index = index + stride
8:     end while
9: end function
```

B. Get In/Out Degrees

This function is to get the in or out degrees for a vertex. For a specific vertex v, we launch a kernel to traverse the row array or column array in parallel. Every time we encounter v in the row array, we know that there is an out-neighbor of v. Every time we encounter v in the column array, we know that there is an in-neighbor of v. Therefore, we loop through either the row array or the column array to get out-degree or in-degree. Here we put the algorithms of the kernel functions for calculating the in-degree and out-degree of a node. Note that, before launching the kernel, *count* is initialized to be 0, and we use its address as the first argument for *atomicAdd*. After all the threads finish their work, *count* would be the in-degree or out-degree of v.

Algorithm 2 Get In Degree

```
1: function GET_IN_DEGREE(v, col_idx, count)
2:     index = blockIdx.x × blockDim.x + threadIdx.x
3:     stride = gridDim.x × blockDim.x
4:     while index < n do
5:         if col_idx[index] == v then
6:             atomicAdd(count, 1)
7:         end if
8:         index+ = stride
9:     end while
10: end function
```

Algorithm 3 Get Out Degree

```
1: function GET_IN_DEGREE(v, row_idx, count)
2:     index = blockIdx.x × blockDim.x + threadIdx.x
3:     stride = gridDim.x × blockDim.x
4:     while index < n do
5:         if row_idx[index] == v then
6:             atomicAdd(&count, 1)
7:         end if
8:         index+ = stride
9:     end while
10: end function
```

C. Get Destination/Source Vertex

The function is to get the destination vertex of a given vertex. For a given vertex *i*, we first call the function get out degrees to get the exact numbers of i's destination vertex, then, we initialize a piece of memory with the number of out degrees as its size. After that, in the kernel function, each thread iterates through the *row_idx* to search for vertex i's existence. If *row_idx[j]* equals i, then the corresponding vertex at position j is one of the destination vertex that we found. *col_idx[j]* then is added to the final results. One point here is that, since multiple threads are sharing the index of the result list, which is *cnt* in the below algorithm, for data consistency we use atomic add to safely increment the pointer. The process of getting the source vertex of a given vertex is similar to getting the destination vertex of a given vertex. The only difference is that, instead of searching in *row_idx* for a given vertex, we search in *col_idx*. If we find that the vertex at position *j* in *col_idx* equals *i*, we add *row_idx[j]* to the final results.

Algorithm 4 Get Destination of Vertex

```
1: function GET_DES_OF_VERTEX(row_idx, col_idx, x, n, count, res)
2:     index = blockIdx.x × blockDim.x + threadIdx.x
3:     stride = gridDim.x × blockDim.x
4:     while index < n do
5:         if row_idx[index] == x then
6:             idx = atomicAdd(count, 1)
7:             res[idx] = col_idx[index]
8:         end if
9:         index+ = stride
10:    end while
11: end function
```

Algorithm 5 Get Source of Vertex

```
1: function GET_SRC_OF_VERTEX(row_idx, col_idx, x, n, count, res)
2:     index = blockIdx.x × blockDim.x + threadIdx.x
3:     stride = gridDim.x × blockDim.x
4:     while index < n do
5:         if col_idx[index] == x then
6:             idx = atomicAdd(count, 1)
7:             res[idx] = row_idx[index]
8:         end if
9:         index+ = stride
10:    end while
11: end function
```

D. Insert Vertex/Edge

To insert a vertex into the graph, we only need to check whether the vertex is in the graph first. If not, we launch a kernel with only one thread to set $v\_d[vertex]$ to be 1. Here is the algorithm for the kernel function. Note that we only launch this kernel if the node is not in the graph.

Algorithm 6 Insert Vertex

```
1: function INSERT_VERTEX(v)
2:     v_d[v] = 1
3: end function
```

To insert an edge into the graph, we first make sure that both sides of the graph are already in the graph and that this edge does not exist. Then if the *delete* is empty, just expand the three coordinate lists by 1 and record the edge in the next position. Otherwise, use the first index recorded in *delete* to store this edge, and remove the index from *delete*. Note that operations for modifying *head*, *tail*, and the size of the coo list are done in the host.

Algorithm 7 Insert Edge

```
1: function INSERT_EDGE(row, col, value, delete)
2:     if delete is empty then
3:         index = index of new place in coo list
4:     else
5:         index = delete[head]
6:     end if
7:     row_idx[index] = row
8:     col_idx[index] = col
9:     value[index] = value
10: end function
```

E. Delete Vertex/Edge

The deleting processes for a vertex or an edge are similar. We first need to check that the input entities do exist. Then we pass in the to-be-deleted inputs into the kernel function to update our row_idx, col_idx, $delete$ arrays.

To delete an edge, all we need to do is to find this edge in row_idx and col_idx arrays. Since these two arrays together represent the edges in the graph, there must be a position index i such that $(row\_idx[i] \rightarrow col\_idx[i])$ is the edge we want to delete. In the kernel function, if one thread finds this edge, it will mark this position i with value -1 meaning that this position's edge has been deleted. *deleted* will also push this position i in for future insertion. Then the kernel will end and return a boolean value.

Deleting a vertex, however, requires us to delete all edges containing this vertex. The kernel function will keep executing through the whole row_idx and col_idx arrays. We use *atomicAdd* to count in number of edges we delete during this process.

Algorithm 8 Delete Edge

```
1: function DELETE_EDGE(row, col, delete, row, col)
2:     if row_idx[index] == row
3:       and col_idx[index] == col then
4:         row_idx[index] = −1
5:         col_idx[index] = −1
6:         add index to delete
7:         return
8:     end if
9: end function
```

Algorithm 9 Delete Vertex

```
1: function DELETE_VERTEX(row, col, delete v)
2:     if row_idx[index] == v
3:       or col_idx[index] == v then
4:         row_idx[index] = −1
5:         col_idx[index] = −1
6:         add index to delete
7:     end if
8: end function
```

4 Experimental Setup

In this part, we are going to discuss how we design and data structure, the hardware and software setup and the design of our experiment. In section 4.1, we are going to talk about the data structure design, how we choose the underlying data structure to store vertex and edges, and manage memory in both GPU and CPU. In section 4.2, we are going to talk about the hardware setup and software setup for our experiments. And in section 4.3, we are going to talk about the design of experiments, what metrics we choose to record and how to compare the results.

4.1 Data Structure Design

For matrix storage, since we are focusing on sparse matrix and COO format, we implement the graph data structure with COO format. In the following part, we are going to talk about the underlying data structure of our design, and the basic functionality our data structure provides.

Since we are making a graph data structure that every edge is directed, with weight, and contains functionality to insert, check, and delete vertices and edges, we come up with several data structures that we need to implement for the graph. For basic COO format, we need 3 data structures to store each edge: *row_idx*, *col_idx*, and *value*. Each vertex was given a unique id to be stored in *row_idx* and *col_idx*. *row_idx* stores the starting point of an edge, *col_idx* stores the ending point of an edge, and value stores the corresponding weight of that edge.

For the implementation, we decided to use an array to store the element, and assign a fixed and large enough number as the size of the array. The reason we chose the static array was that the GPU doesn't support dynamic array-vector, which is part of c++'s standard template library. And using an array with malloc is a more direct way to allocate and free memory before and after the graph computations. And for the element inside the array, we are currently using size_t, which can convert with unsigned integers smoothly, and is more robust against numerical stack overflow.

Apart from the three arrays that store edges-related information, we also need other variables to store vertex-related information. Since we want to check the existence of a specific vertex, we give each vertex a unique id for better access. To mark whether a specific vertex is in the graph, we use an array of boolean values, called *v_d*. For example, suppose we have a vertex with id equals 2 and it is in the current graph. Given that vertex 2 exists, the boolean value in position 2 in *v_d* would be true. Vice versa, given that vertex 3 does not exist, the boolean value in position 3 in *v_d*would be false. With a unique id with each vertex, we can easily and efficiently check whether a given vertex exists in the graph.

Lastly, for delete operation, we decide to use queue as the underlying data structure. Though we cannot directly use queue from c++'s standard template library in GPU, we came up with the idea to use an array combining with two pointers to achieve the same functionality of queue. We initialized deleted as an empty array at the beginning of initialization of the graph, and two pointers, head and tail, marking the start and end of the queue. Everytime user decides to remove one edge from the graph, the GPU would search the *row_idx* and *col_idx* with different threads running concurrently. Once the edge was found, the index of the edge would be updated into *deleted[tail]*, then tail would move to right and increase one. Similarly, when a user wants to insert a new edge into the graph, if the deleted array is not empty, and head and tail are pointing at different positions, then the edge would be inserted into the position of *deleted[head]*, then the pointer of head would move right one position.

4.2 Hardware and Software Setup

We do our test on the cuda3 of cims. The hardware and software setup is listed in Table 1 and Table2.

Table 1. Software Setup

Software	Version
CUDA	11.4
gcc	4.8.5
grid size	80
block size	256
total threads	20480

Table 2. Hardware Setup

Components	Version
CPU	Intel(R) Xeon(R) Gold 5118 @ 2.30GHz
GPU	NVIDIA TITAN V
OS	CentOS-7.9

4.3 Experiment Design

After conducting multiple tests, we determined the optimal GPU configuration to be a grid size of 80 and a block size of 256, resulting in a total of 20,480 threads.

Table 3. Experiment Results: Running time (in seconds) of each function for an initially empty graph.

# of tests	10000		20000		50000		100000		200000		500000		1000000	
device	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU
init	0.19	0	0.18	0	0.19	0	0.17	0	0.19	0	0.17	N/A	0.14	N/A
insert v	0.15	0	0.27	0	0.61	0.01	1.24	0.01	2.48	0.01	6.16	N/A	12.25	N/A
insert e	0.26	0.06	0.55	0.1	1.24	0.62	2.78	2.53	5.34	9.75	14.18	N/A	27.2	N/A
check e	0.19	0.08	0.38	0.2	0.93	1.21	1.96	4.9	4.06	19.5	10.51	N/A	21.81	N/A
check v	0.06	0.01	0.16	0	0.34	0.01	0.75	0.01	1.55	0.01	3.84	N/A	7.48	N/A
get weight	0.19	0.05	0.39	0.23	0.94	1.2	2.02	4.69	4.13	19.15	10.39	N/A	22.51	N/A
in degree	0.27	0.12	0.55	0.52	1.52	2.95	2.92	11.8	5.97	47.82	15.28	N/A	32.83	N/A
out degree	0.27	0.11	0.63	0.48	1.49	2.96	2.98	11.95	5.81	47.64	15.92	N/A	34.16	N/A
neighbors	0.54	0.19	1.19	0.82	3.02	5.17	6.12	20.89	12.8	82.81	31.97	N/A	86.53	N/A
source	0.59	0.12	1.03	0.5	2.66	3.02	5.4	11.89	11.42	47.8	28.85	N/A	61.99	N/A
destination	0.52	0.11	1.09	0.5	2.8	2.98	5.36	11.81	11.56	47.56	29.89	N/A	61.95	N/A
delete e	0.21	0.05	0.42	0.21	1.04	1.22	2.03	4.87	4.28	19.53	10.8	N/A	24.73	N/A
delete v	0.16	0.08	0.35	0.33	0.8	2.13	1.55	8.21	3.31	33.34	9.66	N/A	20.96	N/A
total	3.6	0.98	7.19	4.09	17.58	23.68	35.3	93.76	72.9	375.12	186.82	N/A	414.58	N/A

Table 4. Experiment Results: Running time (in seconds) of each function for graphs with initially 1,000,000 edges

vertex	5000				1000000			
# of tests	1000		10000		1000		10000	
device	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU
init	0.32	0.18	0.35	0.18	0.35	0.22	0.33	0.22
insert v	0.02	0	0.15	0	0.02	0	0.14	0
insert e	0.02	0.25	0.31	9.34	0.04	0.77	0.28	0.02
check e	0.01	0.29	0.26	9.53	0.02	0.77	0.22	8.3
check v	0	0	0.07	0	0	0	0.09	0
get weight	0.02	0.21	0.26	8.51	0.03	0.65	0.23	7.35
in degree	0.05	2.92	0.52	29.08	0.05	2.92	0.5	29.17
out degree	0.05	2.93	0.5	29.11	0.05	2.91	0.52	29.43
neighbors	0.1	5	1.07	49.74	0.01	4.94	1.06	49.78
source	0.09	2.92	1	29.36	0.09	2.9	0.98	29.1
destination	0.09	2.93	0.99	29.29	0.09	2.91	0.99	29.36
delete e	0.01	0.27	0.28	8.69	0.02	0.9	0.25	7.79
delete v	0.03	1.38	0.38	21.28	0.05	0.26	0.42	25.22
total	0.81	19.28	6.14	224.12	0.91	22.39	6.02	223.75

Our evaluation of the data structure involved two distinct types of test cases:

1) We started with an empty graph, progressively adding vertices and edges. Subsequently, each function was tested multiple times, from 10,000 to 1,000,000.

2) The graph was initially populated with a large number of edges (1,000,000), which allowed for testing the functions under conditions of reduced CPU-GPU communication latency. We performed tests on both a densely populated graph (with 5,000 vertices) and a sparsely populated graph (with 1,000,000 vertices). The number of test functions is set to 1,000 and 10,000.

5 Results and Analysis

The result is shown in Table 3 and Table 4, we will explain the two parts separately.

In Table 3, we show our test results on an initially empty graph. We calculate and compare each of our utility functions between the GPU version and the CPU version. The paralleled functions and the sequential functions use the same logic, except that the GPU version benefits from parallel computing on CUDA. Below is our finding from our experiment results:

(1) For functions that require inserting edges, vertex or initializing the graph, GPU tends to spend longer time to execute.

(2) We test the same function with various number of test

times, and when the test times is large, GPU has significantly better performance than CPU. Because of the nature of communication overhead between host(CPU) and device(GPU), testing the same function with GPU usually takes longer time when the test size is small.
(3) Table 4 does not reveal any significant drawbacks from data transfer once the graph has been initialized. It is observed that the operations of inserting vertices (insert v) and checking vertices (check v) remain comparatively slow. However, aside from these operations, an acceleration of 30 to 50 times is witnessed in other computational tasks when utilizing GPU over CPU.

When comparing the performance differences between dense (with 5,000 vertex) and sparse (with 1,000,000 vertex) graph structures as presented in Table 4, operations such as computing in-degree and out-degree, as well as identifying sources and destinations, are slower in dense graphs. This slowdown is attributed to the higher number of atomic operations in dense graphs, which necessitate the addition of numerous locks, thereby reducing the speed of execution.

Furthermore, the acceleration ratio during the deletion of vertices (delete v) which in 1000 times of tests is significantly higher in dense graphs compared to sparse graphs. This is because the deletion of a vertex in dense graphs tends to involve the concurrent deletion of multiple edges, a process where the parallel deletion capabilities of GPUs outperform CPUs substantially.

The experiment results we presented indicate that while specific operations like vertex insertion and verification do not benefit significantly from parallelization, the majority of the graph operations has been substantially optimized on GPU architectures. This enhancement is especially notable in dense graphs where the interconnected nature of vertices means that operations can take advantage of the GPU's parallel processing strengths.

6 Conclusions

In this project, we implement a GPU graph data structure with Coordinate Format to represent the adjacency matrix. Taking the advantage of CUDA programming techniques, our graph library parallels several fundamental graph operations to address computational resource and memory bandwidth constraints on traditional CPU graphs. This proposed data structure is well-suited for graph operations requiring rapid insertion, deletion, and edge lookups, outperforming approaches on CPU. We also provide a flexible and user-friendly high-level programming framework, giving good performance and high programmability. Users with minimal GPU knowledge can still benefit from our easy-to-use frontier.

Looking ahead, we can extend our project for some future work. The first is to realize more storage formats for sparse matrices, such as CSR or ELL [8]. We would also like to test our graph library for some applications. Our work can serve as a bedrock for paralleling applications and tasks on GPU. Furthermore, it is also a good idea to extend our graph for the implementation of some other data structure. A straightforward follow-up would be B-Trees [4] which is also useful in practical problems.

Acknowledgments

Thanks to Prof. Mohamed Zahran, our teaching assistants Ta-Hsien Hsu and MingChieh Yang, and cims for CUDA computing resources.

References

[1] Sarah AlAhmadi, Thaha Mohammed, Aiiad Albeshri, Iyad Katib, and Rashid Mehmood. 2020. Performance Analysis of Sparse Matrix-Vector Multiplication (SpMV) on Graphics Processing Units (GPUs). *Electronics* 9, 10 (2020). https://doi.org/10.3390/electronics9101675

[2] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. arXiv:1710.11246 [cs.DC]

[3] Muhammad Awad, Saman Ashkiani, Serban Porumbescu, and John Owens. 2020. Dynamic Graphs on the GPU. 739–748. https://doi.org/10.1109/IPDPS47924.2020.00081

[4] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. 2019. Engineering a High-Performance GPU B-Tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 145–157. https://doi.org/10.1145/3293883.3295706

[5] Vincenzo Bonnici, Federico Busato, Stefano Aldegheri, Murodzhon Akhmedov, Luciano Cascione, Alberto Arribas, Francesco Bertoni, Nicola Bombieri, Ivo Kwee, and Rosalba Giugno. 2018. cuRnet: An R package for graph traversing on GPU. *BMC Bioinformatics* 19 (10 2018), 221–230. https://doi.org/10.1186/s12859-018-2310-3

[6] Erich Elsen and Vishal Vaidyanathan. 2014. VertexAPI2 – A Vertex-Program API for Large Graph Computations on the GPU. https://api.semanticscholar.org/CorpusID:62278405

[7] Douglas Gregor and Andrew Lumsdaine. 2005. The Parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)* (01 2005).

[8] D.B. Kirk and W.W. Hwu. 2012. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science. https://books.google.com/books?id=E0Uaag8qicUC

[9] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. 2014. GraphLab: A New Framework For Parallel Machine Learning. arXiv:1408.2041 [cs.LG]

[10] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2017. Graph Processing on GPUs: A Survey. *Comput. Surveys* (01 2017). https://doi.org/10.1145/XXXXX

[11] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. arXiv:1701.01170 [cs.DC]

[12] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25 (2014), 1543–1552. https://api.semanticscholar.org/CorpusID:3607461