Dynamic Graphs on the GPU

Jimmy Lord
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
jimmylord@cmail.carleton.ca

December 18, 2020

Abstract

Many real-world problems being studied in computer science rely on graph algorithms. In this paper, we look at various data structures used to store these graphs, and examine which would be particularly suited for use in storing dynamic graph data on the GPU. We recreate a hash table based dynamic graph, test it against various data sets and describe some advantages and disadvantages of this particular data structure. We also test two different CUDA algorithms for inserting edges into the hash table based dynamic graph.

1 Introduction

Graphs are data structures fundamental to many areas of computer science. They are used in the study of countless real-world problems including, but not limited to, physical network design (telecommunications, electrical grids, water/sewer networks, etc), message routing, recommendation engines, fraud detection, advertising and machine learning.

Many graph algorithms exist to aid in these applications. These algorithms include search algorithms, such as breadth-first and depth-first search, shortest path algorithms, such as Dijkstra's algorithm and the A* algorithm, subgraph generation algorithms, such as Kruskal's algorithm for generating a minimum spanning tree, connected components, sorting algorithms, minimum spanning tree, page rank, centrality measurements and many more. These algorithms have been implemented and optimised for a variety of system architectures, be it a sequential design for a single CPU or designed for parallelism with multiple local CPUs or multiple CPUs connected via a network.

In recent years, researchers have been working to improve the performance of these graph algorithms on GPUs, which has it's own set of challenges due to memory restrictions and massive parallelisation.

For many of these applications it is not enough to rely on static graphs, they need to support insertion and deletion of nodes within the graph structure either during the execution of a query or otherwise.

This brings us to the topic of this paper, efficient dynamic graph data structures that support quick insertion and deletion of vertices and edges as well as continuing to provide fast queries for the various graph algorithms used in common practice.

2 Literature Review

When it comes to building a data structure to support the creation and querying of graphs, a lot of factors come into play, such as the density of connections between the nodes in the graph and whether the graph nodes and edges are static or able to change over time. This section will explain some of choices that have been made in the past to help support the needs of the graph algorithms, focusing on choices made for the parallelism offered by GPUs.

2.1 Graph Density

When the graph density is high, with many of the nodes of a graph connected to many of the others, the list of edges is often maintained in an adjacency matrix. Due to the memory requirements of these matrices, the maximum number of vertices can be severely restricted as seen in experiments by Buluc et al. [5].

More often, the graphs being studied are sparse, with few connections between their nodes, which by nature is generally true of large graphs. These graphs store their edges in adjacency lists, which in turn leads to much more variety of possible data structures, each with their own trade-offs between query performance, insertion/deletion speed and other properties.

2.2 Static Graph Data Structures

Much of the literature on graphs algorithms on the GPU is focused on storing and querying data in static graph data structures. These data structures don't accommodate for vertices or edges to be inserted and removed from the graph after it has been built and often require a complete rebuild of the structure if any changes are needed. Many implementations of static graphs use data structures such as Compressed Sparse Row (CSR) and other similar structures to store graph edge lists. These formats are chosen for their compactness and speed of query. Static graphs are often built on the CPU and later transferred to GPU memory. An example of this can be found this paper by Bisson et al. [4]

2.3 Dynamic Graph Data Structures

In recent years, more and more work has been done on dynamic graphs, which unlike static graphs allow for the addition and removal of vertices and edges without rebuilding the entire graph. This leads us to the focus of this paper, data structures for use in dynamic graphs on the GPU. There have been three important frameworks focused on working with dynamic graphs. The following subsections detail some of the efforts.

A summary of the features of the mentioned dynamic graph frameworks can be found in Table 1.

2.3.1 STINGER, cuSTINGER and Hornet

In 2009, the STINGER [3] algorithm for dynamic graphs was proposed. STINGER is a dynamic data structure that allows for the insertion and removal of both vertices and edges. Later, in 2012, STINGER [7] was updated with some optimizations, including parallel implementations of the insertion and removal procedures for edges which led to a massive increase in performance of the algorithm.

This later led to the introduction of cuSTINGER [8] in 2016, which offered a GPU extension of the STINGER data structure designed for CUDA. cuSTINGER provides memory management for dynamic or static graph applications with multiple different allocators that can be configured per application.

In 2018, the Hornet [6] data structure was introduced, a GPU data structure designed for dynamic graphs representing sparse data sets. Hornet is a custom data structure designed for the GPU that offers full dynamic batched updates to both the nodes and the edges of a graph while offering a level of stability after large changes to the graph that previous efforts didn't.

The Hornet data structure consists of an array for each adjacency list that get copied into a new larger block when the block gets full, meaning fast traversal of arrays but slow growth due to allocations and memory copies.

2.3.2 GPMA and GPMA+

In their paper [10], the authors introduce a new data structure for storing graphs on the GPU named GPMA. GPMA is a GPU variation of the Packed Memory Array data structure, a self-balancing binary tree, which maintains the adjacencies in a sorted order. This technique has advantages for certain graph algorithms that benefit from having sorted adjacency lists. Though this comes at the cost of a more expensive mechanism for modifying the edge list due to memory locking that prevents modifications of multiple entries in similar positions in the same pass.

Also presented in the same paper is GPMA+, which is a lock-free version of GPMA that sorts the graph changes on the CPU and sends them to the GPU in batches to avoid the locking mechanism.

2.3.3 aimGraph and faimGraph

In 2017, aimGraph [12] was introduced, similar to cuSTINGER at the time, it allowed for dynamic insertion and removal of edges in its adjacency lists, but not for vertices. One big innovation of aimGraph was that it managed memory for the adjacency lists on the GPU, allowing for faster reallocations when the lists got full.

Winter et al. followed up their work on aimGraph with faimGraph [11] in 2018, a fully dynamic data structure that allows for insertions and removal of both edges and vertices from a graph, previous work only allowed for dynamic edges, not vertices. This new implementation performs all memory management on the GPU.

The faimGraph data structure is a linked list of memory blocks for each adjacency list, with new blocks being added to the list as more edges are inserted.

2.3.4 Dynamic Graphs on the GPU

In 2020, Awad et al. released a new framework [2] which implements a dynamic graph structure using the SlabHash [1] dynamic GPU hash table to store the edge lists in a manner that supports fast insertions and deletions.

The authors of this paper run a variety of tests including a triangle counting algorithm to compare the speed of their dynamic graph to that of faimGraph and Hornet. They also compared the speed of creating, both bulk building and incremental building, and maintaining the graph after many insertions and deletions. The authors acknowledged the

fact that a hash table based approach with a single bucket per table ends up looking similar to the faimGraph data structure.

Name	GPU	Vertices	Edges	Memory Management
GPMA	Yes	Fixed	Dynamic	CPU/GPU
STINGER	No	Fixed	Dynamic	CPU
cuSTINGER	Yes	Fixed	Dynamic	CPU/GPU
Hornet	Yes	Dynamic	Dynamic	CPU/GPU
aimGraph	Yes	Fixed	Dynamic	GPU
faimGraph	Yes	Dynamic	Dynamic	GPU
Awad et al.	Yes	Dynamic	Dynamic	GPU

Table 1: Summary of features of various graph implementations.

3 Problem Statement

The goal of this work is to test the hash table based algorithm introduced by Awad et al. [2] against various data sets to determine it's suitability as a dynamic graph data structure for various data sets. Since graph problems come in many different shapes and sizes, it's important to choose the correct data structure for the problem at hand. Getting a better understanding of each algorithm and how appropriate it is for various graphs with different properties can help researchers choose the correct data structure and possibly the correct algorithm that best matches their needs.

4 Implementation Details

This project is primarily a re-implementation of the work described in the paper by Awad et al. [2] and also a re-implementation of the needed parts of the SlabHash data structure used heavily by their framework. It is implemented on the GPU using Nvidia's CUDA programming language.

At the core of the implementation is a dynamic graph data structure that contains an array of vertices. Each vertex holds an adjacency list stored as a hash table. The hash table is a custom structure that contains a series of buckets, the number of which can be configured at compile time. Each bucket being a linked list of slabs.

A slab is a structure defined as part of the SlabHash hash table [1]. The implementation of the slab in this paper differs slightly from the original. In this case, each slab can hold 31 edges and a pointer to another slab, essentially turning each bucket in the hash table into a linked list of slabs.

An image of the overall data structure can be seen in Figure 1.

4.1 Insert

Two different algorithms for inserting edges were implemented and both led to interesting outcomes described in Section 5. These two algorithms can be described as a naive insert and a warp cooperative insert.

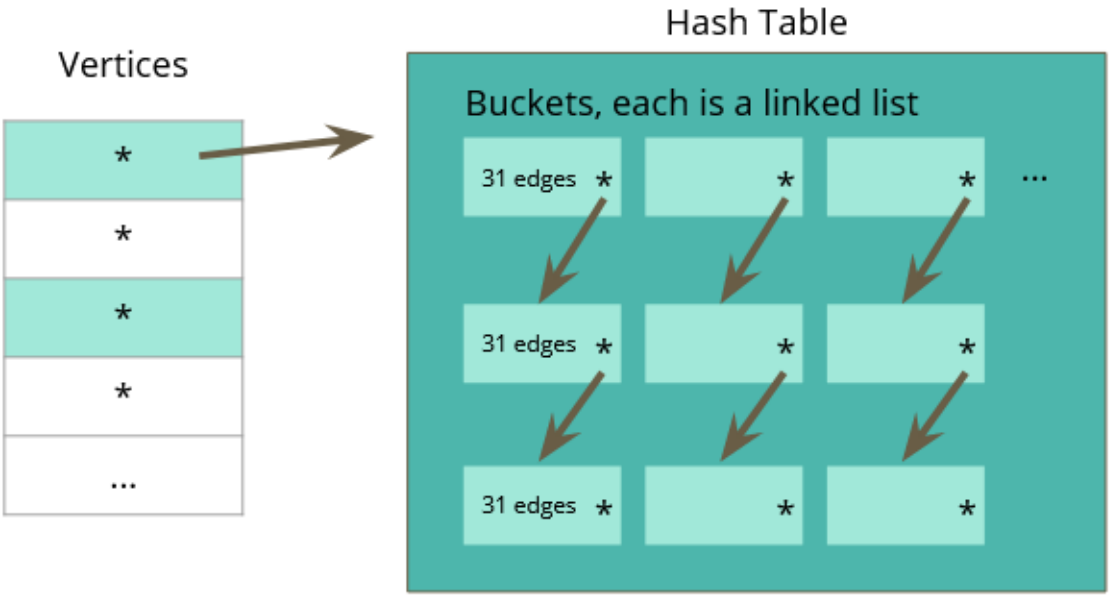

Figure 1: Overview of the dynamic graph data structure

4.1.1 Naive Insert

The Naive Insert algorithm essentially consists of launching one GPU thread for every edge we want to insert into the graphs. There are two pitfalls in this approach, first is thread concurrency the other is branch divergence.

Concurrency issues appear when multiple threads all try to access the same piece of memory at the same time. In our case of inserting edges, the process is as follows. Each thread will be responsible for inserting a single edge into the graph. It starts by calculating the hash of the destination vertex, then it finds the hash table for the first vertex and the bucket within it for the hash value. Once found, a search for an empty slot within the linked list of slabs begins.

This same hash on the same vertex may have been calculated concurrently on many threads and all of these threads could be iterating over the same slab. When two or more threads end up looking at a single slab simultaneously, there won't be a problem until they find the empty slot and try to insert the value, then we have a race condition. This is where we make use of CUDA's atomic functions. In particular atomicCAS, which will atomically compare and swap a single 32-bit value from memory. Due to the SIMT (Single Instruction Multiple Threads) architecture, this will cause a delay for the entire warp of any other threads that are running the same atomic operation on the same piece of memory, i.e. inserting an edge into the same bucket. The procedure can be seen in more detail in Algorithm 1.

For sparse graphs with a low average degree this may not cause much of an issue, but for a graph like soc-LiveJournal1 that have a maximum degree of 23,000, this could cause a lot of delays across many threads in many warps. Some additional graph properties can be seen in Table 2. As can be seen in the table, while a few graphs have a high maximum degree, many graphs have a low average degree which should lessen the impact of the atomicCAS

calls.

The second issue is branch divergence. For example, if one thread of a warp needs to search through a long linked list due to a particularly popular vertex with a large degree, all other threads in that warp will be delayed while the code from one thread follows the list of pointers in the linked list. This particular problem can be remedied using a Warp Cooperative approach, described in the next section.

Algorithm 1 Naive Edge Insert

```
1:  procedure SingleEdgeInsert(numEdges,EdgeList)
2:      index = blockIdx * blockDim + threadIdx
3:      stride = blockDim * gridDim
4:      while index < numEdges do
5:          if v1 != v2 then                                  ▷ Skip edge loops
6:              Find hash table for v1
7:              Calculate hash for v2
8:              Find bucket for v2 hash
9:              while !found and !inserted do
10:                 for i ← 0, 31 do
11:                     if slab[i] == v2 then
12:                         found = true
13:                         break                             ▷ Edge already in list
14:                     end if
15:                     if slab[i] == empty then
16:                         Insert with atomicCAS             ▷ Empty slot found
17:                         inserted = true
18:                         break
19:                     end if
20:                 end for
21:                 if !found and !inserted then
22:                     Follow link to next slab
23:                 end if
24:             end while
25:         end if
26:         index += stride
27:     end while
28: end procedure
```

4.1.2 Warp Cooperative Insert

A warp is a collection of 32 GPU threads that all run in a SIMT manner. All instructions running on a warp are executed in sync with potentially different data. An approach that takes advantage of this what Ashkiani et al. call their "warp-cooperative work sharing strategy" [1].

Due to the memory architecture of GPUs, threads on the same warp share a small block of extremely fast access memory, this allows the 32 threads of a single warp to communicate extremely efficiently. The warp cooperative insert takes advantage of this by having the threads of a warp work together to insert one edge at a time into the graph.

Communication is done with two CUDA instructions, ballot sync, which lets each thread set a single bit of a 32-bit value and shuffle sync which copies the value of a 32-bit variable from one thread of a warp to the other threads. Using these mechanisms we're able to control all 32 threads of a warp and insert values into the hash table much quicker without worrying about branch divergence, since all threads will be doing a small portion of a larger task in tandem. The procedure is shown in more detail in Algorithm 2.

As a simplified example, take a case where the threads of a warp are tasked to insert 32 edges into the graph. The threads start by doing the same work the naive algorithm does, such as calculating hashes, bucket addresses, etc. for the edge they were originally tasked. Then they would all work together, by first communicating the calculated data between their peers within the warp, then each looking a single entry of the slab to determine where to insert the edge.

Since all threads are working together there is no divergence in their action, they are always busy doing their slice of the work. If one edge requires following a long chain of links, all threads need to follow along to check the next slab and so on.

Even though there's a bit of overhead not present in the naive approach, this approach tends to work well for graphs with many buckets with significant chain lengths.

One other potential benefit to this approach is if the edges being inserted are sorted before being inserted, a cooperative warp group could insert many edges into the same bucket without duplicating work. Which would also contribute to performance gains due to improved cache coherence from the memory access patterns of the warp.

Name	Vertices	Edges	Max Degree	Avg Degree
luxembourg_osm	114,599	119K	6	2
germany_osm	11,548,845	12.3M	13	2
road_usa	23,947,347	28.8M	9	1.2
delaunay_n23	8,388,608	25.1M	28	5
delaunay_n20	1,048,576	3.1M	23	5
rgg_n_2_20_s0	1,048,576	6.9M	36	13
rgg_n_2_24_s0	16,777,216	132.5M	40	15
coAuthorsDBLP	299,067	977K	336	6
ldoor	952,203	23.7M	78	24.9
soc-LiveJournal1	4,847,571	68.9M	23K	14.2
hollywood-2009	1,139,905	57.5M	11K	50.5
lp-d6cube	6,184	37.7K	6K	12

Table 2: Properties of various graphs. [9]

5 Experimental Evaluation

All tests were run on a GeForce GTX 970 with 4GB of RAM, which restricts the data sets that can be loaded and the number of buckets that can be assigned to each vertex. Various tests have been run using a few data sets and the results show the average of five runs of each test. The following is an explanation of the results and highlighting some strengths and shortcomings of the two implemented algorithms.

Algorithm 2 Warp Cooperative Edge Insert

```
1: procedure WarpCoopEdgeInsert(numEdges,EdgeList)
2:     index = blockIdx * blockDim + threadIdx
3:     laneID = threadIdx % 32
4:     if v1 != v2 then
5:         toInsert = false                                   ▷ Skip edge loops
6:     end if
7:     Find hash table for v1
8:     Calculate hash for v2
9:     Find bucket for v2 hash
10:    while workQueue ← BallotSync(toInsert) do
11:        workLane = FindFirstSet(workQueue)          ▷ Which thread's work will we do
12:        workV1 = ShuffleSync(workLane)                ▷ Transfer data from work lane
13:        insertThisPass = v1 == workV1                ▷ Insert all edges for same vertex
14:        while innerQueue ← BallotSync(insertThisPass) do
15:            innerLane = FindFirstSet(innerQueue)
16:            workV2 = ShuffleSync(innerLane)
17:            workBucket = ShuffleSync(innerLane)
18:            while !found and !inserted do
19:                found = BallotSync(slab[laneID])              ▷ Check all lanes at once
20:                emptySlots = BallotSync(slab[laneID] == empty)
21:                emptySlot = FindFirstSet(emptySlots)
22:                if laneID == emptySlot then
23:                    Insert with atomicCAS                          ▷ Empty slot found
24:                    inserted = true
25:                    toInsert = false;
26:                    insertThisPass = false;
27:                end if
28:                if !found and !inserted then
29:                    Follow link to next slab
30:                end if
31:            end while
32:        end while
33:    end while
34: end procedure
```

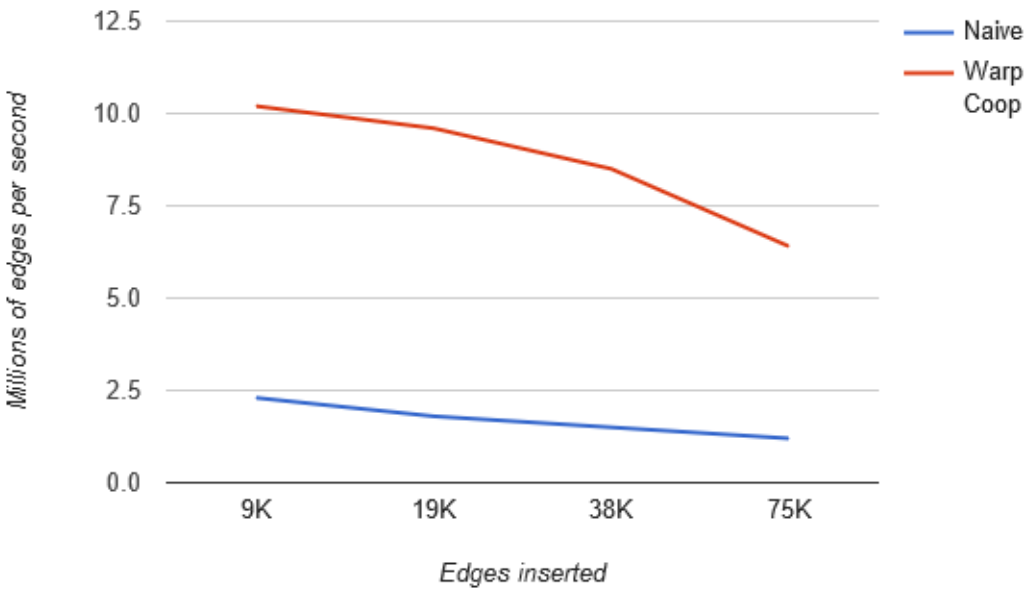

Figure 2: Insert speed for graph with high max degree. Data set: lp-d6cube

5.1 Edge volume

Figure 2 shows the results of inserting a different number of edges, in all cases the edges come from the lp-d6cube data set. As can be seen in Table 2, this graph contains roughly 38k edges and has a fairly high maximum degree of 6k. The high degree means the graph needs to store the edges in hash tables that might have very long chains of slabs, since each bucket in the hash table only holds 31 vertices, as the graph fills up each thread needs to traverse an ever growing linked list to insert new edges.

This test was run on a graph with a single bucket in the hash table of each vertex, the same setup Awad et al. used when running their incremental build tests [2]. As can be seen in the chart, as the number of edges being inserted grows, the performance (measured in edges per second) drops, indicative of the increased cost of traversing of the linked list as the graph grows. The average degree of this graph is only 12, the majority of the vertices will still be able to fit their edge lists inside one slab and not incur that additional cost. A graph of higher density would have a more severe drop in performance as it grows.

Another noteworthy result can be seen in this result. That being the difference in performance between the naive insert algorithm and the warp cooperative algorithm. This can be explained in large part due to branch divergence. In the naive algorithm, threads that have to traverse a large linked list of slabs to insert a single edge slow down all other threads in the warp, while the warp cooperative algorithm doesn't suffer from the drawback, leading to a 4-5 times performance gain.

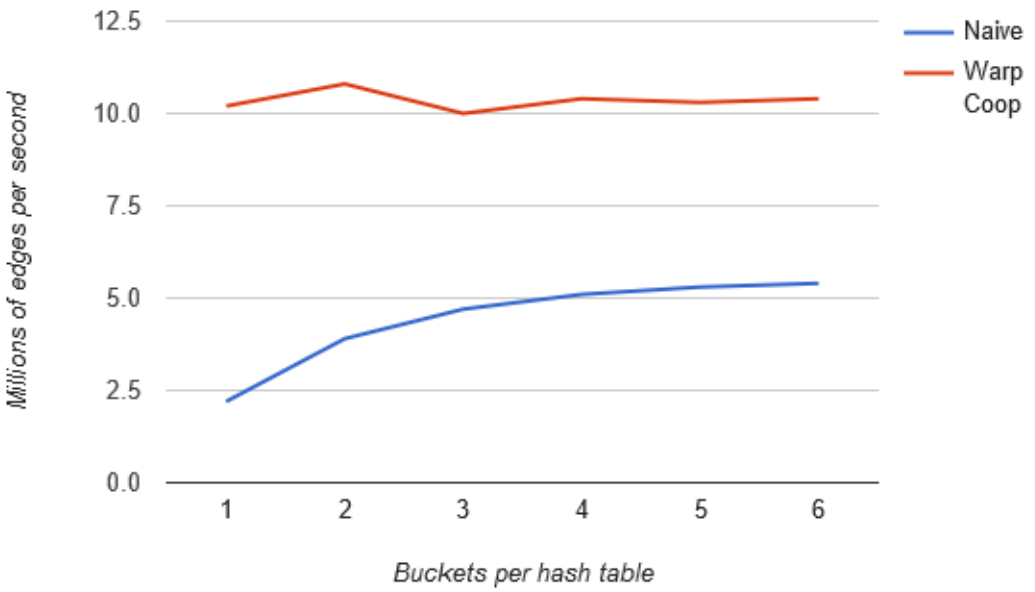

Figure 3: Insert speed with changing number of buckets. Data set: lp-d6cube

5.2 Variable Bucket Count

One key setting that has quite an effect on memory usage versus performance is the number of buckets each hash table has. Figure 3 shows the effect on performance of the two algorithms with a varying number of buckets being assigned to each hash table, but this comes at the cost of additional memory being allocated and mostly unused.

5.2.1 Memory Usage

Hash table usage has many advantages, among them the constant cost of insertions and deletions from the edge lists.

One disadvantage of hash tables turns out to be the memory requirement. The hash table implementation works on 128-byte slabs which represents one bucket of the hash table and can hold up to 31 neighbours for a given vertex and each slab holds a link to the next slab if it overflows. Hash tables have flexibility over arrays in that a good hash function distributes evenly over a number of buckets, but these buckets if left largely unused will require a lot of memory.

So, if we take the following scenario, each vertex has a hash table containing 5 buckets for its edge list, with each bucket at 128-bytes in size, a graph such as road_usa contains about 24 million vertices. The storage for these hash tables would amount to approximately 15 gigabytes of memory. Ideally, we choose the right bucket count for the right purpose.

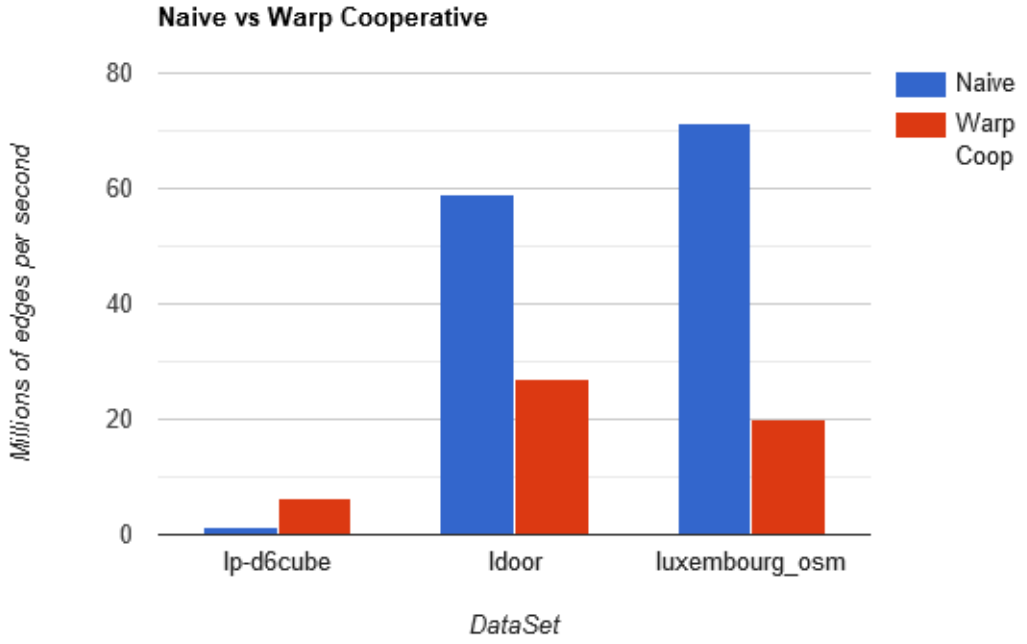

Figure 4: Naive Insert vs Warp Cooperative insert of different data sets

5.2.2 Performance

In Figure 3, we see the performance differences of the two algorithms running with a different number of initial buckets for each vertex. These tests were all run against the lp-d6cube data set, the properties of which were discussed in Section 5.1. As can be seen, increasing the initial bucket count improved performance for the naive algorithm but didn't have any consistent effect on the warp cooperative algorithm.

The reason for the increased performance in the naive algorithm is due to reduced branch divergence. By spreading the edges of all the high degree vertices into multiple buckets, the number of linked list traversals needed by the hardest impacted threads is reduced, giving a general boost to performance. This, of course, comes at the cost of many more slabs being allocated for all of the vertices, and with the average degree of this data set being 12, most of that memory goes unused.

With regard to the lack of change in performance in the warp cooperative algorithm, this can be seen as one of the big benefits of that approach in general, the algorithm itself softens the effect of branch divergence. By having all the threads of a warp work together, no single thread is halting the performance of the others.

5.3 Variable Bucket Count

The final test results can be seen in Figure 4, this chart shows the performance differences between the naive algorithm and the warp cooperative algorithm across 3 data sets, all tests use a single bucket per hash table.

The 3 data sets are lp-d6cube, which has a high maximum degree and low average degree,

ldoor, which has a fairly low maximum degree and low average degree and luxembourg_osm, which has an extremely low maximum and average degree.

The main conclusion that can be drawn from this graph is that when the maximum degree of a graph is low, as in the two cases on the right, the naive insert method has a large advantage over the warp cooperative method. Since branch divergence isn't a factor until you have large linked lists growing inside the buckets, the advantages of the warp cooperative method don't out-weight the overhead.

The luxembourg_osm map in particular is an outlier with a maximum degree of only 6 and an average degree of 2, the hash table never needs to allocate additional slabs and the naive insert algorithm can quickly check the single slab for duplicate edges and empty space for insertion.

6 Conclusions

The graphs being studied are generally large and quite sparse, with few neighbours per vertex, especially the road graphs. Unfortunately, we can't make any general assumptions about their makeup. Graphs with large average degree will benefit from a higher initial bucket count in the hash table for each vertex by avoiding the need for each thread to traverse a long linked list of slabs. This, of course, comes at the cost of increased memory.

As well as the cost of traversal, the overhead cost of the warp cooperative approach made an impact on certain graph types. If the graph had such a low number of adjacencies that branch divergence wasn't much of a factor, a naive edge insertion algorithm could be used to avoid this overhead. Many of the graphs looked at in this work and in the work of Awad et al. [2] have a very low average degree, requiring only a slab or two to hold the adjacency list for each vertex, so don't gain the benefits of a warp cooperative approach.

6.1 Future Research

While the effects would be subtle, the warp cooperative algorithm takes advantage of multiple threads all inserting an edge into the same bucket by inserting them in small batches, but this only applies to the communication happening between groups of 32 threads. In the bigger picture if edge inserts were sorted before being sent to the GPU, performance gains could be achieved. There would also be gains due to the memory locality and cache usage if multiple threads of the same warp were accessing the same, or adjacent, buckets.

Another area that could be improved would be in the number of buckets used by a vertex. In the incremental build scenario that we used for most tests in this paper, we started with a single bucket per hash table, when the number of edges starts to grow it might be possible to redistribute these vertices into a more balanced multi-bucket hash table. This could possibly be a done in a separate phase between insert/query/delete batches, and would be similar in process to a vertex delete and vertex create with the exception that the edges pointing to the vertex in question wouldn't need to be removed from the adjacency lists of the other vertices, which would simplify and speed up the process.

References

[1] S. Ashkiani, M. Farach-Colton, and J. D. Owens. A dynamic hash table for the gpu. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*,

pages 419–429, May 2018.

[2] M. A. Awad, S. Ashkiani, S. D. Porumbescu, and J. D. Owens. Dynamic graphs on the gpu. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–748, May 2020.

[3] D. Bader, J. Berry, A. Amos-Binks, D. Chavarria-Miranda, C. Hastings, and K. Madduri. "STINGER: Spatio-temporal interaction networks and graphs (STING) extensible representation", 2009.

[4] M. Bisson and M. Fatica. Static graph challenge on gpu. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, Sep. 2017.

[5] Aydın Buluç, John R. Gilbert, and Ceren Budak. Solving path problems on the gpu. *Parallel Computing*, 36(5):241 – 253, 2010. Parallel Matrix Algorithms and Applications.

[6] F. Busato, O. Green, N. Bombieri, and D. A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, Sep. 2018.

[7] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, Sep. 2012.

[8] O. Green and D. A. Bader. custinger: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sep. 2016.

[9] misc. Network repository. http://networkrepository.com/misc.php, 2020. Accessed: 2020-11-29.

[10] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *Proc. VLDB Endow.*, 11(1):107–120, September 2017.

[11] M. Winter, D. Mlakar, R. Zayer, H. Seidel, and M. Steinberger. faimgraph: High performance management of fully-dynamic graphs under tight memory constraints on the gpu. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 754–766, Nov 2018.

[12] M. Winter, R. Zayer, and M. Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, Sep. 2017.