Dynamic Hopscotch Hash Tables on the GPU

John Shortt
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
johnshortt@cmail.carleton.ca

December 18, 2020

Abstract

A hash table algorithm that doesn't, to date have a GPU realization, is designed, implemented and tested. This algorithm, Hopscotch Hashing, shows significant performance improvement over the state-of-the-art SlabHash for search operations on tables with up to 1 million (2^{20}) items. Measurements are made on an NVIDIA Titan V GPU for tables containing between 2^{15} and 2^{20} items. Hopscotch Hashing is found to have an average search rate of 6.5 billion operations per second compared to 3.9 for SlabHash on these tables.

1 Introduction

1.1 Parallel Computing

Parallel computing is the study of computer and network architectures to determine how to efficiently solve a problem by using large numbers of interconnected processors to each individually solve part of the problem. Technologies have evolved to allow this to become more and more feasible and the limits of single processor computing power in the light of the thermal barrier have made these architectures more and more important. Many model taxonomies for describing the types of architecture have been described in the literature. The purpose of this paper is to explore the use of a particular model and technology (GPU) to solve a specific problem (dynamic hash tables).

If we consider a model taxonomy that's based on program control and memory access then one model that we come to is one where multiple, identical cores operate on separate memory streams but execute a shared instruction stream. This is the SIMD model. It's particular well suited to problems where data can be partitioned and assigned to each core and each core can compute its part of the solution independently of the others. This kind of problem occurs frequently in image processing problems (both 2D and 3D). Examples are image clipping, linear image transformations, rendering and shading. Because of this the graphics processor unit (GPU), with typically thousands of cores, has been a highly successful implementation of the SIMD model. Contrast this to the MISD model where we have multiple cores sharing a common memory with each core executing a separate instruction stream. This model is that of the multi-core processor (CPU).

1.2 Hash Tables

Researchers have turned to the question of leveraging GPU technology beyond graphics algorithm processing to more general purpose computing problems (GPGPU). One such problem that's received attention recently is that of implementing a hash table that supports search, insert, update

and delete operations. This is a basic capability that is useful in a broad range of application areas (such as bioinformatics, computational geometry, deep learning, visualization and data analysis).

A hash table is a data structure that implements an associative array with the following characteristics: an item with a given key can be stored, retrieved or deleted in $O(1)$ time; if n items are expected to be stored in the table then $O(n)$ memory is used. Note that the number of possible keys, N, can be much larger than n, the expected number of items. For this reason a *hash function* is used to map a key into a position in the data structure. This position is used by the insert, search and delete operations. Since we can have $n << N$ there exists the possibility that two keys hash to the same position and so a method for handling a *collision* is required.

To implement and efficient hash table on a GPU there are a number of design decisions to be made notably: the method for handling collisions, a memory management strategy, and a work distribution (or parallelism) strategy. Collision handling mechanisms have been well-studied [4] and there are essentially two different approaches generally used: *chaining* where items that hash to the same position are stored in a list and *open addressing* where such items are stored at a nearby position.

An empty hash table is initially allocated a fixed amount of memory based on the expected number of items that it will contain. Memory management strategies take two forms that are determined by the decision on how collisions are handled. If chaining is used then the lists that contain collided items are grown as they are filled. If open addressing is used then the table needs to be re-sized and re-filled, with possibly a new hash function, if its allocated space is exhausted.

The work distribution strategies that can be used are strongly influenced by the GPU architecture. Typically operations (insert, search, delete) are batched and work items in the batch are handled by a single or multiple GPU threads.

Results to date have shown that with state of the art GPU hardware rates of operations numbering in the hundreds of millions or billions per second can be achieved. These results appear to be an order of magnitude higher rates than can be achieved with a MISD architecture.

1.3 Project Goals and Results

The purpose of this project is to study recent implementations of GPU-based dynamic hash tables to determine their performance characteristics, strengths and weaknesses and to explore ways to improve upon these implementations. Dynamic hash tables are those which allow further changes in the data structure (via insert or delete) after an initial batch of insert operations is performed. Static hash tables, by contrast, only support an initial batch of insert operations. In this report we focus on dynamic hash tables since many algorithms and applications require this capability.

A dynamic hash table based on the Hopscotch Hashing algorithm[8] has been implemented and tested. For search operations it has been found to exhibit improved or comparable performance characteristics (operations per second and memory consumption) when compared to existing GPU based implementations.

1.4 Structure of the Paper

The remainder of this paper is structured as follows. In Section 2 we review the existing literature related to GPU-based hash table implementations. We also describe the technical details of those that support dynamic operations. In Section 3 we outline the problem that we wish to solve. In Section 4 we describe our proposed solution including technical details of the algorithms that we have implemented. In Section 5 we summarize the results of tests that were performed on an existing dynamic hash table implementation and our hash table implementations. Finally Section 6

summarizes our results and contains suggestions for further work to be done in this area.

2 Literature Review

2.1 GPU Hash Tables

A review of the literature finds no fewer than seven implementations of GPU-based hash tables that fall into three broad categories:

- Static hash table implementations for which there are papers available. Since the focus of this project is on dynamic hash tables this category is included for completeness but no study of these implementations is done. These are HashFight[11] and HashGraph[7].
- Hash tables that are implemented in broadly used GPU libraries. There's limited technical data available about them but they are used as baseline comparison implementations in the papers that fall into the other categories. These are cuDPP[12] and **cuDF**[13]. cuDPP is a static hash table implementation and so is not explored further.
- The category of interest is implementations of dynamic hash tables for which there's significant relevant technical literature. These are **GelHash**[6], **SlabHash**[1] and **WarpCore**[9]

Table 1 summarizes some characteristics found in the referenced literature for these implementations. The performance data is unconfirmed and is likely based on different hardware.

Name	**Code**	**Insert**	**Search**	**Dynamic**	**Features**
cuDF				Yes	Python DataFrame implementation
GelHash	no	800 M/s	2.5 B/s	Yes	group atomicity
SlabHash	github	512 M/s	937 M/s	Yes	slab-based memory allocation
WarpCore	github	1.6 B/s	4.3 B/s	Yes	batch and individual updates, multi-GPU
HashFight	github	700 M/s	2.5 B/s	No	platform neutral; based on data parallel primitives
HashGraph	github	2.5 B/s	2.0 B/s	No	Compressed Sparse Row graph data structure
cuDPP				No	

Table 1: Summary of GPU-based hash table implementations

The thrust of this project will be to perform an in-depth study of four dynamic hash-table implementations namely GelHash, SlabHash, WarpCore and cuDF. The remainder of this literature review is a summary of the papers that describe the implementations of dynamic GPU-based hash tables.

2.2 SlabHash

SlabHash uses chaining as its mechanism for handling collisions. It uses the slab-based allocation scheme used successfully in UNIX and Linux for kernel objects[3] that's known to be fast and to be

resistant to memory fragmentation. Each slab (the basic unit of memory allocation in this scheme) can hold up to 32 hash table entries. Each position in the hash table stores one or more slabs. Multiple slabs are formed into a linked list when the capacity at a given position requires it.

The thousands of threads that can run in parallel on an NVIDIA GPU CUDA manual have a basic unit of scheduling termed a *warp* (see [5] section 4.1). A warp is comprised of 32 threads that execute identical streams of instructions in parallel and in lockstep. The threads within a warp are termed *lanes*. One lane cannot execute the next instruction in the stream until all lanes have completed execution of the current instruction. When a conditional branch occurs (such as an if, while or for statement) all lanes wait at the instruction following the conditional part of the stream until all lanes arrive at that instruction. This is known as *branch divergence* and is an important factor in analyzing the performance of a GPU algorithm implementation. See Figure 1 for a representation of this.

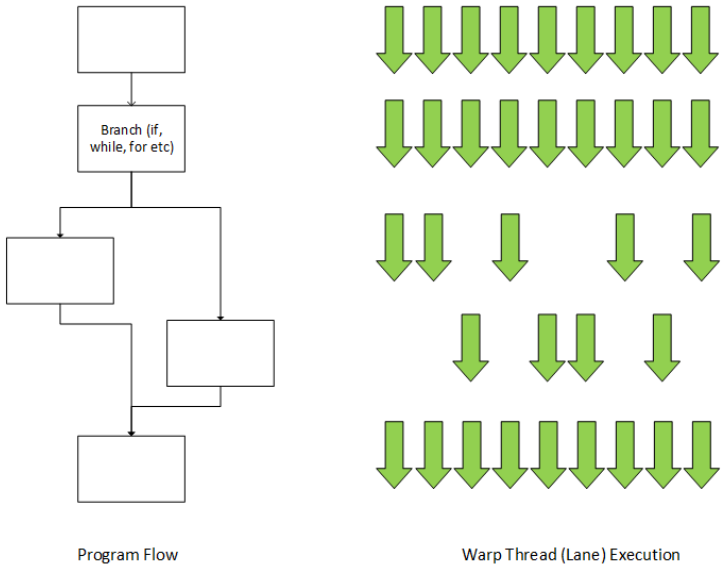

Figure 1: Warp Divergence - lanes that execute one branch code path wait until lanes that execute the other branch complete

GPU memory hardware (bus, cache) is optimized for sequential access. The hardware attempts to coalesce memory access within a warp and so the highest memory efficiency can be achieved by ensuring that each lane in a given warp is accessing memory that can be coalesced with the accesses of the other lanes in that warp i.e that lane memory accesses are spatially localized.

CUDA provides methods that allow lanes to communicate, to exchange data and to perform atomic operations. Using these methods as building blocks, SlabHash implement very fine-grained parallelism for each individual hash table operation by distributing the work for that operation over the lanes in a given warp. This is known as a *warp cooperative* approach.

To see how this is achieved consider the insert operation. A simple implementation might consist of the one described in Algorithm 1 with each thread being responsible for inserting a single item into the table. Note that this algorithm exhibits branch divergence (lines 4, 6 and 9), a race condition (lines 7 and 8) and poor cache affinity (line 11).

Algorithm 1 Simplified high level algorithm for hash table insert

```
1:  procedure INSERTITEM(key, item)
2:      // hash the key to get the position of the item in the hash table
3:      position ← hash(key)
4:      while true do
5:          // if the position is empty then store the key and item there
6:          if bucket.key == EMPTY then
7:              bucket.key ← key
8:              bucket.item ← item
9:          else
10:             // get the next position
11:             position ← position.next
```

SlabHash takes a different approach. Since each slab contains up to 32 items that hash to the same position it assigns each of those 32 entries to a lane and makes that lane responsible for performing any operations to be performed on that entry. This is a warp level algorithm in which the lanes cooperate to achieve the result of inserting an item. Algorithm 2 is the pseudo code of a simplified version of the algorithm implemented in SlabHash. The operations that are described in lines 2,3 and 4 might seem like they require searching and looping to perform but, in fact, each lane performs the work required for the entry that's assigned to it and the CUDA libraries provide APIs that allow the results of these operations to be made known to every other lane.

Algorithm 2 Pseudo code for simplified warp cooperative hash table insert algorithm

```
1:  procedure INSERTITEM(lane, slab, key, item)
2:      while the item for some lane hasn't been inserted do
3:          find a lane with an item to insert
4:          find a lane whose entry in the slab is empty
5:          if no lane has an empty entry then
6:              // this is the case where the slab is full
7:              if there's a next slab then
8:                  make it the current slab
9:              else
10:                 allocate a new next slab and make it current
11:         else
12:             // this is the case where some lane has an empty entry
13:             if it's our lane that has the empty entry then
14:                 store the item
15:                 mark this lane as having no item to insert
```

This algorithm doesn't eliminate or even reduce branch divergence. But since each slab is allocated as contiguous memory and the warp processes a single slab at a time the algorithm achieves greater cache usage than the simplified algorithm. The SlabHash authors credit this for improvements in performance when compared to methods that aren't warp cooperative.

2.3 GelHash

This paper claims improvements over the approaches taken in SlabHash and cites online transaction processing, OLTP, as an application area that demonstrates these improvements. Unfortunately no

implementation of GelHash seems to be available and so it wasn't possible to confirm these claims.

As with SlabHash, collision resolution is by chaining, a specialized approach to memory allocation is defined and a data structure and locking strategy that benefits from the memory allocation approach is employed. GelHash has both a warp-cooperative and non-warp cooperative mode of operation. It has 2 novel capabilities that aren't found elsewhere:

- It supports a locking mechanism that allows multiple operations (insert, delete) to be performed atomically. This intended to support a transaction processing capability.
- It supports an adaptive mode of operation. It can learn an application's typical operation pattern and optimize the use of warp or non-warp modes of operation given this pattern. A benchmark/learning process is required to be run but investigation is required to determine if there are be applications where this isn't possible or doesn't provide consist results.

Performance figures from the paper show that GelHash outperforms SlabHash for tables containing up to 2^{17} items and is outperformed by SlabHash when tables contain 2^{22} items.

2.4 WarpCore

This paper is an update to the authors' earlier work on WarpDrive[10] which was restricted to 32 bit single-value hash tables. It claims the highest performance numbers of any of the gpu-based gash table implementations reviewed (both static and dynamic) and also claims that this represents a two orders of magnitude increase over MISD based solutions. The paper states that an implementation will be released as open-source when the paper is published (it's currently only available on arxiv).

The hash table implementation is described as using open address hashing with a hybrid probing scheme with double hashing and linear probing. It implements warp cooperative algorithms and since it uses open addressing it has no need for a dynamic memory allocation mechanism.

It is also scalable across multi-GPU architectures using NVIDIA's proprietary NVLink technology. An interesting aspect of this implementation is a set of host interfaces that simultaneously and efficiently allow both batch and individual updates. This means that, for example, different host programs performing different task that have different update patterns will work efficiently.

The authors describe how WarpCore can be used to improve the performance of a bioinformations application, metagenomic classification.

2.5 cuDF

cuDF stands for CUDA Data Frame. It's part of the Python CUDA RAPIDs Open GPU Data Science project. A data frame is a popular Python data structure (typically from the PANDAS library) that implements a data table whose rows and columns can be accessed directly by their key. Hash tables are the way this is implemented. This project is well-supported and is likely one of the first places that data scientists consult when looking for GPU libraries to support their projects. Any hash table implementation that seeks to be widely used would need to support all the capabilities of cuDF in a way that applications benefit from.

3 Problem Statement

The objective of this paper is to explore ways to improve hash table performance when compared to existing implementations. Although the results vary depending on the number of items in the

table, SlabHash generally has the best performance characteristics among the implementations reviewed. It also has the benefit of being the most readily available for testing purposes and so is focus of the comparison. Tests of SlabHash performance are based on code that is downloadable from GitHub[2].

The thesis is that performance improvements when compared to SlabHash, especially for search operations, can be achieved. The motivation for this two-fold:

- SlabHash allows the growth of long chains of slabs when many keys map to the same position. This results in linear scanning of the slabs in those chains by a search operation.
- When a slab is partially filled (a frequent case) then only the lanes associated with the non-empty slab entries participate in the search operation. This effectively reduces the search rate.

These limitations have been traded off for the benefit of the warp cooperative approach, namely greater cache utilization, that SlabHash takes. The belief that this is a beneficial trade-off for insert operations but not necessarily for search operations.

Any improvement in hash table performance has the potential of benefiting the numerous algorithms and applications that require a hash table as part of their design or architecture.

4 Proposed Solution

To explore and improve hash table performance we

- Run tests on the existing SlabHash implementation
- Implement a simple hash table that uses open addressing, searches for the nearest entry when collisions occur, doesn't use warp cooperative algorithms
- Implement the Hopscotch Hash algorithms

4.1 SlabHash

Since the code for SlabHash is readily available on GitHub it was straightforward to download and build a benchmark application that the authors had written. As a first step this benchmark application was run using a broad set of table parameters. This will provide a set of performance data against which other implementations can be assessed.

4.2 Simple Hash

In order to understand the SlabHash warp cooperative advantage a hashing algorithm, Simple Hash, that doesn't use a warp cooperative approach and handles collisions using a simple open addressing scheme was implemented. It distributes work to GPU threads in a conventional manner: each thread performs one insert, search or delete operation from a batch.

The benefit of having this simple implementation available is that it can serve as a learning device to better understand more complex algorithms. It also affords the author a simpler introduction to CUDA programming than what follows. It's not intended to be used in any real application.

4.3 Hopscotch Hash

The novel aspect of this project is the implementation of Hopscotch Hashing in the GPU. It uses a form of open addressing that ensures that an item is never inserted more than a given distance, H, from the position that it hashes to. H is specified when the empty table is created and can't be changed thereafter.

If necessary, during an insert operation items are moved, respecting the H distance constraint, to create an empty entry for the item being inserted. In the paper[8] the authors show that the probability that the insert operation fails because the empty entry can't be created in this way is $1/H!$. In the tests that we performed we choose H=32.

To move an item in the table and to ensure that it's not moved further than H from its hash position it's necessary to know what its hash position is. There are two possible ways to do this:

- The hash position of the item can be stored and retrieved when needed. If the position is stored in the hash table or in a parallel data structure this retrieval will be fast. It will need to be updated when the item is moved and it increases the storage required for the hash table.

- The hash position can be re-computed from the key when needed. Since the key is stored in the hash table it's fast to retrieve. The cost of doing this calculation needs to be assessed but since no additional memory access is required for this it's expected that it's not significant.

The second option of re-computing the position by hashing the key was chosen. There may be use cases where the first option is preferable.

The most difficult aspect of implementing Hopscotch Hashing is ensuring that there are no concurrency issues in the insert operation. Since this operation may move multiple entries in order to insert a single key, care has to be taken that these operations are performed in a way that leaves the hash table in a consistent state at all times. And this has to be done in a way that minimizes the impact on performance i.e. the throughput of a large batch of simultaneous insert operations.

To simplify the diagnosis of issues the approach of exhaustively testing the implementation in a serial, single operation at a time mode was undertaken first. A number of issues and edge cases were discovered by doing this. Once these were corrected and it was known that the basic functionality of the algorithms was correct, testing with concurrent operations could be performed. Again the principle of starting small was observed. Most bugs were found and fixed during the construction of a table of just 32 items.

The correct operation of the insert operation was achieved by essentially implementing two cases for updating the table:

- In the first case there's an empty position within H of the position that the key to be inserted hashes to. In this case we insert the item there. We find this entry by linearly probing (searching) for an empty entry.

- In the second case there's no empty position within H of the hash position. In this case we move an item to the empty slot that's closest to the hash position of the item we're inserting.

The insert operation proceeds by performing move operations (the second case) until the first case becomes possible. We may discover that the table is full while doing this but as the authors of the Hopscotch Hashing paper explain the probability of this is $1/H!$ if the keys are distributed randomly.

To ensure that the hash table is kept in a consistent state an atomic Compare and Swap (CAS) operation is used for all memory update operations. Inserting the item in the hash table requires a single memory write to an empty position and so CAS is used to ensure that the position is empty

when the item is written. Moving an item to an empty position requires two memory writes: one to write the item in the empty position and a second to mark the position of the moved item as empty. CAS is used for both of these operations but either of them could fail if another lane or GPU thread has updated the memory. By performing these two operations in the order described (store item in empty position first, mark item's previous position as empty second) we ensure that the state of the hash table is consistent even if one or both of these operations fail.

If any CAS operation fails we take the pessimistic action of restarting the insert operation. There may be performance improvements possible by taking a less drastic action.

Search and delete operations are simpler. They require linearly probing the positions within H of the position that the key hashes to. When deleting a CAS operation is performed to mark the position containing the key as deleted. Since the keys are known to be within H of the hash position there's no need to use a special marker for deleted items as in other open addressing schemes.

5 Experimental Evaluation

For all three hash table implementations (SlabHash, Simple Hash and Hopscotch Hash) a comprehensive set of performance tests was done. This consisted of measuring the throughput rate for insert and search operations for each type of hash table with load factors ranging from 0.60 to 0.95 and for tables ranging in size from 2^{15} to 2^{27}. Load factor is a measure of the proportion of the hash table positions that contain an item. If the load factor is α and the hash table is expected to store n items then the hash table will have α/n positions.

5.1 SlabHash

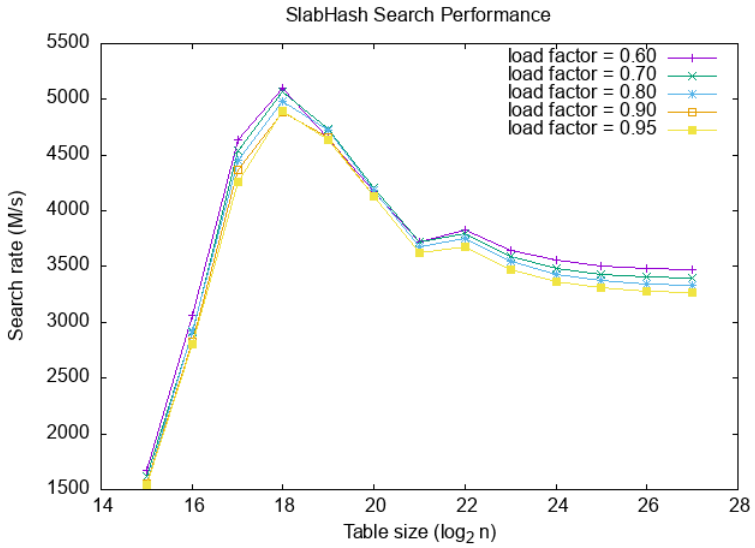

Figure 2: Performance of SlabHash search function in millions of operations per second for different load factors and different table sizes

Examining the performance charts in Figures 2 and 3 we observe the following aspects of SlabHash performance:

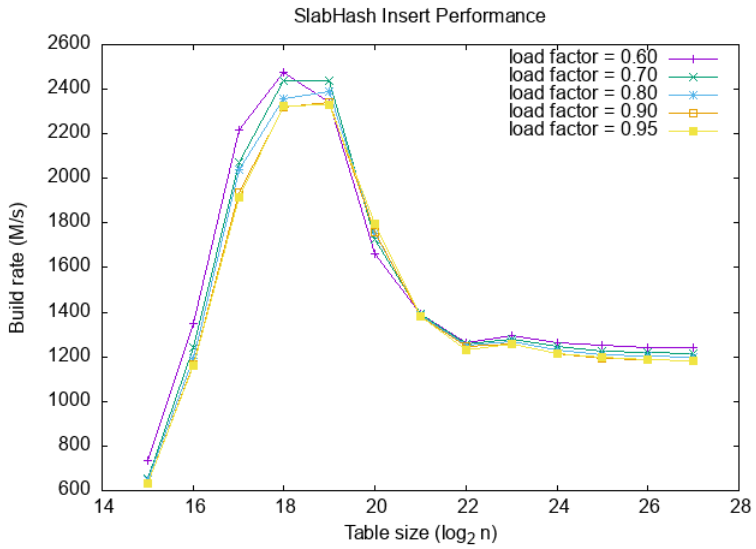

Figure 3: Performance of SlabHash insert function in millions of operations per second for different load factors and different table sizes

- Load factor has a relatively minor impact on performance. For example, the search throughput rates for varying load factors are with 2% of each other for a table with 2^{20} items. The same variation is 8% for insert operations.
- Performance does vary significantly as the size of the table varies. The variation is as large as 100% for insert operations and 50% for search operations over the range of table sizes tested.

5.2 Simple Hash

Figures 4 and 5 are the performance charts for Simple Hash. Simple Hash does not share SlabHash's resilience to changes in load factor. This is to be expected since its linear probing algorithm may require that a large proportion of the hash table be scanned. The fewer empty positions in the table there are the longer an insert or search operation may have to probe.

It is interesting, however, to observe that Simple Hash's throughput for smaller tables with smaller load factors is higher than that of SlabHash. For example, for a table of 2^{18} items with a load factor of 0.60 SlabHash has an insert rate of 2.5 billion per second and a search rate of 5.1 billion per second. Compare this to 3.5 and 10.2 respectively for Simple Hash.

This strongly suggests that a hybrid algorithm or an improvement to SlabHash for smaller tables could result in significantly better performance.

5.3 Hopscotch Hash

Figures 6 and 7 are the performance charts for Hopscotch Hash. For tables of size up to 2^{20} items Hopscotch Hash shows search rates that are typically better than those of SlabHash and Simple Hash. A typical result is for a load factor of 0.80 on a table of size 2^{18} where the Hopscotch Hash search rate is 7.4 billion operations per second compared to 5.0 for both SlabHash and Simple Hash.

Load factor does have an impact on Hopscotch Hash search performance. The worst case occurs for a table of 2^{17} items where the search rate is reduced from 11.2 billion operations

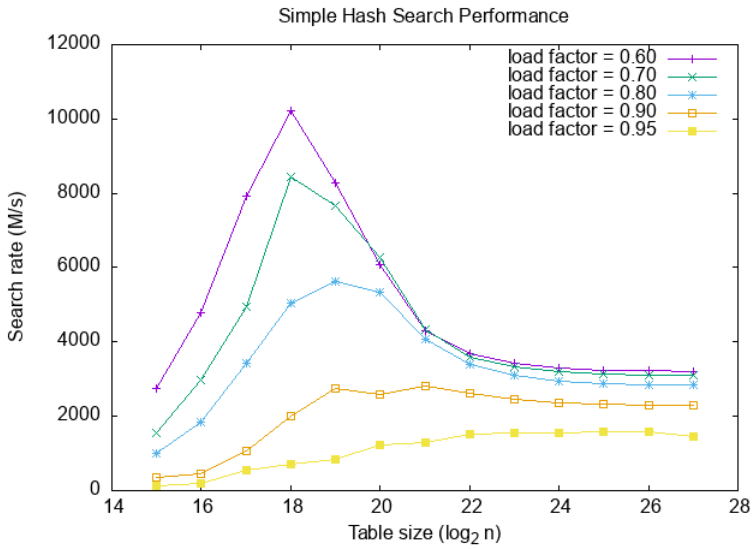

Figure 4: Performance of Simple Hash search function in millions of operations per second for different load factors and different table sizes

Hopscotch Hash insert performance has a very different performance profile when compared to SlabHash and Simple Hash. It scales very well with table size but the aggregate throughput for Hopscotch Hash insert is much lower than either of the other two table types. Further work is required to improve this performance.

6 Conclusions

In this paper we've described two hash table implementations that we undertook to further study the general problem of implementing dynamic data structures on a GPU. Simple Hash, as the name says, is a very simple hash table that is suitable for learning and discovery. Hopscotch Hash is an implementation of a hash algorithm that shows performance benefits for searching when compared to SlabHash.

We conclude that a GPU implementation of a dynamic data structure that doesn't have a natural data-parallel structure is a challenging problem but solving it can result in significant performance benefits. Specifically hash tables are a good candidate for this and making the highest performance possible available to algorithms and applications that need a dynamic hash table will have positive impact.

SlabHash is the current state-of-the-art dynamic hash table for the GPU but in this project we've shown that there's an opportunity to improve its performance in tables of 2^{20} items or less when search is the predominant work load. A hybrid approach could readily achieve this: use Hopscotch Hash for smaller tables, use SlabHash for larger tables.

In the future we'd like to investigate further improvements to our Hopscotch Hashing implementation: improve the performance of insert operations, investigate the benefit of warp cooperative algorithms for searching and inserting, experiment with GPU shared memory for possible performance benefits.

We also observe that there are distinct theoretical computing model aspects to the concept of a

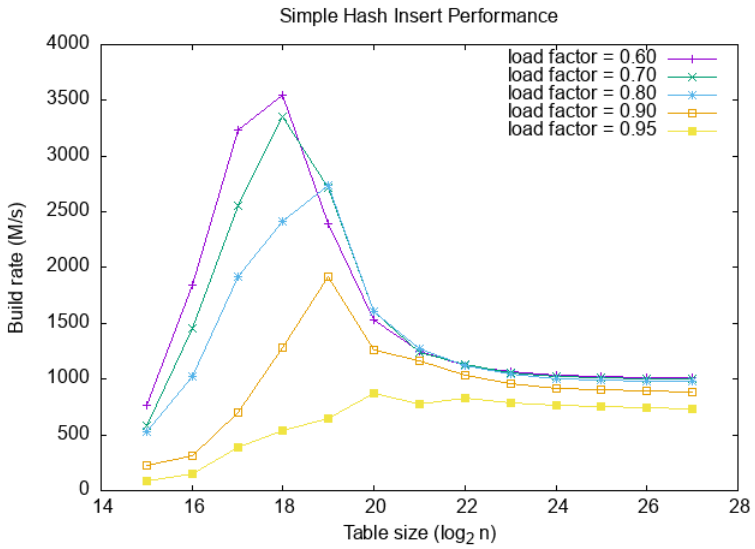

Figure 5: Performance of Simple Hash insert function in millions of operations per second for different load factors and different table sizes

warp cooperative algorithm. Certainly warp divergence is not a phenomenon we see in CPU-based algorithms, multi-threaded or otherwise. Proving the correctness of a warp cooperative algorithm (like a Hopscotch Hashing insert algorithm) seems to be a challenge that should be investigated.

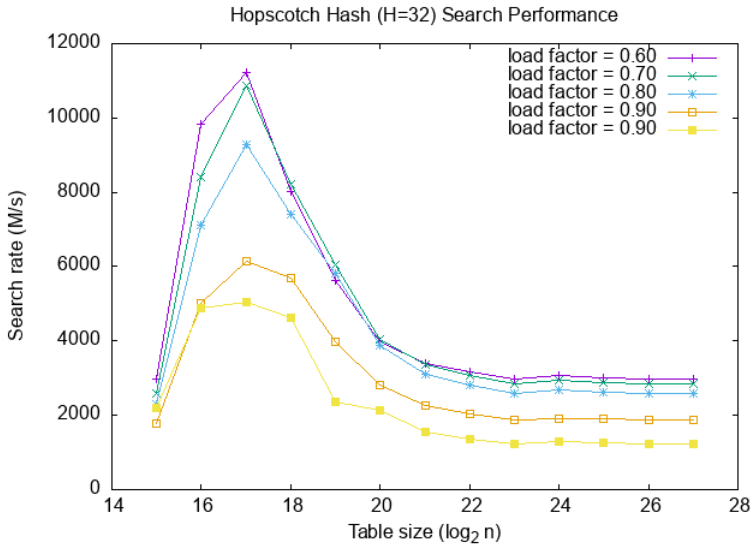

Figure 6: Performance of Hopscotch Hash search function in millions of operations per second for different load factors and different table sizes

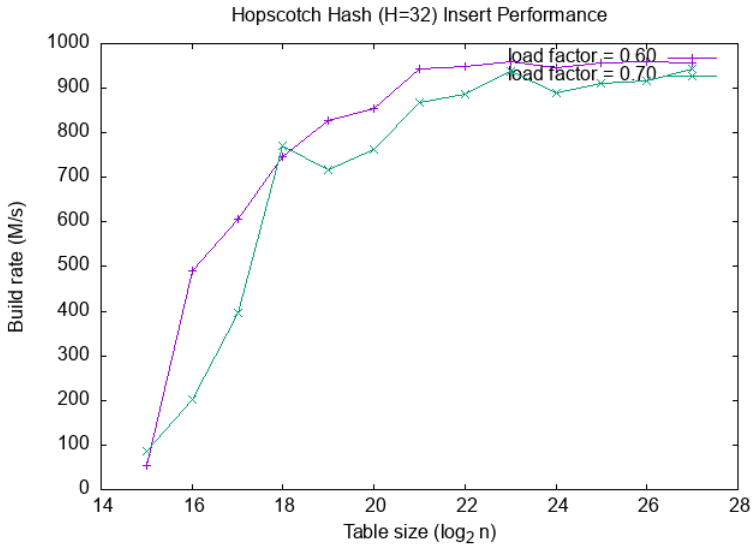

Figure 7: Performance of Hopscotch Hash insert function in millions of operations per second for different load factors and different table sizes

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, 2018.

[2] Muhammad Awad and S. Ashkiani. Slabhash. https://github.com/owensgroup/SlabHash, 2018.

[3] J. Bonwick. *The slab allocator: an object-caching kernel memory allocator*, 1994. USENIX Summer Technical Conference, 1994.

[4] Thomas H. Corman et al. *Introduction to Algorithms, 3rd Edition*. The MIT Press, Cambridge, Massachusetts, USA, 2009.

[5] NVIDIA Corporation. *CUDA C++ Programming Guide v11.1*, 2020. available at https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[6] L. Gao, Y. Xu, C. Xu, R. Wang, H. Yang, Z. Luan, and D. Qian. Towards a general and efficient linked-list hash table on gpus. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1452–1460, 2019.

[7] Oded Green. HashGraph – Scalable Hash Tables Using A Sparse Graph Data Structure. *arXiv e-prints*, page arXiv:1907.02900, July 2019.

[8] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch Hashing. In Gadi Taubenfeld, editor, *Distributed Computing*, pages 350–364, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[9] Daniel Jünger, Robin Kobus, André Müller, Christian Hundt, Kai Xu, Weiguo Liu, and Bertil Schmidt. WarpCore: A Library for fast Hash Tables on GPUs. *arXiv e-prints*, page arXiv:2009.07914, September 2020.

[10] D. Jünger, C. Hundt, and B. Schmidt. Warpdrive: Massively parallel hashing on multi-gpu nodes. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 441–450, 2018.

[11] Brenton Lessley, Shaomeng Li, and Hank Childs. HashFight: A Platform-Portable Hash Table for Multi-Core and Many-Core Architectures. *Electronic Imaging*, 2020(1):376–1–376–13, 2020.

[12] CUDPP 2.0 CUDA Data-Parallel Primitives Library. *Overview of CUDPP hash tables*, 2010. Available at http://cudpp.github.io/cudpp/2.0/hash_overview.html.

[13] RAPIDS Open GPU Data Science. *Welcome to cuDF's documentation!*, 2020. Available at https://docs.rapids.ai/api/cudf/stable/.