GPU Accelerated Concurrent Hash Table in Web Browser

Abstract

In this project, I implement and test concurrent hash tables with the experimental WebGPU API. My implementation is dynamic and outperforms the JavaScript built-in data structure by 13 times.

Keywords: GPU, Web Application, Hash Table, Parallel Programming

1 Introduction

In recent years, web applications have seen a huge increase in popularity due to their great portability and ease of deployment. From simple applications such as calculators, to-do lists and notepads to more complex applications such as business intelligence, 3D games, and even deep learning in web browsers (Smilkov, et al., 2019), they have been built and deployed everywhere. Many traditional desktop applications are replaced by them.

As web applications become more powerful and featureful, achieving good performance has become a new challenge. Typically, web applications offload heavy computation tasks to remote servers and only render the final result on the client device. However, for modern real-time applications such as real-time pose estimation and virtual reality, it would not be feasible due to the large amount of data and the network latency. In such cases, it is more desirable to take full advantage of the computing power at client side and run the computations in parallel inside the web browser.

The ideas of parallel computing and concurrent data structures are still relatively new in the world of web programming. Traditionally, JavaScript, which was the only programming language for web frontend development, did not support multithreading. While the runtime of JavaScript could handle multiple I/O requests initiated by JavaScript code in parallel (Mozilla, n.d.), the JavaScript code itself could not run in parallel. The situation has been changed in the last few years with the introduction of new web standards. WebAssembly defines a new low-level language that can be used as a compilation target, allowing the browsers to run code written in multiple languages at near native speed (Rossberg, et al., 2018). Web Worker (Hickson, 2021) allows JavaScript code to run parallely in the background. These new standards have enabled the web applications to utilize multi-core CPUs more effectively. Most recently, a future web API called WebGPU (Malyshau, Ninomiya, & Fan, 2021) has been proposed by Apple, Mozilla and Google to “expose the capabilities of GPU hardware for the Web”. The new API features support for compute shaders, allowing frontend applications to utilize GPU for tasks beyond graphics processing.

As an experimental standard under active development, WebGPU has not been widely adopted. There are only a few examples that utilize it to accelerate matrix multiplication as of today. In this project, I explore the feasibility of building concurrent data structures with WebGPU. My main contributions are implementing both a linear

probing and a separate chaining dynamic concurrent hash tables with WebGPU and comparing their performance with other popular hash table implementations for web frontend.

2 Related Works

2.1 GPU-Accelerated Hash Tables

The idea of utilizing GPU to accelerate hash tables is not new. Over the past decade, multiple concurrent hash tables have been proposed and implemented for GPU. In this section, I review some of them, summarizing their approaches, strengths, and weaknesses.

2.1.1 GPU Cuckoo Hash Table

In 2009, a GPU-accelerated hash table was proposed by Alcantara et al, and later implemented as part of the CUDA Data-Parallel Primitives Library (CUDPP, n.d.). The hash table uses Cuckoo hashing for resolving collisions.

In Cuckoo hashing (Pagh & Rodler, 2009), the hash table is divided into several smaller sub-tables of equal size and each of them has a different hash function. This provides multiple possible locations for insertion into the hash table for each key. During insertion, the key can be inserted in any sub-table if it is empty. If all of them are occupied, the current keys can be evicted and placed in other sub-tables. The advantage of Cuckoo hashing is that it guarantees constant-time lookup in the worst case. For insertion, it also only needs a single atomic operation in the best case.

In the CUDPP implementation, up to 5 hash functions can be used to build the hash table. Keys are inserted into the first table, evicting inserted keys in case of collision. Evicted keys are in turn inserted in the second table, then from the second to the third, and so on. The implementation achieves good performance. In a benchmark done by Ashkiani et al. (2018), the CUDPP implementation achieved a peak performance of 600 million insertions per second on a data center GPU. However, as a static hash table, its practicality is limited. No keys can be added or removed beyond the initial hash table construction. Rehashing is also not implemented, which makes the initial hash table construction prone to failure if the load factor is high.

2.1.2 Coherent Parallel Hashing

García et al. (2011) proposed a hash table optimized for sparse spatial data which uses Robin Hood hashing technique for collision handling. In Robin Hood hashing (Celis, 1986), the probe sequence lengths are tracked, and a new key can displace an old key when the probe sequence length of the new key is larger than that of the existing key, thus reducing the worst-case search time of the hash table.

In García et al.'s implementation, a spatial hash function is used to improve the memory locality. The Robin Hood hashing techniques also reduce the probing distance when the load factor is high. The algorithm addresses the construction failure issue of CUDPP implementation and is faster than CUDPP implementation under high load factor. This makes the algorithm more space efficient for storing sparse spatial data. However, its peak performance is not as good as Cuckoo hashing due to the overhead of maintaining the probe sequence lengths.

2.1.3 SlabHash

SlabHash is a concurrent dynamic hash table implemented in CUDA proposed by Ashkiani et al. in 2018. The hash table uses separate-chaining techniques to handle collisions, but unlike traditional hash tables which uses linked list for chaining, SlabHash utilizes slab list data structure, which is a modified linked list structure with each node (slab) storing multiple keys and one pointer to the successor slab. The size of each slab is fixed and determined by the hardware architecture of the GPU.

The motivation of SlabHash is to leverage the SIMD compatibility of GPU and reduce the memory access cost. With a good choice of slab size, the threads within each SIMD unit (warp) performs a coalesced memory access for each slab and the search within a slab can be completed with a single SIMD instruction. Therefore, they propose a warp-cooperative work sharing strategy that reduces the cost of memory access and reduces branch divergence. The fixed size of each slab also enables Ashkiani et al. to design a more efficient dynamic memory allocator for allocating slabs compared with CUDA's built-in malloc.

In the benchmarks, Ashkiani et al. show that their method achieves similar performance to the CUDPP implementation on a data center GPU, but the dynamic property allows insertion and deletions after the hash table is constructed, making it more desirable for applications that needs to make updates to the hash table beyond initial construction.

2.1.4 Simple GPU Hash Table

Simple GPU Hash Table is implemented by Farrell (2020). The implementation is a direct porting of the linear probing lock-free concurrent hash table similar to Assignment 4 into CUDA. It is a dynamic hash table which supports both insertions and deletions.

With this simple implementation, it still achieved good performance on GPU with over 285 million insertions per second under 50% load factor on a laptop GPU. However, without the support of resizing, the performance degrades drastically as the load factor increases.

2.2 Hash Tables in Web Browser

In this project, I benchmark my hash table implementation against several widely used hash table packages in JavaScript. In this section, I briefly summarize their implementations.

2.2.1 Built-in Data Types

In JavaScript, built-in dataset such as Map and Set (Mozilla, n.d.) are implemented by the runtime. The standard specifies them to be implemented by "either hash tables or other mechanisms that, on average, provide access times that are sublinear on the number of elements in the collection" (Ecma International, 2015). However, their performance may not be desirable because they support arbitrary types of keys. The Map type also needs to keep track of the insertion orders as specified by the standard.

2.2.2 SharedMap

SharedMap (Momtchev, n.d.) is a concurrent hash table implemented in vanilla JavaScript backed by SharedArrayBuffer. It implements coalesced-chaining collision resolving and uses fine-grained synchronous locking implemented on top of the Atomics

interface. It supports dynamic insertions and deletions and can perform over 1 million operations per second. However, it does not support resizing and its performance decreases drastically when the load factor is greater than 90%.

2.2.3 Fast, Reliable Cuckoo Hash Table

Fast, Reliable Cuckoo Hash Table (Ronomon, n.d.) is an implementation of the Cuckoo hashing technique in JavaScript. Despite being a single-threaded implementation, it achieves a throughput of over 5 million operations per second according to the benchmark data provided by the author.

3 Implementations

In this section, I discuss the technical choices and implementation details of my concurrent GPU hash tables. I start with an overview of the WebGPU API and its current limitations, discussing how they impact my implementation decisions. Then, I discuss the implementation details of my hash tables.

3.1 WebGPU API

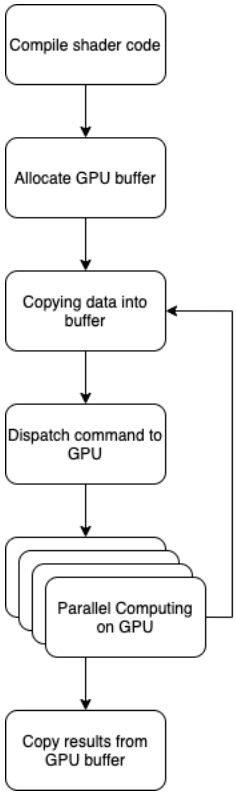

Figure 1

The general workflow of executing computation tasks with WebGPU is illustrated in Figure 1. First, the compute shader code, which is written in either OpenGL Shading Language (GLSL) or WebGPU Shading Language (WGSL), is compiled into binary data on the fly in the web browser. Then, a buffer in GPU memory is allocated for the

computation task and mapped so that the data for this task can be copied from main memory. After that, the dispatch function is called to run the shader code on the GPU. When all the dispatched work is completed, the program may either put more data into the GPU memory and run another pass of the computation, or simply copy the result from GPU memory back to main memory and deallocate GPU buffer if no more computation task needs to be performed.

In the dispatch call, the caller cannot pass custom data to the GPU or specify the number of threads for this computation task. Instead, the caller only specifies the "grid size" with up to three dimensions x, y, and z and an entry point (main function). The shader code will only have access to its position within the grid and is supposed to load and store data into the shared buffer based on this position. This approach simplifies configurations for embarrassingly parallel workloads such as matrix multiplications. However, it also greatly limits the potential of fine-tuning algorithms according to GPU architecture. As of today, the WebGPU standard does not specify how the dispatch call distributes work to GPU threads or the order of dispatching when the workload is greater than the number of GPU threads. Therefore, when designing the algorithm, I have to assume the work within one dispatch can be dispatched to any GPU thread and executed in any order.

The other limitation of WebGPU is that the GPU buffer allocation cannot be done inside the compute shader. In the workflow shown in Figure 1, the GPU buffer has to be allocated by the JavaScript before compute shader executions. Due to this limitation, the resizing has to be initiated by the JavaScript code and cannot be done during the operations on the hash table. In my implementation, I keep track of the number of insertions and deletions with atomic counters and copy them to the main memory at the completion of each dispatch. Then, I use this information to decide the current load factor of the hash table and whether a resizing is required. However, in some cases, the resizing is initiated after the performance starts to degrade. While setting a low resizing threshold could improve the performance in such cases, it also makes resizing more frequent and causes the average throughput to drop.

As a web API, WebGPU is an open standard that works on GPUs from different vendors. However, this portability comes at a cost of performance. In (García, Lefebvre, Hornus, & Lasram, 2011), a query function has been implemented in GLSL for their graphics application. The benchmark shows that the GLSL implementation can be up to 50% slower than CUDA. To achieve high performance across a broad range of devices, WGSL sometimes permits several possible behaviours for a given feature. However, as of today, it has not been fully implemented. Specifically, the atomic types proposed by WGSL are not available on both Google Chrome and Firefox browsers, making it unusable for the purpose of this project. Thus, in this project, I have to use GLSL to implement my hash tables.

3.2 Linear Probing Hash Table

Due to the limitations as mentioned in Section 3.1, I am inspired by Simple GPU Hash Table, which is an easy to implement dynamic hash table that does not use any hardware-specific optimization but still performs well on consumer-grade GPU when the load factor is low. In this implementation, I simply port the lock-free concurrent hash table implemented for Assignment 4 into GLSL.

To address the issue of performance degrading when the load factor is high, I added blocking expansion. When the JavaScript code decides to initiate a resizing, it allocates a new GPU buffer and uses all the GPU threads for resizing, blocking any operations until the resizing is completed. While blocking during resizing is not desirable for performance, this approach makes memory reclamation much easier, because the old hash table buffer is deallocated immediately after the resizing is done.

3.3 Slab ListHash Table

The original motivation of implementing the slab list hash table is to reproduce the great result of SlabHash (Ashkiani, Farach-Colton, & Owens, 2018) on WebGPU. However, as I try to port the algorithm into GLSL, I discover that the warp-cooperative optimizations they propose cannot be implemented in GLSL. Therefore, while I use slab lists instead of traditional linked lists, in my implementation, each GPU thread operates independently without communication. Without warp-cooperation, my implementation does not have the benefits of coalesced memory access or branch divergence reduction, but I still expect it to perform better than traditional linked list-based approaches due to better memory locality.

The insertion is performed by traversing down the slab list and finding an empty spot. If the entire list is filled, a new slab is allocated and linked to the last slab by compare-and-swapping (CAS) the next pointer with the address of the newly allocated slab. If the CAS is successful, then the newly allocated slab is inserted to the slab list by the current thread. Otherwise, another thread has inserted a new slab and the current thread must deallocate the slab. Similar to insertion, deletion is performed by traversing down the slab list, finding the key and CAS it with “tombstone”.

To make this algorithm work, a dynamic slab allocator is required. However, in GLSL, pointers are not supported, and GPU buffer allocation cannot be performed. Therefore, I end up using an array as block memory and the index of the array as address. Since the coalesced allocation within a warp cannot be implemented due to the current limitations of WebGPU, I implement a naïve lock-free slab allocator, which keeps a global array of free slabs. For slab allocation, it traverses down the array, reads its value and tries to CAS it with null when a free slab is found. If the CAS is successful, the slab is allocated, otherwise, it will keep traversing down the array until a new slab is found. Deallocating is also simple by traversing down the array and finding an empty spot, attempting to CAS it with the deallocated address, and keep going until the CAS is successful. This approach causes high contention when multiple threads are trying to allocate new slabs simultaneously. However, with a slab size of 32, I expect slab allocation to occur infrequently.

4 Experiments

Since the WebGPU hash tables are expected to be executed in web browsers on consumer-grade devices, I run the experiments on my laptop in order to evaluate the performance. The configuration of my laptop is listed in Table 1 below. The browser I use is Google Chrome Canary 92.0.4489.0 on macOS.

CPU	Intel Core i9, 2.3 GHz, 8 Cores
RAM	32 GB DDR4 (42.6 GB/s)
GPU	Radeon Pro 5500M, 1.3GHz, 4 GB GDDR6 (192 GB/s), 1536 shaders

Table 1

4.1.1 Load Factor Experiment

In this experiment, the goal is to compare the performance of my hash table implementation under different load factors. Therefore, resizing is disabled. The hash table sizes are fixed to 20 million and are prefilled to the desired load factor. For slab list hash table, I use a fixed number of 62,5000 buckets, which corresponds to 1 slab per bucket on average. Then, 1 million insertion, deletion or query operations are performed and the average throughput is calculated. The keys are randomly generated integers with low contention. The results are collected and plotted as Figure 2, 3, and 4.

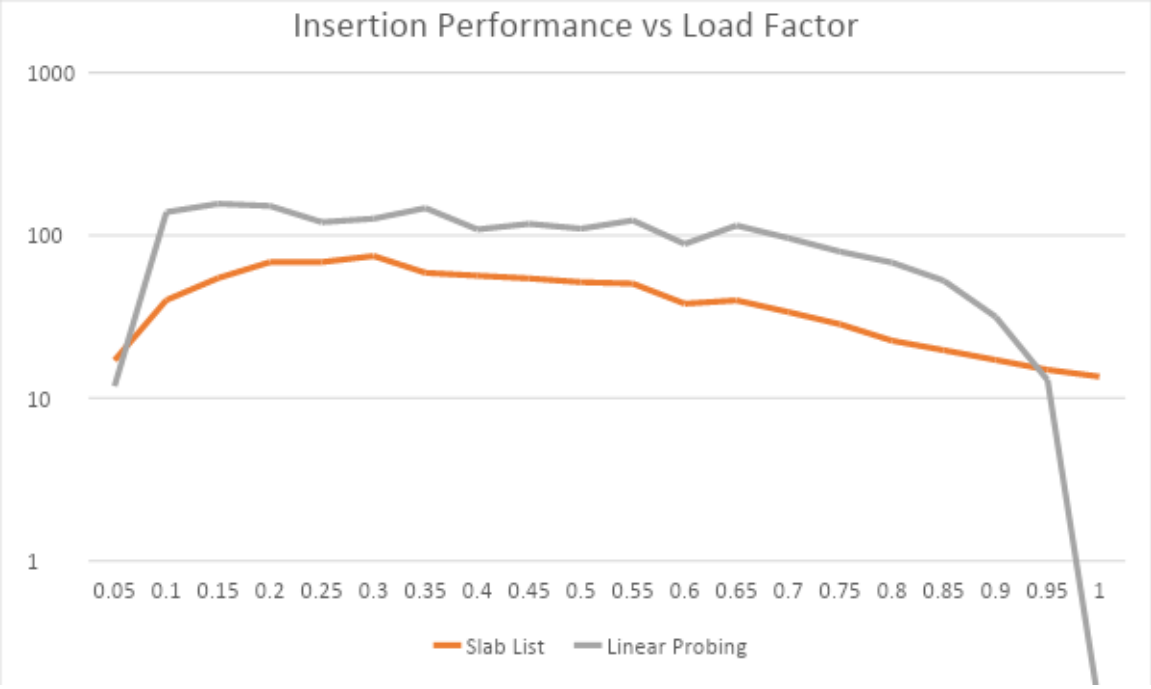

Figure 2

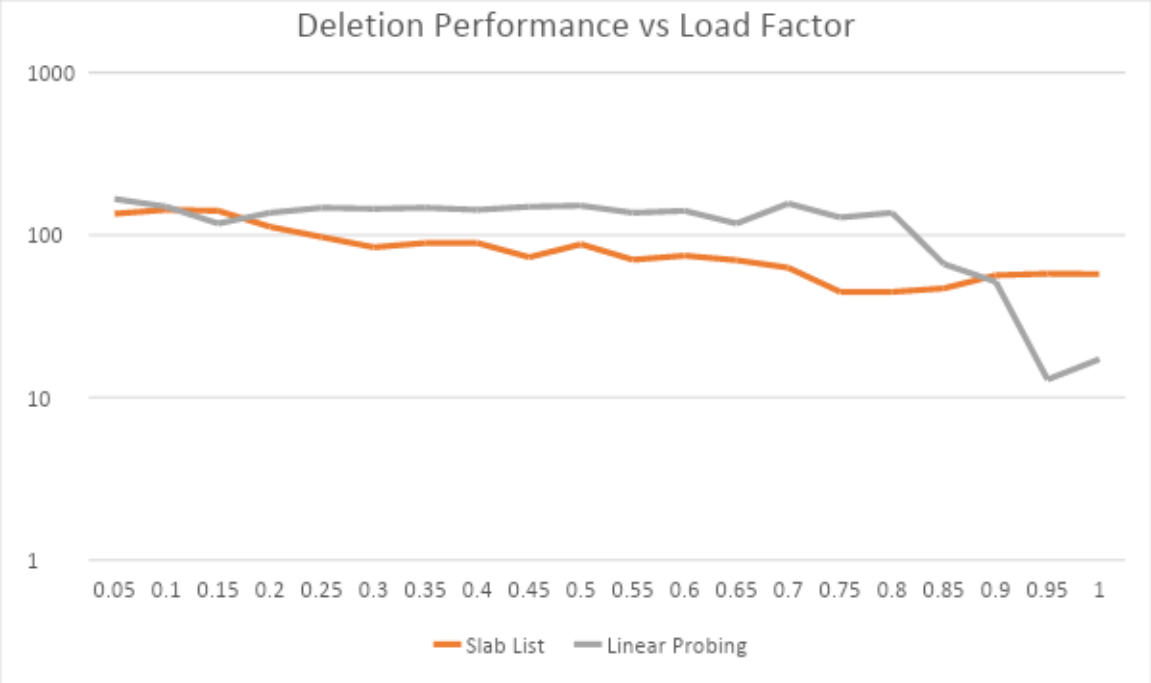

Figure 3

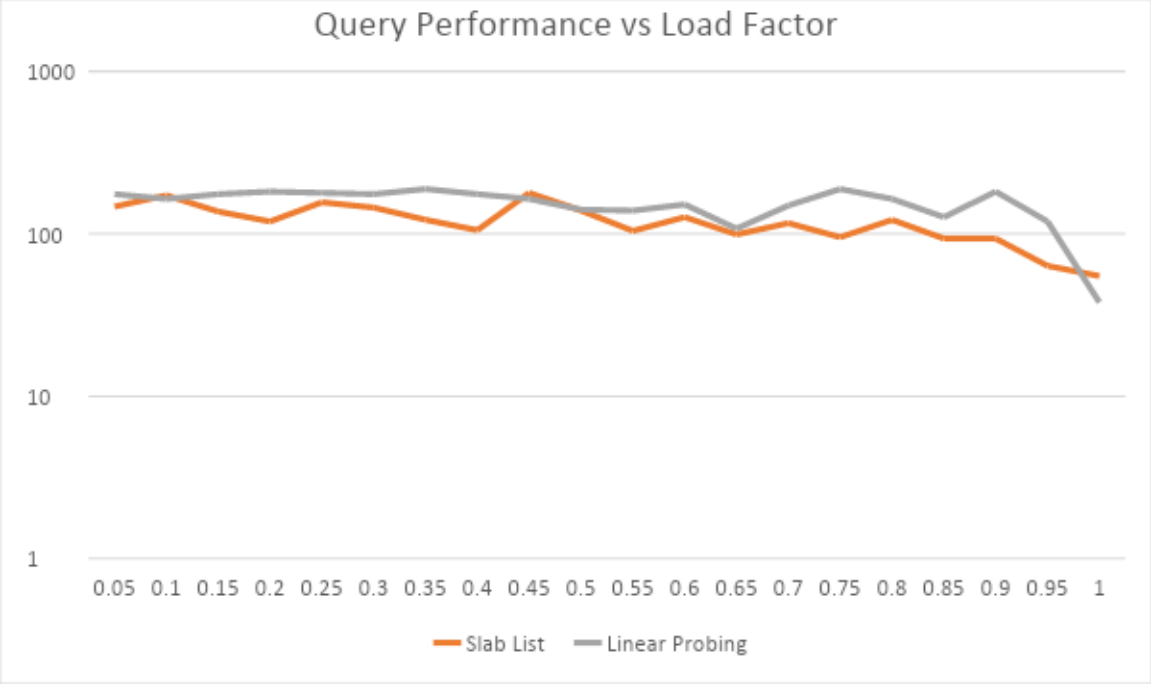

Figure 4

4.1.2 Resizing Experiment

In this experiment, I test the resizing capability of the hash table and its impact on the performance. In this experiment, I run an all-update workload of 50% insertions and 50% deletions over a hash table with an initial capacity of 20 million. For slab list hash table, I

use a fixed number of 62,5000 buckets. When the loader factor of the hash table is greater than the resize threshold, the migration is initiated. For this experiment, I set the migration factor to 2, which means that the hash table is migrated to a new hash table with size of (#insertions - #deletions) × 2.

In this experiment, the workload is executed continuously for 10 seconds with a dispatch size of 1 million, and the throughput data is uniformly sampled with an interval of 500 millisecond from the dispatches. The results are collected and plotted as Figure 5 and 6.

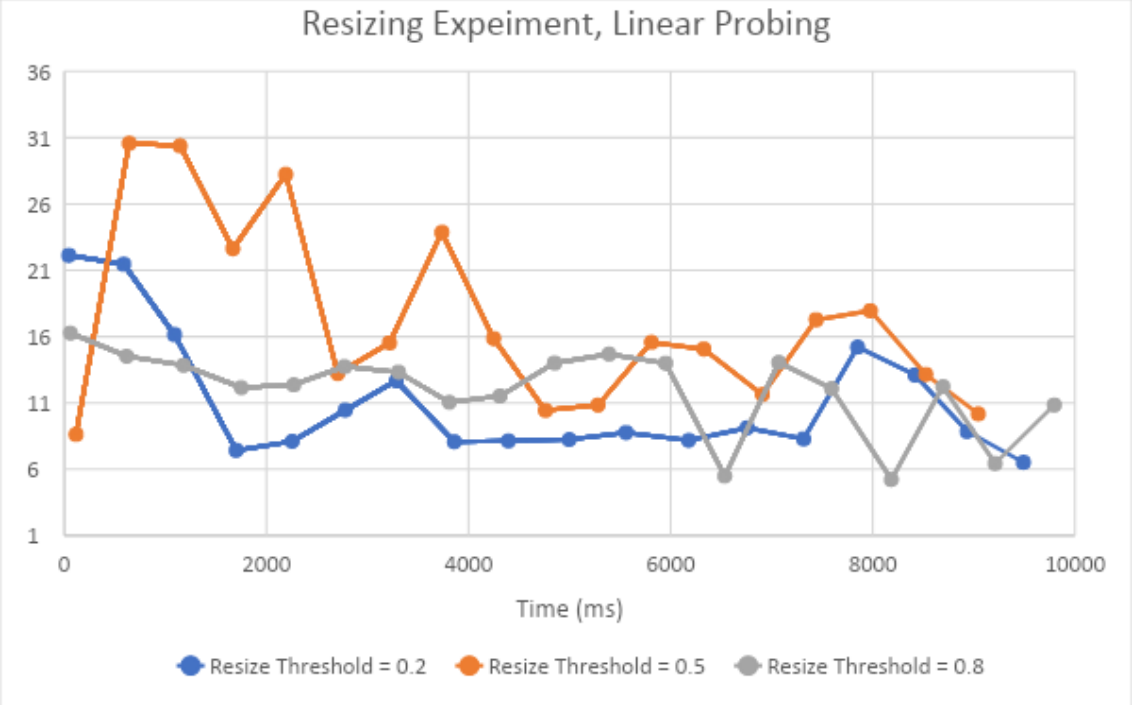

Figure 5

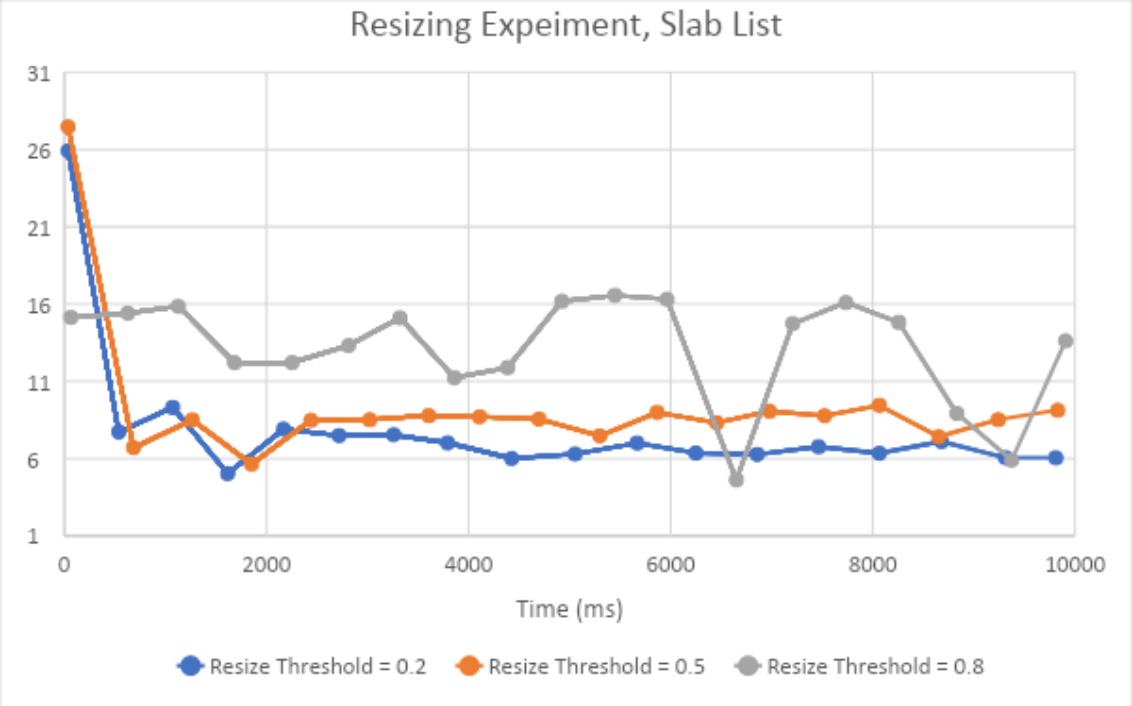

Figure 6

4.1.3 Comparison with Baselines

To compare my implementation against baselines, I run a mixed workload with 10 million operations, the workload consists of 30% insertions, 30% deletions, and 40% of lookups. In this experiment, the capacity of both hash tables is fixed to 20 million and resizing is disabled. After each 1 million operations, the throughput data is collected. In Table 2, the peak throughput and average throughput are listed.

Method	Peak Throughput (ops/s)	Average Throughput (ops/s)
Native Map	10.4 M	5.57 M
Native Set	15.8 M	7.33 M
Fast, Reliable Cuckoo Hash Table	5.51 M	4.47 M
SharedMap	1.69 M	893 K
Linear Probing	**149 M**	**100 M**
Slab List	74.1 M	43.6 M

Table 2

5 Discussions

Overall, the GPU-accelerated hash tables perform pretty well. It is about 13X faster than the built-in hash set and 18X faster than the built-in hash map. It also outperforms all the other baselines implemented in JavaScript.

When comparing the performance between the linear probing and the slab list implementation, their performance differs most significantly in terms of insertion. This is most likely caused by the contention of memory allocator as discussed in Section 3. The performance of the linear probing hash table also degrades more significantly when the load factor is high, which is caused by the long probing sequence when the table is almost full. On the other hand, the slab list approach has better performance in this case at a cost of more memory consumption.

The implementation of resizing leads to a more general hash table that could support continuous insertions and deletions. However, the cost of resizing is high for slab list hash tables due to the high cost of insertion. In the experiment, the performance of the slab list hash table degrades as a smaller resizing threshold is set. On the other hand, the linear probing hash table works better with a smaller resizing threshold due to its huge performance degradation when the load factor is high.

6 Conclusions and Future Work

In this project, I implement and test concurrent hash tables with the experimental WebGPU API. The result is a portable, dynamic and concurrent hash table implementation that runs in web browsers without the requirement of additional configuration. As my implementation outperforms JavaScript implementations by over 10X, there is a huge potential to bring other concurrent algorithms and data structures into web applications by leveraging GPU acceleration.

In terms of the hash table implementation, there are still many optimization techniques worth trying. For linear probing hash tables, the Robin Hood technique may be able to shorten the probing distance and improve its performance under high load factor. For slab list hash tables, the first priority is to improve the efficiency of the memory allocator.

As a future web standard, WebGPU is still under active development. If more information is available regarding its dispatch algorithm and how it maps a computation task to GPU shaders, fine-grained, hardware-specific optimization may further boost the performance of the concurrent hash tables.

7 Works Cited

Alcantara, D. A., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J. D., & Amenta, N. (2009). Real-Time Parallel Hashing on the GPU. *ACM Transactions on Graphics*.

Ashkiani, S., Farach-Colton, M., & Owens, J. D. (2018). A Dynamic Hash Table for the GPU. *Proceedings of IEEE International Parallel and Distributed Processing Symposium.* Vancouver, BC, Canada: IEEE.

Celis, P. (1986). *Robin Hood Hashing.* Retrieved April 2021, from University of Waterloo: https://cs.uwaterloo.ca/research/tr/1986/CS-86-14.pdf

CUDPP. (n.d.). *Overview of CUDPP hash tables*. Retrieved April 2021, from CUDA Data-Parallel Primitives Library: http://cudpp.github.io/cudpp/2.0/hash_overview.html

Ecma International. (2015, June). *ECMAScript® 2015 Language Specification*. Retrieved April 2021, from Ecma International: https://262.ecma-international.org/6.0/#sec-map-objects

Farrell, D. (2020, March 8). *A Simple GPU Hash Table.* Retrieved April 2021, from Nosferalatu: https://nosferalatu.com/SimpleGPUHashTable.html

García, I., Lefebvre, S., Hornus, S., & Lasram, A. (2011). Coherent Parallel Hashing. *ACM Transactions on Graphics*, *30.*

Hickson, I. (2021, January 28). *Web Workers W3C Working Group Note.* Retrieved April 2021, from World Wide Web Consortium: https://www.w3.org/TR/workers/

Malyshau, D., Ninomiya, K., & Fan, J. (2021, April 24). *WebGPU Editor's Draft.* Retrieved April 2021, from WebGPU: https://gpuweb.github.io/gpuweb/

Momtchev, M. (n.d.). *SharedMap*. Retrieved April 2021, from SharedMap: https://mmomtchev.github.io/SharedMap/

Mozilla. (n.d.). *Concurrency model and the event loop*. Retrieved April 2021, from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

Mozilla. (n.d.). *Map*. Retrieved April 2021, from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Mozilla. (n.d.). *Set*. Retrieved 2021 April, from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

Neto, D., Maxfield, M. C., & Sinclair, D. (2021, April 26). *WebGPU Shading Language* . Retrieved from WebGPU Shading Language : https://gpuweb.github.io/gpuweb/wgsl/

Pagh, R., & Rodler, F. F. (2009). Cuckoo Hashing. *Advanced Data Structures Seminar* .

Ronomon. (n.d.). *hash-table*. Retrieved April 2021, from GitHub: https://github.com/ronomon/hash-table

Rossberg, A., Titzer, B. L., Haas, A., Schuff, D. L., Gohman, D., Wagner, L., . . . Holman, M. (2018, December). Bringing the Web Up to Speed with WebAssembly. *Communications of the ACM, 61*(12), 2. Retrieved April 2021, from WebAssembly: https://webassembly.org/

Smilkov, D., Thorat, N., Assogba, Y., Yuan, A., Kreeger, N., Yu, P., . . . Mong. (2019). TensorFlow.js: Machine Learning for the Web and Beyond. *SysML 2019.* Palo Alto, CA.