FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Parallel algorithms for data hashing on GPUs
Student:	Bc. Jan Groschaft
Supervisor:	doc. Ing. Tomáš Oberhuber, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

Hashing is one of the most important operations when developing efficient programs. In conventional sequential programming, hashing is already a relatively well-developed topic. However, when it comes to parallel hashing, especially on the GPU, there are still many open problems to be solved. The aim of the thesis is to study the existing work of hashing on the GPU and implement selected algorithms in the TNL library. More specifically, these are the following points:

1. Study the current implementation of the HashGraph algorithm in the TNL library and implement it using structures called Segments from the TNL library.
2. Implement the optimized version of the HashGraph algorithm and compare it with the original version. Alternatively, design your optimizations.
3. Focus mainly on the possibility of implementing the functionality of inserting and dynamically growing the hash table. Design different solutions and compare them.
4. Test the implementation with a set of unit tests and compare the implementation with other GPU hashing algorithms such as HashGraph or WarpCore.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 31 August 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Parallel algorithms for data hashing on GPUs

Bc. Jan Groschaft

Department of theoretical computer science
Supervisor: doc. Ing. Tomáš Oberhuber, Ph.D.

May 4, 2022

Acknowledgements

I would like to thank my family and friends for support during writing this thesis.

Declaration

In Prague on May 4, 2022 …………………

Citation of this thesis

Groschaft, Jan. *Parallel algorithms for data hashing on GPUs.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce se zabývá problematikou hešovacích tabulek na grafických kartách využívajících technologii CUDA. Je zaveden potřebný teoretický základ, ve kterém je kladen důraz na datovou strukturu HashGraph, propojující hešování a teorii grafů. V práci jsou prezentovány možnosti reprezentace struktury HashGraph v paměti pomocí datových struktur pro práci s řídkými maticemi. Je uvedena upravená verze struktury HashGraph podporující dynamické přidávání a mazání prvků. Popsané datové struktury jsou implementovány v jazyce C++ a TNL knihovny. Implementace jsou porovnány s vybranými volně dostupnými hešovacími tabulkami.

Klíčová slova Hešovací tabulky, GPU, CUDA, HashGraph, TNL, Template Numerical Library.

Abstract

In this thesis, we study hash tables on GPUs using CUDA technology. The necessary theoretical concepts are introduced, with emphasis on the Hash-Graph data structure, which connects hashing to graph theory. The possibilities of using different sparse matrix storage formats to represent HashGraph are analyzed. A modified version of the HashGraph data structure is introduced, providing support for dynamic insertion and removal of elements. The described data structures are implemented in the C++ language using the TNL library. Implementations are compared against selected freely available hash tables.

Keywords Hash tables, GPU, CUDA, HashGraph, TNL, Template Numerical Library.

Contents

List of Figures

List of Tables

Introduction

Motivation

Hash tables are data structures used primarily to represent sets or associative arrays. Data is stored and retrieved using a hash function, which maps individual elements to their probable location inside the data structure. For certain hash functions, data can be stored and retrieved in expected constant time. Because of this, hash tables are widely used and studied.

In recent years, many-core systems have been increasingly popular for their ability to process massive amounts of data in parallel. One such system is the *Graphics Processing Unit (GPU)*. While GPUs have been originally designed to handle graphics rendering, they have since evolved into general-purpose programming platforms suitable for high-performance computing. This gave rise to new kinds of hash tables suitable for GPUs and focusing on highly parallel workloads.

Goals

The goal of this thesis is to study the HashGraph [1] data structure and implement it using the Template Numerical Library. Special attention should be paid to analyzing the possibility of using a data structure called *Segments* as the internal storage for HashGraph.

The original HashGraph paper introduces a standard and an optimized algorithm for building the hash table. Both should be implemented and compared as a part of this work. The main focus is on exploring possibilities of making the HashGraph a dynamic hash table, i.e., supporting the addition and removal of elements once the table has been constructed. Finally, the resulting implementation shall be tested for correctness and its performance shall be compared to state-of-the-art solutions.

Structure of Work

In chapter 1, we describe the hardware architecture of a modern GPU, and we provide an overview of the Template Numerical Library.

Chapter 2 introduces the necessary theoretical framework for analyzing hash tables and related concepts.

Comments on implementation choices follow in chapter 3, and, finally, we describe the correctness tests and experimental performance evaluation in chapter 4.

Chapter 1

Preliminaries

1.1 GPU Architecture

A *Graphics processing unit (GPU)* is a special type of processor designed to perform massively parallel operations. Whereas a standard CPU typically has a few general-purpose high-performance cores, GPUs tend to have many comparatively weaker cores, and are optimized for high throughput workloads.

As the name suggests, GPUs were originally designed as independent programmable processing units responsible for graphics manipulation and offloading the output rendering pipeline from the CPU. However, due to their high performance in certain workloads, they have more recently found use cases in general-purpose computing. Such GPU is referred to as a General-Purpose Graphics Processing Unit (GPGPU). Present-day applications of general-purpose GPU computing include, among others, large-scale numerical simulations, linear algebra processing, molecular dynamics, protein folding, signal processing, ray tracing, and machine learning [2].

1.1.1 Hardware Architecture

Let us briefly describe the high-level hardware architecture of a GPU and highlight some of the important differences between a CPU and a GPU. Throughout this work, we will be focusing on NVIDIA GPUs and their *CUDA (Compute Unified Device Architecture)* general-purpose parallel computing platform and programming model. Note that similar principles should apply to other solutions as well.

Since the invention of the transistor, CPUs have been getting exponentially more powerful. In the early days, the majority of performance improvements came from frequency increases and instruction-level parallelism (ILP). However, both approaches have started to become increasingly difficult to further

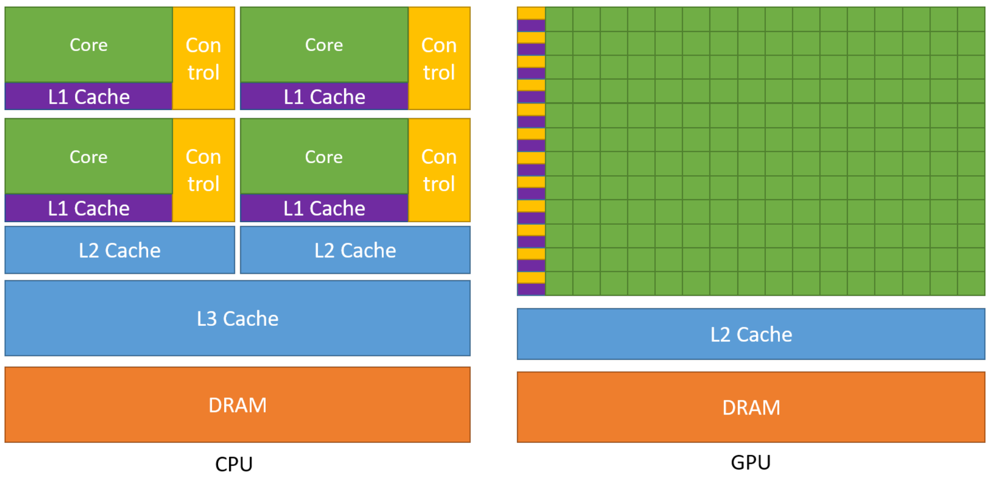

Figure 1.1: High level comparison of CPU and GPU architecture [3].

improve, mainly due to the high thermal output associated with high frequency and the difficulty to scale ILP in inherently sequential programs [4, page 5].

In the early 2000s, there was a paradigm shift towards multicore CPUs, which had the advantage of increasing the package power linearly with the number of cores while also only increasing the power requirements linearly[1]. Modern CPUs are increasingly becoming bottle-necked by the memory subsystem, especially in shared memory architectures, where cache consistency issues arise.

[1]Power requirements increase quadratically with increasing frequency [4, page 1].

In contrast, the GPU is composed of a number of *streaming multiprocessors (SMs)*, a shared L2 cache and shared DRAM. Each SM consists of

- *N streaming processors* (called *CUDA Cores* for NVIDIA GPUs) capable of executing threads
- Registers for the streaming processors
- Instruction cache
- Shared memory

The total number of CUDA Cores (and thus the number of threads that are able to run simultaneously) in a modern GPU is in the order of tens of thousands, in contrast with tens of cores in a typical modern CPU. The hardware differences between CPUs and GPUs can be seen in Figure 1.1.

A group of 32 threads makes up an execution unit called a *warp*. When a warp is scheduled by the SM to run on some particular set of CUDA Cores, all threads in this warp must always execute the same instruction. This architecture is called *Single Instruction Multiple Threads (SIMT)*, and it can be thought of as a generalization of the *Single Instruction Multiple Data (SIMD)* instruction set available in most modern CPUs.

1.1.2 Thread Hierarchy

GPU threads are arranged into groups of 32 called *warps*. Warps are then scheduled by the SM to run on some particular set of CUDA Cores. Individual threads composing a warp all start at the same program address, and all threads in a warp execute one common instruction at a time. All threads in a given warp must either execute the same instruction or be temporarily paused. This architecture is called *Single Instruction Multiple Threads (SIMT)*, and it can be thought of as a generalization of the *Single Instruction Multiple Data (SIMD)* instruction set available in most modern CPUs.

Importantly, when a branch is encountered in the execution path, and only a subset of the warp's threads take the branch, the rest of the threads must idly wait until the execution path is joined again. This is called *warp divergence*. Therefore, a warp is most efficient when all 32 threads agree on the execution path [3, page 111].

In CUDA, threads are logically arranged into *blocks*, which can be either one-, two-, or three-dimensional, and which are themselves arranged into a one-, two- or three-dimensional *grid*. Threads in the same thread block run on the same SM. There can be up to 1024 threads in a single block, and there can be up to $2^{32} - 1$ blocks in the grid. See subsection 1.1.4 for details on how these blocks are used by the programmer.

1.1.3 Memory Model

During execution, a CUDA thread may access several memory spaces, namely

- private local memory visible only to a given thread used to store local variables,
- shared memory local to an SM visible to all threads in a given block,
- global shared memory accessible from all threads,
- constant memory serving as a read-only cache
- texture memory for graphics-related applications

Much like in a CPU, the global memory has an L1 cache per SM and an L2 cache shared by all SMs. Threads also have access to a register file residing on an SM. For a graphical overview of the memory model, see Figure 1.2.

1.1.4 CUDA Programming Model

CUDA comes with a software environment that allows developers to use a high-level programming language, such as C, C++ or FORTRAN, to write programs for execution by the device. We will be describing the CUDA C++ language extensions. From now on, we shall refer to the system on which the GPU resides as the *host*, and to the GPU itself as the *device*.

1.1.5 Kernels

CUDA C++ introduces *kernel* functions that execute on CUDA threads on the GPU device. Those functions are defined using the `__global__` specifier and are callable from the host with the special call operator <<<`gridDim`, `blockDim`>>>. This call results in

$$n = blockDim.x * blockDim.y * blockDim.z * gridDim.x * gridDim.y * gridDim.z$$

threads being scheduled and eventually executed on some SM.

Each of the n threads can be uniquely identified by its thread index inside the block and the blocks' index inside the grid. Inside the function, individual threads can access those indices via the `threadIdx` and `blockIdx` variables automatically created by the CUDA runtime.

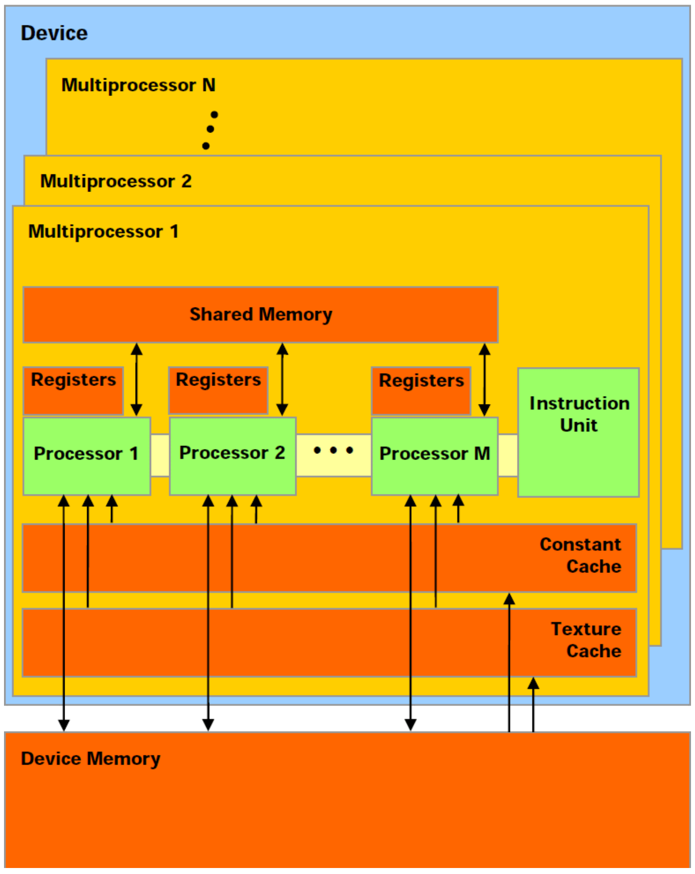

Figure 1.2: CUDA memory model [5].

1.1.6 Synchronization

Within a block, threads can communicate by sharing data through the shared memory, or alternatively—if the shared memory is too small—through the global memory. Threads inside a single block can be synchronized by the `__syncthreads()` function, which serves as a barrier on which arriving threads wait until all threads in the block reach the barrier.

1.1.7 Memory Management

Memory allocated on the host side cannot be used on the device and vice versa. To circumvent this, the programmer must typically first allocate the data on the host side, allocate the appropriate space on the device, copy the data from the host to the device, perform the required computations and finally copy the data back to the host. The CUDA framework provides `cudaMalloc`, `cudaFree` and `cudaMemcpy` functions for exactly those purposes[2].

[2]See [3] for a complete list of memory management options in CUDA

1.2 Template Numerical Library

The *Template Numerical Library (TNL)* is a collection of building blocks that facilitate the development of efficient numerical solvers and HPC algorithms [6]. It is developed at the Department of Mathematics, Faculty of Nuclear Science and Physical Engineering at Czech Technical University in Prague.

TNL is written in C++ and aims to provide unified memory management for CPUs and GPUs, along with many containers and algorithms optimized for highly parallel workloads. The main advantage of TNL is that it alleviates the programmer from manual and often tedious memory and thread management and instead provides a high-level interface with minimal runtime overhead.

1.2.1 TNL Containers and Views

Like in the Standard Template Library of C++, containers in TNL are generic implementations of abstract data types. Additionally, they are also generic over the type of device they are stored on. For example, the class template `TNL::Containers::Array<Value, Device>` implements a contiguous sequence of elements of type *Value* on a device *Device*.

As in STL, containers in TNL are *value types*, which implies that the copy constructor and the assignment operator both produce a deep copy of the given container. How do we pass an object of a type like `TNL::Containers::Array` from the host to a CUDA kernel? Creating a deep copy is not optimal since the internal pointer to the contiguous storage is already allocated on the correct device. We cannot pass a pointer nor a reference to such objects because it would be pointing to an address in the host's address space, which is unusable in the kernel.

TNL solves this by providing *View* types. Specifically, for our array example, there is `TNL::Containers::ArrayView<Value, Device>`. A view provides non-owning access to the underlying container, so copying the view object will result in a new view pointing to the original container. Views can be cheaply copied from the host to a device, but the user must ensure the lifetime of the view does not outlive the lifetime of the viewed container.

1.2.2 TNL Algorithms

TNL provides a variety of algorithms that are also generic over the type of device they can be performed on. Together with containers and views, they create a set of powerful abstractions, making it convenient to write code that can be executed both on a CPU and on a GPU. TNL provides parallel for-loops, sorting algorithms, parallel reductions and scans, numerical solvers, expression templates, and more. Example usage of TNL containers and algorithms can be found in the code section 3.1.

Chapter 2

Theory

In this chapter, we shall describe the theoretical concepts related to hash tables, and then we present the *HashGraph*, a scalable hash table using a sparse graph data structure [1]. We use some definitions and theorems from [7].

2.1 Hashing

Notation 2.1. We shall denote the set of integers $\{0, \ldots, U-1\}$ by $[U]$.

Notation 2.2. We use $\#i$: *condition* to denote the number of integers i that satisfy the given *condition*.

Definition 2.3. Let $\mathcal{U}$ be an arbitrary set called the *universe* and $\mathcal{B} = [m]$ a set of *buckets*. We define a *hash function* $h : \mathcal{U} \to \mathcal{B}$.

It is usual for the universe to be a set of integers $[U]$ and that the size of the universe $|\mathcal{U}|$ is much larger that the number of buckets m, denoted $m \ll U$. We shall also assume that any hash function h can be evaluated in constant time.

A *hash table* is a data structure that has a fixed hash function h, a set m buckets $B = \{b_0, \ldots, b_{m-1}\}$ and stores elements from $\mathcal{U}$ into one of those buckets. Note that multiple elements might be stored in a single bucket, and that an element x might not necessarily reside in the bucket $b_{h(x)}$.

Definition 2.4. Let $h : \mathcal{U} \to [m]$ be a hash function and $x, y \in \mathcal{U}$ two distinct elements of the universe. We say that x *collides* with y if $h(x) = h(y)$.

Definition 2.5. Let $\mathcal{H}$ be a set of hash functions from $\mathcal{U}$ to $[m]$. We say that $\mathcal{H}$ is *c-universal* for some constant $c \geq 1$ if

$$\forall x, y \in \mathcal{U}, x \neq y : \sum_{h \in \mathcal{H}} P(h(x) = h(y)) \leq \frac{c}{m}|\mathcal{H}|.$$

In other words, given two distinct elements $x, y \in \mathcal{U}$ and a hash function $h \in \mathcal{H}$ picked uniformly at random, the probability that x and y collide in the same bucket is at most c-times higher than for a hash function that is completely random.

Theorem 2.6. *Let $\mathcal{H}$ be a c-universal set of hash functions from $\mathcal{U}$ to $[m]$ and let h be a hash function picked from $\mathcal{H}$ uniformly at random. Let also $\mathcal{X} = \{x_1, \ldots, x_n\} \in \mathcal{U}$ a set of items stored in a hash table T, and $y \in \mathcal{U} \setminus \mathcal{X}$ some item not stored in T. Then*

$$\mathbb{E}[\#i : h(x_i) = h(y)] \leq \frac{cn}{m}.$$

Stated differently, given a hash function h picked $\mathcal{H}$ uniformly at random, the expected number of items stored in T that collide with the new element y is at most $\frac{cn}{m}$.

Proof. Let $h \in H$ be a hash function picked uniformly at random. Let us introduce indicator random variables $\forall i \in [n]$

$$A_i \begin{cases} 1 & \text{if } h(x_i) = h(y) \\ 0 & \text{otherwise.} \end{cases}$$

Then $\mathbb{E}[A_i] = P(A_i = 1) = P(h(x_i) = h(y))$. Since $\mathcal{H}$ is universal, we have

$$P(h(x_i) = h(y)) \leq \frac{c}{m}.$$

Let $A = (\#i : h(x_i) = h(y))$ be a random variable, then A can be also written as $A = \sum_i A_i$, and—by linearity of expectation—we have $\mathbb{E}[A_i] = \mathbb{E}[\sum_i A_i] = \sum_i \mathbb{E}[A_i] \leq \sum_i \frac{c}{m} = \frac{cn}{m}$. $\square$

Let us now describe some techniques to handle hash collisions inside hash tables.

2.1.1 Closed Addressing

Definition 2.7. A hash table T using a hash function h with a set of m buckets B uses a so-called *closed addressing* if

$$\forall x \in \mathcal{U} : (x \in T \implies x \in B[h(x)]) .$$

That means for each item x from $\mathcal{U}$, the exact bucket where x is stored is determined by the hash function alone. This, of course, implies that when collisions occur, there will be multiple elements inside a single bucket. Buckets must therefore implement a nested data structure to hold these elements. A question then arises: what sort of data structure should we choose to implement the buckets?

A popular choice is to use a linked list. The combination of closed addressing with linked lists inside buckets is referred to as *separate chaining*.

In a separate chaining hash table T with m buckets using a hash function $h : \mathcal{U} \to [m]$, we define operations *Insert*, *Find* and *Delete* as follows:

- *Insert(x)* inserts x to the front of the linked list in bucket $b_{h(x)}$.
- *Find(x)* will iterate over items in the linked list located in the bucket $b_{h(x)}$ until it either finds x, or until it reaches the end of the list. Finally, it returns a result indicating whether x was found.
- *Delete(x)* will first look for x the same way as Find, and if x is present in $b_{h(x)}$ it removes x from the linked list.

It turns out that if we have a hash function from a c-universal set, using a linked list is enough to support all of these operations in expected constant time.

Lemma 2.8. *Let T be a separate chaining hash table using a hash function from a c-universal set. Suppose T has m buckets $\{b_0, \dots, b_{m-1}\}$ and contains n elements $\{x_1, \dots, x_n\}$. Finally, suppose $n = \mathcal{O}(m)$. Then the expected time complexity for the operations* Insert*,* Find *and* Delete *is constant.*

Proof. Let us analyze all operations one by one.

- Unsuccessful find for item y will start to look in bucket $b_{h(y)}$ and then go through all elements in the bucket one by one. Since $y \notin b_{h(y)}$, we have to traverse all elements in $b_{h(y)}$, and from theorem 2.6 the expected number of those elements is at most $\frac{cn}{m}$, and because $n = \mathcal{O}(m)$, the expected number of elements is $\mathcal{O}(c)$.
- Successful find is the same as unsuccessful find, except the search for item y inside $b_{h(y)}$ may end before we traverse the entire linked list.
- Insert of y first does a find, and if the element is already present in T it terminates. Otherwise, y is inserted into the linked list in $b_{h(y)}$ in constant time to an arbitrary position.
- Delete of y first does a find, and if the element is not present in T, it terminates. Otherwise, y is removed from the linked list in constant time.

□

The ratio of n/m is known as the *load factor*. If the number of elements to be inserted is unknown up-front, the table can be dynamically resized and inserted elements rehashed whenever the load factor grows too much. If the table is resized in a way that $new\_size - old\_size = \Omega(old\_size)$, it can be easily shown that the expected amortized time for insertions is constant.

Using separate chaining is a common choice for hash table implementations, like the STL containers `std::unordered_set` and `std::unordered_map`. An example of how a separate chaining hash table stores elements is drawn in Figure 2.1.

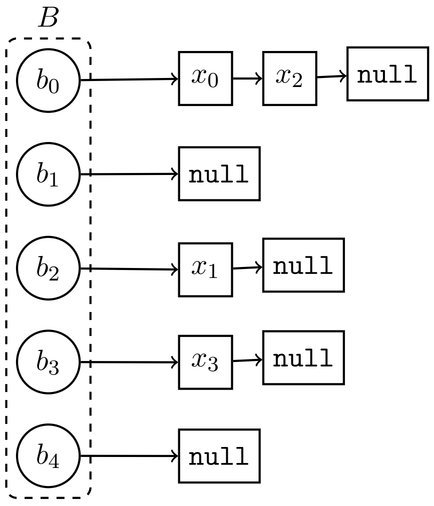

Figure 2.1: An example separate chaining hash table representation with 5 buckets and 4 items

2.1.2 Open Addressing

Definition 2.9. A hash table T using a hash function h with a set of m buckets B uses a so-called *open addressing* if

$$\forall \mathcal{X} \subset \mathcal{U} : (\mathcal{X} \in T \implies \forall b \in B : |b| \leq 1) .$$

The above definition states that for an open addressing hash table T, there is an invariant that every bucket must hold at most one element. Typically, the buckets are implemented as a contiguous array. The advantage over separate chaining is that buckets can store elements of $\mathcal{U}$ directly in the array, making it more space-efficient than having additional pointers for the linked list and more cache-friendly as well.

However, the invariant also implies that in case of a collision between two distinct elements x and y, at least one of them must be placed into a bucket other than $b_{h(x)} = b_{h(y)}$. There are many ways to choose where an element will be placed in case of a collision.

2.1.2.1 Probing

We define a *probing sequence* $\forall x \in \mathcal{U}$ as a function that assigns each element of the universe $\mathcal{U}$ a unique order of buckets in which the probing shall occur. Generally, given the item y, the sequence starts at bucket $b_{h(y)}$.

When inserting an item y into the table, we look to see whether the bucket $b_{h(y)}$ is empty. If so, we insert the element there, otherwise we select the next bucket in the order given by the probing sequence and try again. We continue this process until we find an empty space. Note that for this to work, we must ensure the table is not full before we start inserting.

When finding y, we follow the same probing sequence until we find the element in some bucket, or we terminate if we find an empty bucket.

When deleting y from the table, we first find the element. Then, if found, we can not just delete the item from the table since it would break the probing sequence for consequent finds. Instead, we replace the element with a *tombstone* element, a special type that can be overwritten by insert and is skipped by find.

Commonly used probing algorithms include

- *Linear probing*: The next bucket is determined as $b_{h(y)+l \mod m}$, where l is the current number of elements probed and m is the number of buckets.
- *Quadratic probing*: Same as linear probing, but the next bucket is defined as $b_{h(y)+l^2 \mod m}$.
- *Double hashing*: Let h_2 be a secondary hash function. The probing sequence is defined as $b_{h(y)+l*h_2(y) \mod m}$.

Using linear probing is the most cache-friendly approach since the next element to probe is almost always stored right after the current one[3]. On the other hand, linear probing tends to produce large clusters of full buckets, meaning all operations have to probe through more items.

Quadratic probing attempts to solve this while also being somewhat cache-friendly since the distance between the first few elements in the probing sequence is small.

[3]Except for the last element, for which the next element in the probing sequence is the first one.

2.1.2.2 Cuckoo Hashing

Cuckoo hashing works with two hash functions h_1 and h_2. Given a table T, we maintain an invariant

$$\forall x \in T : (x \in b_{h_1(x)}) \vee (x \in b_{h_2(x)}).$$

When finding an item y, we just search buckets $b_{h_1(y)}$ and $b_{h_2(y)}$ in constant time. Delete is also trivial, we do a find and then remove the found item, or do nothing if find is unsuccessful.

Insert first makes sure item y is not already present by doing a find. Then, if $b_{h_1(y)}$ or $b_{h_2(y)}$ is empty, we put y into any one of them. If both buckets are full, we select one of the two buckets arbitrarily, say, without loss of generality, we choose $b_{h_1(y)}$. We "kick out" the existing item $z = b_{h_1(y)}$, put y in its place, and continue the process by inserting z using h_2. If $b_{h_2(z)}$ is full, we again swap z with the existing item w, and we continue inserting w with h_1 and so on. If we are unable to finish the insert after some pre-defined amount of attempts—called the *insertion timeout*—we rehash the table with a new pair of functions h_1' and h_2'.

Claim 2.10. *For some choices of h_1, h_2 and some insertion timeout, the expected time complexity of insert is $\mathcal{O}(1)$.*

Proof. See [8]. □

2.2 HashGraph

Let us now present *HashGraph* [1], a novel approach introduced by Oded Green that examines hash tables from the point of view of graph theory. HashGraph supports two operations, *find* and *build* which constructs the HashGraph given a set of items $\mathcal{X} \subset \mathcal{U}$.

Definition 2.11. A *graph* is a pair $G = (V, E)$, where V is a set of vertices and $E = \{\{u, v\} | u, v \in V\}\}$ a set of edges.

Definition 2.12. A *bipartite graph* is a graph $G = (\{A \cup B\}, E)$, where $A \cap B = \emptyset$ and

$$\forall \{u, v\} \in E : (u \in A \wedge v \in B) \vee (u \in B \wedge v \in A)$$

Definition 2.13. Let $\mathcal{U}$ be a universe, $B = \{b_0, \ldots, b_{m-1}\}$ a set of m buckets and $\mathcal{X} \subset \mathcal{U}$ a set of elements. Let h be a hash function $h : \mathcal{U} \to [m]$. A *hash graph* is a bipartite graph $G = (([m] \cup \mathcal{X}), E)$, such that

$$(\{u, v\} \in E, \text{WLOG } u \in [m], v \in \mathcal{X}) \Leftrightarrow h(v) = u$$

That is, we can represent the relationship between elements and their corresponding buckets by a bipartite graph. Note that under this definition, any closed addressing hash table has a corresponding hash graph. See Figure 2.2 for an example of a hash table representation using a hash graph.

To see why using a graph to represent the relationship between buckets and elements is beneficial, we observe that every hash graph is sparse.

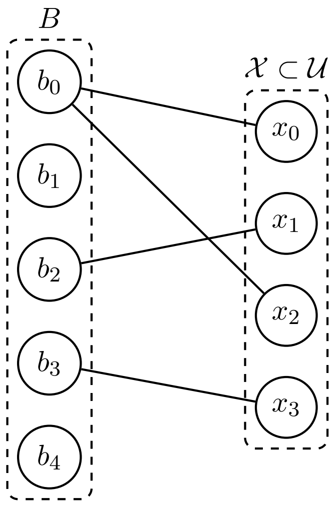

Figure 2.2: An example hash graph representation of a hash table with 5 buckets and 4 items

Definition 2.14. We call a graph $G = (V, E)$ *sparse*, if $|E| \in \mathcal{O}(|V|)$. Otherwise, we refer to G as *dense*.

Observation 2.15. *Every hash graph $G = (([m] \cup \mathcal{X}), E)$ is sparse, since it has exactly $|\mathcal{X}|$ edges.*

How should we represent a hash graph in a computer program? A common way to represent graphs in memory is the *adjacency matrix*.

Definition 2.16. Let $G = ((A \cup B), E)$ be a bipartite graph. An *adjacency matrix $A = \mathbb{N}^{|A|\times|B|}$* of G is defined as

$$A_{i,j} \begin{cases} 1 & \text{if } \{i, j\} \in E \\ 0 & \text{otherwise.} \end{cases}$$

Definition 2.17. We call a matrix $A = \mathbb{N}^{n\times m}$ *sparse*, if

$$\{\#(i, j) : A_{i,j} = 0\} = \mathcal{O}(min(n, m)).$$

Otherwise, we call to A *dense*.

It follows from observation 2.15 that the adjacency matrix of every hash graph is also sparse. This means that we can use various sparse matrix storage formats to represent hash tables. There are many such formats [9], but the original HashGraph uses the *Compressed Sparse Row (CSR)* storage.

2.2.1 Compressed Sparse Row

In a sparse matrix, the majority of its elements are zero by definition. It is, therefore, wasteful to store all $n * m$ elements of a square matrix with n rows and m columns. One of the data structures to represent sparse matrices is the *CSR* format.

In CSR, we store all non-zero elements of the matrix $A = \mathbb{N}^{n \times m}$ in a one-dimensional array called *values*. For each element in this array, we have to remember its column index, so we also keep an array called *columns* of the same size. Finally, we have an array *rows* of size $n + 1$, where we store for each row of A the index to the *values* array to keep track of where elements of individual rows begin and end. For example, to retrieve all elements from the 3rd row, we look at the elements $values[rows[2] \dots rows[3]]$. An example of a matrix and its CSR representation can be found if Figure 2.3.

$$\begin{pmatrix} 0 & 3 & 0 & 1 & 2 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix}$$

$$values \quad \begin{array}{|c|c|c|c|c|c|c|} \hline 3 & 1 & 2 & 5 & 1 & 4 & 2 \\ \hline \end{array}$$

$$columns \quad \begin{array}{|c|c|c|c|c|c|c|} \hline 1 & 3 & 4 & 0 & 0 & 4 & 3 \\ \hline \end{array}$$

$$rows \quad \begin{array}{|c|c|c|c|c|c|} \hline 0 & 3 & 4 & 4 & 6 & 7 \\ \hline \end{array}$$

Figure 2.3: A 5×5 matrix and its corresponding CSR representation

2.2.2 Representing Hash Graph With CSR

We want to represent a hash graph $G = (([m] \cup \mathcal{X}), E)$ using a hash function h and its corresponding adjacency matrix A similarly to the CSR format. We make a few observations.

1. Since the adjacency matrix contains only ones and zeros, we don't need to store them in the *values* array. Instead, we will replace all 1's in the matrix with the corresponding item. Put differently, $\forall j : A_{i,j} = 1$ we store $\mathcal{X}[j]$ into the *values* array.

2. If a pair $x, y \in \mathcal{X}, x \neq y$ collide and get stored in the same bucket, i.e., when they are in the same row of the adjacency matrix, the column they are at is irrelevant to us for the purposes of hashing. Thus, we can also drop the *columns* array.

Figure 2.4 illustrates the way HashGraph is stored. Notice the differences from the separate chaining representation in Figure 2.1. Instead of having a linked list in each bucket, the lists are flattened into contiguous arrays, and those arrays are then laid out in memory one after another in a single allocation.

rows

Bucket	b_0	b_1	b_2	b_3	b_4	
Rows	0	2	2	3	4	4

values

x_0	x_2	x_1	x_3

Figure 2.4: An example of HashGraph representation corresponding to the example in Figure 2.2

2.2.3 HashGraph Build

The only thing that remains is to show how to efficiently construct a HashGraph from a set of items $\mathcal{X} \subset \mathcal{U}$.

Definition 2.18. Let $\oplus$ be a binary associative operator. Then an *exclusive scan*[4] is a function of an array of n elements

$$(x_0, \ldots, x_{n-1})$$

and returns the $n + 1$ element array

$$(0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}).$$

Claim 2.19. *Given an array of n elements and p processors, exclusive scan can be computed in $\mathcal{O}(n/p + \log p)$ time.*

Proof. See [10]. $\square$

When building a HashGraph, we first need to determine the number of buckets to be used. The fewer buckets we use, the more collisions are going

[4]Sometimes also called a *prefix sum*

Algorithm 1: HashGraph-Build-V1

```
Input  : An array $\mathcal{X}$ of $n$ items to insert
         The number of buckets $m$
         A hash function $h : \mathcal{X} \to [m]$
1  //first we count how many elements are in each bucket
2  for $i = 0 \ldots m-1$ do in parallel
3      $CounterArray[i] \leftarrow 0$
4  for $i = 0 \ldots n-1$ do in parallel
5      $AtomicAdd(CounterArray[h(\mathcal{X}[i])], 1)$
6  //now we know the bucket offsets
7  $Rows = ExclusiveScan(CounterArray)$
8  //finally, we reorder the input array into their buckets
9  for $i = 0 \ldots m-1$ do in parallel
10     $CounterArray[i] \leftarrow 0$
11 for $i = 0 \ldots n-1$ do in parallel
12     $pos \leftarrow Rows[h(\mathcal{X}[i])] + AtomicAdd(CounterArray[h(\mathcal{X}[i])], 1)$
13     $Values[pos] \leftarrow \mathcal{X}[i]$
14 $HG \leftarrow (Rows, Values)$
15 return $HG$
Output : HashGraph HG
```

to occur per bucket. On the other hand, too many buckets will leave more of them empty, wasting space. It has been shown in [1] that setting the number of buckets to the total number of elements in the table is a good choice.

The original HashGraph build is summarized in Algorithm 1. Note that because we set $m = n$, the load factor of HashGraph table is always 1.

Observation 2.20. *Let p be the number of processors, n the number of elements to be inserted, and $m = n$ the number of buckets. Then Algorithm 1 runs in $\mathcal{O}(n/p + \log p)$ time.*

Proof. All four for loops of Algorithm 1 are embarrassingly parallel. As described in claim 2.19, the exclusive scan on line 7 runs in $\mathcal{O}(n/p + \log p)$ time. □

The authors of HashGraph also present an optimized cache-friendly build algorithm. The main idea is to reduce the number of cache-misses incurred by accesses to memory in a non-predictable way. We observe that Algorithm 1 has three points where memory accesses have poor spacial locality.

1. On line number 5 we access the *CounterArray* in pretty much random order.

2. We do it again on line number 12.
3. Finally, on line number 13 we access the $Values$ array non-sequentially.

To mitigate this, we can divide the build algorithm into two phases. In the first phase, we partially reorder the input array $\mathcal{X}$ so that elements that fall into nearby buckets are closer together in memory. Then, in the second phase, we build the HashGraph the same way as in Algorithm 1. This approach is laid out in Algorithm 2.

Let us analyze the memory access patterns of Algorithm 2. Assume that *BinCnt* is set so that an array of integers of size *BinSize* is small enough to fit into the cache.

1. First, on line 6 we no longer miss the cache, since *BinOffsets* fits into it.
2. Same situation occurs on line 14.
3. On line 15, we are still likely to miss the cache many times.
4. For the remainder of the algorithm, we use the reorganized array $\mathcal{X}_{reorg}$, so we are much less likely to miss the cache when indexing by the hash value of elements since nearby elements now have hash values close together.

Note that Algorithm 2 does not improve the time complexity of Algorithm 1.

2.2.4 Relationship With Sorting

When analyzing algorithms 1 and 2 more carefully, we can observe an interesting relationship with sorting. In both of them, we end up sorting the input array $\mathcal{X}$ of items to be inserted by the bucket they belong into, i.e., we sort individual elements by the hash function. Both of the algorithms use the *bucket sort* algorithm[5] to do the sorting, with the latter *HashGraph Build V2* algorithm additionally optimizing memory access patterns.

Bucket sort is summarized in Algorithm 3. If we only need the buckets themselves, we can skip constructing the sorted list in steps 5 to 8.

Notice that we are not restricted to the choice of bucket sort. We can use any sorting algorithm to construct the HashGraph. This leads to a generalized HashGraph build procedure, which is presented in Algorithm 4.

[5]Note that we are using the bucket sort algorithm to sort elements into hash table buckets!

Algorithm 2: HashGraph-Build-V2

```
Input : An array $\mathcal{X}$ of $n$ items to insert
        The number of buckets $m$
        A hash function $h : \mathcal{X} \to [m]$
        Number of bins $BinCnt$
1  $BinSize = \lceil m/BinCnt \rceil$
2  //first we count how many elements are in each bin
3  for $i = 0 \ldots BinCnt - 1$ do in parallel
4      $BCounterArray[i] \leftarrow 0$
5  for $i = 0 \ldots n - 1$ do in parallel
6      $Bin \leftarrow h(\mathcal{X}[i])/BinSize$
7      $AtomicAdd(BCounterArray[Bin], 1)$
8  //now we know the bin offsets
9  $BinOffsets = ExclusiveScan(BinCounterArray)$
10 for $i = 0 \ldots BinCnt - 1$ do in parallel
11     $BinCounterArray[i] \leftarrow 0$
12 for $i = 0 \ldots n - 1$ do in parallel
13     $Bin \leftarrow h(\mathcal{X}[i])/BinSize$
14     $pos \leftarrow BinOffsets[Bin] + AtomicAdd(BinCounterArray[Bin], 1)$
15     $\mathcal{X}_{reorg}[pos] \leftarrow \mathcal{X}[i]$
16 //first we count how many elements are in each bucket
17 for $i = 0 \ldots m - 1$ do in parallel
18     $CounterArray[i] \leftarrow 0$
19 for $i = 0 \ldots n - 1$ do in parallel
20     $AtomicAdd(CounterArray[h(\mathcal{X}_{reorg}[i])], 1)$
21 //now we know the bucket offsets
22 $Rows = ExclusiveScan(CounterArray)$
23 //finally, we reorder the $\mathcal{X}_{reorg}$ array into their buckets
24 for $i = 0 \ldots m - 1$ do in parallel
25     $CounterArray[i] \leftarrow 0$
26 for $i = 0 \ldots n - 1$ do in parallel
27     $pos \leftarrow Rows[h(\mathcal{X}_{reorg}[i])] + AtomicAdd(CounterArray[h(\mathcal{X}_{reorg}[i])], 1)$
28     $Values[pos] \leftarrow \mathcal{X}_{reorg}[i]$
29 $HG \leftarrow (Rows, Values)$
30 return $HG$
Output : HashGraph HG
```

Generalizing the HashGraph build procedure to essentially a sorting problem is an important step to simplifying future analyses and exploring alternative implementations. An example might be using Quicksort, Bitonic sort, Radix sort, or Merge sort in step 9 of Algorithm 4. See [11] for a survey of

Algorithm 3: Bucket Sort

```
Input  : Input $\mathcal{X} = (x_0, \dots, x_{n-1})$
         Buckets $B = (b_0, \dots, b_{m-1})$
         Key function $k : \mathcal{X} \to [m]$
1 for $i = 0 \dots m-1$ do
2     $b_i \leftarrow \emptyset$
3 for $i = 0 \dots n-1$ do
4     Insert $x_i$ into bucket $b_{k(x_i)}$
5 $S \leftarrow emptyList$
6 for $i = 0 \dots m-1$ do
7     Append items from $b_i$ to $S$
8 return $S$
Output : Sorted list $S$ by the key function $k$
```

Algorithm 4: Generalized HashGraph-Build

```
Input  : An array $\mathcal{X}$ of $n$ items to insert
         The number of buckets $m$
         A hash function $h : \mathcal{X} \to [m]$
1  //first we count how many elements are in each bucket
2  for $i = 0 \dots m-1$ do in parallel
3      $CounterArray[i] \leftarrow 0$
4  for $i = 0 \dots n-1$ do in parallel
5      $AtomicAdd(CounterArray[h(\mathcal{X}[i])], 1)$
6  //now we know the bucket offsets
7  $Rows = ExclusiveScan(CounterArray)$
8  //finally, sort the input array $\mathcal{X}$ w.r.t. the hash function $h$
9  $Values \leftarrow Sort(\mathcal{X}, h)$
10 $HG \leftarrow (Rows, Values)$
11 return $HG$
Output : HashGraph HG
```

GPU sorting algorithms. Studying ways to improve Bucket sort (or the closely related *Counting sort*) might also be relevant [12].

Observation 2.21. *Let p be a number of processors, n number of elements to be inserted and $m = \Theta(n)$ the number of buckets. Let T_{sort} be a time complexity of sorting n elements on p processors. Then the time complexity of Algorithm 4 is $\mathcal{O}(n/p + \log p + T_{sort})$.*

Proof. Because $n = \Theta(m)$ and the first two loops are embarrassingly parallel, both of them take $\mathcal{O}(n/p)$ time. As per 2.19, the exclusive scan runs in

$\mathcal{O}(n/p + \log p)$ time. Finally, we presumed that sorting takes T_{sort} time. □

2.2.5 HashGraph Find

To find item y in a HashGraph, we first determine the target bucket as $b = h(y)$, and then we query the *Rows* array to get an offset into the $Values$ array for bucket b as $offset \leftarrow Rows[b]$. Then, similarly to linear probing described in section 2.1.2.1, we iterate over all elements in the bucket until we either reach the end or until we find y.

This is summarized in Algorithm 5.

Algorithm 5: HashGraph-Find

```
Input  : HashGraph $HG$
         Item $y$ to find
1 $b \leftarrow h(y)$
2 $offset \leftarrow HG.Rows[b]$
3 $length \leftarrow HG.Rows[b+1] - HG.Rows[b]$
4 for $i = 0 \ldots length - 1$ do
5     if $HG.Values[offset + i] = y$ then
6         return True
7 return False
Output : True if $y$ is present in HG, False otherwise
```

Observation 2.22. *Let* HG *be a HashGraph with* m *buckets, a hash function* h *from a c-universal set, and finally let* $n = \mathcal{O}(m)$ *be the number of elements present in* HG. *Then the expected time complexity of Algorithm 5 is constant.*

Proof. Let y be the element we want to find and $b = h(y)$ the bucket where we shall look for it. From lemma 2.8, it follows that the expected number of items in b is constant, and so we expect that looping through all elements in b will take us constant time. □

If we want to do a bulk search of many keys in parallel, we can easily extend Algorithm 5. In the original HashGraph paper, this method is referred to as *HashGraph-Probe-Standard.*

While Algorithm 6 is simple, it may result in inefficient cache access patterns. Consider the situation where we want to look up two subsequent elements $y_1 = \mathcal{Y}[i]$ and $y_2 = \mathcal{Y}[i+1]$. We have no way of guaranteeing that the bucket $b_{h(y_1)}$ lies anywhere close to $b_{h(y_2)}$. Therefore, we might fetch all elements from one bucket to see if it contains a given element, and then before we get to processing another element that hashes to the same bucket, the

Algorithm 6: HashGraph-Probe-Standard

```
Input  : HashGraph $HG$
         Array of $n$ items $\mathcal{Y}$ to find
1 for $i = 0 \ldots n-1$ do in parallel
2     $y \leftarrow \mathcal{Y}[i]$
3     $Result[i] \leftarrow HashGraph\_Find(HG, y)$
4 return $Result$
Output : Array $Result$ where $i$-th element is True if and only if $\mathcal{Y}[i]$ is
         present in HG
```

contents of that bucket might have already been evicted from the cache.

To solve this, another version of the bulk find algorithm is introduced in [1] called *HashGraph-Probe-New*. The main idea is to first sort the input array $\mathcal{Y}$ of items to find by the hash value so that subsequent items hash to buckets that are next to each other. We already know from section 2.2.4 that we can use the HashGraph Build algorithm to do exactly that. The algorithm then can be summarized in two steps:

1. Create a second HashGraph $HG2$ from the elements we want to find using the same hash function as in the original table.

2. Go through all the elements in $HG2$ (notice they are all stored contiguously in the $Values$ array) and try to find them using Algorithm 5.

This procedure is summarized in 7. We add that creating the entire $HG2$ is not necessary, we can just partially sort the items we want to insert like in steps 1 to 15 of Algorithm 2.

2.3 Alternative Sparse Matrix Storage Formats

In section 2.2.1 we showed how HashGraph uses concepts from sparse matrix storage formats. In particular, we outlined the similarities with the Compressed Sparse Row format. A natural question arises whether we could leverage other storage formats to implement HashGraph. In this section, we shall describe a few other sparse matrix storage formats and comment on their viability as the internal storage for buckets in HashGraph.

2.3.1 Ellpack

The *Ellpack* format [13, page 6] is similar to the CSR format. Instead of having irregularly sized rows and indexing them by the *Rows* array, in Ellpack, all rows have the same size that is equal to the largest number of non-zero

Algorithm 7: HashGraph-Probe-New

```
Input  : HashGraph $HG$
         Array of $n$ items $\mathcal{Y}$ to find
1 // construct a second HashGraph using the hash function $h$
2 $h \leftarrow HG.h$
3 $m \leftarrow n$
4 $HG2 \leftarrow HashGraph\_Build(\mathcal{Y}, m, h)$
5 $\mathcal{Y}_{sorted} \leftarrow HG2.Values$
6 for $i = 0 \ldots n-1$ do in parallel
7     $y \leftarrow \mathcal{Y}_{sorted}[i]$
8     $Result[i] \leftarrow HashGraph\_Find(HG, y)$
9 return $Result$
Output : Array $Result$ where $i$-th element is True if and only if $\mathcal{Y}[i]$ is
         present in HG
```

elements in any given row. Subsequently, this means that rows with the lower number of non-zero elements must be padded with zeroes. The memory layout of the Ellpack format can be seen in Figure 2.5.

Note that unlike in CSR, in Ellpack we do not need to keep the *Rows* array because the offset of i-th row can be calculated as $i * n$, where n is the maximum number of non-zero elements over all rows.

It is clear that the Ellpack format is very inefficient in case there are a few rows with many non-zero elements and many rows with few non-zero elements, since there will be large padding overhead. In the special case that there is at least one row with all elements being non-zero, the Ellpack format will use $\mathcal{O}(nm)$ memory for a matrix $A^{n \times m}$.

Let us now consider the possibility of using Ellpack to store the internal state of HashGraph. First, there is no strong guarantee that elements will be evenly distributed into buckets, and therefore there might be relatively large padding overhead. On the other hand, if we use a c-universal hash function, and we have enough buckets, then it follows from theorem 2.6 that the expected number of elements in each row is constant.

Another point to consider is that using Ellpack might speed up accessing individual buckets due to the ability to compute the bucket offset without having to access memory to do so.

$$\begin{pmatrix} 0 & 3 & 0 & 1 & 2 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix} \qquad \begin{array}{c} \textit{values} \\ \begin{array}{|c|c|c|} \hline 3 & 1 & 2 \\ \hline 5 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 1 & 4 & 0 \\ \hline 2 & 0 & 0 \\ \hline \end{array} \end{array} \qquad \begin{array}{c} \textit{columns} \\ \begin{array}{|c|c|c|} \hline 1 & 3 & 4 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 4 & 0 \\ \hline 3 & 0 & 0 \\ \hline \end{array} \end{array}$$

Figure 2.5: A 5 × 5 matrix and its corresponding Ellpack representation

2.3.2 Sliced Ellpack

The *Sliced Ellpack* format [14] is a modification of Ellpack that tries to reduce the padding overhead in case of uneven distribution of non-zero elements across rows. We divide rows into groups of 32, and we only pad rows to the maximum number of non-zero elements in a row inside the given group. The benefit is that a row with a relatively high number of non-zero elements only affects the padding of 31 surrounding rows and not the entire matrix.

The now-familiar example matrix, this time using Sliced Ellpack storage format, is outlined in Figure 2.6. Note that for simplicity and compactness reasons, the example uses groups of size 2 instead of 32.

Importantly, we must now again store the information where individual rows begin. We only need to keep a single number per 32 rows, since all 32 will always have the same length. Using Sliced Ellpack as the backing data structure for HashGraph should be a compromise between CSR and Ellpack. The overhead associated with padding should be lesser than when using plain Ellpack, but at the same time, we need to perform one additional memory access when determining bucket offsets, albeit stored more compactly than in CSR.

$$\begin{pmatrix} 0 & 3 & 0 & 1 & 2 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 2 & 0 \end{pmatrix} \qquad \begin{array}{l} \textit{values} \\ \begin{array}{|c|c|c|} \hline 3 & 1 & 2 \\ \hline 5 & 0 & 0 \\ \hline \end{array} \\ \begin{array}{|c|c|} \hline 0 & 0 \\ \hline 1 & 4 \\ \hline \end{array} \\ \begin{array}{|c|} \hline 2 \\ \hline \end{array} \end{array} \qquad \begin{array}{l} \textit{columns} \\ \begin{array}{|c|c|c|} \hline 1 & 3 & 4 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \\ \begin{array}{|c|c|} \hline 0 & 0 \\ \hline 0 & 4 \\ \hline \end{array} \\ \begin{array}{|c|} \hline 3 \\ \hline \end{array} \end{array}$$

Figure 2.6: A 5×5 matrix and its corresponding Sliced Ellpack representation

2.4 Dynamic HashGraph

In this section, we will explore ways how to modify HashGraph in a way that it can support dynamic operations, namely *Insert* and *Delete* operations. So far, we have only described HashGraph as an entirely static data structure, supporting only *Build* and *Find* operations.

Let us begin by analyzing the original CSR variant of HashGraph. Because of the way that elements are compressed into individual buckets in a single contiguous memory allocation, there is not any space available for the insertion of elements.

2.4.1 Rehashing

The solution at hand is to always rehash the entire table into a new HashGraph when inserting new items. While this is a naive approach in the general case, it can be a good strategy if the number of elements we want to insert n is roughly at least the same as number of elements that are already present in the table, or in other words, if $n = \Omega(|HG|)$.

The procedure is also very simple to implement. We need to concatenate the existing $Values$ array inside the HashGraph with the new array of elements to be inserted and then call $HashGraphBuild$ from algorithm 1, 2 or 4. This is summarized in Algorithm 8.

Algorithm 8: HashGraph-Insert-Rehash

```
Input  : HashGraph $HG$
         An array $\mathcal{X}$ of $n$ items to insert
1 $\mathcal{X}' \leftarrow Concatenate(HG.Values, \mathcal{X})$
2 $HG' \leftarrow HashGraph\_Build(\mathcal{X}')$
3 return $HG'$
Output : HashGraph $HG'$ containing $HG \cup \mathcal{X}$
```

Rehashing the entire table is, of course, not a good solution when the number of items to insert is too small, i.e., in the order of $n = o(|HG|)$, because the rehashing cost would dominate the insertion time per element.

2.4.2 The Hornet Data Structure

Authors of HashGraph suggest using the *Hornet* data structure [15] to support dynamic operations. Hornet is designed to efficiently represent the storage layer of dynamic graphs and matrices. The data structure is essentially a set *block-arrays*, where each block-array consists of contiguous memory chunks of the same size called *blocks*. Each block has a size of power of two. Block-arrays are coupled with auxiliary data structures that support fast allocation and deallocation of individual blocks, those being a *Vectorized Bit Tree* and B^+ *Tree* respectively.

Instead of having individual buckets laid out contiguously in memory one after another, Hornet can be used to represent adjacency lists of the bipartite graph associated with the hash table, along with supporting extending or shrinking the adjacency lists, thus providing the basis for Insert and Delete operations.

While the investigation on the usage of Hornet alongside HashGraph has started [1, page 11], there was no publicly available version of it at the time of writing this thesis. We shall not delve deeper into analyzing the Hornet data structure in this text and will instead attempt to explore other options.

2.4.3 Using Techniques From Open Addressing

Let us return to the main problem of why HashGraph is not able to support dynamic operations—the lack of space in the densely packed array of buckets. We could add some empty positions to the end of each bucket that could be eventually filled up with new items.

How many empty positions should we reserve in each bucket? If we build a HashGraph with n elements, we have $m = n$ buckets, and so having 1 empty space per bucket means there will be 50% empty positions in total. Similarly, having 2 empty spaces in each bucket will result in 2/3 of the positions being empty. Having too many empty slots per bucket is not desirable, as it will negatively affect the memory footprint of the data structure.

We shall describe a possible way to extend HashGraph with empty slots, so it can support Insert and Delete operations. We start with a version that uses a CSR-like storage format akin to the original HashGraph.

2.4.3.1 Build

The procedure to build such HashGraph can be found in Algorithm 9. It can be trivially extended to a version similar to Algorithms 2. Notice that we initialize the size of each bucket to e on line 3 and that we initialize the $Values$ array to a special value *k_empty* on line 10. This will be important during insertion.

Algorithm 9: HashGraph-Dynamic-Build

```
Input  : An array $\mathcal{X}$ of $n$ items to insert
         The number of buckets $m$
         A hash function $h : \mathcal{X} \to [m]$
         Number of empty slots per bucket $e$
1  //first we count how many elements are in each bucket
2  for $i$ = $0 \ldots m - 1$ do in parallel
3      $CounterArray[i] \leftarrow e$
4  for $i$ = $0 \ldots n - 1$ do in parallel
5      $AtomicAdd(CounterArray[h(\mathcal{X}[i])], 1)$
6  //now we know the bucket offsets
7  $Rows = ExclusiveScan(CounterArray)$
8  //initialize all elements to a special empty value
9  for $i$ = $0 \ldots m * (e + 1) - 1$ do in parallel
10     $Values[i] \leftarrow k\_empty$
11 //finally, we reorder the input array into their buckets
12 for $i$ = $0 \ldots m - 1$ do in parallel
13     $CounterArray[i] \leftarrow 0$
14 for $i$ = $0 \ldots n - 1$ do in parallel
15     $pos \leftarrow Rows[h(\mathcal{X}[i])] + AtomicAdd(CounterArray[h(\mathcal{X}[i])], 1)$
16     $Values[pos] \leftarrow \mathcal{X}[i]$
17 $HG \leftarrow (Rows, Values)$
18 return $HG$
Output : HashGraph HG
```

2.4.3.2 Insert

Now that we have built a HashGraph with some empty slots, we can implement a dynamic insert. As we discussed, there are only a few empty slots per bucket, and we might run out of space after a few insertions. So far, HashGraph has adhered to a closed-addressing hash table definition outlined in 2.7. For the purposes of supporting dynamic insertions, we allow items to be placed in buckets other than the ones corresponding to the hash function of said items.

This will leave us with a hash table that uses neither closed nor open-addressing—there can be multiple elements per bucket, and at the same time, items can be stored in buckets not corresponding to their hash functions. Put differently, neither the condition in definition 2.9 nor in definition 2.7 will not hold.

When inserting element y, we will go through the bucket $b_{h(y)}$ and attempt to find an empty slot. If we find one, we will place y there and finish. Otherwise, we will choose another bucket according to some probing sequence (see section 2.1.2.1 about probing sequences) and try to put the element there.

Algorithm 10 outlines the way insert might be implemented with a simple linear probing over all buckets. Note that it can be easily extended to bulk insertion of multiple elements in parallel with the use of *Compare-And-Swap* instruction on line 10.

Algorithm 10: HashGraph-Dynamic-Insert

```
Input  : Dynamic HashGraph $HG$
         Item $y$ to insert
1  //probe over all buckets
2  for $i = 0 \dots m - 1$ do
3      $bucket \leftarrow (h(y) + i) \mod m$
4      $offset \leftarrow HG.Rows[bucket]$
5      $length \leftarrow HG.Rows[bucket + 1] - HG.Rows[bucket]$
6      //try to find an empty space inside the bucket
7      for $i = 0 \dots length - 1$ do
8          if $HG.Values[offset + i] = y$ then
9              return ErrorDuplicateEntry
10         if $HG.Values[offset + i] = k\_empty$ then
11             $HG.Values[offset + i] \leftarrow y$
12             return Success
13 return ErrorNoSpaceLeft
Output : Status indicating whether the insert was successful
```

If the table is full, new inserts will fail with the error *ErrorNoSpaceLeft*. In that case, the table needs to be rebuilt to make space for additional items.

2.4.3.3 Find

Finding elements is also very similar to how it works in open-addressing tables. We start looking for y in $b_{h(y)}$ and iterate over all elements in that bucket until we either find it or until we reach an empty slot, which means the element is not in the hash table. If we neither find y nor reach any empty slot, we continue the process by looking into the next bucket defined by the probe sequence.

Refer to Algorithm 11 to see how Find is implemented with linear probing over all buckets.

Algorithm 11: HashGraph-Dynamic-Find

```
Input  : Dynamic HashGraph $HG$
         Item $y$ to find
1  //probe over all buckets
2  for $i = 0 \ldots m - 1$ do
3      $bucket \leftarrow (h(y) + i) \mod m$
4      $offset \leftarrow HG.Rows[bucket]$
5      $length \leftarrow HG.Rows[bucket + 1] - HG.Rows[bucket]$
6      //try to find $y$ inside the bucket
7      for $i = 0 \ldots length - 1$ do
8          if $HG.Values[offset + i] = y$ then
9              return True
10         if $HG.Values[offset + i] = k\_empty$ then
11             return False
12 return False
Output : True if $y$ is present in HG, False otherwise
```

2.4.3.4 Delete

When deleting an element, we first try to find it. If the element is not present in the table, we return immediately. Otherwise, we replace the current value with a special value called a *delete-marker*[6],[7].

2.4.3.5 Storage Format Considerations

In previous sections, we have outlined an extension to the HashGraph data structure, adding support for Insert and Delete operations. We can now consider using storage formats other than CSR, like the ones described in sections 2.3.1 and 2.3.2.

Using Ellpack and Sliced Ellpack in static HashGraph had the disadvantage of potentially wasting storage space if elements were unevenly distributed into buckets. With dynamic HashGraph, we can use those empty slots to store newly inserted items. This allows us to use the space allocated by the data structure more efficiently and, at the same time, keep the benefit of having equally-sized buckets and, therefore, cheaper indexing.

[6]We can again use a *Compare-And-Swap* instruction if we want to support concurrent operations.

[7]Sometimes also called a *tombstone*.

Algorithm 12: HashGraph-Dynamic-Delete

```
   Input  : Dynamic HashGraph $HG$
            Item $y$ to delete
1  //probe over all buckets
2  for $i$ = 0 . . . $m - 1$ do
3      $bucket \leftarrow (h(y) + i) \mod m$
4      $offset \leftarrow HG.Rows[bucket]$
5      $length \leftarrow HG.Rows[bucket + 1] - HG.Rows[bucket]$
6      //try to find $y$ inside the bucket
7      for $i$ = 0 . . . $length - 1$ do
8          if $HG.Values[offset + i] = y$ then
9              $HG.Values[offset + i] \leftarrow DeleteMarker$
10             return $True$
11         if $HG.Values[offset + i] = k\_empty$ then
12             return $False$
13 return $False$
   Output : True if $y$ was present in HG, False otherwise
```

Chapter 3

Realization

In this chapter, we first describe State-Of-The-Art implementations of hash tables on GPGPUs and then outline our own implementation of static and dynamic HashGraph in the TNL library.

3.1 State-Of-The-Art

A survey of existing approaches to GPU hash tables implementations is provided by Lessley et al. [16]. We outline some notable implementations below.

- Alcantara et al. [17] introduce analysis and implementation of a set of parallel hash tables using open addressing, separate chaining, and cuckoo hashing as a part of the *CUDA Data Parallel Primitives Library (cuDPP)*.

- Khorasani et al. [18] provide a double hashing based approach called *Stadium hashing*, which stores data in host memory and only keeps an auxiliary densely-packed data structure on the GPU called the *ticket board*. The ticket board contains one bit per table slot, indicating whether the slot is occupied. Optionally, each slot might also carry a few bits of additional information describing the key currently present in the slot. The key idea is to speed up probing by traversing through this densely packed ticket board quickly.

- *cuDF* [19] is part of *NVIDIA RAPIDS* framework. It serves as a pandas-like [20] data frame manipulation library that also includes an open addressing hash table for CUDA using linear probing.

- *SlabHash* [21] is a dynamic hash table implementation that utilizes techniques of separate chaining. The linked lists inside the table are comprised of *slabs*, and each slab is capable of storing multiple items.

- *WarpCore* [22] is a novel implementation of an open addressing dynamic hash table that utilizes using multiple threads per single probe using CUDA's cooperative groups, thus supporting faster searches in high load-factor scenarios. The implementation achieves up to 1.6 billion inserts and up to 4.3 billion retrievals per second on a single GV100 GPU.

3.2 TNL Segments

TNL supports data structures for sparse matrices outlined in section 2.3. They can be accessed through an abstraction layer called *segments*, which represents data structures for manipulation of several arrays having different sizes in general.

Notably, a TNL segment does not hold any data itself. The data of the sparse matrix is stored in one contiguous array, and a segment only provides a mapping between the indices of this single array and the logical representation.

Listing 3.1 shows how the `TNL::Algorithms::Segments::CSR` segment provides indexing capabilities to matrix values stored in CSR format.

```
TNL::Containers::Array<int> values {3, 1, 2, 5, 1, 4, 2};
TNL::Algorithms::Segments::CSR<TNL::Devices::Host, int> segment
  {3, 1, 0, 2, 1};

segment.forAllElements([=] (int segmentIdx, int localIdx, int globalIdx)
{
      cout << segmentIdx         << '␣'
           << localIdx           << '␣'
           << globalIdx          << '␣'
           << values[globalIdx] << '\n';
});

// output
// 0 0 0 3
// 0 1 1 1
// 0 2 2 2
// 1 0 3 5
// 3 0 4 1
// 3 1 5 4
// 4 0 6 2
```

Listing 3.1: Example of CSR Segment in TNL

TNL provides multiple segment types. For our purposes, we use

- `TNL::Algorithms::Segments::CSR`
- `TNL::Algorithms::Segments::Ellpack`
- `TNL::Algorithms::Segments::SlicedEllpack`

which correspond to the storage formats described in sections 2.2.1, 2.3.1 and 2.3.2 respectively.

3.3 Storage of Key-Value Pairs

So far, we have talked about storing a set of items from some universe $\mathcal{U}$ in a hash table. We can refer to those items as *keys*. Often, we want to store some metadata associated with each key, or in other words, we want to store a pair $(key, value)$. For the purpose of hashing, the items $(key, value)$ and $(key, value')$ are considered equivalent. We have to slightly modify the operations supported by hash tables.

- *Find* searches for a given *key* the same as before, but instead of returning a boolean value indicating the presence of the key, it returns the associated *value* (or some special *empty* value in case the *key* is not found).
- *Insert* looks for an empty slot, and stores both *key* and *value* if it finds one. If it encounters a slot with the same *key* already present in the hash table, it returns with a failure.
- *Delete* tries to find a given *key*, and if present, it deletes both the *key* and the associated *value*.

We explore two ways how to store key-value pairs.

3.3.1 Array-Of-Structures

The first option is to directly store the pair $(key, value)$ in each slot instead of just plain keys. Such memory layout is called an *array of structures*, signifying that we have a single array that stores structs of keys and values.

This has the advantage that the value of a given key is always stored in an adjacent place in memory, making the access to both of them at the same time cache-efficient.

Conversely, this can also be a disadvantage. Consider a hash table with relatively many collisions. We have to potentially traverse multiple slots during Find, Insert, or Delete operations, each time fetching both the key and the value from the cache, only to continue looking for our key in a different slot. This can be especially inefficient if the values have a large memory footprint compared to the keys.

3.3.2 Structure-Of-Arrays

We can also take, in some sense, the opposite approach and use a *structure of arrays instead.* That means storing all the keys in one array and all values in a separate one. Those arrays are intrinsically connected by indices, meaning that a key stored at i-th index in the *keys* array has the associated value stored at *i*-th index in the *values* array.

Since keys are densely packed together, it makes iterating over them faster. On the other hand, we have to perform extra memory access to an entirely different memory location when we want to fetch the value.

3.4 Summary of Implemented Hash Tables

Here we describe hash tables implemented as a part of this thesis using TNL.

3.4.1 HashGraphSet

HashGraphSet represents a hash graph implementations of a set of items. The C++ class definition can be seen in listing 3.2.

```
template<
  typename Key,
  typename Hash,
  typename Device = TNL::Devices::Host,
  typename Segment = TNL::Algorithms::Segments::CSR<Device, Key>,
  HashGraphType HashGraphBuildType = HashGraphType::v2,
  typename Index = int>
class HashGraphSet;
```

Listing 3.2: HashGraphSet class declaraion

The class has several template parameters.

- *Key* is the type of the stored items.
- *Hash* is a function object representing the hash function.
- *Device* is the device the hash table is stored on. Can be specified either as `TNL::Devices::Host`, representing the CPU, or `TNL::Devices::Device`, representing a GPU.
- *Segment* is the type of segment that is used to index the internal array of stored items. See section 3.2.
- *HashGraphBuildType* is an enumeration stating which algorithm shall be used for building the hash graph. Options are *HashGraphType::v1* and *HashGraphType::v2* for Algorithms 1 and 2 respectively.
- *Index* is the integral type used to store indices (like the *Rows* array). For example, using a 32 bit integer means we can index up to 2^{32} items.

HashGraphSet is a static hash table, meaning that it only supports Build and Find operations. We also provide limited Insert functionality with the naive rehash-everything algorithm 8. This data structure can store keys of arbitrary types as long as the user provides the corresponding hash function.

3.4.2 HashGraphSetDynamic

HashGraphSetDynamic is an extension of HashGraphSet, and it has the same set of template parameters. Additionally, it adds support for Insert and Delete operations. The implementation follows the theoretical description that we presented in section 2.4.3.

As mentioned in section 2.4.3, we use compare-and-swap instructions provided by the CUDA platform to support parallel operations. Thus, we can only store specific data types that are either 16-, 32-, or 64-bit wide.

3.4.3 HashGraphMap and HashGraphMapDynamic

HashGraphMap is like a HashGraphSet, but it stores key-value pairs. It has one additional template parameter *Storage*, which specifies whether keys and values are stored in an array-of-structures, or in a structure-of-arrays as outlined in sections 3.3.1 and 3.3.2. The static map can—like the static set—store arbitrary types of key-value pairs.

HashGraphMapDynamic is an extension of HashGraphMap supporting Insert and Delete operations. This implementation does not support executing all kinds of operations concurrently. In other words, we can only run any single one of Find, Insert, or Remove at a time. The reason for this is that in order to support those operations concurrently, some additional synchronization would be needed. Consider the following sequence of operations:

1. Insert of key k and value v is executed. Suppose k is not present in the table. Because there is no intrinsic operation in CUDA to store both k and v atomically, we must first store k into some empty bucket using atomic compare-and-swap, and only if we successfully insert the key we store the value v.
2. Find of key k is executed, and it reads the key that was just inserted, but the value is not yet stored, so we return an incorrect result.
3. Insert completes the store of v.

In addition, the dynamic map also uses compare-and-swap instructions when manipulating the keys. This means that we are again limited to 16-, 32-, or 64-bit keys. Note that values can be of arbitrary type.

Chapter 4

Testing

In this chapter, we describe the testing and evaluation methodology of our implementation. We then compare our implementation against selected existing implementations under various scenarios.

4.1 Testing Methodology and Correctness Tests

In our performance evaluation scenarios, we store 4-byte integer keys and 4-byte integer values unless stated otherwise. Each test is run 10 times, and only the average result of those runs is shown. We measure mainly the number of operations performed by a given data structure in a given amount of time, shown in the order of billion operations per second. Given two time durations T_1 and T_2, we also define a *speedup* of T_2 over T_1 as

$$\Delta T = \frac{T_1}{T_2}.$$

We only measure the time the operations themselves take, i.e., we exclude the time to set up the input data. The hash function used is the *Murmur hash*, which is available from [23].

We used the *Google Test* [24] testing framework to implement a set of tests checking the correctness of our implementation of static and dynamic HashGraphs. We test all supported operations for individual sets and maps. Our tests can run either on a CUDA device, or on the host CPU.

4.2 Existing Solutions

We selected the following implementations of hash tables to compare against.

- **std::unordered_set** from the C++ standard template library. This container is usually implemented as a separate chaining hash table, and it only supports sequential operations on the CPU. We choose this container mainly to provide a reference base-case scenario.

- **WarpCore** is an implementation of the hash table described in section 3.1. It is available from Github [25].
- **SlabHash** has also been described in section 3.1. The implementation is open source and available from Gihub [26].
- **HashGraph** is an implementation of the hash table introduced in the original HashGraph paper [1]. It is available from [27].

We believe the chosen GPU hash table implementations are among the most performant solutions currently available.

4.3 Testing Environment

All tests have been performed on a machine running *Arch Linux* running on kernel *5.17.4-arch1-1*, with the following hardware configuration:

CPU: 2x Intel® Xeon® CPU E5-2630 v3 @ 2.40GHz (8 cores, 20480 KB cache)

RAM: 125Gi

GPU: NVIDIA Quadro P6000, 24576 MiB

The CPU tests were compiled using the *g++ 11.2.0* compiler with flags set to *-O3 -march=native -funroll-loops -ftree-vectorize -mavx.*

The GPU tests were compiled using the *nvcc V11.6.112* compiler with flags set to *-O3 -arch=compute_70 -code=sm_70 –ptxas-options=-O3 –expt-relaxed-constexpr –extended-lambda -march=native -Xcompiler -funroll-loops -Xcompiler -ftree-vectorize.*

4.4 CPU Tests

Since our implementation is based on and extends the Template Numerical Library, all data structures presented in this work are capable of running both on a CPU and a GPU. We leverage this fact and test our static set implementation `HashGraphSet` against the hash table implementation provided in STL, namely the `std::unordered_set`.

Note that we use the Murmur hash rather than the default STL hash function for the unordered set because the default hash function is usually implemented as an identity function for integers, and so it can heavily skew synthetic benchmark results.

4.4.1 Segment Memory Usage

We begin by analyzing the storage space required by different kinds of sparse matrix storage formats. Figure 4.1 shows the number of slots allocated when n items $\{1, \ldots, n\}$ are used to build the table. We observe the following:

- CSR always allocates exactly as many slots as the number of items that are used to build the table. In other words, no space is wasted.
- Using the Murmur hash, Ellpack allocates 10 times as many slots as the number of items, which means that at least one bucket has 10 collisions. This shows the Ellpack format can be extremely wasteful, especially when used for static hash tables. For dynamic tables, the space is still initially wasted, but subsequent inserts can occupy these empty slots.
- Sliced Ellpack is somewhere in between, as expected.

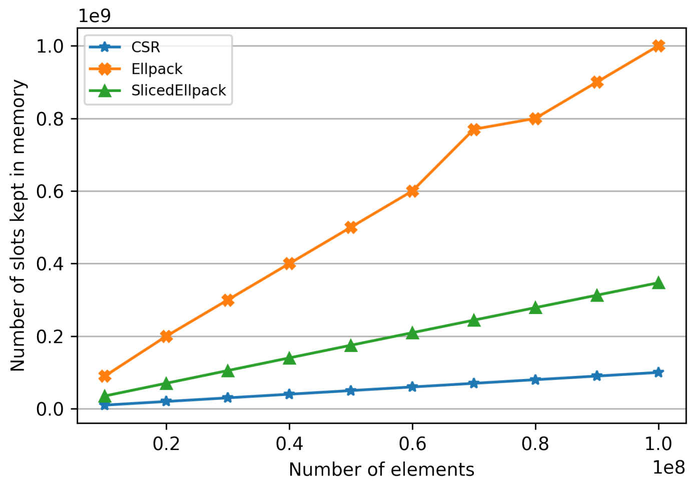

Figure 4.1: Memory footprint of different segment types.

4.4.2 Build

We start with a scenario where we build a set of n integers $\{1, \ldots, n\}$.

In the case of `std::unordered_set`, this is done by calling *insert* n times. The array of buckets can be pre-allocated.

For `HashGraphSet`, the build procedure is called. We compare the usage of CSR, Ellpack, and SlicedEllpack segments as the underlying storage type. In this particular test, we use the enhanced Algorithm 2 for building the hash graph. We examine the differences between Algorithm 1 and 2 in a later section.

Figure 4.2 shows the relationship between n and the number of operations processed per second[8]. We see that the CSR and SlicedEllpack outperform the Ellpack variant, likely because the internal storage for Ellpack is much larger, lowering the locality of individual items. Because the STL set is usually implemented as a separate chaining table, it means that a new node has to be dynamically allocated for every insert, which is the main reason why the performance is lower.

The speedup of our implementation over the STL container is summarized in Figure 4.3.

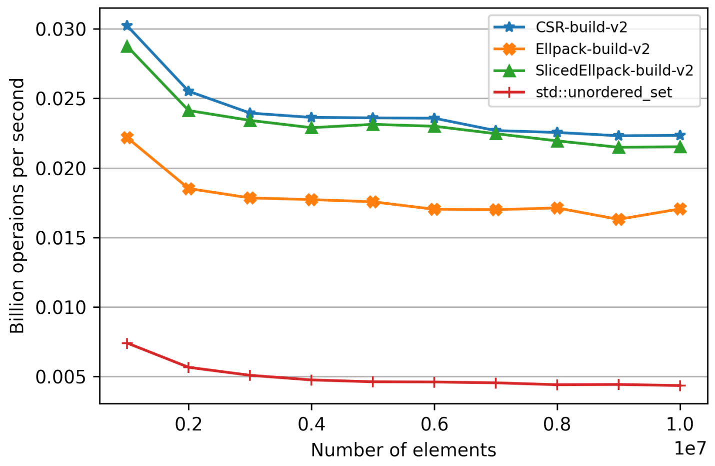

Figure 4.2: CPU build performance of `std::unordered_set` and `HashGraphSet`.

[8]For example, if $n = 1000$ and the number of operations per second processed is 100, then the build took $\frac{1000}{100} = 10$ seconds.

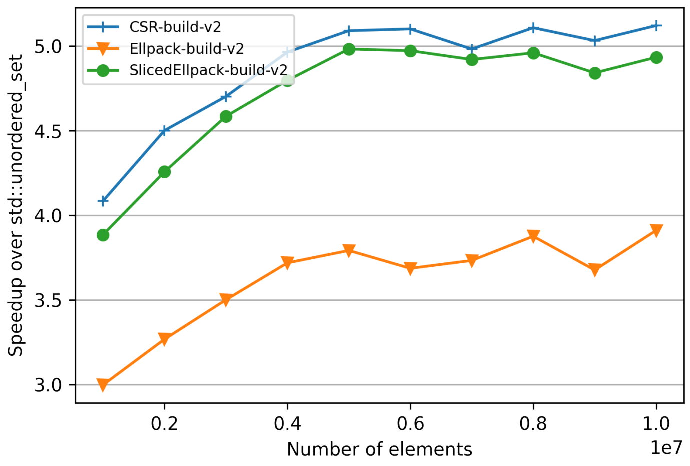

Figure 4.3: CPU build speedup of `HashGraphSet` over `std::unordered_set`.

4.4.3 Retrieve

For retrieval tests, we build a table with n integers $\{1, \ldots, n\}$ and then we retrieve all of them. The time for building the table is excluded from the measurement.

We used the probing Algorithm 7 for `HashGraphSet`. Retrieval performance is summarized in Figure 4.4. All versions of the hash graph show similar performance, while the STL is again much slower, likely due to the internal linked lists not utilizing the cache fully.

4.5 GPU Tests

We now proceed to evaluate our implementations on a GPU. First, we study some of the differences between various hash-graph-related algorithms and implementation details described in chapters 2 and 3. We then compare them against state-of-the-art GPU implementations publicly available.

4.5.1 Comparing Hash Graph Build Algorithms

In this section, we focus on comparing the two different build algorithms, namely HashGraph-Build-V1[9] and HashGraph-Build-V2[10]. The number of

[9]See Algorithm 1

[10]See Algorithm 2

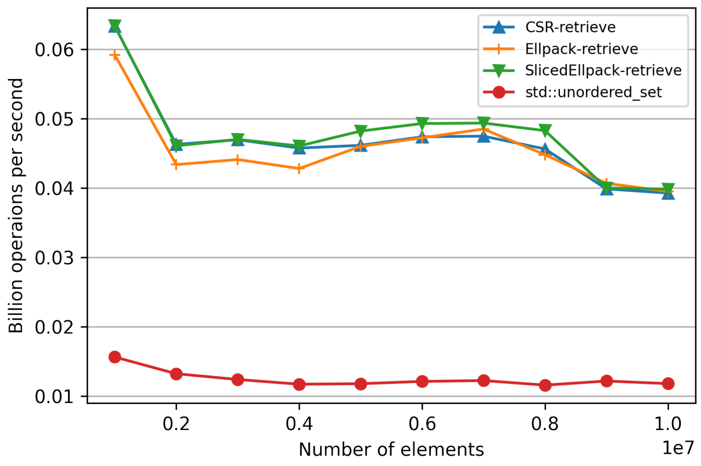

Figure 4.4: CPU retrieve performance of `std::unordered_set` and `HashGraphSet`.

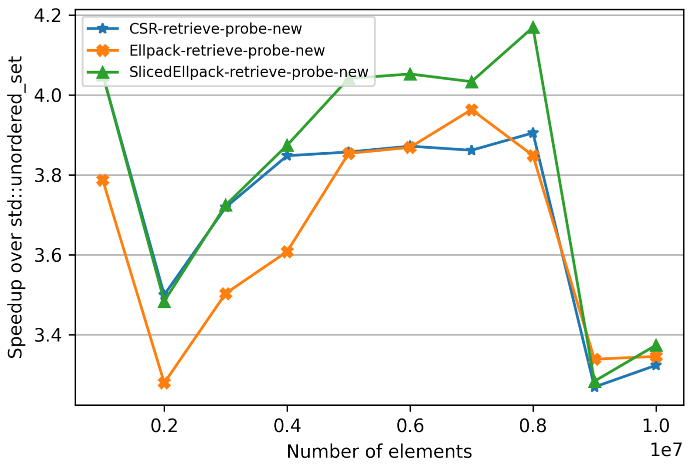

Figure 4.5: CPU retrieve speedup of `HashGraphSet` over `std::unordered_set`.

bins in the V2 version was set to 32000. We again test the static `HashGraphSet`.

The results can be seen in Figure 4.6. We also include the open-source hash graph implementation [27] for comparison, which internally uses the V2 build algorithm and a CSR-like memory layout. It is labeled *HashGraph* in the legend.

Notice that the difference between performance on the CPU and the GPU is around 20-fold on our hardware configuration. As expected, the V2 algorithm outperforms the V1 algorithm for all segment types. From now on, we shall only consider the V2 build algorithm.

When compared to `std::unordered_set`, the speedup is even larger. Refer to Figure 4.7 for speed up of the V2 build algorithm running on GPU compared to the STL container.

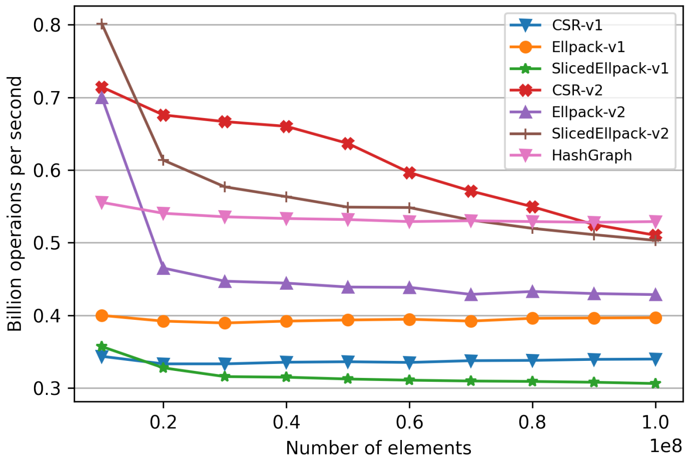

Figure 4.6: GPU build performance of V1 and V2 build algorithms.

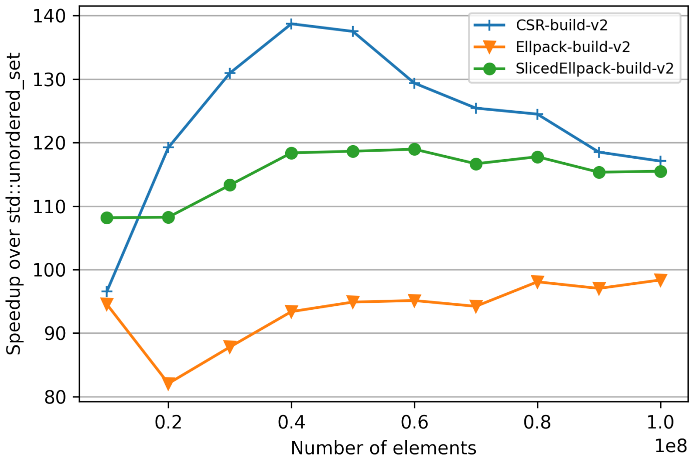

Figure 4.7: GPU build speedup of `HashGraphSet` over `std::unordered_set`.

4.5.2 Comparing Static And Dynamic Hash Graph

We continue by comparing the original static hash graph with the dynamic versions we introduced in section 2.4.3. Specifically, we measure the differences between `HashGraphSet` and `HashGraphSetDynamic`.

4.5.2.1 Build

As in previous sections, we start with the build performance of n integers $\{1, \ldots, n\}$. We expect the dynamic versions to be slower because they have to initialize some additional empty slots during the build. Results can be seen in Figure 4.8.

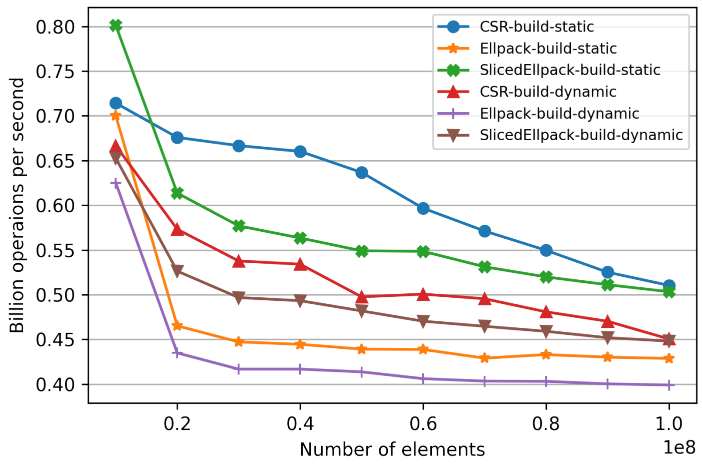

Figure 4.8: GPU build performance of static and dynamic set. The V2 build algorithm is used in all cases.

4.5.2.2 Retrieve

We use the HashGraph-Probe-Standard algorithm to compare the retrieval performance of the two sets. We first build a table of n integers $\{1, \ldots, n\}$, and then we retrieve all of them. Only the retrieval time is considered in the measurement.

From Figure 4.9, we can see that the retrieve performance of the dynamic set is virtually identical to the static version. This is because after a build, every item $x \in T$ is stored in the bucket $b_{h(x)}$, and therefore the dynamic version behaves exactly like the static one.

Note that if we were to insert additional items after the build, new items would not be necessarily stored in the bucket determined by the hash function, and the retrieve would start to behave more like a traditional open-addressing hash table. We shall investigate such a scenario in a later section.

The open-source HashGraph implementation unfortunately only supports the build operation, and so it is not included in this test.

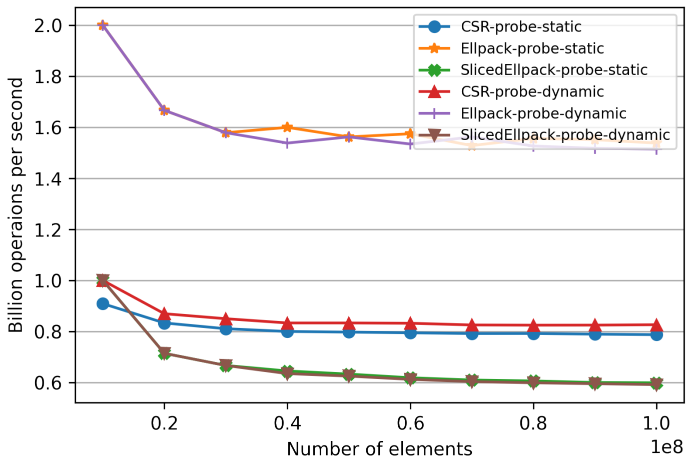

Figure 4.9: GPU retrieve performance of static and dynamic set.

4.5.3 Comparing Hash Graph Probing Algorithms

As outlined in section 2.2.5, there are two algorithms for finding items stored in a hash graph, namely the HashGraph-Probe-Standard 6 and HashGraph-Probe-New 7. We can follow their performance in Figure 4.10. The static `HashGraphSet` data structure was used.

We observe that Ellpack is the fastest one using the HashGraph-Probe-Standard algorithm, followed by all the HashGraph-Probe-New algorithms. It seems most time is spent in building the second hash graph during HashGraph-Probe-New, and subsequent probing takes very little time and is not impacted by the storage format.

The large performance advantage of Ellpack is attributed to one fewer memory access because it does not have to look up the offset of each bucket.

When compared to the `std::unordered_set`, the speedup of retrieving items is over 100× in some cases, as illustrated by the Figure 4.11.

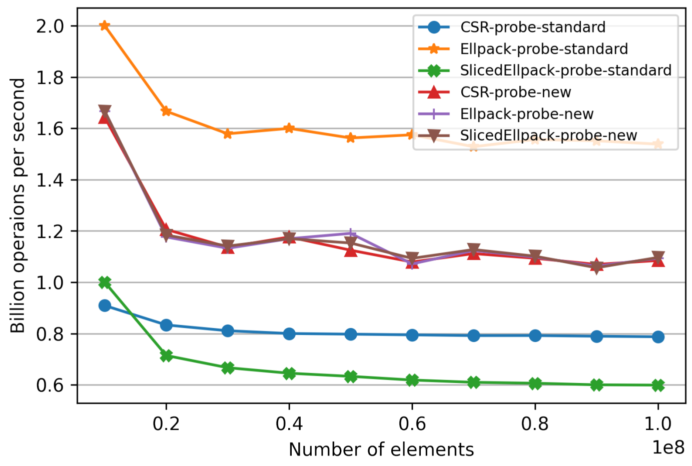

Figure 4.10: GPU retrieve performance of HashGraph-Probe-Standard 6 and HashGraph-Probe-New 7.

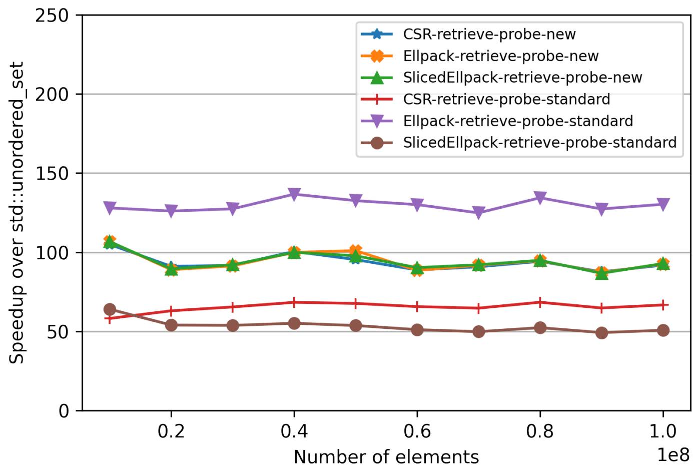

Figure 4.11: CPU retrieve speedup of `HashGraphSet` over `std::unordered_set`.

4.5.4 Hash Graph Map Performance

In this section, we compare the performance of `HashGraphMapDynamic` to the WarpCore and SlabHash implementations. We store and retrieve n key-value 4-byte integer pairs $((1,1),\dots,(n,n))$.

For hash graph map, we test both the array-of-structures and the structure-of-arrays storage types, referred to as *aos* and *soa* in the measurements.

4.5.4.1 Build

Figure 4.12 shows the performance of individual implementations when building the table. We see that WarpCore is clearly the most performant in this scenario. One possible reason might be that WarpCore uses a single atomic store instruction to store the key-value pair, as long as the size of the pair does not exceed 8 bytes. Since we tested with 4-byte keys and 4-byte values, this reduces the number of memory writes by half. We believe that such optimization could be made for the hash graph as well, as long as the array-of-structures storage is used.

SlabHash seems to have a high constant overhead when initializing the table because the build rate increases with the size of the input set.

Our implementations seem to be able to process around half the number of items per second compared with the `HashGraphSet`. The map effectively utilizes twice the amount of memory, reducing the amount of effective cache available by half.

A surprising result is that the array-of-structures variant outperforms the structure-of-arrays one for all segment types, even with 4-byte values.

The same result presented differently is laid out in table 4.1, where we can see the time in milliseconds taken to build the hash tables.

N	**SlabHash**	**WarpCore**	**CSR-aos**	**E-aos**	**SE-aos**	**CSR-soa**	**E-soa**	**SE-soa**
10	196.0	13.0	35.0	53.0	40.0	33.0	35.0	33.0
20	208.2	26.0	77.9	130.0	95.0	68.6	89.0	75.0
30	218.0	41.0	119.2	196.0	145.0	106.9	140.0	119.3
40	226.7	54.0	162.1	263.6	197.0	143.8	189.0	163.2
50	232.6	68.0	212.0	331.9	251.9	185.5	238.3	208.9
60	245.9	82.0	259.0	401.0	307.4	224.5	287.8	254.7
70	252.4	95.0	307.4	482.4	366.4	266.7	339.3	304.7
80	258.8	109.1	354.2	539.0	415.9	312.9	387.6	349.4
90	272.8	123.0	404.5	607.9	475.8	355.6	437.8	397.2
100	280.8	137.0	460.2	679.1	537.1	407.8	490.0	447.9

Table 4.1: Time in milliseconds to build the hash table of *N* million key-value pairs on a GPU. *Ellpack* is shortened to *E*, *Sliced Ellpack* is shortened to *SE*.

4.5.4.2 Retrieve

Retrieval performance measurements follow. We first show the HashGraph-Probe-Standard algorithm in Figure 4.13. SlabHash shows excellent performance, followed by Ellpack using the structure-of-arrays layout and Warp-

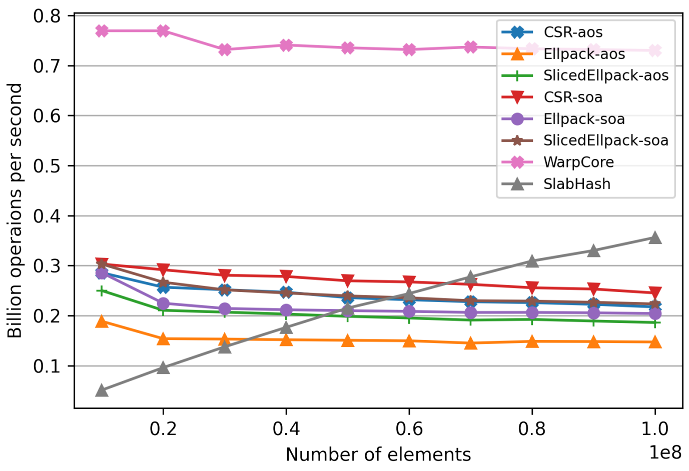

Figure 4.12: GPU build performance of `HashGraphMapDynamic`, `WarpCore` and `SlabHash`.

Core.

Figure 4.14 shows the usage of HashGraph-Probe-New. We can see that rearranging the data when building the second hash graph is again dominating the total time. While the Ellpack variants are slower when using this algorithm, CSR and Sliced Ellpack are somewhat faster.

The measurements of both algorithms are also summarized in Tables 4.2 and 4.3.

N	**SlabHash**	**WarpCore**	**CSR-aos**	**E-aos**	**SE-aos**	**CSR-soa**	**E-soa**	**SE-soa**
10	4.0	6.0	11.0	6.0	11.0	11.0	6.0	11.0
20	9.0	13.0	24.0	14.0	29.0	24.0	13.0	29.0
30	13.0	20.0	36.0	21.0	46.0	37.0	20.0	46.0
40	18.0	27.0	49.0	28.0	64.0	49.0	27.0	64.0
50	22.0	34.0	61.0	35.0	82.0	62.0	34.0	82.0
60	27.0	40.6	73.4	43.0	100.0	74.0	41.0	100.0
70	31.0	47.0	86.0	54.4	118.0	87.0	47.0	117.8
80	35.7	54.0	98.0	63.0	135.8	99.0	55.0	135.3
90	42.0	61.0	110.4	74.1	153.0	112.0	62.0	153.0
100	45.0	68.0	123.0	85.7	171.0	124.0	69.0	171.0

Table 4.2: Time in milliseconds to retrieve N million key-value pairs from the hash table on a GPU using the HashGraph-Probe-Standard algorithm 6. *Ellpack* is shortened to *E*, *Sliced Ellpack* is shortened to *SE*.

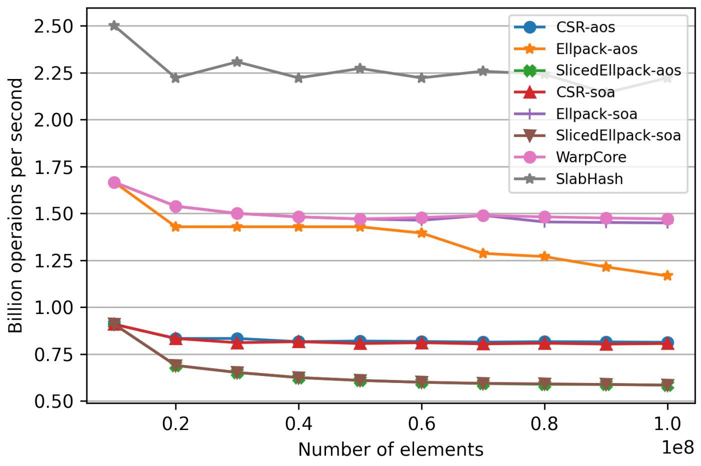

Figure 4.13: GPU retrieve performance of `HashGraphMapDynamic` using the HashGraph-Probe-Standard algorithm 6, `WarpCore` and `SlabHash`.

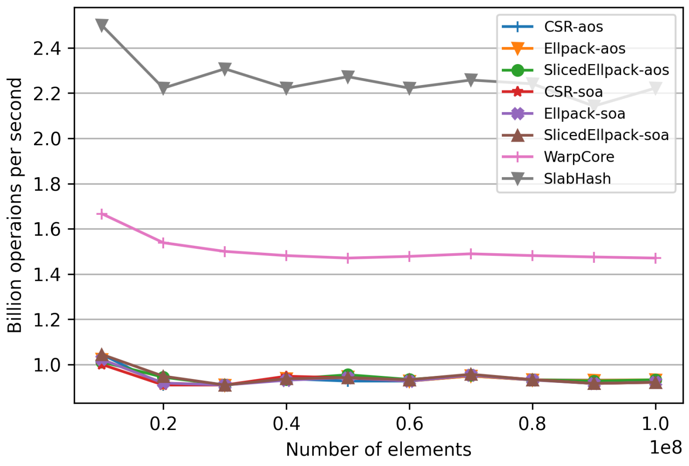

Figure 4.14: GPU retrieve performance of `HashGraphMapDynamic` using the HashGraph-Probe-New algorithm 6, `WarpCore` and `SlabHash`.

4.5.4.3 Insert

Finally, we measure the insert performance. We insert n key-value pairs $\{(1,1),\ldots,(n,n)\}$, but instead of inserting them all at once into an empty table, we first allocate an empty table with enough space and then we insert the items in 100 iteration. In the first iteration, we insert items

$$\{(1,1),\ldots,(n/100,n/100)\},$$

N	SlabHash	WarpCore	CSR-aos	E-aos	SE-aos	CSR-soa	E-soa	SE-soa
10	4.0	6.0	9.6	9.8	9.9	10.0	9.8	9.6
20	9.0	13.0	22.0	21.8	21.2	22.0	21.8	21.1
30	13.0	20.0	33.0	33.0	33.0	33.0	33.0	33.0
40	18.0	27.0	42.7	42.7	42.8	42.2	43.0	42.7
50	22.0	34.0	54.0	52.9	52.4	53.2	53.1	53.1
60	27.0	40.6	64.8	64.8	64.3	64.8	64.8	64.4
70	31.0	47.0	73.8	73.8	73.5	73.5	73.5	73.2
80	35.7	54.0	85.9	85.9	85.8	85.7	85.9	85.8
90	42.0	61.0	96.7	96.8	97.1	98.3	98.3	98.3
100	45.0	68.0	107.5	107.3	107.3	108.6	108.5	108.6

Table 4.3: Time in milliseconds to retrieve N million key-value pairs from the hash table on a GPU using the HashGraph-Probe-New algorithm 7. *Ellpack* is shortened to *E*, *Sliced Ellpack* is shortened to *SE*.

in the second iteration we insert

$$\{(n/100+1, n/100+1), \dots, (2n/100, 2n/100)\},$$

and so on.

Note that in this scenario, the `HashGraphMap` behaves more like an open-addressing hash table with linear probing, and all the underlying segment types allocate the same amount of slots. Thus, we expect Ellpack to do well while not having any memory overhead compared to CSR.

Results are displayed in Figure 4.15, and WarpCore again has superior performance. Interestingly, the array-of-structures version of Ellpack outperforms the structure-of-arrays one, unlike in previous tests. Sliced Ellpack is consistently one of the worst performers among the tested implementations.

N	SlabHash	WarpCore	CSR-aos	E-aos	SE-aos	CSR-soa	E-soa	SE-soa
10	67.5	12.0	30.0	25.0	29.0	33.0	28.0	33.0
20	56.1	24.0	56.0	46.0	61.1	66.0	56.0	70.0
30	80.3	36.0	82.0	66.0	94.0	98.0	83.0	108.0
40	106.3	48.0	107.0	85.5	125.0	130.0	111.0	145.0
50	132.7	60.0	131.0	105.0	156.0	161.9	138.0	181.9
60	295.3	72.1	156.0	124.5	186.9	193.0	165.0	218.0
70	391.0	84.0	180.8	144.0	217.0	225.0	192.0	254.0
80	208.9	96.0	205.0	163.0	248.0	257.0	219.0	290.5
90	458.4	108.0	230.0	183.0	279.0	289.0	246.1	327.0
100	1403.7	120.0	254.8	202.0	309.0	321.0	273.3	363.0

Table 4.4: Time in milliseconds to insert N million key-value pairs into the hash table on a GPU. *Ellpack* is shortened to *E*, *Sliced Ellpack* is shortened to *SE*.

4.5.5 Retrieval After Dynamic Insertion

We conclude this chapter by showing a special case for retrieval of items in `HashGraphMapDynamic`. We first allocate an table T and insert n items into

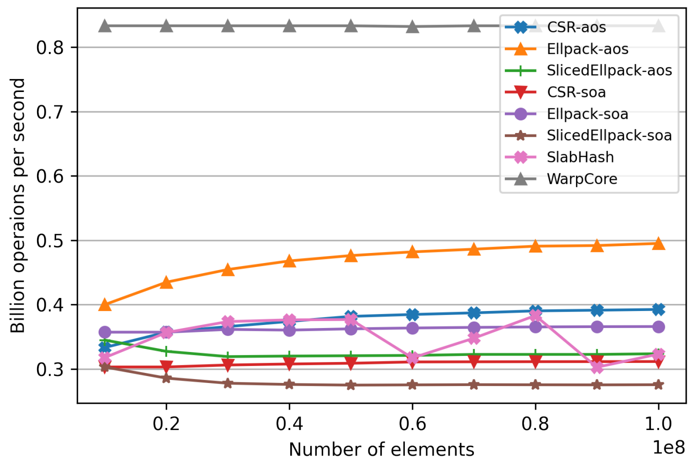

Figure 4.15: GPU insert performance of `HashGraphMapDynamic` using V2 build algorithm 2, `WarpCore` and `SlabHash`.

it in batches of $n/100$ like in section 4.5.4.3. This results in some items $x \in T$ not being stored in bucket $b_{h(x)}$. Thus, T behaves like an open-addressing table with the linear probing scheme.

The purpose of this last test is to show that the performance of retrieving items scattered outside their original buckets does not differ substantially from retrieving right after a build, where each item $x \in T$ is stored in the bucket $b_{h(x)}$.

The results of this test for HashGraph-Probe-Standard and HashGraph-Probe-New algorithms are shown in Figures 4.16 and 4.17, respectively. We can see that the performance is not degraded compared to retrieving items right after build, which is shown in Figures 4.13 and 4.14. HashGraph-Probe-New is again limited by the time building the second hash graph.

4.6 Commentary

Our experimental tests show that the GPU-based hash tables are faster than STL's sequential containers by order of 100× in our testing environment.

4.6.1 Segment Types

We conclude the following about using different sparse matrix storage formats for hash graphs:

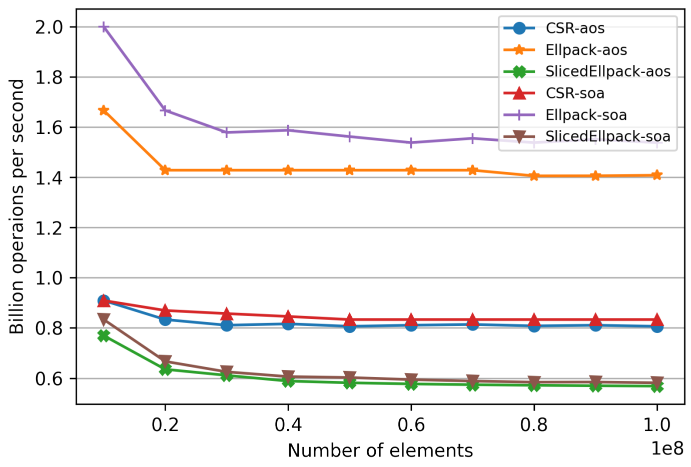

Figure 4.16: GPU retrieve performance of `HashGraphMapDynamic` using HashGraph-Probe-Standard algorithm 6 after dynamically inserting items.

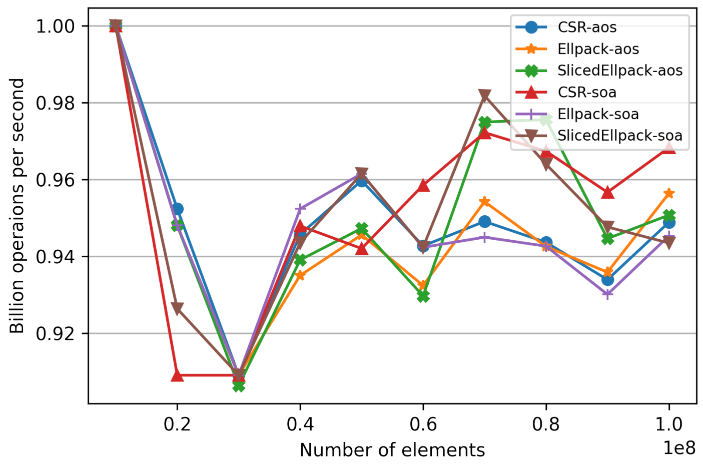

Figure 4.17: GPU retrieve performance of `HashGraphMapDynamic` using HashGraph-Probe-New algorithm 6 after dynamically inserting items.

- The CSR storage was usually the fastest for building the table, and it also uses the least amount of memory.
- The Ellpack storage has superior retrieval performance at the cost of a potentially much larger memory footprint. It also trails CSR when building the table, which we think is caused by the large number of empty slots that need to be initialized.

- The Sliced Ellpack does not seem to be the best choice in any scenario, and like the classic Ellpack, it requires more memory.

4.6.2 Dynamic Hash Graph

The dynamic version of the hash graph we introduced in section 2.4.3 seems to have the same performance when retrieving items compared to the static version. It is slightly slower at building the table due to the additional slots being initialized.

4.6.3 AOS and SOA Storage

When it comes to our map implementations, they are generally around two times slower than the corresponding set version. We believe further work is needed to investigate the root cause of this relatively large discrepancy.

The structure-of-arrays storage type seems to outperform the array-of-structures one, except for the dynamic insert test case.

4.6.4 Comparison to State-Of-The-Art Implementations

Our tests show that the WarpCore implementation consistently shows the best performance, with the one exception being retrieval, where SlabHash is around 50% faster.

Our map implementations lag behind WarpCore in all build tests, and only the Ellpack variant is able to match it for retrieval.

Chapter 5

Conclusion

The goal of this work was to study the HashGraph algorithm, implement it using the Template Numerical Library, and study the impact of using different sparse matrix storage formats for HashGraph's internal representation. Additionally, our aim was to introduce a dynamic version of HashGraph capable of supporting the insertion and deletion of elements.

We first outlined concepts about GPU hardware and software architecture, followed by an introduction to TNL. We followed by describing the theoretical concepts behind hashing and formalizing the hash graph data structure. Furthermore, we showed the relationship between the hash graph build algorithm and sorting.

We described the CSR, Ellpack, and Sliced Ellpack sparse matrix storage formats, along with their representation in TNL called Segments. We also proposed two different approaches for extending hash graphs to support storing key-value pairs. Finally, a new dynamic version of hash graph capable of supporting the insertion and deletion of items was introduced.

The concepts described in the theoretical part were implemented using TNL, and the performance of this implementation was evaluated against existing state-of-the-art GPU solutions.

5.1 Future Work

It seems our map implementations are unnecessarily slower than their set counterparts. Further work needs to be done in order to see if any improvements can be made in this regard.

Usage of some techniques employed by WarpCore may be brought over to the hash graph implementation, like using cooperative groups for each individual operation to reduce warp divergence, or combining storage of key-value pair into one atomic instruction.

The dynamic hash graph presented in this thesis uses linear probing to access elements outside the first bucket. Analysis of different probing schemes is another thing to be considered.

Last but not least, a variety of sorting algorithms may, in theory, be used to build a hash graph. It remains to be seen if this has practical use cases.

Bibliography

[1] Green, O. HashGraph – Scalable Hash Tables Using A Sparse Graph Data Structure. 2019, doi:10.48550/ARXIV.1907.02900. Available from: `https://arxiv.org/abs/1907.02900`

[2] Oancea, B.; Andrei, T.; et al. GPGPU Computing. *CoRR*, volume abs/1408.6923, 2014, 1408.6923. Available from: `http://arxiv.org/abs/1408.6923`

[3] NVIDIA CORPORATION. CUDA C++ Programming Guide. Available from: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

[4] Etiemble, D. 45-year CPU evolution: one law and two equations. *CoRR*, volume abs/1803.00254, 2018, 1803.00254. Available from: `http://arxiv.org/abs/1803.00254`

[5] Yang, X.; Parthasarathy, S.; et al. Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining. *Proceedings of The Vldb Endowment - PVLDB*, volume 4, 03 2011.

[6] Oberhuber, T.; Klinkovský, J.; et al. Template Numerical Library. Available from: `https://tnl-project.org`

[7] Mareš, M. Lecture notes on data structures. Available from: `http://mj.ucw.cz/vyuka/dsnotes`

[8] Pagh, R.; Rodler, F. F. Cuckoo hashing. *Journal of Algorithms*, volume 51, no. 2, 2004: pp. 122–144, ISSN 0196-6774, doi: https://doi.org/10.1016/j.jalgor.2003.12.002. Available from: `https://www.sciencedirect.com/science/article/pii/S0196677403001925`

[9] Shahnaz, R.; Usman, A.; et al. Review of Storage Techniques for Sparse Matrices. In *2005 Pakistan Section Multitopic Conference*, 2005, pp. 1–7, doi:10.1109/INMIC.2005.334453.

[10] Sengupta, S.; Harris, M.; et al. Scan Primitives for GPU Computing. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, edited by M. Segal; T. Aila, The Eurographics Association, 2007, ISBN 978-3-905673-47-0, ISSN 1727-3471, doi:10.2312/EGGH/EGGH07/097-106.

[11] Arkhipov, D. I.; Wu, D.; et al. Sorting with GPUs: A Survey. *CoRR*, volume abs/1709.02520, 2017, `1709.02520`. Available from: `http://arxiv.org/abs/1709.02520`

[12] Kolonias, V.; Housos, E. A High-Performance Implementation of Counting Sort on CUDA GPU. 02 2011.

[13] Bell, N.; Garland, M. Efficient Sparse Matrix-Vector Multiplication on CUDA. 01 2009.

[14] Oberhuber, T.; Suzuki, A.; et al. New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA. *CoRR*, volume abs/1012.2270, 2010, `1012.2270`. Available from: `http://arxiv.org/abs/1012.2270`

[15] Busato, F.; Green, O.; et al. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–7, doi:10.1109/HPEC.2018.8547541.

[16] Lessley, B. Data-Parallel Hashing Techniques for GPU Architectures. *CoRR*, volume abs/1807.04345, 2018, `1807.04345`. Available from: `http://arxiv.org/abs/1807.04345`

[17] Alcantara, D. A.; Volkov, V.; et al. Chapter 4 - Building an Efficient Hash Table on the GPU. In *GPU Computing Gems Jade Edition*, edited by W. mei W. Hwu, Applications of GPU Computing Series, Boston: Morgan Kaufmann, 2012, ISBN 978-0-12-385963-1, pp. 39–53, doi:https://doi.org/10.1016/B978-0-12-385963-1.00004-6. Available from: `https://www.sciencedirect.com/science/article/pii/B9780123859631000046`

[18] Khorasani, F.; Belviranli, M. E.; et al. Stadium Hashing: Scalable and Flexible Hashing on GPUs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 63–74, doi:10.1109/PACT.2015.13.

[19] Team, R. D. cuDF - GPU DataFrame Library. Available from: `https://github.com/rapidsai/cudf`

[20] McKinney, W. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, edited by S. van der Walt; J. Millman, 2010, pp. 51 – 56.

[21] Ashkiani, S.; Farach-Colton, M.; et al. A Dynamic Hash Table for the GPU. *CoRR*, volume abs/1710.11246, 2017, `1710.11246`. Available from: `http://arxiv.org/abs/1710.11246`

[22] Jünger, D.; Kobus, R.; et al. WarpCore: A Library for fast Hash Tables on GPUs. *CoRR*, volume abs/2009.07914, 2020, `2009.07914`. Available from: `https://arxiv.org/abs/2009.07914`

[23] SMHasher. Available from: `https://github.com/aappleby/smhasher`

[24] Google. GoogleTest - Google Testing and Mocking Framework. Available from: `https://github.com/google/googletest`

[25] Jünger, D.; Kobus, R.; et al. WarpCore: A Library for fast Hash Tables on GPUs. Available from: `https://github.com/sleeepyjack/warpcore`

[26] Ashkiani, S.; Farach-Colton, M.; et al. SlabHash - A warp-oriented dynamic hash table for GPUs. Available from: `https://github.com/owensgroup/SlabHash`

[27] Tripathy, A.; Fender, A. Hashgraph - Multi-GPU Hash Table Backed by Sparse-Graph Data Structures. Available from: `https://github.com/alokpathy/hashgraph`

Appendix **A**

Contents of CD

```
README.md ........................ the description of media contents
src/ ............................. the source code related to this work
  README.md ...................... the description of the implementation
  src/include/ ................... the implementation of data structures
  src/benchmark/ ................. the benchamring framework used
  src/test/ ...................... the unit tests
  third_party/ ................... the source code of third party hash tables
thesis/ .......................... the LaTeX source files of this thesis
thesis.pdf ....................... this thesis in PDF format
```