FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Algorithms and data structures for hashing on GPU
Student:	Askar Kolushev
Supervisor:	Ing. Tomáš Oberhuber, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2021/22

Instructions

1. Study the basics of programming GPU using CUDA.
2. Learn the fundamentals of the development of parallel algorithms with TNL library (www.tnl-project.org).
3. Learn and understand parallel hashing algorithms for GPUs, namely Cuckoo hashing and hashgraph algorithm.
4. Implement both algorithms into TNL library to run on CPU and GPU.
5. Implement unit tests for testing correctness of the implemented algorithms.
6. Perform measurement of speed-up compared to appropriate containers in STL library and [1] based on data from real problems (hashing of unstructured numerical meshes for example).

[1] https://github.com/alokpathy/hashgraph

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague November 30, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Algorithms and data structures for hashing on GPU

Askar Kolushev

Department of Informatics
Supervisor: Ing. Tomáš Oberhuber, Ph.D.

January 5, 2021

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on January 5, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kolushev, Askar. *Algorithms and data structures for hashing on GPU.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Hešovací tabulka je typ datové struktury, která umožňuje vložení a vyhledávání klíčů se složitostí O(1). Některé hešovací tabulky jsou optimalizované pro práci na grafických procesorech (GPU). Tato práce je zaměřena na studium různých typů hešovací tabulek a algoritmů a jejich implementaci pro GPU. Cílem je seznámit se s programováním pomocí nástroje CUDA® a knihovny Template Numerical Library (TNL - www.tnl-project.org). Teoretická část popisuje podstatu hešování a představuje některé datové struktury a algoritmy přizpůsobené pro paralelní zpracování na GPU. V praktické části jsou některé z těchto přístupů (jmenovitě Cuckoo hashing a HashGraph) implementovány s pomocí TNL. Nakonec je implementace testována. Testy porovnávají výkon na GPU a CPU s implementací kontejneru std::unordered_set ze standardní šablonové knihovny (STL) jazyka C++. Výsledky ukazují, že všechny testované přístupy fungují na GPU mnohem rychleji. Nejrychlejším nalezeným přístupem byl HashGraph verze 1.0. Ten je nejlepší pro použití v mnoha aplikacích, které vyžadují vložení a vyhledávání neuspořádaných klíčů na GPU.

Klíčová slova Hešování, hešovací tabulka, grafický procesor, GPU, CUDA, Template Numerical Library, TNL, Cuckoo hashing, HashGraph.

Abstract

A hash table is a type of data structure that enables insertion and probing of keys having same format with an average complexity of O(1). Some hash tables are also more optimized for working on graphics processors (GPU). The goal of this work is to get familiar with programming for GPU using CUDA® framework and Template Numerical Library (TNL - www.tnl-project.org) and use it to realize several hashing approaches. Theoretical part describes the idea of hashing and introduces some data structures and algorithms intended to adapt it for parallel programming for GPU. In practical part, some of these approaches (namely, Cuckoo hashing and HashGraphs) are implemented using TNL. Finally, the implementation is tested. The tests compare its performance on GPU and CPU with the implementation of std::unordered_set container in Standard Template Library of the C++ language. The results show that all tested approaches perform faster on GPU. The fastest approach found was HashGraph of version 1.0. Thus, it is the best examined choice to be used in many applications that require storing and probing unordered keys on GPU.

Keywords Hashing, hash table, graphics processing unit, GPU, CUDA, Template Numerical Library, TNL, Cuckoo hashing, HashGraph.

Contents

List of Figures

List of Tables

Introduction

Motivation and objectives

Graphical processing units (GPU) are fundamentally different from the Central processing units (CPU) in terms of their ability to perform specific types of tasks. The design of a GPU is focused on optimization of parallel computations at a cost of efficiency on sequential operations. That creates performance issues while implementing there the techniques specifically designed for the CPU. To overcome this problem, new algorithms and data structures need to be created and tested that would exploit the GPU's advantages on parallel computing without suffering from their drawbacks. One example of a traditional task in Computer Science is to store large amount of data that have same format but different values.

There exist numerous types of data structures and related algorithms that solve this problem. All those approaches vary in time and memory efficiency while performing specific operations. One popular data structures fulfilling said purpose is called hash table.

It's biggest advantage is the ability to insert new elements (rows) and query previously inserted ones with a constant average cost.

The idea of a hash table is to introduce a hash function that would find a hash value for each piece of data inserted. A hash function maps each data element to a particular value of fixed size (an integer in our case). The element is then inserted into a 1D array with its hash value as the index (called the row of the table).

Assuming that the hash function itself has time complexity of O(1), the operations of element insertion and probing has constant complexity in most cases. This makes hash tables very useful in implementation of sets and maps, because those classes are focused on fast accessing of the elements by value and do not depend on their order. The only situation when this access is not constant happens when two different data elements have the same hash value and are thus supposed to be inserted into the same row of the table. This

event is called a collision, and different types of hash tables employ various strategies for handling it.

Another feature that differentiates these types is the support of *dynamic allocation* in some of them. It supposes an ability to add new entries after the initial building is finished. *Static allocation* only allows insertion of the elements on the building stage.

Three examples of hash tables and associated algorithms were considered:

- Cuckoo hashing
- HashGraph V1
- HashGraph V2

These data structures were designed specifically for parallel usage with the performance on GPU in mind.

Implementation of those algorithms for GPU requires using special tools. One relatively simple instrument suitable here is CUDA® – Compute Unified Device Architecture. It is a parallel computing platform and programming model developed by NVIDIA. It provides a high-level interface for developing programs on a number of languages that allows to easily execute specified parts of code in parallel on GPU kernels while the rest of project is compiled for the CPU.

This programming model introduces additional issues to the developing process that require the developer to pay special attention to some details of function design and memory management. Template Numerical Library (TNL®) is a project whose idea is to develop a library of template classes for C++ language. Its functionality allows to overcome some of the issues related to programming with CUDA by providing a unified interface for implementing parallel algorithms on both CPU and GPU with focus on numerical calculations and linear algebra.

Problem statement

The purpose of this work is to familiarize with nuances of programming for GPU using CUDA, then study the basic functionality of the TNL library and extend it by providing a set of classes implementing the types of hash tables mentioned above. Performance of these algorithms needs to be compared with each other as well as with the implementation of similar functionality from the standard C++ library.

Chapter 1

State-of-the-art

As a well-known data structure in Computer Science, the hash table was explored for a long time and was developed in many forms. The main reason to invent new versions of the structure and related algorithms is the necessity to tackle the problem of hash collisions between the inserted elements. Most of the specific types of hash tables can be grouped by in the following way based on their approach to this problem [1]:

- *Open-addressing* – insertion attempts for each key are repeated to a new position after each collision until a free location for it is found. This requires using multiple hash functions (one for each new candidate position). Iterating over possible rows generates additional overhead which becomes greater as the load factor increases. When the table is full, the process of insertion transforms into an infinite loop. The solution in this case is to reconstruct the table from scratch using larger number of rows (and thus increased range of hash values).

- *Perfect hashing* – hash function is sequentially constructed in such a way that no collisions happen in the end. This ensures that the query will always complete in a single step, but doesn't allow dynamic allocation.

- *Separate chaining* – multiple elements are hashed into the same row in case of collision. As the result, instead of just one unique element, each row stores a pointer to the first node of a linked list of keys hashed into it. The query operation in this case performs a linear search within all elements sharing the same hash value as the queried key.

- *Spatial hashing* – applied for looking up geometric primitives in a 2D or 3D space divided into uniform cells. Each object is mapped to its cell, and each cell is hashed into a hash table based on its positional coordinates.

A specific approach must be taken for each of these types to make them efficient on GPU. That means developing unique structures and algorithms that would take advantage of GPU parallelism. Some of the most popular approaches follow:

- Open-addressing
  - *Cuckoo hashing* – in case of a collision, the inserted element is extracted from the table and then inserted back using another hash function [2].
  - *Double hashing* – new hashes are calculated for the element until it can be inserted into corresponding position [3].
  - *HashGraph* – compressed sparse row (CSR) data structure is used to represent the table contents in a form of a graph [4].
  - *Robin Hood hashing* – key ages are tracked and the youngest keys are evicted and replaced with the oldest ones [5].
  - *Stadium hashing* – data are stored in CPU while an auxiliary structure called Ticket board is used on GPU to access it [6].
  - *WarpDrive* – coalesced groups (CG) of threads are used to parallelize linear probing [7].
- Perfect hashing
  - *Perfect spatial hashing* – table and hash function are constructed over a sparse set of multi-dimensional spatial data while ensuring locality of hashed points [8].
- Separate chaining
  - *Slab hash* – a GPU-optimized data structure called slab list is used instead of a traditional linked list [9].
- Spatial hashing
  - *Compact spatial hashing (CSH)* – data are written into a sparse table with perfect hashing. Large number of empty entries in the resultant table are exploited to compress it to a size proportional to the number of keys divided by load factor [10].
  - *Exclusive grouped spatial hashing (EGSH)* – compacts the table by gathering points with same data values into groups to avoid storing duplicates [11].
  - *Voxel hashing* – the world is partitioned into several voxels (values in 3D grid), each containing multiple points. The points are hashed based on the coordinates of voxels they are assigned to [12].

Chapter 2

Theory

2.1 General structure of GPU. Streaming multiprocessors, warps and memory hierarchy.

Computational power of a GPU is represented by a number of *streaming multiprocessors (SM)*. Each SM combines several GPU cores and is responsible for processing a number of parallel threads grouped into *thread warps* (usually by 32 in each). This structure creates a hardware connection between threads that introduces logical implications for the program design. Compared to threads on multi-core CPU that perform different actions and only require synchronization at specific points, GPU threads do not act as independently. Instead, each operation is executed simultaneously on all threads of the warp assigned to the SM. This approach is called *Single Instruction Multiple Threads (SIMT)*.

Based on lifetime duration and availability to the threads, GPU memory is structured into the following levels:

- *Thread private local memory* is available to a single thread and is persisted until its execution is finished.
- *Block shared memory* is common for all threads of a single block and is freed when all of them finish execution.
- *Global memory* is available to all threads within the application and is persisted through the whole run of the program.
- *Constant and texture* memory spaces are shared and persisted just as global memory, but they are read-only.

Apart from the listed memory levels, one should also consider the hierarchy of device's memory cache. Cache is another form of memory used is many types of processors. It is characterized by smaller volume but much shorter

latency. A GPU uses two levels of cache called L1 and L2. Each unit of the smaller L1 is utilized by a single SM while the larger L2 is used commonly by all L1 units.

The purpose of cache memory in GPU, just like on CPU, is to speedup memory accesses. Latency of L1 is higher than of L2, and the main memory is slower than both of them. The idea is thus to store data to and read it from the faster cache device instead of looking up the main memory. Its smaller size, however, requires synchronizing the levels with each other. To increase the number of *cache hits* (successful reads of data from the cache without accessing the next level), the data are cached in a way that respects *temporal* and *spatial locality*. The former principle assumes that recently accessed variables and arrays will be accessed again soon, and thus should be copied to the cache for that case. The latter one supposes that the accesses are done sequentially, meaning that future memory reads will be performed to addresses next to the recently accessed ones. Thus, caching is done in large portions of sequentially stored data rather than its single units. Understanding this structure gives an advantage in a form of performance boost. To properly exploit temporal and spatial locality, one should aim to access same pieces of data repeatedly and move over data sequentially instead of jumping between the addresses located far from each other.

2.2 General-purpose computing on GPU and CUDA® programming model

The structural differences between the CPU and the GPU make thread execution on the latter one much worse optimized for sequential operations. On the other hand, this construction allows it to support thousand of parallel threads at the same time (compared to the average of 8 to 16 threads on most modern CPUs).

This makes the GPU extremely helpful in specific applications like data processing, machine learning, numerical calculations and, most of all, graphics processing. These areas employ algorithms that suppose performing simultaneous computations of similar type on numerous elements of data, and thus benefit from large number of parallel threads. However, natural inefficiency of each separate GPU thread due to poor optimization of flow control and data caching make them impossible to use as the only processing unit in a system. The only sensible approach in this case is to make use of both CPU and GPU in a way that would allow us to benefit from their advantages. The goal of a programmer is thus to identify in which stages of the algorithm advantages of the GPU would outweigh its drawbacks and load it with work exactly there. Evidently, this creates a need for an interface that would allow for easy switch of the work focus between the two devices. CUDA programming model by NVIDIA was developed with exactly this idea in mind.

CUDA provides a software development environment that introduces specific extensions for a list of high-level languages to support its own programming model scalable for automatically adapting an application to different number of processing cores. The language used in this work was C++. These extensions include new standard functions, structures, macros and even slight syntax modifications.

2.2.1 Kernel and threads. Thread hierarchy and synchronization

The model itself is based on an idea of a *kernel* represented as a user function that is being run in parallel. All stored data and running functions that are created on the CPU and the GPU are referred to as *device* and *host* respectively. Execution starts in a standard way for the C++ language on the host, and an altered syntax is used to start the kernel function on a provided number of threads.

Parallel threads in CUDA are organized into 1, 2 or 3-dimensional *blocks*. Within each block, its threads are distinguished by unique indices that are represented by 3-element integer vectors and are accessible from inside the kernel code through a variable called `threadIdx`. As all threads of a single block are run on the same processor core, the number of these threads is limited. The size of the block can be learned by any kernel from a vector variable called `blockDim`. If the required quantity of threads exceeds this limit, multiple blocks can be started to run the same kernel. One should also remember that the threads within one block are forced to share limited resources of the core it is running on. Just like the threads, blocks are placed into a grid with up to 3 dimensions and their vector indices are available to the threads by reading `blockIdx` variable.

By combining dimensions and index of its block with its own coordinates within that block, each thread can uniquely identify itself and the piece of data it is supposed to process. This simplifies adaptation of kernels for processing data containers with up to three dimensions (arrays, matrices and volumes). For example, when working with $N * M$ matrices, the kernel are started in two dimensions bounded by N and M, and coordinates of each kernel tell it which element of the matrix is to be processed by it.

The number of warps needed to run all required threads can exceed combined capacity of all SMs. This would create a need to start some of them one by one rather than in parallel. Moreover, this behavior cannot be predicted as the scalability of the model supposes that an application should adapt to different kinds of GPUs it could be run on. Thus, one can never assume the execution order of thread blocks. It is impossible to synchronize all threads in a grid, however on the block level it can be done by calling `__syncthreads()` function. It acts as a barrier for the threads and does not let them continue their execution until the rest of them reach that point.

2.2.2 Memory management in CUDA

Memory management in CUDA is similar in its principles to simple memory management used in C language (with functions like `malloc()`, `memcpy()`, `free()` etc.). CUDA provides a set of functions to allocate, copy, move and free data (most widely used thus being `cudaMalloc()`, `cudaMemcpy()` and `cudaFree()`) - basically, with the same functionality except that they operate on the memory of GPU. This introduces a difficulty when operating on two devices that is discussed below.

2.2.3 Common issues and limitations

It is a programmer's responsibility to allocate and deallocate the memory for data, and they should also keep in mind that the two devices have separate memories and address spaces, which makes it impossible to access the same data from the host and the device interchangeably (for example, by using pointers). In other words, one cannot construct an object by the CPU and then pass it to a CUDA kernel by reference (or vice-versa). Being nothing more than an address in the device's memory, the pointer will direct the kernel to data in the host memory, and thus will be useless for a device not having an access to it. To partially solve this problem, CUDA provides operations for transferring data between the two devices (for example, `cudaMemcpy()` with the 4th enum parameter specifying direction of copying – e.g. `cudaMemcpyHostToDevice`). The problem is that copying data between the devices is a very slow operation (especially when its amount is large), and thus should be used as little as possible.

A well-known problem while working with multi-threaded applications is race conditions. It is a situation when two or more threads attempt to write data into the same memory, which causes undefined behavior. CUDA provides a solution for that problem in a form of *atomic operations*. It is a set of functions that, when being called from one thread on some value in the memory, block all other threads that write into the same memory until the operation is finished. As an example, CUDA's

```
int atomicAdd(int* address, int val)
```

adds *val* to the value stored to *address* and returns its previous value. All other threads writing into *address* are blocked until adding is done. There exist other functions of various types but they only work with primitive types of integers and numbers with floating point.

SIMT approach can negatively affect performance when the threads *diverge* from each other. Thread divergence takes place when the algorithm steers the threads to different branches of execution, and it makes the performance dependent on the similarity of those branches. This includes if-else-then statements and loops of variable length

2.3 TNL - Template Numerical Library

Template Numerical Library comprises multiple classes for C++ language. One of its goals is to re-implement and extend the functionality of Standard template library (STL) of C++ for scientific computations. By setting specific parameters, it allows to choose which device (GPU or CPU) must be used for data storage and operation execution. Targeting for general-purpose GPU usage allows it to be especially focused on sparse linear algebra and numerical meshes.

At the current stage of development, TNL provides classes for unified memory management between devices, instruments for parallel reduction, numerical solvers, structures representing sparse matrices and more. All these features simplify general-purpose programming for a GPU and exploit its advantages for parallelism. Some classes that were used in this work are explained next

2.3.1 Class TNL::Algorithms::ParallelFor

TNL's template class `TNL::Algorithms::ParallelFor` is dedicated to organizing parallel execution of kernels and their synchronization without using CUDA-specific macros, functions and syntax constructs. The class specialized by template parameters specifying the device and mode (synchronous/asynchronous) has only one static method `exec` that accepts bounds of the loop, a lambda-function representing its body and variadic parameters to that function. Lambda-function then accepts a current index in the for-loop and than can proceed with its actions accordingly. In case of Device passed being `TNL::Devices::Cuda`, lambda is run in parallel on the number of kernels specified by bounds. For `TNL::Devices::Host`, it is nothing but a for-loop standard for C and C++ languages. To support 3-dimensional execution of kernels, TNL provides classes `TNL::Algorithms::ParallelFor2D` and `TNL::Algorithms::ParallelFor3D` in a similar manner. The only difference is the number of bounds to be provided to `exec` method and integer indices accepted by the lambda-function.

An important note is that CUDA requires each kernel function to be marked with a `__device__` macro and `__host__` macro if it will be called from CPU as well. TNL unites them into a single `__cuda_callable__` macro that will only be resolved to `__host__ __device__` if the code is compiled with CUDA compiler. Same is true for the lambda-function used in `ParallelFor`.

2.3.2 Classes Array, StaticArray, Vector and ArrayView

Another major point to care about when programming with CUDA is working with memory. General principles are what is usual for manual memory man-

```
#include <TNL/Algorithms/ParallelFor.h>
#include <TNL/Devices/Cuda.h>

using namespace TNL::Algorithms;
using namespace TNL::Devices;

int main(void) {
  auto body = [] __cuda_callable__ (
    int i, int j, int k) mutable {
      printf("%d %d %d\n", i, j, k);
  };
  ParallelFor2D<Cuda>::exec(0, 0, 12, 15, body, 20);
  return 0;
}
```

Figure 2.1: Example of using `TNL::Algorithms::ParallelFor2D`. Each kernel prints three integers in ranges: 0..11, 0..14 and 20.

agement. However, the programmer must also keep in mind that two devices have separate memories. What is allocated on CPU is not accessible form GPU without calling special functions, and vice-versa. TNL employs a set of classes to simplify this task with a simple interface.

Class `TNL::Containers::Array` is a template container class used as a one-dimensional dynamic array. Its most important feature is that like many other TNL classes it allows to specify which device to allocate the data on. Thus, an object of this class stored on the Host device can point to data on the CUDA device. For the Device template argument set to `TNL::Devices::Host`, the class behaves almost like `std::vector` with some minor differences. For `TNL::Devices::Cuda`, some operations use the versions implemented for GPU in parallel. For example, `Array<T, Device>::setValue` sets all elements of the array to the same value in parallel.

In order to use the class properly, one has to remember that any data element can be accessed only from the device that it was allocated on. Copying between devices is possible, and the methods `getElement` and `setElement` allow working with GPU data from the CPU. However, as was mentioned before, those operations are slow, and must be used as little as possible. As it is always the case with data allocated on GPU, it can only be efficiently used in parallel operations. The way to do it in TNL (except for a limited number of methods that are provided in Array class itself) is by means of `TNL::Algorithms::ParallelFor`.

One problem with this is that the lambda function supplied as the body of the parallel for-loop is called from a CUDA kernel, and the host-function

calling it stores the Array instance in the CPU-memory. Thus, the lambda cannot accept the Array by reference. Data allocated by the Array is stored on the GPU, but passing the Array to the kernel by value will result in deep copying of the contents, which will generate an enormous overhead. To resolve this issue, Template Numerical Library provides a special set of classes called *View*. `TNL::Containers::ArrayView` and the views for some other classes give an access to data owned by the Array. In other words, they let the user read and/or change it, but do not allow to delete or allocate it. Passing it to a CUDA-kernel will trigger a shallow copy, thus saving a lot of execution time. This means that the data will be the same as pointed to by the original Array object, but the wrapper object will be different. Provided that the data itself is allocated on a CUDA device, it makes this approach perfect for using with `TNL::Algorithms::ParallelFor`. This also allows the lambda-function to accept an ArrayView in its capture list.

Class `TNL::Containers::Vector` is another container class derived from `TNL::Containers::Array` and in addition to memory management functionality, it is used for algebraic operations like Scan (see below). Another derived class, `TNL::Containers::StaticArray`, accepts a template parameter specifying its size at the compile time (analogous to `std::array`).

Both classes (and some not mentioned here) provide an access to their views through methods `getView()` and `getConstView()`.

2.3.3 TNL::Algorithms::Scan and Segments::CSR

`TNL::Algorithms::Scan` has an interface similar to `ParallelFor`. It is a template class containing only one static method `perform` that accepts input array, start and end indices, operation itself (in the form of a lambda function) and the initial value. Apart from the Device to execute the operation on, the class accepts another template parameter to choose between exclusive and inclusive scan (using `TNL::Algorithms::ScanType`). See Sec. 2.5.1.4 for theoretical explanation.

TNL also provides a set of classes for working with Segments. In this work, we are interested in a class called `TNL::Algorithms::Segments::CSR` that represents a popular Compressed sparse row format (see Sec. 2.5.1.2). Having all non-zero elements of a matrix written subsequently into a one-dimensional array, one can use an object of this class to index that array instead of the original matrix. Just like rows of a 2D array, the segments in the CSR represent matrix rows and have their own inner indexation. However, as the CSR only stores non-zero elements, the segments have unequal sizes. A CSR class instance keeps track of these sizes and simplifies indexing the "flattened" version of the original matrix. In other words, it provides a mapping from 2D indices of matrix elements to 1D indices of its compressed version. Thus, the object helps with indexing the data but does not store it.

```
#include <TNL/Algorithms/ParallelFor.h>
#include <TNL/Containers/Array.h>
#include <TNL/Containers/ArrayView.h>
#include <TNL/Devices/Cuda.h>

using namespace TNL::Algorithms;
using namespace TNL::Devices;

int main(void) {
  Array<int, Cuda> arr(25);
  auto assign = [] __cuda_callable__ (int i,
    ArrayView<int, Cuda> view) mutable {
      view[i] = i;
  };
  ParallelFor<Cuda>::exec(0, 25, assign,
    arr.getView());

  auto print = [] __cuda_callable__ (int i,
    typename Array<int, Cuda>::ConstViewType view) {
      printf("%d %d\n", i, view[i]);
  };
  ParallelFor<Cuda>::exec(0, 25, print,
    arr.getConstView());
  return 0;
}
```

Figure 2.2: Example of using `TNL::Containers::Array` and its view. Array elements are assigned number from 0 to 24, and then the contents are printed (both done in parallel).

In order to be aware of the index sizes, CSR class accepts them as a TNL `Vector` to either a constructor or a setter method.

The class has methods for learning the number of segments and the size of each of them. But most importantly, the method `getGlobalIndex` returns the index in the compressed array by accepting a segment index and an element index inside the segment.

Just like `Array` and texttttVector, `CSR` has its own View class called `CSRView` that can be used to access the segments and indices but cannot initialize it by accepting the segment sizes.

2.4 Cuckoo hashing

Cuckoo hashing is one version of an algorithm for managing hash tables and resolving hash collisions designed specifically to benefit from parallelism of GPU. Its main idea is based on assigning a set of possible position for insertion of each key using multiple hash functions. This guarantees the insertion in a constant time while still keeping the query time relatively short.

2.4.1 Overview

The main idea of this strategy is to exchange the newly inserted keys with previously existing ones and reinsert them using another hash function until no keys that need reinsertion are left. Probing in this case is just an iteration over hash functions looking for the actual position of the key we are interested in.

Each table therefore depends on three parameters that determine its behavior and performance. The work done by Alcantara et al. states the most optimal values for these parameters with respect to the number N of input elements.

- *Table size* - specifies how many rows are to be used in the table and thus the maximal hash value of each key (with 0 being minimum). The optimal size for N input keys was found to be $1.25 * N$.
- *Number of hash functions* - 4 is said to be the best choice to balance all metrics.
- *Maximum number of iterations* - needed to indicate an infinite loop produced by the building process (see below). Preferred value is $7 * log(N)$.

2.4.2 Construction

Starting the procedure of table creation is trivial.

The first step is to take the first hash function h_1 and insert the first key k_1 into the array representing the table under the position $h_1(k_1)$. If the same operation is performed with other keys, it will eventually lead to a collision in most cases. It happens when $h_n(k_i) = h_n(k_j)$ for $i < j$ and will lead to inserting the key k_i into a position that was previously occupied by k_j. The solution used in this type of algorithm is to exchange the newly inserted key with previous contents of its row and then check if evicted content was free. If not, then the next step is to identify which hash function h_n that content was previously inserted with. After that, the element is placed back to the table, but this time into position $h_{n+1}(k_i)$. If h_n was the last function in the list of available ones, then $h_0(k_i)$ is used instead.

The problem of this procedure is that it can eventually lead to an infinite loop of evictions and re-insertions of keys. Being a probabilistic approach,

Cuckoo hashing cannot guarantee our safety from this event, nor can we be sure that it does not happen. The only way to algorithmically detect this situation is to limit the maximum number of insertions that the operation can perform when started with a single key. This number is exactly what the third parameter described above stands for. In case the maximum number of operations is reached, it is a sign that the construction process cannot finish in current configuration, and the whole procedure needs to be restarted with different set of hash functions. All in all, the steps are described in Alg. 1 and 2.

```
   Input: TableEntry Entry, HashTable Table
   Result: Boolean value indicating success
 1 HFun := HFArray[0];
 2 for Int i := 0 to Iterations do
 3 |   Location = HFun(Entry);
 4 |   swap(Entry, Table[Location]);
 5 |   if Entry is empty then
 6 |   |   return True;
 7 |   end
 8 |   // Find next HashFunction to re-insert evicted entry
 9 |   for Int h := 0 to HashFunctions do
10 |   |   HFun := HFArray[h];
11 |   |   if HFun(Entry) = Location then
12 |   |   |   HFun := HFArray[(h + 1) mod HashFunctions];
13 |   |   |   break;
14 |   |   end
15 |   end
16 end
17 // Maximum number of iterations reached => report an error
18 return False;
```

Algorithm 1: Algorithm to insert a single entry into a Cuckoo hash table.

2.4.3 Probing

The probing mechanism is simple - each key queried is being hashed by all hash functions one by one. The values returned by them are being used as indices for the table elements until some of them yields an index of the element equal to the one being requested. If a relevant value has not been found after going through all hash function, the key is reported as missing.

```
   Input: EntryArray Input, Int TableSize, Int HashFunctions, Int
          Iterations
   Result: Cuckoo hash table with keys inserted
 1 Table := EmptyEntryArray(TableSize);
 2 repeat
 3 |   HFArray := RandomInit(HashFunctions, TableSize);
 4 |   foreach Entry in Input do
 5 |   |   Success := InsertOne(Entry, Table);
 6 |   |   if not Success then
 7 |   |   |   break;
 8 |   |   end
 9 |   end
10 until Success;
```

Algorithm 2: Algorithm to build a table from a list of entries.

```
   Result: An Entry if it was found, NULL otherwise
   Input: CuckooHashTable Table, Entry Probed
 1 foreach HFun in HFArray do
 2 |   Index := HFun(Probed);
 3 |   Found := Table[Index];
 4 |   if Found = Probed then
 5 |   |   return Found;
 6 |   end
 7 end
 8 // No hash function yielded an index of the probed key
 9 return NULL;
```

Algorithm 3: Key probing algorithm of Cuckoo hash table.

2.4.4 Improvements and parallelization

Modifying the building algorithm into a parallel version is quite straightforward. The actions needed are similar for each entry, and the structure guarantees finishing the operations in a constant time, which makes it naturally fitting for a GPU implementation. Thus, the idea is to insert all keys at the same time, thus transforming the for-loop on lines 4-9 of the Alg. 2 into a CUDA-kernel indexed by the index inside Input array. All threads then need to have an access to a global flag indicating a failure.

Probing in parallel employs a similar idea of devoting a separate thread for each key. Conveniently, the number of hash functions is constant for the entire table, which allows us to parallelize the algorithm on them as well. More on that is in the section 3.

One clear drawback of the introduced algorithm is the part of insertion operation on the lines 8-14 in *Algorithm 1.* The purpose of this for-loop is

to find which of the hash functions was previously used to insert the evicted entry to the table and find the new one to pick a new position for it. In the original version presented above, the operation is linear with respect to the total number of hash functions used by the table. Instead of doing this, the index of each hash function can be stored into the table together with the entry itself. It will replace the for-loop with a constant operation of accessing a single value from the array with known index, though it means that more memory will be used for storing these arrays.

2.5 HashGraph

HashGraph employs a different approach to hashing. This concept represents hash values and respective data entries as vertices of a bipartite graph. Correspondence between the keys and their hash values is viewed as the graph's edges, and all these data are stored into a memory-efficient data structure called *(CSR)*.

2.5.1 Background

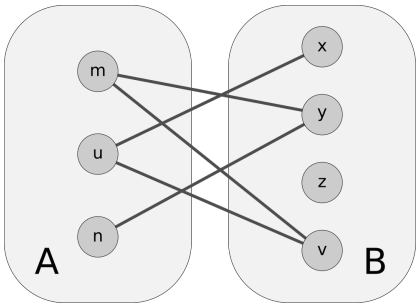
2.5.1.1 Bipartite graphs

A *bipartite graph* is a special type of a graph whose set of vertices can be divided into two subsets, such that none of its edges connects vertices from the same subset. In other words,

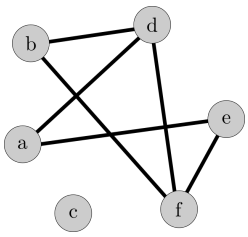
$$G = (V, E)$$

$$V = A \cup B,\ A \cap B = \emptyset$$

$$\forall e \in E,\ e = u, v : (u \in A\ \&\ v \in B)\ or\ (v \in A\ \&\ u \in B)$$



This defining property makes bipartite graphs a perfect illustration for hashing. If we have sets of data entries and their possible hash values as vertices, we will be able to depict them as a graph. Each edge will then connect a vertex representing a data entry to the one standing for its hash value. The fact that the two vertices will be naturally located in two different subsets makes the result a bipartite graph.



(a) Graph

	a	b	c	d	e	f
a	0	0	0	1	1	0
b	0	0	0	1	0	1
c	0	0	0	0	0	0
d	1	1	0	0	0	1
e	1	0	0	0	0	1
f	0	1	0	1	1	0

(b) Adjacency matrix

Figure 2.3: Representation of a graph by an adjacency matrix.

2.5.1.2 Compressed Sparse Row

In order to work with a hash table in a form of a bipartite graph in a real computer program, one needs a way to represent it in a more abstract form. In Computer Science, there exist several approaches to view a graph as a usable data structure, the simplest being adjacency lists, edge lists and adjacency matrices. This section describes the concept of a data structure called *Compressed Sparse Row* which can be viewed as a form of adjacency matrix, but more efficient from both space and time points of view.

Classic version of adjacency matrix is depicted on Fig. 2.3. The idea of this way of storing information about a graph is to have each row and a column of a matrix representing a single vertex. Each entry inside of it has a binary value indicating whether or not the graph contains an edge connecting these two vertices.

This traditional method is practical but has one significant problem. The issue is that the structure in a form of a 2D-array basically considers all possible edges and indicates which of them exist in reality. That demands excessive amounts of memory space to be used, especially for graphs with many vertices. A graph with a few or no edges at all would use exactly as much memory as a complete graph with the same number of vertices. A much more efficient approach would be to only store positions of entries having a value of 1. That is exactly what the idea of a Compressed Sparse Row is based on.

In CSR, all non-zero values in the matrix are written into a one-dimensional array. Their positions in the original matrix are stored in two separate arrays representing column and row indices respectively.

2.5.1.3 CSR representation of a graph

One feature of a graph's adjacency matrix that we can benefit from is the fact that all non-zero elements of it can never be anything but 1. This leaves us no need to store the actual values of the matrix as it is enough to remember the positions at which these 1's are located. Moreover, instead of storing their

row and column indices in two arrays, we can remember each row as a list of column indices where this row contains 1's. Finally, we can write these new rows subsequently into a one-dimensional array preserving only the order of values, and remember the positions from which each row begins there in a second array of offsets. This would allow to store the graph in the most memory-efficient way. Formally, the following three arrays are used (see also Fig. 2.4):

- *Vertices* - each only once.
- *Offset* array used divide the third array into parts relevant for each vertex
- *Edges* - represented each by the vertex adjacent to a one in the first array - stored in a specific order that is managed by *Offset*. Explained in details below.

In this structure, *Offset* maps each element of *Vertices* to a subarray of *Edges* that represents a list of its neighbors. Specifically, for each vertex v having an index n in the first array, its neighborhood is listed in the third one from index $Offset[n]$ to index $Offset[n+1]-1$.

The same ideas for both standard adjacency matrix and Compressed Sparse Row apply for directed graphs. The only difference is that existence of an edge going in one direction does not imply that the opposite one will be present as well. In other words, the classical matrix might become asymmetric, and CSR will represent only out-neighborhoods (see Fig. 2.5).

Procedure of construction of CSR representation of a given graph is presented on Alg. 4. In this version, the input graph is provided in a form of adjacency lists (where each vertex is mapped directly to a list of its neighbors), because this representation is the most trivial and is naturally very close to the idea of hashed values. PrefixSum operation used on the line 5 is described in both sequential and parallel forms in the next section. We are not concerned about creating a parallel version of Alg. 4 as construction of HashGraph itself discussed in Section 2.5.2.

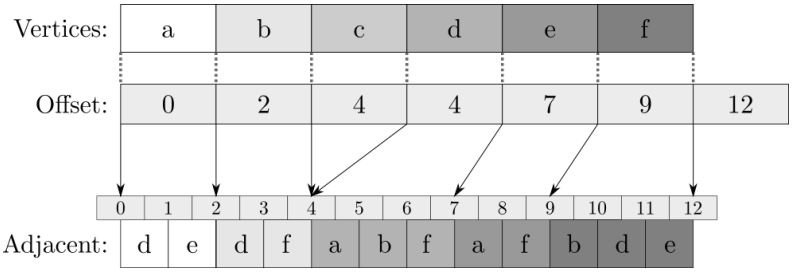


Figure 2.4: Compressed Sparse Row representation of the graph from Fig. 2.3

2.5.1.4 Prefix sum or Scan

Prefix sum (also known as *Scan*) is a sequence of numbers constructed from another sequence such that each $i-1$-th element of it is equal to sum of first i elements in the original sequence. Formally,

$$S = a_1, a_2, a_3, ...a_n$$

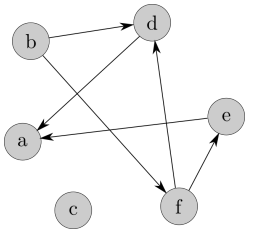
$$PrefixSum(S) = 0, a_1, \sum_{i=1}^{2} a_i, \sum_{i=1}^{3} a_i, ... \sum_{i=1}^{n} a_i$$

Of course, the same logic can be applied not only to sequences but to one-dimensional arrays as well.

Operation to find a Scan sequentially is very trivial (see Alg. 5). All we have to do is to set the first element of output array and then iterate over the other ones while setting each i-th element to the sum of the previous one and i-th element of the input.

Parallel version of it was described by M. Harris et al. [13]. It requires more effort and is presented in Alg. 6. It consists of two large steps:

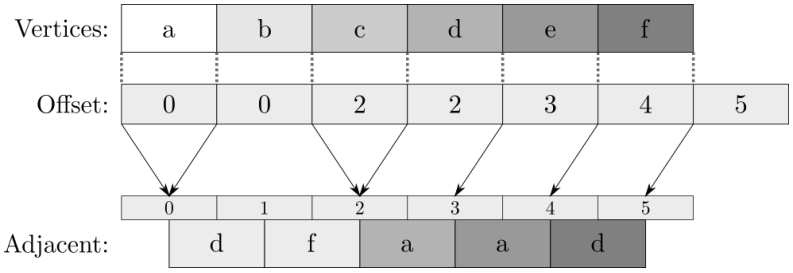
1. *Reduction step.* In the first step, values of input are grouped into pairs and sum of each pair is written instead of its second element. All sums are grouped and summed the same way until only one new sum produced. When this has happened, every second element will contain partal sum and the complete sum will be written into the last element. (Fig. 2.6)



(a) Directed graph

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	0	0	0	1	0	1
c	0	0	0	0	0	0
d	1	0	0	0	0	0
e	1	0	0	0	0	0
f	0	0	0	1	0	0

(b) Adjacency matrix



(c) Compressed Sparse Row

Figure 2.5: Representations of a directed graph.

Input: Graph G
Result: CSR representation of the graph
1 int CounterArray[]; // Size of each node's neighborhood
2 **for** *int i := 0 to* $|V(G)|$ **do**
3 | CounterArray[i] := nAdj(G[v_i]);
4 **end**
5 int Offset := PrefixSum(CounterArray);
6 **for** *int i := 0 to* $|V(G)|$ **do**
7 | CounterArray[i] := 0;
8 **end**
9 Vertex Edges[]; // Represents edges as a list of adjacent vertices
10 **for** *int i := 0 to* $|V(G)|$ **do**
11 | **foreach** *Vertex v : Adj(G[v_i])* **do**
12 | | Edges[Offset[i] + CounterArray[i]++] := v;
13 | **end**
14 **end**

Algorithm 4: Algorithm to transform a graph into a Compressed Sparse Row form.

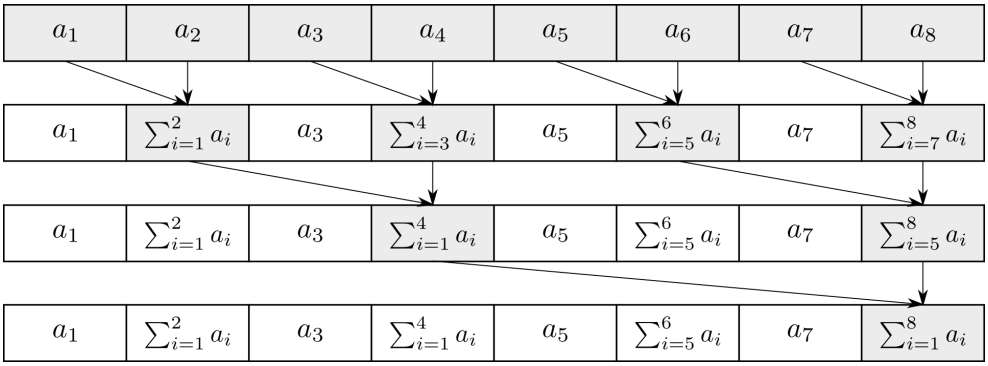


Figure 2.6: Reduction - first step of parallel scan finding.

2. *Down-sweep phase* - First, the last element is set to 0. After that, having $d = 0$ and incremented on each step, the following is repeated while possible:

   - 2^{d+1} equally distanced elements are divided into 2^d pairs of neighbors. Each n'th element starting from the last one is paired with the one of index $n - \frac{N}{2^{d+1}}$ (for the array of N elements);
   - in each pair, the left element is replaced with the right one;
   - old value of the left element is added to the value of the right one.

   After 2^d has reached the total size of the array, the operation is finished and the array is transformed into the prefix sum of its original self. The

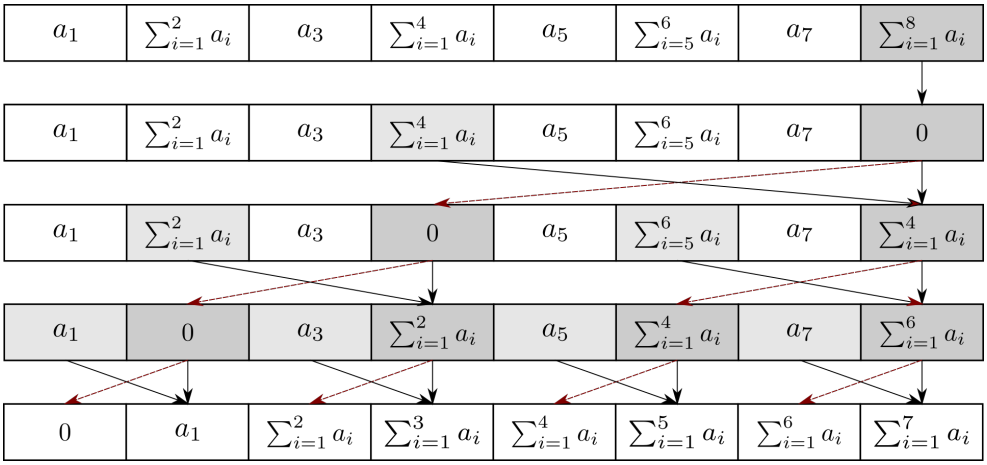


Figure 2.7: Down-sweep - second step of parallel scan finding.

zero inserted in the beginning of the phase is translated into the first element.

It must be noted that all terms and algorithms shown here and used in HashGraphs represent so-called *exclusive* scan. In *inclusive* version, each element of the output sequence is equal to sum of all elements including the current one. In *exclusive* scan, the operation is done only on preceeding elements. This requires supplying initial element as a parameter. For sum, this role is usually played by 0.

Moreover, the operation itself can be different from the sum (for example Prefix product). The contruction algorithm in this case does not differ by anything other than that operation.

Input: int Array[]
Result: Scan of input array

```
1 int Result[];
2 Result[0] := 0;
3 // Note that the loop starts from the second element (index 1)
4 for int i := 1 to len(Array) do
5 |   Result[i] := Result[i - 1] + Array[i - 1];
6 end
```

Algorithm 5: Sequential algorithm to find a scan of a sequence.

2.5.2 HashGraph algorithms

As mentioned above, HashGraph represents data entries and their hash values as a bipartite graph. It is done by viewing stored data entries as one subset of the vertex set and their hash values as another one. Each edge (u, v) between

```
   Input: int Array[]
   Result: Scan of input array
 1 // Reduction step
 2 for int k := 0 to M - 1 do
 3 |  int offset := 2^k;
 4 |  for int j := 1 to 2^(M-k-1) in parallel do
 5 |  |  Array[j * 2^(k+1) - 1] := Array[j * 2^(k+1) - 1]
 6 |  |     + Array[j * 2^(k+1) - 2^k - 1]
 7 |  end
 8 end
 9 // Down-sweep phase
10 for int k := M-1 to 0 do
11 |  int offset=2^k for int j := 1 to 2^(M-k-1) in parallel do
12 |  |  int dummy := Array[j * 2^(k+1) - 2^k - 1]
13 |  |  Array[j * 2^(k+1) - 2^k] := Array[j * 2^(k+1) - 1]
14 |  |  Array[j * 2^(k+1) - 1] := Array[j * 2^(k+1) - 1] + dummy
15 |  end
16 end
```

Algorithm 6: Parallel scan finding algorithm.

the two subsets then denotes that the entry u is hashed into the value v. In practice, the graph's adjacency matrix is stored as a Compressed Sparse Row.

The idea is presented in two versions that have same structure but different construction algorithms. Both are well optimized for running in parallel and unique in terms of their *load factor*. The load factor (referred to as C_V) is a relation of actual number of elements inserted into hash table to the maximum one. A low factor means that many positions for the data entries are left empty, which makes the structure less memory efficient. The higher factor, on the other hand, increases the probability of collision in most of hash table types, which increases the time of each operation. An advantage of HashGraph is that it can have a load factor of 1 (maximum possible) without suffering from related slow downs in construction.

Original versions of Alg. 7 provided in the author's paper had the line 18 written as

```
pos = AtomicAdd(CounterArray[H_A[i]], 1)
```

instead of what is shown here. This version cannot possibly be correct, because it would mean assigning value *pos* to 0 multiple times (because $CounterArray$ is assigned to all 0's in the previous step). It is assumed to either be a mistake, or a misunderstanding in the pseudo-code. The same is true for line 38 in Alg. 8. In both algorithms, argument N refers to the number of elements in the input array, and V stands for the number of vertices in the resultant graph.

Some of the vertices can be left empty meaning that $V = N/C_V$. As our goal is to construct a HashGraph with $C_V = 1$, we can assume that $V = N$.

2.5.2.1 HashGraph Version 1.0

Version 1.0 of HashGraph creation procedure (Alg. 7) is nothing but a parallel form of CSR construction algorithm described in Alg. 4 but with table keys and hash values as vertices.

Input: Entry A[], int V, int N, int C_v, Function Hash
Output: HashGraph with given values

```
1  int H_A[];
2  for int i := 0 to N-1 in parallel do
3  |  H_A[i] := Hash(A[i]) mod V;
4  end
5  int CounterArray[];
6  for int i := 0 to V-1 in parallel do
7  |  CounterArray[i] := 0;
8  end
9  for int i := 0 to N-1 in parallel do
10 |  AtomicAdd(CounterArray[H_A[i]], 1);
11 end
12 int OffsetArray[] := PrefixSum(CounterArray);
13 for int i := 0 to V-1 in parallel do
14 |  CounterArray[i] := 0;
15 end
16 int E[]; // Analogue of Edges array in Alg. 4
17 for int i := 0 to N-1 in parallel do
18 |  int pos := OffsetArray[H_A[i]] + AtomicAdd(CounterArray[H_A[i]],
   |    1);
19 |  E[pos] := A[i];
20 end
```

Algorithm 7: HashGraph construction algorithm Version 1.0.

2.5.2.2 HashGraph Version 2.0

Version 2.0 of the algorithm (Alg. 8) is designed to be more memory efficient because it exploits the cache better by improved locality.

It is achieved through dividing data into bins of equal size and reordering it accordingly. The goal is to decrease the distance between memory accesses at the building stage, and thus increase the number of cache hits by exploiting spatial locality of memory accesses (see Sec. 2.1 for more details). The procedure is divided into three phases:

1. *Bin counting.* Count the number of elements to be inserted into each bin.

2. *Data reorganization.* Reorder the input data in a more cache-efficient manner.

3. *HashGraph creation.* Put the data on their final places ready for probing.

One of the parameter of this algorithm is the number of bins. The authors suggest 32000 as optimal value for hundreds of millions data entries that they have tested it on. A modification that was done here compared to the original version (except for the fix mentioned above) is taking *ceiling* of the fraction as the value for $BinSize$ on the first line. Without it (if $BinSize = V/Bins$), the value of *bin* on line 4 might be greater than $Bins$, causing the line 5 to go out of bounds for $BinCounterArray$. Reorganization of input data requires allocating a separate array which increases memory comsumption. The usage of 8 for-loops instead of 5 in the Version 1.0 can also introduce additional slow-downs, however improved space locality is aimed to compensate it and yield faster performance.

2.5.2.3 Probing a HashGraph

From the graph point of view, the simplest way to probe a key is to take the vertex representing its hash value and look for the it within its neighbors like it is done Alg. 9. This procedure is referred to as *HashGraph–Probe–Standard*.

Another algorithm (Alg. 10) is called *HashGraph–Probe-New*. It is designed to query many keys at a time. The idea is to build another HashGraph from the queried keys using the same hash function and consider all possible hash values in parallel. For each value, the pair of the corresponding vertices is taken from both graphs. Intersections of neighborhoods of these vertices in all pairs will then yield a list of found keys.

HashGraph–Probe–Standard can also be used for multiple keys. To do so, a parallel for-loop would iterate over the keys. As we cannot start another for loop from inside a parallel one without knowing its bounds in advance, the loop that actually finds the keys in the $Edges$ array (Alg. 9 - line 4) cannot be parallel.

Input: Entry A[], int V, int N, int C_v, int Bins, Function Hash
Output: HashGraph with given values

```
1  int BinSize := ⌈V/Bins⌉;
2  int BinCounterArray[Bins];
3  for int i := 0 to Bins in parallel do
4  |  BinCounterArray[i] := 0;
5  end
6  for int i := 0 to N-1 in parallel do
7  |  int bin := Hash(A[i])modV/BinSize;
8  |  AtomicAdd(BinCounterArray[bin], 1);
9  end
10 int BinOffsetArray[] := PrefixSum(BinCounterArray);
11 int BinCounterArray[Bins];
12 for int i := 0 to Bins-1 in parallel do
13 |  BinCounterArray[i] := 0;
14 end
15 Entry A_reorg[N];
16 for int i := 0 to Bins-1 in parallel do
17 |  int bin := Hash(A[i])modV/BinSize;
18 |  int pos :=
   |   BinOffsetArray[bin]+AtomicAdd(BinCounterArray[bin], 1);
19 |  A_reorg[pos] := A[i];
20 end
21 for int i := 0 to V-1 in parallel do
22 |  CounterArray[i] := 0;
23 end
24 int H_A[N];
25 for int i := 0 to N-1 in parallel do
26 |  int pos := hash(A_reorg[i].val) mod V;
27 |  AtomicAdd(CounterArray[pos], 1);
28 |  H_A[i] := Hash(A_reorg) mod V;
29 end
30 int OffsetArray[] := PrefixSum(CounterArray);
31 for int i := 0 to V-1 in parallel do
32 |  CounterArray[i] = 0;
33 end
34 Entry E[N];
35 for int i := 0 to N-1 do
36 |  int pos = OffsetArray[H_A] + AtomicAdd(CounterArray[H_A], 1);
37 |  E[pos] = A_reorg[i];
38 end
```

Algorithm 8: HashGraph construction algorithm Version 2.0.

```
   Input: HashGraph HG, Key K
   Output: True if the key is found
 1 int Hash := HG.HashFun(key);
 2 int Start := HG.Offset[Hash];
 3 int End := HG.Offset[Hash + 1];
 4 for int i := Start to End do
 5 |   if HG.Edges[i] = K then
 6 |   |   return True;
 7 |   end
 8 end
 9 return False;
```

Algorithm 9: HashGraph–Probe–Standard algorithm.

```
   Input: HashGraph HG, Keys K[]
   Output: Keys from K that were found
 1 bool Found[len(K)] = Array of False; HG' := HashGraph(K);
 2 for int i := 0 to V-1 in parallel do
 3 |   for int v1 = HG.Offset[i] to HG.Offset[i + 1] do
 4 |   |   for int v2 = HG'.Offset[i] to HG'.Offset[i + 1] do
 5 |   |   |   if HG.Edges[v1] == HG'.Edges[v2] then
 6 |   |   |   |   Found[HG.Edges[v1]] = True;
 7 |   |   |   end
 8 |   |   end
 9 |   end
10 end
```

Algorithm 10: HashGraph–Probe–New algorithm.

Chapter 3

Realization

3.1 Implementation

Cuckoo hash table and HashGraph of versions 1.0 and 2.0 were realized using Template Numerical Library. The classes make use of TNL Arrays and each has its View class implementing most of the functionality. The View-classes for the hash tables store Views of the tables' member TNL Arrays as their class members.

Each class representing a table has two derived classes providing Set and Map interfaces (with their own views in case of HashGraph). The Table class in both cases accepts a template parameter called Item that can be either of type `Value<T>` (for Set) or `Pair<K, V>` (for Map). These two data types are used as simple wrappers for actual data entries. They both have a member called `key` that is used in the table's logic. This allows to unify that logic in one parent Table class. Type of the key is supplied as the second template parameter called `Key`. Considering that a Map is nothing but a version of

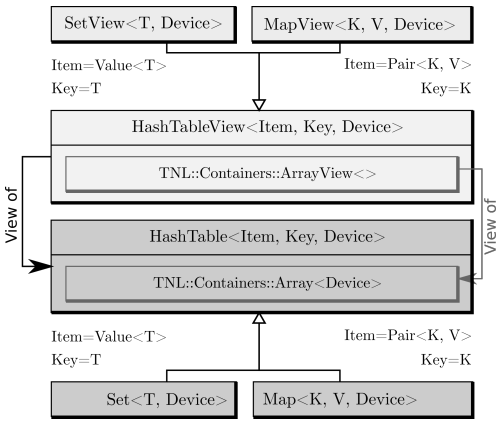


Figure 3.1: General class diagram for the implemented classes

Set storing key-value pairs, it is enough to demonstrate functionality and performance for the Set version. Thus, from now on this text will refer to Set derived class interchangeably with the parent Table class representing the table itslef. The child class for map is different only in its ability to return associated value for each key queried. The third template parameter `Device` specifies whether the data is allocated on CPU or GPU, and is passed further to all used TNL containers.

Each implemented class has a constructor designed to accept a TNL Array of values used as input data entries. The array's `Device` parameter should be the same as the table's. It also contains a `find` method that has the following prototype:

```
void find(
        typename Array<Key, Device>::ConstViewType
                keys,
        ArrayView<bool, Device> success)
```

This method queries each element from the `keys` array view and sets the element of `success` with the same index to a boolean value indicating whether or not the key was found in the table. A version to probe a single key is provided as well. All it does is constructing a single-element array with that key and return success flag as a boolean value.

Constructor of each table's view accepts a pointer to the table itself and extract views from its member arrays. The view construction is triggered by the constructor of the table, and a pointer to it is stored as the table's member. The view can be returned from the table using a getter method, and is being destroyed by the table's destructor. Map and Set views derive from the table view. Each implements an interface that allows to query a key from inside a CUDA kernel. The actual algorithm for a single-element probing is realized in the table view method with the following prototype:

```
template<bool writeResult>
__cuda_callable__ bool find(
        const K& key, Item* item) const
```

This method returns True if the key is found. If $writeResult = true$, the found item into the address supplied as the second argument (used in Map views). Set and Map views just provide interfaces for this method.

3.2 Class HashFunction

Trivially, all hashing algorithms are based on usage of hash functions. In this work, they are implemented as a template functor class with specializations for all data types used in testing. Each specialization uses its own approach (iterative multiplication and addition for integer arrays, `std::hash`

for `std::string`s etc.). However, the resultant value, even modulo the maximum required number, cannot be used as a hash value in itself. The reason is that Cuckoo hashing generates its hash functions randomly and has to regenerate them after each failed attempt to build the table. Thus, each hashing operation must be dependent on unique attributes of each hash function. The easiest way to realize that is to use the following formula:

```
(a * spec(value) + b) % PRIME % max
```

, where `spec` is a method specialized for each hashed data type, `a` and `b` are unique attributes of the `HashFunction` instance, `PRIME` is defined as 334214459 and `max` is maximum acceptable hash value (relevant to the context of usage). Each instance can be initialized by specific values of `a` and `b`.

3.3 Implementation of Cuckoo hashing

The approach is realized in classes `CuckooHashMap` and `CuckooHashSet` derived from `CuckooHashTable` and their common corresponding view class called `CuckooHashTableView` with derived Map and Set views.

By definition of Cuckoo hash table, it contains two one-dimensional arrays: an array of stored items (the table itself) and the array of hash-functions. Both are represented by `TNL::Containers::Array` in this implementation. Apart from the input array, the constructor of the table accepts *table size*, *number of hash functions* and maximum *number of iterations*. By default, these arguments are set to the values suggested by the authors and described in Sec. 2.4.1. However, the advised choice of 4 hash functions has proven not enough for larger data as it forces the building algorithm to go into an infinite loop of failed insertions. In our testing setup, we used the value of 6, and it was practically found that the number actually growth as the size of input data increases.

3.3.1 Storing and indexing

As mentioned in Sec. 2.4.4, it is more time-efficient to store each data key with an integer index representing the hash function used to hash the key. It might be useful later to find a new function to re-insert the key after eviction. The easiest way to do that was to introduce a new structure called `Entry` that would store both values together, and then use `Array<Entry, Device>` as the main data storage.

The process of building the table supposes constant eviction and re-insertion of the keys that were previously stored there. That is the natural way of Cuckoo hash table to handle the hash collisions. Having numerous threads performing those operation simultaneously will inevitably lead to race conditions. The only tool offered by CUDA to fight the race condition is the usage of

atomic operations (in our case, `atomicExch` can be a solution). The problem with provided operations is that they only work for a limited number of standard primitive types from the C language. Thus, applying atomic operations to the instances of `Entry` is impossible. The solution used here was storing table contents separately from their indices. The class `CuckooHashTable` uses two TNL Arrays to represent the data. The class attributes are listed as follows:

- `Array<Entry, Device> m_content` keeps the data entries in the same order as they were supplied;
- `Array<int, Device> m_table` references then by indices in the order specified by the algorithm;
- `Array<HashFunction<Key>, Device> m_hashFunctions` contains a number of hash functions given as constructor parameter.

All evictions and insertions are then done on `m_table` using atomic operations, and probing of an element in position `i` is done by accessing `m_content[m_table[i]]`.

3.3.2 Initialization of hash functions

Initialization of hash functions is done in parallel as well. Each kernel started by `ParallelFor` is supplied with integer index `i` of its hash function and is supposed to randomly initialize it. Limitiations of CUDA framework do not allow calling a host function (not defined with a special macro) from a device kernel. That prevents using the `rand` function from standard C library. Interface of `ParallelFor` does not allow to generate and pass a separate random integer to each instance of the kernel either. So instead, on each re-initialization, a new integer `r` was generated randomly for all hash functions. That value was supplied to all parallel kernels together with the index `i` of a hash function they had to initialize. Each kernel then would pass the following values to a constructor of its corresponding hash function:

$$a = r * i + 10538$$

$$b = r * i + 152324$$

. The constant integer numbers chosen are irrelevant and could really be set to any random value. This way each hash function was different from the others while each re-generation produced a new set of functions written into the later used array.

3.3.3 Insertion and probing

Insertion of each data key from the input array was done by a procedure called `insert_impl` defined as `__cuda_callable__`. The procedure is called from `ParallelFor` and is supplied a view of a common 1-element `result` array indicating success. `ParallelFor` itself is enclosed in a do-while loop that checks the success after each attempt to insert all the keys. If insertion of at least one key is failed, the only value inside `result` is set to false and the loop is started again from initializing the hash functions.

As for all classes created here, probing is done by the `find` method. In case of Cuckoo hashing, it is done by executing a `ParallelFor2D`. The first dimension of its execution specifies the key in queried array to look for, and the second one determines which hash function to use to find a candidate position of the key. When a particular kernel is lucky to be provided an index of a key with the index of the function that the key was hashed with, it sets the element in the `success` array with corresponding index to true. In case of a Map, it also writes a value mapped to by the found key into another output array.

In case of `CuckooHashTableView` and its derived Map and Set classes, they iterate over the hash functions sequentially until the key is found. That is the only way to make this method usable from inside a CUDA kernel as a ParallelFor cannot be executed from there.

3.4 Implementation of HashGraph

As HashGraph is nothing but a version of Compressed Sparse Row (Sec. 2.5.1.2), their structures look very much alike.

3.4.1 Structure

Unlike in Cuckoo hash table, here we do need multiple hash functions, nor do we need to deal with race conditions. This allows us to store the data as they are in a single array without using `Entry` wrapper or a separate array of indices. Thus, apart from the view pointer, both versions of `HashGraph` (1.0 and 2.0 - classes `HashGraphV1` and `HashGraphV2`) have only these class members:

- `Array<Item, Device> m_content` - inserted data of type Item. Corresponds to *Edges* array in Alg. 4;
- `Vector<int, Device> m_offset` - same use as in CSR;
- `HashFunction<Key> m_hash` - one hash function is enough.

However, we also have a variant of the version 1.0 implemented using `TNL::Algorithms::Segments::CSR` whose functionality substitutes all the

actions needed to process the Offset array (while its instance replaces the array itself in the HashGraph's class members). Each segment corresponds to a separate hash value, and its indices iterate over the keys hashed into it. *CounterArray* from the original algorithm is then used to initalize the segment sizes. This variant is realized in `HashGraphS` class. From now on, we will refer to it as *Segment* version, while the other two will be called *Array* versions. All classes have their corresponding views (`HashGraphSView`, `HashGraphV1View`, `HashGraphV2View`) and derived Map and Set implementations with Map and Set views. The last two are derived from the View class and used for calling its method for probing a single key.

The process of construction of the Array versions in our implementation is very close to the one described in Sec. 2.5.2. It is only worth mentioning that the function $PrefixSum$ used in Alg. 7 and 8 works by copying the $CounterArray$ into $OffsetArray$ and then passing it to TNL's `Scan` class's `exec` method with lambda returning sum of its arguments (See Sec. 2.3.3).

The Segment version is constructed in a similar way to the version 1.0 except that the first part of the building process is done by the HashGraph class itself instead of the View class. This had to be done so because the $CounterArray$ dictates the segment sizes for the CSR instance, and the CSRView cannot accept it as the views do not allocate any data in TNL. Thus, only the actual data placement is done by the view constructor.

3.4.2 Probing

Algorithm *HashGraph–Probe–New* supposes building a second HashGraph of queried keys and running parallel kernels in 3 dimensions. The first one would go through all hash values, and it is possible to do using `ParallelFor` because the number of hash values is known in advance. However, the neighborhoods of all hash values have unequal sizes, which makes using `ParallelFor3D` impossible (because it requires the sizes of all three dimensions to be equal and known at launch time).

Somewhat optimal solution is to iterate in parallel over possible hash values, and started normal for-loops inside the kernel for neighborhoods of each of them in both HashGraphs. This way has experimentally proven to be slower than *HashGraph–Probe–Standard* which was our primary choice.

To realize it, we run a `ParallelFor` through all queried keys. Each kernel then calls the single-key finding method that iterates in a sequential for-loop over all keys having the same hash value as the one it is responsible for. In practice, it means going from $OffsetArray[hv_i]$ to $OffsetArray[hv_i + 1]$, where $hv_i = hash(k_i)$ for each parallelly probed key k_i (see Alg. 9).

Probing is done identically for both Array versions 1.0 and 2.0.

The Segment version uses a variant of the Probe–Standard by iterating over a segment corresponding to the key's hash value. The upper bound

for the inner for-loop is learned from the `getSegmentSize()` method, and `getGlobalIndex` is used to index `m_content` looking for the key.

Chapter 4

Testing

4.1 Recapitulation of the tested classes

The classes tests on which are described further implement two general approaches: Cuckoo hashing and HashGraph. Map classes derived from them store key-value pairs and can be used for probing values by keys, while Sets do the same with just singular keys. View classes are used to be passed to CUDA kernels without performing deep copy of data provided that the data itself are allocated on the proper device. There exists a class for each approach serving as a parent for Map and Set and a View class with Map and Set views derived from it. Below is the hierarchy of the implemented classes:

- `std::unordered_set` - standard implementation from the STL library. Represents a hash table-based set that can only be run on a CPU and was only tested for comparison with our implementation.
- `CuckooHashTable` - based on Cuckoo hashing (Sec. 2.4). Building is done by repeated eviction and insertions of keys with several hash functions until a free position is found (Alg. 2). Probing is done by hashing the queried key with all hash functions taking the results as potential positions (Alg. 3). Implementation is described in Sec. 3.3.
  - `CuckooHashMap`
  - `CuckooHashSet`
- `CuckooHashTableView`
  - `CuckooHashMapView`
  - `CuckooHashSetView`
- *HashGraphs* - represents hash-entry mapping in a form of bipartite graph stored as a Compressed sparse row (Sec. 2.5.1.2). The idea is described in Sec. 2.5 and implementation is in Sec. 3.4. Has two versions

of the probing algorithms: HashGraph–Probe–Standard (Alg. 9) and HashGraph–Probe–New (Alg. 10). Building is also introduced in two versions described below. Presented here in three variants:

- `HashGraphV1` - a version of HashGraph with a simpler building algorithm (Alg. 7) described in Sec. 2.5.2.1.
  - `HashGraphV1Map`
  - `HashGraphV1Set`
- `HashGraphV1View`
  - `HashGraphV1MapView`
  - `HashGraphV1SetView`
- `HashGraphV2` - a version of HashGraph whose building algorithm (Alg. 8) is designed to be more cache-efficient due to a better spatial locality. Described in Sec. 2.5.2.2. This class is used for testing HashGraph–Probe–New algorithm along with HashGraph–Probe–Standard used in the other two classes.
  - `HashGraphV2Map`
  - `HashGraphV2Set`
- `HashGraphV2View`
  - `HashGraphV2MapView`
  - `HashGraphV2SetView`
- `HashGraphS` - based on HashGraphV1 but uses TNL's `CSR` class (Sec. 2.3.3) instead of managing some of the inner array by itself.
  - `HashGraphSMap`
  - `HashGraphSSet`
- `HashGraphSView`
  - `HashGraphSMapView`
  - `HashGraphSSetView`

4.2 Implementation of tests

The nature of TNL allows to compile and run our implementation in both GPU and CPU and compare the results obtained from both options.

The tests consisted of unit tests implemented using *GoogleTest* library and performance tests on large amounts of data. In order to test probing mechanism (and thus correctness of insertion as well), two types of tests were used in both test sets:

- *Correct query* - probing previously inserted keys (expecting success);

- *Wrong query* - probing keys that were not previouly inserted (expecting failure).

While testing Set classes meant nothing but checking if the inserted keys were found (and not vice-versa), tests for Maps also checked if found keys were associated with correct values.

4.2.1 Unit tests

GoogleTest is a C++ library used for unit testing (testing of separate functions). It provides a simple interface for writing test suites and running them in parallel and is used for testing all classes in Template Numerical Library.

In our implementation, we used tests on correct and wrong queries on both Sets and Maps implemented by all classes. Testing data consisted of consecutive numbers, and the test were done with 4 primitive C data types - `int`, `long`, `float` and `double` (in all 16 combinations as key-value pairs in case of Maps). The tests included testing keys separately and by whole arrays - both using a specific array-probing method and a view class from inside a ParallelFor body.

4.2.2 Performance tests

Performance is tested on two sets of data which is similar in format but while one is generated randomly (using a Python script), the second one is taken from a real-life case. We will describe the first one because we can control its amount, and thus it is more representative in terms of size vs performance dependency trends. The results obtained on the second set will be provided as well. In our case, it allows to observe the performance on inputs of small sizes and non-linear size differences. There is no other practical difference between them as they have the same format.

The data consist of tetrahedral meshes cells represented as integer numbers grouped by 4. In the code these groups are stored as instances of 4-element static arrays (`TNL::Containers::StaticArray<4, int>`). The data are read from the text files whose sizes differ by whole millions of entries (to form a good illustration). In tests, we are interested in the way the execution time depends on the number of input entries. We do not test Maps alongside Sets because all difference between the classes lies in functionality, while building and querying logic is absolutely identical. Randomly generated tests contain 8 sets of input data: from $5 \cdot 10^6$ to $3 \cdot 10^7$ entries with $5 \cdot 10^5$ between the sets.

To register the time spent on each operation, we used functions of the standard library of the C language, namely the header `<ctime>`. It allows to register processor time at any moment. To measure performance time, it is possible to subtract registered times and represent them in seconds. To wrap this functionality into a convenient interface, we introduced a class `Measurable` that stores two registered times and can return their difference.

Registering is triggered by calling protected methods `begin_operation()` and `end_operation()`, and `double` time difference is returned by a public method `elapsed()`. The classes we being tested are then derived from `Measurable`. The registering methods are called before and after relevant operations (building and probing), and the execution time is used by the testing function for further analysis.

It was important to compare our implementation to analogous classes in the standard library. In case of STL, the functionality of a hash table is implemented by the `std::unordered_set` class. It has a different interface from our implementation, mainly not being able to query multiple elements at the same time. To test it alongside our classes with the same test suites, we use a special wrapper class `StdSetWrapper` that inherits from `Measurable` and uses an internal `std::unordered_set` attribute to implement `find` method in the same fashion. The only difference that cannot be overcome is the device storing the content. Unlike our classes, `std::unordered_set` can only accept multiple values in a form of `std::vector` iterators. This forces us to store the data twice - once into an `std::vector` for the `StdSetWrapper` and another one into `TNL::Containers::Array` for our hash tables. Storing a copy of data on the CPU will also allow us to access it in the testing functions without using the expensive `getElement` methods of the TNL arrays.

The data are read from a text file by a `get_data` function that accepts the name of the file and an `std::vector` as an output parameter to fill with data. The same data are written into an `Array<T, Device>` returned from the function.

The function `test_class` accepts the class as a template parameter and runs three tests on that class:

- Building;
- Correct query;
- Wrong query.

Each test is represented by a separate template function and it writes the double value representing execution time (returned from `elapsed()`) into a global 2-dimensional array. Each of the functions has a specialization for `StdSetWrapper` that uses the `std::vector` instead of a TNL `Array`. The function `test_building` returns the instance of the table built. Intuitively, it cannot test correctness of the building process by itself as the contents are not seen before the probing stage. The functions performing probing testing pass the values to be queried to the `find_invalid` function that runs the query and checks the results in a `ParallelFor` loop. While `test_correct_query` probes the same input array that was used for building, `test_wrong_query` "ruins" the input values by calling the `ruin` procedure that replaces the first value of the cell with its negative version. This way we make sure that the

key will be missing from the table because the integers in the cells are always positive.

On the top level, each of the classes is passed as a template parameter to the `test_class` function that runs `test_building` in a loop for each input file and stores the resultant tables in an `std::vector`. The tables are then passed to the probing test suites. In case a test fails functionally along the whole process, a failing assertion stops the execution. Otherwise, the two-dimensional array of performance times is being filled and then printed to the console.

For HashGraphV2, we made a comparison of HashGraph–Probe–Standard and HashGraph–Probe–New. A graph was constructed from $4 \cdot 10^7$ entries, and then both correct and wrong queries were performed with different numbers of keys.

We run the tests on a single CPU thread and in multi-threaded GPU version. We compare the time needed to complete the tests and calculate the speedup resulting from the transition to GPU.

4.3 Testing setup

The test is performed on a machine with the following characteristics:

CPU:	Intel® Core™ i9-9900KF CPU @ 3.60 GHz (8 cores, 16384 KB cache)
RAM:	31Gi
GPU 0:	GeForce RTX 2070 SUPER, 7979 MiB
GPU 1:	GeForce RTX 2070 SUPER, 7982 MiB

In terms of software configuration, the system used was *Arch Linux 5.9.13-arch1-1*. We used *g++ 9.3.0* as host compiler and *NVIDIA® Cuda V11.1.105*. Unit test were done using *Google Test V1.10.0*

Below are the results obtained in the tests. The first section of this chapter shows and analyzes the results from testing on randomly generated values, the second one shows that findings are consistent with real data.

After comparing the results acheived after running the tests on CPU and GPU, we calculate the speedups between them as the relation of times spent on both devices:

$$\Delta T = \frac{T_{CPU}}{T_{GPU}}$$

4.4 Results of testing with randomly generated data

Each test was run on both CPU and GPU. In the results, we consider the time spent on tests execution versus the number of input values.

4.4.1 Results from running tests on CPU

4.4.1.1 Building test

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
$5 \cdot 10^6$	1.84	2.23	**0.83**	1.90	**0.97**	2.03	**0.91**	2.22	**0.83**
$1 \cdot 10^7$	3.97	4.80	**0.83**	4.26	**0.93**	4.51	**0.88**	4.49	**0.88**
$1.5 \cdot 10^7$	6.79	7.38	**0.92**	6.65	**1.02**	7.01	**0.97**	6.78	**1.00**
$2 \cdot 10^7$	10.10	10.09	**1.00**	9.05	**1.12**	9.57	**1.06**	9.05	**1.12**
$2.5 \cdot 10^7$	13.89	12.71	**1.09**	11.47	**1.21**	12.08	**1.15**	11.34	**1.22**
$3 \cdot 10^7$	18.11	15.35	**1.18**	13.88	**1.30**	14.68	**1.23**	13.63	**1.33**
$3.5 \cdot 10^7$	22.79	18.08	**1.26**	16.31	**1.40**	17.16	**1.33**	15.92	**1.43**
$4 \cdot 10^7$	27.91	20.81	**1.34**	18.31	**1.52**	19.65	**1.42**	18.21	**1.53**

Table 4.1: Time in seconds spent on building hash tables of our implementation (CuckooHashSet, HashGraphSSet, HashGraphV1Set and HashGraphV2Set - Sec. 4.1) with N elements of randomly generated data on a single CPU thread with speedup compared to std::unordered_set.

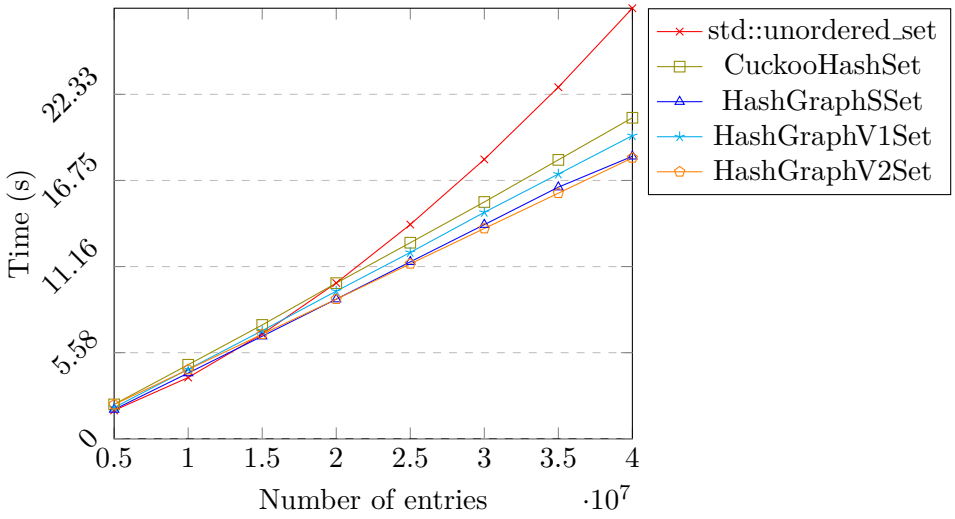


Figure 4.1: Graph of performance of building hash tables on CPU with randomly generated data - see Table 4.1.

As depicted on the graph, the amount of time spent by all of our classes on building responds linearly to the size of input data. Compared to STL version showing an exponential trend, it makes HashGraphs advantageous on data larger than $1.5 \cdot 10^7$ elements even on the CPU (and Cuckoo hashing for more than $2 \cdot 10^7$). The values are very close to each other with HashGraphV2 slightly overtaking its competitors, which might be explained by its better spatial locality. Despite algorithmic similarity, the Segment version of HashGraph shows better results than HashGraphV1.

4.4.1.2 Correct query test

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
$5 \cdot 10^6$	1.81	7.16	**0.25**	2.49	**0.73**	2.30	**0.79**	1.56	**1.16**
$1 \cdot 10^7$	3.94	15.13	**0.26**	5.21	**0.76**	4.76	**0.83**	3.32	**1.19**
$1.5 \cdot 10^7$	6.75	23.19	**0.29**	7.87	**0.86**	7.21	**0.94**	5.05	**1.34**
$2 \cdot 10^7$	10.03	31.25	**0.32**	10.54	**0.95**	9.67	**1.04**	6.79	**1.48**
$2.5 \cdot 10^7$	13.78	39.38	**0.35**	13.25	**1.04**	12.17	**1.13**	8.56	**1.61**
$3 \cdot 10^7$	17.91	47.48	**0.38**	15.94	**1.12**	14.65	**1.22**	10.34	**1.73**
$3.5 \cdot 10^7$	22.52	55.60	**0.41**	18.67	**1.21**	17.10	**1.32**	12.07	**1.87**
$4 \cdot 10^7$	27.56	63.76	**0.43**	21.34	**1.29**	19.57	**1.41**	13.82	**1.99**

Table 4.2: Time in seconds spent on probing N previously inserted elements of randomly generated data over hash tables on a single CPU thread with speedup compared to std::unordered_set. Each table is built from the same N elements and implemented by one of our classes (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1).

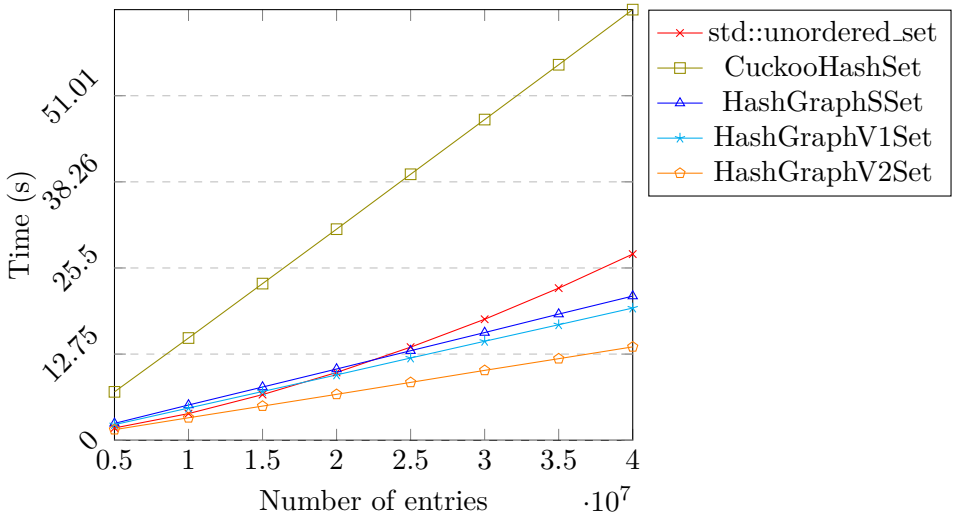


Figure 4.2: Graph of performance of probing correct keys over hash tables on CPU with randomly generated data - see Table 4.2.

When it comes to probing, both Array versions again show much better results than std::unordered_set with Segment variant being slightly slower than HashGraphV1 (although still linear) and significantly worse than HashGraphV2. Cuckoo hash table, on the other hand, cannot compete with neither of those as its linear growth is about three times faster. However, considering exponential trend of the STL implementation, it is expected to start losing the battle on much greater amounts of data. Unfortunately, we cannot check this hypothesis as the memory capacity of our equipment is too limited.

4.4.1.3 Wrong query test

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
$5 \cdot 10^6$	1.79	6.87	**0.26**	2.07	**0.86**	1.84	**0.97**	1.10	**1.63**
$1 \cdot 10^7$	4.11	14.59	**0.28**	4.39	**0.94**	3.85	**1.07**	2.41	**1.71**
$1.5 \cdot 10^7$	7.79	22.35	**0.35**	6.61	**1.18**	5.83	**1.34**	3.66	**2.13**
$2 \cdot 10^7$	12.40	30.13	**0.41**	8.88	**1.40**	7.84	**1.58**	4.96	**2.50**
$2.5 \cdot 10^7$	17.83	37.97	**0.47**	11.17	**1.60**	9.88	**1.80**	6.26	**2.85**
$3 \cdot 10^7$	24.07	45.78	**0.53**	13.45	**1.79**	11.92	**2.02**	7.59	**3.17**
$3.5 \cdot 10^7$	31.16	53.62	**0.58**	15.79	**1.97**	13.93	**2.24**	8.86	**3.52**
$4 \cdot 10^7$	39.07	61.46	**0.64**	18.04	**2.17**	15.95	**2.45**	10.17	**3.84**

Table 4.3: Time in seconds spent on probing N missing keys over hash tables on a single CPU thread with speedup compared to std::unordered_set. Each table is built from different N elements of randomly generated data and implemented by one of our classes (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1).

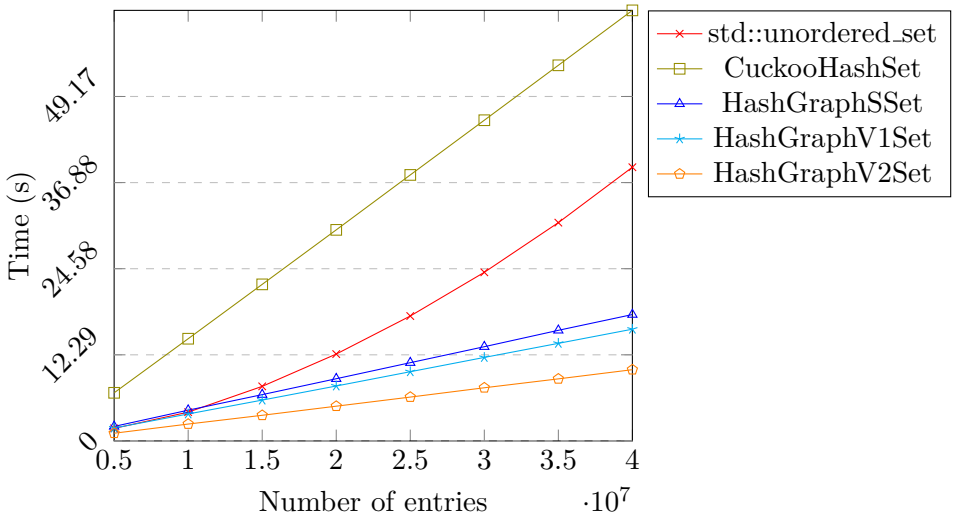


Figure 4.3: Graph of performance of probing wrong keys over hash tables on CPU with randomly generated data - see Table 4.3.

Trends for probing wrong keys look very close to the previous test except that HashGraphs show slightly better results here. Interestingly, the opposite is true for the STL and Cuckoo versions.

4.4.1.4 Comparing HashGraph probing algorithms

Both probing algorithms react linearly to the increase in the number of queried keys. HashGraph–Probe–Standard looks faster in all cases. Moreover, its growth rate is steeper than the one of HashGraph–Probe–New which leaves

N	STANDARD	NEW
$1 \cdot 10^7$	3.28	13.79
$2 \cdot 10^7$	6.67	20.47
$3 \cdot 10^7$	10.16	27.07
$4 \cdot 10^7$	13.74	33.69

(a) Testing with *correct* queries - probing N different previously inserted keys.

N	STANDARD	NEW
$1 \cdot 10^7$	2.53	12.63
$2 \cdot 10^7$	5.06	18.30
$3 \cdot 10^7$	7.60	23.94
$4 \cdot 10^7$	10.13	29.55

(b) Testing with *wrong* queries - probing N missing keys.

Table 4.4: Comparing times spent by two HashGraph (Sec. 2.5) probing algorithms (Probe–Standard - Alg. 7 and Probe–New - Alg. 8) on a single CPU thread. The table is built from $4 \cdot 10^7$ elements of randomly generated data and implemented as HashGraphV2 class (see Sec. 4.1).

no hope for the latter one to overtake it for larger data. This contradicts our expectations as the New algorithm is supposed to be more time-efficient.

4.4.2 Results from running tests on GPU

Unsurprisingly, running Cuckoo hashing and HashGraphs in parallel on GPU makes them much faster than on CPU, and std::unordered_set cannot compete with them anymore.

4.4.2.1 Building test

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
$5 \cdot 10^6$	1.84	0.62	**2.97**	0.15	**12.27**	0.16	**11.50**	0.22	**8.36**
$1 \cdot 10^7$	3.97	1.25	**3.18**	0.29	**13.69**	0.31	**12.81**	0.44	**9.02**
$1.5 \cdot 10^7$	6.79	1.88	**3.61**	0.43	**15.79**	0.46	**14.76**	0.65	**10.45**
$2 \cdot 10^7$	10.10	2.48	**4.07**	0.58	**17.41**	0.61	**16.56**	0.92	**10.98**
$2.5 \cdot 10^7$	13.89	3.06	**4.54**	0.73	**19.03**	0.77	**18.04**	1.15	**12.08**
$3 \cdot 10^7$	18.11	3.67	**4.93**	0.88	**20.58**	0.92	**19.68**	1.38	**13.12**
$3.5 \cdot 10^7$	22.79	4.24	**5.38**	1.02	**22.34**	1.07	**21.30**	1.62	**14.07**
$4 \cdot 10^7$	27.91	4.86	**5.74**	1.17	**23.85**	1.22	**22.88**	1.86	**15.01**

Table 4.5: Time in seconds spent on building hash tables of our implementation (CuckooHashSet, HashGraphSSet, HashGraphV1Set and HashGraphV2Set - Sec. 4.1) with N elements of randomly generated data in parallel on GPU with speedup compared to std::unordered_set running on CPU.

At the building stage, all our classes behave in a linear fashion. The Segment HashGraph and the HashGraphV1 show exactly the same performance here. Contrary to our expectations, HashGraphV2 takes more time than them despite its better space locality. It can be explained by more complicated building algorithm, but generally we can conclude that the goal of

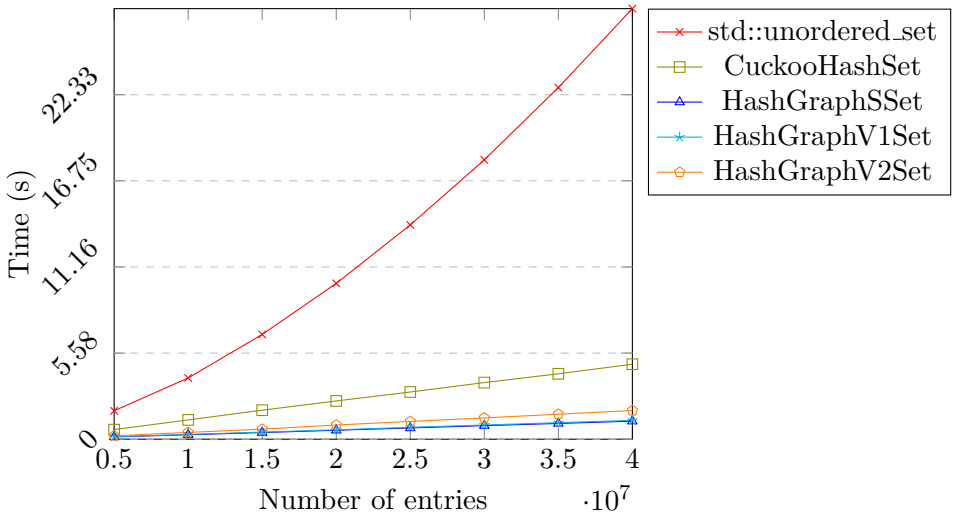


Figure 4.4: Graph of performance of building hash tables on GPU with randomly generated data - see Table 4.5.

its improvements is not reached. In any case, both versions still show much better results than Cuckoo hashing that took almost 5 seconds to build on the largest input set of data compared to the HashGraphs spending between 1 and 2 seconds. Compared to the Standard implementation, we get a speedup of 4.45 to 12.3 times.

4.4.2.2 Correct query test

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
$5 \cdot 10^6$	1.81	0.77	**2.35**	0.32	**5.66**	0.25	**7.24**	0.27	**6.70**
$1 \cdot 10^7$	3.94	1.53	**2.58**	0.64	**6.16**	0.50	**7.88**	0.52	**7.58**
$1.5 \cdot 10^7$	6.75	2.29	**2.95**	0.96	**7.03**	0.74	**9.12**	0.79	**8.54**
$2 \cdot 10^7$	10.03	3.07	**3.27**	1.37	**7.32**	1.05	**9.55**	1.12	**8.96**
$2.5 \cdot 10^7$	13.78	3.85	**3.58**	1.72	**8.01**	1.31	**10.52**	1.40	**9.84**
$3 \cdot 10^7$	17.91	4.63	**3.87**	2.06	**8.69**	1.58	**11.34**	1.68	**10.66**
$3.5 \cdot 10^7$	22.52	5.41	**4.16**	2.40	**9.38**	1.84	**12.24**	1.96	**11.49**
$4 \cdot 10^7$	27.56	6.20	**4.45**	2.75	**10.02**	2.11	**13.06**	2.24	**12.30**

Table 4.6: Time in seconds spent on probing N previously inserted elements of randomly generated data over hash tables in parallel on GPU with speedup compared to std::unordered_set running on CPU. Each table is built from the same N elements and implemented by one of our classes (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1).

Again, HashGraphs were the winners of the comparison being very close to each other, and Cuckoo hash table is more than twice slower than them.

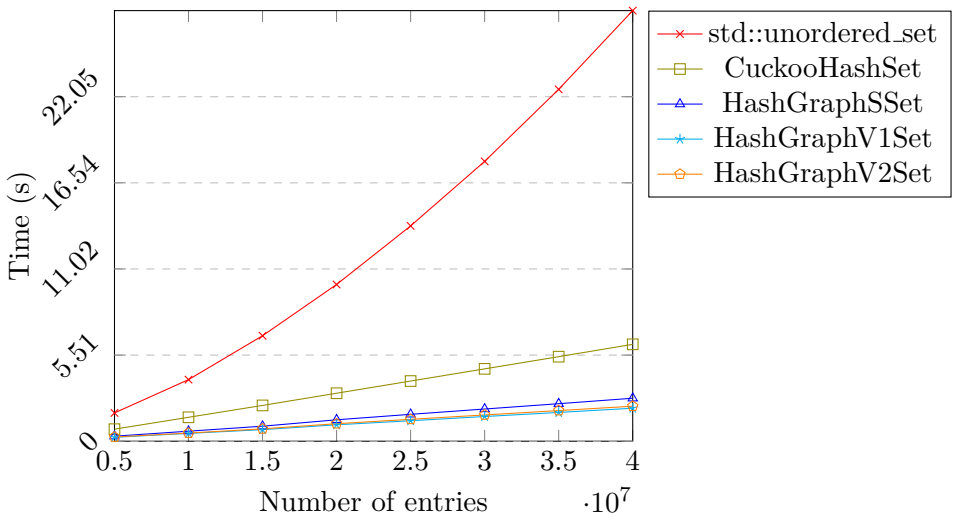


Figure 4.5: Graph of performance of probing correct keys over hash tables on GPU with randomly generated data - see Table 4.6.

The version 2.0 of HashGraph still shows worse performance than version 1.0 but this time better than the Segment version. Still, all four of them are much faster than the standard class running on CPU, and the gap between them grows with the size of input data.

4.4.2.3 Wrong query test

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
$5 \cdot 10^6$	1.79	0.59	**3.03**	0.25	**7.16**	0.15	**11.93**	0.15	**11.93**
$1 \cdot 10^7$	4.11	1.17	**3.51**	0.50	**8.22**	0.30	**13.70**	0.30	**13.70**
$1.5 \cdot 10^7$	7.79	1.76	**4.43**	0.75	**10.39**	0.45	**17.31**	0.45	**17.31**
$2 \cdot 10^7$	12.40	2.35	**5.28**	1.07	**11.59**	0.63	**19.68**	0.63	**19.68**
$2.5 \cdot 10^7$	17.83	2.95	**6.04**	1.34	**13.31**	0.79	**22.57**	0.79	**22.57**
$3 \cdot 10^7$	24.07	3.54	**6.80**	1.61	**14.95**	0.95	**25.34**	0.95	**25.34**
$3.5 \cdot 10^7$	31.16	4.14	**7.53**	1.89	**16.49**	1.10	**28.33**	1.10	**28.33**
$4 \cdot 10^7$	39.07	4.74	**8.24**	2.16	**18.09**	1.26	**31.01**	1.26	**31.01**

Table 4.7: Time in seconds spent on probing N missing keys over hash tables of s in parallel on GPU with speedup compared to std::unordered_set running on CPU. Each table was built from different N elements of randomly generated data and implemented by one of our classes (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1).

General comparison between the trends here is quite similar to the Correct query version except that all values are smaller (just like it was in CPU testing).

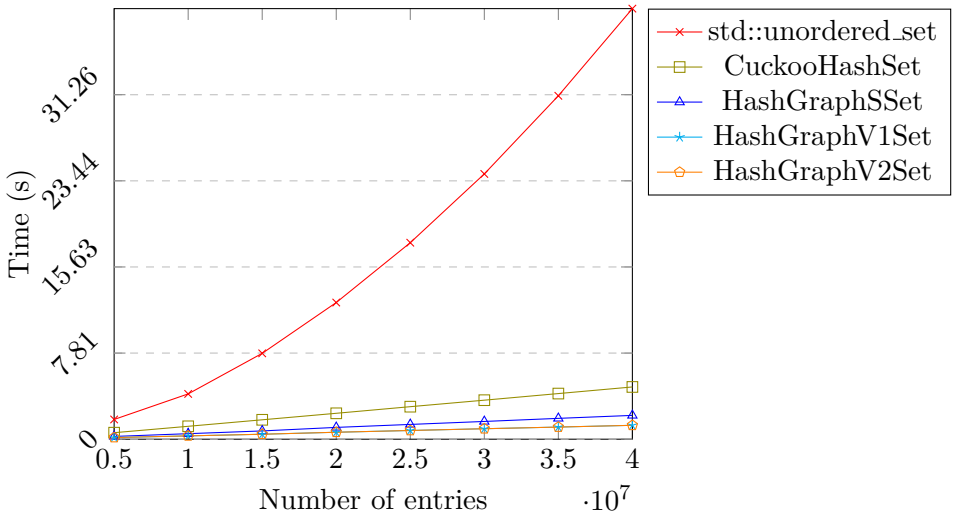


Figure 4.6: Graph of performance of probing wrong keys over hash tables on GPU with randomly generated data - see Table 4.7.

4.4.2.4 Comparing HashGraph probing algorithms

N	STANDARD	NEW
$1 \cdot 10^7$	0.52	3.37
$2 \cdot 10^7$	1.11	4.98
$3 \cdot 10^7$	1.68	6.40
$4 \cdot 10^7$	2.24	7.66

(a) Testing with *correct* queries - probing N different previously inserted keys.

N	STANDARD	NEW
$1 \cdot 10^7$	0.30	2.07
$2 \cdot 10^7$	0.62	2.90
$3 \cdot 10^7$	0.94	3.66
$4 \cdot 10^7$	1.25	4.41

(b) Testing with *wrong* queries - probing N missing keys.

Table 4.8: Comparing performances of two HashGraph (Sec. 2.5) probing algorithms (Probe–Standard - Alg. 7 and Probe–New - Alg. 8) in parallel on GPU. The table is built from $4 \cdot 10^7$ elements of randomly generated data and implemented as HashGraphV2 class (see Sec. 4.1).

All values are much smaller than in the CPU version (as expected). However, both preserve linear dependency and the New algorithm is still much slower than the Standard one.

4.4.3 Speedup on GPU compared to CPU

It is clear that all our classes show themselves much more efficient when run on GPU. This is an illustrated analysis of the speedup.

N	CuckooHashSet	HashGraphSSet	HashGraphV1Set	HashGraphV2Set
$5 \cdot 10^6$	3.60	12.67	12.69	10.09
$1 \cdot 10^7$	3.84	14.69	14.55	10.20
$1.5 \cdot 10^7$	3.93	15.47	15.24	10.43
$2 \cdot 10^7$	4.07	15.60	15.69	9.84
$2.5 \cdot 10^7$	4.15	15.71	15.69	9.86
$3 \cdot 10^7$	4.18	15.77	15.96	9.88
$3.5 \cdot 10^7$	4.26	15.99	16.04	9.83
$4 \cdot 10^7$	4.28	15.65	16.11	9.79

Table 4.9: Speedup of building hash tables (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1) with randomly generated data of size N after switching from using a single CPU thread to a parallel GPU run.

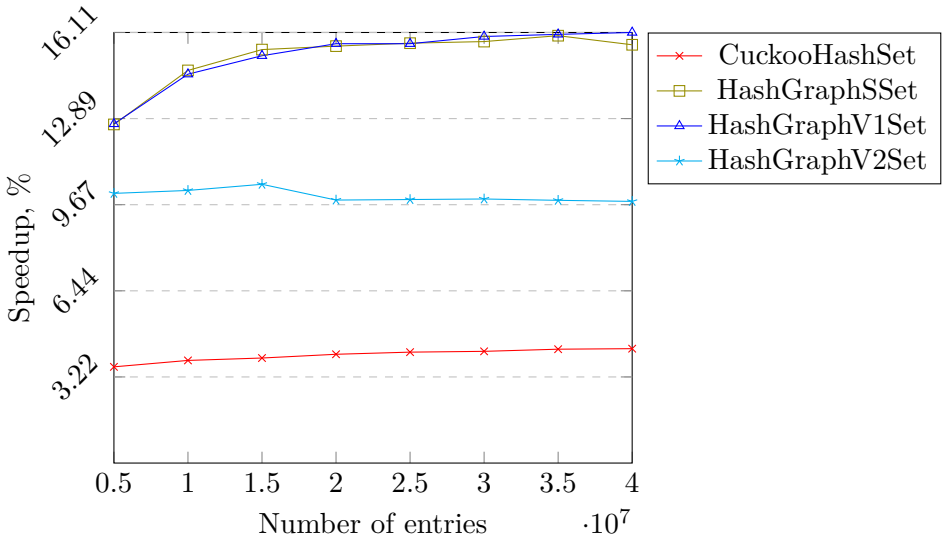


Figure 4.7: Graph of speedup of building hash tables with randomly generated data - see Table 4.9.

4.4.3.1 Building test

Clearly, all three types of hash tables build faster on GPU. And the rate of this acceleration does not seem to depend much on the size of input data, although the trends for HashGraphV1 and HashGraphS are rather logarithmic than constant. As we can observe, it exceeds 15 times for the largest input data while HashGraphV2 and Cuckoo hashing follow almost constantly around 10 and 3.5 times respectively. HashGraph version 2.0 seems to have a tiny decreasing and Cuckoo hashing - a slightly increasing trend, but that is hardly noticeable.

4.4.3.2 Correct query test

N	CuckooHashSet	HashGraphSSet	HashGraphV1Set	HashGraphV2Set
$5 \cdot 10^6$	9.30	7.78	9.20	5.78
$1 \cdot 10^7$	9.89	8.14	9.52	6.38
$1.5 \cdot 10^7$	10.13	8.20	9.74	6.39
$2 \cdot 10^7$	10.18	7.69	9.21	6.06
$2.5 \cdot 10^7$	10.23	7.70	9.29	6.11
$3 \cdot 10^7$	10.25	7.74	9.27	6.15
$3.5 \cdot 10^7$	10.28	7.78	9.29	6.16
$4 \cdot 10^7$	10.28	7.76	9.27	6.17

Table 4.10: Speedup of probing N previously inserted elements of randomly generated data over hash tables of our implementation (CuckooHashSet, HashGraphSSet, HashGraphV1Set and HashGraphV2Set - Sec. 4.1) after switching from using a single CPU thread to a parallel GPU run. Each table was built from the N elements.

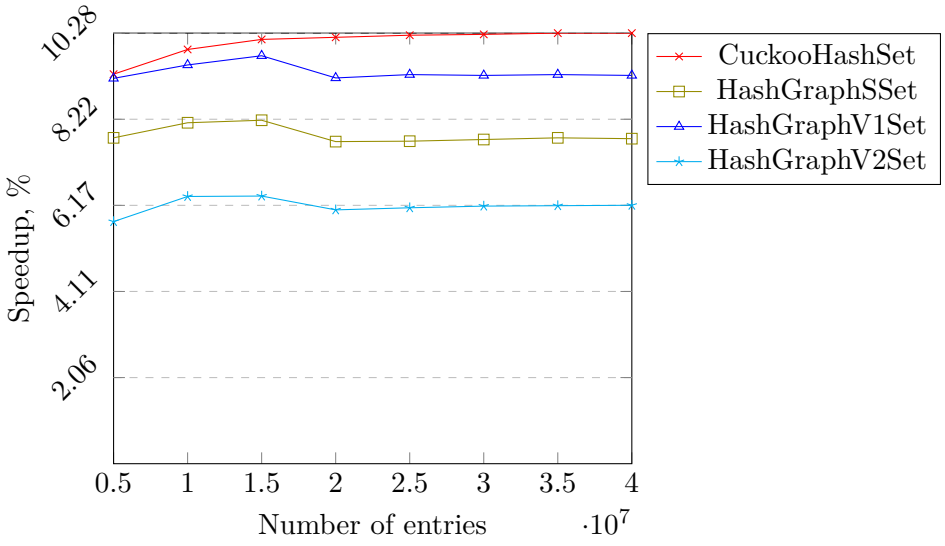


Figure 4.8: Graph of speedup of probing correct keys over hash tables with randomly generated data - see Table 4.10.

These results look more optimistic for Cuckoo hashing, although as we remember, it lost the comparison in actual values to the HashGraphs. Its probing speed has become around 10 times faster on GPU. HashGraphV1 has started at roughly the same point but then stabilized at around 9.3 times. Segment and version 2.0 followed it with the same shape at around 7.7 and 6.15 respectively.

4.4.3.3 Wrong query test

N	CuckooHashSet	HashGraphSSet	HashGraphV1Set	HashGraphV2Set
$5 \cdot 10^6$	11.64	8.28	12.27	7.33
$1 \cdot 10^7$	12.47	8.78	12.83	8.03
$1.5 \cdot 10^7$	12.70	8.81	12.96	8.13
$2 \cdot 10^7$	12.82	8.30	12.44	7.87
$2.5 \cdot 10^7$	12.87	8.34	12.51	7.92
$3 \cdot 10^7$	12.93	8.35	12.55	7.99
$3.5 \cdot 10^7$	12.95	8.35	12.66	8.05
$4 \cdot 10^7$	12.97	8.35	12.66	8.07

Table 4.11: Speedup of probing N missing keys over hash tables of the same size N built with randomly generated data in our implementation (CuckooHashSet, HashGraphSSet, HashGraphV1Set and HashGraphV2Set - Sec. 4.1) after switching from using a single CPU thread to a parallel GPU run.

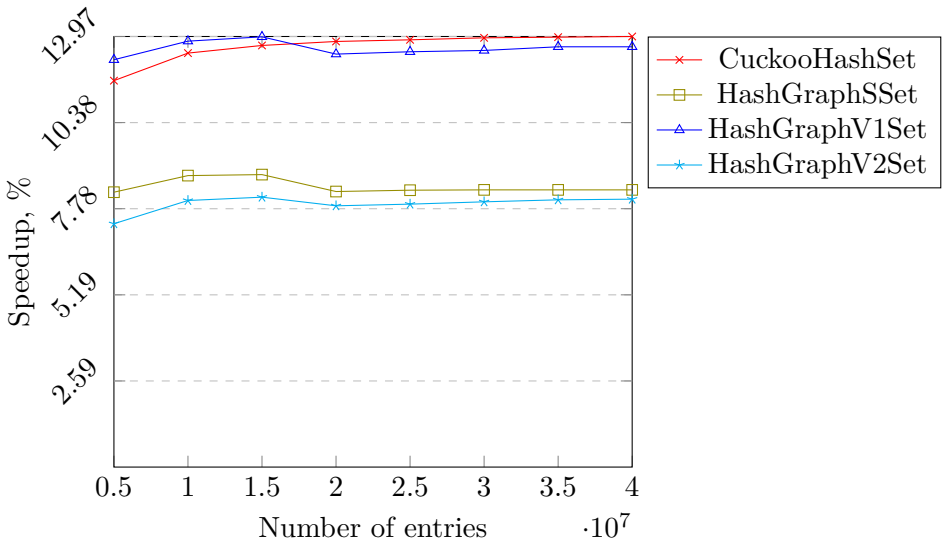


Figure 4.9: Graph of speedup of probing wrong keys over hash table with randomly generated data - see Table 4.11.

In this test, all types of hash tables improved even greater with more than 12 times speedup for Cuckoo hashing and HashGraphV1. HashGraphS and HashGraphV2 were close to each other at around 8 times speedup.

4.4.3.4 HashGraph probing algorithms

We see that both algorithms improve their performance on GPU, and especially for the wrong queries. However, even in this rating HashGraph–Probe–Standard has shown better results.

N	STANDARD	NEW
$1 \cdot 10^7$	5.58	4.08
$2 \cdot 10^7$	5.21	4.10
$3 \cdot 10^7$	5.21	4.22
$4 \cdot 10^7$	5.18	4.38

(a) Testing with *correct* queries - probing N different previously inserted keys.

N	STANDARD	NEW
$1 \cdot 10^7$	7.42	6.11
$2 \cdot 10^7$	6.95	6.31
$3 \cdot 10^7$	6.91	6.46
$4 \cdot 10^7$	6.91	6.63

(b) Testing with *wrong* queries - probing N missing keys.

Table 4.12: Comparing speedups of two HashGraph (Sec. 2.5) probing algorithms (Probe–Standard - Alg. 7 and Probe–New - Alg. 8) with randomly generated data after switching from using a single CPU thread to a parallel GPU run. The table is built from $4 \cdot 10^7$ elements of randomly generated data and implemented as HashGraphV2 class (see Sec. 4.1).

4.5 Results from testing with real data

Performing same tests with real data yielded very similar results showing the same general trends. For the smallest input sizes, it was possible to see the time delays fluctuating in a less linear fashion, which is especially clear when looking at the speedup readings. However, as the amounts of data grow, the observations seen in the previous section start showing themselves again.

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1,313	$3.21 \cdot 10^{-4}$	$3.51 \cdot 10^{-4}$	**0.915**	$3.01 \cdot 10^{-4}$	**1.07**	$3.36 \cdot 10^{-4}$	**0.955**	$1.42 \cdot 10^{-3}$	**0.226**
3,698	$8.87 \cdot 10^{-4}$	$1.04 \cdot 10^{-3}$	**0.855**	$8.44 \cdot 10^{-4}$	**1.05**	$9.40 \cdot 10^{-4}$	**0.944**	$2.40 \cdot 10^{-3}$	**0.370**
29,674	$7.04 \cdot 10^{-3}$	$8.22 \cdot 10^{-3}$	**0.856**	$6.92 \cdot 10^{-3}$	**1.02**	$7.72 \cdot 10^{-3}$	**0.912**	$1.32 \cdot 10^{-2}$	**0.533**
240,373	$5.84 \cdot 10^{-2}$	$6.87 \cdot 10^{-2}$	**0.850**	$5.87 \cdot 10^{-2}$	**0.994**	$6.51 \cdot 10^{-2}$	**0.897**	$1.02 \cdot 10^{-1}$	**0.574**
1,939,414	$6.40 \cdot 10^{-1}$	$7.30 \cdot 10^{-1}$	**0.876**	$6.10 \cdot 10^{-1}$	**1.05**	$6.65 \cdot 10^{-1}$	**0.962**	$8.43 \cdot 10^{-1}$	**0.759**

Table 4.13: Time in seconds spent on building hash tables of our implementation (CuckooHashSet, HashGraphSSet, HashGraphV1Set and HashGraphV2Set - Sec. 4.1) with N elements of real data on a single CPU thread with comparison to std::unordered set.

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1,313	$3.22 \cdot 10^{-4}$	$1.17 \cdot 10^{-3}$	**0.276**	$5.41 \cdot 10^{-4}$	**0.595**	$4.83 \cdot 10^{-4}$	**0.667**	$2.97 \cdot 10^{-4}$	**1.08**
3,698	$8.42 \cdot 10^{-4}$	$3.26 \cdot 10^{-3}$	**0.258**	$1.46 \cdot 10^{-3}$	**0.578**	$1.31 \cdot 10^{-3}$	**0.642**	$8.78 \cdot 10^{-4}$	**0.959**
29,674	$7.15 \cdot 10^{-3}$	$2.60 \cdot 10^{-2}$	**0.275**	$1.17 \cdot 10^{-2}$	**0.609**	$1.04 \cdot 10^{-2}$	**0.684**	$6.29 \cdot 10^{-3}$	**1.14**
240,373	$6.08 \cdot 10^{-2}$	$2.15 \cdot 10^{-1}$	**0.283**	$9.57 \cdot 10^{-2}$	**0.636**	$8.58 \cdot 10^{-2}$	**0.709**	$5.14 \cdot 10^{-2}$	**1.18**
1,939,414	$6.47 \cdot 10^{-1}$	2.39	**0.271**	$8.87 \cdot 10^{-1}$	**0.729**	$8.15 \cdot 10^{-1}$	**0.794**	$5.30 \cdot 10^{-1}$	**1.22**

Table 4.14: Time in seconds spent on probing N previously inserted elements of real data over hash tables on a single CPU thread with comparison to std::unordered set. Each table is built from the same N elements and implemented by one of our classes (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1).

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1,313	$2.24 \cdot 10^{-4}$	$1.07 \cdot 10^{-3}$	**0.210**	$4.74 \cdot 10^{-4}$	**0.473**	$4.58 \cdot 10^{-4}$	**0.489**	$2.24 \cdot 10^{-4}$	**1.00**
3,698	$6.21 \cdot 10^{-4}$	$2.95 \cdot 10^{-3}$	**0.211**	$1.27 \cdot 10^{-3}$	**0.490**	$1.10 \cdot 10^{-3}$	**0.562**	$5.67 \cdot 10^{-4}$	**1.10**
29,674	$5.08 \cdot 10^{-3}$	$2.38 \cdot 10^{-2}$	**0.213**	$1.01 \cdot 10^{-2}$	**0.502**	$8.73 \cdot 10^{-3}$	**0.582**	$4.45 \cdot 10^{-3}$	**1.14**
240,373	$4.45 \cdot 10^{-2}$	$1.97 \cdot 10^{-1}$	**0.227**	$8.22 \cdot 10^{-2}$	**0.542**	$7.12 \cdot 10^{-2}$	**0.625**	$3.64 \cdot 10^{-2}$	**1.22**
1,939,414	$5.93 \cdot 10^{-1}$	2.28	**0.260**	$7.34 \cdot 10^{-1}$	**0.808**	$6.45 \cdot 10^{-1}$	**0.920**	$3.61 \cdot 10^{-1}$	**1.64**

Table 4.15: Time in seconds spent on probing N missing keys over hash tables on a single CPU thread with comparison to std::unordered set. Each table is built from different N elements of real data and implemented by one of our classes (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1).

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1,313	$3.02 \cdot 10^{-4}$	$2.63 \cdot 10^{-3}$	**0.115**	$3.22 \cdot 10^{-4}$	**0.938**	$3.11 \cdot 10^{-4}$	**0.971**	$6.57 \cdot 10^{-4}$	**0.460**
3,698	$8.35 \cdot 10^{-4}$	$3.43 \cdot 10^{-3}$	**0.243**	$3.80 \cdot 10^{-4}$	**2.20**	$3.60 \cdot 10^{-4}$	**2.32**	$7.10 \cdot 10^{-4}$	**1.18**
29,674	$6.96 \cdot 10^{-3}$	$6.69 \cdot 10^{-3}$	**1.04**	$1.03 \cdot 10^{-3}$	**6.79**	$1.02 \cdot 10^{-3}$	**6.86**	$1.79 \cdot 10^{-3}$	**3.89**
240,373	$6.02 \cdot 10^{-2}$	$3.75 \cdot 10^{-2}$	**1.60**	$7.10 \cdot 10^{-3}$	**8.48**	$7.43 \cdot 10^{-3}$	**8.10**	$1.10 \cdot 10^{-2}$	**5.48**
1,939,414	$6.57 \cdot 10^{-1}$	$2.47 \cdot 10^{-1}$	**2.66**	$5.74 \cdot 10^{-2}$	**11.5**	$6.07 \cdot 10^{-2}$	**10.8**	$8.73 \cdot 10^{-2}$	**7.53**

Table 4.16: Time in seconds spent on building hash tables of our implementation (CuckooHashSet, HashGraphSSet, HashGraphV1Set and HashGraphV2Set - Sec. 4.1) with N elements of real data in parallel on GPU with comparison to std::unordered set running on CPU.

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1,313	$3.32 \cdot 10^{-4}$	$1.79 \cdot 10^{-4}$	**1.85**	$6.46 \cdot 10^{-4}$	**0.514**	$4.82 \cdot 10^{-4}$	**0.689**	$5.04 \cdot 10^{-4}$	**0.659**
3,698	$8.96 \cdot 10^{-4}$	$4.52 \cdot 10^{-4}$	**1.98**	$6.46 \cdot 10^{-4}$	**1.39**	$4.60 \cdot 10^{-4}$	**1.95**	$4.70 \cdot 10^{-4}$	**1.91**
29,674	$7.10 \cdot 10^{-3}$	$4.69 \cdot 10^{-3}$	**1.51**	$1.98 \cdot 10^{-3}$	**3.59**	$1.56 \cdot 10^{-3}$	**4.55**	$1.56 \cdot 10^{-3}$	**4.55**
240,373	$6.02 \cdot 10^{-2}$	$3.71 \cdot 10^{-2}$	**1.62**	$1.68 \cdot 10^{-2}$	**3.58**	$1.33 \cdot 10^{-2}$	**4.53**	$1.35 \cdot 10^{-2}$	**4.45**
1,939,414	$6.40 \cdot 10^{-1}$	$2.97 \cdot 10^{-1}$	**2.15**	$1.28 \cdot 10^{-1}$	**5.01**	$1.02 \cdot 10^{-1}$	**6.30**	$1.09 \cdot 10^{-1}$	**5.87**

Table 4.17: Time in seconds spent on probing N previously inserted elements of real data over hash tables in parallel on GPU with comparison to std::unordered set running on CPU. Each table is built from the same N elements and implemented by one of our classes (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1).

N	std::unordered_set	CuckooHashSet		HashGraphSSet		HashGraphV1Set		HashGraphV2Set	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1,313	$2.36 \cdot 10^{-4}$	$1.27 \cdot 10^{-4}$	**1.86**	$3.85 \cdot 10^{-4}$	**0.613**	$2.27 \cdot 10^{-4}$	**1.04**	$2.12 \cdot 10^{-4}$	**1.11**
3,698	$5.93 \cdot 10^{-4}$	$3.28 \cdot 10^{-4}$	**1.81**	$3.98 \cdot 10^{-4}$	**1.49**	$2.19 \cdot 10^{-4}$	**2.71**	$2.14 \cdot 10^{-4}$	**2.77**
29,674	$5.03 \cdot 10^{-3}$	$3.47 \cdot 10^{-3}$	**1.45**	$1.40 \cdot 10^{-3}$	**3.59**	$8.54 \cdot 10^{-4}$	**5.89**	$8.26 \cdot 10^{-4}$	**6.09**
240,373	$4.46 \cdot 10^{-2}$	$2.82 \cdot 10^{-2}$	**1.58**	$1.24 \cdot 10^{-2}$	**3.60**	$7.40 \cdot 10^{-3}$	**6.03**	$7.33 \cdot 10^{-3}$	**6.09**
1,939,414	$5.94 \cdot 10^{-1}$	$2.28 \cdot 10^{-1}$	**2.61**	$9.77 \cdot 10^{-2}$	**6.08**	$5.90 \cdot 10^{-2}$	**10.1**	$5.78 \cdot 10^{-2}$	**10.3**

Table 4.18: Time in seconds spent on probing N missing keys over hash tables of s in parallel on GPU with comparison to std::unordered set running on CPU. Each table was built from different N elements of real data and implemented by one of our classes (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1).

N	CuckooHashSet	HashGraphSSet	HashGraphV1Set	HashGraphV2Set
1,313	0.134	0.935	1.08	2.16
3,698	0.302	2.22	2.61	3.38
2,9674	1.23	6.76	7.61	7.39
240,373	1.83	8.28	8.76	9.26
1,939,414	2.95	10.60	11.00	9.66

Table 4.19: Speedup of building hash tables (CuckooHashSet, HashGraphSSet, HashGraphV1Set or HashGraphV2Set - Sec. 4.1) with real data of size N after switching from using a single CPU thread to a parallel GPU run.

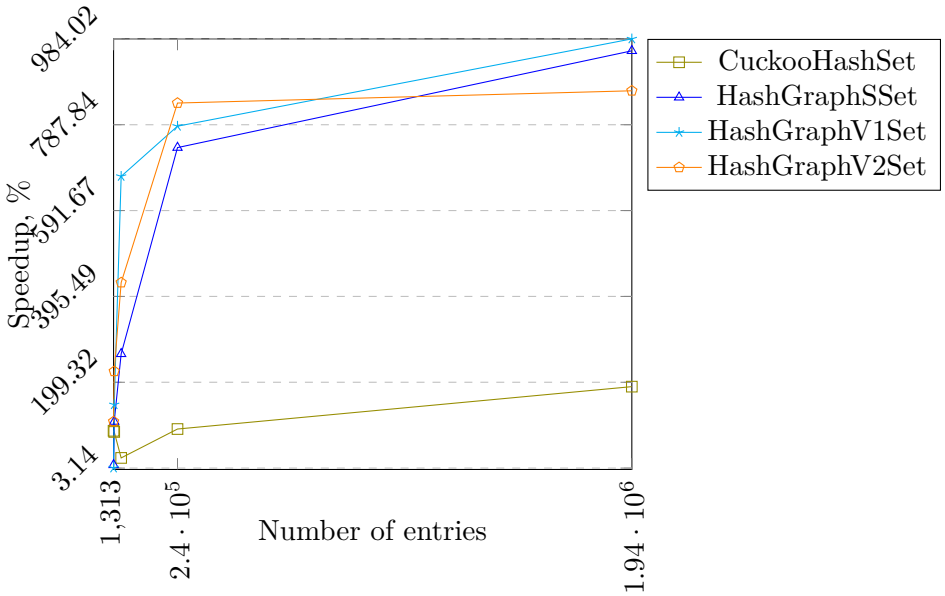


Figure 4.10: Graph of speedup of building hash tables with real data - see Table 4.19.

N	CuckooHashSet	HashGraphSSet	HashGraphV1Set	HashGraphV2Set
1,313	6.53	0.837	1.00	0.589
3,698	7.22	2.25	2.85	1.87
29,674	5.55	5.93	6.70	4.04
240,373	5.78	5.69	6.46	3.79
1,939,414	8.04	6.94	8.01	4.86

Table 4.20: Speedup of probing N previously inserted elements of real data over hash tables of our implementation (CuckooHashSet, HashGraphSSet, HashGraphV1Set and HashGraphV2Set - Sec. 4.1) after switching from using a single CPU thread to a parallel GPU run. Each table was built from the N elements.

N	CuckooHashSet	HashGraphSSet	HashGraphV1Set	HashGraphV2Set
1,313	8.39	1.23	2.02	1.06
3,698	8.98	3.19	5.04	2.65
29,674	6.87	7.23	10.2	5.38
240,373	6.98	6.64	9.62	4.96
1,939,414	10.00	7.51	10.90	6.24

Table 4.21: Speedup of probing N missing keys over hash tables of the same size N built with real data in our implementation (CuckooHashSet, HashGraphSSet, HashGraphV1Set and HashGraphV2Set - Sec. 4.1) after switching from using a single CPU thread to a parallel GPU run.

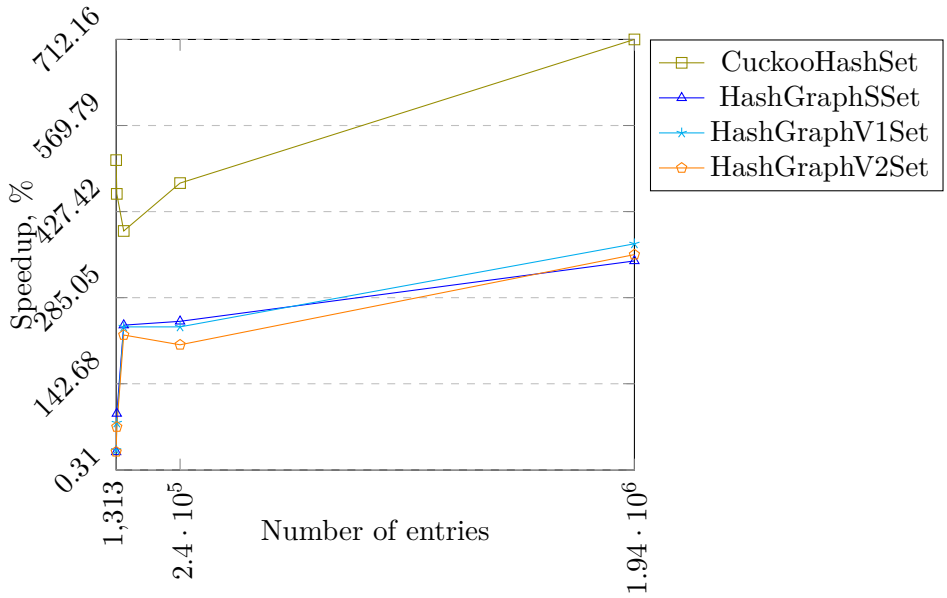


Figure 4.11: Speedup of probing correct keys over hash tables with real data - see Table 4.20.

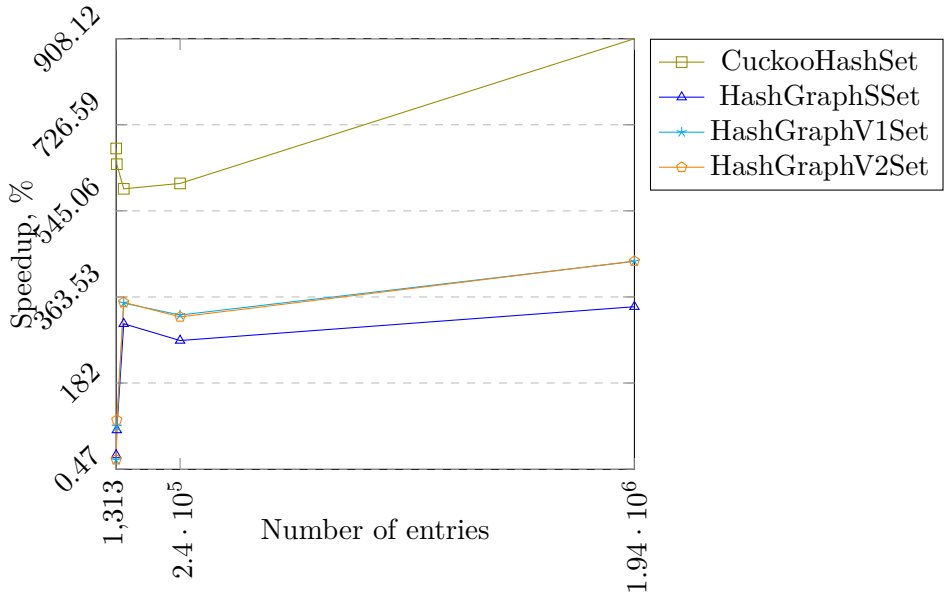


Figure 4.12: Speedup of probing wrong keys over hash tables with real data - see Table 4.21.

Conclusion

The purpose of this thesis was to understand the principles of programming for GPU, study and implement some algorithms and data structures for hashing on GPU, test the implemented approaches and compare their performances to each other and to the version in the standard library of C++.

We studied the ideas behind CUDA programming model - a tool that allows to develop programs with easy shifting between CPU and GPU using high-level languages like C++. We also got familiar with Template Numerical Library that simplifies this task even further by providing a convenient interface for the functionality that CUDA is often used for.

Next, this thesis dived into understanding of the idea of a hash table that encorporates a large set of data structures and related algorithms for time-efficient storage and probing of data. From the vast family of those structures, each of which employs different techniques to handle hash collisions, we selected and explored two approaches that specifically aim to be efficient with parallel execution using GPU. The first approach of our choice was Cuckoo hashing whose idea is based on continuous eviction and reinsertion of keys using different hash functions. The second was a HashGraph - a structure in two versions that views a hash table as a bipartite graph and represents it in a space-efficient form of a Compressed sparse row.

Both types of hashing structures were implemented by means of TNL with derivations allowing to use each of them as a Set or a Map. Version 1.0 of HashGraph was done in two variants: one closely following the original algorithm and the second one making use of TNL's class representing CSR (we called it the Segments variant). After that, their functionality and speed were tested. The first test suite consisted of GoogleTest-based unit tests, while the second one tested the performance with real and randomly generated data. In the second set, the classes' speed was compared to the one of std::unordered_set that appears to be a version of hash set used in the Standard template library of C++.

The tests have shown a manyfold increase in efficiency for our implemen-

tation on GPU compared to CPU. Both approaches proved to be faster in parallel than the STL implementation, but all three versions of HashGraph have shown better results than Cuckoo hashing. In fact, they were so fast that they managed to overtake the std::unordered_set even on CPU when tested with large amounts of data. It was also observed that querying all classes with existing keys was a bit slower than doing so with keys that were not inserted.

Contrary to our expectations, the building algorithm for the HashGraph version 2.0 was slower then for 1.0 and Segments variant. Its performance while probing the correct keys was also slightly better, although this difference disappeared while probing wrong keys on GPU. Moreover, HashGraph–Probe–New has failed to overtake the HashGraph–Probe–Standard in the probing time on both devices and with no regard to the number of keys queried.

When comparing time spent by all the classes on building on both devices, we are most satisfied with HashGraph version 1.0 in both classical and Segments variants. HashGraph version 2.0 followed and Cuckoo hashing was the worst. In querying, on the other hand, Cuckoo hashing has shown itself better. HashGraphV1 followed with Segments version speeding up more, and HashGraphV2 was the last one.

In conclusion, from the algorithms that we have explored, the most efficient choice for both building and running on GPU from the performance point of view appears to be a HashGraph of the version 1.0, and it is better implemented without Segments. That is also true for CPU when used for large data.

Possible future improvements

The most helpful addition to our implementation can be a support for dynamic allocation, that being an ability to insert new keys after the process of building was finished. It is possible to realize with Cuckoo hashing already, although the effect of this on its performance is uncertain and must be tested. Similar is believed to be possible for the HashGraphs, however the dynamic versions their algorithms have not been published yet.

Bibliography

[1] Lessley, B.; Childs, H. Data-Parallel Hashing Techniques for GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems*, volume 31, no. 1, 2020: pp. 237–250, doi:10.1109/TPDS.2019.2929768.

[2] Alcantara, D. A.; Volkov, V.; et al. Building an Efficient Hash Table on the GPU. In *GPU Computing Gems Jade Edition*, Elsevier, 2012, pp. 39–53, doi:10.1016/b978-0-12-385963-1.00004-6. Available from: `https://doi.org/10.1016/b978-0-12-385963-1.00004-6`

[3] Cormen, T. H.; Leiserson, C. E.; et al. *Introduction to algorithms*. MIT press, 2009.

[4] Green, O. HashGraph - Scalable Hash Tables Using A Sparse Graph Data Structure. *CoRR*, volume abs/1907.02900, 2019, `1907.02900`. Available from: `http://arxiv.org/abs/1907.02900`

[5] Celis, P. *Robin Hood Hashing*. Dissertation thesis, CAN, 1986.

[6] 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)*, Oct 2015, ISSN 1089-795X, pp. 63–74, doi:10.1109/PACT.2015.13.

[7] Jünger, D.; Hundt, C.; et al. WarpDrive: Massively Parallel Hashing on Multi-GPU Nodes. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 441–450, doi:10.1109/IPDPS.2018.00054.

[8] Lefebvre, S.; Hoppe, H. Perfect Spatial Hashing. *ACM Trans. Graph.*, volume 25, no. 3, July 2006: p. 579–588, ISSN 0730-0301, doi:10.1145/1141911.1141926. Available from: `https://doi.org/10.1145/1141911.1141926`

[9] Ashkiani, S.; Farach-Colton, M.; et al. A Dynamic Hash Table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, May 2018, doi:10.1109/ipdps.2018.00052. Available from: https://doi.org/10.1109/ipdps.2018.00052

[10] Tumblin, R.; Ahrens, P.; et al. Parallel Compact Hash Algorithms for Computational Meshes. *SIAM Journal on Scientific Computing*, volume 37, no. 1, Jan. 2015: pp. C31–C53, doi:10.1137/13093371x. Available from: https://doi.org/10.1137/13093371x

[11] Duan, W.; Luo, J.; et al. Exclusive grouped spatial hashing. *Computers & Graphics*, volume 70, 2018: pp. 71–79.

[12] Nießner, M.; Zollhöfer, M.; et al. Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)*, volume 32, no. 6, 2013: pp. 1–11.

[13] Harris, M.; Sengupta, S.; et al. *Parallel prefix sum (scan) with CUDA*, volume 39. 08 2007, pp. 851–.

Appendix A

Acronyms

CPU Central processing unit

CSR Compressed sparse row

CUDA Compute unified device architecture

GPU Graphics processing unit

SIMT Single instruction multiple threads

STL Standard template library

TNL Template numerical library

Appendix B

Content of enclosed media

- `bin` ..... directory containing binary files
- `CuckooHash` ..... directory with Cuckoo hash table, view, Map and Set
- `data` ..... directory with text files with data used for performance testing
- `HashGraph` ..... directory with HashGraph implementation
  - `HashGraphV1` .. directory with HashGraph Version 1.0 implementation
  - `HashGraphV2` .. directory with HashGraph Version 2.0 implementation
  - `HashGraphMap.h` ..... Map classes derived from both versions
  - `HashGraphSet.h` ..... Set classes derived from both versions
- `UnitTests` ..... directory with unit tests implemented with GoogleTest
- `HashFunction.h` ..... HashFunction used by all implementations
- `main.cpp` ..... file with main() function
- `Makefile`
- `Measurable.h` ..... base class enabling time measurement
- `Pair.hpp` ..... class used by Map classes
- `readme.txt` ..... description of contents
- `test_set.h` ..... performance tests for all set classes
- `test_set.hpp`
- `thesis.pdf` ..... thesis in PDF format
- `Value.hpp` ..... class used by Set classes for consistency with Maps