Efficient Hash Tables on the GPU

By

Dan Anthony Feliciano Alcantara

B.S. (University of California, Davis) 2005
PhD (University of California, Davis) 2011

Dissertation

Submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

Nina Amenta

Kwan-Liu Ma

John Owens

Committee in Charge

2011

To my parents.

Contents

Abstract

Advances in GPU architecture have made efficient implementations of hash tables possible, allowing fast parallel constructions and retrievals despite the uncoalesced memory accesses naturally incurred by hashing algorithms. The key is to mitigate the penalty of these accesses by minimizing the number that occur and utilizing the cache (when one is available). Most work done on parallel hashing is ill-equipped for this objective and relies on the theoretical PRAM model, which abstracts away the difficulties of programming on actual hardware. We examine hashing schemes from a practical perspective using NVIDIA's CUDA architecture.

Our main contribution is a set of parallel implementations for open addressing, chaining, and cuckoo hashing. We analyze each method and identify when applications should use one over another. Because each makes different performance trade-offs, we compare them using three metrics: *memory usage*, *construction time*, and *retrieval efficiency*. Retrieval efficiency considers both the average time and deviation from it, since answering some queries can be several orders of magnitude more difficult than others.

Our quadratic probing implementation shows this as the hash table becomes more compact: on a GTX 470, using datasets containing 10M random key-value pairs, it has respective rates of [369M, 723M, 539M] pairs per second (pps) for insertion, retrieving every input item, and retrieving 10M keys absent from the table when using $2N$ space. For $1.05N$ space, these rates significantly drop to [162M, 208M, 46M] pps, reflecting the difficulty of terminating both insertions and queries.

Applications requiring more robust retrieval could benefit from our chaining implementation, which eschews linked lists and uses radix sort for an efficient parallel construction. When using $2N$ space, the rates are [344M, 436M, 624M] pps, while for $1.05N$ the rates are [449M, 211M, 126M] pps. For compact tables, its construction rate is almost 3x faster than quadratic probing with a smaller drop in retrieval efficiency for failed queries.

However, the number of probes required to answer queries grows drastically for compact tables, leading to poorer retrieval rates. Cuckoo hashing is better suited for these cases, trading a more complicated construction for guaranteed constant time retrievals. It has rates of [366M, 670M, 501M] pps using $2N$ space and [133M, 386M, 258M] pps for $1.05N$. Our method is also adaptable and can be specialized for situations where multiple values are stored per key.

Acknowledgments

Thousands of caffeinated drinks later, I've finally reached the end of this phase in my life. Throughout my career, I've had the honor of meeting some extremely bright folks, traveled around the globe, and worked on challenging research. There are many people I'd like to thank for coming along for the ride, but I've pared down my list to keep this section short and sweet.

First and foremost, I want to thank my parents for supporting me over the years. They kindled my passion for computers at a very young age and nurtured it, shaping me into who I am today. It's impossible to enumerate the sacrifices they have made – and continue to make – to help me realize my full potential.

I am greatly indebted to my advisor Nina Amenta, whom I have worked with since my undergraduate years here. She took me under her wing and gave me my first taste of research, supporting me the entire way. I thank John Owens and Michael Mitzenmacher for their advice; every nugget of knowledge that they shared was golden. I also thank the other collaborators on my hashing work, Vasily Volkov, Shubho Sengupta, Fatemeh Abbasinejad, and Andrei Sharf, who provided vital insights.

I'm grateful for all of my friends, but most especially Tony Bernardin and Deboshmita Ghosh. They helped me keep my sanity during my graduate work and helped me to remember that there is life outside my apartment (I know, it's hard to conceive).

I thank Chris Co and Jim Shearer for helping me score internships at Google and Apple, respectively. The skills I picked up while working at those companies were extremely handy for my academic research. I also want to thank the guys on Apple's OpenCL team for mentoring me; specifically Affie Munshi, Inam Rahman, and Mon Ping Wang. The experiences I had and the things I learned about GPU programming were invaluable for my last year here.

With that out of the way: so long, and thanks for all the fish!

Funding sources My research was primarily funded by the National Science Foundation (awards CCF-0635250 and IIS-0964357), the AGEP program, the GAANN fellowship in 2005, and work-study awards from UC Davis.

Additional funding came from the NSF (awards CCF-0541448, CCF-0625744, CNS-0721491, and IIS-0964473), the SciDAC Institute for Ultrascale Visualization, Cisco, Google, NVIDIA, Microsoft (Award 024263), Intel (Award 024894), U.C. Discovery (Award DIG07-10227), and Par Lab affiliates.

Chapter 1

Introduction

The advent of programmable rendering pipelines for Graphics Processing Units (GPUs) was a boon for computer graphics, allowing shader programs to reconfigure how the GPU processed data and rendered images for display. Before long, the highly parallel architecture of the GPU was recognized for its extremely fast number crunching abilities, giving rise to techniques for applying the GPU to non-graphical computations. However, these methods were relatively unintuitive: they required shoehorning data into textures and operating on them using shaders that didn't actually produce pixels for display. Since the introduction of parallel programming architectures that remove the need for these workarounds, the popularity of the GPU has expanded to applications completely unrelated to computer science, like biology, chemistry, and even finance.

These applications build upon and rely on data structures that can be both built and used efficiently in parallel environments. Ideally, the data structures would be constructed on the GPU itself using an efficient parallel method to avoid being the bottleneck for a parallel application. Several such algorithms were recently demonstrated by researchers for hierarchical spatial data structures, like octrees [8,40,42] and k-d trees [43] that can be used for real-time raytracing. In general, however, the problem of defining parallel-friendly data structures that can be efficiently created, updated, and accessed is a significant research challenge [25], and the toolbox of efficient data structures and their associated algorithms on scalar architectures

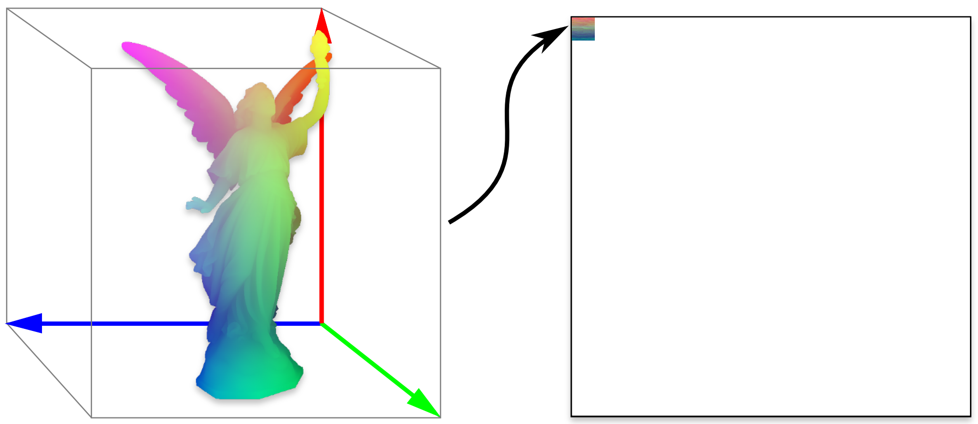


Figure 1.1. Typical sparse spatial dataset. Pictured on the left is a voxelized version of the Lucy model, color-coded so that red, green, and blue map to the x, y, and z coordinates of each voxel. Storing the entire voxel grid is extremely wasteful since the majority of the cells are empty (right).

like the CPU remains significantly larger than on parallel architectures like the GPU.

We focus on compact data structures that provide efficient random-access to data pulled from a sparse universe. One example of this type of data is the set of voxels of a grid occupied by a surface mesh; Figure 1.1 shows an example of the Lucy dataset[1] embedded in a voxel grid. The number of occupied voxels is often expected to be $O(N^2)$; storing the entire N^3 grid is extremely wasteful because the majority of the grid is empty. A more practical option is to store only the occupied voxels: querying the data structure with a voxel ID either returns a value associated with the voxel or fails, indicating that the voxel was empty.

Hash tables are popular for this type of data because they can be constructed to answer queries with an average of $O(1)$ memory accesses; other basic data structures like sorted arrays or linked lists require $O(\lg N)$ or $O(N)$ memory accesses, respectively. Applications using these "spatial hash tables" include volumetric painting [23] and collision detection between deforming models [2]; an example of the latter is shown in Figure 1.2[2].

[1]Provided by the Stanford 3D Scanning Repository.
[2]Thanks to Daniel Vlasic for providing the models.

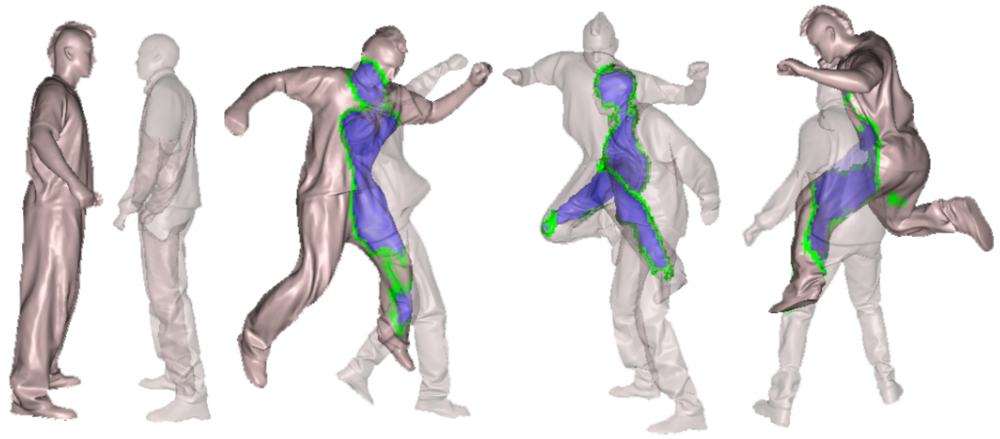


Figure 1.2. GPU hash tables are being constructed and queried every frame to perform Boolean intersections for these two animated models. Blue parts of one model represent voxels inside the other model, while green parts mark surface intersections. These images were produced using a 128^3 voxel grid for point clouds of approximately 160k points. We achieve frame rates between 25–29 fps on a GTX 280, with the actual computation of the intersection and flood-fill requiring between 15–19 ms. Most of the time per frame is devoted to actual rendering of the meshes.

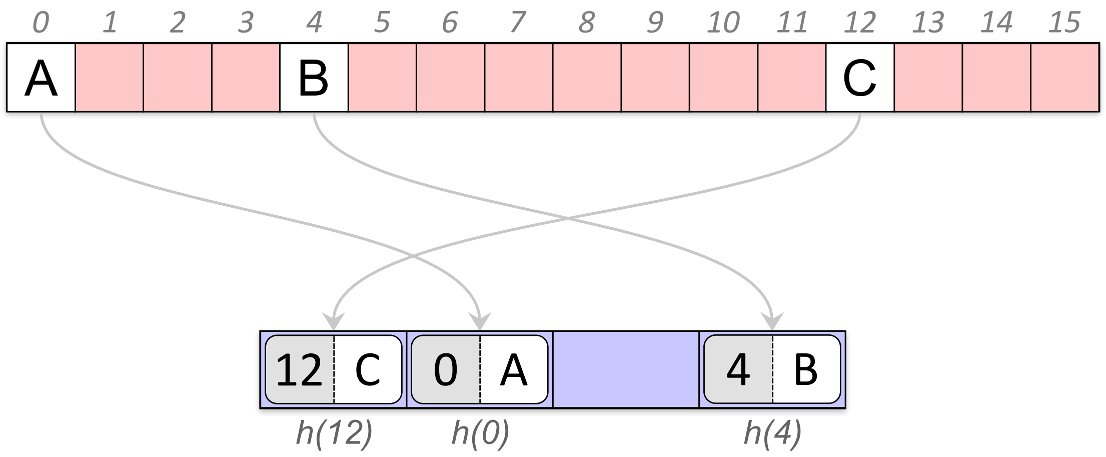


Figure 1.3. While allocating storage for the value of every possible key in an array allows directly indexing into the structure, it is wasteful when the array is mostly unused (top). A hash table can be used instead, which allocates far less space than the array (bottom). In this example, each slot holds both a key and its value. The table is indexed into using a hash function $h(k)$. Because multiple keys may map to the same location, the key contained in the slot and the query key are compared on a retrieval to ensure the right value is returned.

A basic hash table consists of a set of slots that is slightly larger than the size of the input data (Figure 1.3). Every possible item from the universe is mapped to one of these slots by a hash function, but each location can have multiple items mapping into it because the number of slots is significantly smaller than the size of the universe. When two or more input items map to the same slot, a *collision* occurs. Traditional collision-resolution methods include open addressing, which marches through the hash table until an empty slot is found, and chaining, which keeps separate linked lists of all items falling into each slot. Although these methods can be effective in a serial environment, they must be adapted for the highly parallel GPU:

- **Serialization** is difficult to avoid during parallel insertion, but is required to prevent race conditions. For example, two threads attempting to insert into the same slot simultaneously could overwrite each other's insertion without a lock or atomic operation.

- **Memory accesses are slow** when they cannot be *coalesced*, or grouped together; uncoalesced memory accesses are an order of magnitude slower than coalesced accesses. Threads must access nearby locations in memory for coalescing to occur, but hash tables exhibit little locality in either construction or access: they achieve $O(1)$ retrievals by using randomness to scatter items throughout the table. Moreover, the scattering limits the effectiveness of the cache on modern GPUs.

- **Many probes may be required** to find an item in the hash table, which can increase with more tightly packed hash tables. This leads to some inefficiency on the GPU, where the SIMD cores force all threads to wait for the worst-case number of probes. Chaining, for example, partitions the items into variably-sized buckets and has an *expected* constant look-up time, but the lookup time for *some* item in the table is $\Omega(\lg \lg N)$ with high probability.

Relatively little work has focused on practical parallel implementations of hash tables to address these issues. The influential work of Lefebvre and Hoppe [23], among the first to use the GPU to access a hash table, addressed the issue of variable probe length by using a *collision-free hash table*. This guarantees that an item can be accessed in worst-case $O(1)$ time, limiting the number of probes and memory accesses required to answer a query. The trade-off for this efficient retrieval is an inherently serial construction algorithm that is performed as a preprocessing step on the CPU, resulting in slow build times and limiting their applications to static data sets.

In this dissertation, we explore the difficulties of building hash tables on the GPU itself to reduce serialization of parallel applications. We aim for parallel-friendly construction methods that allow hash tables containing millions of items to be built at rates fast enough for interactive applications, while still providing efficient random access to their contents. Our contribution is a set of four different parallel hash table implementations, each of which can be built and queried on the GPU. We analyze the strengths and weaknesses of each method, discuss the trade-offs that can be made to change how they perform, and examine situations where each outperforms the other.

We begin in Chapter 2 with a brief overview of GPU programming and considerations that must be made for efficient use of the hardware. The chapter also covers previous work on parallel hash tables and general GPU data structures and describes our criteria for judging parallel hash table implementations. Chapter 3 describes and analyzes our implementations of the common open addressing methods, while Chapter 4 explores our chaining algorithm. Chapters 5 and 6 discuss our implementations of cuckoo hashing, which address issues with retrieval efficiency faced by the other methods. Chapter 7 concludes with thoughts about future directions.

Chapter 2

Background

To better understand our algorithms, this chapter goes over background material required to frame our discussion. Section 2.1 gives a brief overview of general-purpose computing using GPUs. We exclusively use NVIDIA's CUDA parallel computing architecture, though the ideas we describe can also be applied using OpenCL, a similar but less-adopted framework. Section 2.2 discusses previous work on GPU data structures, as well as relevant work on hash tables. We close with Section 2.3, which outlines how we judge the performance of hash tables we explore in later chapters.

2.1 Parallel computing using CUDA

The introduction of parallel programming architectures like CUDA lowered the barrier to entry for general-purpose computing on GPUs. These computing frameworks provide essential functionality for parallel applications, such as scattered writes in memory and atomic operations, that were simply not available using previous methods. We briefly introduce the features of CUDA relevant to our hashing work in this section; those interested in a full guide to CUDA programming should read NVIDIA's extensive guide [33]. Specifically, we use *CUDA C*, a high-level GPU programming language that extends C with extra constructs for dealing with the hardware.

Programs run on the GPU are called *kernels* and typically consist of just a few small functions. Kernels are executed in parallel by *threads*, each performing the same set of instructions on different data. Simple examples include programs that apply the same affine transformation to every input point, or compute the hash function value of every input key. While this model is more restrictive than the "Multiple-Instruction, Multiple-Data" model used by CPUs, which allow threads to operate independently and perform completely different tasks, the lack of flexibility allows the hardware to be highly optimized for data-parallel tasks and throughput.

One of the biggest limitations is that copying data to and from the GPU is very expensive. Kernels generally do not have access to the host system's memory, so data has to be copied onto the GPU before processing it and written back afterward. Parallel applications can avoid repeatedly incurring this expensive data transfer by using data structures that can be built and used entirely in parallel, allowing the data to stay on the GPU while it is processed.

Threads are grouped together into *thread blocks* of up to 512 threads, which are assigned to different *streaming multiprocessors* (*SM*s) for execution. Because literally millions of threads can execute the kernel and only a finite number of SMs exist, the thread blocks are queued up for the SMs and fed in as the thread blocks finish; thread blocks can complete execution before others are even started, so there is no way to globally synchronize all of the threads without finishing the kernel. Threads in the same block can, however, locally synchronize using execution barriers, guaranteeing that they have all reached the same point before continuing.

Multiple thread blocks can be handled by an SM simultaneously, but there is a hard limit on the number of threads the SM can handle and the threads must partition the resources available to each SM; among these are a set of fast registers accessible only by each thread, and a low-latency (but small) on-chip memory that can be accessed by threads in the same block.

Each SM breaks its thread blocks into groups of 32 consecutive threads called *warps*. SMs manage and schedule when each of their warps will be executed by their SIMD cores, with each thread running the same instructions in lockstep – even when a branch occurs. If any threads in the warp fail to take the same branch as all of the other threads, the instructions for all of the branches are executed (though the results are not stored if the branch would not have been taken). These divergent branches cause some performance degradation, and should be avoided when possible. By extension, threads in the same warp also perform memory accesses together; when two or more threads simultaneously write to the same location, CUDA guarantees that one arbitrary thread will succeed.

There are two main types of memory that we are concerned with: low-latency *shared memory* and high-latency *global memory*. Shared memory is typically used as a cache for global memory and as a scratchpad for threads working in the same thread block, allowing them to cheaply communicate and work together to process larger batches of data. Accessing it can be as fast as using registers with certain memory access patterns. However, it is small (typically 16KB) and partitioned so that threads from different blocks are unable to access another block's shared memory. Moreover, it does not persist between kernel executions, so results must be output to global memory.

Global memory is abundant and accessible to all threads, but is orders of magnitude slower to access. To hide the latency, SMs automatically context switch to other warps while the memory transactions are being performed. They are more able to do so when they have a high *occupancy* and are free to switch between as many warps as needed while waiting for other warps.

SMs further hide latency by reading up to 128-byte segments of memory with a single transaction. Although the requirements change with every generation of GPUs, in general the memory requests of threads in a warp accessing the same segment of memory are *coalesced*, or combined, together into fewer memory transactions. Threads failing to satisfy the requirements incur separate transactions.

For example, if 16 threads are attempting to access memory locations that are distant from one another, 16 separate memory transactions have to be performed. This creates a huge bottleneck for parallel applications and is especially damaging for hash tables, since by nature they exhibit very little spatial locality. Reducing the number of these uncoalesced accesses is therefore a high priority.

For situations where race conditions are difficult or impossible to avoid, atomic operations can be performed on both shared and global memory. Atomics perform a series of actions that cannot be interrupted; examples include incrementing a counter and conditionally setting a memory location based on its current value. These operations are extremely helpful for threads in different thread blocks communicating with each other. However, they are costlier than a normal memory access, especially when many threads are performing the same atomic operation on the same memory location.

Newer GPUs from NVIDIA are based on the *Fermi* architecture, which introduces significant changes to how applications perform [32]; these cards have higher *compute capabilities*, signifying that they have more functionality than previous generations of cards. Among the changes are more efficient atomic operations and a cached memory hierarchy to further reduce latency when accessing global memory. Each streaming multiprocessor now has access to its own small L1 cache, as well as a larger, L2 cache shared between all of the SMs. These caches can be helpful for hashing methods like linear probing, where a thread may have to probe sequential locations in memory to answer queries, and result in different disadvantages for the hash tables when built on different GPUs.

2.2 Related work

We break this section into two pieces, beginning with a brief review of work on GPU data structures, then discuss work on hash tables.

2.2.1 GPU data structures

The majority of work on GPU data structures are for structures that are produced on the CPU as a pre-processing step, but used on the GPU to accelerate parallel applications. More recent work has focused on producing these data structures on the GPU directly using parallel construction algorithms.

The GLIFT library of Lefohn et al. [24] allows a method for generating multiresolution adaptive data structures that can be updated. These can be used to generate quadtrees and octrees, which are sparse spatial data structures that partition space into increasingly smaller boxes [8, 40]. One application is to convert oriented point cloud data into surface meshes; Zhou et al. [42] build their octrees on the GPU at fast enough rates to allow interactive surface creation and deformation.

KD-trees also partition space, but do so using a hierarchy of axis-aligned splitting planes. Previous work has relied on building the kd-tree on the CPU side, then using it on the GPU to accelerate nearest neighbor queries and raytracing operations [12, 21]. However, the recent work of Zhou et al. [43] constructs the KD-trees at interactive rates on the GPU, allowing for a raytracing application with dynamic scene geometry where the kd-tree is built for each frame.

Sorted arrays are a popular focus of research. Most work has focused on producing faster and faster GPU implementations of the *radix sort* algorithm, which is highly parallelizable [19, 20, 29, 38]. Sorted arrays can be used as alternatives to hash tables that use minimal space and can be constructed at extremely fast rates. Retrieval timings are highly varied, however, as a binary search is required. If the queries are sorted, the branching patterns and memory reads will tend to be coalesced into fewer memory transactions, reducing the cost of parallel queries significantly. Randomly accessing the array incurs as many as $O(\lg N)$ probes in the worst case, which requires many more probes than the hash tables we explore.

2.2.2 Hash tables

Although much work has been done on hash tables in sequential computing, the bulk of parallel hashing work has been done for theoretical PRAM models; relatively little work has been done for practical parallel implementations.

Open addressing hash tables are among the most basic hashing schemes, where N input items are distributed between $S_T \geqslant N$ slots [31,35]. Each of these slots is capable of holding one key-value pair. Collisions are handled by probing the slots until an empty slot is found; the simplest method is *linear probing*, which scans every slot in a sequential manner until an empty slot is found. Other methods generate different probe sequences: *quadratic probing* generates a probe sequence that makes increasingly larger jumps using a quadratic function, while *double hashing* uses a secondary hash function to tailor the sequences for each key. While they can be very fast for both construction and retrieval on the GPU, requiring very few probes to answer a query when the table contains few items, problems arise when trying to make a compact table: in the worst case, the whole table will have to be traversed to terminate a query.

Previous work on parallel open addressing hash tables [16, 36] has no clear adaptation for a highly parallel GPU environment, but we do present a parallel implementation of the common probing schemes in Chapter 3.

Chaining handles collisions by appending all items hashing into the same slot into a linked list associated with the slot. You can expect a fair distribution of the items with a good hash function, resulting in lists with an expected average length of $O(N/S_T)$. As with open addressing, however, the number of probes increases greatly as the number of slots shrinks. More importantly, linked lists are horribly inefficient for the GPU: space is wasted storing pointer information for each node, and following the pointers is extremely inefficient if the nodes are scattered throughout memory. This makes parallel chaining methods that use linked lists, like the work by Shalev and Shavit [39], impractical for the GPU. This is underscored by the poor performance of the implementation by Sanders and Kandrot [37]; they found that their version was just as slow as their serial implementation and concluded that hash tables were a poor fit for the GPU.

Replacing the linked lists with contiguous arrays results in much better memory access patterns [4,5]. We present a parallel chaining implementation that is based on a radix sort in Chapter 4.

Collision-free hashing addresses the issue of repeated probing by guaranteeing that every item can be located in exactly one location in the table, allowing all queries to be answered with a single probe. One of the earliest examples is the work of Fredman et al. [14]. They proved that if the size of the hash table is much larger than the number of items (specifically $\Theta(N^2)$), then with some constant probability a randomly chosen hash function will cause no collisions, giving constant lookup time. While the space requirements would be impractical for even small data sets, they reduce the space required to $O(N)$ by first partitioning the input into tiny buckets. A collision-free hash table is then built and stored for each smaller bucket (Figure 2.1). With enough buckets, the number of items expected to fall within each is $O(1)$ and the expected size of the hash table drops to $O(N)$ while retaining $O(1)$ retrieval efficiency. This method has a straightforward parallel implementation, theoretically requiring $O(\lg N)$ time on a CRCW-PRAM [26]. However, the main drawback is the amount of space required by the table: the basic version of their hash table requires $6N$ space, but it can be reduced to $N + o(N)$ with extra bookkeeping.

Much research followed along the same vein, mostly in the early nineties [7,17, 18,26,27]. While these approaches achieve impressive results (guaranteeing $O(1)$ lookup time, $O(\lg \lg N)$ parallel table construction time, and $O(N)$ size), the implicit constant factors, especially on space, are unreasonable, and the constructions use features of the theoretical PRAM model not available on actual GPUs.

Work was also done toward constructing *minimal* perfect hash tables, which store N items in exactly N locations. Minimal, or even near-minimal, perfect hash tables reduce the space overhead at the cost of increased construction time.

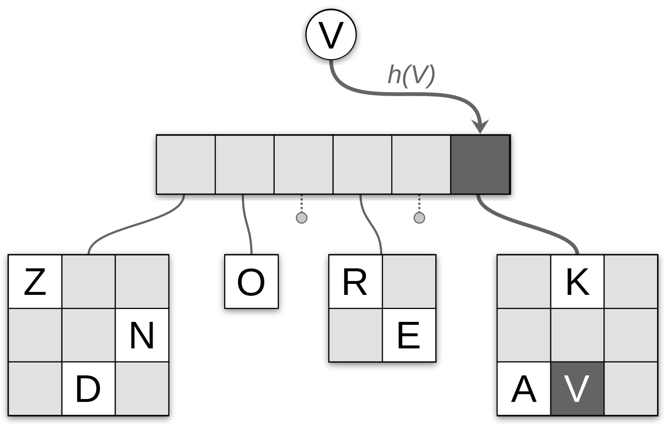


Figure 2.1. Dynamic perfect hashing uses a two-tiered hash table. The first tier breaks the input into smaller bins, while the second level consists of collision-free hash tables of each bucket.

The spatial hash table construction used by Lefebvre and Hoppe [23] was based on one of these approaches [13]. Problematically, such constructions are not only expensive but also seem to be inherently sequential: the hash functions are built in a way that the location of an item depends on the locations already taken by earlier items.

Multiple-choice hashing alleviates some of the balancing issues that could occur with chaining, where one bucket may have a large number of items but another may be empty. Azar et al. [6] considered the usual chaining construction and showed that using $H \geqslant 2$ hash functions and storing an item into the bucket containing the smallest number of items reduces the expected size of the longest list from $O(\frac{\log N}{\log\log N})$ to $O(\frac{\log\log N}{\log H})$. Vöcking [41] extended this work, using a hash table split into H equally-sized subtables, each consisting of multiple buckets and associated with its own hash function. Items may hash into one bucket in each subtable and are inserted into the least loaded of the buckets available to them, breaking ties to the left. This combination is shown to be effective at balancing out the bucket loads, reducing the expected size of the longest list to $O(\frac{\log\log N}{H})$.

One problem with this method is that all H lists have to be traversed to find an item, which exacerbates the problem of trying to find keys that are not stored in the table. Another problem is that it is not clear how to parallelize the choice of bucket. By the time that a thread determines which of the H buckets is least full, another thread may have inserted into the bucket and skewed the distribution.

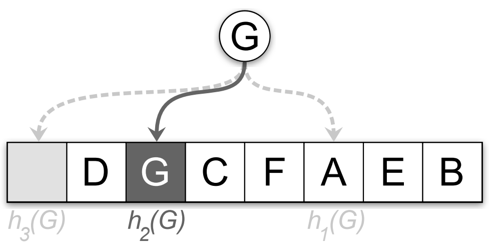


Figure 2.2. Cuckoo hashing uses H hash functions $h_i(k)$ to give keys a restricted set of insertion locations; here, each key may be placed in one of three locations. It finds a conflict-free configuration where each key is assigned its own slot and guarantees that queries can be answered within H probes, regardless of how packed the table is.

Adler et al. [1] describe a method where threads managing items communicate in parallel with threads managing the hash table slots, but this type of coordination would require inefficient repeated global synchronization between kernel launches.

Cuckoo hashing is a variation of open addressing that limits the number of slots an item can fall into, preventing situations where the majority of a tightly packed hash table must be probed in order to answer a query. It uses multiple hash functions to assign each item a small, random set of slot choices for insertion [9, 34]. To handle collisions, items are moved around after their initial placement to accommodate new items. The number of hash functions is a small, fixed number (typically set to three or four), guaranteeing that any query can be answered after checking a constant number of slots (see Figure 2.2). The cost of this guarantee and higher table occupancy is a slower and more difficult construction algorithm.

From a GPU perspective, this guarantee is excellent because it restricts the number of uncoalesced memory accesses required to answer any query, regardless of how packed the table is. However, there are two main problems with performing cuckoo hashing. First, items are moved throughout the table, meaning that each iteration incurs many uncoalesced global memory accesses. Secondly, cuckoo hashing can fail during construction and require rebuilding the entire table from scratch with new hash functions.

We examine two methods for parallelizing this method, which attack these problems in different ways. Chapter 5 first examines a two-level approach that uses multiple smaller cuckoo hash tables, while Chapter 6 looks at ways to reduce the chances of failure for a straightforward parallel cuckoo hashing implementation.

2.3 Experimental setup

2.3.1 Performance metrics

Given the wide variety of hashing algorithms available, we examine how they perform based on three criteria:

- **Construction time** measures how long it takes to insert all of the input items in parallel. For tables that are meant to be built, queried, and immediately discarded, a fast construction speed would be ideal. Tables that are meant to be built once and queried repeatedly might instead spend extra time in the construction phase to speed up queries.
- **Retrieval efficiency** determines how quickly a query can be answered. We are interested both in the average number of probes required to find an item and the cost of handling harder to answer queries because these numbers can be drastically different. As memory accesses can be costly, the number of probes required to find an item should be kept to a minimum.
- **Memory usage** is a measure of how much memory is occupied by the hash table. Applications using multiple data structures might emphasize more compact hash tables because GPU memory is relatively limited. On the other hand, using larger hash tables might be acceptable because they usually provide a speed boost for both construction and retrieval. We focus on the memory usage rather than the traditional load factor as this value is a more useful metric for determining how much memory is required for a parallel application.

Hash table designs that emphasize one metric will often require a trade-off for one of the others. We will discuss when using one hash table over another makes more sense for different situations.

Note that in sequential computing, one big advantage of the hash table over other structures is that it is an inherently dynamic data structure; deletions and insertions have $O(1)$ expected time. However, it is more complicated on the GPU because thousands of threads could be modifying the structure simultaneously. If any one of these modifications fails, e.g. an attempt to insert into a full table, the execution pipeline would likely have to be interrupted and the hash table rebuilt. We focus on handling these cases with fast constructions, which implicitly handles modifications. This is a heavyweight approach, but one congruent with a parallel processor with massive, structured compute capability.

2.3.2 Testing procedure

To analyze the performance of the hash tables, we measure the time taken to build and query it under different settings. We present our results as rates to show trends in performance; they can be easily converted back to actual times. Higher rates indicate better efficiency because the data structure can process the input more quickly. Reported rates are averaged over multiple runs to reduce the noise arising from the randomness of kernel execution and some of the algorithms, themselves. Before each run, we shuffled all of the input and query keys to simulate random access. When possible, we do not include the time taken to allocate the memory, as we consider it a pre-processing step.

2.3.3 System configuration

We test all results using CUDA 3.2 with an NVIDIA GTX 280 and an NVIDIA GTX 470; the exact hardware specs are listed in Table 2.1. We used driver version 260.24 under 64-bit Ubuntu Linux. Aside from the general performance improvements inherent in newer cards, these two GPUs are separated by a significant leap in hardware architecture, with some of the most important changes including a cached memory hierarchy and significantly faster atomic instructions. This causes some hash tables that perform poorly on older cards to drastically improve on later generations of cards.

	GTX 280	**GTX 470**
Manufacturer	EVGA	EVGA
Model	SSC	SuperClocked
Cores	240	448
Core clock speed	648 MHz	625 MHz
Shader clock speed	1404 MHz	1250 MHz
Memory clock speed	2322 MHz (effective)	3402 MHz (effective)
Memory interface	512-bit	320-bit
Memory bandwidth	148.6 GB/sec	136 GB/sec
Device memory cache	No	Yes
Compute capability	1.3	2.0

Table 2.1. Technical specs for our GPUs pulled from the manufacturer's website.

Chapter 3

Open addressing

In this chapter, we explore a parallelized implementation of open addressing hash tables. These hash tables consist of $S_T \geq N$ slots, where each slot of the table can store one of the N items from the input. In order to insert an item with key k, a series of probes is performed to find an empty slot, beginning with the slot $h(k)$; the item is immediately inserted into the earliest possible empty slot in the series (Figure 3.1).

These hash tables are commonly used in sequential computing, but they face some issues in highly parallel GPU environments. Serial implementations, for example, have no chance of losing any items during insertion: nothing can be inserted into the slot between checking the slot and actually writing the item in. However, race conditions like this exist in parallel implementations because multiple threads may be attempting to insert items into the same locations simultaneously.

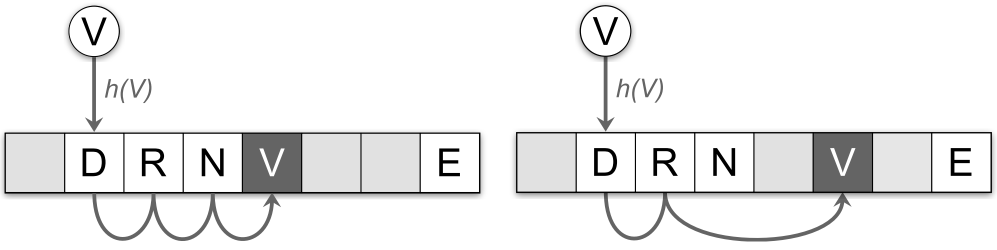


Figure 3.1. Examples of linear probing (left) and quadratic probing (right).

We give an overview of a parallel construction in Section 3.1, then describe the implementation in Section 3.2. We then present and analyze results in Section 3.3 and discuss limitations in Section 3.4.

3.1 Overview

Our parallel construction method assigns each input item to a different thread, then has each thread simultaneously probe the hash table for empty slots. Threads are prevented from overwriting other keys being inserted by forcing serialization of accesses to the table. We do so using atomic check-and-set operations, which check if any given slot is empty and immediately inserts the item if it is.

Parallel retrievals don't encounter the same issues since no changes are made to the structure. To perform a query, a thread simply marches along the table using the query key's probe sequence and stops when either the key is found or an empty slot is discovered; the latter case indicates that the query key would have been inserted in that location had the key existed in the input.

3.1.1 Parameters

There are three main parameters that can be used to adjust the implementation's performance: the number of slots in the table, the probe sequence, and the maximum allowed number of probes in a sequence.

The number of slots in the hash table, denoted S_T, must be greater than or equal to the number of items in the input. This parameter is the main factor in determining the efficiency of both the construction and retrieval rates.

Allocating more slots makes it much easier to find an empty slot in the table, which results in shorter probe sequences. As shorter probe sequences result in fewer memory accesses, this greatly reduces the average number of uncoalesced memory accesses that are performed. Using fewer slots instead allows the table to fit in a much smaller amount of memory, which can be important for space-constrained applications. In our experiments, we found that setting $S_T \approx 1.25N$ struck a good balance between the construction and retrieval time trade-offs.

Probing scheme	**Hash function**
Linear probing	$h(k) = g(k) + iteration$
Quadratic probing	$h(k) = g(k) + c_0 \cdot iteration + c_1 \cdot iteration^2$
Double hashing	$h(k) = g(k) + jump(k) \cdot iteration$

Table 3.1. Open addressing hashing schemes

The probe sequence determines the order in which the slots are examined when trying to either insert or find an item; the three common ones are shown in Table 3.1.

- *Linear probing* advances along the slots one at a time. While linear probing hash tables can take advantage of caches because neighboring slots are visited, the problem is that items that can cluster around a particular slot. This can cause extremely long probe sequences until an empty slot is found, which is exacerbated when the hash table becomes more and more full. However, it is guaranteed to visit every slot in the table.

- *Quadratic probing* mitigates the clustering issue by using a sequence generated by a quadratic function, which takes longer and longer leaps after each probe. Typically, $c_0 = 0$ and $c_1 = 1$ for the hash function. There is still an issue when a large number of items hash into the same slot, however: since all of these items use the exact same probe sequence, they will repeatedly collide.

- *Double hashing* generates probe sequences that are tailored for each key. To do this, it uses a second hash function to determine the jump taken on collision, making it unlikely for items colliding to be taking the same sequence of jumps.

Both quadratic probing and double hashing are able to find empty slots much more easily than linear probing, as they can jump out of tight clusters in the hash table more effectively. While linear probing may be friendlier for a cache, its benefit is generally outweighed by the longer probe sequences it must use to find an empty slot. Quadratic probing is also able to make some use of the cache because of its relatively small jump size, but the ability of double hashing to do so is dependent on how the second hash function is set.

The maximum allowed length of a probe sequence is useful for estimating when we have hit an endless cycle. Without tailoring the hash functions for every given situation, it is possible to encounter a probe sequence that will either never find an empty slot or take too long, triggering the kernel watchdog and stopping the construction process prematurely. Detecting these situations correctly would require that a thread keeps track of which slots it has accessed, but that would be very expensive. Instead, we limit the number of probes that a thread may use in order to find an empty slot, then declare failure if the limit is hit and restart the construction with a new hash function. We found that setting it to the smaller of N or 10,000 iterations worked well.

3.1.2 Hash functions

In addition to the implementation's parameters, the hash function is another big factor in the hash table's performance. Typical requirements for the hash function include distributing the items evenly through the table, lowering the average number of probes required to find any given item, and being fast to compute.

The ideal hash function would assign each item to its own location in the table with no collisions, meaning that queries could be answered after checking a single entry. However, these *perfect hash functions* are usually difficult or impractical to construct. The work done by Fredman et al [14] shows that using N^2 slots for N items makes it highly probable that any hash function chosen results in a collision-free function, but the space requirements are unmanageable for even small input sizes. Lefebvre and Hoppe [23] showed that it's possible to generate collision-free hash functions using nearly N space by combining two imperfect hash functions, but the amount of work required to produce these hash functions is a serial process that has no obvious method of parallelization. In general, however, the benefits of a perfect hash function are minimal because hash tables can be constructed in a way that effectively limits the number of probes required to find an item to just one or two.

Complex hash functions like MurmurHash [3] are quite effective at creating fair distributions of keys, ensuring that changing even one bit of the key causes an avalanche of changes to the bits in the hash function's value. However, this wasn't necessary for many of the datasets we tried. We instead rely on the observation of Mitzenmacher and Vadhan [30] that simple randomized hash functions work well in practice. This stems from the fact that there is an inherent randomness in the data that balances out the weakness of the hash function. For most of our datasets, we achieved fast construction and retrieval times using randomly generated hash functions of the form:

$$g(k) = (f(a,k) + b) \bmod p \bmod S_T$$

Here, a and b are randomly generated constants, p is a prime number, and S_T is the number of slots available in the hash table. Note that it is possible to generate new hash functions by simply changing the constants a and b: this is important since it is possible for the hash table construction to fail, especially for extremely packed tables.

When $f(a,k) = a \cdot k$, we have a basic linear polynomial hash function. Under certain circumstances this forms a *2-universal family* of hash functions, which has the mathematical property that the probability of any two items colliding is $\leqslant \frac{1}{S_T}$. We can set $p = 4,294,967,291$ (the largest unsigned 32-bit prime number) and randomly generate[1] constants $a \in [1, p-1]$ and $b \in [0, p-1]$, but to be an actual 2-universal family, every key k in the input would need to satisfy $k < p$. Care must be taken to ensure that 64-bit intermediate values are calculated: we found that

[1]It is highly important to use a good random number generator to produce good hash functions. We used the Mersenne Twister created by Matsumoto and Nishimura [28], available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`.

the hash function could behave poorly otherwise. In practice, hash functions from this family worked well and were able to limit the number of slots under heavy contention in most cases.

For some of our datasets, we found that setting $f(a, k) = a\ XOR\ k$ gave a fair distribution even when using only 32-bit intermediate values. This saves some time for hash table constructions that need to repeatedly recalculate the hash function's value, and highlights the fact that some time should be taken to tailor the function family specifically for patterns in the input data. For example, hash tables storing spatial data could benefit from a hash function that maps spatially-local input items to nearby slots.

3.2 Implementation

Parallel insertion is performed using Algorithm 3.1. We begin by allocating an array, *table*[ ], of size S_T to store the hash table. Every slot stores a 64-bit integer, which holds a 32-bit key and its value in the same entry. This allows both the key and value to be inserted or pulled simultaneously, using fewer 64-bit memory accesses instead of more 32-bit memory accesses. The alternative is to use two separate 32-bit arrays, but the performance is highly dependent on the GPU.

We repeatedly loop until the structure is built, but in practice only one attempt was required. At the beginning of every attempt, the table is filled with $\varnothing$, a special value that indicates that the slot is empty; we specifically set $\varnothing$ to be the key-value pair (`0xffffffff`, `0`). We follow by generating a randomized linear polynomial hash function using the method described in Section 3.1.2. We then build the table by inserting all items into it simultaneously, with a different thread assigned to each item.

Code for the insertion is shown in Listing 3.1; actual implementations should remove code irrelevant to the chosen open addressing method. This requires 64-bit global memory atomic operations that are available only on GPUs with a compute capability of at least 1.2; the algorithm can be easily adjusted to work with 32-bit

Algorithm 3.1 Process for creating an open addressing hash table.

```
1: allocate enough memory for $table[\,]$, which will contain $S_T$ 64-bit slots
2: repeat
3:   fill each slot with $\varnothing$
4:   generate a new hash function for the current attempt
5:   for all key-value pairs $(k, v)$ in the input do
6:     repeat
7:       atomically check-and-set $table[location]$
8:       advance $location$ to next location in probe sequence
9:     until $\varnothing$ is found or max probes hit
10:  end for
11: until hash table is built
```

atomic operations instead, with extra steps for moving the values into the right location after their corresponding keys have been inserted.

Threads complete their work when they successfully place their item, but a thread block will not complete until all of its threads are done. Thus we choose a relatively small thread block size – 64 threads – that minimizes the number of threads within a block kept alive by a single thread's long probe sequence.

Double hashing requires computing a second hash function, $jump(k)$, whose value is stored to prevent having to recompute it after each collision. Care must be taken so that the jump distance is greater than 0 to prevent an item from repeatedly failing to insert itself into the same occupied slot. We use the simple $jump(k) = 1 + (k \bmod jump\_prime)$, where $jump\_prime = 41$. Lowering this prime forces the double hashing scheme to take smaller jumps, which speeds up retrievals since all locations are more likely to be cached, while using larger primes decreases the construction time instead, since the scheme can jump out of crowded areas more quickly. The function worked well for larger hash tables, but it did fail many times for hash tables containing less than 5K elements, which requires smaller jumps.

Listing 3.1. Parallel insertion of items into an open addressing table.

```
__device__ bool insert_entry(const unsigned  key,
                             const unsigned  value,
                             const unsigned  table_size,
                                   Entry    *table) {
  // Manage the key and its value as a single 64-bit entry.
  Entry entry = ((Entry) key << 32) + value;

  // Figure out where the item needs to be hashed into.
  unsigned index            = hash_function(key);
  unsigned double_hash_jump = jump_function(key) + 1;

  // Keep trying to insert the entry into the hash table
  // until an empty slot is found.
  Entry old_entry;
  for (unsigned attempt = 1; attempt <= kMaxProbes; ++attempt) {
    // Move the index so that it points somewhere within the table.
    index %= table_size;

    // Atomically check the slot and insert the key if empty.
    old_entry = atomicCAS(table + index, SLOT_EMPTY, entry);

    // If the slot was empty, the item was inserted safely.
    if (old_entry == SLOT_EMPTY) return true;

    // Move the insertion index.
         if (method == LINEAR)    index += 1;
    else if (method == QUADRATIC) index += attempt * attempt;
    else                          index += attempt * double_hash_jump;
  }

  return false;
}
```

The thread begins by concatenating the key and its value together, then determining where in the hash table the item should be inserted. A **for** loop is used to probe the table up to $kMaxProbes$ times for an empty slot for insertion. On each attempt, an atomic check-and-set (`atomicCAS()`) inserts the entry into the table, but only if the slot contains $\varnothing$. Because the operation is atomic, other threads attempting to insert into the same slot will fail and continue probing.

Inserting into the table after the table's initial construction is possible by following the same procedure. However, the cost of packing more items into the table increases dramatically once the table has reached a certain load factor. Moreover, a failed insertion would require rebuilding the whole table from scratch.

Parallel retrieval essentially follows the same search process as the construction. Query keys are distributed among the threads performing the retrieval, using the exact same sequence of probes that would have been taken if the query key were being inserted. The probes stop immediately after finding the query key in the table, which returns its value, or either finding an empty slot in the table or hitting the maximum probe sequence length, both of which indicate that the probe would have been found by that point.

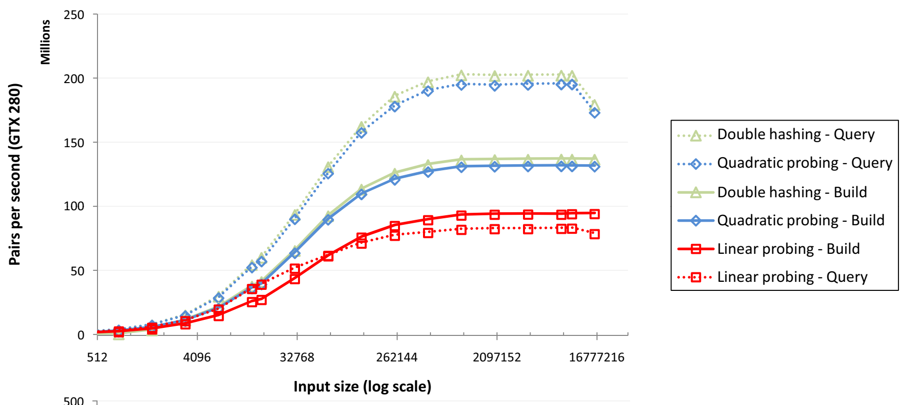
3.3 Performance analysis

We present results using the setup described in Section 2.3.2.

3.3.1 Comparisons between probing methods

Figure 3.2 shows the effect of the input size on both the insertion and retrieval rates for each of the different probing methods. We assume a fixed table size of $1.25N$, producing hash tables with a load of 80%. Our data consisted of randomly generated 32-bit keys paired with 32-bit values, where all of the input keys were unique. We also tested the performance on increasingly finer voxelizations of the Lucy dataset, but the results were similar to the random data case and are omitted.

Construction rates were computed using the time taken to insert all of the input items into the table in parallel. After an initial ramping up period, the construction



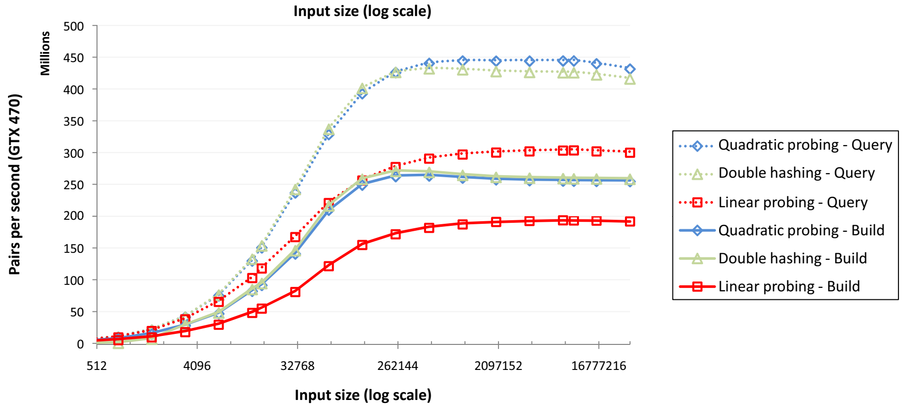


Figure 3.2. Effect of input size on construction retrieval rates for tables containing $1.25N$ slots on both the GTX 280 (top) and 470 (bottom).

rates using all three probing methods becomes more or less flat, meaning that the time required to construct the table increases linearly with the input size. Performance for all three methods on the GTX 470 is roughly double that of the GTX 280; one likely cause is the speed boost atomic operations received on Fermi cards. Linear probing does consistently worse than both quadratic probing and double hashing, reflecting the problems linear probing encounters when trying to escape crowded areas of the table. Double hashing has a slight edge over quadratic probing because it can jump away more readily.

Retrieval rates measure how quickly the hash table can be queried for all of the input items in parallel, with each query assigned to a different thread. We see similar trends here, which is expected because retrievals mimic the insertion process without any slow atomic operations. All three methods get a performance boost, with quadratic probing and double hashing getting a more than 2x boost on the GTX 470. Although linear probing still lags behind the other methods,

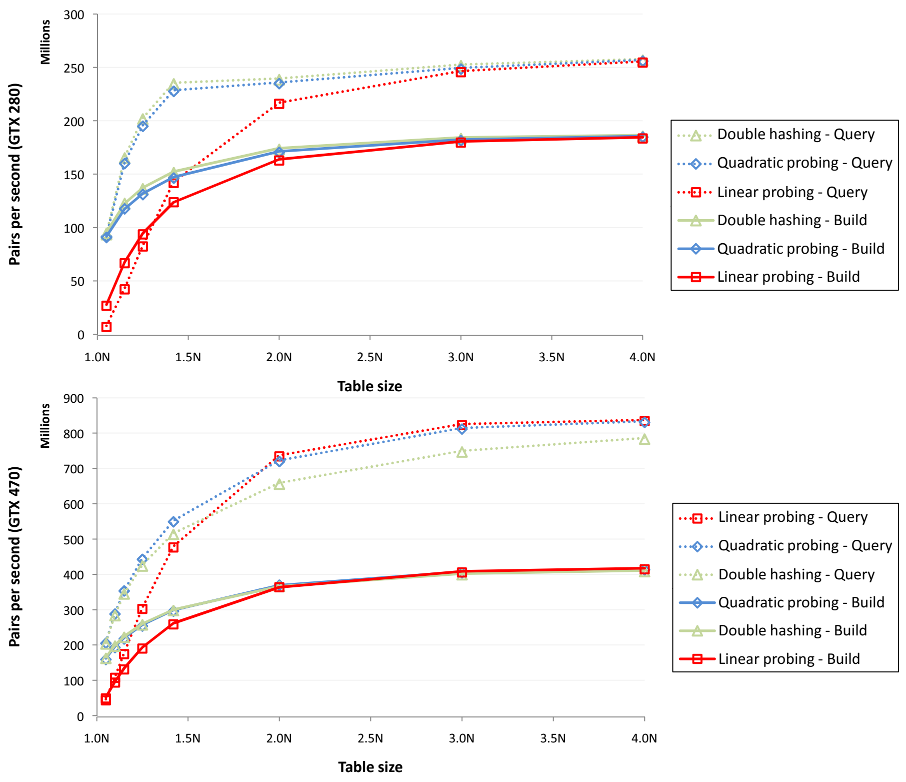


Figure 3.3. Effect of the table size on construction and retrieval rates for tables containing 10 million items.

it gets an even bigger 4x performance boost. The sharp decline in retrieval rates for the GTX 280 for larger input sizes does not appear when using two separate 32-bit arrays to store the keys and values, but storing the data this way only hurt performance on the GTX 470.

Quadratic probing has an obvious advantage over double hashing on the GTX 470, which is a direct result of the jump function chosen for double hashing. As mentioned earlier, allowing larger jumps decreases construction times because the average number of probes required to insert an item decreases. Conversely, decreasing the jump size allows taking advantage of the cache, speeding up the retrievals. Our GTX 280 results corroborate this: double hashing consistently performed slightly better than quadratic probing even for larger jumps, suggesting that the cache is able to reduce the memory traffic slightly.

Figure 3.3 shows the effect of modifying the table size while keeping the input size fixed at 10 million items, effectively changing the load of the hash table and

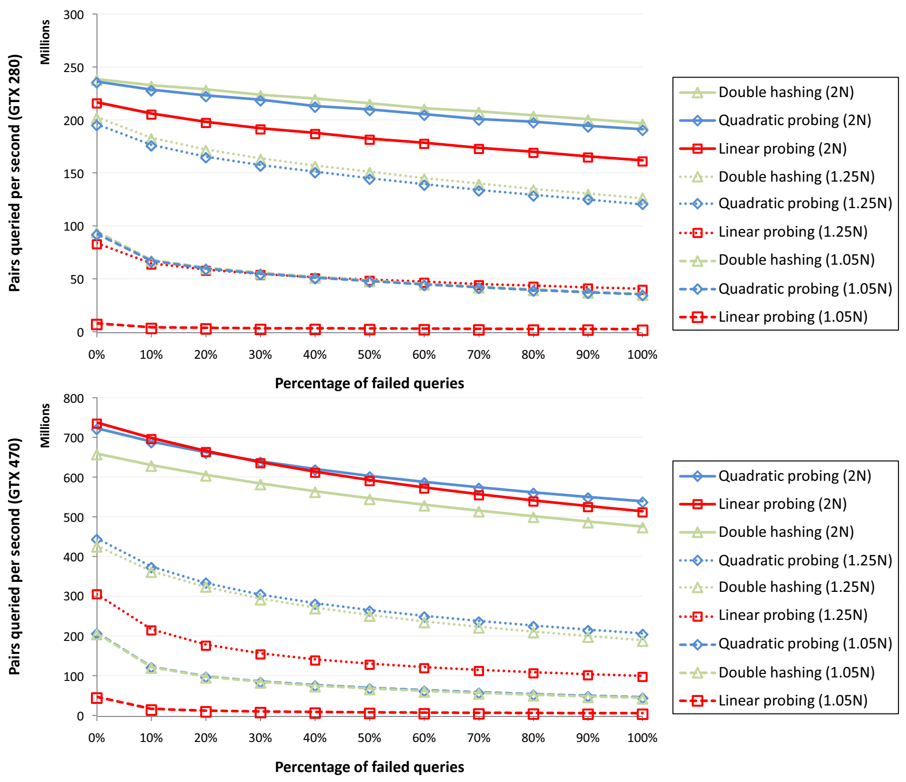


Figure 3.4. Effect of querying hash tables containing 10M items with keys that were not part of the original input items. Each table was queried for 10M unique keys with different mixtures of keys that were and were not part of the original input: having 0% "failed queries" indicates that the original input items were queried from the table, while 100% "failed queries" indicates that none of the input items were queried.

the number of empty slots available. All rates begin to degrade significantly below $S_T = 2N$, drastically dropping as the table size approaches $S_T = N$. The effect is most significant for linear probing, which is burdened by the sheer number of slots it checks. When $S_T \geqslant 2N$, the construction rates for all three methods are similar, reflecting the ease of finding empty slots. On the GTX 280, retrievals for all three methods seem capped at 250M pairs per second for the largest table sizes, with linear probing constantly lagging behind. For these cases, the table is very sparse and can answer queries after just one or two probes. Surprisingly, linear probing outperforms the other two methods on the GTX 470. Because the cache pulls in a line of entries, it is highly likely that all of the locations that a thread needs get cached and can be checked quickly.

Figure 3.4 shows the result of failing to successfully find a query key for hash tables of size $S_T = 1.05N, 1.25N$, and $2N$. We fixed both the input size and query size at $N = 10,000,000$, but replaced random keys in the query set with keys that were not part of the original input; we call any query that fails a "bad" query. The percentage of replaced query keys varied per trial.

In general, threads handling bad queries have fewer chances to exit out compared to threads capable of finding their keys: to terminate, they must either find an empty slot or hit the maximum probe sequence length. This is represented by the downward slope in the rates as larger percentages of the queries fail, though the effect is less pronounced when using larger table sizes because more empty slots exist.

On the GTX 280, it is always advisable to use double hashing because its larger jumps result in faster query termination, regardless of the number of bad queries. Linear probing consistently lags behind: when $S_T = 1.05N$, bad queries hardly have an effect because a good portion of the table is already being probed to answer regular queries.

On the GTX 470, the cache gives quadratic probing an edge over double hashing; even linear probing does better than double hashing when $S_T = 2.0N$. Another difference is that bad queries have a stronger effect, causing a significant drop in performance even when only 10% of the queries fail. This is because threads in the same block still have to wait for the slowest among them to finish, even though the majority of queries finish quickly.

3.3.2 Comparisons with radix sort and binary search

We also compared quadratic probing hash tables against using a sorted array of the input items, which requires using a binary search to perform a query. Sorted arrays can be constructed extremely quickly using GPU implementations of the radix sorting algorithm; we used Duane Merrill's radix sort [29], the fastest GPU implementation available at the time of writing. Figure 3.5 shows how quickly it performs relative to quadratic probing: although quadratic probing has an edge

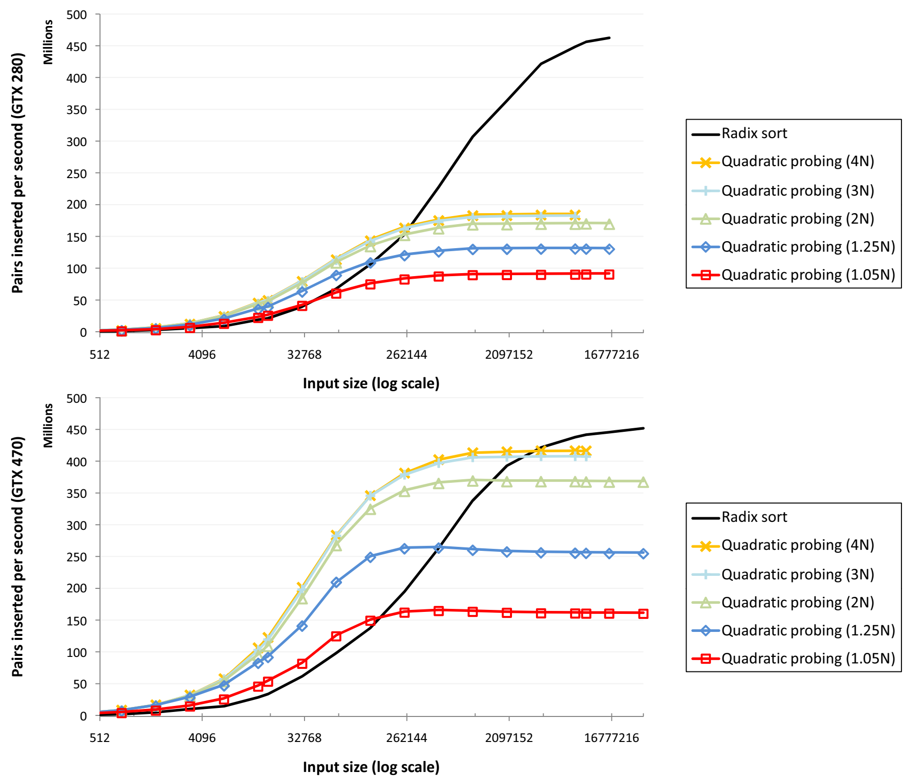
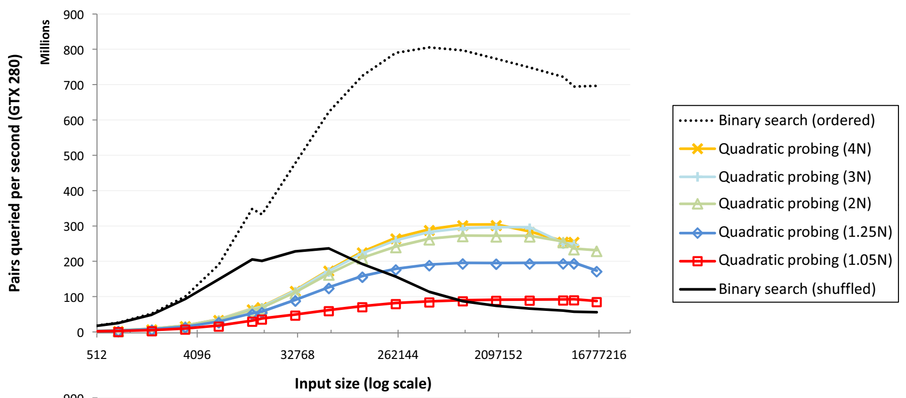


Figure 3.5. Effect of input size on the construction of a radix-sorted array and quadratic probing hash tables with differing numbers of slots.

for smaller datasets, radix sort is a great fit for the GPU and achieves rates that hashing cannot.

Figure 3.6 shows the accompanying query rate graphs, which compare binary searches to retrievals from the hash tables. Binary searches require $O(\lg N)$ memory accesses to find an item, but the time taken is highly dependent on how well ordered the queries are. For example, for a sorted array containing 10 million items, querying it with the sorted input items takes about 17 milliseconds on the GTX 470. However, shuffling the queries completely before performing them, representing random access, takes about 140 milliseconds. We provide rates for both of these cases as guidelines for comparison, thought they do not represent the absolute performance extremes[2].

[2]Unless explicitly stated, whenever we talk about a “binary search” without any information about the query order, we are referring to the binary search with shuffled queries; it provides a more accurate comparison for random-access performance.



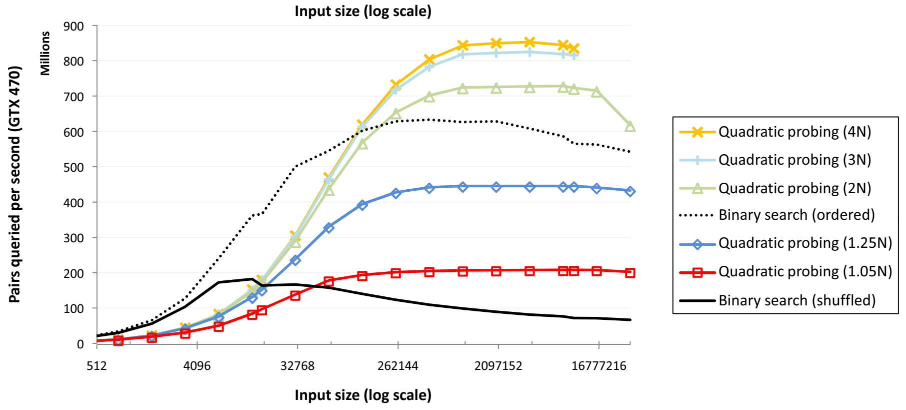


Figure 3.6. Effect of input size on retrieval rates from a sorted array and quadratic probing hash tables with differing numbers of slots. Two rates are included for the binary search, which test retrieval rates when the queries are completely shuffled and completely sorted; the former case represents the random access case that we are interested in, though the latter provides some interesting context.

In the ordered query case, binary search generally outperforms hashing because the binary search's memory accesses repeatedly coalesce: any divergence in memory accesses will be near the end of the query, but even in this case warps will be likely to read from the same segment of memory. Interestingly, the GTX 470 performs worse than the GTX 280 in this case. We believe this is related to the slightly better hardware specs of our specific GTX 280; the binary search derives little benefit from the cache available on the 470 because it repeatedly makes large hops through memory – the cache could even hurt performance because memory transactions are enlarged to fill the cache lines[3]. When the queries are shuffled, the results are much worse because warps must find items located in distant parts of

[3]The cache *can* be partially disabled, but it would likely degrade performance for the rest of the application.

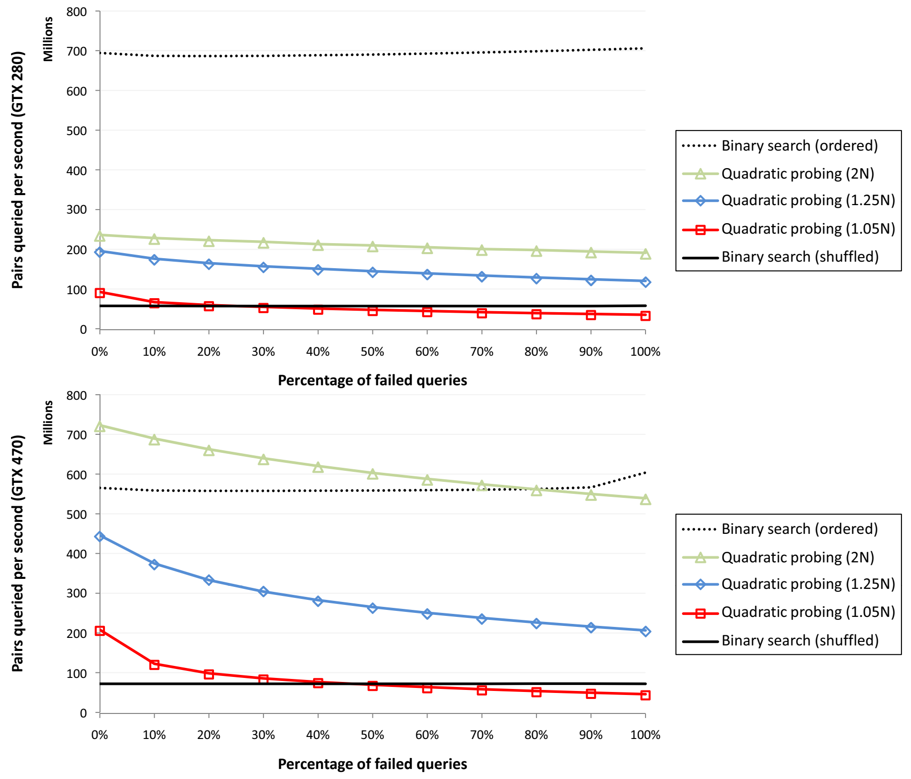


Figure 3.7. Effect of failed queries on the query rates of binary search and three quadratic probing hash tables with differing numbers of slots. The structures all contained 10M items. Two rates are included for the binary search, which test retrieval rates when the queries are completely shuffled and completely sorted.

the array. This results in a high number of uncoalesced memory accesses without spatial locality.

On the GTX 280, quadratic probing can perform better than the binary search with shuffled queries because the number of probes required to find items is smaller than the $O(\lg N)$ probes needed by the binary search. However, all of the quadratic probing hash tables perform drastically worse than the binary search with sorted queries, even when the average number of probes required to find an item is low. In contrast, the performance of the hash tables is dramatically improved for the GTX 470 – the quadratic probing hash tables with $S_T \geqslant 2.0N$ have higher retrieval rates than even the binary search with sorted queries.

We also tested how failed queries affect retrieval rates, shown in Figure 3.7; the queries were constructed in the same way they were constructed for Figure 3.4. The binary search we used does not provide an early-out for queries that are smaller than the first array element or bigger than the final array element; the early-out degrades performance when all queries can be answered, but only provides a marginal benefit when the queries cannot. Nonetheless, binary search performance actually increases with higher percentages of failed queries.

All of the hash tables perform worse as the number of bad queries increases, showing the difficulty in terminating these queries – the query rates when $S_T = 1.05N$ even dip below the shuffled binary searches because so much of the hash table has to be probed before termination.

3.3.3 Probe sequence lengths

The retrieval rates we have presented in this section have all been for retrieving N from a hash table containing N items, effectively averaging the cost of answering all of the queries. The problem is that some queries are much, much more expensive to answer than other queries; query inputs consisting mainly of these harder to find keys will have very poor performance. Because threads in the same block have to wait for the longest query among them, driving down the number of probes is key to achieving better performance in both construction and retrieval.

To analyze how skewed the distributions are, we took statistics on how many probes were required by each thread to find their query in the table, assuming that either all or none of the queries could be found. Table 3.2 shows statistics taken for hash tables containing 10 million items with different table sizes and different probing schemes. It gives the average number of probes required to find an item, as well as the number of probes required to find 80%, 90%, 99%, and 100% of the items in the table; these milestones show how quickly the distribution tapers off. A single, representative trial is used for each configuration.

When all of the queries could be found, the median number of probes required to find an item was just one, reflecting the ease with which many of the threads could find an empty slot during construction. The average number of probes grew significantly higher as the table became more compact and the table had higher

Queries failed	Probing scheme	Slots	AVG	50%	80%	90%	99%	100%
None	Linear	$1.05N$	11.06	1	6	16	195	5144
		$1.25N$	2.99	1	3	6	29	301
		$2.00N$	1.50	1	2	3	7	43
	Quadratic	$1.05N$	3.39	1	4	8	29	264
		$1.25N$	2.13	1	3	4	12	80
		$2.00N$	1.43	1	2	2	5	22
	Dbl. hashing	$1.05N$	3.23	1	4	7	28	237
		$1.25N$	2.02	1	3	4	11	54
		$2.00N$	1.39	1	2	2	5	20
All	Linear	$1.05N$	223.40	77	340	622	1820	5650
		$1.25N$	12.97	6	19	34	95	354
		$2.00N$	2.49	1	3	5	13	58
	Quadratic	$1.05N$	21.85	15	35	50	98	318
		$1.25N$	5.43	4	9	12	23	78
		$2.00N$	2.13	1	3	4	8	26
	Dbl. hashing	$1.05N$	21.42	15	34	49	97	359
		$1.25N$	5.06	4	8	11	22	81
		$2.00N$	2.01	1	3	4	7	27

Table 3.2. Statistics for a single run of querying the hash table with different space constraints and probing methods. The two extremes are shown, where either none of the queries fail (top) or all of them fail (bottom). Numbers represent the average number of probes required to find an item (AVG), and the number of probes required to find 50%, 80%, 90%, 99%, and 100% of the items the table.

loads. Unsurprisingly, linear probing had the hardest time, regardless of the table size. For the cases where $S_T < 2N$, linear probing correspondingly has the worst retrieval performance in Figure 3.3. We can see why by looking at the probe distribution: it has a much heavier and larger tail than the other two schemes, requiring many more probes.

Double hashing has a consistently lower average and better distribution than the other cases, but it actually had worse retrievals than the other two methods on the GTX 470 for the largest table size we tested. Since the number of probes required to insert a key for the other two methods is small, it is highly likely that all of the necessary probes were pulled into the cache.

When no query keys are found in the table, threads have to iterate until they find an empty slot in the table. Similar trends are seen between the three probing schemes, but the average is considerably higher and the probe distribution is skewed much more heavily toward requiring many more probes to terminate. This results in the drastic performance drops in Figure 3.4.

3.4 Limitations

Under the right conditions, open addressing hash tables can be viable data structures for random access on the GPU. However, they suffer from many limitations.

Performance drops significantly for compact tables. Although these hash tables perform well when enough empty slots are available, both the time taken to build and query the table increase significantly once the table size shrinks past $1.5N$, or 67% load. This can be problematic for applications that require a compact hash table.

High variability in probe sequence lengths means that an open addressing hash table can find *most* items with a small number of probes, while requiring a significantly higher number of probes to find the other items. The number of these items increases for more compact tables, which have higher loads. The problem here is twofold. First, the effort required to query the items for these harder-to-find items can be higher than simply doing a binary search through a sorted array, which is guaranteed to terminate after $O(\lg N)$ steps. Second, parallel queries consisting mainly of these items will suffer from a performance penalty since threads have to wait for the slowest thread in their block.

Removing items from the table is not straightforward. It is not sufficient to simply fill the removed item's slot with $\varnothing$ because it would break queries for any items that would fall into that slot in the table. One option would be to fill in the slots of deleted entries with "tombstone" markers, indicating that an item was deleted and that queries should continue past them. However, removal times would be unaffected since the slots are still visited.

3.5 Summary

When in need of a data structure that provides random access to its contents, quadratic probing and double hashing hash tables generally perform well when the size of the table is much larger than the number of input items (at least twice as large). For these cases, the number of probes required to find an item is low since the table occupancy is low. However, performance of the hash tables decreases significantly for smaller hash tables as the probe sequence lengths become arbitrarily long.

Linear probing is a wildcard whose performance depends entirely on the table size and the GPU architecture. When a cache is available, it makes a lot of sense to use it because it is more efficient than the other two methods. Otherwise, using it is just not practical.

In Chapters 5 and 6, we discuss a different open addressing method called *cuckoo hashing*. It ensures that queries can be answered in a small, constant number of probes, regardless of the table's size. This results in fast retrievals for very tightly packed tables, as well as limiting the effect of failed queries. The trade-off, however, is a longer and more complicated construction process.

Chapter 4

Chaining

Chaining, unlike open addressing, treats each slot of the hash table as a bucket that can store multiple items. Any items falling within the bucket are inserted into a linked list associated with that specific bucket (Figure 4.1). To retrieve an item from the table, the bucket potentially containing the query key is computed and its list traversed. Using good hash functions is important because retrievals are $O(N)$ in the worst case, when all items hash into the same bucket. This problem is mitigated by using more buckets for the structure, reducing the average amount of items per bucket.

Problematically, chaining hash tables using linked lists are terribly inefficient on the GPU. There are several reasons why:

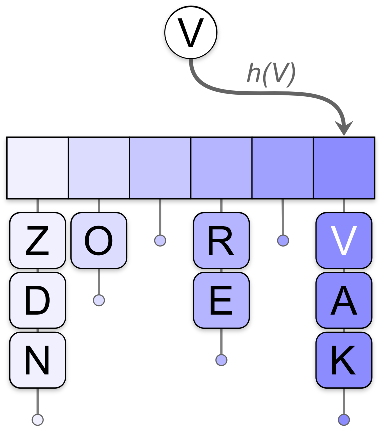


Figure 4.1. Chaining attaches a list to each hash table slot. All items hashing into the same slot are inserted into its list.

- Linked lists store two things with every node: the item itself, and a pointer to the next node in the list. The memory available on a GPU is relatively limited, so having to store a pointer within every node of the list is expensive.
- To find an item, the list must be traversed from the beginning, following pointers until either the item or the end of the list is reached. This entails hopping around memory, giving caching hierarchies little opportunity to improve performance.
- Threads simultaneously inserting into the same linked list have to serialize their actions to prevent clobbering each other's writes.

By sacrificing the ability to modify the structure after the initial construction of the hash table, we can avoid both the storage and performance overhead of pointer traversal associated with linked lists. In this chapter, we present an algorithm based on a radix sort, which can be performed very quickly. Our algorithm is described in Section 4.1, while the implementation is presented in Section 4.2. Results are presented in Section 4.3, while limitations are presented in Section 4.4.

4.1 Overview

Our chaining implementation avoids linked lists by storing all the items falling within a bucket into contiguous locations of an array. The n items of each bucket are stored such that its items start at index k and end at index $k + n - 1$. The array is segmented to that each bucket's items are sequentially stored end to end (Figure 4.2). Storing the data in this manner means that we only need to record the indices of each bucket's first item in the array; the number of items within each bucket can be computed by taking the difference between these indices. The goal of our construction algorithm is to produce two arrays: *bucket_contents*[ ], containing all of the items rearranged into their buckets, and *bucket_starts*[ ], which records where each bucket begins in *bucket_contents*[ ].

The performance of our data structure is mainly dependent on one parameter: the number of buckets in the hash table, B. Increasing this number decreases the

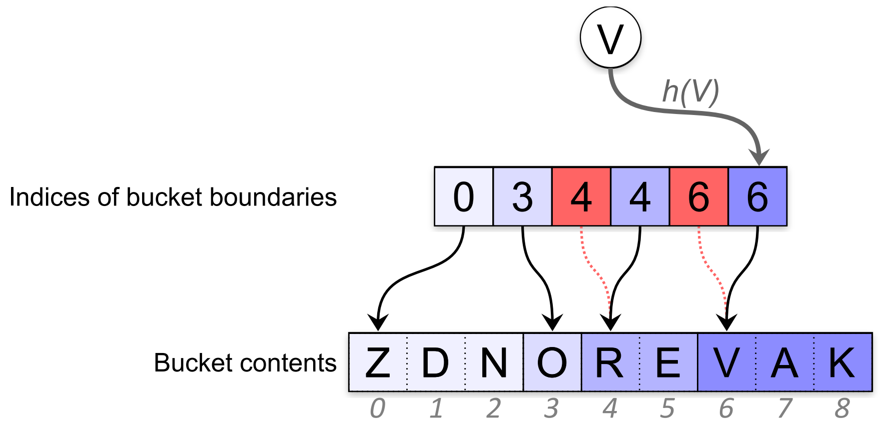


Figure 4.2. To avoid pointer traversal, the lists from Figure 4.1 can be stored as a single array with each bucket's items stored contiguously. The array index to each bucket's first item is recorded; bucket sizes can be computed by taking the difference of these indices. Empty buckets (red) are given a length of zero by pointing both of their boundaries to the same location.

average size of the buckets, reducing the average number of probes required to find an item and speeding up retrievals. However, it increases the amount of space used by the structure, since the starting locations of more buckets must be recorded. It also increases the amount of time taken to build the table since the radix sort must do more work (we discuss this in Section 4.3). Decreasing B, on the other hand, places more items in each bucket, meaning that more items must be checked for each query. The effect is exacerbated when queries cannot be answered, as every item in the query key's bucket must be examined before declaring failure. We found the most balanced setting to be $B \approx 0.5N$.

An example of our construction procedure is detailed in Figure 4.3. We begin by identifying which bucket every input item falls into, then store these IDs in a separate array. We then rearrange the input items into their respective buckets by performing a key-value radix sort, with the "keys" represented by the bucket IDs and the "values" being the input items. Because we are sorting based on what bucket each item falls into, the contents are each bucket are placed contiguously in memory, with the buckets ordered by their index. We store this array as $bucket\_contents[\,]$.

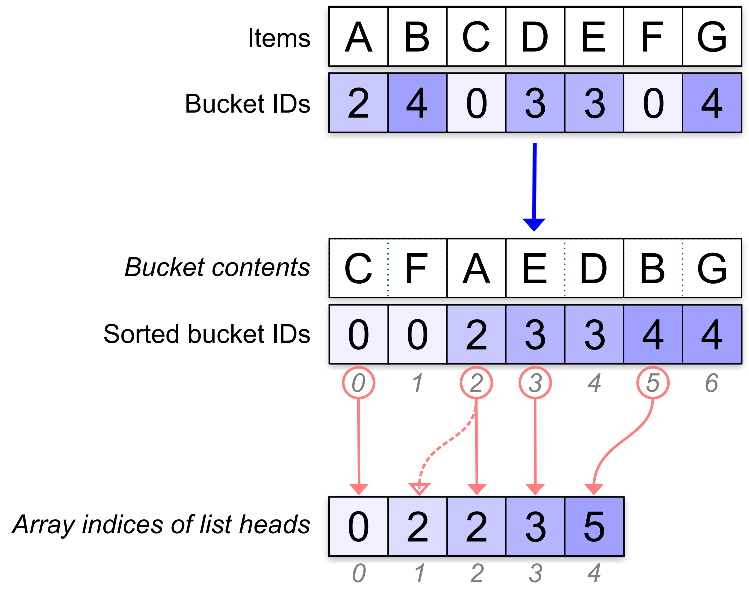


Figure 4.3. By performing a radix sort on the bucket IDs computed for each item, the input data is rearranged so that each bucket's contents are contiguous in memory (top half). Differences between neighboring entries of the sorted bucket indices array then indicate the boundaries between the lists (bottom half). The list for bucket 1 contains nothing, so its starting location is recorded as being the same as the location for bucket 2 (dotted arrow), effectively giving it a length of 0.

Finally, we identify boundaries between different buckets by examining neighboring entries in the sorted bucket IDs array. Because the array has been sorted, having two different buckets listed in consecutive entries shows the end of the former bucket and the start of the latter bucket; all of these boundaries are recorded in *bucket_starts*[ ].

Extra care must be taken for buckets that contain no items, which do not appear in the sorted list. In the Figure, the boundaries between most of the buckets is easily discovered, but no items hashed into bucket #1. These empty buckets can be inferred when a boundary is found between two other buckets, as the difference between the bucket IDs would be greater than one. In these situations, we record the location of the bucket as being the same as the location of the next bucket, effectively indicating that the bucket contains no items.

From this point, the data structure is ready for use. To perform a query, the bucket b containing the query key is first computed. The difference between two consecutive entries in *bucket_starts*[ ] indicates how large each bucket is: we pull information about where the data for buckets b and $b+1$ begin, allowing us to identify where the bucket's data begins and ends. Finally, the items are examined until either the query key or the end of the bucket is found.

4.2 Implementation

Once the number of buckets has been decided, we generate a hash function using the method described in Section 3.1.2 and allocate the memory for the arrays needed by the hash table:

- *bucket_contents*[ ], an array of N 64-bit entries that holds all of the items in the table. Every bucket's items are stored contiguously, with each bucket stored in consecutive order.

- *bucket_starts*[ ], an array of B 32-bit integers that holds the index of each bucket's first item, implicitly recording where each bucket ends. The index of the first item for bucket b is recorded at *bucket_starts*$[b-1]$; the items of bucket 0 implicitly start at *bucket_contents*[0]. The final element stores the number of items in the table to mark the end of the last bucket.

- *which_bucket*[ ], a temporary array containing N 32-bit integers.

Our first kernel uses *which_bucket*[ ] to record the index of the bucket each item falls into. We use thread blocks containing 64 threads, with each thread assigned to a different input item; the i^{th} thread computes the bucket for the i^{th} item using the hash function, then stores the result in *which_bucket*$[i]$. We then use Duane Merrill's radix sort [29] to rearrange the input data into their respective buckets, producing *bucket_contents*[ ] and the sorted *which_bucket*[ ] array.

Another kernel produces *bucket_starts*[ ] by examining neighboring entries of the sorted *which_bucket*[ ] array; the snippet we use is presented in Listing 4.1. We use N threads, each checking for differences between consecutive entries of *which_bucket*[ ]. If they are different, then the thread has discovered the boundary

Listing 4.1. Compacting down the information about each bucket

```
__device__ void find_boundaries(const unsigned  num_keys,
                                const unsigned  num_buckets,
                                const unsigned *which_bucket,
                                      unsigned *bucket_starts) {
  // Each thread looks at one entry in the sorted bucket index list.
  unsigned index = threadIdx.x +
                   blockIdx.x * blockDim.x +
                   blockIdx.y * blockDim.x * gridDim.x;
  if (index >= num_keys)
    return;

  unsigned previous_bucket = (index > 0 ? which_bucket[index-1] : 0);
  unsigned my_bucket       = which_bucket[index];

  // bucket_starts[i] stores the starting location of bucket i+1.
  // If the previous entry lists a different bucket index, a boundary
  // was found; record the array index as the beginning of the
  // affected buckets.
  if (previous_bucket != my_bucket) {
    for (unsigned i = previous_bucket; i < my_bucket; ++i) {
      bucket_starts[i] = index;
    }
  }

  // The last thread closes off all buckets at the 'end', which
  // includes the final bucket containing an item and all buckets
  // after it (which are empty).
  if (index == num_keys - 1) {
    for (unsigned i = my_bucket; i < num_buckets; ++i) {
      bucket_starts[i] = num_keys;
    }
  }
}
```

Listing 4.2. Querying the hash table

```
__device__ unsigned query_table(const unsigned  num_buckets,
                                const Entry    *table,
                                const unsigned *bucket_starts,
                                const unsigned  key) {
  // Figure out what bucket it's in.
  const unsigned bucket_id       = hash_function(num_buckets, key);
  const unsigned list_start      = (bucket_id > 0 ?
                                        bucket_starts[bucket_id-1] : 0);
  const unsigned next_list_start = bucket_starts[bucket_id];

  // Traverse the bucket's linked list.
  unsigned location = bucket_data_starts_at;
  unsigned value = NOT_FOUND;
  while (location < next_list_start) {
    Entry current_entry = table[location];
    if (get_key(current_entry) == key) {
      value = get_value(current_entry);
      break;
    }
    location++;
  }
  return value;
}
```

between the buckets named by *which_bucket*$[i-1]$ and *which_bucket*$[i]$. Because buckets might be empty, i is recorded as the starting index for all buckets b with index *which_bucket*$[i-1] < b \leq$ *which_bucket*$[i]$. A special case occurs for the last thread with index x, which must cap the final bucket(s). This set includes the last non-empty bucket (named by *which_bucket*$[x]$) and all of the empty buckets after it. From here, *which_bucket*[ ] is no longer needed.

Querying the table is done by the code in Listing 4.2. The thread performing the query first computes the bucket its query falls into, then retrieves the starting

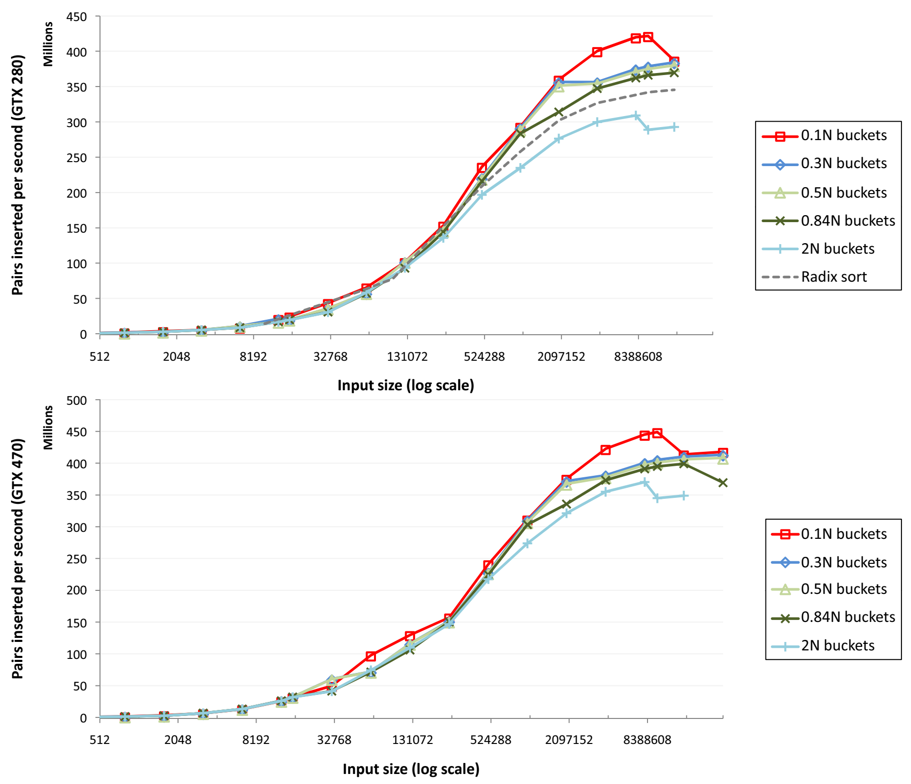


Figure 4.4. Effect of the input size on construction rates for chaining hash tables with varying numbers of buckets. The rate for a 32-bit key, 64-bit value radix sort is shown for comparison on the GTX 280.

location of the bucket's list. It also pulls the starting location of the next bucket to find the length of the list. It then simply loops over each entry of the list probing for the query. Note that a minimum of two memory accesses are required to perform any query, since information about the lists must always be pulled.

4.3 Performance analysis

Again, we present results using the setup described in Section 2.3.2.

4.3.1 Effect of changing the number of buckets

Figure 4.4 shows how the input size affects the insertion rates for chaining hash tables. Data consisted of increasingly large amounts of random 32-bit key-value pairs, where all of the keys were unique. We plot the rates for 5 different settings of the number of buckets, ranging from $0.1N \leqslant B \leqslant 2N$.

The rate at which the table can be built tends to decrease as the number of buckets is increased, with the hash table using $B = 0.1N$ generally being built the quickest. Both the GTX 280 and 470 show the same trends and the exact same odd behavior: all five tables start out with similar rates then break away after specific input sizes. This is directly attributable to the way the radix sort implementation works: to sort 32-bit keys, it uses 8 passes and bins that are 4-bits wide. Our buckets are consecutively numbered from 0 to $B - 1$, so the upper bits are always going to be 0 and the later radix sort passes have nothing to do. The points in the graph where the performance drops suddenly indicate where the bucket IDs require more bits and trigger another pass.

To show this, we plotted the rate for performing a 32-bit key, 64-bit value radix sort alongside the hash table rates for the GTX 280 in the Figure as a baseline. All of the keys were unique and ranged from 0 to 4,294,967,295, representing the worst case scenario where all 32 bits have to be examined. For the most part, the hash tables have higher insertion rates than the radix sort, indicating that they benefit from having fewer passes. They are, however, dragged down by the search for the bucket boundaries.

Figure 4.5 plots the rate at which all of the input items are retrieved from the hash tables. All queries are shuffled randomly prior to being performed to represent random access. On the GTX 470, the retrieval rates for the hash tables all follow a mostly linear trend, but have a slight downward decline as the input size increases. Because each of the buckets are contiguously stored, the cache can pull up the contents of the bucket and significantly reduce the number of trips to the device memory. The graph for the GTX 280 is a bit odd: the peak indicates that chaining works well for inputs containing 250K items or less, but the rates drop sharply afterward and plateau like the GTX 470 rates[1]. We found that the height of the plateau was related to how many probes were performed:

[1] The drop in performance for larger datasets appears again for our other results, but we don't know what causes it. However, we have had similar issues in the past caused by faulty graphics card drivers that were fixed in later updates.

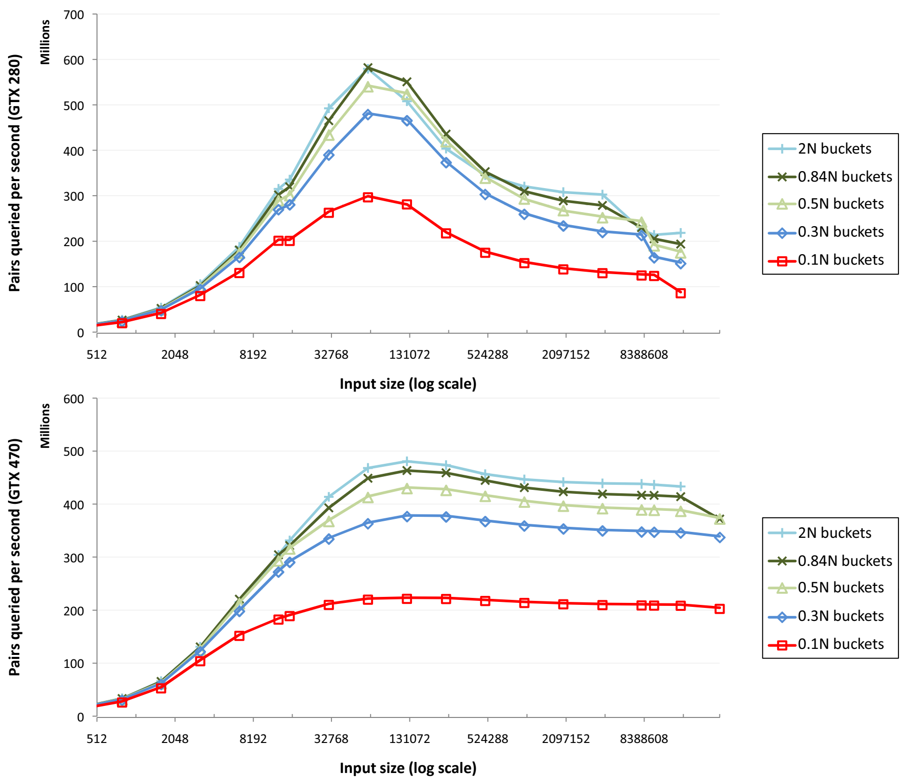


Figure 4.5. Effect of the input size on retrieval rates for chaining hash tables with varying numbers of buckets.

prematurely stopping all queries after each probe shows that the plateau lowers with longer probe sequences but eventually settles.

The trends show that going for a higher construction rate results in a slower retrieval. Using more buckets decreases the average number of items in each bucket and increases the retrieval rate, but with diminishing returns: if all of the query keys exist in the table, there is little benefit to adding more buckets when the average size of each bucket is one or less because the queries will always hit occupied buckets.

However, the advantage is more pronounced when the queries have a chance of failing (Figure 4.6). We built chaining hash tables with a varying number of buckets on both GPUs but the trends were similar; we omit the GTX 280 graphs for brevity. The cost of answering a bad query for hash tables with only $B = 0.1N$ is expensive, with the retrieval rate decreasing by over 40% when none of the queries can be found. This is because the entire bucket's contents must be searched in

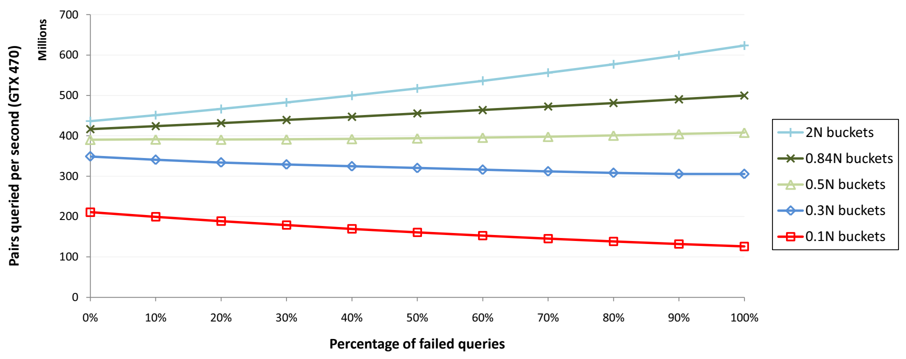


Figure 4.6. Effect of querying the hash table with keys that cannot be found in a hash table containing 10M items on the GTX 470. Each hash table was queried 10M times with increasing percentages of these bad queries. Trends for the GTX 280 were the same and are omitted.

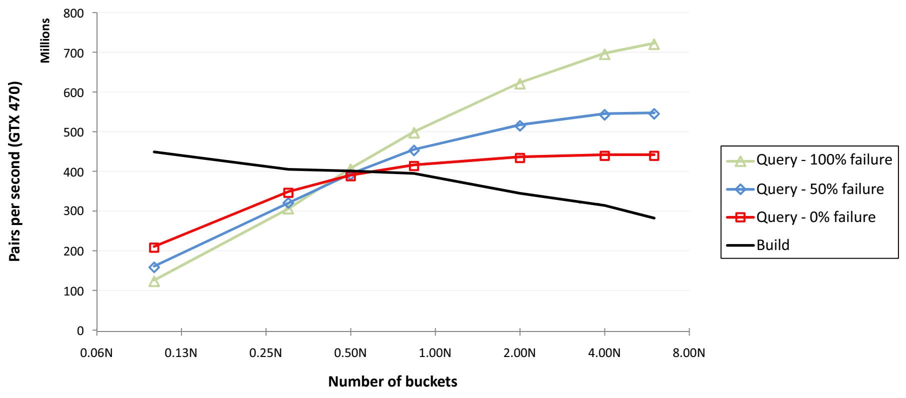


Figure 4.7. Effect of the number of buckets used by hash tables containing 10M items on construction and retrieval rates. Results are shown for the GTX 470, but trends for the 280 are similar. Rates for three different scenarios are presented, which range from finding all to finding none of the query keys.

order to confirm that the query won't be found: fewer buckets means the average number of items in each bucket is relatively high. As expected, the cost of a failed query decreases as the number of buckets increases. For the cases where a large number of buckets are available, it is likely that the failed queries are hitting empty buckets and providing faster retrievals. There is a tipping point near $B = 0.5N$ where failed queries actually become *cheaper* to answer than when all of the keys are retrieved; this robustness can be useful for applications that depend on having consistent performance.

We examined the behavior more closely in Figure 4.7. We again built hash tables with 10M items and queried them with increasingly higher percentages of failed queries, but we focus on the effect of varying the number of buckets. The construction rate again behaves as expected, decreasing as the number of buckets increases. However, the cost of a fast construction and smaller memory usage is the big drop in retrieval rates for $B < 0.8N$.

The retrieval rates show a balanced point at $B = 0.5N$. After this point it is considerably cheaper to fail to answer a query than it is to succeed, since many of the buckets hit by the queries could be empty. This advantage only grows as the number of buckets increases, which increases the likelihood that some buckets are empty. Before this point, though, it is slightly tilted in the other direction: the larger buckets favor queries that can exit early by succeeding.

The most balanced trade-off between the construction and retrieval times occurs around $B = 0.5N$ since it results in a consistent retrieval rate. Significantly higher query rates can be obtained at a marginal construction cost by moving to $B \approx 0.84N$: although it helps little for successful queries, failed queries gain a significant speed boost.

4.3.2 Comparisons with other methods

As we did with our open addressing hash tables, we compared the performance of our chaining hash tables against binary searching a radix-sorted array of the input items[2]. We also compared against quadratic probing hash tables, which were the most balanced of the three methods we presented in Chapter 3. To form a fair comparison, we choose pairs of hash tables occupying the same amount of memory: a quadratic probing hash table with N slots occupies as much space as a chaining hash table containing $2B$ buckets. This is due to the fact that each hash table slot requires 8 bytes to store a key and value, while a bucket only needs to store 4 bytes for where its data is located.

Figure 4.8 compares the construction rates of all of the methods, with three different space configurations for both chaining and open addressing. Radix sort

[2]This is a 32-bit key, 32-bit value radix sort and different from the radix sort we showed in Figure 4.4.

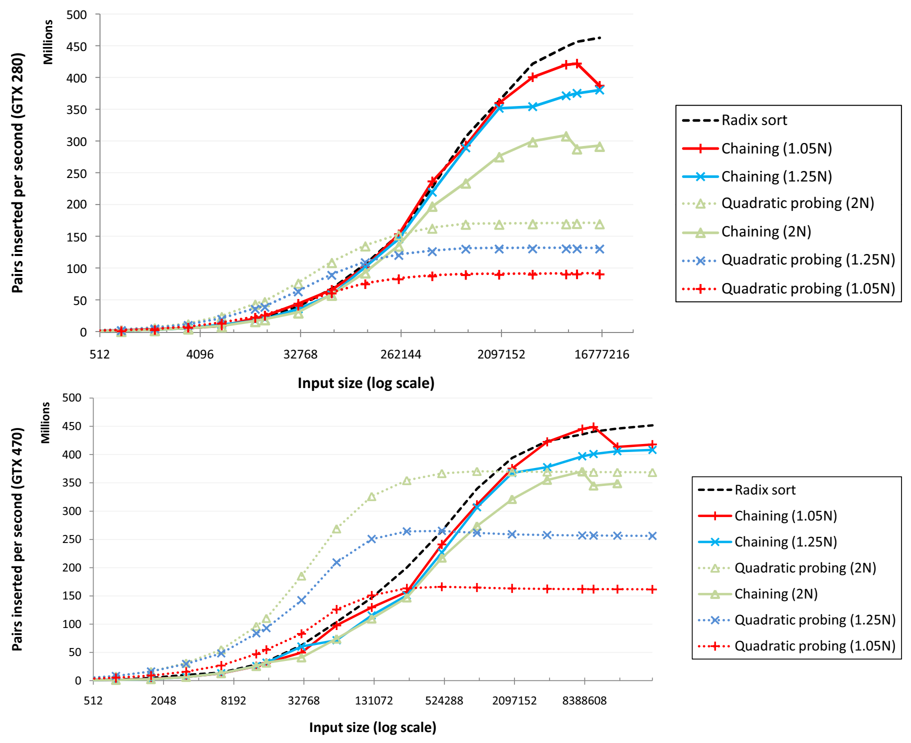


Figure 4.8. Comparison of insertion rates between chaining, open addressing, and radix sorting an array of the input items for increasingly larger sets of random data. We timed both the GTX 280 (top) and GTX 470 (bottom); each hash table type was tested with three different space usages.

almost always performs better than chaining because of the extra work required to find the boundaries between the buckets – even though chaining uses a radix sort as part of its construction, it uses a more expensive 32-bit key, 64-bit value sort.

On the GTX 280, the chaining hash tables have build rates that are around 3x as fast as the equivalent quadratic probing construction. Chaining continues to perform well on the GTX 470, but quadratic probing gets a significant boost because of the faster atomics, and has its rates ramp up much more quickly than the radix sort needed by our chaining algorithm. Moreover, the quadratic probing construction continues to be faster than the chaining construction for the $2N$ case. In this case, there is much less work to do because most threads will be able to find empty slots quickly.

Figure 4.9 compares the corresponding retrieval rates. Like we did with the comparison graphs in the previous chapter, we present binary searching rates for

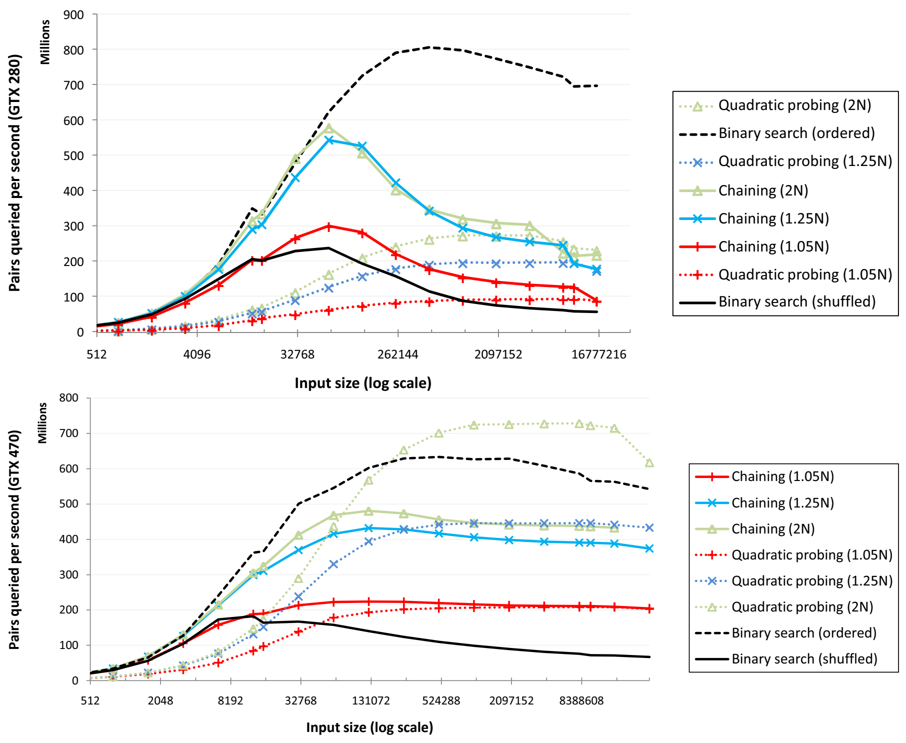


Figure 4.9. Comparison of query rates between chaining, quadratic probing, and binary searching a sorted array of the input items for increasingly larger sets of random data. We timed both the GTX 280 (top) and GTX 470 (bottom); each hash table type was tested with three different space usages. Queries were completely shuffled to represent random access, though we also show the rate for a binary search when the queries are sorted to serve as a performance baseline.

when the queries are either shuffled (representing random-access) or completely sorted; the latter is shown as a performance baseline. The Figure shows that while radix sort has the fastest insertion rates, randomly accessing elements of the sorted array using binary searches performs worse than all of the hash tables after a certain input size; this point occurs earlier on the GTX 470.

Chaining has a major advantage over quadratic probing on the GTX 280 until the hash table contains around 8 million items, when the rates meet. This is interesting because chaining requires more probes, on average, to answer a query and requires two extra memory accesses to pull information about the query key's bucket; we take statistics on this later in the section. The story is different on the GTX 470, where all of the retrieval rates get a boost. Quadratic probing performs especially well when using $2N$ space, reflecting the table's sparsity. Even quadratic

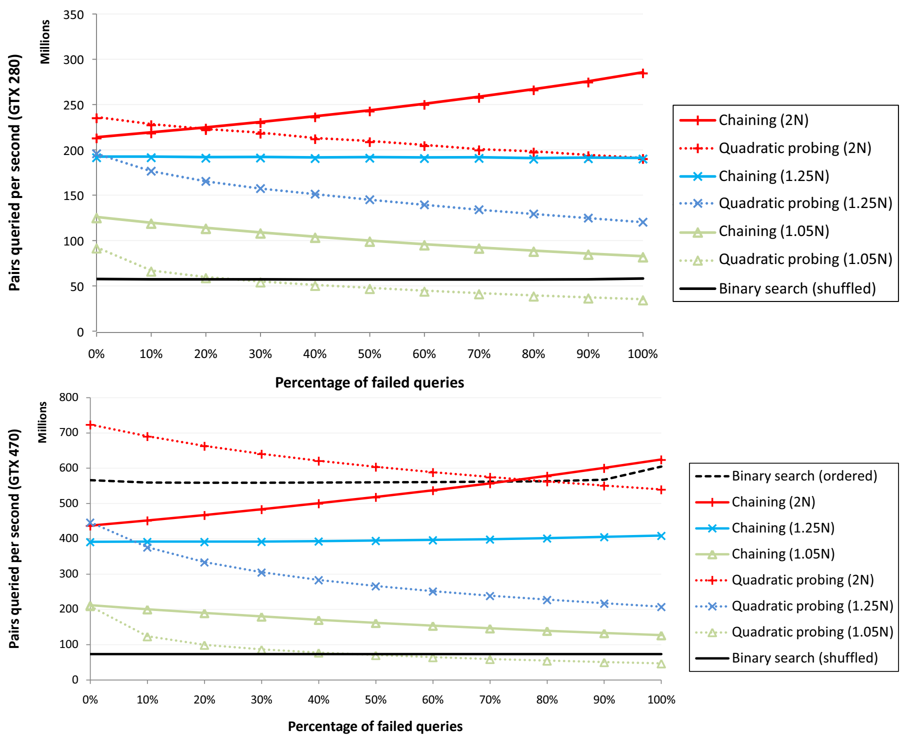


Figure 4.10. Comparison of how well different data structures handle failed queries. Each structure contained 10M items and was queried with 10M different keys. The queries consisted of a mixture of both the input keys and keys not existing in the hash table. The binary search with ordered queries was omitted from the GTX 280 graph to make the graph clearer; it hovered far above the other lines in the graph and handled queries at a rate of round 690M pairs per second.

probing with $1.25N$ space has higher retrieval rates than all of the chaining tables when $N \geqslant 1,000,000$.

When looking for query keys that are not in the table, you should almost always use a chaining hash table on the GTX 280 since it will generally outperform the corresponding quadratic probing table using the same amount of memory (Figure 4.10). On the GTX 470, quadratic probing has a significant performance advantage over chaining for larger table sizes because of the speed boost the architecture provides, giving it comparable insertion rates and significantly faster retrieval rates. Chaining does, however, maintain the advantage for smaller table sizes because it is much more robust. Moreover, it is consistently faster than the

Queries failed	Scheme	Size	AVG	50%	80%	90%	99%	100%
None	Chaining	$1.05N$	6.00	6	9	11	16	29
		$1.25N$	2.00	2	3	4	6	12
		$2.00N$	1.25	1	2	2	3	7
	Quadratic probing	$1.05N$	3.39	1	4	8	29	264
		$1.25N$	2.13	1	3	4	12	80
		$2.00N$	1.43	1	2	2	5	22
All	Chaining	$1.05N$	10.00	10	13	14	18	31
		$1.25N$	2.00	2	3	4	6	12
		$2.00N$	0.50	0	1	1	3	7
	Quadratic probing	$1.05N$	21.85	15	35	50	98	318
		$1.25N$	5.43	4	9	12	23	78
		$2.00N$	2.13	1	3	4	8	26

Table 4.1. Statistics for a single run of querying the chaining and open addressing hash tables with different space constraints. The number under each percentage indicates how many probes were required for that percentage of threads to answer their queries. The number of buckets used by chaining for table sizes $1.05N$, $1.25N$, and $2N$ was $0.1N$, $0.5N$, and $2N$, respectively.

binary search with shuffled queries and can be faster than even the binary search with sorted queries.

We can see why chaining does so well against quadratic probing by taking statistical distributions on the number of probes required to answer a query for both hash tables (Table 4.1). Both types of hash tables were each built with the three different space constraints. Each hash table configuration was built twice with 10,000,000 items; the first was queried with all of the input keys, while the second was queried entirely with keys that were not present in the table. The number of probes in the chaining case does not include the two memory reads required to determine bucket locations and lengths.

While chaining has a higher median when all of the query keys can be found in the table, it is far more robust at distributing its items than quadratic probing. This is evident in the maximum number of probes required to complete all of the queries, which is much lower. For the $1.05N$ case, its probe distribution skews upwards when none of the queries can be found, resulting in a decrease in performance. Conversely, the probe distribution skews downwards for the $2N$ case because many of the queries land in buckets that are empty, resulting in the faster queries. The robustness of the $1.25N(B = 0.5N)$ case is shown by how the distribution did not change at all between the two trials.

On the other hand, the probe distributions for quadratic probing always skews upward as the percentage of bad queries increases, regardless of the table size. This allows chaining to gain the performance advantage for the $1.05N$ and $1.25N$ cases. Despite requiring a lower average number of probes to answer a query in the $2.0N$ case, where all queries can be found, chaining still performs significantly worse on the GTX 470, which is possibly due to the extra memory accesses required to grab the bucket information.

An interesting thing to note is that the average number of queries required to answer a probe when none of them can be found is the average length of each bucket. The same isn't true for the case when all of the queries can be found; the average is lower for $1.05N$ because the threads can short-circuit upon finding their query keys, and higher for $2.0N$ because the threads always hit an occupied bucket.

4.4 Limitations

Modifying the table is difficult because the buckets are tightly packed, with each bucket butting up against its neighbors. Removals can be performed by clearing the slot of the entry to be deleted, but this would not decrease the query probe lengths because the bucket sizes cannot be changed without affecting neighboring buckets. Extra work could be done, however, to move dead slots to the end of each bucket. Items could be inserted into these dead slots, but they cannot be inserted into a full bucket without rebuilding from scratch.

Temporary storage is required during construction to store which bucket each item falls into. While it doesn't need to be stored after its use, the extra space it requires prevents building chaining hash tables for larger input sizes with looser space constraints.

Variability in the length of probe sequences is smaller than for quadratic probing, leading to more robust retrievals. However, the number of probes required to answer a query can still get fairly high for more compact tables.

4.5 Summary

We introduced a method for building chaining hash tables that trades the costs of faster construction and smaller table sizes with more expensive retrievals. It provides a good alternative to using a radix sorted array: the extra bucketing step we use allows us to replace the binary search with an array traversal, where the length of each traversal is typically smaller than the length of the corresponding binary search. It is consistently better than using open addressing on the GTX 280, and retains a performance advantage on the GTX 470 when using compact hash tables. If the size of the table is unimportant, using open addressing methods makes much more sense because of the significant boost in performance they provide.

However, while its retrievals are more robust to outliers than the open addressing schemes we discussed previously, chaining hash tables still require a relatively large and variable number of probes to answer its queries for compact tables. The methods we discuss in the following chapters address this by making different trade-offs from chaining, utilizing a slow construction algorithm to guarantee that the retrievals will be fast.

Chapter 5

Two-level cuckoo hashing

As we have seen in previous chapters, the performance of open addressing methods like quadratic probing and double hashing is highly dependent on the occupancy of the table: compact tables result in skyrocketing probe sequence lengths. *Cuckoo hashing* addresses this by limiting the number of slots an item can be located in, guaranteeing that all items can be found with a small, constant number of probes. One big problem with cuckoo hashing, however, is that it can fail to insert items, requiring that the whole table be rebuilt from scratch with new hash functions.

In this chapter, we look at a method of reducing the cost of failure by using a two-level hash table. Our first level works similarly to chaining, which partitions the items into smaller buckets. We then build cuckoo hash tables for each of these smaller buckets individually. The effects of failing to build the cuckoo hash table for one bucket are then limited to rebuilding the cuckoo hash table for just that

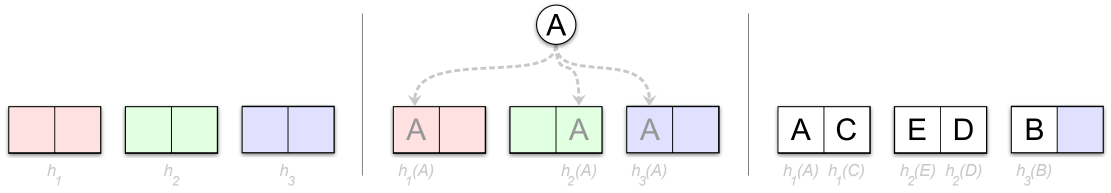


Figure 5.1. Cuckoo hash table with three subtables, each with its own hash function (left). Every key is assigned one slot in each subtable (center). The construction algorithm finds a conflict-free configuration where each item resides in one of its possible slots (right).

one bucket. This scheme reduces the cost of failure while still providing the same constant-time retrievals guaranteed by regular cuckoo hashing.

The parameters we chose for our implementation allow the data structure to be constructed at interactive rates for millions of items, use approximately $1.42N$ space for N pairs of keys and values, and limit the number of global memory accesses required to answer any query to just four. We found it to be more efficient than regular open addressing and chaining on the GTX 280, but worse on the GTX 470; the version of cuckoo hashing we present the next chapter tackles cuckoo hashing differently and does not suffer from the same issue.

We give an overview of our algorithm in Section 5.1, giving an introduction to cuckoo hashing and describing our solution for parallelizing it. We go into the details of the construction of a basic hash table that stores a single value for each key in Section 5.2, then briefly describe how to extend our construction algorithm for more specialized hash tables in Section 5.3. We discuss our results in Section 5.4 and our method's limitations in Section 5.5. Finally, we close with some thoughts in Section 5.6.

5.1 Overview

Like other open addressing methods, cuckoo hashing uses a table consisting of $S_T \geqslant N$ slots to store N items, with each slot capable of holding a single key-value pair from the table. The key difference is how cuckoo hashing handles a collision: instead of continuing to search for an empty slot like quadratic probing, it instead evicts items from the table to make room, forcing them to be reinserted elsewhere.

We use a variant where the cuckoo hash table is broken into H equally-sized subtables, each associated with its own independent hash function. Items can hash into one location in each subtable, but they are always located in exactly one slot once inserted. This limitation guarantees that an item can be found after checking only H locations; typically H is set to either 3 or 4. Figure 5.1 shows an example of a cuckoo hash table using three subtables.

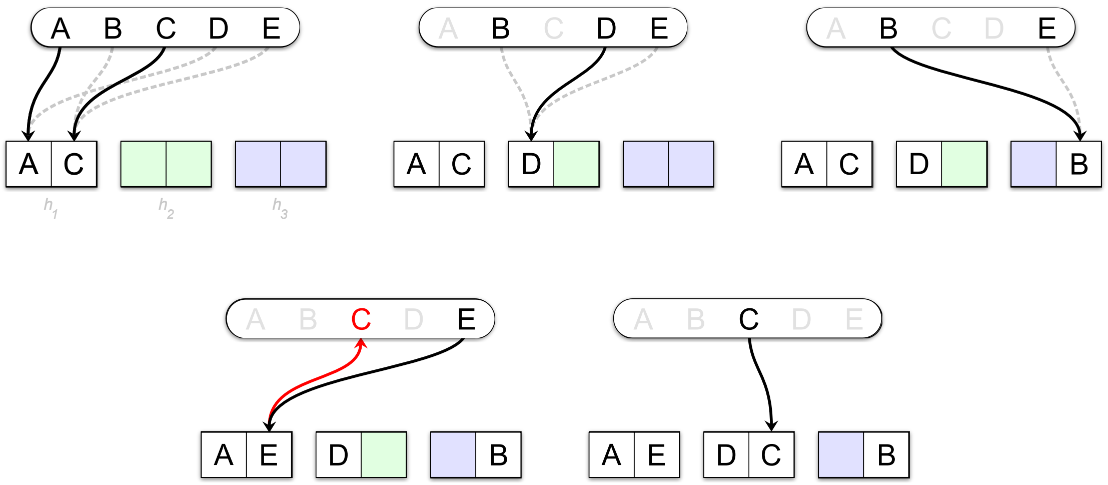


Figure 5.2. Parallel cuckoo hashing with three subtables. At every iteration, threads simultaneously write unplaced input keys into the same subtable using the subtable's $h(k)$. CUDA ensures one write will succeed for every slot; threads synchronize and check the subtable to see which keys were successfully written. Overwritten keys move onto the next subtable. This process iterates through all of the subtables in round-robin order until termination.

The sequential construction algorithm inserts items one by one. The current item being inserted first checks its H slots to see if any of them are empty, immediately inserting itself if one is available. If not, it evicts one of the items occupying one of its slots and takes the slot for itself. This starts a recursive process where the evicted item must be reinserted into the table following the same procedure. Although this process might continue forever without finding a location for every item with an unfortunate choice of hash functions, the probability of this is provably small when using two hash functions, and empirically unlikely using more hash functions. Intuitively, moving the items during an insertion allows items to flow toward less contested slots in the table.

We modified the standard cuckoo hashing algorithm to work in parallel, allowing all of the key-value pairs to be inserted into the structure simultaneously. For simplicity, we assume that a thread manages the same key-value pair during the entire process and is responsible for finally storing it in the table. The main issue is to define the semantics of parallel updates correctly, making sure that collisions between items are properly handled.

At the beginning of every iteration, all threads managing items that have not yet been stored simultaneously write their keys into the same subtable using the associated hash function; CUDA semantics ensure that exactly one write will succeed for each slot that is written into. We then invoke a thread synchronization primitive to ensure that all threads have had a chance to insert their items. Any threads expecting their keys to be in the subtable then check to see if this is true; if not, the threads move onto the next iteration and try again. The procedure visits each subtable in round-robin order, wrapping back to the first subtable once all of the others have been visited. This stops once all keys are placed or too many iterations occur; in the latter case, we assume it won't stop and restart with new hash functions. An example of the process is shown in Figure 5.2.

This procedure is efficient for small datasets, but GPU performance can suffer as the input size increases for two reasons. First, the hash functions distribute items as evenly as possible into the table, potentially sending nearby input items to distant locations in the structure. Because hash tables for reasonably-sized datasets must reside in global memory, every iteration of the construction incurs highly uncoalesced memory accesses. Second, the algorithm can fail to insert items and require the whole table be rebuilt from scratch, even if only one item failed to be stored.

We address these issues by using a two-level hash table, shown in Figure 5.3. Our first level partitions the input items into smaller buckets using a hash function $g(k)$. We then build a cuckoo hash table for each bucket in parallel, with a different CUDA thread block handling each bucket. This allows us to build each cuckoo hash table in shared memory, reducing the cost of the memory accesses incurred during construction. Moreover, it mitigates the cost of a cuckoo hashing failure since each cuckoo hash table is independent: failing to build one does not cause the others to be rebuilt from scratch.

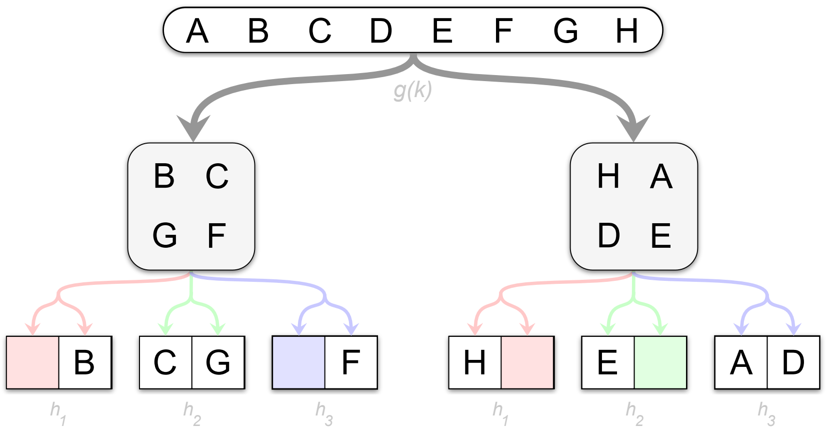


Figure 5.3. High-level construction overview of our structure. The input (top) is first partitioned into smaller buckets using a hash function $g(k)$ (center). A cuckoo hash table is then built with each bucket's contents in parallel (bottom) in shared memory. Each bucket uses its own $h_i(k)$.

5.1.1 Parameters

Capacity of the buckets for the first level must be set so that each individual cuckoo hash table can fit within shared memory; again, this lets all threads in the same thread block work together to build the cuckoo hash table for a single bucket. Thread blocks may have either 16 kB or 48 kB available, depending on the compute capability of the GPU. In practice, less than this is actually usable by a kernel, but it still allows cuckoo hash tables containing several thousand slots to be created.

Setting this number low produces cuckoo hash tables that have fewer items, which tend to take fewer iterations to build. This is ideal for increasing construction rates, since it allows thread blocks to vacate the GPU more quickly and allow other thread blocks to build their cuckoo hash tables. It also limits the number of threads that are waiting for the table to be constructed: even if one item is unplaced, the entire thread block must continue iterating.

Alternatively, it can be set higher to promote more compact hash tables. However, there is a hard limit on the size of a thread block, which is 512 or 1024 threads (again, depending on the GPU). Allowing a bucket to hold more than these numbers requires the threads to juggle multiple items during insertion, which increases the construction time. For performance, we settled on setting the capacity, denoted C, to match these limits.

Bucket occupancies would ideally be as close to C pairs as possible without being overfilled since this requires redistribution with a new hash function. However, it is difficult to achieve this without tailoring a specific $g(k)$ for every given input. To achieve fast construction times, we instead rely on randomly generated hash functions and aim to fill the buckets to a percentage of their actual capacity; we denote the target percentage as T. Aiming for a lower T makes it less likely that any given hash function will assign too many items to any given bucket, but increases the amount of memory required by the hash table.

C and T are directly related because bigger buckets can more reliably support a higher average occupancy without overfilling. Empirically, we found that setting $T = 80\%$ when $C = 512$ overfilled buckets on only 0.5% of the trials we ran when building a table for 5 million random key-value pairs. We found that it was possible to maintain a similar failure rate while setting $C = 1024$ or $C = 2048$ and aiming for target average occupancies of $T = 85\%$ and $T = 90\%$, respectively. We tested sizes up to $C = 8192$, which allowed us to reliably hit occupancies of 95%, but settled on $T = 80\%$.

After setting both C and T, the number of buckets we require can be computed as $B = \frac{N}{C \cdot T}$.

The number of hash functions used for the cuckoo hash tables determines how many slots an item can choose from when being inserted into the hash table. Using more hash functions makes the construction process easier because there are more ways to resolve collisions, which makes it possible to make more compact hash tables. However, it increases the number of probes required to answer a query and decreases the rate at which retrievals can occur.

For $H = 2$, the expected maximum number of steps required to insert an item is $O(\lg N)$. A recent result shows that the expected maximum number of steps required to insert an item can be polylogarithmic for sufficiently large H, and it is believed that the expectation is actually logarithmic; Frieze et al. [15] have more background and details. Nonetheless we find that the number of iterations is reasonable in practice.

One major issue with using only two hash functions is that the cuckoo hash table *must* have slightly more than $2N$ slots. Using three or four functions drops the minimum table size to around $1.1N$ or $1.03N$ slots, respectively [10]. The extra probes required when using more hash functions are easily balanced out by the efficient memory usage, so our cuckoo hash tables use $H = 3$. Because it is very difficult to build near-minimal hash tables, we introduce some slack and set the size of each cuckoo hash table to be $S_{cuckoo} = 3 \cdot (\frac{N}{4} + \frac{N}{8})$, producing hash tables that are slightly larger than $1.1N$.

For performance reasons, we use the same size for all of the cuckoo hash tables regardless of the number of items actually contained in each bucket. While it wastes space when buckets are underfilled, the constant size prevents the extra bookkeeping needed to track where each bucket's cuckoo hash tables are located in memory, which requires reading extra memory locations during retrieval.

The overall table size can be calculated by the formula $\frac{S_{cuckoo}}{C \cdot target}$. Using $H = 3$, $C = 512$, and a target occupancy of 80% results in tables of size $S_{cuckoo} = 576$, giving an overall table size of about $1.42N$, while using $C = 1024$, a target load of 85%, and $S_{cuckoo} = 1152$ produces a table of size $1.33N$.

5.2 Basic hash tables: One value per key

We begin by discussing the implementation of a basic hash table, which stores a single value for each key. The input consists of pairs of 32-bit keys and values, where none of the keys are repeated; we discuss how to handle repeated keys in Section 5.3. Note that a 32-bit value could represent an index into another structure, allowing a key to reference more than just 32 bits of data. We build the table in two phases: Phase 1 partitions the pairs into the smaller buckets, and phase 2 builds cuckoo hash tables for each bucket in parallel.

Algorithm 5.2 Distribution of input into buckets containing at most C pairs each

```
1: estimate number of buckets $B$ required
2: allocate output arrays and scratch space
3: **repeat**
4:    set all bucket size counters $count[i] = 0$
5:    generate hash function $g(k)$
6:    **for all** $k \in$ keys in parallel **do**
7:       compute $g(k)$ to determine bucket $b_k$ containing $k$
8:       atomically increment $count[b_k]$, learning internal $offset[k]$
9:    **end for**
10: **until** all $count[i] \leq C$
11: perform prefix sum on $count[\,]$ to determine $start[\,]$
12: **for all** key-value pairs $(k, v)$ in parallel **do**
13:    store $(k, v)$ in $shuffled[\,]$ at index $start[b_k] + offset[k]$
14: **end for**
```

5.2.1 Phase 1: Partitioning the input pairs

Phase 1 distributes the pairs into small buckets containing at most C pairs each using a hash function $g(k)$. We follow the procedure described in Algorithm 5.2 and illustrated in Figure 5.4, launching kernels that use one thread for each input pair. Two arrays in global memory are produced for phase 2: *shuffled*[ ], which contains the input keys and values swizzled together so that each bucket's contents are contiguous in memory, and *start*[ ], which indicates where each bucket's data starts in the first array. This scattering operation allows blocks in phase 2 to perform coalesced memory reads while building the cuckoo hash tables.

The hash function, $g(k)$, is a linear polynomial of the form:

$$g(k) = (a \cdot k + b) \bmod p \bmod B$$

Here, a and b are 16-bit integer constants and $p = 1900813$ is a prime number larger than the number of buckets. If we find that $g(k)$ overfills any bucket, we

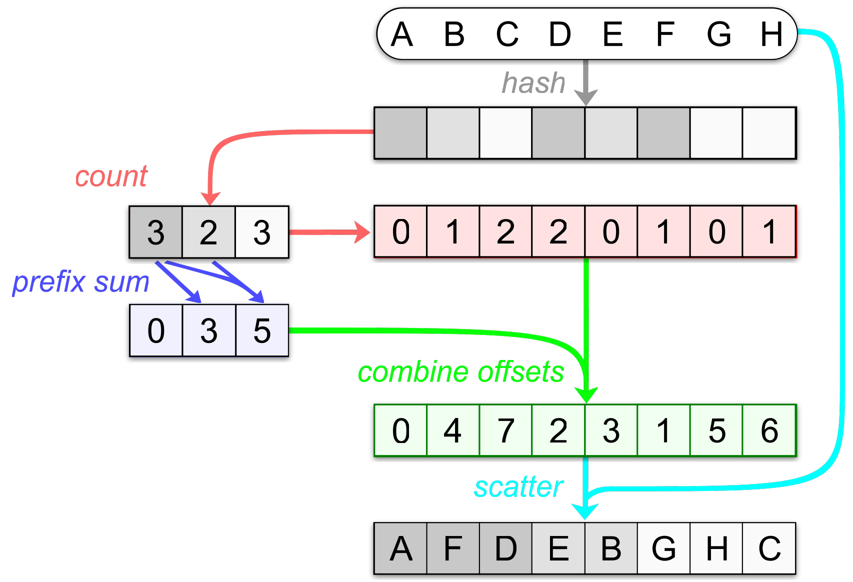


Figure 5.4. Phase 1 breaks the input (top) into small buckets and rearranges the input so that each bucket's contents are contiguous in memory (bottom). Threads first take the keys and figure out which bucket they hash into, represented by the gray squares. Counters tracking each bucket size are atomically incremented (red) and used to give pairs an ordering among all other pairs within the same bucket. A scan operation determines where each bucket's data starts (blue), which are added to the offsets to determine where in the array the item goes (green). Items are then scattered into the proper locations (cyan).

restart the process with new a and b. Interestingly, we found that setting $a = 1$ and $b = 0$ for the first attempt worked well for many of our datasets. Even though it is not random, it seems to work and provide a small, but noticeable, improvement in our retrieval times. For subsequent attempts, we just generate two random numbers.

After the number of buckets has been decided, we allocate memory for the phase:

- *shuffled*[ ] stores all of the items after they have been scattered into their buckets for phase 2.
- *start*[ ] determines where each bucket's data starts in *shuffled*[ ].
- *count*[ ] stores the number of items falling in each bucket.

- *offset*[ ] stores each key-value pair's position inside its bucket, relative to the other pairs.

Distribution of the pairs into the proper buckets is performed by a series of kernels, where each thread manages a single input pair. We first launch a kernel to perform steps 6–9 of Algorithm 5.2, which determines how many pairs hash into each bucket. Each thread is given the same constants a and b, packed into one 32-bit integer, allowing all threads to construct $g(k)$ and compute which bucket their key k will hash into. It then atomically increments the relevant counter $count[g(k)]$ and saves the previous counter value[1]. If any counter is found to exceed C, a global error flag is set to show that a new hash function is needed and the process is restarted with a new $g(k)$.

A prefix sum is then performed on the counters using the CUDPP library [20], effectively marking off a contiguous set of indices for each bucket's data in *shuffled*[ ]. Threads finally scatter their items into *shuffled*[ ] by combining the offset given by the atomic increment and the bucket data locations given by the prefix sum.

5.2.2 Phase 2: Parallel cuckoo hashing

Phase 2 works on a local scale, independently building a cuckoo hash table for each of the buckets in shared memory; the process is shown in Algorithm 5.3. The bulk of this phase is performed by a single kernel, with each thread block building the cuckoo hash table for a different bucket. This phase produces the only two global memory arrays needed by our final structure: *seed*[ ], which contains the seed used to generate each bucket's $h_i(k)$, and *cuckoo*[ ], which contains all of the cuckoo hash tables created for every bucket. Every slot in *cuckoo*[ ] is initialized with a special constant $\varnothing$, which flags a slot as being vacant and prevents answering queries incorrectly[2].

[1]For GPUs with caches, it can be highly beneficial to pack the counters for multiple buckets into a single 32-bit unsigned integer. Since every thread is hitting the same small number of counters, they are likelier to stay within the cache if the amount of memory used by the counters is small. When this happens, the step can be sped up considerably.

[2]We chose `0xffffffff`, the maximum value of an unsigned 32-bit integer.

Algorithm 5.3 Parallel cuckoo hash table construction using three subtables

```
 1: initialize the cuckoo[ ] array
 2: allocate final arrays
 3: generate a set of random 16-bit integer seeds
 4: for all buckets $b$ in parallel do
 5:     while the cuckoo hash table has not been successfully built do
 6:         generate the $h_i(k)$ using one seed
 7:         pre-compute $h_i(k)$ for all keys $k$
 8:         while any $k$ is uninserted and failure is unlikely do
 9:             write all uninserted $k$ into slot $g_1(k)$
10:             synchronize threads and check subtable $T_1$
11:             write all uninserted $k$ into slot $g_2(k)$
12:             synchronize threads and check subtable $T_2$
13:             write all uninserted $k$ into slot $g_3(k)$
14:             synchronize threads and check subtable $T_3$
15:         end while
16:     end while
17:     write subtables $T_1, T_2, T_3$ into cuckoo[ ] and the final seed used into seed[$b$]
18: end for
```

The hash functions used by the cuckoo hash tables are similar to $g(k)$, taking the form:

$$h_i(k) = (a_i \cdot k + b_i) \bmod p \bmod S$$

Here, S is the size of a single subtable. Each $h_i(k)$ uses its own a_i and b_i; all are generated by XORing a single 16-bit integer *seed* with different constants[3]. This cuts down the number of memory accesses required to retrieve a bucket's hash functions to just one when answering queries. While this produces a set of weak hash functions, they worked well in practice and allowed most of the cuckoo

[3]We used the arbitrary set (0xffff, 0xcba9, 0x7531, 0xbeef, 0xd9f1, 0x337a).

hash tables to be built after a single attempt. The question of what implementable hash functions to use for cuckoo hashing for provable performance guarantees is a current research topic in theoretical computer science; see the original paper [34] for more on the theory and practice of the choice of hash function for cuckoo hashing, and subsequent work [30] for an analysis of why weak hash functions generally perform well in practice.

Before launching the kernel, the CPU prepares by generating a random set of seeds to pass in; this allows each thread block to make several attempts to build its cuckoo hash table without exiting the kernel for new hash functions. Because each table is built independently, the seed chosen by one bucket may be different than the one chosen by another and must be saved.

Cuckoo hashing is performed by each thread block using the parallel construction procedure sketched out in the previous section. Pairs within the bucket are distributed evenly across all of the threads. Shared memory is used as a temporary storage for building the cuckoo hash tables, which are written out to global memory once built.

The code snippet in Listing 5.1 shows how a thread managing at most one pair prepares for each construction attempt; the code can be generalized using *for* loops to iterate through all of a thread's pairs if necessary. We suggest using templatized code and loop unrolling to prevent unnecessary branching when possible.

At the beginning of every attempt, a new seed is chosen and used to create the constants for all of the $h_i(k)$. Threads begin by generating the constants used by the $h_i(k)$, then precompute and store the subtable locations each of their keys hash into. They also keep track of which subtable the key *should* be located in, which can be invalidated when another key overwrites it during insertion.

Listing 5.2 shows the process threads follow to perform the parallel construction. Variables with the sh_ prefix are located in shared memory. During this process, only keys are being written into the subtables; values are written only af-

Listing 5.1. Code snippet showing how threads prepare to perform cuckoo hashing with three subtables and each thread manages at most one pair.

```
unsigned seed_index = 0;
do {
  // Use one seed to create 6 hash function constants.
  short random_number_seed = random_numbers[seed_index++];

  unsigned constants_0 = random_number_seed ^ 0xffff;
  unsigned constants_1 = random_number_seed ^ 0xcba9;
  unsigned constants_2 = random_number_seed ^ 0x7531;
  unsigned constants_3 = random_number_seed ^ 0xbeef;
  unsigned constants_4 = random_number_seed ^ 0xd9f1;
  unsigned constants_5 = random_number_seed ^ 0x337a;

  // Figure out what index the key-value pair has in each subtable.
  uchar index_x = ((constants_0 * key + constants_1) % kPrime) % S;
  uchar index_y = ((constants_2 * key + constants_3) % kPrime) % S;
  uchar index_z = ((constants_4 * key + constants_5) % kPrime) % S;
  uchar in_subtable = kInvalidSubtable;

  // Perform cuckoo hashing.
  ...
} while (*sh_hash_isnt_built && seed_index < MAX_NUM_SEEDS);
```

Listing 5.2. Code snippet for performing cuckoo hashing using three subtables when each thread manages at most one pair.

```
for (iteration = 1; iteration <= MAX_ITERATIONS; ++iteration) {
  if (threadIdx.x == 0) *sh_hash_isnt_built = 0;

  // Keep trying the subtables until it successfully stays in.
  if (has_pair && in_subtable == kInvalidSubtable) {
    sh_subtable_1[index_x] = key;
    in_subtable = 1;
  }
  __syncthreads();
  if (has_pair && in_subtable == 1 && sh_subtable_1[index_x] != key) {
    sh_subtable_2[index_y] = key;
    in_subtable = 2;
  }
  __syncthreads();
  if (has_pair && in_subtable == 2 && sh_subtable_2[index_y] != key) {
    sh_subtable_3[index_z] = key;
    in_subtable = 3;
  }
  __syncthreads();

  // If overwritten, another iteration is needed.
  if (has_pair && in_subtable == 3 && sh_subtable_3[index_z] != key) {
    *sh_hash_isnt_built = 1;
    in_subtable = kInvalidSubtable;
  }
  __syncthreads();

  if (*sh_hash_isnt_built == 0) {
    break;
  }
  __syncthreads();
}
```

ter the subtables have been successfully built. Threads not assigned a pair remain idle.

Each thread begins by writing each of its keys into the first subtable and recording that its keys should be in the first subtable. At this point, other threads may have overwritten its keys. After synchronizing, each thread checks if its keys are still in the subtable. Overwritten keys are then written into the next subtable and recorded as being there. After synchronizing again, only keys which are expected to be in the next subtable are verified; because the previous subtable wasn't written into, their status couldn't have changed. If there are still unstored keys in the bucket after visiting all of the subtables, all threads return to the first subtable and continue the process. From this point on, previously inserted keys could be overwritten and all keys expected to be in the current subtable have to check after synchronization.

This process continues with all threads iterating through the subtables in round-robin order. Using $C = 512$ and $T = 80\%$, we found that tables needed an average of 5.5 iterations to finish, with fuller tables requiring more iterations. For these settings, we assume that the $h_i(k)$ we chose will never allow the process to complete after 25 iterations and declare failure. We restart the process for the bucket with all new $h_i(k)$ and empty subtables. Note that this restart occurs independently of all of the other thread blocks, preventing a stray item from forcing a rebuild of the entire data structure. Only a few buckets fail to build with their first hash functions, and they usually succeed after restarting once.

Once all keys have been successfully inserted into the cuckoo hash table, threads write out information back from shared memory into global memory. First, threads write the subtables into *cuckoo*[ ], with values stored immediately after their respective keys; this allows them to be read together during retrieval. Finally, the first thread of each block writes out the seed needed by the bucket for its $h_i(k)$ into *seed*[ ].

Our final structure contains four things: the total number of buckets B, the two constants for $h(k)$ stored as a single 32-bit integer, the *cuckoo*[ ] array, and the *seed*[ ] array. Figure 5.5 gives an overview of the procedure when performed on an actual dataset.

5.2.3 Retrieval

Retrieval of a query key k can be performed on the hash by first determining which bucket a query key falls into using $g(k)$; the constants required are passed into the GPU as kernel arguments. The seed for the bucket is then read from *seed*[ ] and used to recreate the $h_i(k)$. Each $h_i(k)$ is tried in order until either the key is found in the bucket's cuckoo hash table, which returns the key's value, or until all possible locations have been checked, which results in the special constant $\varnothing$ being returned to indicate that the query key wasn't found. The maximum number of global memory reads when using three hash functions is four: one for the seed, and three for probing the cuckoo hash tables.

5.3 Extending the basic hash table

The algorithm can be adapted to handle multiple copies of the same key in the input, leading to useful hash table specializations. Each specialization stores each key only once in the data structure, regardless of how many copies there are. The multiple values are either stored in an auxiliary array, or ignored entirely.

Phase 1 proceeds in almost exactly the same way as before, with $g(k)$ directing all pairs with the same key into the same bucket. As a result, it is possible for each bucket to end up with more than C pairs; to handle this, we relax the limitation and instead require that a bucket receives no more than C unique keys. Unfortunately, there is no easy way to check this until we repeatedly fail to build a bin's cuckoo hash table in phase 2; in these cases, we have to restart from scratch with a new $g(k)$. The likelihood of this actually happening is relatively low because the number of buckets is determined by the total number of key-value pairs, effectively lowering the number of unique keys assigned to each bucket.

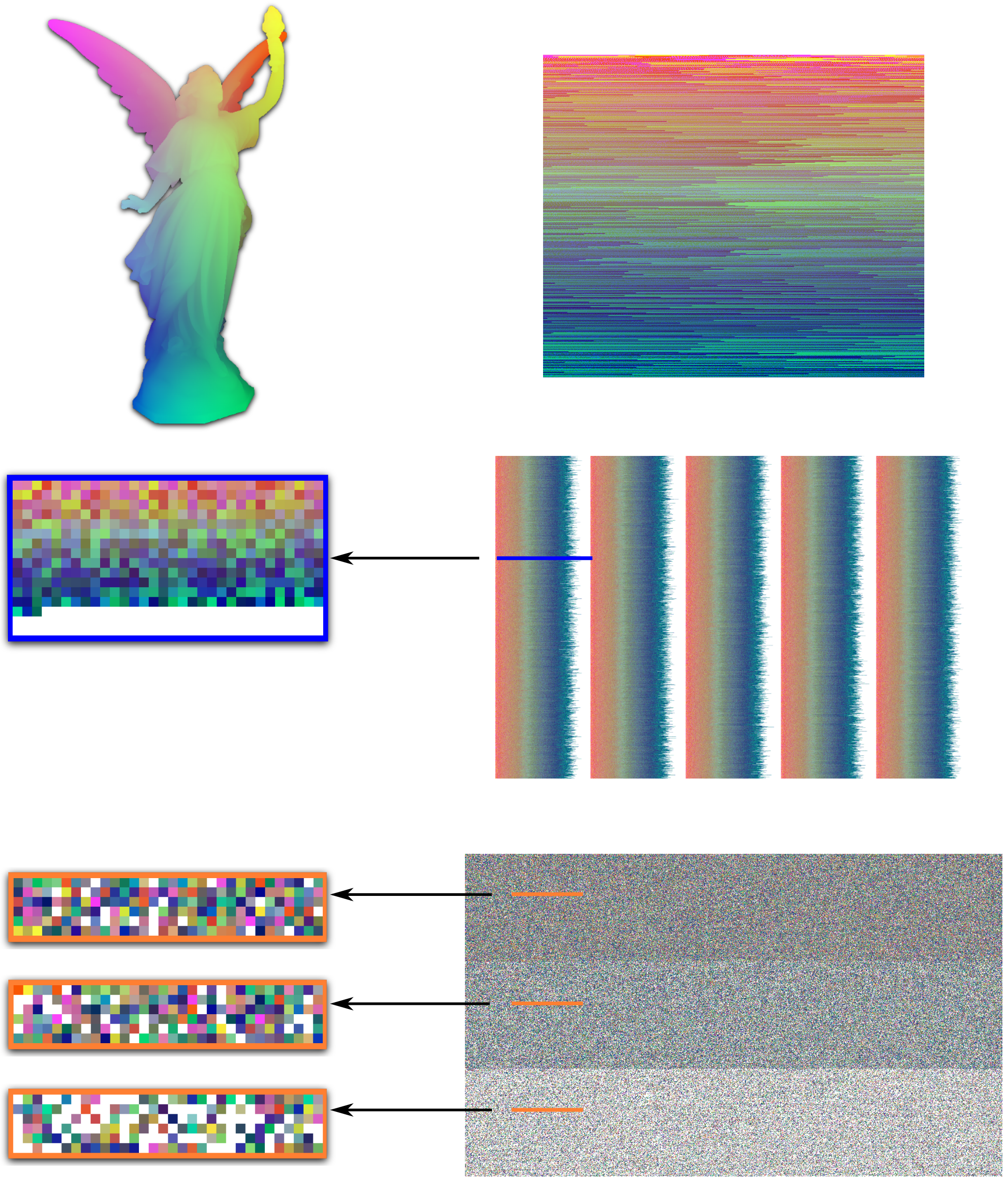


Figure 5.5. View of the memory when building a hash table for a voxelized Lucy model, colored by mapping x, y, and z coordinates to red, green, and blue respectively (top left). The 3.5 million voxels (top right) are input as 32-bit keys and placed into buckets of ≤ 512 items, averaging 409 each (center right). For clarity, the buckets are separated in this diagram, but are actually stored tightly by the algorithm. Each bucket then builds a cuckoo hash with three sub-tables and stores them in a larger structure with 5 million entries (bottom right). Close-ups follow the progress of a single bucket, showing the keys allocated to it (center left; the bucket is linear and wraps around) and each of its completed cuckoo sub-tables (bottom left).

The mechanics of phase 2 are somewhat different because different threads can have copies of the same key. Like before, threads insert all of their keys into the same subtable every iteration; any copies of the same key will always be written into the same slot. In these cases, we consider all copies to have been successfully stored as long as one copy stays in the subtable after the iteration. Conversely, if the single key instance is overwritten on a subsequent iteration, all copies of the key are evicted. This is already implicitly handled by the code shown in Listing 5.2 since subtables store only keys; the subtables don't track which specific pair the key belongs to. In the end, the cuckoo hash table will contain exactly one copy of each key.

With these modifications, further processing can be done to specialize the table.

Sets are basic data structures for storing lists of non-repeated elements; a typical operation is to check if some element is a member of the set. After building our hash table with a list of input keys, membership in the set can be checked by querying the hash table with the key.

Compacting hash tables extend sets, counting the number of unique keys in the input and assigning a unique ID to each, which effectively "compacts" the input. The compacting hash table consists of a hash table and a compacted list of the unique keys; this combination allows $O(1)$ translation between the IDs and the keys in both directions. It is most useful for memory-intensive applications where a lot of data is stored on a per-key basis: it determines how big an array is needed to store the data, and directs each unique key to a different location in the array. One typical scenario is first generating or selecting keys in parallel, and then counting the number of unique keys created and compacting them into an array.

When building a compacting hash table, *cuckoo*[ ] is initialized differently. Instead of filling the entire array with $\varnothing$, we initialize the half dedicated to storing values with zeros. Construction assigns all keys a value of one, so that only valid keys in the cuckoo hash tables have a non-zero value. Once *cuckoo*[ ] has been written out to global memory at the end of phase 2, we post-process it by doing a parallel prefix sum over the values. This creates a unique value from 0 to $k-1$ for every valid key stored in the hash table, where k is the number of unique keys. The compacted list can be created by writing out each key to a separate array into the slot identified by its ID.

Together, the array of unique keys and the hash table gives a mechanism for translating between keys and their indices in constant time, in both directions. We call this a *two-way index*. Of course, a two-way index could also be constructed by compacting a sorted list of keys. But while the sorted list version can also translate indices to keys in constant time, translating keys to indices would require a binary search.

Multi-value hash tables allow keys to have multiple values, which is represented in the input by pairs with the same key. These are useful for applications that require aggregating multiple values for the same key in a single structure. We can construct it using the shared memory atomic operations available on GPUs with compute capability 1.2.

The multi-value hash table stores an auxiliary array, *values*[ ], with all of the values rearranged so that all of a key's values are contiguous in memory; querying the hash table returns how many values a key has and the location of its first value in *values*[ ]. Specifically, our parallel multi-value construction produces a hash table in which a key k is associated with a count c_k of the number of values with key k, and an index i_k into a data table in which the values are stored in locations $i_k \ldots i_k + c_k - 1$.

Construction of multi-value hash tables is more complicated because we need to lay out the *values*[ ] array, but the process is analogous to the one used in Phase 1; the main difference is that it all occurs in shared memory within a single thread block (Figure 5.6).

Once the cuckoo hash table has successfully inserted all of the keys, the cuckoo hash table contains exactly one instance of each key. To count how many values

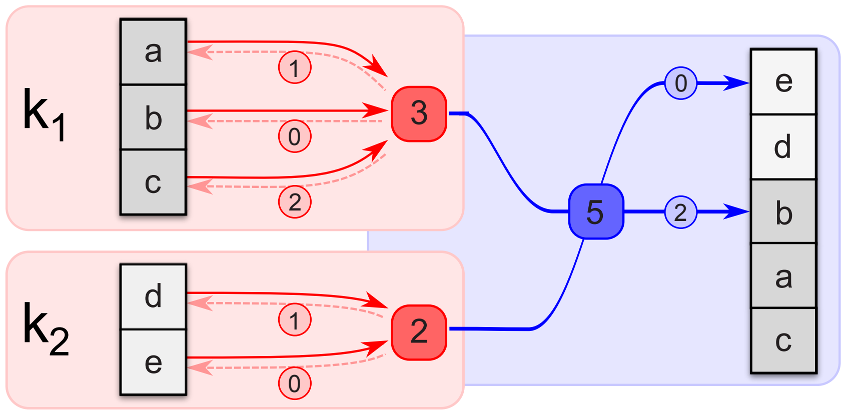


Figure 5.6. Laying out the *values*[ ] array for a multi-value hash table in shared memory, where the two keys k_1 and k_2 each have multiple values. Threads managing each value atomically increment a counter associated with its key (red), storing the previous counter value for an ordering among the key's other values. The counters themselves are then atomically added to another counter, indicating where each key's values should start in *values*[ ]. The ordering and the key counters can be added together to indicate where each value should be copied.

each key has, we allocate a shared memory array the same size as the cuckoo hash table and initialize it with zeros. Threads atomically increment the counter associated with each of their keys and store the previous counter, giving a relative ordering for each pair with the same key.

Next, a contiguous chunk of the *values*[ ] array is reserved for each unique key. While a parallel prefix sum can be used, we used the simpler approach of atomically adding the size counters to a separate counter and saving the counter's previous value as the index of the key's first value in *values*[ ]. Finally, a scattering operation is performed to write each value into the *values*[ ] array at the correct offset.

Afterward, each k is associated with its i_k and c_k in the cuckoo hash table, which then gets written to global memory.

5.4 Performance analysis

We use the same testing setup described in Section 2.3.2 to measure the memory usage, construction speed, and retrieval rates of the hash tables.

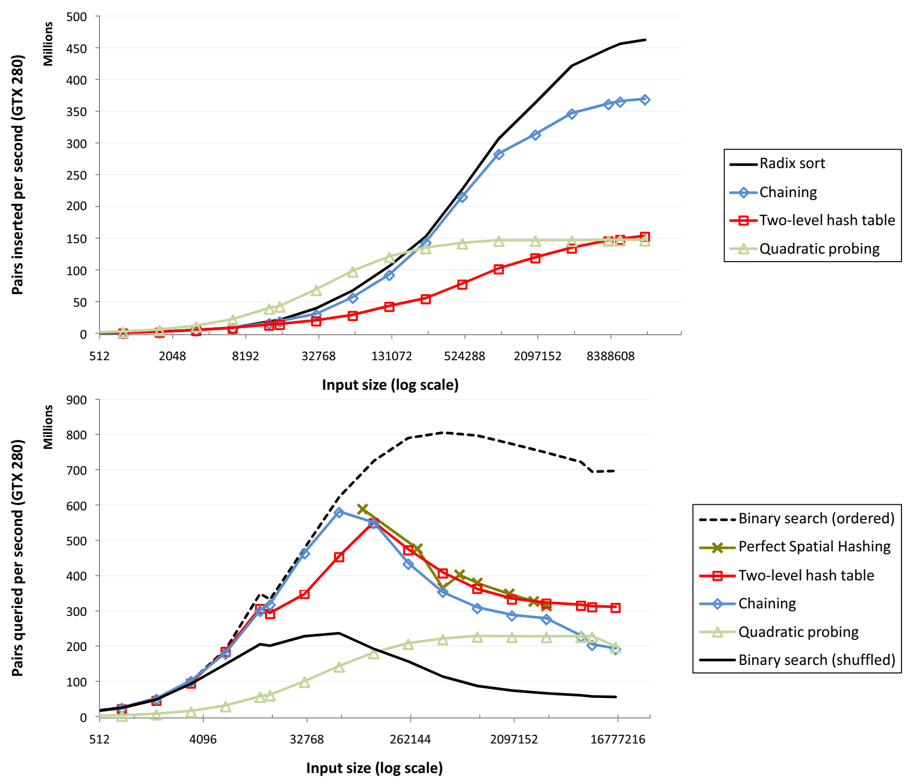


Figure 5.7. Effect of the input size on the insertion (top) and query (bottom) rates for the GTX 280.

5.4.1 Basic hash table

We compare the results with the quadratic probing and chaining implementations using the same amount of memory as our table using the parameter settings we described in Section 5.1.1. We also ran the tests on sorted arrays, which can hold all N items in N space but requires binary searches. We used Duane Merrill's key-value radix sort implementation [29]. As with previous chapters, the binary search with ordered queries serves as a guideline for performance; the binary search with shuffled queries represents the random-access that we are looking to compare our hash tables against.

Figure 5.7 shows the insertion and query rates for the data structures on a GTX 280. Radix sort has the highest overall insertion rate because it greatly reduces uncoalesced input and output by sorting in shared memory and then writing out data to global memory; our hash algorithm reads data in a perfectly coalesced fashion but the writes are highly uncoalesced. The retrieval rates are hampered by the $O(\lg N)$ binary searches, which grew increasingly worse for larger sets: memory reads were very uncoalesced and most threads required the full set of iterations to find their pairs, resulting in the lowest retrieval rate amongst all of the data structures. Chaining has similar insertion rates to radix sort because it uses one for construction, but the extra time and memory spent allows it to gain much better retrieval performance.

Our two-level cuckoo hash table has the slowest construction rates, with most of the effort spent counting the bucket sizes and building each of the cuckoo hashing tables. However, it has the highest query rates of all of the methods for hash tables containing more than 100K items because it limits how long probe sequences can be. Although quadratic probing is also an open addressing method, like cuckoo hashing, it has better insertion rates because it avoids the extra work we need to partition the items into buckets and limit the probe sequence lengths. The trade-off is its poor retrieval performance.

Also plotted are the query rates on the GTX 280 for hash tables created using our implementation of Perfect Spatial Hashing, which guarantees that any item can be found after a single probe[4]. Our space-optimized implementation uses a binary search to find the optimal size of the offset table, with a typical space usage of $1.16N$. Although it can answer queries by looking at only one location in the hash table, resulting in rates that match or beat ours, constructing it is a highly serial process. Insertion rates were omitted from the graph because they were built on the CPU as a preprocessing step, taking several orders of magnitude longer than the other methods.

Behavior is completely different on the newer GTX 470; Figure 5.8 shows the result of performing the same test. Again, the two-level cuckoo hash table has the worst insertion rates, but the disparity in performance is much more significant. The drop in the graph is a result of the counting step performed in Phase 1: when there are smaller input sizes, there are fewer buckets and their global memory

[4]The timings were taken using older GPU drivers, though the trends should be the same.

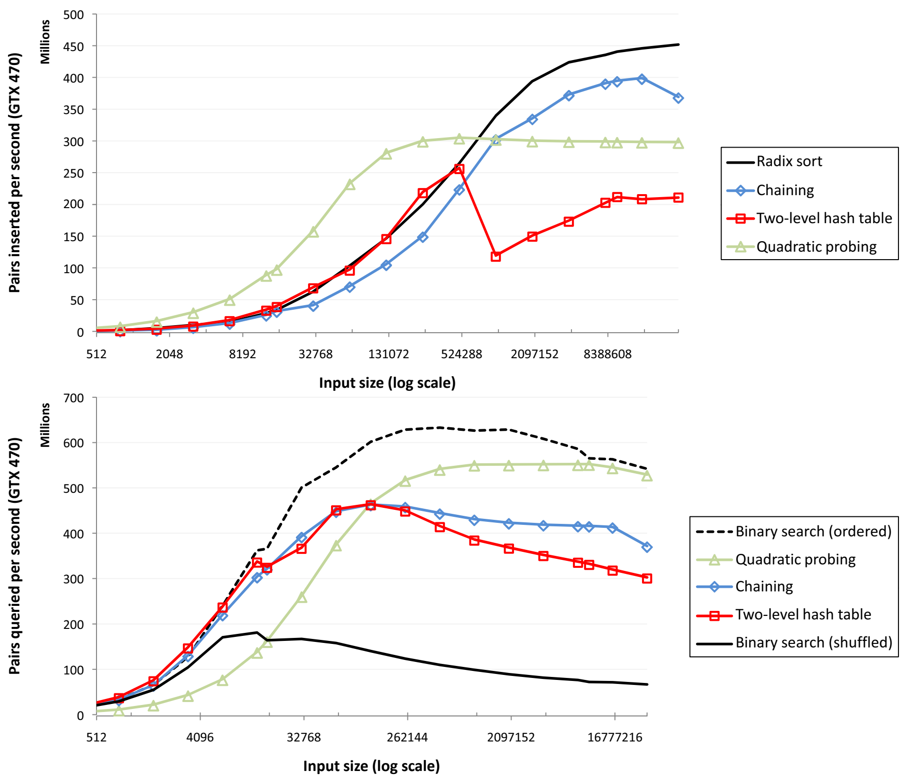


Figure 5.8. Effect of the input size on the insertion (top) and query (bottom) rates for the GTX 470.

counters can all fit within the L2 cache, allowing extremely fast atomic increments of the counters. Once the counters no longer fit in the cache, performance severely drops and never fully recovers. Moreover, the two-level hash table's query rates are sub-par compared to the others. Despite its advantage over binary search, it seems that it is a better idea to use one of the other hash tables on modern GPUs. Quadratic probing, for example, also performs atomic operations in global memory for insertion, but it has a good chance of caching many of the entries along its probe sequences, leading to higher construction and retrieval rates.

This is further shown in Figure 5.9, which shows the effects of querying the structures with keys that are absent in the data structure. Each test case sends the same number of queries, but with an increasingly higher percentage of absent keys. Queries for absent keys cause our hash table to use the worst-case number of probes, but our overall retrieval rates are only slightly increased; even in the worst

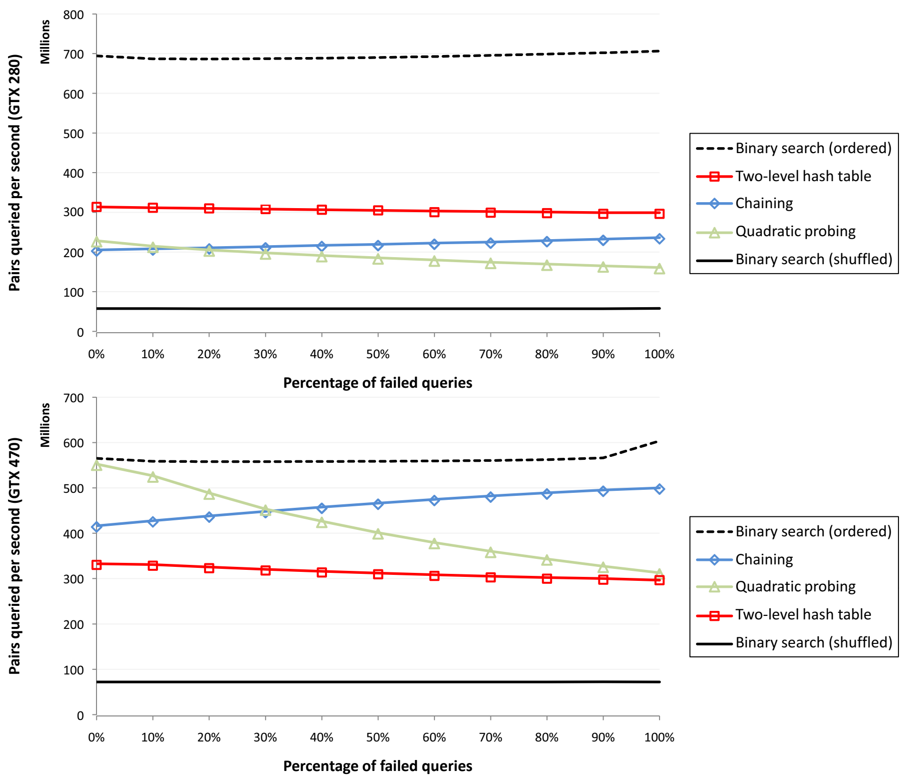


Figure 5.9. Comparison between the two-level cuckoo hashing table and other methods when searching for increasing percentages of keys that are not in the table. The hash tables all use the same amount of memory ($1.42N$) to hold 10 million items.

case, we need to check only three locations. Binary search times actually decrease slightly because the searches can terminate faster.

On the GTX 280, our two-level hash table consistently outperforms the other methods, but it apparently gains no performance increase with the move to the GTX 470 and ends up performing consistently worse. We guess that this is because it cannot take advantage of the cache like the other methods. Using this much storage, chaining and quadratic probing require an average of around 3 probes to answer queries, making it extremely likely for a cache line to catch all of the probed locations.

Gaining any sort of advantage in retrievals for the GTX 470 comes only when the table is more compact, which is where the other hash tables have trouble. This requires increasing the bucket size and using higher target loads, which results in

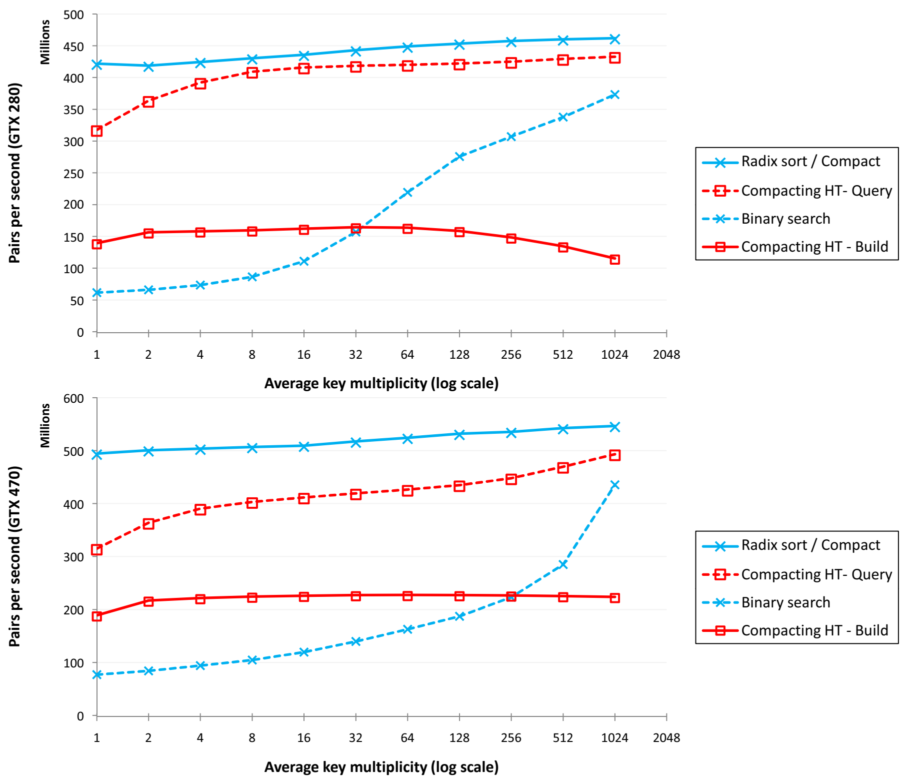


Figure 5.10. Timings examining the effect of repeated keys in the input when building a compacting hash table containing 10M items. The hash table is compared against an equivalent structure based around a radix sort. Each key was queried once for every time the key appeared in the input.

a large decrease in the construction rates. These are unlikely to be worth the trade-off unless the table is heavily reused, though.

5.4.2 Hash table specializations

The performance of the compacting hash tables (Figure 5.10) and multi-value hash tables (Figure 5.11) was tested on datasets of a fixed size, but with an increasing average number of times a key is repeated. We compared against an equivalent structure that could be created by compacting a sorted list of the pairs. Hash table construction initially speeds up a little with the modified algorithm, and then greatly increases at higher multiplicities. This is mainly due to the extra atomic operations required; the compacting hash is less affected because it does not use the extra atomics. It is always slower to construct than the alternative, but the

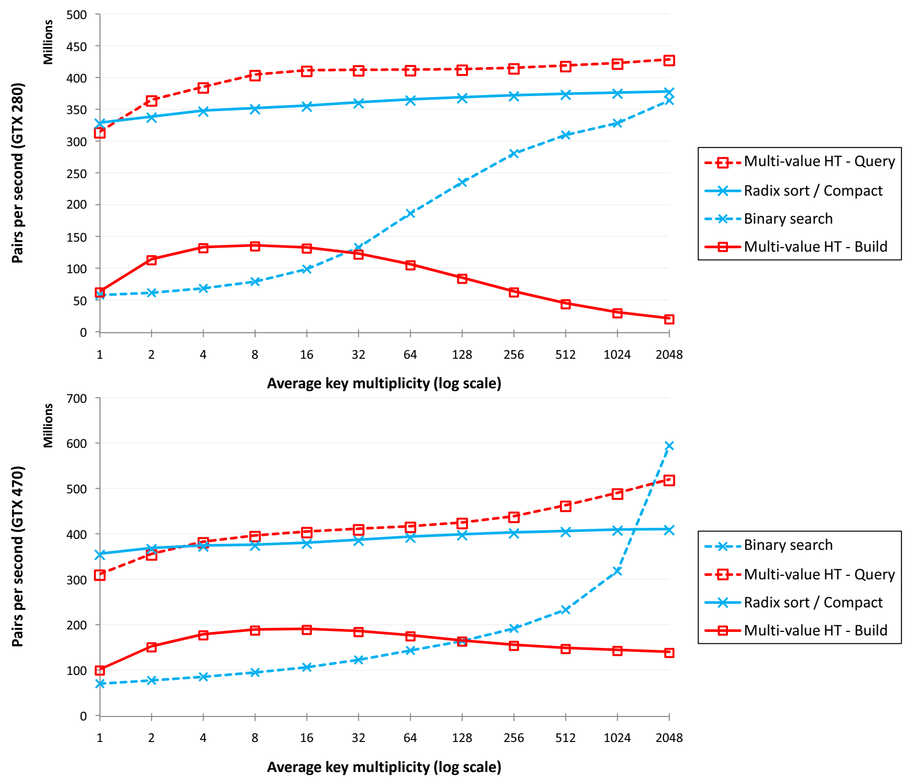


Figure 5.11. Timings examining the effect of repeated keys in the input when building a multi-value hash table containing 10M items. The hash table is compared against an equivalent structure based around a radix sort. Each key was queried once for every time the key appeared in the input.

significantly faster retrievals can be worth the extra work for smaller multiplicities, especially if the table is heavily queried after being built.

5.5 Limitations and trade-offs

The previous sections described our two-level hash table and the parameters we believe strike the best balance between the construction, retrieval, and space usage constraints. However, the algorithm has some obvious issues.

Recent GPU architectures negate most of the benefits of our two-level structure: while our hash table outperforms the other hashing methods on the GTX 280, the advancements in the 400 series allow hash tables that were previously inefficient to become more efficient.

Restarts during Phase 1 were uncommon, with hash tables usually able to rebuild themselves successfully with a single new $g(k)$. For our basic hash table, a restart was required for $\frac{22}{25000}$ trial runs (0.088%) for 1 million random items, and $\frac{125}{25000}$ trial runs (0.5%) for 5 million random items. In both of these cases, construction time increased by 50%, but they were still within the constraints needed for interactive applications. It is possible to decrease restarts by lowering the number of average number of items per bin, at the expense of further space overhead.

Memory usage of the basic hash table is difficult to shrink. The most obvious way to do so would be to increase the bucket capacity and target load factor; this had a minor effect on our retrieval timings, but increased the amount of time spent in construction building the denser cuckoo hash tables. Even better space usage could come from a dynamic cuckoo hash table size, which allocates tables that are the exact size required for each bucket. The problem is that extra information must be stored about each cuckoo hash table's locations and size, increasing the amount of time and effort required to build and query the table. Preliminary results showed that retrieval rates dropped by about 25%, reflecting the extra memory accesses required.

Subtables inflate probe sequence lengths because their size restricts the number of items that can be found after a single probe. In contrast, regular open addressing methods often allow at least half the items to be found with a single probe.

Specializations of the hash table amplify these issues as the number of unique keys in the input decreases. Because we can't easily determine how many unique keys there are in the input until we attempt to build the cuckoo hash tables, we end up overestimating the number of bins required and end up with a sparser structure. As a side effect, expensive restarts become more uncommon: because the amount of bins allocated depends on the total number of items and not the number of unique keys, it becomes more difficult to overfill a bucket.

For space-constrained applications, it could be better to use a combination of sorting (to build a compacted list of input) and our basic hash table. This way, the hash table can allocate just enough memory to store the unique keys.

5.6 Summary

We've demonstrated a hash table construction that can somewhat deal with the shortcomings of chaining and open addressing, namely the uncapped number of probes required to find items in the table. We also introduced a way of extending the algorithm to allow different hash table specializations to be built. On the GTX 280, our data structure has much higher rates and is more robust at handling hard to answer queries than the other hash tables we've described, at the cost of a more complicated construction process. This is a useful trade for applications that will be building the table and querying it very heavily before discarding it.

However, the algorithm doesn't adapt well to the newer architectures of modern GPUs. Although its guaranteed $O(1)$ retrievals make it robust against queries that fail, it doesn't help when the queries are slower to perform on the GTX 470. In the next chapter, we explore another way of using cuckoo hashing that is much more flexible and has better performance than the methods we have explored so far.

Chapter 6

Cuckoo hashing with stashes

While our two-level cuckoo hashing table attempts to reduce the effect of a failure by building many smaller cuckoo hash tables, this is ultimately very costly. Moreover, its performance does not benefit from the newer GPU architecture like our other hash table implementations: by its nature, cuckoo hashing must distribute items through the table to reduce contention for each slot, preventing it from taking advantage of the cache.

To improve performance, we must instead focus on minimizing the number of memory accesses performed by the algorithm rather than improving spatial locality: construction and retrieval performance are limited almost entirely by the time used for these accesses because they are almost always uncoalesced. This is already handled in retrievals because cuckoo hashing guarantees a constant worst-case bound on the number of accesses per lookup, but the construction algorithm is less clear.

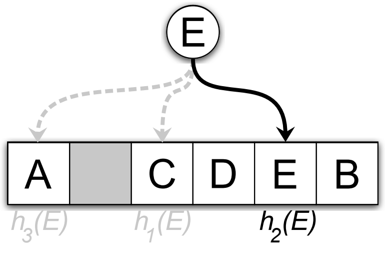


Figure 6.1. Instead of using subtables, this version of cuckoo hashing uses one big table, with each of the hash functions $h_i(k)$ mapping into every slot.

We utilize the lessons learned from our other hash tables to produce a parallel cuckoo hashing algorithm that avoids most of our previous version's issues. Results from the open addressing schemes indicate that the half of initial items can usually be inserted into the table after a single probe. Although the maximum number of probes required to insert an item can be high, the majority of threads continue finding slots relatively quickly. Because the cost of a harder insertion is amortized with the cost of the cheaper insertions, building the cuckoo hash table directly in global rather than shared memory is a viable option. This obviates the need for using a two-tiered scheme and all of the extra complexity it creates, leading to a faster, streamlined construction.

In this chapter, we introduce a different method of parallelized cuckoo hashing using one big table in global memory, where the hash functions can map items to any location in the table (Figure 6.1). Although this exposes the possibility of a complete rebuild when an item fails to be inserted, we discuss methods for reducing the chances of this happening to arbitrarily low percentages, including using a small secondary hash table to catch any items that would trigger a failure.

We give an overview of our algorithm in Section 6.1, then go into the details of the construction of a basic hash table that stores a single value for each key in Section 6.2. We briefly describe how to extend our construction algorithm for more specialized hash tables in Section 6.3, then analyze the hash table's performance in Section 6.4 and limitations in Section 6.5. We then summarize our findings in Section 6.6.

6.1 Single-level cuckoo hashing

The approach we use to parallelize cuckoo hashing differs in significant ways from the method we introduced in the previous chapter:

- We build the hash table in global memory rather than in shared memory. Getting into a position where using shared memory was possible required a lot of extra work and memory traffic to move items to the right locations.

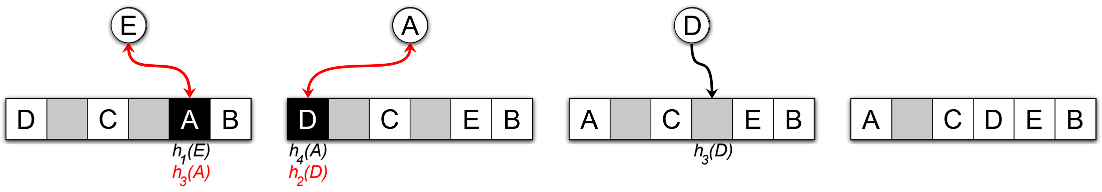


Figure 6.2. Insertion into the table has each thread managing a different item after every insertion attempt until it finds an empty slot. This example shows a thread beginning with the key E (left) and repeatedly *swapping* instead of *evicting* keys in the table, iterating until it finds a slot or gives up (right).

- We stop using subtables and instead allocate all of the hash table's slots to one big table. Subtables limit the number of items that could be found with fewer probes because the number of slots available in the subtables is artificially restricted.

- Rather than have a thread attempt to repeatedly insert the same item, we instead have the thread *atomically swap* the item it is holding with the item in the slot it wants. This relies on efficient global memory atomic operation performance, but it essentially allows threads to work independently: although they still have to wait for the rest of the thread block to finish inserting their items, they don't need to explicitly synchronize after every insertion.

Like our previous version, each of the hash functions is used in round-robin order: items evicted from slot $h_i(k)$ is reinserted into slot $h_{i+1}(k)$. When moved from its final slot, or when being inserted for the first time, the item uses $h_1(k)$ (Figure 6.2). While the serial algorithm would normally check all of an item's locations or randomly select one, our method can reduce the average number of probes needed for a retrieval by forcing items to prefer their earlier slots.

At every iteration, each thread uses an atomic operation to swap its current item with the item in the slot indicated by the current hash function. If the slot is empty (represented by the entry $\varnothing$[1]), then the thread immediately terminates. If not, the thread must reinsert the item that it removed using the new item's next

[1] $\varnothing$ = (`0xffffffff`, `0`)

hash function. To determine which hash function was used, we recompute the value of all the hash functions and compare against the index of the slot it was evicted from; this prevents incurring extra memory accesses to store this information.

This process continues until either an empty slot is found or too many iterations have been taken, at which point we assume that the process will never terminate. Interestingly, Kirsch et al. [22] found that the number of items hitting this point is consistently small (typically fewer than four items); our own experiments mirror their results for cuckoo hash tables several orders of magnitude larger than they tested. Rather than immediately rebuilding the entire hash table from scratch, they first attempt to store all of the items that could not be inserted into a small secondary structure called a *stash*. It shares many similarities with a CPU's victim cache, which catches items evicted from the main cache.

A rebuild is avoided if the stash manages to store all of the items it receives; when the stash is large enough, the chance of a rebuild is reduced to almost nil. The trade-off is that both the main hash table and the stash have to be checked when answering a query. We use them for our cuckoo hash tables because the cost of a rebuild far outweighs the cost of the additional probes. If any item fails to be inserted into the stash, however, we must declare failure and trigger a complete rebuild.

With our base configuration, the hash table uses $1.25N$ space for N items and allows each item four choices, guaranteeing that any item can be found after at most four probes (five if our stash is activated).

6.1.1 Parameters

Our implementation has many parameters that can be tweaked to affect the algorithm's performance, reducing memory accesses, reducing the storage required for the table, or even driving the failure rate arbitrarily low.

The hash functions we use are randomly generated and take the form:

$$h_i(k) = (f(a_i, k) + b_i) \bmod p \bmod S_T$$

See Section 3.1.2 for a discussion on these. The hash functions we use need to be fast to compute because we repeatedly compute them whenever a thread swaps keys; previous hash tables were able to compute the functions once and cache the results. We had good success with two different families of functions.

A regular linear polynomial worked well for our datasets. Rather than aim for a 2-universal family of functions, which requires using 64-bit values and slower computations, we aim lower and limit the computations to 32-bit values. S_T is the number of slots in the hash table, while we set $p = 334,214,459$. Each $h_i(k)$ uses its own randomly-generated constants $a_i \in [1, p)$ and $b_i \in [0, p)$. For many datasets, we found that we had better distributions by replacing the multiplication with an XOR operation and using $p = 4,294,967,291$; your mileage may vary, but the results we present later in the chapter are based on these functions. While both of these families produce a set of weaker hash functions, they do limit the number of slots under heavy contention and allow the table to be built successfully in most of our trials.

The number of hash functions is also very important. Using more functions allows each key more choices for insertion, making it easier to build the table and decreasing the construction time. However, this increases the average number of probes required to find an item in the table, increasing retrieval times. This is more obvious when querying for items not in the structure. Using four hash functions yields a good balance between all three metrics, though the hash table can easily be constructed with other numbers of hash functions.

The size of the table affects how easily the construction algorithm can find a conflict-free configuration of the items. Given a fixed input dataset, bigger tables have less contention for each slot, reducing both the average length of an eviction chain and the average number of probes required to retrieve an item.

When memory usage is important, using more hash functions makes it easier to pack more items into a smaller table. Theoretical bounds on the minimum table size for differing numbers of hash functions were calculated by Dietzfelbinger et al. [10], but we found the smallest practical table sizes with our hash functions were slightly larger at $\sim 2.1N$, $1.1N$, $1.03N$, and $1.02N$ for two through five hash functions, respectively.

The advantage of cuckoo hashing over the other hashing methods becomes more apparent as the table size approaches the minimum, but it becomes increasingly difficult to build the table. We suggest using $1.25N$ space to balance out the construction and retrieval rates, as well as limit the number of restarts that occur.

The maximum number of iterations tells the algorithm when the insertion process is unlikely to terminate. Threads usually finish well before reaching a chain of this length, but the limit must be set carefully as it is the only way for the algorithm to terminate with a bad set of hash functions. Setting this number too high greatly increases the amount of time required to acknowledge that a failure has occurred, but setting it too low will cause the algorithm to declare failure too early.

There is no theoretical value for the limit, but it is dependent on both the table size and the number of items being inserted: cramming a large number of items into a small hash table greatly increases slot contention, resulting in longer eviction chains. For four hash functions and $1.25N$ space, we let threads follow chains of at most $7\lg(N)$ evictions. For other table sizes, we use an empirically determined formula to set the number of iterations allowed.

Empirically, we had a thread fail to insert a key into a cuckoo hash table for 10M items only in only 3 out of 10000 trials, where we built hash tables with different table sizes. However, we did see failures more frequently for small tables containing 10K items or less, which are more difficult for our algorithm to handle.

The stash has to be designed to catch as many of the items as possible that fail to be inserted into the hash table. The implementation proposed by Dietzfelbinger et al. uses a constant-sized array, which can catch a limited number of items that fail to be inserted into the main table. This is a viable strategy for small stashes, but keep in mind that a stash of size 4 will essentially double the number of global memory accesses required for a cuckoo hash table using 4 hash functions.

We instead use a small secondary hash table as the stash. Each item stash may be located in exactly one location, determined by a single hash function. As shown by Fredman et al. [14], a hash table composed of k^2 slots has a high probability of storing k items in a collision-free manner using a random hash function; we use this approach and set the size of the stash to 101 slots, which has a high probability of holding up to 10 items without collisions. Our stash is associated with its own random hash function, which directs each item to exactly one location in the table. We trigger a full rebuild of the entire table when two items collide in the stash[2].

Because the number of items failing to insert themselves into the main hash table was almost always less than 5, regardless of the number of input items, the stash was nearly always able to prevent the rebuild and dramatically reduce the failure rate. While it does add an extra memory access for a bad retrieval, the very small performance hit is worth the prevention of a complete rebuild.

6.2 Building and querying a basic hash table

We begin by discussing the implementation of a basic hash table, which stores a single value for each unique key. The input consists of N pairs of 32-bit keys and values, where none of the keys are repeated. The value could represent indices into another structure, allowing a key to reference more than just 32 bits of data; we briefly discuss this in Section 6.3.

6.2.1 Construction

The goal of construction is to produce a data structure consisting of five things:

- *cuckoo*[ ], the cuckoo hash table itself.
- S_T, the number of slots in the hash table.

[2] The stash could instead be rebuilt with a new hash function, but a lot of effort would be expended to find the items currently stored in the stash and move them over. For the GTX 470, another alternative is to use a linear probing hash table containing 16 slots; a stash this small is likely to be stored entirely in the cache.

Listing 6.1. Code snippet for inserting a new item using four hash functions.

```
typedef unsigned long long Entry;
#define get_key(entry)          ((unsigned)((entry) >> 32))

// Load up the key-value pair into a 64-bit entry.
unsigned key   = keys[thread_index];
unsigned value = values[thread_index];
Entry    entry = (((Entry)key) << 32) + value

// Repeat the insertion process while the thread still has an item.
unsigned location = hash_function_1(key);
for (int its = 0; its < max_iterations; its++) {
  // Insert the new item and check for an eviction.
  entry = atomicExch(&table[location], entry);
  key   = get_key(entry);
  if (key == KEY_EMPTY) return true;

  // If an item was evicted, figure out where to reinsert the entry.
  unsigned location_1 = hash_function_1(key);
  unsigned location_2 = hash_function_2(key);
  unsigned location_3 = hash_function_3(key);
  unsigned location_4 = hash_function_4(key);
       if (location == location_1) location = location_2;
  else if (location == location_2) location = location_3;
  else if (location == location_3) location = location_4;
  else                             location = location_1;
};

// Try the stash.  It succeeds if the stash slot it needs is empty.
unsigned slot           = stash_hash_function(key);
Entry replaced_entry    = atomicCAS(stash + slot, SLOT_EMPTY, entry);
*stash_was_used         = 1;
return (replaced_entry == SLOT_EMPTY);
```

- *stash*[ ], the secondary data structure for catching items that fail to be inserted in *cuckoo*[ ].
- The constants for all i hash functions, a_i and b_i.
- A flag indicating whether the stash had to be used. If this is off, the number of probes for a retrieval is unaffected.

We begin by allocating an array of S_T 64-bit slots for *cuckoo*[ ] and an array containing 101 64-bit slots for *stash*[ ]. Each slot holds a key-value pair: a key and its value are stored in the upper and lower 32 bits of the slot, respectively. Moving them together in this manner allows us to use a single atomic swap to move them around the table; storing them separately incurs extra work, either to ensure that they remain together during the insertion process or to insert the value into the hash table after all keys have been placed.

Construction may require several attempts if the algorithm cannot find a conflict-free configuration of the items, so we have to reinitialize the hash table at the start of every attempt. This involves filling both *cuckoo*[ ] and *stash*[ ] with the special entry $\varnothing$, which alerts threads that they have found an empty slot. We also generate random constants for all of the hash functions we are using and reset the stash flag. The constants and the flag are passed into the retrieval kernel directly as arguments, so they do not incur extra global memory accesses.

The cuckoo hashing algorithm is then performed by a single kernel (Figure 6.1). The kernel is launched with N threads, each managing one item at a time from the input. Recall that threads complete their work when they successfully place their item (or the item(s) displaced by their item), but a thread block will not complete until all of its threads are done. Thus we choose a relatively small thread block size – 64 threads – that minimizes the number of threads within a block kept alive by a single thread's long eviction chain.

Using `atomicExch()` operations, each thread repeatedly swaps its current key-value pair with the contents of the slot determined by the current hash function.

The atomic operations guarantee that no items are lost when threads simultaneously write into the same slot. If the swap returned an $\varnothing$ item, the thread declares success and stops.

Otherwise, the thread must reinsert the item it evicted. Because hash functions are used in round-robin order, the thread must figure out which hash function was previously used to insert it. It calculates the value of $h_i(k)$ for all of the hash functions, then checks which of those functions points to the slot it was evicted from. The evicted item is then reinserted using the next hash function in the series. Alternatives to this approach would store the index of the hash function last used to store the item, but these all have different drawbacks and usually require extra memory traffic that we want to avoid.

One potential issue with this is that our code snippet short-circuits after finding the earliest match, preventing the item from utilizing its full set of assigned slots when multiple hash functions return the same value. If $h_2(k) == h_3(k)$, for example, the thread will assume that the item was evicted using $h_2(k)$ and reinsert it using $h_3(k)$, permanently locking the item into the slot. It can be addressed either by advancing to the next function that doesn't point to the same location, or by always randomly picking a hash function to use. We just choose to ignore the problem because it rarely happens; it usually occurred when the hash table contained only a few hundred slots and didn't cause any issues. An even rarer occurrence would lock two keys to the same slot, but we didn't encounter this in our tests.

If a thread hits the maximum eviction chain length, it then gets one attempt to find an empty slot in the stash using its hash function. If it succeeds, a global flag is set to indicate that the stash is being used and must be checked during retrievals. Otherwise, we declare failure and start from scratch.

After being built, items can continue to be added to the cuckoo hash table by following the same procedure, but care must be taken to resize the hash table when necessary, which requires rebuilding the whole hash table from scratch.

6.2.2 Retrieval

Retrieval of a query key can be performed on the cuckoo hash table by first checking all of the locations identified by the hash functions (Figure 6.2). Each location is checked in order, but a thread short-circuits if it either finds the key early or encounters an empty slot in the table. Because the hash functions are all used in order, it is impossible for a key to be located in any of the slots following an empty slot – otherwise, the query key would have used it[3]. If the query fails to be found in the main cuckoo hash table, a "not found" value is returned, signalling the thread to check the stash (if it is active) before giving up entirely.

6.3 Specializing the hash table

Our single-level cuckoo hash table can be specialized in the same way as the two-level cuckoo hash table (Section 5.3). In this section, we briefly describe how to implement the equivalent structures.

Sets can be implemented by changing the construction to eliminate all copies of a key that are encountered. The input consists of 32-bit keys, which may or may not be unique. Correspondingly, each slot of the hash table is modified to hold only 32-bit entries. Similarly to the basic hash table construction, each thread manages a single key and uses 32-bit `atomicExch()` operations to swap their key with the contents of the table. The stopping condition is modified so that a thread stops upon either finding an empty slot or exchanging its key with another copy that was previously stored in the table. Once all threads have stopped, multiple copies of the same key may still be inserted in the table, so a cleanup phase must occur afterward to erase all copies of a key except the first.

Compacting hash tables can be built by further altering the set construction algorithm. It returns to using 64-bit slots, which allows pairing keys with their unique IDs in the hash table. During the cleanup phase, the first copy of each key in the table is assigned a value of one, with all other slots in the table filled with $\varnothing$,

[3]Allowing deletions from the structure requires disabling this to avoid erroneously claiming that a query key does not exist in the table.

Listing 6.2. Code snippet for retrieving an item from the main table using four hash functions. Failing to find the key here triggers a stash check, if it is active.

```
#define get_value(entry) ((unsigned)((entry) & 0xffffffff))
const Entry kEntryNotFound = (Entry)(-1);

// Compute all possible locations for the key.
unsigned location_1 = hash_function_1(key);
unsigned location_2 = hash_function_2(key);
unsigned location_3 = hash_function_3(key);
unsigned location_4 = hash_function_4(key);

// Keep checking locations until the key is found, all slots
// are checked, or if an empty slot is found.
Entry entry;
if (get_key(entry = cuckoo[location_1]) != key) {
  if (get_key(entry) == kNotFound)
    return kEntryNotFound;

  if (get_key(entry = cuckoo[location_2]) != key) {
    if (get_key(entry) == kNotFound)
      return kEntryNotFound;

    if (get_key(entry = cuckoo[location_3]) != key) {
      if (get_key(entry) == kNotFound)
        return kEntryNotFound;

      if (get_key(entry = cuckoo[location_4]) != key)
        return kEntryNotFound;
    }
  }
}

return get_value(entry);
```

Algorithm 6.4 Multi-value hash table construction

```
1: radix sort the input key and value arrays
2: determine what and where the unique keys are in the sorted key array
3: determine how many values each key has by differencing the locations of con-
   secutive unique keys
4: perform a prefix-sum to assign each key a unique ID
5: create an auxiliary array keyInfo[ ] to store the location and number of each
   key's values
6: create a basic hash table using the unique keys and their IDs
```

effectively identifying which slots have unique keys. A prefix-sum is then performed over the values, creating a unique value from 0 to $k-1$ for every unique key stored in the hash table, where k is the number of unique keys. We use another pass to interleave the keys with their values in the table, and finally create the compacted list by writing out each key to a separate array into the slot identified by its ID.

Multi-value hash tables are constructed in a completely different manner than they were for the two-level hash table. As input, a multi-value hash table takes in one array of keys and one array of values, where each of a key's values is represented as different key-value pairs with the same key. Intuitively, the main goal during construction is to boil down the input to a single value for each key so that it can be stored using a basic hash table. The value here is a unique ID that indexes into an auxiliary array that stores more information about the key. This process can be adapted to store data other than 32-bit integers.

The multi-value hash table stores two auxiliary arrays: *values*[ ], which stores each key's values contiguously, and *keyInfo*[ ], which stores the location of every key's values and the number of values each has. Construction of the table is detailed in Algorithm 6.4; the majority of the algorithm is spent pre-processing the input. We first use a radix sort on the input pairs to rearrange the data, placing all copies of the same key adjacent in memory. Additionally, all values of each key end up in a contiguous location within their array; this array is stored as *values*[ ].

Next, we examine the sorted keys array to find the indices of the first instance of each key; these indices also indicate where in *values*[ ] each key's values start, and can be differenced to determine how many values each key has. This information is stored in *keyInfo*[ ] as uint2s.

We then create a scratch array of flags indicating whether each key was unique, then perform a prefix-sum to assign each key a unique ID. These unique IDs are used to index into *keyInfo*[ ] and to compact down the unique keys. The compacted list and their IDs serve as the input for the creation of a cuckoo hash table using the basic algorithm.

Querying the structure with a valid key first performs a retrieval using the hash table. The result is then used as an index into *keyInfo*[ ] to return information about the key's values.

6.4 Analysis

We use the same testing rig described in Section 2.3.2, testing on both the GTX 280 and GTX 470. The results we report consist entirely of randomized key-value pairs.

6.4.1 Parameter effects on the basic hash table

We begin by examining the effect of the input size on the hash table when it is built with different table sizes, effectively changing how many of the slots are occupied. Insertion rates are displayed in Figure 6.3; the trends for both GPUs are the same, ramping up and plateauing. This suggests that the rate of insertion on both GPUs scales linearly with the input size. Unsurprisingly, building a table with tighter space constraints is very hard to do; inserting pairs into the hash table using $1.05N$ space instead of $2N$ space drops performance by half.

Query rates are shown in Figure 6.4. The downward trend in the rates as the table size increases can be easily explained by the increasing average number of probes required to find an item as the table shrinks: bigger tables have more empty slots and can short-circuit the retrieval more easily. We see that the rates again

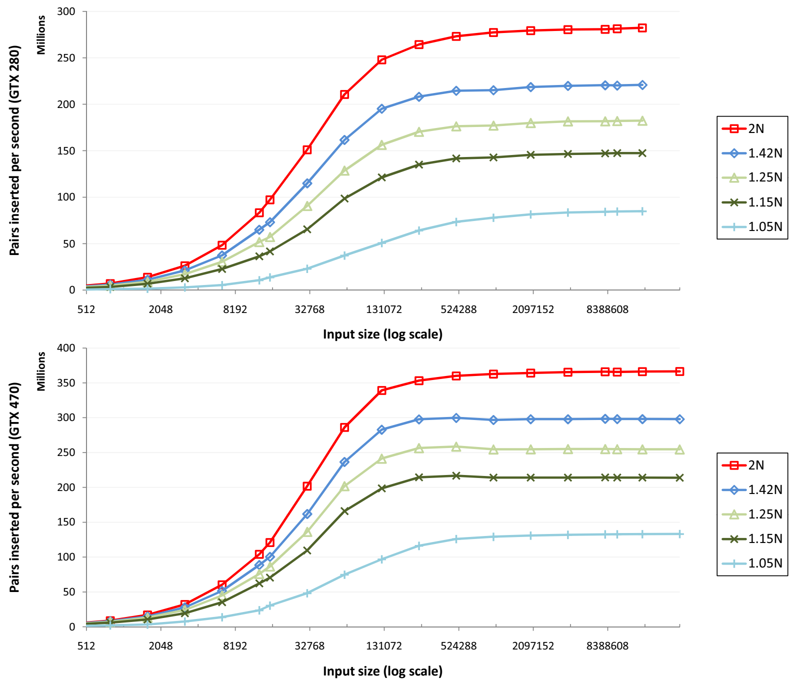


Figure 6.3. Effect of the input size on insertion rates for cuckoo hash tables with different space constraints. All of the hash tables used four functions; higher rates are better.

plateau after a certain point, with the rates being higher on the 470. We again see the same inexplicable behavior on the GTX 280 that we saw for our chaining implementation, where the rates peaked on the left and sharply dropped off on the right for larger input sizes. This is strange because the two hash tables have completely different memory access patterns.

As expected, cuckoo hashing is highly robust against answering bad queries (Figure 6.5) and its performance degrades linearly as the average number of probes approaches the maximum required to answer a query; in this case, 4 probes to check each of the slots indicated by the hash function, as well as a 5th probe into the stash if invoked. The glitch in the graph for the GTX 280 for a table of size $2N$ corresponds to the performance drop on in Figure 6.4.

Figure 6.6 shows the effect of playing with the table's size and the number of functions used to build the table; both of these are closely related. For construction, hash tables built using any given number of functions suffer larger and larger performance penalties as the table size approaches the minimum allowed by cuckoo

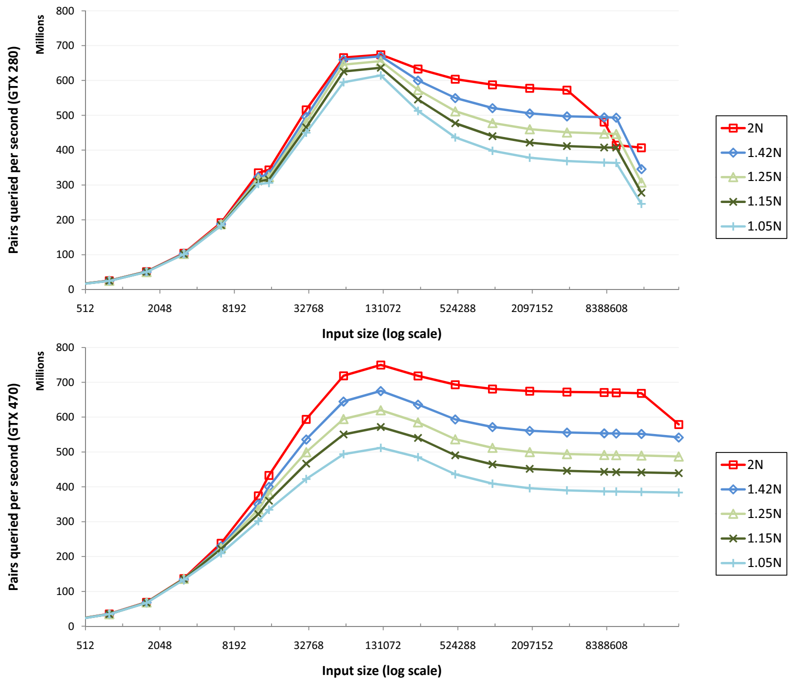


Figure 6.4. Effect of the input size on query rates for cuckoo hash tables with different space constraints. All of the hash tables used four functions; higher rates are better.

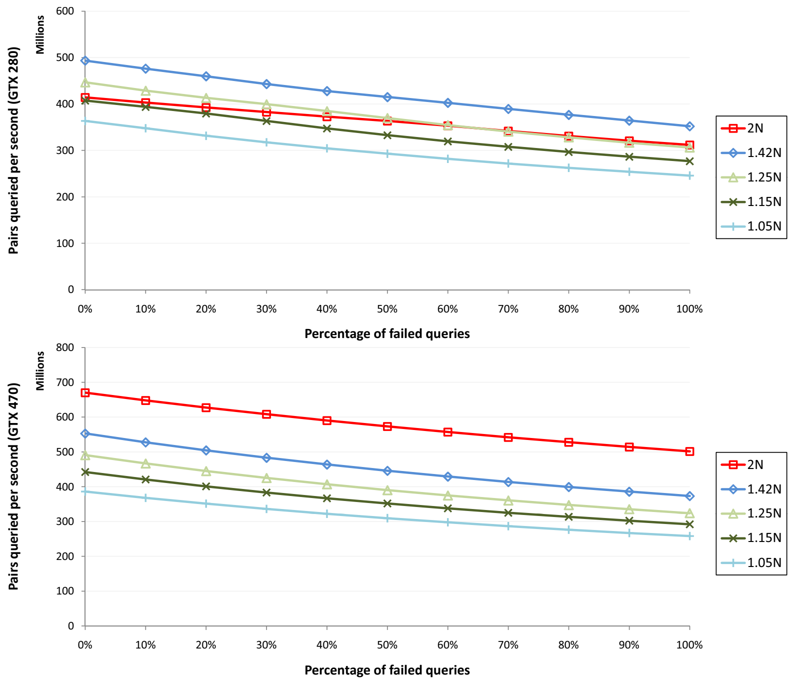


Figure 6.5. Effect of searching for query keys not in the hash table. All of the tables used four functions; higher rates are better. The $2N$ case for the GTX 280 corresponds to the sharp decrease in performance for large hash tables on the 280, but the trends are otherwise the same.

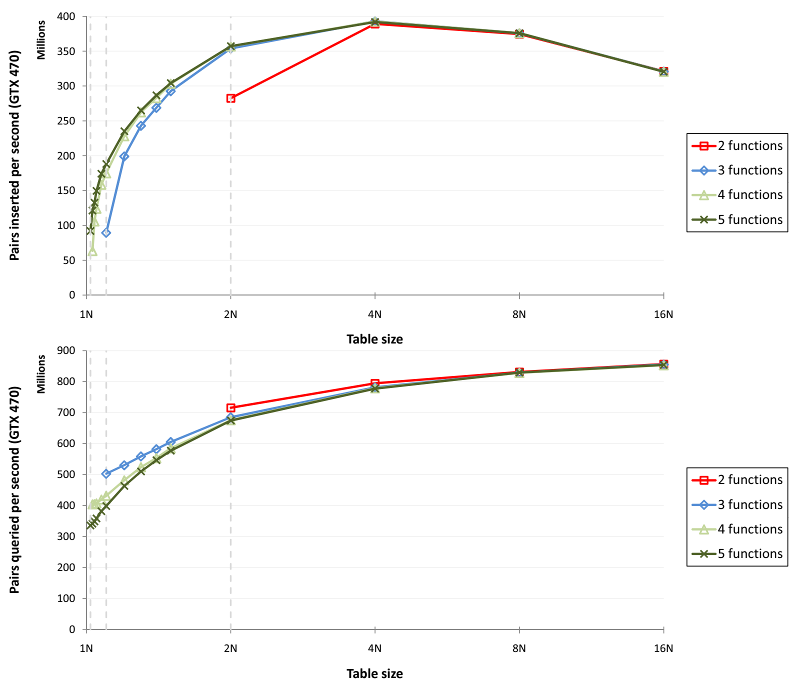


Figure 6.6. Effect of hash functions and table sizes on construction (top) and retrieval (bottom) rates for a hash table containing 1,000,000 items on the GTX 470 (trends on the GTX 280 were similar). Minimum table sizes for each number of functions are represented by vertical dashed lines. The construction rates increase when the table size approaches $4N$, but decrease afterward because of the time required to initialize the table.

hashing. Retrieval rates also drop because the average number of probes required to find an item increases as the number of empty slots shrinks.

On the other end of the spectrum, the fastest construction time is achieved using around $4N$ slots, regardless of the number of input items. Insertion rates start decreasing after this point because the amount of time required to initialize the table outweighs the time required to actually perform cuckoo hashing.

The fastest general construction speed is attained when using five hash functions because it offers items more locations to choose from, though the benefit over using four hash functions is minimal. It also has the slowest retrieval rates when compared to the other options. However, it may still be practical when minimizing memory usage is the most important factor for an application, since it can produce the most compact cuckoo hash tables.

6.4.2 Comparisons between the data structures

In this subsection, we perform comparisons of the main data structures we have discussed so far, which includes quadratic probing, chaining, both cuckoo hashing variants, and a radix sort/binary search combination. We test three different hash table sizes ($1.05N$, $1.42N$, and $2.0N$), representing three possible choices that an application could make when using the data structures. For each case, we show the results of testing the data structures on both the GTX 280 and 470, showing how well construction and retrievals scale with input size, as well as how robust each method is for dealing with queries that cannot be found in the table.

General trends show that aiming for a more compact hash table often increases the time taken to build or query the structure, but sacrificing memory often allows marked improvements in performance. As stated in previous chapters, rates for the binary search with ordered queries are shown as a guideline for performance; binary searching with shuffled queries represents the random access that we are actually looking to compare our hash tables against. When we talk about results for a plain "binary search", we specifically mean the shuffled query case.

6.4.2.1 Hash tables using $1.05N$ space

Figures 6.7 and 6.8 shows performance profiles for both the GTX 280 and GTX 470 for tables of size $1.05N$. We omit results for the two-level hash table because producing a hash table this small is difficult using that scheme: it would require allocating many large buckets and filling them as close to the bucket's capacity as possible, which makes it highly likely that bucket will overfill and require a rebuild.

As we learned in Chapter 3, quadratic probing just does not perform well with tables this small: the limited number of slots it has available makes it difficult for the algorithm to terminate both its insertions and retrievals. This results in the poor insertion and retrieval performance on both GPUs – for smaller input sizes, it even performs worse than a binary search. Although it performs better than the binary search for larger input sizes when all of the queries can be found, its retrieval rate quickly drops when only 20% fail.

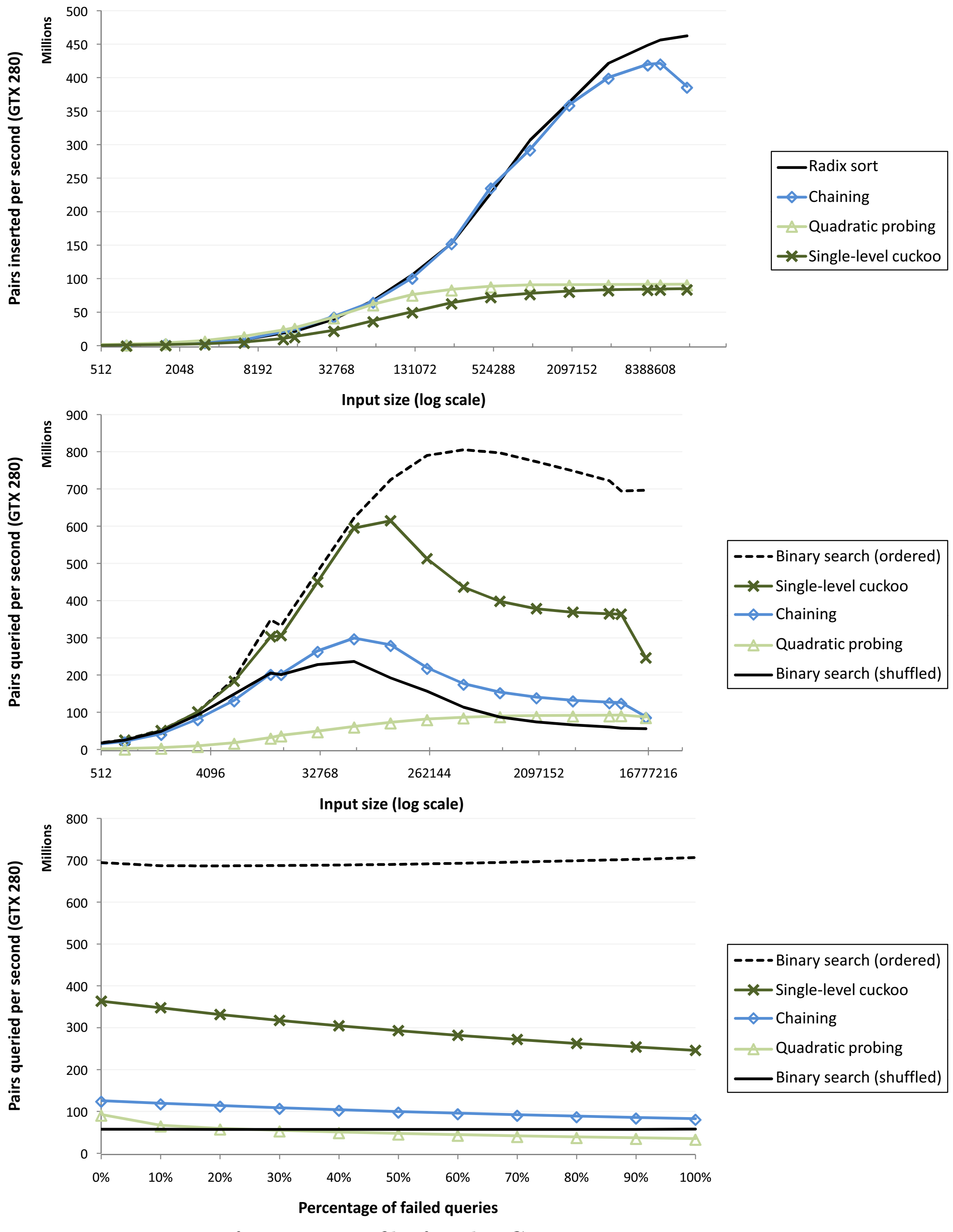


Figure 6.7. Performance profile for the GTX 280 using using $1.05N$ space, meant to show trends in the data. Charts show insertion rates versus input size (top), query rates versus input size (middle), and the effect of bad queries (bottom).

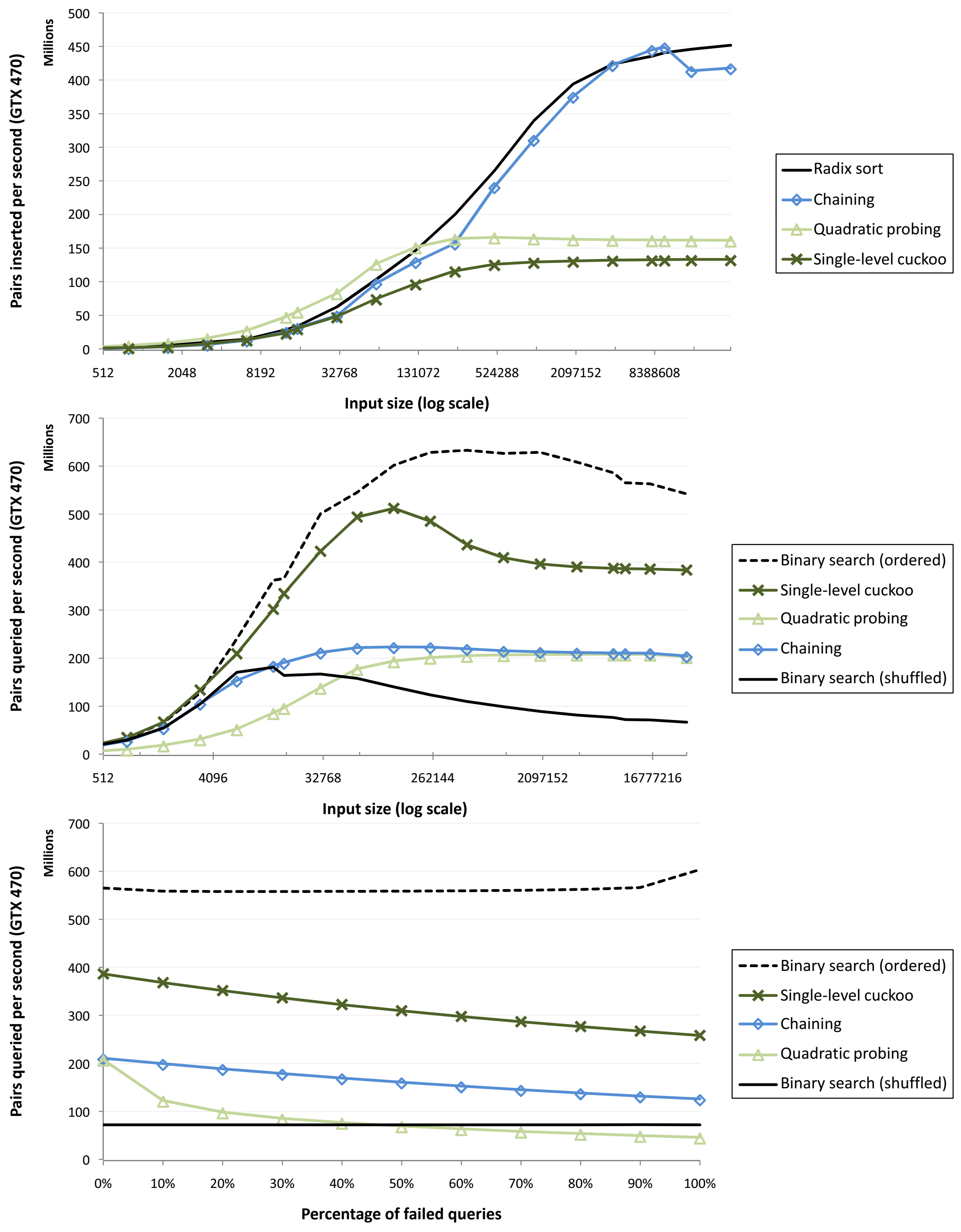


Figure 6.8. Performance profile for the GTX 470 using using 1.05N space, meant to show trends in the data. Charts show insertion rates versus input size (top), query rates versus input size (middle), and the effect of bad queries (bottom).

In contrast, chaining hash tables can be built quickly at this table size because it excels at fast constructions for small table sizes: the radix sort it uses as part of its construction has less work to do and the algorithm has less bucket boundaries to search for. Moreover, its retrievals are consistently faster than binary search and quadratic probing, allowing it to work well when an application needs a hash table that is built and used only briefly.

Like quadratic probing, our single-level cuckoo hashing table has difficulty being built because of the tight space constraints, resulting in the worst insertion performance among all of the data structures. However, its retrievals significantly outperform all of the other methods and it can handle bad queries in a robust manner. If a compact hash table is required and will be reused many times, the single-level cuckoo hashing scheme is definitely the right method to use.

6.4.2.2 Hash tables using $1.42N$ space

We chose to compare all of the data structures using $1.42N$ space because it is the default setting for our two-level hash table. Figure 6.9 shows a performance profile of all of the data structures we have discussed so far on a GTX 280. These graphs correspond to and continue the discussion started in Section 5.4; see that section for the discussion about the other data structures' performance.

Our single-level cuckoo hash table's insertion performance falls somewhere between chaining and quadratic probing, which is a good trade-off for the excellent retrieval performance it provides. In contrast, radix sort provides the best insertion rate, but requires a slow binary search. An interesting thing to note is that the trends in the insertion rate graph are similar for both quadratic probing and the single-level cuckoo hashing scheme; this is because they are both open addressing methods and work similarly. The two-level scheme does not exhibit the same behavior because of the bucketing it performs on its first level. The advantage cuckoo hashing has over quadratic probing is likely due to the way it processes the collisions: both methods can finish their insertions relatively quickly because 30%

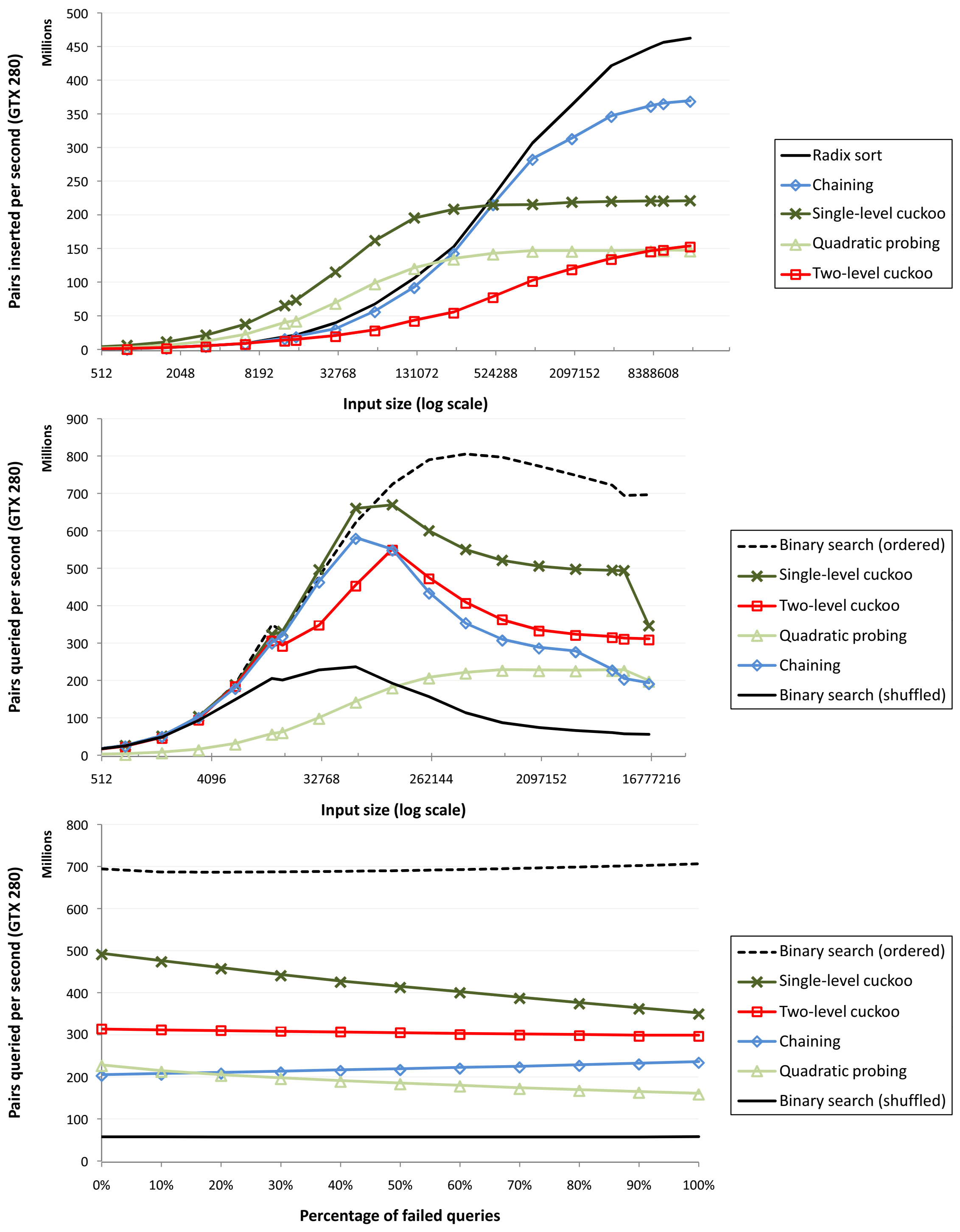


Figure 6.9. Effect of input size on the insertion (top) and query rates (center) of the data structures on a GTX 280. Also shown is the effect of searching for query keys not in a hash table containing 10 million items (bottom). All hash tables use $1.42N$ space.

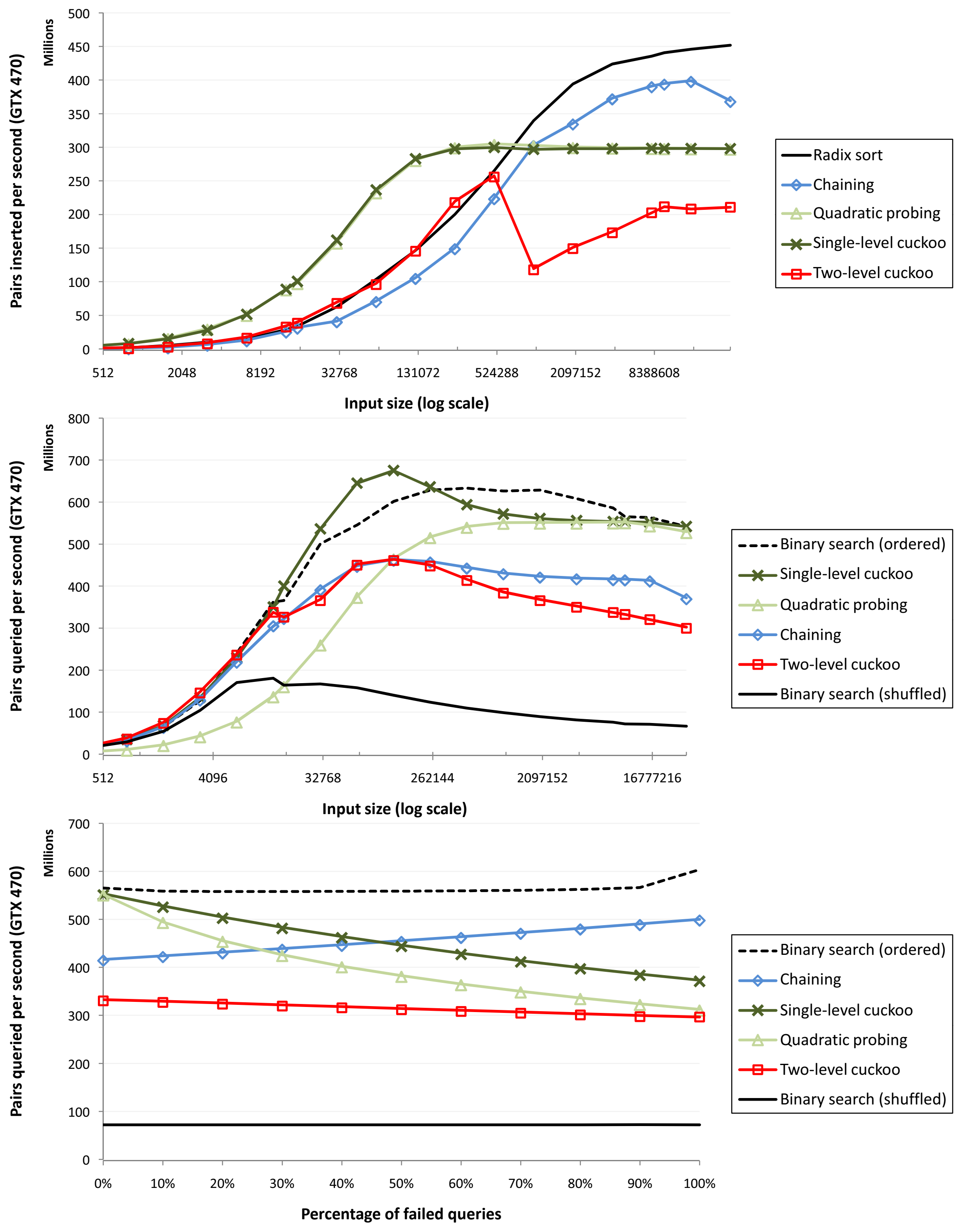


Figure 6.10. Effect of input size on the insertion (top) and query rates (center) of the data structures on a GTX 470. Also shown is the effect of searching for query keys not in a hash table containing 10 million items (bottom). All hash tables use $1.42N$ space.

of the table is unoccupied, but cuckoo hashing can move an item out of crowded areas much more quickly than quadratic probing can.

Our single-level cuckoo hash table outperforms our two-level hash table for insertions, reflecting the much simpler and more flexible construction of the new scheme. Moreover, it is more efficient at handling queries than even the two-level hash table. Although it uses four hash functions instead of three (like the two-level scheme), both tables effectively use the same number of memory accesses to perform a retrieval: in addition to the three probes into the table, the two-level hash table needs to read the seed used to generate a bucket's cuckoo hash functions[4]. Another factor in the performance is that we aren't using subtables: the two-level scheme only allows about a third of its items to be located in the first subtable, which means that more probes are required on average to find items.

Figure 6.10 shows the equivalent graphs for the GTX 470. Here we see that quadratic probing and our one level-scheme now have almost identical insertion performance on the newer GPU architecture, reflecting how the cache can catch the slots visited by quadratic probing. For retrievals, cuckoo hashing starts off stronger but eventually merges with quadratic probing, tying for the best retrieval rates among all of the data structures. Either of these methods would be a good choice if it is known ahead of time that the queries will consist entirely of the input keys.

However, the main difference between them is the performance disparity that arises when the queries start failing. Although both take a performance hit, quadratic probing takes a larger one, due to the number of probes it requires to terminate a query in the worst case. Chaining, on the other hand, actually has an increase in performance as the hash table encounters higher and higher percentages of bad queries. It also has the second highest insertion rate among the methods we tested. All things considered, chaining can be a good alternative to using cuckoo hashing under the right circumstances.

[4] We don't count the probe into the stash because it was unused in the vast majority of cases.

6.4.2.3 Hash tables using $2N$ space

We also examine the case where the hash table is twice as big as the input at $2N$ (Figures 6.11 and 6.12). Both quadratic probing and the single-level cuckoo hash table benefit from the larger table size because half of the slots are now empty, allowing them to terminate both insertions and queries much more quickly. On the GTX 280, single-level the cuckoo hash table dominates retrieval performance, both when the keys can and cannot be found in the hash table. For smaller input sizes, it is still possible that chaining can handle bad queries better than cuckoo hashing, but chaining's poor insertion rates make it a very unattractive method to use. Because more buckets are being used, extra passes are needed by the radix sort to move the items around during construction, resulting in a performance degradation. It always seems advisable to use cuckoo hashing, here.

For the GTX 470, the construction rates for both the single-level cuckoo hash table and quadratic probing completely overtake chaining and approach the speed at which the radix sort can be performed. Moreover, the retrieval rates for both of these methods outpace chaining, resulting in a large performance difference when all of the queries can be found in the hash table.

After the table contains around 500K items, quadratic probing retrieval rates completely overtake those of cuckoo hashing – it even becomes resistant to failed queries because it has so many empty slots. Quadratic probing gains the advantage because our cuckoo hashing implementation uses 4 hash functions, which increases the worst case number of probes for answering queries; Figure 6.6 shows that reducing the number of hash functions to two would increase the query rates to be on par with quadratic probing.

Unlike for the GTX 280, chaining retrievals can overtake both of these methods if the hash table is queried for many times for keys that aren't in the table; whether or not it is a better choice than quadratic probing is, again, dependent on the application.

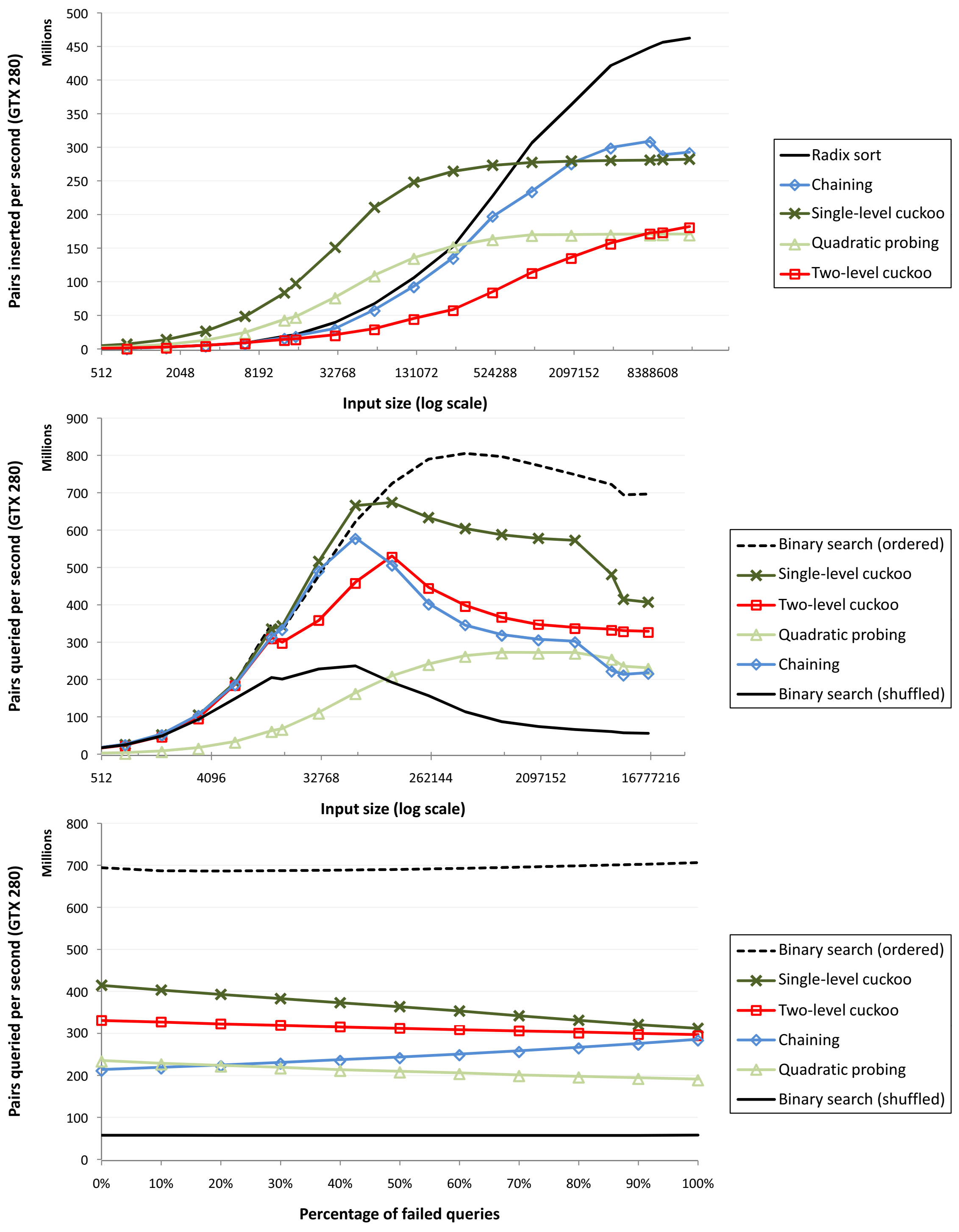


Figure 6.11. Effect of input size on the insertion (top) and query rates (center) of the data structures on a GTX 280. Also shown is the effect of searching for query keys not in a hash table containing 10 million items (bottom). All hash tables use $2N$ space.

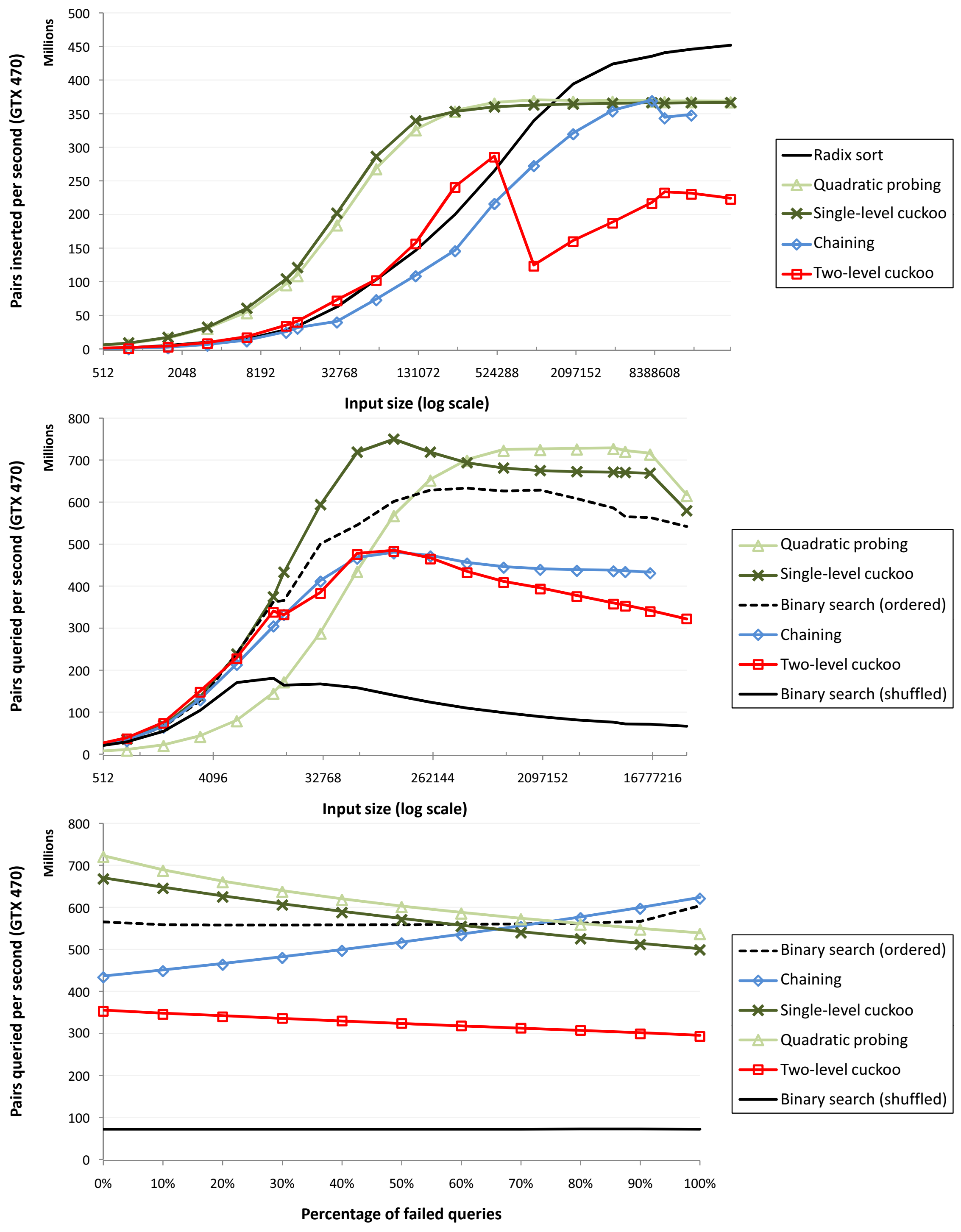


Figure 6.12. Effect of input size on the insertion (top) and query rates (center) of the data structures on a GTX 470. Also shown is the effect of searching for query keys not in a hash table containing 10 million items (bottom). All hash tables use $2N$ space.

6.4.2.4 Summary

Table 6.1 summarizes our findings for the GTX 470; we focus mainly on the 470's results because they are the most useful going forward.

6.4.3 Comparisons with the hash table specializations

We tested the performance of our compacting and multi-value hash tables by repeatedly building them with datasets of a fixed size, but with an increasingly higher average number of times a key is repeated.

The compacting hash table was compared against two equivalent data structures: a structure based around sorted arrays and compacted lists, and our previous two-level compacting hash table (Figure 6.13).

Both the single-level and two-level compacting hash tables consistently take longer to build than the radix-sorted structure. For the single-level table, the biggest gap occurs when there are no duplicates in the list; these differences arise from the extra work required to process the input keys. Once there is some key repetition, construction times for both methods drop. Although our single-level compacting hash table uses nearly the same algorithm, our faster cuckoo hashing procedure results in significantly higher build rates. Its retrievals also execute significantly faster than both of the other methods.

The multi-value hash table was compared against the equivalent two-level multi-value hash table, and a sorted structure based around sorted and compacted lists (Figure 6.14). To build the latter, we follow the construction procedure for the multi-value hash table until the construction of the hash table itself. Accessing the information for each key can then be done by employing binary searches rather than querying our hash table.

Because construction of our single-level hash table requires an additional step beyond construction of the sorted structure, our construction times are always slower. However, our retrievals are consistently faster than binary searching the structure until it becomes possible for a key to be repeated over two thousand times;

Quadratic probing	
Table size	*Remarks*
$1.05N$	There are better alternatives.
$1.42N$	There are better alternatives.
$2N$	When $N \geqslant 500,000$, the cache allows it the fastest constructions and retrievals. It can require a large number of probes to find some items, though most items require significantly less. Although it handles bad queries gracefully because the table is half-empty, chaining has better retrieval rates if queries have a high chance of failing ($\geqslant 80\%$).
Chaining	
$1.05N$	It has the fastest construction rates, but has slower retrievals because its buckets are larger. It is useful for situations where a temporary hash table is needed and used only briefly.
$1.42N$	It has the fastest construction rates when $N > 1,000,000$; the radix sort slowly gains speed before this point. Quadratic probing and single-level cuckoo hashing have faster retrievals when the queries can all be found, but chaining handles bad queries more robustly: it should be used when queries have a decent failure rate ($\geqslant 50\%$).
$2N$	It can't be constructed or queried as quickly as the other methods, but it is still useful if the queries have a high chance of failing.
Single-level cuckoo hashing with stashes	
$1.05N$	It has the fastest retrievals and slowest construction rates; it's great if it is built once and queried many times afterward.
$1.42N$	Cuckoo hashing is faster than chaining for $N < 1,000,000$, but chaining overtakes it for larger N. When queries are likely to succeed, it has the highest retrieval rates.
$2.0N$	Use it instead of quadratic probing for $N < 500,000$ because it has a higher retrieval rate up to this point.

Table 6.1. Summary of results for the GTX 470.

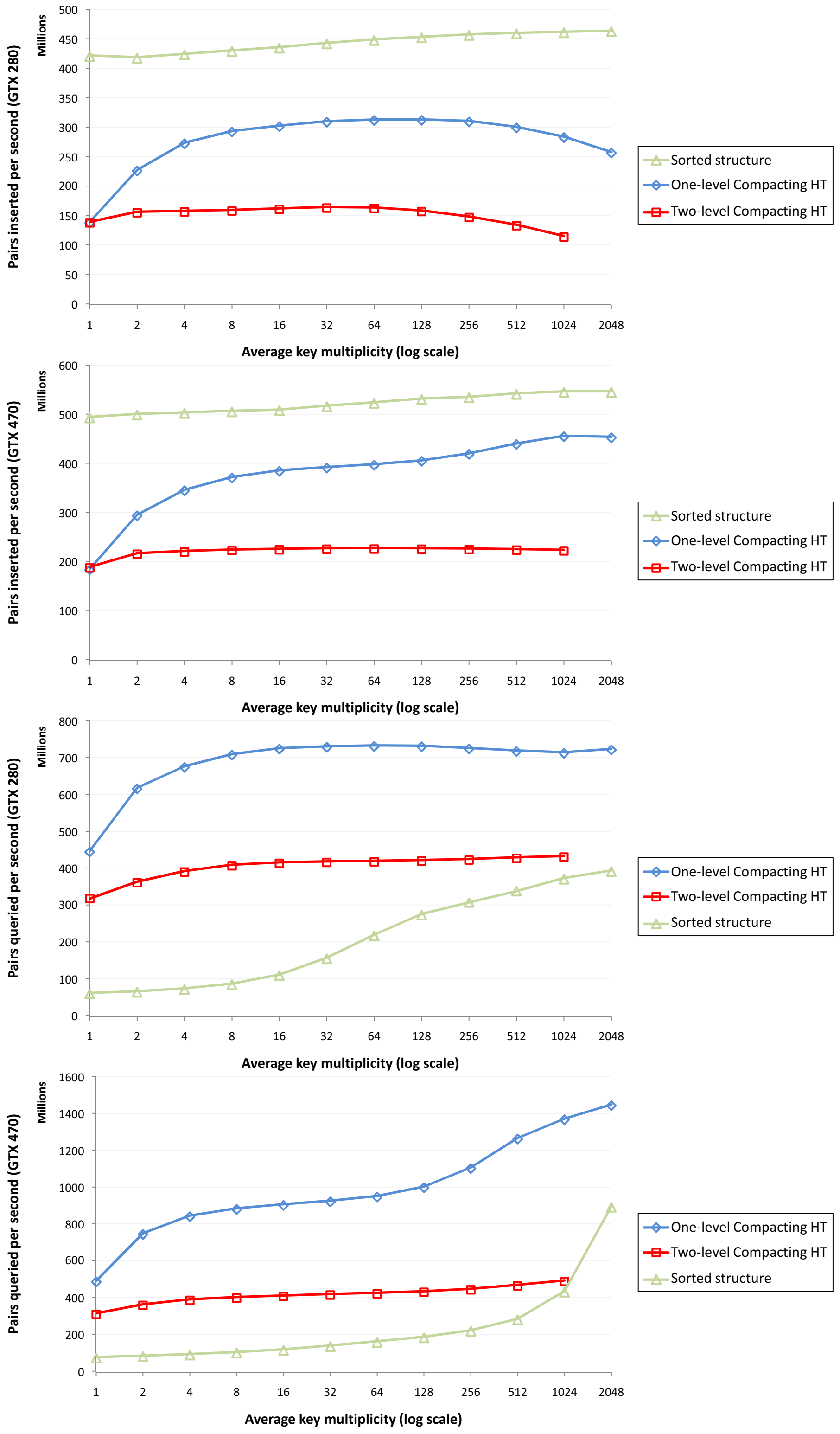


Figure 6.13. Compacting hash table timing comparison of construction (top half) and retrievals (bottom half) on both the GTX 280 and 470. Inputs consisted of 10 million keys with increasing multiplicity on the keys. Each key was queried once for every time the key appeared in the input.

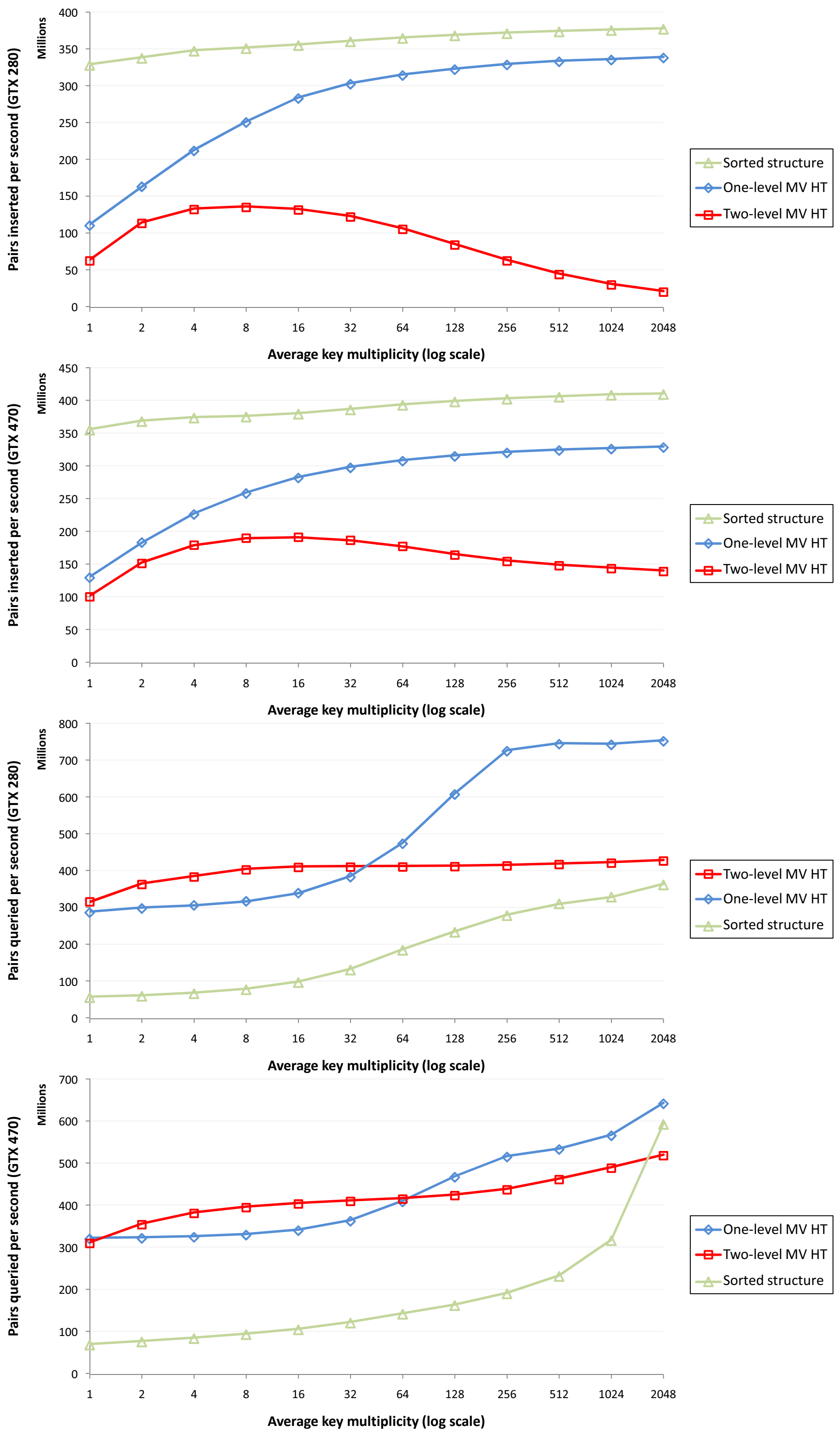


Figure 6.14. Multi-value hash table timing comparison of construction (top half) and retrievals (bottom half) on both the GTX 280 and 470. Inputs consisted of 10 million pairs with increasing multiplicity on the keys. Each key was queried once for every time the key appeared in the input.

it seems that binary search has the potential to overtake it for higher multiplicities. Moreover, our single-level hash table's size is always $1.25K$, where K is the number of unique keys in the input, while the two-level version's size is much larger at $1.42N$; as the average multiplicity of each key increases, the two-level multi-value hash table becomes more and more sparse.

The high plateau in retrieval rates for the GTX 280 likely corresponds to the peak in Figure 6.4. Because the keys are repeated so many times, the number of keys contained in the hash table is very small: once a key has an average of 128 copies, the 10M key-value pairs produce a hash table containing less than 78K items.

6.5 Limitations

The algorithm we described and the parameters we chose strike a good balance between the construction rates, retrieval times, and memory usage. However, there are still some drawbacks to be aware of.

Determining which function was used to insert an item requires that a thread recomputes the values of all of the hash functions for its new key after every iteration. This gets very costly if the hash function is hard to compute, reducing the performance of both construction and retrievals. Given the trends in GPU architecture, specifically the cost of computation versus the cost of a memory access, the extra computations are likely to remain a better trade-off than storing extra information in memory.

Restarts are rare, but expensive because the entire table must be rebuilt from scratch. This effectively multiplies construction time by the number of attempts needed to build the table. If a consistent construction time is important, the parameters can be changed to make the table easier to build.

The maximum number of iterations can be hard to pin down. As the table size approaches the theoretical minimum for the given number of hash functions, the average number of iterations performed by the construction algorithm rises very quickly. If set incorrectly, a thread will iterate far too many times before using the stash, making the cost of a rebuild even higher.

Sparsity of the compacting hash table becomes extremely high with high key multiplicity. If space usage is an issue, a procedure similar to the one followed for the multi-value hash table can be used, where the input is pre-processed using a radix sort. However, our construction can actually beat the radix sort times with smaller datasets; performing a radix sort beforehand will guarantee that construction is always slower.

6.6 Summary

The single-level cuckoo hashing table we've introduced in this section is highly robust and generally performs better than all of the other methods we have previously discussed. Like the two-level cuckoo hash table, it caps the number of probes required to find any item in the table. However, it's much more flexible and overcomes most of the previous method's shortcomings, resulting in significantly better performance in every situation we considered.

Although all of its memory accesses are highly uncoalesced, it performs well on both older and more modern hardware – it even gets a speed boost that the two-level cuckoo hash table doesn't get with the move to the Fermi architecture. It trades a faster construction for faster retrievals in compact tables, where the other hashing methods have trouble. On the GTX 280, it consistently has the best retrieval performance out of the methods we considered.

However, on the GTX 470, the other methods can perform better under the right conditions. For compact tables, it has the slowest construction but consistently has the best retrieval rates because it spends the time to make all of the queries easy to answer. For slightly larger hash tables (on the order of $1.42N$), the single-level cuckoo hashing table still performs well, but chaining can outperform if the hash table is repeatedly queried with keys that can't be found. Combined with its faster construction, chaining can be a better option for these situations.

For even larger hash tables, the cuckoo hash table's guarantee on the number of probes can actually become a liability. The results we presented comparing the single-level hash table always requires using four probes in the worst case, stemming from the fact that we use four hash functions. Even without this guarantee, the other methods have a low average number of probes and a small deviation from it. Moreover, they can take advantage of the cache because of the way that the elements are stored. In these cases quadratic probing outperforms the cuckoo hash table. Ultimately, the correct hash table to use for an application is dependent on how the hash table is to be used.

Chapter 7

Conclusion

We've presented a set of hash tables that are implemented on the GPU using fast, parallelized constructions. This allows parallel applications to construct them on the fly rather than having them constructed CPU-side and copied over later. Our designs are flexible, allowing them to be tailored for different applications. We examined how they performed under different situations and how well they balanced out the three constraints of memory usage, construction speed, and retrieval efficiency.

Despite repeatedly incurring uncoalesced memory accesses during retrievals, they can be queried at high rates and are almost consistently faster than using binary searches through a sorted array for random-access; for cases where the queries are completely sorted before accessing the data structures, the binary searches often perform better than hashing because their memory accesses repeatedly coalesce.

We hope future work will use other hashing algorithms to address different trade-offs between the metrics, particularly to address building dense hash tables at fast rates and to optimize for the fastest possible lookup times. We see a few potential avenues for future work that will make GPU-based hash tables more useful for the community:

- Our single-level cuckoo hashing implementation is able to take advantage of the faster atomic operations provided by modern GPUs, but has poor caching performance because the hash functions map everywhere in the hash table. One thing to try would be to allow multiple items to hash into each cuckoo hash table slot, allowing multiple items to reside in the cache line during retrievals. We saw the advantage this conferred to linear probing, which benefited greatly when its probes were entirely cached. Recent work on cuckoo hashing by Dietzfelbinger et al. [11] discusses this and other methods for reducing memory traffic, but they have yet to be tried on the GPU.

- Modifying the contents of the data structures we presented can be very difficult, since an insertion failure requires rebuilding the whole structure from scratch. We offset this by providing fast constructions, allowing the table to be rebuilt quickly with new items. In general this is a failing of many GPU data structures, such as spatial data structures like k-d trees or bounding volume hierarchies. Designing complex incremental parallel data structures for GPUs remains an active and interesting problem.

- We do not know how to handle input which exceeds the memory capacity of a single graphics card. Extending the algorithm to handle out-of-core input and multiple graphics cards is a challenging problem.

- Different applications also require different hash table features. The most efficient implementation of multiple-value hashing is not necessarily going to be the most efficient implementation when the multiple values can be aggregated (i.e. by counting or averaging), and yet another implementation might be more efficient when unique keys are guaranteed. A standard data structures library may have to include all of these specialized variants, and perhaps others.

References

[1] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen, "Parallel randomized load balancing," in *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '95. New York, NY, USA: ACM, 1995, pp. 238–247.

[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the GPU," *ACM Transactions on Graphics*, vol. 28, no. 5, pp. 154:1–154:9, Dec. 2009. [Online]. Available: http://www.idav.ucdavis.edu/publications/print_pub?pub_id=973

[3] A. Appleby, "Murmurhash project." [Online]. Available: http://code.google.com/p/smhasher/

[4] N. Askitis, "Fast and compact hash tables for integer keys," in *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ser. ACSC '09, 2009, pp. 113–122.

[5] N. Askitis and J. Zobel, "Cache-conscious collision resolution in string hash tables," in *String Processing and Information Retrieval*, ser. Lecture Notes in Computer Science, M. Consens and G. Navarro, Eds. Springer Berlin / Heidelberg, vol. 3772, pp. 91–102.

[6] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," *SIAM Journal on Computing*, vol. 29, no. 1, pp. 180–200, Feb. 2000.

[7] H. Bast and T. Hagerup, "Fast and reliable parallel hashing," in *ACM Symposium on Parallel Algorithms and Architectures*, 1991, pp. 50–61.

[8] C. DeCoro and N. Tatarchuk, "Real-time mesh simplification using the GPU," in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, Apr./May 2007, pp. 161–166.

[9] L. Devroye and P. Morin, "Cuckoo hashing: Further analysis," *Information Processing Letters*, vol. 86, no. 4, pp. 215–219, 2003.

[10] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink, "Tight thresholds for cuckoo hashing via XORSAT," in *37th International Colloquium on Automata, Languages and Programming*, Jul. 2010, pp. 213–225.

[11] M. Dietzfelbinger, M. Mitzenmacher, and M. Rink, "Cuckoo hashing with pages," in *European Symposium on Algorithms*, 2011, in submission.

[12] T. Foley and J. Sugerman, "KD-Tree acceleration structures for a GPU raytracer," in *Graphics Hardware 2005*, Jul. 2005, pp. 15–22.

[13] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud, "Practical minimal perfect hash functions for large databases," *Communications of the ACM*, vol. 35, no. 1, pp. 105–121, Jan. 1992.

[14] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with $O(1)$ worst case access time," *Journal of the ACM*, vol. 31, no. 3, pp. 538–544, Jul. 1984.

[15] A. Frieze, P. Melsted, and M. Mitzenmacher, "An analysis of random-walk cuckoo hashing," in *Proceedings of the 12th International Workshop and 13th International Workshop on Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, ser. APPROX '09 / RANDOM '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 490–503. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03685-9_37

[16] H. Gao, J. F. Groote, and W. H. Hesselink, "Almost wait-free resizable hashtable." in *IEEE International Parallel & Distributed Processing Symposium 2004*, 2004.

[17] J. Gil and Y. Matias, "Fast hashing on a PRAM—designing by expectation," in *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, 1991, pp. 271–280.

[18] ——, "Simple fast parallel hashing by oblivious execution," *SIAM Journal of Computing*, vol. 27, no. 5, pp. 1348–1375, 1998.

[19] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High performance graphics coprocessor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, Jun. 2006, pp. 325–336.

[20] M. Harris, J. D. Owens, S. Sengupta, Y. Zhang, and A. Davidson, "CUDPP: CUDA data parallel primitives library," 2009, http://gpgpu.org/developer/cudpp/.

[21] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, ser. I3D '07. New York, NY, USA: ACM, 2007, pp. 167–174. [Online]. Available: http://doi.acm.org/10.1145/1230100.1230129

[22] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal of Computing*, vol. 39, pp. 1543–1561, December 2009. [Online]. Available: http://dx.doi.org/10.1137/080728743

[23] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 579–588, Jul. 2006.

[24] A. Lefohn, "Glift: Generic data structures for graphics hardware," Ph.D. dissertation, Department of Computer Science, University of California, Davis, Sep. 2006.

[25] A. E. Lefohn, J. Kniss, R. Strzodka, S. Sengupta, and J. D. Owens, "Glift: Generic, efficient, random-access GPU data structures," *ACM Transactions on Graphics*, vol. 26, no. 1, pp. 60–99, Jan. 2006.

[26] Y. Matias and U. Vishkin, "On parallel hashing and integer sorting," in *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, 1990, pp. 729–743.

[27] ——, "Converting high probability into nearly-constant time, with application to parallel hashing," in *ACM Symposium on the Theory of Computing (STOC)*, 1991, pp. 307–316.

[28] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 3–30, January 1998. [Online]. Available: http://doi.acm.org/10.1145/272991.272995

[29] D. Merrill and A. Grimshaw, "Revisiting sorting for GPGPU stream architectures," Department of Computer Science, University of Virginia, Tech. Rep. CS2010-03, Feb. 2010.

[30] M. Mitzenmacher and S. Vadhan, "Why simple hash functions work: Exploiting the entropy in a data stream," in *SODA '08: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, Jan. 2008, pp. 746–755.

[31] J. I. Munro and P. Celis, "Techniques for collision resolution in hash tables with open addressing," in *Proceedings of 1986 ACM Fall Joint Computer Conference*, ser. ACM '86, 1986, pp. 601–610.

[32] NVIDIA Corporation, "NVIDIA's next generation CUDA compute architecture: Fermi," 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf.

[33] ——, "NVIDIA CUDA compute unified device architecture, programming guide," 2011, http://developer.nvidia.com/.

[34] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *9th Annual European Symposium on Algorithms*, ser. Lecture Notes in Computer Science, vol. 2161. Springer, Aug. 2001, pp. 121–133.

[35] W. W. Peterson, "Addressing for random-access storage," *IBM Journal of Research and Development*, vol. 1, pp. 130–146, April 1957.

[36] C. Purcell and T. Harris, "Non-blocking hashtables with open addressing," University of Cambridge Computer Laboratory, Tech. Rep., 2005.

[37] J. Sanders and E. Kandrot, *CUDA By Example: An Introduction to General-Purpose GPU Programming.* Addison Wesley, 2011.

[38] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[39] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *Journal of the ACM*, vol. 53, pp. 379–405, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1147954.1147958

[40] X. Sun, K. Zhou, E. Stollnitz, J. Shi, and B. Guo, "Interactive relighting of dynamic refractive objects," *ACM Transactions on Graphics*, vol. 27, no. 3, pp. 35:1–35:9, Aug. 2008.

[41] B. Vöcking, "How asymmetry helps load balancing," *Journal of the ACM*, vol. 50, no. 4, pp. 568–589, Jul. 2003.

[42] K. Zhou, M. Gong, X. Huang, and B. Guo, "Highly parallel surface reconstruction," Microsoft Research, Tech. Rep. MSR-TR-2008-53, 1 Apr. 2008.

[43] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time KD-tree construction on graphics hardware," *ACM Transactions on Graphics*, vol. 27, no. 5, pp. 126:1–126:11, Dec. 2008.