Karlsruhe Institute of Technology

Accelerating Minimal Perfect Hash Function Construction Using GPU Parallelization

Master's Thesis of

Stefan Hermann

at the Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Peter Sanders
Second reviewer:	TT-Prof. Dr. Thomas Bläsius
Advisor:	M.Sc. Hans-Peter Lehmann

1. May 2023 – 2. November 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 2. November 2023

. . . . . .

(Stefan Hermann)

Abstract

A Minimal Perfect Hash Function (MPHF) maps a set of N keys to $[N] := \{0, .., N-1\}$ without collisions. This thesis contributes significant advances to the following generic MPHF construction algorithm. In a first step, the keys are distributed into *buckets* of different expected sizes. We show that determining the expected bucket sizes is an optimization problem which can be solved using the Euler-Lagrange equation. The optimization significantly improves performance compared to the state-of-the-art. In a second step, the buckets are primarily ordered by non-increasing size. We demonstrate that ordering buckets of equal size secondarily by increasing *expected* size improves space consumption. The buckets are processed in that order in a third step by finding a hash function that maps the keys of the bucket to $[N]$ without collisions. In a last step, an identifier of the hash function of each bucket is stored in a compressed manner. We present a new compression technique which assigns the identifiers to different encoders, such that all identifiers within the encoders follow almost the same statistical distribution. Thereby, improving the compressibility of the identifiers significantly.

We harness the power of GPU parallelization to further accelerate MPHF construction. Our GPU implementation constructs an MPHF of 1.73 bits per key in just 36 ns per key with a CPU query time of 44 ns. Such a fast query time and simultaneously low space requirement is unreachable by current state-of-the-art implementations like PTHash. An MPHF that has a higher space consumption of 1.88 bits per key is constructed 9926 times faster by our implementation compared to PTHash. Most of our contributions are applicable beyond our specific implementation and can improve the performance of state-of-the-art approaches like PTHash.

Zusammenfassung

Eine Minimale Perfekte Hashfunktion (MPHF) bildet eine Menge von N Schlüsseln kollisionsfrei auf die Menge $[N] := \{0, .., N-1\}$ ab. Diese Thesis leistet einen signifikanten Beitrag für den folgenden generischen MPHF Konstruktionsalgorithmus. Im ersten Schritt werden die Schlüssel in *Buckets* unterschiedlicher erwarteter Größe verteilt. Wir zeigen, dass die Wahl der erwarteten Bucketgröße ein Optimierungsproblem darstellt welches durch die Euler-Lagrange Gleichung gelöst werden kann. Dies resultiert in eine signifikante Verbesserung im Vergleich zum derzeitigen Stand der Forschung. Im zweiten Schritt werden die Buckets primär in nicht aufsteigender Größe geordnet. Wir zeigen, dass der Platzbedarf verbessert wird wenn Buckets gleicher Größe sekundär in aufsteigender Erwartungsgröße angeordnet werden. Die Buckets werden dann im dritten Schritt in dieser Reihenfolge verarbeitet indem eine Hashfunktion gefunden wird welche alle Schlüssel des Buckets kollisionsfrei auf $[N]$ abbildet. Abschließend wird für jeden Bucket ein Identifikator der Hashfunktion komprimiert gespeichert. Wir präsentieren eine neue Kompressionstechnik, welche die Identifikatoren in unterschiedliche Enkodierer anordnet, sodass alle Identifikatoren innerhalb eines Enkodierers der gleichen statistischen Verteilung folgen. Dies verbessert die Komprimierbarkeit der Identifikatoren.

Wir nutzen die parallele Leistungsfähigkeit von GPUs um die Konstruktion von MPHFs weiter zu beschleunigen. Unsere GPU Implementierung konstruiert eine MPHF mit 1,73 Bits pro Schlüssel in nur 36 ns pro Schlüssel mit einer CPU Abfragezeit von 44 ns. Eine solch geringe Abfragezeit bei gleichzeitig niedrigem Platzbedarf ist nach heutigem Stand, wie z.B. mit PTHash, nicht erreichbar. Eine MPHF, die einen höheren Platzbedarf von 1,88 Bits pro Schlüssel aufweist, wird mit unserer Implementierung 9926 mal schneller konstruiert als durch PTHash. Die meisten unserer Beiträge sind über unsere spezifische Implementierung hinaus anwendbar und können selbst modernste Techniken weiter verbessern.

Acknowledgements

I thank my supervisor Hans-Peter for detailed feedback on this thesis. I also thank Giulio, Prof. Dr. Peter Sanders and Hans-Peter for interesting and fruitful discussions. For proofreading this thesis I thank Frank. I am also grateful that it was possible for me to extensively use computing resources of the institute.

Contents

1. Introduction

In computing, we often require data structures which quickly map a set of N keys to a range of integers $[M] := \{0, .., M-1\}$. This mapping is referred to as a *hash function*. A hash function itself does not store any information about the key set. It maps the input to an integer, regardless of whether multiple keys map to the same integer. If multiple keys map to the same integer we refer to this as a *collision*. A *perfect* hash function (PHF) is a mapping without collisions. This requires that $M \geq N$. To prevent collisions, a PHF generally has to store information about the key set. Collisions are more likely for an decreased M/N ratio. The amount of storage required to prevent collisions increases accordingly. In the extreme case of $N = M$, the mapping is referred to as *minimal* PHF (MPHF). The asymptotic lower bound for the entropy of an MPHF is $\log_2(e) \approx 1.44$ bits per key [1]. This theoretical lower bound can almost be reached in practice. Lehmann et al. [2] present ShockHash which can result in an MPHF of just 1.52 bits per key. It is well established in literature [3, 1] to assess PHF construction techniques based on three metrics: (I) construction time, (II) query time and (III) space consumption.

1.1. Applications

A PHF requires less space compared to a classical hash map, which stores the entire key set explicitly. The reduced space consumption allows the PHF to be available in a higher level of the memory hierarchy, possibly resulting in a faster query speed. Because of this advantage, PHFs find widespread practical application, e.g. in compressed full-text indexes [4], computer networks [5], databases [6], prefix-search data structures [7], language models [8], bioinformatics [9], as well as Bloom filters [10] and many more.

1.2. Scope and Contribution

This thesis considers construction of *static* PHFs. The entire key set is known when construction begins. Adding or removing keys is not supported. *Dynamic* PHFs support those functionalities but are inherently more complex [11]. Furthermore, this thesis primarily considers minimal PHFs, but most results are also applicable to non-minimal PHFs.

The generic PHF construction technique used in this thesis is a *bucket based* approach. First, all N keys are distributed into buckets. We refer to bucket size as the number of keys in a bucket. The buckets are ordered primarily by non-increasing size. Next, they are *inserted* in that order. A bucket is inserted by finding a hash function which maps the keys of the bucket injectively to the integers $[M]$ without creating collisions with previous

insertions. Once a function is found, an identifier of the function, called *pilot*, is stored in a compressed manner. The PHF is queried by determining the bucket of the key, accessing the pilot of the bucket and applying the hash function.

The procedure has been investigated in FCH [12], CHD [13], PTHash [3] and others. Bucket based approaches are well known for their fast query time. Some pilot compression techniques allow querying the MPHF using only a single memory access. Other compression techniques result in a lower space consumption but in an increased query time. However, even the most space efficient bucket based implementation [13] is not competitive in terms of space consumption with ShockHash which can reach 1.52 bits per key [2]. We remark that the word "bucket" is used somewhat ambiguous in literature. The definition in this introduction of bucket based approaches is used consistently in this thesis.

This thesis contributes significant advances to the bucket based technique:

1. When distributing the keys to buckets we can freely choose the expected size of each individual bucket. We show that optimal expected bucket sizes can be expressed as the solution of a differential equation.

2. We demonstrate that the ordering step should not only primarily sort by non-increasing bucket size. A significant space reduction can be achieved by sorting buckets of equal size secondarily by increasing expected bucket size.

3. We provide multiple improvements to the encoding step. Overall, reducing space consumption while maintaining query time.

4. We present an efficient GPU implementation to further accelerate all steps of MPHF construction.

Our improvements are applicable to bucket based state-of-the-art techniques (such as PTHash [3]). Overall, our implementation can construct MPHFs of 1.73 bits per key, while maintaining fast query and construction time of 44 ns per key and 36 ns per key respectively. This result is for 100 million keys on a Nvidia RTX 3090.

1.3. Outline

The next chapter introduces the required foundations for this thesis. Chapter 3 reviews several approaches to construct a PHF. They expose different tradeoffs between the three metrics. Chapter 4 explains the construction algorithms of this thesis in detail. In Chapter 5 we analyze the brute force search algorithm, to obtain an optimal distribution of bucket sizes. GPU implementations of the algorithms are described in Chapter 6. Finally, we present experimental results and comparisons with state-of-the-art approaches in Chapter 7. We summarize this thesis in Chapter 8 and propose several ideas for future works.

2. Preliminaries

In this chapter we introduce the concepts required to understand implementation and analysis of the algorithms.

Model and Assumptions. For algorithmic analysis we assume the word random-access machine model [14]. In this model, a random-access machine can perform bitwise and arithmetic operations on words in constant time. Furthermore, we assume that we have access to a fully randomized hash function which can be evaluated in constant time. We assume that the hash values of the input keys are free of collisions and uniformly distributed.

Notation and Definitions. We define $[N] := \{0, 1, ..., N-1\}$, $\log(x)$ is the natural logarithm and a div $b = \lfloor a/b \rfloor$. Given a set of points $V \subset \mathbb{R}^N$, the *Pareto front* is defined as the set $\{v \in V \mid \forall_{c \in V} \exists_{d \in [N]}\ c_d \leq v_d\}$. Intuitively, it contains all points for which there is no other point which is smaller in all dimensions. Those are the points of interest in a multi-dimensional minimization problem.

2.1. GPU

A Graphics Processing Unit (GPU) is an electronic circuit, initially designed to accelerate tasks such as video game rendering and image processing. It is either integrated into a CPU or a motherboard, or it otherwise resides on a discrete graphics card. GPUs are specialized for performing the same procedures independently on a large amount of data. Historically, those algorithms were fixed by the manufacturer or only configurable e.g. a rendering pipeline. Over the last decades, GPUs have become a general-purpose platform, opening the door for a new field of programming: GPU computing.

In comparison to a CPU, a GPU has a few fast cores optimized for latency. A GPU has many slower cores and is optimized for throughput. The advantage of a high throughput comes at the cost of specific programming constraints. Some problems cannot be solved efficiently within those constraints.

2.1.1. High-Level Architecture

In the following we take a closer look at a generic modern GPU architecture and its implications on GPU programming. Manufacturer or device specifics will only be addressed if relevant for this thesis. However, to understand the scale of the discussed components, exact numbers based on the "Nvidia RTX 3090" [15] are provided.

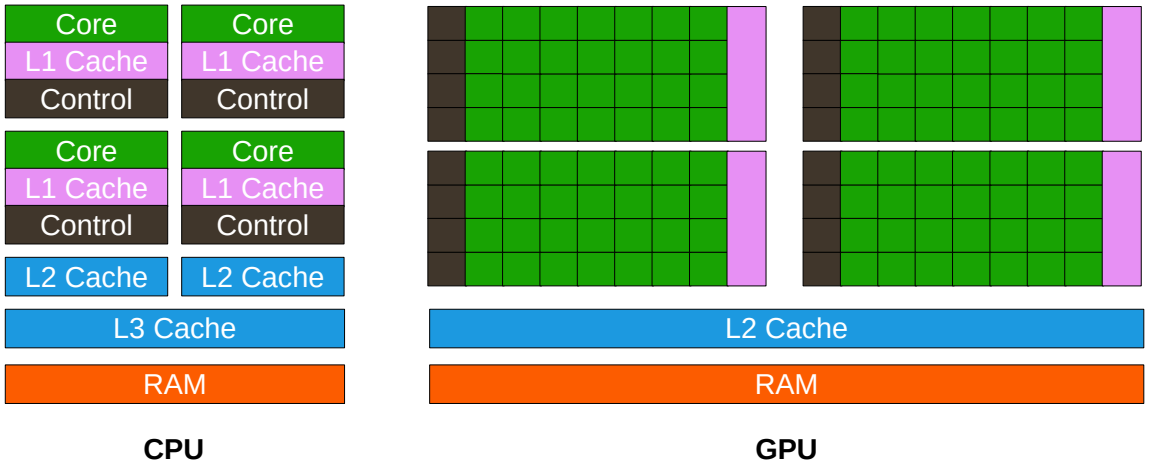

Figure 2.1.: Comparison between a quad core CPU and a GPU. The GPU shown here has four SMs. Each is made up of four warps. A warp has seven SPs.

A GPU is made up of a single *global memory* and physically separated (RTX 3090: 82) *Streaming Multiprocessors* (SM). The global memory can directly be accessed by the SMs. The CPU can initiate data transfer between global GPU memory and CPU RAM.

Each SM contains (RTX 3090: 128) Streaming Processors (SP) that are responsible for actual processing. This results in a total of $128 \cdot 82 = 10946$ SPs on the RTX 3090. The SPs are grouped in constellations called *warp* (Nvidia terminology). The number of SPs in each warp is 32 (Nvidia). Thus, there are 4 warps per SM on the RTX 3090.

A comparison between a CPU and a GPU is illustrated in Figure 2.1. A GPU dedicates much more space to actual computing cores compared to a CPU.

2.1.2. Execution Model

The execution model of GPUs is *Single Program, Multiple Data*. Each SP belonging to the same warp performs the same instruction at the same time. This is also referred to as *lock-step*. SPs have direct access to the SMs registers and contain the execution cores for typical (RTX 3090: 32 bit) integer and floating point operations. SPs also have special execution units e.g. for functions like exp, sin, cos.

When code is placed on a GPU, called a *shader*, it is dispatched in a specified number of *workgroups*. Each workgroup applies exactly the same algorithm, as specified by the shader. Symmetry is usually broken using an identifier of the workgroup, which is unique within one dispatch. The user also specifies the number of *threads* per workgroup, called the *workgroup size*. Again, each thread runs the code of the shader and symmetry is broken using a thread identifier which is unique within one workgroup.

A *warp scheduler* decides which thread is to be executed on a SP. To allow for quick switching between different threads, a GPU has a large number (RTX 3090: 65536) of registers. This flexibility allows the scheduler to hide the latency of certain operations. E.g. a global memory read requires several hundred clock cycles. To increase the utilization of the available resources, other threads continue execution in its place, drastically increasing the total throughput. This of course requires a thread to be available for execution in the

first place. Thus, it is important to saturate the SMs with threads. The RTX 3090 allows up to 12 times more threads than there are SPs.

Recall that all SPs in a warp execute the same instruction at the same time. This somewhat constraints the warp scheduler because it cannot choose threads arbitrarily. In fact, only threads with the same instruction pointer will be scheduled for execution together. This behavior profoundly impacts the design decisions for shaders. Threads cannot be scheduled together if they follow different control paths. If there are not enough threads within one workgroup following the same control path, then some SPs of a warp will have to idle. Thus, when developing shaders, it is crucial to minimize divergent control paths.

Cache. Each SM has a separate L1 cache of global memory. The size of the L1 cache on the RTX 3090 is up to 128 KiB per SM or 10 MiB in total. This can significantly reduce latency of memory accesses, provided that they are local in space and time. However, having a fast cache in each SM comes at the cost of cache incoherency. Because of the large number of threads, cache coherence protocols like on a CPU are not viable. Regaining a coherent view on the global memory of a GPU requires explicit flushing and invalidation of the caches by software. The total size of L1 GPU cache is similar in magnitude to one of a usual CPU. However, the amount of cache per thread of a GPU is drastically smaller compared to a CPU. The RTX 3090 can have up to 125,952 threads, which results in just 83 bytes of L1 cache per thread. There are also other kinds of caches like instruction cache, constant cache and texture cache, but they are not discussed in this introduction.

Shared Memory. Another type of memory residing on the SM is called *shared* memory. Access time of shared memory is similar to L1 cache. Usually (like on the RTX 3090), L1 cache and shared memory reside on the same hardware. Consequently, if a shader requires more shared memory it can use less L1 cache. The difference to cache is that shared memory is explicitly handled by software. This allows it to be conveniently used to share data across all threads of the same workgroup. It can also be used as intermediate storage if registers are not sufficient.

Each SM has 32 *memory banks*. A memory bank is responsible for handling access to a distinct subset of shared memory. More specifically, the $i-th$ memory bank handles the word-sized memory addresses $\{k \mid k \bmod 32 = i\}$. A memory bank processes a single read and write access of one word at a time. If a memory bank is accessed for n different addresses at the same time we refer to that case as a n-way bank *conflict*. Such conflicts result in serialization of the accesses, drastically reducing performance. However, if multiple threads read the same address, the respective memory bank broadcasts the result to all participating threads.

2.1.3. Global Memory

The global memory of a GPU (RTX 3090: 24 GB) is directly accessible by the SMs. The theoretical bandwidth between the SMs and the global GPU memory is quite high (RTX 3090: 936 GB/s). Such a high bandwidth however comes at the cost of a specific access

pattern. Memory is read in large and aligned transactions of 128 byte. For example, on the RTX 3090, reading 32 words from the same 128 byte aligned address results in one single transaction only. The full utilization of a single memory transaction is also referred to as *coalesced* access. At the same time, reading 32 words from different 128 byte aligned addresses results in a total of 32 memory transactions.

A GPU also has a single (6 MiB) L2 cache of global memory which is shared by all SMs. Again, the size is comparable in magnitude to those of a CPU, while the amount of L2 cache per thread is very low.

The CPU can explicitly transfer data to or from the GPU global memory. There is also the option of enforcing coherence between a memory space in CPU RAM and a memory space in global GPU memory. In any case, transfer times will turn out to be a bottleneck in our implementation.

2.1.4. Nomenclature Confusion

Different manufacturers and even different APIs use different names for the same component. We will stick to names used in this GPU introduction which are based on the Vulkan specification [16] and NVIDIA terminology. Be aware that in other literature global memory is sometimes referred to as shared memory, shared memory as local memory, local memory for other kinds of memory and warps as subgroups or wavefronts.

2.2. Vulkan

Vulkan[16] is a low-level, cross-platform, graphics and compute API. Compared to other APIs like CUDA, Vulkan is explicit and highly configurable. In the following we will look at the core components of a Vulkan program.

2.2.1. Shader

A shader is code to be executed on the GPU. In practice, shaders are usually written in Graphics Library Shading Language (GLSL). GLSL has a C-style syntax. It has a main function which is the entry point for all threads. Symmetry between the threads can be broken using a thread identifier. Likewise, symmetry between workgroups is broken using a workgroup identifier. A shader main function has neither explicit input nor output values. Results are written to global buffers. Input data can be accessed either by global buffers or by so called *push constants*. Push constants are a convenient way to pass a very small amount (e.g. 256 bytes) of constant data to all shaders of an invocation. The advantage is that this data can be accessed much faster than global buffers.

The GLSL code is then compiled to cross-platform SPIR-V bytecode. Vulkan compiles the SPIR-V code to machine code. The shader compilation is performed at C++ runtime allowing insertion of *specialization constants* into the shaders. Specialization constants are placeholders in the SPIR-V which are replaced with variables of the CPU. The shader compiler can then apply the optimizations usually associated with constants e.g. modular

arithmetic and loop unrolling. Thus, specialization constants should be preferred over push constants, if the constant is known at compile time of the shader.

2.2.2. Command Buffer

A command buffer is a list of instructions to be executed on a GPU. Possible commands include memory transfers between CPU and GPU, invocation of shaders, timestamps, barriers and many more. Barriers can be applied to enforce sequentiality of certain commands. Otherwise, the commands may get reordered for the purpose of optimization. Such a command buffer can then be submitted for execution to a GPU. When submitting, a semaphore can be used to wait until the command buffer has finished execution.

2.3. Standard Algorithms

We define some standard algorithms and techniques, which will be used extensively in this thesis.

2.3.1. Standard Parallelization

We will often refer to the "Standard Parallelization" technique. It is the trivial parallelization inside one workgroup with W threads, if there are T independent tasks T_i. Each thread is identified using the workgroup $localId \in [W]$. Algorithm 1 is applied in that case.

Algorithm 1 Standard Thread Level Parallelization

```
1: for i ← localId; i < T; i+ = W do
2:     doTask(T_i)
```

2.3.2. Prefix Sum

Given an array of values $x[]$, the exclusive prefix sum is defined as $y[i] = \sum_{j=0}^{i-1} x[j]$. In the following we look at two GPU implementations of the parallel prefix sum (PPS) algorithm. The first one is for calculating the prefix sum inside one workgroup. The second one is for calculating the prefix sum of a large buffer.

2.3.2.1. Local Parallel Prefix Sum

We implemented the standard parallel prefix sum algorithm inside one workgroup [17]. The algorithm runs in $O(\log_2 N)$, if the number of threads is proportional to N. During the algorithm we also obtain the sum of all elements, which is not needed for the prefix sum itself, but we often require it for another purpose.

2.3.2.2. Global Parallel Prefix Sum

If the prefix sum has to be performed on a larger scale input $x[]$, we cannot perform the prefix sum efficiently in a single workgroup. Let N be the size of x and let E be the number of elements a local prefix sum workgroup can handle. We invoke $\lceil N/E \rceil$ local prefix sum workgroups. Each workgroup g performs prefix sum on a subsection $x[g \cdot E, .., (g+1) \cdot E - 1]$ of the input. Each workgroup also stores the local sum in a separate buffer y. $y[g]$ is the sum of the subsection of workgroup g. We then call global parallel prefix sum recursively on y, if y has more than one element. Finally, a second shader adds the prefix sum of $y[g]$ to all elements of the respective subsection $g + 1$.

2.3.3. Pair Building

Given a set $[N]$ we want to determine all $N \cdot (N-1)/2$ unique pairs. A pair is a subset of S with two elements. This can be accomplished trivially using a nested loop. The outer loop iterates $i \in [N]$ and the inner loop iterates $j \in [i]$. Unfortunately, the inner loop iteration is dependent from the outer iteration and thus cannot be parallelized efficiently. If we want to apply the standard parallelization method to all pairs, we require a closed form solution for the n-th pair. We consider two options: (I) We precompute all pairs $[\{1,0\},\{2,0\},\{2,1\},\{3,0\},..,\{3,2\},\{4,0\},..]$ in an array. We can access the n-th pair independent of N (i.e. no separate array is required for each value of N). The downside of this approach is that the maximal value must be known in advance, to allow precomputing. (II) We came up with the closed form Algorithm 2, which only requires a single computationally expensive integer division. Note that the branch does not introduce divergence, because the condition of the branch is constant across all participating threads. To the best of our knowledge, the currently only known inverse pairing function [18] which is applicable here requires a square root. This is computationally even more expensive than a division. We will compare (I) and (II) in the evaluation chapter.

Algorithm 2 Closed Form Pair Builder

```
function GETPAIR(n, N)
    value ← n · 2 + 1
    even ← ¬(N&1)
    divisor ← N − even
    y ← value div divisor
    x ← value − y · divisor
    if even then
        x ← x + (x > y)
    else
        y ← y + (x < y)
    return (x, y)
```

2.4. Bucket Based PHF

In the following we define the *bucket based* PHF construction framework, on which part of related work and the entire working principle of this thesis is based on. The first step of the generic construction algorithm is to distribute the keys into *buckets*. The function assigning a key to a bucket is independent of the input key set. After all keys are assigned to buckets, the bucket *sizes* can be determined. The size of a bucket is the number of keys mapped to it. The second step of the algorithm is ordering the buckets by non-increasing size. In the third step, buckets are *inserted* sequentially, usually based on the order of the second step. Insertion of a bucket means that an injective hash function is found such that all keys of that bucket are mapped to integers on which no key from a previous bucket has been mapped to. We will also refer to those integers as the *positions* of the keys. In the fourth step, an identifier of the insertion hash function of each bucket is stored, allowing access when querying the PHF. We refer to those identifiers as the *pilots*. The pilot values are then encoded in some manner. Algorithm 3 shows a pseudo code description of the construction and query algorithm. Note that the word "bucket" is used ambiguously throughout PHF literature. We use the definition of this section in this thesis.

Algorithm 3 Bucket Based PHF

```
1:  function CONSTRUCT(keys)
2:      for all key in keys do
3:          bucketId ← BUCKET(key)
4:          buckets[bucketId].APPEND(key)
5:      buckets ← SORTBYSIZE(buckets)
6:      pilots ← SEARCH(buckets)
7:      encoded ← ENCODE(pilots)
8:      return encoded
9:  function QUERY(key)
10:     bucketId ← BUCKET(key)
11:     pilot ← DECODE(bucketId)
12:     position ← HASH(pilot, key)
13:     return position
```

3. Related Work

This chapter introduces a variety of PHF algorithms. The first three sections consider bucket based approaches. We then proceed with other interesting solutions for PHF construction. Finally, we discuss parallel implementations of PHF algorithms.

3.1. CHD

In 2009, Belazzougui et al. introduced "compressed hash and displace" [13] (called CHD in the following) which is a bucket based approach (Section 2.4). In CHD the keys are uniformly distributed to all buckets. Thus, all buckets have expected equal size. The buckets are inserted in non-increasing size. Insertion of a bucket B_i means that a function f_i is found such that all keys $k \in B_i$ are mapped to positions where no key of a previously inserted bucket is mapped to. CHD uses the additive displacement function $f_i(k) := h(k,0) + d_{A,i} \cdot h(k,1) + d_{B,i} \bmod M$, where $h(k,s)$ is a pseudo random function with seed s. Thus, they have to determine a pair of displacements $d_{A,i}$ and $d_{B,i}$. The advantage of displacement hashing is that the computationally expensive hash functions h only has to be evaluated once. After that, f_i can be evaluated for any displacement pair using only addition and multiplication. When searching such a pair, CHD proceeds in a linear manner: They first use $d_{A,i} = 0$ and try $d_{B,i} = 0, 1, .., M-1$. If no mapping can be found they continue with $d_{A,i} = 1$ and so on. In the last step they encode the values $v_i = M \cdot d_{A,i} + d_{B,i}$ which uniquely identify the pairs. v_i is then stored using Fredriksson-Nikitin [19] encoding, retaining constant time access.

3.2. FCH

In 1992 Fox, Chen and Heath [12] (called FCH in the following) proposed a bucket based approach for MPHF construction. The bucket assignment function partitions the buckets into two sets of buckets. Buckets in the same set have the same expected bucket size. There are two parameters to this assignment function. p_1 controls the number of keys in the sets. p_2 controls the number of buckets used in the sets. Again, a pseudo random function h with seed s_1 is applied. The key $x \in S$ is then assigned to

$$\text{bucket}(x) = \begin{cases} h(x, s_1) \bmod p_2 & h(x, s_1) \bmod N < p_1 \\ p_2 + (h(x, s_1) \bmod (m - p_2)) & \text{otherwise} \end{cases}$$

The threshold p_1 is set to $0.6n$ and p_2 to $0.3n$. Accordingly, roughly 60% of the keys belong to 30% of the buckets. The result of the function is that there are two different expected bucket sizes. This is in contrast to CHD, which only uses a single expected bucket

size. The buckets are then ordered by non-increasing size. In FCH they use an additive displacement $d_i \in \{0, .., M-1\}$ such that $f_i(k) := h(k, s_2 + b_i) + d_i \bmod M$. The global seed s_2 has to be determined such that there are no collisions within all buckets. If a value of s_2 does not work, they have to try a new value of s_2 on all buckets. To speed up search of s_2, they add an additional degree of freedom b_i which is a single bit. They then find d_i for all buckets independently. If none of the M possible displacements work, they can use the other b_i and try all displacements again. If that still fails, they have to restart the search from scratch using a new s. To accelerate the search of a displacement, they keep track of free positions in an array. They can then *align* d_i based on the position $h(k, s + b_i) \bmod M$ of one key relative to a free position. However, they still have to brute-force check if all other keys are now also on a free position as well. Finally the seeds are stored compactly. FCH is outdated compared to the current state-of-the-art.

This thesis proposes a novel acceleration structure, which allows aligning not a single key but an entire bucket.

3.3. PTHash

Recently, Pibiri and Trani introduced PTHash [3] which is based on FCH with several improvements. The bucket assignment function and the ordering step of FCH is maintained. They made several changes to the searching and encoding steps. First of all, during search, PTHash uses the hash function $f_i(k) := h(k, s) \oplus h(d_i, s) \bmod M$. Compared to FCH, they still have to find a global seed s such that all keys k of the same bucket map to unique values of $h(k, s)$. However, they no longer need an additional degree of freedom b_i for each bucket. The reason for this is that PTHash uses XOR ($\oplus$) as displacement operator. Recall that in FCH, they used an additive displacement. This means that if two keys of the same bucket share the same value of $h(k, s) \bmod M$ they cannot be inserted with that seed. That is because any additive displacement still results in the same position of those keys. Now in PTHash, the XOR operator ensures randomness of the positions. Note that the displacement value $h(d_i, s)$ only has to be calculated once for every d_i. The new positions of all keys of that bucket can then be calculated using only a single XOR and modulo operation. In PTHash it was observed that most of the execution time is spent with inserting the last few buckets. PTHash found a way to work around this issue with only small costs in terms of query time. The idea is to enlarge the domain N to a larger space M to increase the probability that a position is free. The increased probability is particularly significant for the last few insertions. They can retain minimality of the PHF by applying an additional function to keys which were mapped outside of N during search. The function maps those keys back to free positions inside N. Note that this works because for any key that is mapped outside of N, there is an additional free position inside of N.

Finally, PTHash also experiments with several different encoding techniques to store the pilots in a compressed manner. Usually, a decreased space consumption of one encoding comes at the cost of an increased query time. This allows choosing a tradeoff between space consumption and query time. Compared to FCH and CHD, PTHash has better construction speed and faster queries. Only CHD is slightly more space-efficient.

In this thesis we show that the high execution time for the last few buckets can be completely avoided by optimizing the distribution of bucket sizes. An initial experiment suggests that this improvement is applicable to PTHash.

3.4. Fingerprints

In Fingerprint-Based perfect hashing (FMPH) [20, 21] keys are first mapped to a bit array using a hash function. If only a single key maps to a certain position, it is set to one. Otherwise, if zero or multiple keys (i.e. a collision) map to a position it is set to zero. Any keys participating in a collision perform this procedure recursively on a new bit array in a next *level*. Finally, the bit arrays of all levels are concatenated and equipped with a constant time ranking structure. The querying algorithm has to traverse all levels until the key is mapped to the position of a one in the bit array. Determining the rank of this position yields the result. Varying the size of the bit arrays relative to the keys participating in the respective level provides different tradeoffs in the three measures. FMPH requires at least 3 bits per key in practice [21]. An average of just 1.5 level accesses per query results in a fast query time, but the queries are still slower compared to bucket based approaches, which only require a single memory access.

3.5. RecSplit

In RecSplit [1] the keys are first distributed into partitions of expected equal size. Note that we avoid terminology confusion by replacing the word "bucket" of the original paper with "partition", which is more coherent within this thesis. For each partition they search independently for a hash function that splits the keys into subsets of specified size. This splitting strategy is applied recursively on the subsets, materializing a *splitting tree*. Once the leaves have sufficiently small size, they brute-force search for a bijection inside the leaf. The query algorithm has to read the offset of the partition, traverse the splitting tree and apply the bijection seed.

By varying the partition and the leaf size, they can achieve a tradeoff between the three measures. RecSplit is of particular interest when near optimal space is required. The space usage was even further improved in ShockHash [2], which is built on RecSplit. However, bucket based approaches usually have faster query speed.

3.6. Linear Systems

An MPHF $f(x)$ can be constructed by solving a linear system of equations [22]. The system consists of equations in the form of

$$w_{h_1(x)} + w_{h_2(x)} + \ldots + w_{h_r(x)} = f(x) \mod n,$$

where $h_i : S \to [m]$ is a random hash function, and h_i are m variables whose values are in $[n]$. The system can be solved in linear time if the ratio between the number of variables

and equations is large enough. They then only have to store the values of w_i which is the solution of the system. Queries can be answered in constant time by evaluating the equation. Perfect hashing based on linear systems is not competitive with state-of-the art bucket based approaches.

3.7. Parallel Implementations

PHF construction is in the class of "embarrassingly parallelizable" problems. In fact, *any* PHF construction algorithm can be parallelized. One approach is distributing the keys into *partitions* of expected equal size. The PHF construction algorithm can then be applied independently on each partition. The only additional cost is that we have to store and query the offset of each partition.

RecSplit is already using such partitions anyway. Those partitions are originally intended for reducing the splitting tree height. This means RecSplit construction can be parallelized without additional cost in terms of space consumption or query time. Bez et al. [23] provides both a parallel CPU and a GPU implementation which makes use of this fact. The SIMD and GPU implementations achieve significant speedups compared to the original CPU implementation. To the best of our knowledge, RecSplit is the only MPHF construction technique which has a GPU implementation.

However, there are several parallel CPU implementations. PTHash-HEM [24] employs the partitioning approach. They also looked into another parallelization technique for PTHash. First, they spawn K threads. The threads then search independently for pilots of K consecutive buckets. Once a pilot is found, the thread pauses until it can commit its work. Once the work is committed, it continues with the next bucket. More details can be found in the original paper.

BBhash [25] is a FMPH technique which presents a CPU parallel implementation. Among other things, they use atomic-or functions to concurrently fill the bit arrays of the different levels.

Finally, [26] models MPHF construction as a SAT instance. Although theoretically interesting, their results show that the approach scales exponentially in the input size. However, the problem of SAT solving is extensively researched and there are countless parallel solvers both for GPU and CPU. Such a solver could be applied to parallelize the SAT-based PHF approach. But to the best of our knowledge there is no such specific implementation.

4. Working Principle

The algorithm is given an input set of N keys called S taken from a universe U. In the following we discuss the working principle which is built on the generic bucket based approach specified in Section 2.4. Beside brute force search we also consider three other search approaches. The first approach is based on the aligning technique of FCH. An algorithm is presented in this thesis that allows aligning an entire bucket instead of individual keys. The second approach is to limit the pilot value, in an effort to speed up the query time without excessive space requirements. Finally, we outline a search technique as an idea for future work. However, the main focus of this thesis is brute force search.

4.1. Bucket Assignment and Ordering

In preparation of the searching step, the input keys are assigned to buckets and the buckets are sorted.

Bucket Assignment. The keys are distributed into buckets using a bucket assignment function. In the related work section we discussed three bucket based approaches. CHD uses buckets of equal expected size. FCH and PTHash use two different expected bucket sizes. Our implementation uses an optimized bucket assignment function which is derived in Chapter 5 after we have established the necessary foundations in this chapter.

Bucket Sorting. The buckets are sorted by non-increasing size. Sorting is crucial for reducing the construction time. A bucket is more difficult to insert if it has more keys. Especially, if the ratio between the number of free positions and N is already low. It is therefore more efficient to insert the larger buckets earlier. This is done in all of the three bucket-based approaches (CHD, FCH and PTHash). We observed that the insertion order of buckets which have equal size decisively impacts the total space consumption. We consider different insertion orders for buckets of equal size experimentally in Section 7.2.3.

4.2. Displacement Hashing

We now discuss the hash function which is used in the searching step. For each bucket $B_i \subset S \subset U$ we have to find a function $f_i(k) : U \rightarrow [M]$ that maps all keys $k \in B_i$ to free positions without collisions. The number of positions is M. A *local collision* occurs when multiple keys of the same bucket map to the same position. We define $h(v, s)$ as a hash function with seed s and some input v.

To map a key k to a position we use a function $f_i(k)$ very similar to FCH. But instead of a global hash seed we use a separate seed $s_i \in \mathbb{N}_0$ and an additive displacement $d_i \in [M]$ for each bucket. We define for the remainder of the thesis:

$$f_i(k) := h(k, s_i) + d_i \bmod M$$

A *pilot* $p_i \in \mathbb{N}_0$ that uniquely identifies a pair of s_i and d_i is defined as

$$p_i := s_i \cdot M + d_i$$

Note that our implementation employs "partitioning", which means that M is of small constant expected size. Hence, the space of the pilot is independent of the actual input size.

The main advantage of displacement hashing is that the computationally expensive hash functions h only has to be evaluated once for every s_i. We refer to $h(k, s_i) \bmod M$ as the *initial position* of key k and seed s_i. After determining the initial positions, f_i can be calculated for all displacements with computationally inexpensive addition and subsequent modulo M operation. The modulo operation can be reduced to a conditional subtraction of M, because the sum of the initial position and the displacement will never exceed $2 \cdot (M - 1)$. Another advantage of additive displacement is that we only have to check the initial positions for local collisions. Any additive displacement does not change whether two keys map to the same position or not.

In our case there are also some hardware specific advantages to additive displacement hashing which are discussed in the implementation chapter (Section 6.4.5.4).

4.3. Brute Force Search

We now look at the first search algorithm which is based on brute force search of the pilot values. Brute force search is also applied by CHD and PTHash. First, we allocate an array $free$, which stores for each position if it is occupied or free. Initially, all positions are free. The buckets are inserted in the order of the sorting step. For each bucket B_i we try out seeds $s_i \in \{0, 1, ..\}$ and displacements $d_i \in [M]$ linearly until a pilot $p_i = s_i \cdot M + d_i$ is found. We say that a pilot is found if there are no local collisions and the positions $f_i(B_i)$ are free. As mentioned previously, we only have to check for local collisions once for every s_i. Once a pilot p_i is found, we mark the according positions $f_i(B_i)$ as occupied in the array $free$. Note that all keys of a bucket are mapped using the *same* pilot. In the query algorithm we only access the pilot of the key's bucket and there is no data stored on a per-key basis.

Algorithm 4 describes the brute force search algorithm in pseudo code. When searching for a pilot, the outer loop iterates linearly over the seeds s. The initial positions are calculated in each iteration. Note that there are many ways on how hasLocalCollisions(*positions*) can be implemented. The optimal algorithm depends on the underlying hardware, the size of the bucket and M. We will look at different implementations in Chapter 6. The next loop iterates over all possible displacement values d. If it turns out that the positions for all keys of a bucket are free, we can return the pilot $M \cdot s + d$. The early exit condition in the for loop (line 18) is crucial to reduce the search time, but represents a

challenge for parallelization on a lock-step architecture. In the implementation chapter (Section 6.4.5), we show how the nested loops (line 13 and 15) can be parallelized without causing excessive divergence while maintaining the early exit optimization. We analyze this algorithm in detail in the next chapter.

Algorithm 4 Brute Force Search

```
1:  function SEARCH(buckets)
2:      free ← [True] · M
3:      for all bucket in buckets do                    ▷ In non-increasing order
4:          pilot ← SEARCHBUCKET(bucket, free)
5:          pilots.APPEND(pilot)
6:      return pilots
7:  function SEARCHBUCKET(bucket, free)
8:      for all s in ℕ_0 do
9:          initialPositions ← [f(k, s · M) for k in bucket]
10:         if HASLOCALCOLLISIONS(initialPositions) then
11:             s ← s + 1
12:             continue                    ▷ Local collision. Check next seed.
13:         for all d in [M] do                    ▷ Try displacements linearly.
14:             canAssign ← True
15:             for all pos in initialPositions do
16:                 if !free[pos + d mod M] then
17:                     canAssign ← False
18:                     break      ▷ One position was occupied. Try new displacement.
19:             if canAssign then                              ▷ A pilot is found.
20:                 for all pos in initialPositions do
21:                     free[pos + d mod M] ← False      ▷ Mark all positions occupied
22:                 return M · s + d                           ▷ return the pilot
```

4.4. Indexed Search

In order to speed up the brute force search FCH keeps track of free positions. Because of the additive displacement, *one* key of a bucket can be aligned to one free position by calculating the displacement d as the difference between the initial position of the key and one of the free positions. However, FCH still has to check if the positions of the other keys are on free positions as well. In this chapter we propose an approach to align *all* keys of a bucket. We provide a GPU implementation and evaluation of the approach in the respective chapters.

4.4.1. Pattern Index

We define $\Phi(d) := \{p \mid p \text{ is free} \land (p + d) \bmod N \text{ is free}\}$. In other words $\Phi(d)$ is the set of all free positions p, such that $p + d \bmod M$ is a free position as well. The *pattern index* stores the result of $\Phi(d)$ for all defined inputs d explicitly. We now derive the domain of Φ. For each of the F free position, there are a total of $F - 1$ other free positions which all result in an entry. We do not store entries redundantly, i.e. if p is an entry because $p + d_1 \bmod M$ is free, we do *not* have to store $p + d_1 \bmod M$ with $d_2 = M - d_1$ because $p + d_1 + d_2 \bmod M = p$ is already covered by the index. Our way to achieve this is to only consider values d smaller or equal than $\lceil M/2 \rceil$. Thus, the index comprises $F(F-1)/2$ entries and is defined for $d \in \{1, .., \lceil M/2 \rceil\}$.

The initial positions of a bucket for a certain seed s can be interpreted as a fully connected directed graph. Each node corresponds to the initial position of one key. Each weighted edge corresponds to the difference between the adjacent initial positions. The direction of the edge is chosen such that d stays within the domain of Φ. A *search pattern* is an open walk which covers all nodes of the graph. A walk is a sequence of nodes $(v_0, .., v_k)$ which are connected via $(e_1, .., e_k)$. We will discuss how the walk is selected later. The search for a displacement proceeds as follows. We start at the first edge e_1 of the walk. We lookup the corresponding difference d_1 of the edge in the pattern index. The pattern index provides a set of free positions $I_0 = \Phi(d_1)$. We can now align v_0 and v_1. If the walk is in the direction of e_1, then $D'_0 = I_0 - v_0 \bmod M$ are the displacements which align v_0 and v_1 to free positions. If the walk is in the opposite direction of e_1, we use $D'_0 = I_0 - v_1 \bmod M$ as possible displacement values. We now perform the lookup for e_2 with difference d_2. Again, we are provided with positions $I_1 = \Phi(d_2)$ such that v_1 can be placed on I_1 and v_2 can be placed on $I_1 + d_2$ (as always, reversed if the walk is in the opposite direction of the edge). We set $D_1 = I_1 - v_1$. We now have the sets D'_0 and D_1. Displacements in D'_0 align v_0 and v_1 to free positions. Displacements in D_1 align v_1 and v_2. To align all three positions at once, we only consider displacements which are in *both* lists. Accordingly, $D'_1 = D_1 \cap D'_0$ and we continue until D'_{k-1} is reached. If D'_{k-1} is empty, we have to restart the algorithm with an incremented seed s and respective new initial positions. Otherwise we have found a displacement value such that all initial positions align to free positions. If there are multiple viable displacements in D'_{k-1} we choose the smallest value to reduce entropy. We demonstrate the procedure on an example in Figure 4.1.

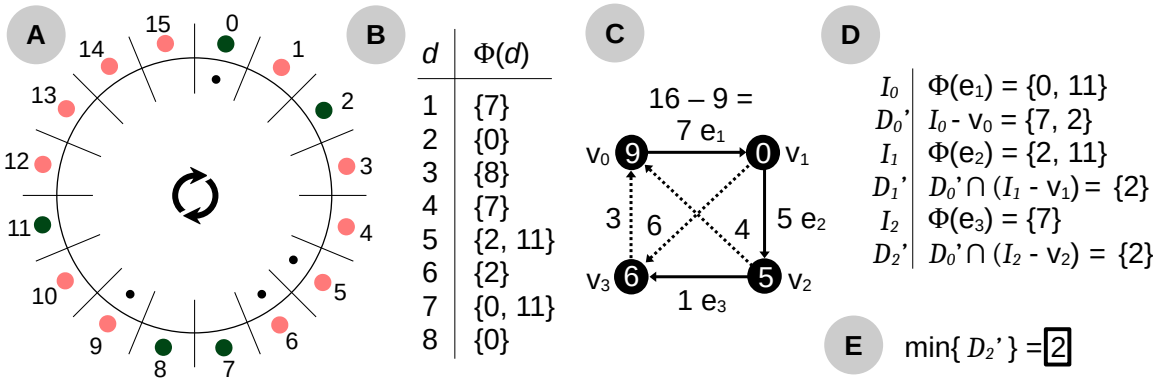

Figure 4.1.: We show an example with M = 16. In *A*, the outer ring marks the 5 free positions with dark green color and the occupied positions in light red color. The inner ring shows the initial positions of 4 keys. Note that an additive displacement with subsequent modulo operation can be seen as a rotation. We now want to rotate the inner ring such that the initial positions align with free positions. *B* shows the content of the pattern index. *C* shows the pattern graph. The chosen walk is marked by continuous lines. *D* shows the calculations to finally find the correct displacement value in *E*.

Selection of the walk is an optimization problem. For each step, the algorithm has to fetch a set I_i from the pattern index. Furthermore, a list intersection has to be calculated. And last but not least, if a set I_i is empty, we can stop early. Thus, it is crucial to keep $|I_i|$ low. The amount of possible walks through a fully connected graph is enormous and we can only consider a small subset in practice.

4.4.2. Concrete Algorithm

We show a heuristic for quickly finding a good walk. First, we restrict our walk to a path, i.e. nodes cannot repeat themselves. Then, the algorithm starts by looking at a random subset of all edges. The size of the subset is a parameter to be optimized. We start with the edge in the subset which is yielding lowest $|I_0|$. From v_1 on, we continue greedily by always selecting an adjacent edge of v_i with minimal $|I_i|$.

Once a bucket is inserted, we have to consider the consequences for the pattern index and have two options. (I) We keep the index but each time a free position is fetched from the index we check if the position is really still free. (II) We discard the index and regenerate it completely. In our implementation we will consider a combination of both approaches: We always check if the positions are still free and after a certain amount of insertions, we regenerate the index. A third option would be to have a dynamic index, which is not part of this thesis.

4.4.3. Analysis

The space consumption and construction time of the index is in $\theta(F^2)$, because it stores all pairs of the F free positions. In the following we look at the search time. The probability that a certain pilot is viable is $(1-\alpha)^b$, where α is the ratio of the number of occupied positions relative to the total number of positions and b is the bucket size. This is again assuming negligibility of local collisions. For each seed s, there are a total of M possible displacement values. Thus, the probability that one seed will not result in a viable pilot is $P \coloneqq 1-(1-(1-\alpha)^b)^M$. Because the probabilities are independent, the expected amount of seeds that have to be tried is $1/P$.

Recall that the index comprises $\theta(F^2) = \theta(((1-\alpha)\cdot M)^2)$ entries. For simplicity, we leave out the optimization that the search algorithm picks a minimal edge. Instead it picks a random edge with difference d_1 between the adjacent initial positions. $|\Phi(d_1)|$ is Poisson distributed with expected value in $\theta(\frac{(1-\alpha)\cdot M^2}{M}) = \theta((1-\alpha)^2 \cdot M)$. We then pick d_2 at random, retrieving $|\Phi(d_2)|$ entries. We now perform the list intersection using the naive $\theta(X \cdot Y)$ comparison algorithm, where X and Y are the sizes of the lists. $|\Phi(d_1)|$ and $|\Phi(d_2)|$ are independent. Thus, the expected time for the first list intersection is in $\theta((1-\alpha)^4 \cdot M^2)$. For each list entry, the probability that the entry is also present in the other list is $1/M$. Asymptotically, this probability is independent for each entry. The expected size of the first intersection list is therefore in $\theta(\frac{(1-\alpha)^2\cdot M}{M}) = \theta((1-\alpha)^2)$. The second list intersection has expected time of $\theta((1-\alpha)^4 \cdot M)$ which is the product of the size of the resulting list from the previous intersection and the size of the list $\Phi(d_3)$. Each following intersection time is reduced by a factor of M. There are $b-1$ intersections. Hence, the search time for one displacement is approximated using

$$\sum_{n=0}^{b-2}(1-\alpha)^4 \cdot M^{2-n} < M^2(1-\alpha)^4 \sum_{n=0}^{\infty}\frac{1}{M^n} = \frac{M^2(1-\alpha)^4}{1-\frac{1}{M}}$$

For large M, we only have to check a single seed and the total expected search time for one bucket is in $O(M^2(1-\alpha)^4)$. This is an entirely different behavior compared to brute force search which requires expected time of $\theta(\frac{1-(1-\alpha)^b}{\alpha(1-\alpha)^b})$ which we will see in the later analysis of brute force search. From an asymptotic perspective, the indexing approach looks like a good option for the last buckets, when α is high. Our experiments involve a combination of both approaches. The search commences with brute force search and switches to index sort for the last few buckets.

Future Work: Sparse Index. Previously we interpreted the initial positions as a fully connected graph. A walk was used to traverse through the graph. Selection of the walk in a fully connected graph has an enormous and unused degree of freedom. We can decrease the domain of Φ to reduce the large index size of size $\theta(F^2)$. This effectively removes edges of the graph. However, we can still find a walk as long as the graph remains connected. One strategy to reduce the domain is to randomly remove each value from the domain with a probability p. The graph for a bucket of size b remains connected with constant probability if $p \in \theta(\log(b)/b)$ [27]. Future work should experiment with a decreased index of size $\theta(F^2 \log(b)/b))$.

4.5. Randomized Search in Pilot Limited Space

An issue we will encounter during the encoding phase is a tradeoff between space consumption and query speed. One can apply a variable space encoding for storing a pilot, which requires little space but has a slow query speed. The other option is to use a constant amount of space for a pilot which allows us to access it directly. In the following, a new technique is proposed, which aims at retaining fast query time and small space at the cost of construction time. The challenge of the technique is to find pilot values within a predefined space constraint to allow direct access.

We start processing the buckets in non-increasing size as usual. For example, one can apply the brute force search algorithm. Only a small modification to the brute force search is required: If no viable pilot can be found within the pilot space a different insertion strategy is applied. We resort to inserting the bucket *with collision*. The collisions have to be resolved to retain a perfect hashing. This is done by reversing the insertion of all other buckets involved in the collision. They are reinserted using the brute force approach. If no pilot can be found for reinsertion, the algorithm is applied recursively. Thus, the buckets are no longer necessarily inserted in non-increasing size. In the following we call the bucket causing the collisions A and the pilot we choose for A is called p. The set of buckets that have to be reinserted is called B_p. The interesting part of this algorithm is choosing p such that minimal construction time overhead is caused. Considering the recursive consequences of choosing a certain pilot p to its fullest extent does not seem viable. Instead, a heuristic is applied to choose a pilot value. We want to avoid reinserting large buckets if only a small number of free positions is available. One idea is choosing the pilot which causes minimal expected total insertion time by the following heuristic $p = argmin_{p'}\{\sum_{b \in B_{p'}} c(b)\}$. $c(b)$ is a heuristic describing the cost for reinserting a bucket b. In Equation 5.2 we show how such a heuristic can work. However, there are several issues with that approach: (I) The algorithm can get stuck in an infinite loop. For example, if one bucket causes reinsertion of another bucket, which in turn causes reinsertion of the first bucket. (II) The pilot space can be large and evaluating the heuristic for all pilots is too expensive (III) $c(b)$ requires knowledge about the insertion probability, but the heuristic is unaware of recursive re-insertions and thus cannot know the insertion probability at the time the bucket will get re-inserted. (IV) The current formula for $c(b)$ involves computationally expensive exponentiation.

We resolve issue (I) and (II) by only considering a random subset of the pilot space. Issues (III) and (IV) are solved by using an easy to compute approximation of c namely $c(b) = 2^s$. Where s is the size of b and 2^s can be computed using a bit shift. It might seem that this approximation is too strong, but we observed that it works well in practice.

It is possible that no solution can be found within the pilot constraint. One reason could be that all pilots of a bucket cause a local collision. If such a condition is detected, the search algorithm restarts from scratch. A *restart counter* is incremented. A certain offset is added to all pilots depending on the restart counter, which ensures that only new pilots are tested. Another reason to restart the search is if a certain number of reinsertions are exceeded, possibly indicating that there is no viable solution within the pilot space. Unfortunately, the restart counter represents additional data that has to be stored and accessed for each query.

4.6. Future Work — Dynamic Bucket Search

Like previously, limiting the pilot space allows direct access to the individual pilots. In the following we present another approach for searching within the pilot space. It is a suggestion for future work and we do not provide an implementation. The idea is to gradually reduce the expected bucket size while searching to increase the probability that a pilot is viable. We now show how this could be accomplished. Brute force search is used and the pilots are tried linearly as usual. We define that a pilot p is within a predefined limited pilot space $[S]$. The function $\psi : [S] \rightarrow [0, 1]$ decides if a key k is still participating in the search based on its normalized hash value $h_k \in [0, 1)$. When the brute force search is testing pilot p, it will only consider the keys k for which $h_k < \psi(p)$. Therefore, $\psi(p)$ dynamically changes the expected bucket size. In the beginning all keys participate and therefore $\psi(0) = 1$, because zero is the first pilot we try. It has to be guaranteed that at least one pilot is found within the limited pilot space. This is achieved by setting $\psi(S - 1) = 0$. This means that if no pilot $p < S - 1$ worked, we store $S - 1$ because the dynamic bucket size is zero. Once a pilot p is found then all keys which were not included ($h_k \geq \psi(p)$) have to be considered in a next *level*. One idea is to apply this algorithm recursively on those keys resulting in multiple levels. Another idea is to use the usual brute force search without pilot limit in the second level.

The query algorithm for some hash h_k remains simple. We directly access the pilot p of the corresponding bucket. If $h_k < \psi(p)$, we use the pilot as usual to map to a position. If otherwise $h_k \geq \psi(p)$ we proceed with the query algorithm of the next level.

Note that when encoding the pilot it will always use the same number of bits $\lceil \log_2(S) \rceil$ to allow direct access (compact encoding). S should clearly be a power of two, such that no space is wasted. According to Shannon entropy [28], compact encoding is space optimal if and only if each pilot has the same probability of occurring because each pilot has the same code length in the compact encoding. Smaller pilots are more likely, because we try pilots linearly. However, now that we can choose $\psi(p)$ we can increase the probability that a later pilot is viable by decreasing the expected bucket size with increasing pilot value. $\psi(p)$ should therefore be strictly monotonically decreasing. Part of future work is finding $\psi(p)$ to ensure constant probability of pilots. However, another optimization aspect of $\psi(p)$ is the expected number of keys which will get mapped in the first level. This number should be high such that only the first level is accessed with high probability, consequently reducing query time. Apparently, $\psi(p)$ provides a tradeoff between space consumption and query time. It would be interesting to explore that tradeoff and of course implement the described algorithm in future work.

4.7. Encoding

Once all pilots have been found in the searching phase, we have to store them in a compressed manner. As we will see, encoding is a tradeoff between space and query time. The time it takes to encode pilots is usually negligible. In the following we look at several encodings. Except for the Golomb encoding they were all used in PTHash [3].

Compact. All pilots are stored using the same bit length l. We use $l = \lceil \log_2(p_{max} + 1) \rceil$ such that the largest pilot p_{max} can still be encoded. The i-th pilot can be accessed by reading bits $i \cdot l$ to $(i + 1) \cdot l - 1$. Thus, allowing direct access. The downside is that the required space does not scale linear with the amount of pilots. Each time a pilot is appended, there is a chance that a larger value of l is required. Consequently, increasing the space usage of all pilots. We provide a more detailed analysis of the asymptotic space usage of compact encoding in the implementation chapter (Section 6.5.2.1).

Dictionary. The dictionary is a list of all distinct pilot values. We can then encode a pilot using reference to the value stored in the dictionary. Compact encoding is applied to store both the dictionary values and the references.

Golomb-Rice. Golomb-Rice code [29] splits a value v in two parts. The $\tau \in \mathbb{N}_0$ lower bits is the *fixed* part. The other bits are the *unary* part. To encode the pilots, the fixed part of each pilot is stored in a compact encoding. The unary part is concatenated and a constant time rank structure is built to allow direct access of the individual values. Golomb codes provide optimal (i.e. closest to entropy) space consumption for geometric distributed values. In the implementation chapter we show how τ is determined. Note that τ is not necessarily the same for all pilots.

Others. We also applied Elias-Fano encoding [30]. Elias-Fano encoding is a succinct data structure to store monotonically increasing numbers. Elias-Fano is applied on the prefix sum of the pilots. The pilots can still be accessed using the difference of two consecutive entries. Finally, we also tested "Simple Dense Encoding" by Fredriksson and Nikitin [19] which was applied in CHD. CHD is so far the most compact bucket based approach.

5. Brute Force Search Analysis

In the following we analyze the pilot value and the execution time of brute force search. This allows us, based on a few approximations, to find an optimal distribution of bucket sizes. In the following we use the symbols b for the size of a bucket, F as the number of free positions, N as the total number of keys, B as the total number of buckets, $A := \frac{N}{B}$ as the average bucket size and M as the number of output positions.

5.1. Pilot

In the following we derive the expected value for a pilot of a specific bucket. We assume that $b \in o(\sqrt{M})$ which allows us to neglect local collisions. The asymptotic bound of b is a consequence of the birthday paradox [31]. Consequently, each key of one bucket has the same probability $\frac{F}{M}$ of being mapped to a free position. We apply the full randomness assumption, which means that the position of the keys are statistically independent. Thus we can multiply the probabilities to find the probability of mapping all keys to free positions for a single pilot

$$p = \left(\frac{F}{M}\right)^b$$

We assume that each trial (trying a pilot) is independent, with constant probability p of success. Accordingly, we can model the pilot as being geometrically distributed. The expected value for the pilot and for known F and b is therefore

$$\mathbb{E}[pilot \mid F, b] = \left(\frac{F}{M}\right)^{-b} \tag{5.1}$$

It is important to consider that b and F are random variables as well because the bucket size is Poisson distributed and F is not known before the sorting step. Consequently, the distribution of the pilot is a linear combination of geometric distributions.

5.2. Execution Time

In the following we analyze the execution time for inserting one bucket. Again, we assume that local collisions are negligible. We even go one step further and assume that the entire searching time for a bucket is spent in line 16 of Algorithm 4. In line 16 we check if the position of one key for one pilot is free. We refer to one execution of line 16 as one *search iteration*. If the position is free, we refer to this as a success. The key argument for deriving the execution time is that the pilot is found once b *consecutive successes* are

observed. Because if a key collides, we can immediately restart with a new pilot and the first key, until all keys do not collide. The constant and independent success probability of a trial, i.e. a search iteration, is $p = \frac{F}{M}$. The formula for the expected number of trials for b consecutive successes is known [32]. We can now state the expected number of search iterations

$$\mathbb{E}[T_{search} \mid p, b] = \frac{1 - p^b}{(1 - p)p^b} \tag{5.2}$$

Based on our assumptions, this is proportional to the expected search time of the bucket.

5.3. Optimal Distribution

For the following analysis we assume $N = M$ and that the user specifies the average bucket size $A = N/B$. We now want to find the *bucket assignment* function $bucket : [0, 1) \to [0, 1)$. The bucket assignment function uses the normalized hash value to assign a normalized bucket. E.g. given a uniformly distributed hash $h \in [2^{64}]$ and the number of buckets B, the hash is assigned to bucket $\lfloor B \cdot bucket(h/2^{64}) \rfloor$.

We define the *bucket size* function $b : [0, 1) \to \mathbb{N}$ which describes the bucket size $b(\frac{i}{B})$ of each bucket $i \in [B]$, where i is the index of the bucket after the sorting step. We also define the *fill* function $\alpha : [0, 1) \to [0, 1)$. The value $\alpha(\frac{i}{B})$ is the ratio between the number of occupied positions and N when inserting bucket i. Accordingly, $1 - \alpha(\frac{i}{B})$ is the probability that a random position is free when inserting bucket i. Notice that α is the sum of all previously inserted bucket sizes divided by the total number of positions N. Vice versa, $b(\frac{i}{B}) = N \cdot (\alpha(\frac{i+1}{B}) - \alpha(\frac{i}{B}))$ holds. The reason for normalizing everything will become apparent with

$$b\left(\frac{i}{B}\right) = N \cdot \left(\alpha\left(\frac{i+1}{B}\right) - \alpha\left(\frac{i}{B}\right)\right) \tag{5.3}$$

$$= \frac{N}{B} \cdot \left(B \cdot \left(\alpha\left(\frac{i}{B} + \frac{1}{B}\right) - \alpha\left(\frac{i}{B}\right)\right)\right) \tag{5.4}$$

$$\implies \lim_{B\to\infty} b\left(\frac{i}{B}\right) = \lim_{B\to\infty} A \cdot \alpha'\left(\frac{i}{B}\right) \tag{5.5}$$

while we assume that the limit exists. We can now express the bucket sizes as the derivative of α if B approaches infinity. Note that N approaches infinity as well, because the user specifies the constant $A = \frac{N}{B}$. Consequently, we are allowed to neglect local collisions.

In the following we first show how to numerically optimize $\alpha()$ for minimal construction time. We then reincorporate local collisions in Section 5.3.2. We present a closed form solution for $\alpha()$ in Section 5.3.3 which results in a constant expected pilot value. Interestingly, it also serves as an approximation for the numerical solution of minimal execution time. Finally, we show how the solutions for $\alpha()$ translate to a bucket assignment function $bucket()$ in Section 5.3.4.

5.3.1. Minimal Construction Time

The first step is to express the expected number of search iterations of a single bucket with normalized index $x \in [0, 1)$ based on $\alpha()$. In Equation 5.2 we described the expected number of search iterations based on the size of the bucket and the probability that a random position is free. We can now describe both of these parameters in terms of a single function $\alpha(x)$. By definition, the probability that a random position is free is $p = 1 - \alpha(x)$. We derived in Equation 5.5 that the bucket size is $s = A \cdot \alpha'(x)$. We substitute the two parameters into Equation 5.2 to describe the expected number of search iterations for the bucket with normalized index x in terms of the *Lagrangian*

$$L(x, \alpha(x), \alpha'(x)) = \frac{1 - (1 - \alpha(x))^{A \cdot \alpha'(x)}}{\alpha(x) \cdot (1 - \alpha(x))^{A \cdot \alpha'(x)}}. \tag{5.6}$$

We now want to minimize the total expected construction time, which is the sum of the expected search times of all buckets. Recall that the number of search iterations is proportional to the construction time. Optimizing search iterations is therefore identical to optimizing for construction time. Furthermore, it is more convenient and again equivalent in terms of the optimization to minimize the average expected search time of all buckets instead of the total expected search time. We therefore divide by the bucket count B. We express the average expected search time in terms of a functional $S[\alpha]$. Since we are looking at an infinitesimal case, the sum 5.7 is equal to the integral 5.8.

$$S[\alpha] = \lim_{B \to \infty} \sum_{x=0}^{B-1} L\left(\frac{x}{B}, \alpha\left(\frac{x}{B}\right), \alpha'\left(\frac{x}{B}\right)\right) \frac{1}{B} \tag{5.7}$$

$$= \int_0^1 L(x, \alpha(x), \alpha'(x))\, dx \tag{5.8}$$

We apply the Euler-Lagrange equation to minimize the average expected construction time $S[\alpha]$. However, we are only allowed to do so if α is in C^∞. This is currently not the case because the bucket sizes $A \cdot \alpha'(x)$ are restricted to natural numbers. Before we can continue, we have to relax the optimization problem by allowing bucket sizes in $\mathbb{R}$. Only now can we state that the function α is a stationary point of S if and only if

$$\frac{d}{dx}\left(\frac{\partial L}{\partial \alpha'}\right) - \frac{\partial L}{\partial \alpha} = 0$$

We apply Mathematica [33] to determine the Euler-Lagrange equation. The closed form expression is so long that we do not show it here. The differential equation is solved with the boundary conditions $\alpha(0) = 0$ and $\alpha(1) = 1$ because at the beginning all positions are free and at the end all positions are occupied. Unfortunately, the differential equation is numerically unstable when using those boundary conditions. To avoid this issue we set $\alpha(0) = \epsilon$ and $\alpha(1) = 1 - \epsilon$ where ϵ is a very small positive value. We then find a numerical solution for $\alpha()$ using Mathematica. The resulting function can be seen in Figure 5.1.

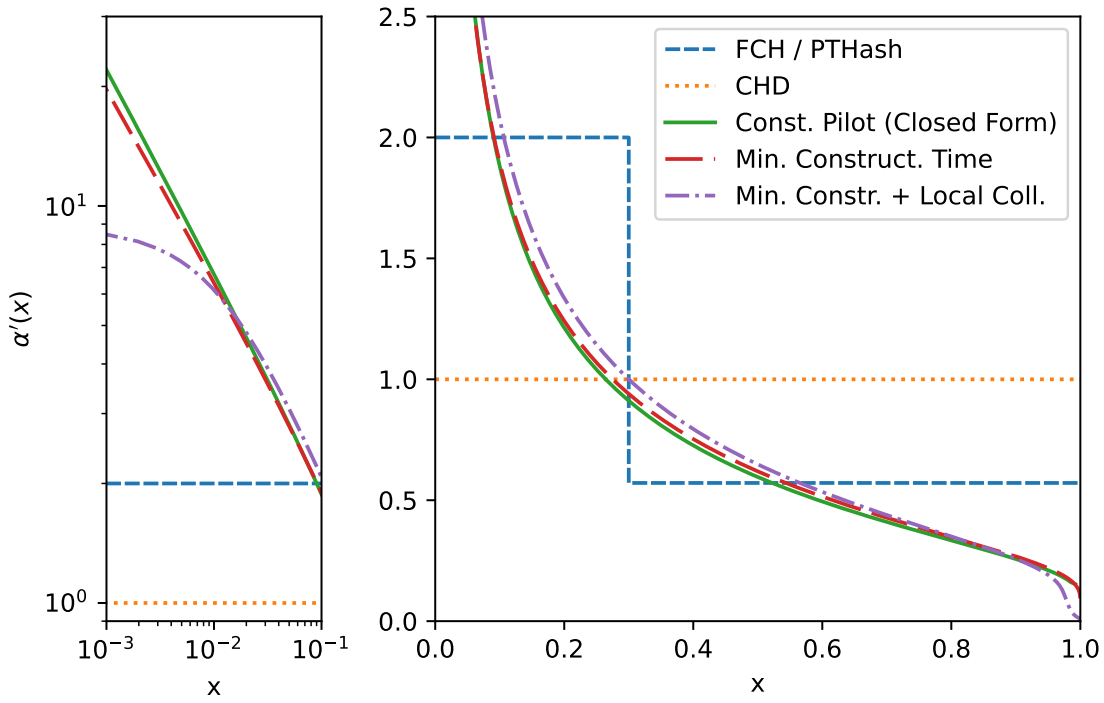

Figure 5.1.: The resulting functions in terms of $\alpha'(x)$. To recall, $\alpha'(x)$ is the expected bucket size relative to the average bucket size A for normalized bucket index x. The left plot shows the section for the first large buckets in logarithmic scale. The right plot shows the entire domain in linear scale. We show state-of-the-art functions. CHD uses the same expected bucket size for all buckets. FCH and PTHash use two different expected bucket sizes. In this plot $A = 10$. Other practical values of A result only in marginal different functions. We use $N = 2048$ for the function which takes local collisions into account.

5.3.2. Considering Local Collisions

Until now we assumed that local collisions are negligible. However, our parallelization technique splits the MPHF instance into many smaller MPHF partitions. This means that local collisions are not negligible, especially for large average bucket sizes. Note that we attempted to incorporate local collisions into Equation 5.2 by decreasing the probability of a free position for insertions of the same bucket (birthday paradox). We were not able to find a closed form solution even when applying the Stirling approximation. Instead, we use a simplistic and effective work around. Previously, a bucket x had to deal with the probability $p = 1 - \alpha(x)$ that a position is free. $\alpha(x)$ is the relative number of occupied positions *before* bucket x is inserted. We now set $p = 1 - \alpha(x) - \frac{A \cdot \alpha(x)}{N}$, which is the probability that a position is free *after* x is inserted. This approximation is good enough to stop the creation of buckets which are too large to be efficiently inserted because of local collisions. Note that the exact solution for Equation 5.2 has to be somewhere in between the two variants for p. We applied this modification to Equation 5.6 and again used Mathematica to solve the Euler-Lagrange equation numerically. The resulting function

can be seen in Figure 5.1. The decisive difference can be seen in the left plot. The largest bucket size is much smaller compared to the original solution. Hence, local collisions are manageable.

5.3.3. Constant pilot

It is beneficial to have a constant expected pilot value if the compact encoding is applied. As an explanatory counterexample: Consider a single bucket which has a much higher pilot value than all other buckets. The one bucket increases the width required to store the pilots of all buckets. Thus, resulting in a higher space usage compared to equal pilots. We apply Equation 5.1 to calculate the expected pilot of a bucket. $p = 1-\alpha(x)$ and $s = A \cdot \alpha'(x)$ is substituted for insertion probability and bucket size respectively. The expected pilot is set to a constant to obtain the differential equation:

$$\frac{1}{(1-\alpha(x))^{A\cdot\alpha'(x)}} = C \iff \sqrt[A]{\frac{1}{(1-\alpha(x))^{A\cdot\alpha'(x)}}} = \frac{1}{(1-\alpha(x))^{\alpha'(x)}} = \sqrt[A]{C} \tag{5.9}$$

Thus, the result is independent of A. With the boundary condition $\alpha(0) = 0$, Mathematica finds the closed form solution for the inverse function:

$$\alpha^{-1}(x) = x + (1-x)\cdot\log(1-x) \tag{5.10}$$

The solution satisfies $\alpha(1) = 1$. We show this function in Figure 5.1. Interestingly, the graph shows that this function is almost identical with the numerical solution of the construction time optimization. We discuss application of this function as a closed form approximation of the numerical solution in Section 5.3.5.1.

5.3.4. Bucket assignment function

Until now we have only considered the solution in terms of $\alpha()$. However, our original goal was to find a function $bucket : [0,1) \to [0,1)$ that provides us with the normalized bucket index for a given normalized hash. This is the function which is actually required by an implementation. We now have to find a bucket assignment function such that the resulting bucket sizes resemble the desired bucket sizes $A\alpha'(x)$. To accomplish this we make the approximation that $A\alpha'(x)$ does not only describe the actual bucket size for bucket index x *after* sorting, but also the *expected* bucket size *before* sorting. Intuitively, the sorting step "compensates" for the fact that the actual bucket sizes are Poisson distributed in their expected size. The approximation error is smaller for larger average bucket sizes as a consequence of the law of large numbers.

The probability that a key ends up in a bucket is proportional to the bucket's expected size. We can therefore interpret $\alpha'()$ as a probability density function. Then $\alpha()$ is its probability mass function. The input of $bucket()$ is uniformly distributed. According to the inverse transform sampling method [34] we can use

$$bucket(x) := \alpha^{-1}(x)$$

for our bucket assignment function to achieve the targeted distribution of bucket sizes. The bucket assignment function of the closed form solution 5.10 is therefore

$$bucket(x) = x + (1 - x) \cdot \log(1 - x) \tag{5.11}$$

5.3.5. Quality and Relevance of the Solution

To derive the bucket assignment function 5.11 we made several approximations. The first approximation is that B and N approach infinity with a constant average bucket size $A = N/B$. The second approximation is that $\alpha(x) \in C^{\infty}$, specifically allowing non-natural bucket sizes. We then assume that the distribution of bucket sizes after sorting is equal to the distribution of expected bucket sizes before sorting. Finally, we obtained a closed form approximation for the numerical solution. We conducted a simulation to show both the quality of our approximations and the relevance of our solution.

5.3.5.1. Quality

The simulation proceeds according to the generic bucket based construction framework. We first distribute the keys to buckets according to the closed form bucket assignment function shown in Equation 5.11. The buckets are then ordered by non-increasing size. We calculate for each bucket the expected number of search iterations until a pilot is found using Equation 5.2. A search iteration corresponds to calculating and checking the position of one key for one pilot. We show simulation results in Figure 5.2 for one million keys and an average bucket size of 11. For comparison, we also provide the "target" which is how the simulation results would look like if all our approximations were exact. The target can be obtained by applying the numerical solution of Section 5.3.1 to Equation 5.2. Although it looks like the target is a constant function, it is monotonically decreasing and about 50% lower on the right compared to the left. This is not visible because of the logarithmic scale.

Our approximations are more accurate if the curve obtained by the simulation is closer to the target curve. The most obvious difference between the curves is probably that our simulation results in a sawtooth function, while the numerical solution is smooth. This is because bucket sizes are natural numbers. Whenever the bucket size decreases by one, the curve jumps down because the bucket is much easier to insert. After each jump, the curve increases steadily because the number of free positions decreases with every insertion. The next difference is towards the end where the simulated curve drops significantly below the target. We hypothesize that this is because of the assumption that the sorting compensates for the fact that the sizes are Poisson distributed. We further hypothesize that the approximation only works well for the larger bucket sizes as a result of the law of large numbers. At the very end of the curve there is a peak. This is unavoidable because

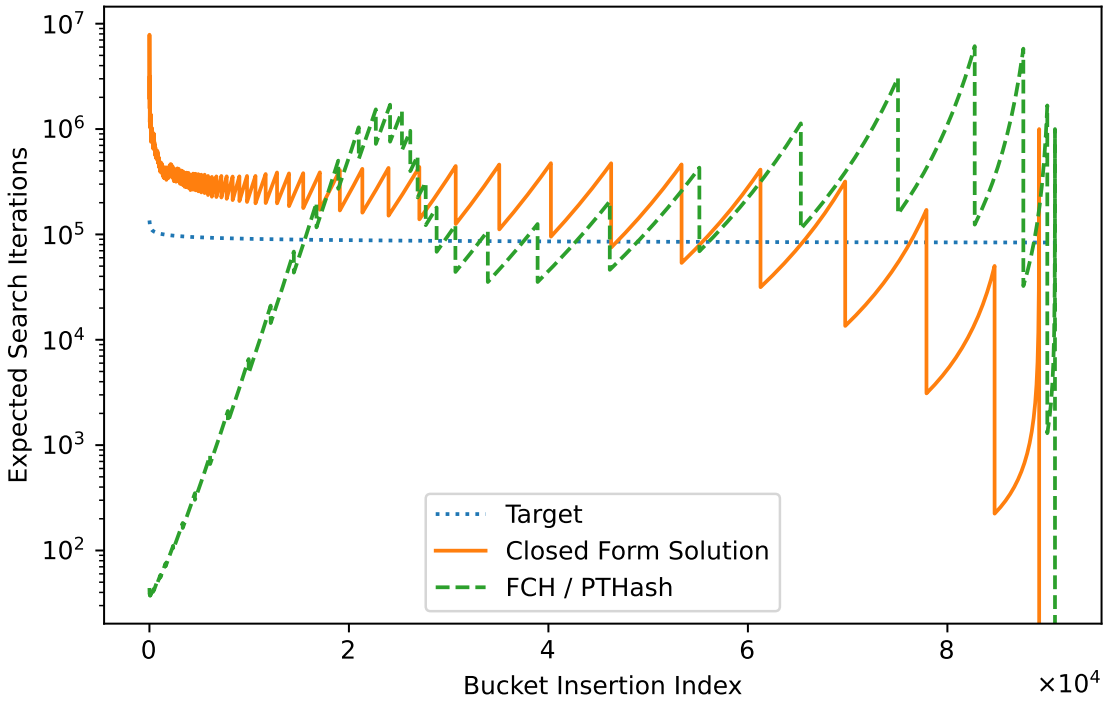

Figure 5.2.: Expected search iterations for all buckets calculated using Equation 5.2 for FCH and our closed form solution based on a simulation with $A = 11$ and $N = 10^6$. The target is the expected search iterations for the optimal distribution based on our assumptions.

the last bucket only has one free position in a minimal PHF. After that, all other buckets have a size of zero (not visible because of the logarithmic scale). Finally, there is another peak at the very beginning of the simulated curve. This is because we approximated the numerical solution using the closed form solution 5.10. The specific reason can be seen in the left plot of Figure 5.1. The bucket size of the closed form approximation is slightly higher around $x \approx 0$ compared to the numerical solution. Recall that the relative bucket size $\alpha'(x)$ is the derivative of the inverse assignment function. It is therefore possible to correct the closed form solution 5.10 by adding a linear correction term like

$$bucket(x) = (x + (1 - x) \cdot \log(1 - x)) \cdot (1 - c) + x \cdot c \tag{5.12}$$

We determined numerically that the linear coefficient $c = 0.024$ aligns the closed form solution to the numerical solution for $x \approx 0$. This correction solves the problem of the peak at the beginning and results in an approximately constant function for $x \approx 0$. The correction decreases the total number of search iterations by 10%.

We remark that we also experimented with other average bucket and input sizes. There were no qualitative differences. To summarize, the approximations of the model result in some deviations compared to the simulated results. Future work should further improve the model to bring simulated and theoretical results closer together.

5.3.5.2. Relevance

To show the relevance of our optimization we also applied the simulation on the state-of-the-art bucket assignment function of FCH and PTHash. The issue discussed in PTHash regarding the large number of search iterations towards the last few insertions is quite apparent in Figure 5.2. For an average bucket size of $A = 11$ our optimized distribution results in 2.3 times fewer total search iterations compared to the FCH function. Admittedly, the difference is larger for higher average bucket sizes. For an average bucket size of 7, which is quite small in practice, our optimized distribution is just 1.03 times better. However, there are not only differences in construction time but also in the space consumption between the optimized and the FCH assignment functions. Again, the differences are greater for larger average bucket sizes. We show this in the evaluation in Figure 7.3. The correlation between the distribution of bucket sizes and space consumption should be researched in future work.

To summarize, optimized bucket assignment functions are of particular relevance for large average bucket sizes.

5.3.6. Recommendation for Future Work

We recommend future work to use the bucket assignment function 5.12. There is no practical advantage in using the numerical solution. Our initial experiments suggest that a significant speedup can be achieved when substituting this function into the PTHash [3] implementation. In this thesis we use partitioning. We therefore have to use the numerical solution which takes into account local collisions.

5.4. The Average Bucket Size Tradeoff

The average bucket size is the central parameter of the generic bucket based construction algorithm. In the following we find that it introduces a tradeoff between entropy of the MPHF and the expected number of search iterations. Applying the constant pilot model greatly simplifies this analysis. By definition of the model in Section 5.3.3, the following expression is a constant for all $0 \leq x < 1$

$$C := (1 - \alpha(x))^{\alpha'(x)} = \frac{1}{e} \approx 0.368 \tag{5.13}$$

We evaluated the constant numerically using Mathematica.

5.4.1. Pilot

According to Equation 5.9 the probability that a pilot is viable is $p := (1-\alpha(x))^{A\alpha'(x)} = C^A$ for all buckets. Because the probability is a constant in this model, all pilots follow the same geometric distribution. We can therefore calculate the entropy of a bucket using the known formula for the entropy of a geometric distribution and divide by A to obtain the entropy in bits per key as

$$\frac{-p\log_2(p) - (1-p)\log_2(1-p)}{Ap} = \log_2(e) - \frac{(1-p)\log_2(1-p)}{Ap} \tag{5.14}$$

Equation 5.14 is not intuitive. Asymptotically equal and much more informative is

$$(5.14) \sim \log_2(e)\left(1 + \frac{1}{A}\right) \tag{5.15}$$

Therefore, the space consumption is asymptotically reciprocal to the average bucket size with an offset equal to the theoretical lower bound for an MPHF of 1.44 bits per key.

5.4.2. Search Iterations

We now derive the expected number of search iterations per key. We first state the resulting equation and explain afterwards.

$$\frac{1}{A}\int_0^1 \frac{1-(1-\alpha(x))^{A\cdot\alpha'(x)}}{\alpha(x)\cdot(1-\alpha(x))^{A\cdot\alpha'(x)}}dx \tag{5.16}$$

$$= \frac{1}{A}\int_0^1 \frac{1-C^A}{\alpha(x)\cdot C^A}dx \tag{5.17}$$

$$= \frac{1}{A}\left(\frac{1}{C^A} - 1\right)\int_0^1 \frac{1}{\alpha(x)}dx \tag{5.18}$$

$$= \frac{e^A - 1}{A}\frac{\pi^2}{6} \tag{5.19}$$

We start with Equation 5.6 for the expected number of search iterations of one bucket. We integrate over all buckets to obtain the average expected number of search iterations per bucket. The formula is divided by A to obtain the expected number of search iterations per key in 5.16. We then substitute 5.13 to obtain 5.17, simplify to 5.18 and solve the remaining integral using Mathematica to obtain 5.19. Asymptotically, the expected number of search iterations is in $O(e^A)$.

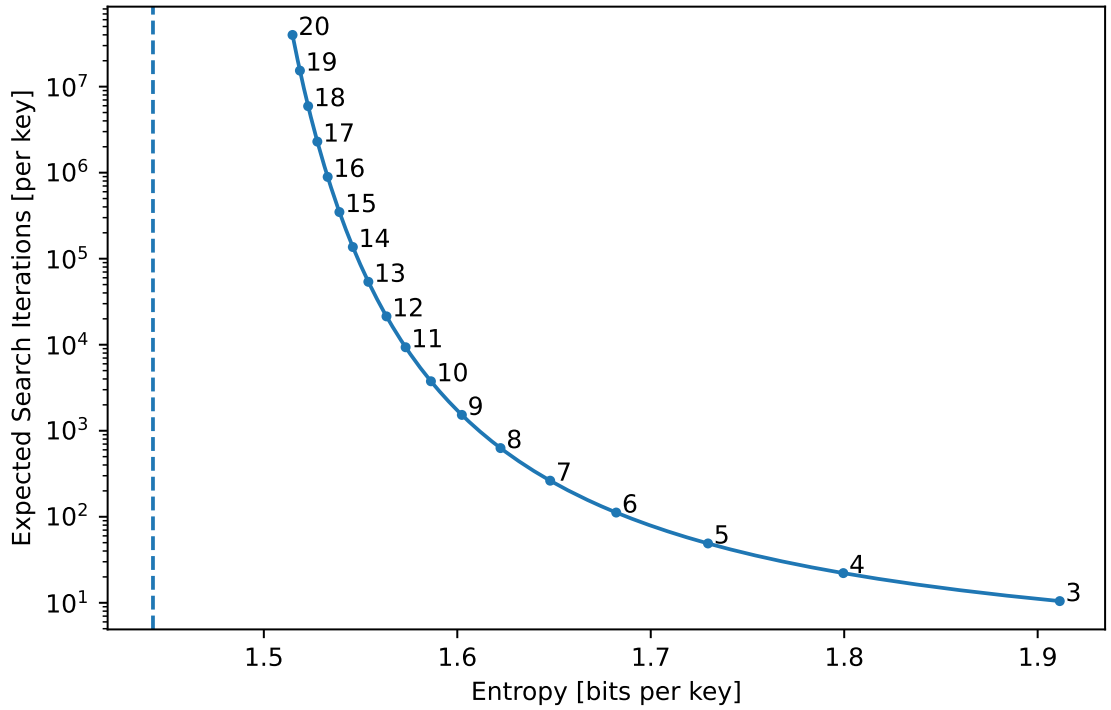

Figure 5.3.: Tradeoffs between entropy and search iterations based on the constant pilot model. The dashed line is the theoretical lower bound for MPHFs. The numbers on the curve are the average bucket sizes.

5.4.3. Tradeoff

Applying the formula for entropy 5.14 and for the expected number of search iterations 5.19 for different average bucket sizes reveals the tradeoffs that can be achieved. We show tradeoffs for a range of average bucket sizes in Figure 5.3. Space consumption converges towards the theoretical lower bound of 1.44 bits per key when linearly increasing the average bucket size. However, at the same time, the number of search iterations grows exponentially. In our implementation, it will turn out to be impractical to go beyond an average bucket size of 12.

6. Implementation

This chapter is about the GPU implementation. Note that if the workgroup size of a shader is not mentioned explicitly, it will be optimized experimentally in the evaluation chapter.

6.1. High-Level Parallelization

Partitioning separates the key set into P independent subsets of expected equal size. The expected partition size N/P is a hyperparameter. An MPHF of each subset can be constructed concurrently. The prefix sum of the partition sizes has to be calculated and stored. The prefix sum represents the partition *offsets*. The query algorithm has to be adjusted for partitioning. When querying a key, we first determine its partition $p \in [P]$. We then perform the query algorithm of partition p and add the offset of partition p to obtain the result.

Advantages. The partitioning approach allows concurrent construction and discloses a clear mapping to the GPU hardware: each workgroup constructs an MPHF of one partition. There are several other advantages of the partitioning approach: (I) It simplifies finding hyperparameters which are dependent on the input size. This dependence is negligible for large partition sizes, given that the coefficient of variation of a Poisson distribution is $(N/P)^{-1/2}$. (II) Any PHF construction algorithm which is polynomially bounded will be linear in expectation across the entire input [1]. (III) Partitioning introduces locality. The MPHF of each partition only needs a constant amount of memory to be constructed. (IV) Encoding specific advantages which are discussed later.

Disadvantages. A disadvantage of partitioning is that the offset and size of the respective partition need to be accessed additionally in the query algorithm. Another downside is that the probability of local collisions is increased.

General Data and Control Flow. The implementation begins by allocating all GPU buffers that will be required by the following implementation. The buffers are "device local", i.e. they reside on the GPU global memory and can be accessed by the GPU without communication to the CPU. After creation of the buffers, the 128-bit input keys are transferred from CPU RAM to the device local key buffer of the GPU. The entire GPU algorithm is then added to a single command buffer and submitted. While the GPU is working, the CPU waits for the GPU to finish. The results (i.e. the pilots and the partition offsets) are then transferred from GPU to CPU. The results are encoded in parallel on the CPU.

Bucketing Strategy. The user defines the average bucket size N/B. We now have two reasonable options to choose the number of buckets of a partition. Either we choose the number of buckets based on the *actual* size of each partition independently, or on the constant *expected* size of a partition. The former approach guarantees that the required average bucket size is met (within rounding error). In terms of construction time the former approach requires us to first determine the partition sizes, such that we can calculate the bucket count of each partition. Only then we can even begin to determine the bucket sizes. In the latter approach the bucket sizes can be determined first and the partition size is calculated as the sum of the bucket sizes. This makes construction of the latter approach faster. In terms of query, the latter approach allows fetching the pilot of a bucket independently of partition specific data. The former approach would add a data dependency: Before we even know the bucket of a key we first have to determine the bucket count of the partition. This is clearly not beneficial for query time. There are also some encoding specific advantages when using the same number of buckets in each partition. Furthermore, the discrepancy between the targeted average bucket size and the actual bucket sizes gets smaller in the latter approach with increasing partition size. Thus, we decided to use the latter.

6.2. Partition and Bucket Assignment

The partition and bucket assignment function rely on the following concept: Given a value v which is uniformly distributed in $[0, 1)$, some integer value z and their product $p = v \cdot z$. The integer part $c = \lfloor p \rfloor$ is uniformly distributed in $[z]$ and the remainder $r = p - c$ is uniformly distributed in $[0, 1)$. Equally important is the fact that c and r are statistically independent.

6.2.1. Hash Value Usage

The 128 bit hash value h is split in two parts of 64 bit. The first part h_{high} is used for partition and bucket assignment. The second part h_{low} is used in the searching phase. On the one hand, this separation allows several optimizations. On the other hand, the separation increases the risk of a hash collision: If two keys with same h_{low} end up in the same bucket of the same partition, we cannot find a pilot for that bucket, because any pilot will map equal keys to equal positions. To estimate the probability of a hash collision we first assume that all buckets have equal size N/B. This approximation should be good enough to get a grasp of the magnitude of the issue. The probability of a collision inside a single bucket is $p_1 = \frac{2^{64}!}{2^{64 \cdot N/B}(2^{64}-N/B)!}$ according to the birthday paradox formula. Because there are B buckets, the total probability of a collision is $p_2 = 1 - (1 - s)^B$. If we were to fully use the 128 bit, the collision probability would be $p_3 = \frac{2^{128}!}{2^{128 \cdot N}(2^{128}-N)!}$. We show two examples with $N/B = 8$. In the first example, $N = 10^6$, then $p_2 \approx 10^{-13}$ and $p_3 \approx 10^{-27}$. In the second example, $N = 10^9$, then $p_2 \approx 10^{-10}$ and $p_3 \approx 10^{-21}$. The relative difference between p_2 and p_3 gets smaller for increasing N. Intuitively, because there are more buckets in total and thus, more information of h_{high} can be utilized. Although hash

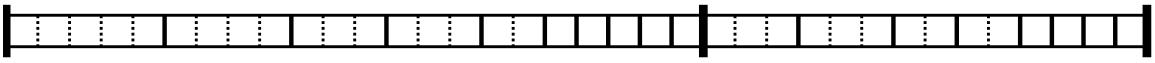

Figure 6.1.: The targeted arrangement of keys: Inside the two partitions (big lines), the buckets (bold lines) are sorted in non-increasing size. The cells represent the keys belonging to a bucket.

collisions are unlikely, we show options on how one could improve p_2 in future work: (I) If there is a h_{low} collision, use h_{high} for searching additionally, then $p_2 = p_3$. (II) Rehash all keys of a partition if it has a h_{low} collision. (III) Use more than 128 bit hash values.

6.2.2. Partition Assignment

We again split h_{high} in two parts of 32 bits, namely $h_{partition}$ and h_{bucket}. The partition is the integer part of $h_{partition} \cdot P$, where $h_{partition}$ is interpreted in $[0, 1)$.

6.2.3. Bucket Assignment

A bucket assignment function $bucket(h) : [0, 1) \rightarrow [0, 1)$ defines which values of h are assigned to bucket $\lfloor B \cdot bucket(h) \rfloor$. In Chapter 5 we showed how $bucket(h)$ can be derived for the brute force search algorithm.

We want to be able to experiment with arbitrary bucket assignment functions without manually implementing the function both in C++ and GLSL. At the same time, the function should be fast to evaluate and bit exact across platforms. We do so by sampling the bucket assignment function in equally sized intervals on the C++ side, essentially building a lookup table. The sample points are then used to evaluate $bucket(h)$ on both the GPU and CPU side. The result of the sampling is an array of S sampled values $d[]$. To evaluate $bucket(h)$ we apply $h_{bucket} \in [0, 1)$ from the previous partitioning step. We interpolate linearly between the sample point $s = \lfloor h_{bucket} \cdot S \rfloor$ and $s + 1$. The interpolation value $i = h_{bucket} - s \in [0, 1)$ is applied. Again i is independent of s and uniformly distributed. To summarize, the bucket is $d[s] \cdot i + d[s + 1] \cdot (1 - i)$.

6.3. Key Rearrangement

The keys in global GPU memory are in arbitrary order. In the next preprocessing steps, the keys are rearranged by applying the partition and bucket assignment functions. The goal is to rearrange them such that the search step can access them efficiently. More precisely, the data is laid out such that keys belonging to the same bucket are consecutive and buckets of the same partition are consecutive in non-increasing size. The arrangement is illustrated by Figure 6.1.

The following introduces a five step procedure to reach the targeted rearrangement. An overview of the steps is provided by Figure 6.2. It illustrates the general data and control flow of the preprocessing steps to put the following implementation details into perspective.

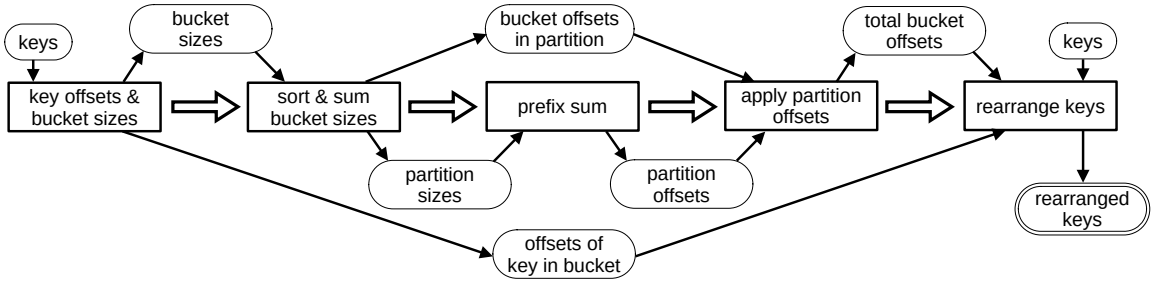

Figure 6.2.: An overview of the preprocessing steps: Shaders in rectangles, executed in the order indicated by the large arrows. Rounded rectangles are buffers. Data dependencies are indicated by small arrows.

6.3.1. Key Offsets and Bucket Sizes

In the first step, a shader reads the keys. It then applies the bucket and partition distribution function. A counter of the respective bucket b is incremented for each key. We do so using the $atomicAdd(bucketSizes[b], 1)$ function. The value previously stored at the counter position is returned. We use this value as an offset of the key relative to the bucket start position. We store the key offset in a buffer for the later rearranging step. To increase efficiency, each workgroup loads a total of 128 consecutive keys. By reasonably assuming that no bucket will contain more than 256 keys, we can encode the key offset in a single byte. Concatenation of the offsets results in only a single 128 byte write back to a buffer. Thus, fully utilizing the memory bandwidth when storing the offsets. Unfortunately, this optimization is just a drop in the ocean, considering that the *atomicAdd* instruction will cause a memory transaction for each key.

6.3.2. Sort and Sum Bucket Sizes

In the second shader, each workgroup reads the bucket sizes of one partition and then performs counting sort. The following is a detailed description of how the shader works.

1. The number of buckets for each of the 256 possible bucket sizes is counted in parallel. This is basically a histogram h of the bucket sizes. During construction of the histogram we obtain $o_i = h[size(B_i)]{+}{+}$ for the individual buckets B_i which are required later. Note that the order in which the buckets are processed is crucial to reduce the final entropy. We evaluate different orders in Section 7.2.3. A histogram of the sizes is actually sufficient to know where each bucket ends and begins inside each partition as shown in Figure 6.2. Accordingly, the histogram is written to a buffer to allow access to the individual buckets in the search step (not shown in Figure 6.2).

2. Similar to counting sort, we now perform local parallel prefix sum (see Section 2.3.2.1) of the histogram h to obtain h_+. The sort permutation is now calculated. Each bucket B_i is sorted to position $o_i + h_+[size(B_i)]$. This permutation is required at the end of the searching step to write back the pilots to the original bucket position.

3. To find the offset of each bucket inside a partition we proceed as follows. We multiply the histogram $h'[s] = h[s] \cdot s$ by the respective sizes s. By calculating a local prefix sum again we obtain h'_+, which are the positions for the first bucket of each bucket size. Thus, the offset of a bucket with size s_i relative to the start of the partition is: $o_i \cdot s_i + h'_+[s_i]$. We write this result to a buffer.

4. The previously performed prefix sum also yields the sum of all bucket sizes which is the partition size. We also store it in a buffer.

To summarize, the bucket size histogram, bucket permutation, bucket offsets relative to partition and size is determined for all partitions.

6.3.3. Final Steps

The third step is applying the global prefix sum algorithm (Section 2.3.2.2) on the partition sizes to obtain partition offsets. The fourth step adds the partition offsets to the relative bucket offsets to obtain the total bucket offsets. Again, one workgroup is responsible for one partition.

In the last shader, the keys are read again. We calculate the final position of each key as the sum of the respective total bucket offset and the offset of the key relative to the bucket (obtained in the first step). We write the key to that position and reach our goal. In practice, this step is the most expensive one, because each key will cause an individual memory transaction. However, our experiments suggest that the entire procedure is still faster than sorting the whole key set on the GPU. In the evaluation we show the execution times of each step.

6.4. Search

The search step is responsible for finding the pilot values of each bucket. Each workgroup performs the search for one partition. A total of P workgroups is invoked accordingly. We implemented the brute force, the pattern search and the randomized search algorithms. Before we explore the implementation of those algorithms separately, we examine some common functionalities.

6.4.1. Hash Function

The displacement hash functioned is implemented according to the definition in Section 4.2. Recall that the pilot value p is split in two parts $s \in \mathbb{N}_0$ and $d \in [M_i]$. M_i is the size of the respective partition. We encode s and d using pilot $p = s \cdot M_i + d$. Accordingly, $s = p \text{ div } M_i$ and $d = p \bmod M_i$. To obtain the initial position for key k and seed s we use $pos(k, p) = h(k, s) \bmod M_i$. Where $h(k, s) = remix(k_1 \oplus remix(k_2 \oplus s))$ uses the murmur3 remix function [35]. GPUs are usually 32 bit machines. This is why we split the key k in two parts k_1 and k_2. The 32 bit Murmur3 hash function can be evaluated using only a few 32 bit multiplications, bit shifts and XOR operations. During search, we add the displacement d to the initial positions and calculate the position using modulo M_i. We replace the

computationally expensive modulo operation with a conditional subtraction of M_i because $d < M_i$. In the query algorithm we encounter the situation where we have to find the position for key k with pilot p directly, which is $pos(k, p) = ((h(k, s) \bmod M_i) + d) \bmod M_i$. Substitution delivers $((h(k, p \operatorname{div} M_i) \bmod M_i) + (p \bmod M_i)) \bmod M_i$. We can simplify this to

$$pos(k, p) = (h(k, p \operatorname{div} M_i) + p) \bmod M_i$$

if we can guarantee that $h(k, s) + p$ will never overflow. GPUs are usually 32 bit machines. By shifting the result of h one bit to the right, we effectively limit the output to values smaller than 2^{31}. We reasonably assume that there will never be a pilot value p greater than 2^{31}. Thus, the sum will not overflow and we apply the simplification. Even if the CPU is a 64 bit machine, we have to perform the same shift operation during querying, to obtain the same result. Limiting the hash value in this manner has no practical relevance in terms of the randomness of the initial position. We argue that the initial positions are approximately uniformly distributed as long as M_i is several magnitudes smaller than 2^{31}. Usual values of M_i are around 2^{11}. Another potential for optimization is that the above formula involves a division and modulo operation using M_i as divisor. Both of those instructions can be optimized to a multiplication by first calculating and then applying the inverse of M_i. Consequently reducing the total number of integer divisions from two to one.

6.4.2. Free Array

The free array is a bit array $free[]$ made up of 32 bit integers and is stored in shared memory. We use $free[p/32]\&(1 << p\%32)$ to check if a position p is free. The division and modulo operations are optimized by the compiler to bit shifts and masks respectively.

6.4.3. Local Collisions

Before we can check for local collisions we have to evaluate the initial positions $h(k, s) \bmod M_i$ for all keys the bucket. The keys of that bucket are loaded from global memory. The consequence of the previous key rearrangement is that we can access the keys in the required order. Because of this locality, the caches are utilized, decreasing the number of global memory reads. We now look at two algorithms to identify local collisions.

Bit Array. Using a second bit array similar to $free$ we can check whether a position is already used by the bucket. We set the bits of the initial positions as being occupied. This step is done in parallel by applying the standard parallelization technique (see Algorithm 1) on the keys. Because of the parallelization we have to use *atomicOr* operations for insertions. The atomic function returns the previous content. If the previous content indicates that the respective position is already taken, then there is a local collision. Each time before checking, we have to reset the array.

Comparisons. We perform comparisons of a bucket with size b for all $b(b-1)/2$ pairs of initial positions. By applying Algorithm 2 we can generate one such pair based on a

loop index in closed form (there is no dependence on previous iterations). This allows for trivial parallelization of the loop across the threads.

Scaling and Combinations. The bit array approach scales in $\theta(b + M_i)$. It is linear in b, because of the insertions. The cleanup work of the array causes the algorithm to be linear M_i too. The approach of performing all comparisons is clearly in $\theta(b^2)$ and independent of M_i. Thus, the comparisons algorithm can be faster for small b. Consequently, we introduce a hyperparameter L. If the bucket size is smaller than L we perform comparisons, otherwise the bit array is used. L is optimized experimentally in the evaluation (Section 7.3.1).

6.4.4. Input and Output Mechanisms

All of the following search algorithms begin by reading the partition size M_i and the partition offset. The partition offset is the position of the first key in the rearranged key buffer. We now want to extract the buckets from the key buffer. This can be done by fetching the bucket size histogram of the partitions. The histogram allows us to identify the beginning and ending of a bucket inside the key buffer. The histogram was generated earlier in the bucket sorting shader. The sorting shader also generated a bucket permutation. At the end of the search shader, we apply the permutation to write back all pilot values to their position before the sorting step.

6.4.5. Brute Force Search

We discuss the GPU implementations of the brute force search algorithm. Local collision detection on the GPU has already been discussed. We now show the implementation for the search of the displacement value d, given the initial positions and the free array. Algorithm 4 uses a nested loop for searching a viable displacement. The outer loop iterates over displacement values until one is found. The inner loop iterates over all the positions and checks if they are free. If a position is occupied, the inner loop exits early and continues with testing the next displacement. Recall that the streaming processors of a GPU work in lock-step (Section 2.1). This constraint represents a challenge for a parallel implementation of the nested loop if we want to utilize the option of an early exit. We now show three different approaches on how displacement search can be implemented on a lock-step architecture. An illustration of the approaches can be found in Figure 6.3. The following algorithms use the function FREE(d, k) to check whether the position for key index k using displacement d is free.

6.4.5.1. Horizontal

The idea of the first algorithm is to "flatten" the nested loop. We do not utilize the early exit condition. From an asymptotic perspective, removing the early exit is not a good idea. However, the idea of "flattening" and removing the early exit allows us to calculate the displacement value d and the index of the inner loop k based on a single loop index i. We use $d = i/b$ and $k = i\%b$, where b is the size of the bucket. The standard parallelization technique (Algorithm 1) can then be applied to the loop index i. Note that the division

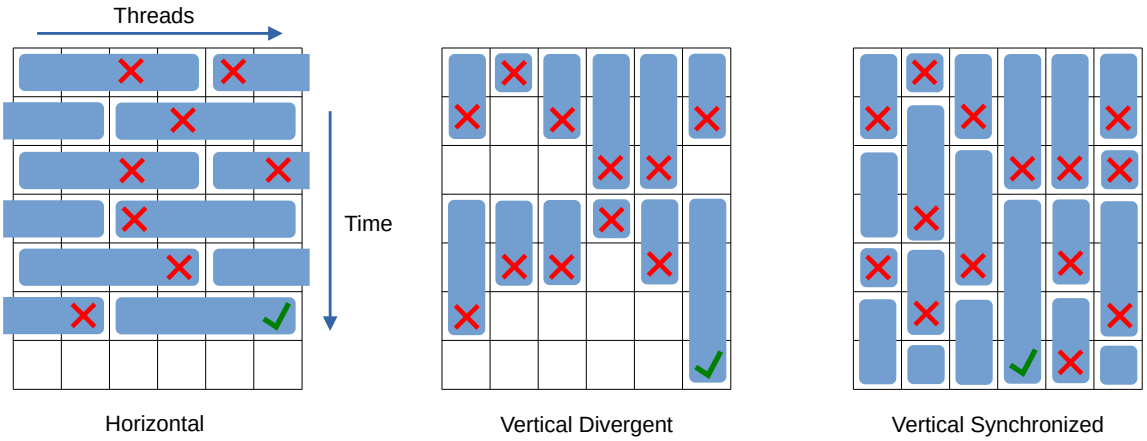

Figure 6.3.: Each column represents one thread. Each row represents one lock-step execution of applying the displacement to an initial position and checking if the position is free. Accordingly, each cell represents one key. In this example, the bucket size is 4. The blue boxes represent one displacement value that is tried. The red cross means that a collision was detected. The green tick means that the last key did not collide, meaning that a displacement was found.

and modulo instructions are manually optimized because the increment of the loop index is the workgroup size, which is a constant. If a thread determines that a position was occupied, it marks the displacement value as not viable. After a workgroup barrier, the last thread working on a bucket ($k = b - 1$) checks if the displacement value is still viable. If that is the case, then a displacement value is found. However, multiple threads might find a viable displacement in the same iteration. To reduce entropy, the result should be the smallest viable displacement. Thus, we use the minimum of the displacements using an atomicMin instruction on shared memory. Atomic operations tend to be faster on newer architectures inside one group compared to a reduction in shared memory. A pseudo code description can be found in Algorithm 5.

6.4.5.2. Vertical Divergent

The second idea is a very direct translation of the single threaded Algorithm 4 to a lock-step architecture. We only parallelize the outer loop by applying the standard parallelization technique (Algorithm 1). The inner loop remains unchanged. The possibility of an early exit is used. However, because of the lockstep architecture, the loop only exits once all threads have completed the loop. This means that there is divergence between the threads and the advantage of the early exit is dampened. If a loop does not exit early, it has found a displacement and we again obtain the minimum of all displacements found in that iteration. A pseudo code description can be found in Algorithm 6.

6.4.5.3. Vertical Synchronized

This last idea of implementing a nested loop on a lock-step architecture has been described by Sanders [36] in an abstract manner. The state of the nested loop is a tuple (d, k). The

outer loop iterates the displacements d. The inner loop iterates k, which is the index of the key that is currently being tested. We then emulate the behavior of both loops in a single loop. The behavior of the inner loop is emulated by incrementing the key index in each iteration. If a position is occupied, we can emulate the early exit. The behavior of the inner loop in case of an early exit is emulated by setting $k = 0$ and the outer loop is emulated by setting a new displacement. Unfortunately, we can no longer obtain a new displacement in a closed form like in the previous two approaches. Each thread might require a different number of iterations before continuing with a new displacement. To ensure that the first displacement that is found is also the smallest one, we have to obtain the new displacement by atomically incrementing a shared variable *dCount*.

A displacement is viable if the last key of a bucket has no collision. We then perform the atomicMin instruction as usual because multiple viable displacements might be found in the same iteration by different threads. A barrier is used to ensure that all threads see that a viable displacement is found. A pseudo code description can be found in Algorithm 7.

Algorithm 5 Horizontal

```
shared dViable ← ∞                          ▷ The displacement value that has been found.
i ← threadId                                ▷ Local thread identifier is the start index.
while dViable == ∞ do                       ▷ Search until a displacement is found.
    d ← i/bucketSize                        ▷ The displacement this thread is testing.
    k ← i%bucketSize                        ▷ The key index this thread is testing.
    if ¬FREE(d, k) then                     ▷ A collision is detected.
        MARKCOLLISION(d)                    ▷ Mark this displacement as not viable.
    BARRIER()                               ▷ All threads can see if a collision occurred.
    if k == bucketSize − 1 and HASNOCOLLISION(d) then   ▷ There was no collision.
        ATOMICMIN(dViable, d)               ▷ The thread commits the result.
    i ← i + threadCount                     ▷ Increment by the number of threads in this workgroup.
    BARRIER()                               ▷ All threads can see if a displacement was found.
```

Algorithm 6 Vertical Divergent

```
shared dViable ← ∞
d ← threadId
while dViable == ∞ do
    viable ← True
    for all k in [bucketSize] do
        isFree ← FREE(d, k)
        if ¬isFree then
            viable ← False
            break
    if viable then
        ATOMICMIN(dViable, d)
    i ← d + threadCount
    BARRIER()
```

Algorithm 7 Vertical Synchronized

```
shared dViable ← ∞
shared dCount ← threadCount
d ← threadId
k ← 0
while dViable == ∞ do
    isFree ← FREE(d, k)
    k ← k + 1
    if ¬isFree then                         ▷ Early exit
        k ← 0                               ▷ Next displacement
        d ← ATOMICADD(dCount, 1)
    else if k == bucketSize then
        ATOMICMIN(dViable, d)
    BARRIER()
```

6.4.5.4. Discussion

Calculating and checking the new position of a specific key based on a specific displacement can be seen as independent tasks. We showed three approaches for laying out those tasks along the thread and time dimension. Figure 6.3 visualizes the layout of each approach. The horizontal approach has the advantage that a result can theoretically be obtained after just a single iteration. Vertical approaches require at least b iterations until a result is obtained. The vertically synchronized approach is asymptotically optimal. However, it requires a shared atomic displacement counter to guarantee that the smallest viable displacement value is among the first displacement values that are found. The vertical divergent approach does not require a shared counter but has significant divergence. Furthermore, the vertical divergent approach requires a barrier only once for each displacement. This is less than the synchronized approach, where a barrier is used after every key. The divergent approach also has another advantage which is part of the discussion in the following paragraph.

Memory Banks. In Section 4.2 we discussed some advantages of displacement hashing. We now look at a hardware and implementation dependent advantage. For analysis we use the following hardware assumptions: (I) A warp has 32 threads. (II) A Streaming Multiprocessor has 32 memory banks. (III) Each memory bank stores 32 bits. (IV) The scheduler ensures that there are no simultaneous memory accesses of different warps inside the Streaming Multiprocessors. We use a 32 bit integer array in shared memory to store which positions are free. In an obvious manner, the i-th integer stores positions $[i \cdot 32..(i+1) \cdot 32 - 1]$. The j-th memory bank stores the integers $[j, j+32, j+64, ..]$. A k-bank-conflict occurs if one memory bank is accessed for k different memory addresses. In that case, the accesses are serialized. This results in a decreased throughput by a factor equal to the highest k across all banks. However, querying a memory bank for the same address from multiple threads results in a broadcast which does not decrease throughput. We now consider the performance of the three approaches in terms of bank conflicts. We begin with the vertical divergent approach. The outer loop ensures that the threads check consecutive displacements at all times. The inner loop ensures that all threads check the position of the same key using different displacements. Thus, the 32 memory accesses are completely consecutive. Consequently, either one or two memory banks are accessed for the same address resulting in broadcasting. There are no bank conflicts. Generally speaking, if multiple threads check the same key for different displacements they can not cause a bank conflict to each other. The other two approaches are not optimal. The horizontal approach results in the most bank conflicts because there is a maximal number of different keys tested in every iteration. Somewhere in between those two extremes is the vertical synchronized approach. Because of the early exit, the threads generally process different keys in the same iteration. The first keys are more likely to be used in an iteration compared to the other keys. Consequently, the probability that a conflict occurs is lower compared to the horizontal approach, but higher compared to the synchronized approach. We remark that it is possible to implement the vertical synchronized approach without bank conflicts. However, it requires more instructions and performs slightly worse. For simplicity, we only evaluate the three approaches discussed here.

Hardware Specific Optimizations. We also consider removing the barriers. This is an optimization which is not guaranteed to work across different GPUs. It is not certain that the threads see the changes to $dViable$ after one thread has found a displacement and called the atomic-min function on $dViable$. In that case, the algorithm would run until all threads have found a displacement for themselves. This would be clearly disadvantageous. However, removing the barrier is an advantage if the threads are able to see the changes of $dViable$ anyway. We use this optimization whenever possible.

6.4.6. Indexed Search

In the following we show a GPU implementation for the pattern index approach. We are given the initial positions of the keys of one bucket. Those positions were previously checked for local collisions. Our goal is to determine if a viable displacement exists. If at least one exists, the minimal viable displacement of that bucket needs to be returned. In the following we first show how the pattern index can be constructed. The second part shows how to implement the pattern search on the GPU.

6.4.6.1. Pattern Index Construction

Recall that the pattern index p is a function $\Phi(d) = \{p \mid p \text{ is free} \wedge (p+d) \bmod N \text{ is free}\}$. We precompute all results of $\Phi(d)$ to allow fast access. The first step is to find all free positions and write them to a list. We can then determine $|\Phi(d)|$ for the entire domain by looking at any pair of free positions, calculating the difference d between the two positions and incrementing the respective counter. A prefix sum over all $|\Phi(d)|$ determines the offset for each d. Finally, the pairs can be inserted at the respective position using the previously obtained offset. Each pair $\{p_1, p_2\}$ is stored using a 32 bit int of p_1 (WLOG $p_1 + d \bmod M = p_2$). In future work one could try a more compact encoding of the index. E.g. one can enforce a monotonic order of all $\Phi(d)$ and then apply difference encoding or even Elias-Fano. This is particularly interesting on a GPU, given the low amount of shared memory per thread and the large size of the index.

6.4.6.2. Pattern Search

We are given the collision free initial positions for all keys of a bucket. Our goal is to find a viable displacement value. The index is applied to align the pattern to free positions. Recall that a heuristic is used to determine a walk in the pattern graph. The first step is to only look at a random subset of all edges. We then determine the one difference between initial positions d of the subset with lowest $|\Phi(d)|$. One pair of the initial positions with difference d is the starting edge. As explained in the working principle chapter, we traverse the graph by performing list intersections at each step. List intersections are performed by applying the standard parallelization (Algorithm 1) on the pair builder function (Algorithm 2). If the values of a pair are equal, it is appended to the output. The walk continues with the next edge. We use the edge with difference d such that $|\Phi(d)|$ is minimal.

6.4.6.3. Discussion

The space for the pattern index grows in $O(F^2)$ where F is the number of free positions. This would result in a very large index at the beginning of insertions, when all positions are free. Thus, we will consider the pattern index approach only in combination with the brute force approach. A parameter will define when to switch from brute force search to index search.

6.4.7. Randomized Search in Pilot Limited Space

The user defined a pilot limit for each bucket. The following implementation finds a pilot within that constraint. We first proceed as usual, processing the bucket in non-increasing size using the brute force algorithm. The randomized search begins only if a pilot was not found within the pilot limit. In that case, the bucket is added to a list L. The randomized search algorithm is now applied to all buckets in L until L is empty. Once L is empty, the brute force search continues.

The randomized search for a bucket works as follows. First, the brute force search is tried again for that bucket (with the exception of the first bucket added to L, because we just attempted the brute force approach on that bucket unsuccessfully). If the brute force search cannot find a pilot within the pilot limit, the actual randomized search starts. A random subset of pilot values is considered for the bucket. We now have to choose a pilot value which causes the least amount of reinsertion. The heuristic from Section 4.5 is applied to find the pilot. We then forcefully insert the bucket using that pilot. Any colliding bucket is added to L and the previously occupied positions of each bucket are freed. This procedure implies that we have to keep track which position is occupied by which bucket.

It is possible that all pilots for a bucket cause a local collision, rendering the bucket impossible to insert. In that case, the search algorithm is started from scratch for that partition and the restart counter of that partition is incremented. The counter introduces an offset to all pilots, ensuring that all pilots have not been tried in the previous iteration. Another reason to restart the search is if no pilot can be found after a certain amount of reinsertions or if L grows too large. Unfortunately, the existence of the counter implies that we have to store it for each partition and access it during querying to calculate the actual pilot of a bucket.

6.5. Encoding

Once the searching phase has finished we have to transfer the pilots to the GPU and encode them in a compressed manner. In the working principle we discussed several encoding techniques. PTHash [3] provides implementations for the compact, dictionary, Elias-Fano and Simple Dense encodings. We apply them in our implementation. We also added the Golomb-Rice encoding using functionalities provided by the PTHash implementation such as a constant time rank structure on bit vectors.

6.5.1. Monolithic

A *monolithic* encoding applies one encoding technique on all input values. This can be problematic if the pilots of the buckets follow different probability distributions. In the following we show this in an example using the compact encoding. There are 100 buckets, 99 of which have a low expected pilot value. Only the pilot of one bucket has a relatively high expected value. W.h.p. the one pilot is relatively large and will require a lot of bits to encode. Unfortunately, compact encoding means that all the other pilots are encoded using the same number of bits. This results in a huge waste of space. That issue exists for most encoding techniques. Consequently, it is desirable that all pilots handled by one encoder should follow the same statistical distribution.

6.5.2. Orthogonal

PTHash also observed this phenomenon and offers a solution. The pilots of all buckets which have equal expected bucket size are encoded using the same monolithic encoder. In the case of PTHash there are two different expected bucket sizes. Unfortunately, equal expected bucket size is not equivalent to equal statistical distribution of the pilot value, because the sorting step breaks the symmetry. In this thesis we consider bucket assignment functions (see Section 5.3) resulting in a different expected bucket size for every bucket of one partition. We assume that the variations of the number of keys in each partition is negligible. Thus, the n-th bucket of each partition has the same expected bucket size. We now define the *orthogonal* encoding. The orthogonal encoder is made up of as many monolithic encoders as there are buckets in each partition. The n-th monolithic encoder encodes the pilot values of the n-th bucket of all partitions.

An advantage of the orthogonal encoding is that all pilots in one encapsulated monolithic encoder not only share the same expected bucket size, but also the same statistical distribution. This is because all partitions and their buckets are symmetrical in expectation.

In PTHash, the authors experimented with using different encoding techniques for their two monolithic encoders. We adapt this to our orthogonal encoding. For example, some encoders use compact encoding and the others use Golomb encoding. However, it is convenient to use a faster encoding for the buckets of larger size, because they tend to be queried more often [3]. A more dense but slower encoding can be used for the smaller buckets. In Section 7.5, we experiment with combinations of encoding techniques.

Beside the space advantages of orthogonal encoding there are also some query time advantages. Larger buckets are even more likely to be cached, because they are stored consecutively.

Future work which does not employ partitioning can also adapt this encoding. Consecutive buckets only have marginal differences in the statistical distribution of their pilots. Therefore, the pilots could be grouped and a different encoder can be used on each group.

6.5.2.1. Analysis of Compact Orthogonal Encoding

Applying the orthogonal variant of the compact encoding allows us to derive a simple upper bound on the asymptotic space consumption. There are as many buckets in one monolithic encoder as there are partitions P. All of those buckets follow the same distribution. We look at the insertion of a single bucket. During insertion we have probability p that a random position is free and a given bucket size b. The pilot follows a geometric distribution (Equation 5.1). However, both b and p are distributions themselves. Thus, the actual distribution of the pilot is a linear combination of geometric distributions. We now consider the pilot value of all buckets in one encoder. The orthogonality ensures that all pilots follow the same distribution. The pilots are a sample of size P of this distribution. The maximum expected pilot of that sample for a single geometric distribution is in $\Theta(\log(P))$ [37]. This asymptotic result holds for any linear combination of geometric distributions as a direct consequence of the linearity of expectation values. We apply the well known inequality $\mathbb{E}[\log(X)] \leq \log(\mathbb{E}[X])$. Then, the expected number of bits required to store all P pilots of one monolithic compact encoder encapsulated in an orthogonal encoder is in $O(P \cdot \log_2(\log(P)))$.

To avoid this scaling issue, PTHash-HEM [24] introduced partitioned compact encoding. The values are first split into blocks of e.g. 256 values. The compact encoding is applied to each block independently. Consequently, the space consumption is in $O(P)$ like most of the other encodings.

6.5.2.2. Selecting Golomb-Rice parameter

We now want to find the optimal Golomb parameter τ for a monolithic Golomb encoder encapsulated in an orthogonal encoder. Again, all pilots of one encoder follow the same distribution: a linear combination of geometric distributions. Kiely [29] provides the following formula to choose the space optimal Golomb parameter, based on the success probability p of a *single* geometric distribution

$$\tau(p) = \max\left\{0, \left\lceil \log_2\left(-\frac{\log_2 \frac{\sqrt{5}+1}{2}}{\log_2(1-p)}\right)\right\rceil\right\} \tag{6.1}$$

We now simply assume that the pilots follow a single geometric distribution. Experimentation with manual selection of τ suggests that this assumption works very well in practice. p is calculated as the inverse of the average value of all pilots. Note that Golomb encoding scales in $O(P)$, because the space consumption of each pilot is independent.

6.5.3. Details

As a proof of concept, we provide a GPU parallel implementation for monolithic dictionary encoding. For convenience, we use the CPU implementation provided by PTHash [3] for all the other encodings. The main downside of CPU encoding is that the pilots are not compressed during transfer from GPU to CPU, resulting in a longer transfer duration. The orthogonal encoding can be trivially parallelized. Encoding for each of the encapsulated monolithic encoders can be performed concurrently in different CPU threads.

6.5.4. Partition Offset Encoding

During querying we require the size and the offset of a partition. The offset is stored and the size is calculated as the difference of consecutive partition offsets. The Elias-Fano encoding seems like a perfect choice in terms of space consumption, given the monotone increase of offset values. However, it is much more important to have fast access to partition data, because one partition will be queried much more often than one bucket. Unfortunately, querying Elias-Fano is quite slow. At the same time, partition data only makes up for a small portion of total space, so space consumption is not critical here. We therefore also try compact encoding as a faster alternative. To avoid excessive space consumption we only store the *difference* of the partition offset to its expected value. We compare both techniques experimentally in Section 7.5.3.

7. Evaluation

We experimentally evaluate the performance of our implementation. Performance is primarily measured in terms of construction time, query time and space requirements. The experimental setup and an initial configuration is described in Section 7.1. Different techniques are compared by respectively changing the initial configuration. We also show how the input size influences the performance of different techniques. Pareto fronts in Section 7.6 show which tradeoffs can be achieved between the three metrics using the configuration which is optimal in our setup. Finally, our implementation is compared to state-of-the-art MPHF construction techniques in Section 7.7.

7.1. Experimental Setup

Hardware. MPHF construction is performed on an Nvidia RTX 3090 GPU [15]. We have outlined some specifications of that GPU in Section 2.1. The MPHF is queried on an Intel Core i7-11700 CPU. One CPU core has 48 KiB L1 and 512 KiB L2 data cache. The CPU has a total of 16 MiB L3 cache. The machine has 64 GiB of dual-channel DDR4-3200 RAM.

Software. The C++ implementation is compiled using GCC 11.4.0 on Ubuntu 22.04.1. The compiler options `-march=native` and `-O3` are used. We use Vulkan 1.3.236 to interface with the GPU.

Metrics. We use `std::chrono::high_resolution_clock` to measure query and overall construction time on the CPU. We also use GPU timestamps provided by Vulkan for more detailed information on the execution time of individual shader invocations. To measure query times, a single thread performs the query on all inputs of the MPHF once. There is no dependence between input and output of the MPHF queries. The queries do not use manual software prefetching. We apply the same evaluation procedure on state-of-the-art techniques. The input keys are uniformly distributed 128 bit values.

Initial Configuration. We now define an initial configuration which is used throughout the evaluation. The input size is 100 million keys. We choose the hyperparameter $L = 14$ to check for local collisions (see Section 6.4.3). The vertical synchronized brute force search technique is applied. We use an average bucket size of 7 and a partition size of 2048. The bucket assignment function of Section 5.3.2 is applied. In the sorting step, buckets of equal size are sorted by *increasing* expected bucket size. We discuss sorting strategies in Section 7.2.3. Finally, the pilots are encoded using the orthogonal variant of the Golomb encoding (Section 6.5.2). Note that all shaders, if not stated otherwise in the implementation chapter, use a workgroup size of 32 threads. This is exactly the size of

A	Bucket Sizes	Sort	Partition PPS	Apply Offsets	Rearrange
4	290 ps	45 ps	<1 ps	3 ps	340 ps
7	274 ps	21 ps	<1 ps	2 ps	337 ps
10	256 ps	13 ps	<1 ps	1 ps	335 ps

Table 7.1.: Execution time for the key rearrangement steps measured in ps (picosecond) per key for different average bucket sizes A.

one warp. Experimentation with larger workgroup sizes consistently resulted in worse execution time. In this initial configuration, the MPHF is constructed in 10 ns per key, requires 1.83 bits per key and has a query time of 46 ns per key. In the following we show how changes of this initial configuration affect the performance in terms of the three metrics.

7.2. Preparation

In preparation of the searching step, we allocate the required buffers, transfer the 128 bit keys and rearrange them. In the following we show how each step influences the total construction time.

7.2.1. Key Allocation and Input Transfer

Allocation of the buffers and transfer of the input keys make up for a significant portion of the total construction time. Allocation of all buffers requires 0.44 ns per key. This is mostly independent of the configuration, because the constant space of the input keys dominate the total memory consumption. If possible, one should try to reuse those buffers when constructing multiple MPHFs to avoid the allocation overhead.

Transferring the input data from RAM to GPU requires 5.8 ns per key. This corresponds to a throughput of 2.75 GB/s, which is typical for CPU to GPU transfers. The key transfer makes up about half of the total construction time in the initial configuration. This should be considered when applying our implementation. If possible, keys should be generated directly on the GPU, avoiding the massive transfer overheads.

Our evaluation results will always include the allocation and transfer overheads.

7.2.2. Key Rearrangement

The keys are rearranged using the implementation described in Section 6.3 to allow efficient access during the searching phase. The performance of the individual rearranging steps is measured using GPU timestamps. Table 7.1 shows experimental results for different average bucket sizes. Preprocessing is cheaper for larger average bucket sizes because there are less buckets to process. Most of the preprocessing time is spent to determine the bucket sizes and to rearrange the keys. This is because the required memory accesses are not coalesced. Computing the partition offsets using a parallel prefix sum (PPS) and

applying the partition offsets to the bucket offsets is negligible. All steps still require less than 1 ns per key in total.

7.2.3. Sorting

Part of the previous rearrangement is sorting the buckets by non-increasing order of bucket size. This is done to reduce the construction time. However we observed that the order of buckets of equal size greatly affects space consumption. In the following we compare three different sorting techniques in terms of the total space consumption based on the initial configuration: (I) a pseudo-random permutation for buckets of equal size results in 1.873 bits per key, (II) sorting buckets of equal size by their expected bucket size in decreasing order results in 1.882 bits per key and (III) in 1.829 bits per key when sorting buckets of equal size in increasing order of expected size. We use (III) in the initial configuration for reasons that are now obvious. However, further research is required to either prove that this is the space optimal order or to find an even better order. We remark that the improvements of the sorting step do not require orthogonal encoding or partitioning. We also observed the same differences in space consumption when using compact instead of Golomb encoding.

7.3. Search

In the following we compare different techniques concerning the searching step.

7.3.1. Local Collision Detection

Section 6.4.3 introduces two techniques to detect local collisions. One can either use a bit array or compare all positions. Comparing the positions in parallel requires a closed form pair generator. Section 2.3.3 introduces two implementations of a pair generator. One approach is using a precomputed lookup table of pairs. The other one calculates a pair using Algorithm 2. In Section 6.4.3 we show that the bit array approach scales in $\theta(b + M_i)$ and the comparison approach in $\theta(b^2)$, where b is the bucket size and M_i the partition size. The idea is to use the bit array for large buckets and the comparison technique for small buckets. We introduce the parameter $L \in \mathbb{N}$ accordingly. If the bucket size is smaller than L we apply comparisons. The bit array is used otherwise. Clearly, no local collision detection has to be performed for a bucket size of one. Figure 7.1 shows how L influences the total construction time. Because of our experimental results we use the precomputed comparison technique with $L = 14$ in the initial configuration. This is a marginal optimization and only apparent when drastically scaling up the relative time spent checking for local collisions.

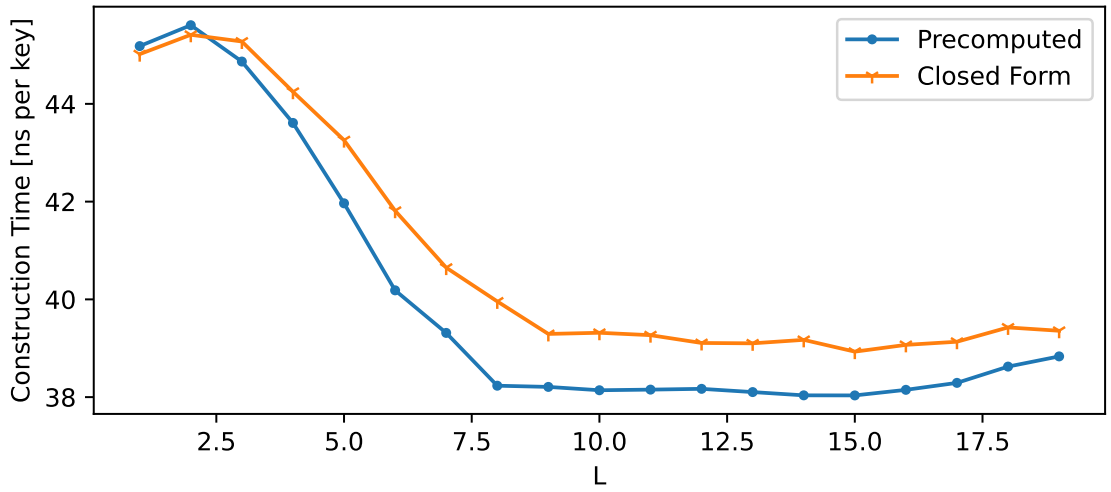

Figure 7.1.: Total construction time for different values of L. Checking for local collisions makes up only for a small portion of the total construction time. For this plot only, we artificially increased that time by redundantly checking for local collisions 200 times to make the differences more prominent.

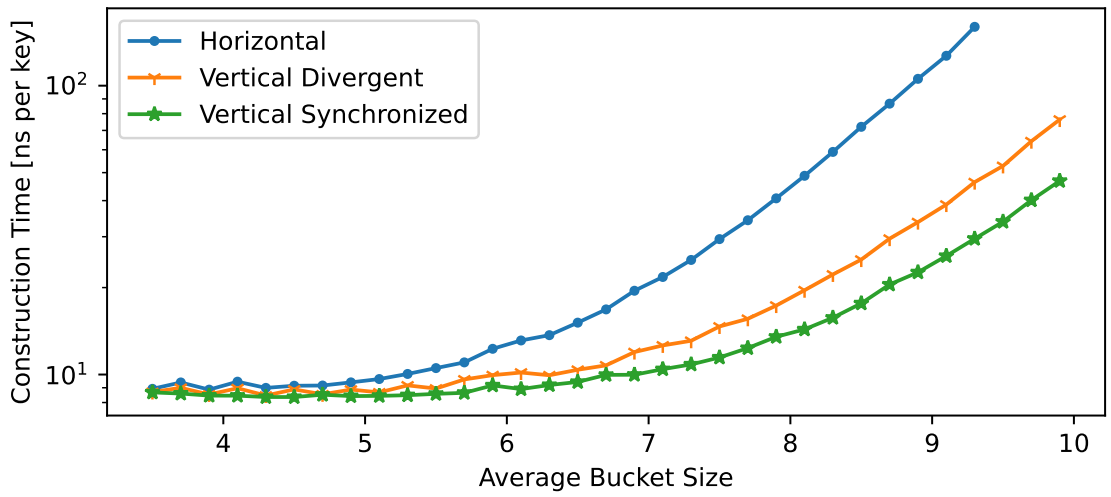

Figure 7.2.: Construction time of the different implementations for a range of average bucket sizes.

7.3.2. Brute Force

The first search strategy we evaluate is the brute force approach.

7.3.2.1. Techniques

In Section 6.4.5 we introduced three techniques to implement brute force search on the GPU. We compare those techniques in Figure 7.2 for different average bucket sizes. The comparison shows that the vertical synchronized technique consistently outperforms the other two in our experimental setup.

7.3.2.2. Bucket Size Distribution

The distribution of the bucket sizes significantly impacts the performance of the algorithm. In Section 5.3 we introduced an analytical approach for finding an optimal distribution. We presented three solutions: (1) a numerical solution to minimize total construction time, (2) a variation of 1 which takes into account local collisions and (3) a closed form approximation of 1 which is originally intended to achieve constant expected pilot values. Figure 7.3 compares the performance of 2, 3 and the distribution functions used in FCH. The closed form approximation performed poorly in our setting because it had to deal with a lot of local collisions. Local collisions are a result of the relatively small partition sizes. Future work which uses much larger or no partitions can use the closed form solution.

The numerical distribution which takes into account local collisions performed best in our scenario. One reason why it outperforms the distribution of FCH becomes apparent by looking at Figure 7.4. The FCH distribution of bucket sizes results in two "hills". One at insertion index 70 and the other one between index 180 and 260. This is because there are only two different expected bucket sizes. The "waves" are a result of the discrete nature of bucket sizes: The probability of insertion decreases steadily with each insertion until the bucket size decreases abruptly by one. The averaging causes those jumps to be less prominent. The integral of the two shown functions is proportional to the respective total search time, if we neglect local collisions. For larger average bucket sizes, the two "hills" of FCH are even more prominent relative to our optimized distribution. According to Figure 7.3, our optimized distribution is 0.1 bits per key better than the distribution of FCH when compared for an equal construction time of 100 ns. Surprisingly, at a certain point the space consumption even increases when using the FCH distribution of bucket sizes for larger average bucket sizes.

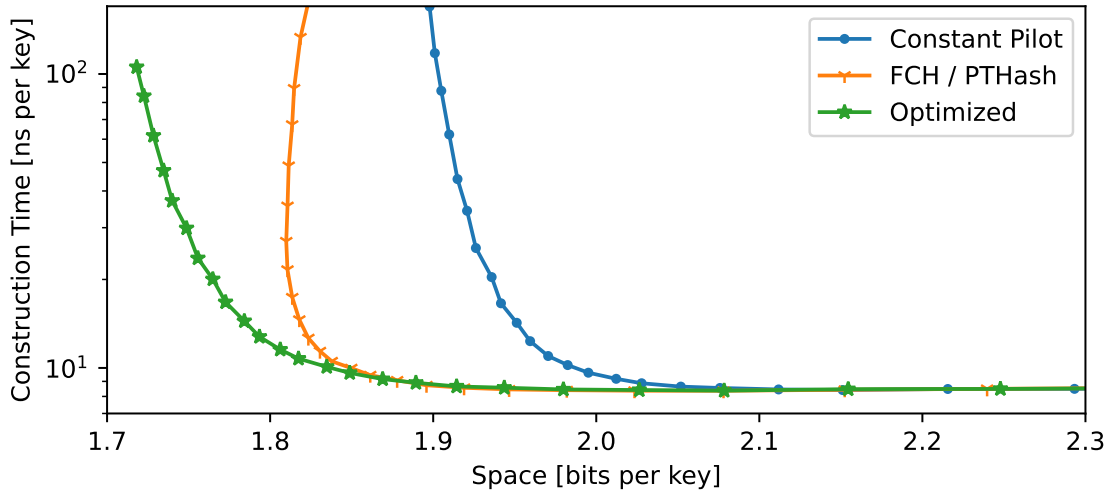

Figure 7.3.: Construction time versus space consumption for different distributions of bucket sizes. The curves are generated by gradually changing the average bucket size. Differences in query time are below 1 ns. "FCH" is the distribution from related work which uses two different expected bucket sizes. "Optimized" is our numerical solution which takes into account local collisions. "Constant Pilot" is the closed form approximation.

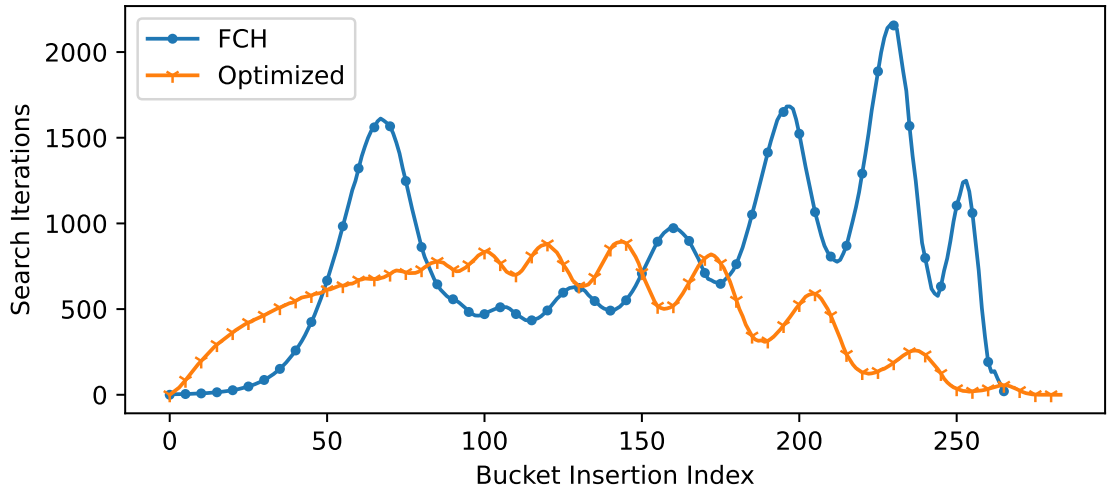

Figure 7.4.: We compare the performance of the FCH distribution of bucket sizes and our optimized distribution. The graph shows the expected number of search iterations at all bucket insertion indices. The expected iteration count is calculated using Equation 5.2. One iteration corresponds to calculating the position of one key and checking if the position is free. The values are averaged across all partitions. This plot was generated such that both distributions result in 1.9 bits per key by choosing the required average bucket sizes respectively.

7.3.3. Indexed Search

The brute force search algorithm scales exponentially in terms of bucket size. In Section 4.4 we introduced a technique which allows aligning an entire bucket to free positions. Analysis suggests that the approach scales better in terms of bucket size. Unfortunately, it requires a large amount of space to store the index. This is not a strength of our GPU because it only has 83 bytes of shared memory per thread. The indexing approach is completely dominated by brute force search. However, we still consider it theoretically interesting and hope to inspire future search algorithms. In particular, we pointed out improvements to reduce the large space overheads in Section 4.4.3 and Section 6.4.6.1. An implementation on a CPU would be interesting given its relatively large amount of L1 cache per thread.

7.3.4. Randomized Search in Pilot Limited Space

Compact encoding allows for direct access to individual pilots. The disadvantage is the space consumption. In Section 4.5 we showed that the space consumption of the compact orthogonal encoding is in $O(B \cdot \log_2(\log(B)))$, where B is the total number of buckets that have to be encoded. The idea of the randomized search introduced in Section 4.5 is to limit the pilot values to some maximum value. Consequently reducing the space consumption to $O(B)$. If a pilot is not found within this constraint the bucket is inserted with collisions. Buckets involved in the collision are reinserted. If the search does not finish within a certain number of reinsertions, we restart the search of the partition. Unfortunately, the randomized search approach suffers from something similar like the indexing approach. We have to keep track which bucket occupies which position to allow efficient evaluation of the heuristic. The space overhead results in less workgroups being active at the same time, because there is not enough shared memory available. This is very much apparent when choosing a pilot limit that is so large that no randomized search is required because all pilots can be found using the brute force approach. In that scenario, the randomized search approach is still about 10 times slower than the usual brute force approach which does not have to keep track of the individual bucket positions. Again, we recommend a CPU implementation for future work.

7.4. Construction Time — Scaling

The GPU is a throughput optimized device. It requires a large number of independent processing tasks to fully utilize its resources. Figure 7.5 shows how the construction throughput is influenced by input size. The device is fully utilized at an input size of at least 25 million. A CPU alternative should be considered for input sizes which are significantly below that threshold. The GPU throughput drops noticeably at an input size of 75 million. GPU timestamps show that determining the bucket sizes gets slower. Presumably because the GPU cannot handle the global atomic functions, which we use to count the bucket sizes, in linear time.

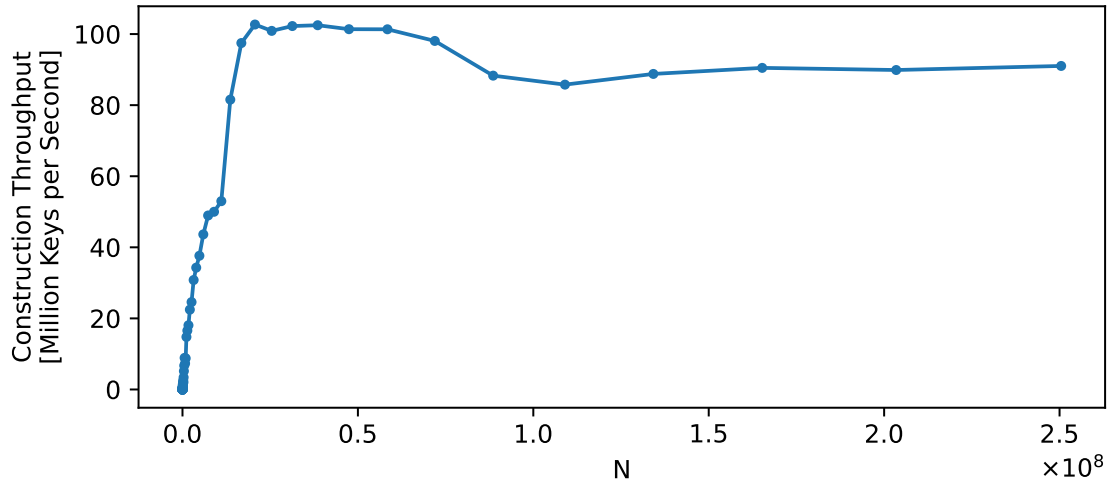

Figure 7.5.: Construction throughput for different input sizes.

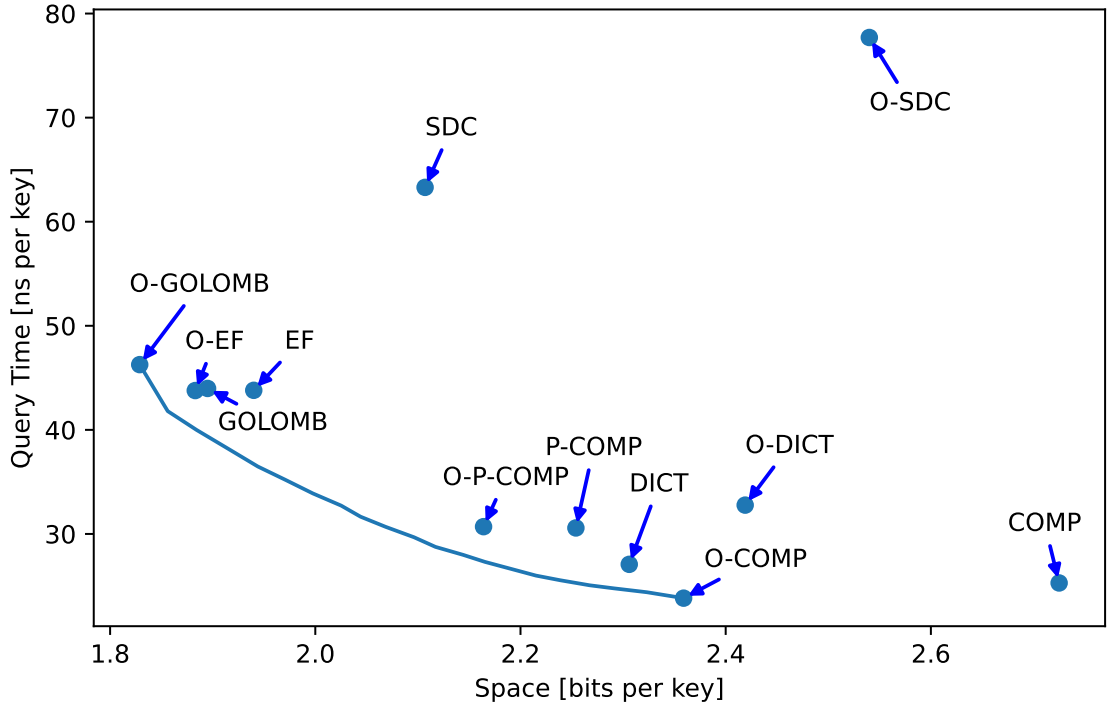

Figure 7.6.: Query time and space consumption of different encoding techniques. "EF" is Elias-Fano, "DICT" is dictionary, "COMP" is compact, "P-COMP" is partitioned compact, "SDC" is Solid Dense Coding. "O-" is the respective orthogonal variant. The orthogonal encoding also allows using different monolithic encoder techniques. The curve is generated by encoding the buckets of larger expected size using compact encoding and smaller buckets using Golomb encoding. One can gradually change the proportion of the respective techniques to obtain a smooth connection between the orthogonal Golomb and the orthogonal compact encoding. The curve also represents the Pareto front in this configuration.

7.5. Encoding

In the following we compare different encoding techniques in terms of the three metrics and for different input sizes.

7.5.1. Query Time and Space — Tradeoff

Figure 7.6 compares the tradeoff between query time and space of different encodings. An orthogonal encoder using a combination of the compact and the Golomb technique represents the Pareto front. The orthogonal encoding improves space consumption of the compact and Golomb encoding by 0.36 and 0.07 bits per key respectively when compared to their monolithic variant. Orthogonal and monolithic encodings have the same query time when using the Golomb and compact technique. Elias-Fano was also improved by orthogonal encoding, but it is not part of the Pareto front. The orthogonal technique worsened the performance of Solid Dense Coding and the dictionary encoding. Using the partitioned compact encoding improves the space consumption compared to the normal compact encoding. However, partitioning the compact encoding also increases the query time such that it is also not part of the Pareto front.

7.5.2. Query Time and Space — Scaling

Figure 7.7 shows how space and query time relate to input size. Note the sudden increase in query time beyond L3 capacity. This demonstrates the advantage of an MPHF. It has a small size and can be present higher in the memory hierarchy. In terms of encoding techniques, Golomb encoding has a high overhead for small input sizes. However, we are more interested in large inputs. Golomb encoding has a constant per key space usage as N approaches infinity. In Section 6.5.2.1 we show that an upper limit for the expected space usage per key of orthogonal compact encoding is $O(\log_2(\log(N)))$ bits per key. We do not have an asymptotic lower bound, but compact encoding surely does not have a constant space usage per key. This can also be seen by the slight but visible space increase of compact encoding in Figure 7.7. The scaling issue is addressed using partitioned compact encoding resulting in a constant expected space usage per key. We have seen in the previous section that partitioned compact encoding is not part of the Pareto front which was measured for $N = 10^8$. For inputs larger than that, one has to reevaluate the Pareto front. At some large enough value of N, the partitioned compact encoding will eventually be part of the Pareto front, given its better scaling behavior.

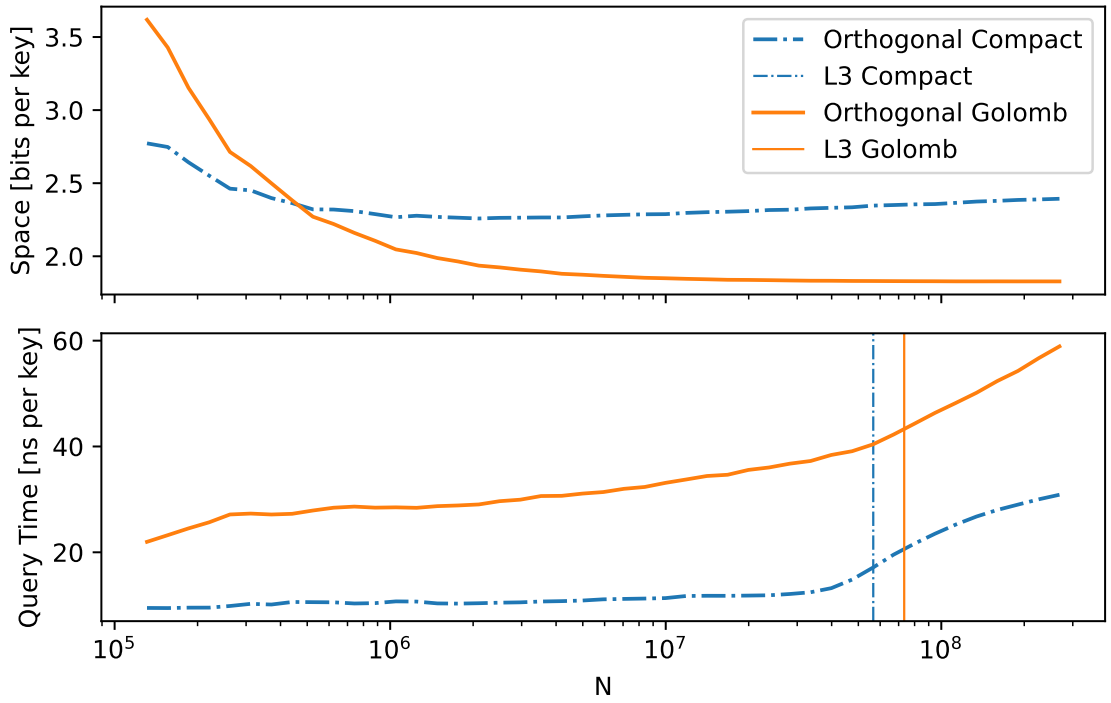

Figure 7.7.: Space and query time for different input sizes using either the orthogonal compact or the orthogonal Golomb encoding. The vertical lines represent the input size at which the total space consumption of the respective technique matches exactly the size of the L3 cache capacity.

7.5.3. Partitioning Overhead

A disadvantage of the partitioning approach is that additional partition offset data needs to be accessed in each query resulting in an increased latency. The partition size is also required but is calculated as the difference of consecutive offsets. We remark that partition data should not be dismissed as a troublesome artifact of parallelization. Partition data carries actual informational content which theoretically reduces the amount of data required for the pilot values. Section 6.5.4 introduces two techniques for partition offset encoding. In the following we first compare them in terms of space and then in terms of query time. Elias-Fano encoding is the first technique and results in an amortized space consumption of 0.007 bits per key. The second technique stores the difference to the expected offset value of each partition in a compact encoder. Surprisingly, it requires 0.007 bits per key too. However, we can expect the Elias-Fano encoding to be more space efficient for larger inputs given its linear scaling, while storing the difference to the expected value is not linear. Elias-Fano results in a total query time of 49 ns per key. The compact encoding reduces this to just 41 ns per key. Note that we also tested the query algorithm without accessing the partition data. The query time in that case is 39 ns. This of course does not result in a valid MPHF query but provides insight into the overhead caused by fetching partition data. The overhead of just 2 ns is quite small, presumably because the partition data is available higher in the memory hierarchy compared to pilot data.

7.5.4. Encoding Time

The encoding time generally makes up for only a small portion of the total construction time. We implemented the dictionary encoding on the GPU which requires just 10 ps per key. For convenience we used a CPU implementation for the other encoding techniques. The orthogonal encoding is trivially parallelized by concurrently encoding the encapsulated monolithic encoders. Orthogonal Golomb encoding requires about 150 ps per key on the CPU and orthogonal compact encoding just 52 ps per key. Encoding the partition offsets is negligible because of the relatively small amount of data. We measured an amortized 1 ps per key for the compact encoding of partition offsets.

7.6. Pareto Fronts

Our implementation is configurable to reach different tradeoffs between the three metrics. By increasing the average bucket size A, we can obtain a more compact representation at the cost of a higher construction time. The orthogonal encoding allows choosing a tradeoff between space consumption and query time by setting the proportion of Golomb and compact encoders. Changing those two parameters effectively creates a three dimensional Pareto front. We visualize the Pareto front using two dimensional Cross sections. Figure 7.8 shows the three possible orthogonal planes. In each plane we provide slices at three different depths. All data points which are shown are within the specified error of the respective depth.

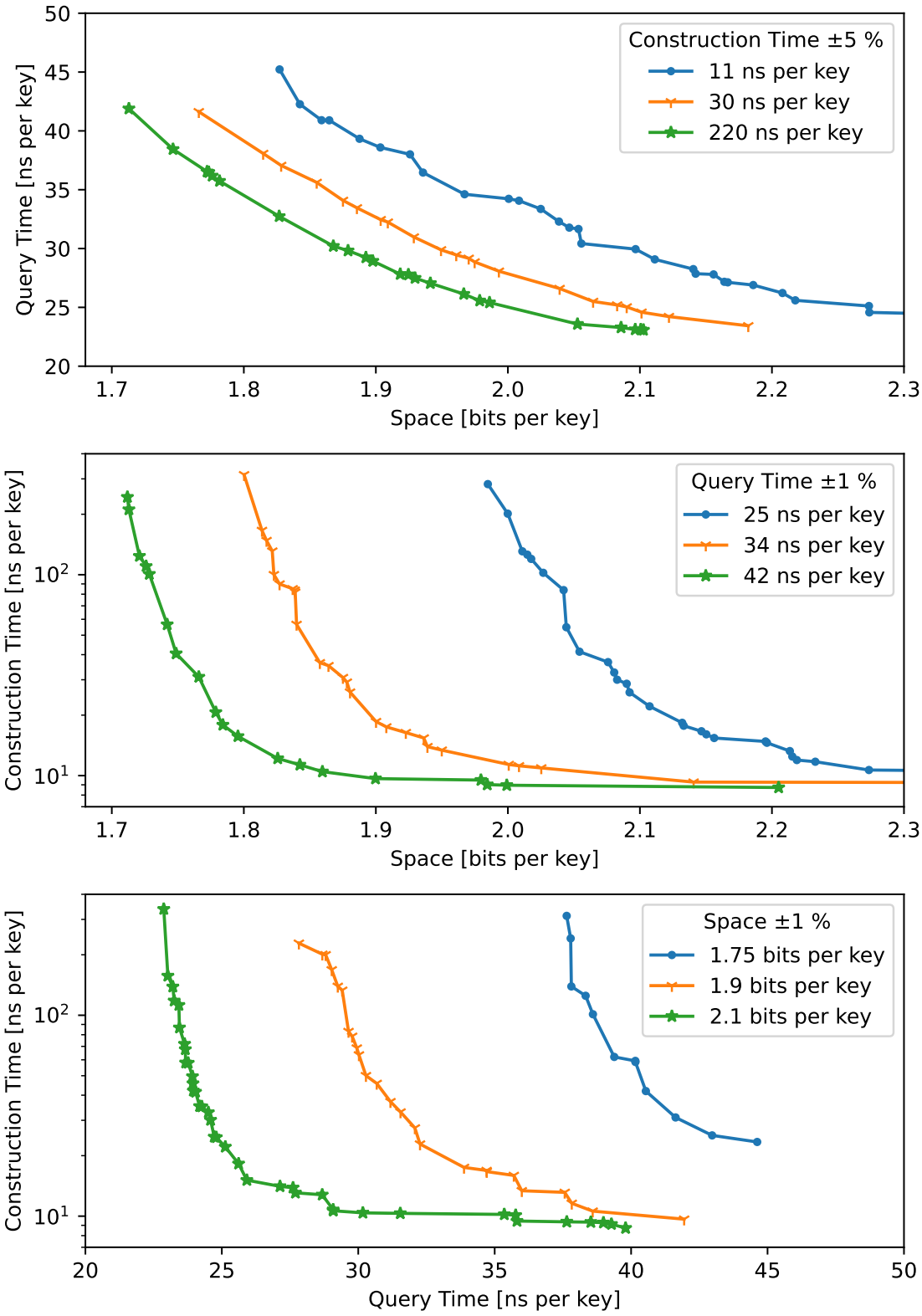

Figure 7.8.: Slices of the Pareto front for all three planes

Method	Construction	Query	Space	Energy
PTHash [24], c = 10.0, α = 0.99, PC	191 ns	27 ns	3.09 bit	9.9 μJ
PTHash [24], c = 4.5, α = 0.99, D-D	1096 ns	21 ns	2.42 bit	57.0 μJ
PTHash [24], c = 3.5, α = 0.99, EF	4963 ns	35 ns	1.97 bit	258.1 μJ
GPURecSplit [23], l = 8, b = 2000	32 ns	141 ns	1.71 bit	-
GPURecSplit [23], l = 10, b = 2000	63 ns	121 ns	1.61 bit	23.6 μJ
GPURecSplit [23], l = 14, b = 2000	231 ns	117 ns	1.58 bit	86.6 μJ
Our, A = 7, Orthogonal-Compact	10 ns	24 ns	2.25 bit	-
Our, A = 8.5, Orthogonal-Golomb	18 ns	45 ns	1.77 bit	-
Our, A = 10, Orthogonal-Compact	52 ns	23 ns	2.18 bit	16.6 μJ
Our, A = 11, Orthogonal-Golomb	122 ns	43 ns	1.71 bit	39.0 μJ
Our, A = 13, Orthogonal-Golomb	913 ns	43 ns	1.69 bit	292.2 μJ

Table 7.2.: Metrics for different competitor configurations. All metrics are given per key. Energy is measured using the difference of the power consumption relative to idle power. We measured power consumption of GPU implementations during the searching phase only. We therefore cannot provide accurate data for configurations that are dominated by transfer overheads.

7.7. Comparison with state-of-the-art

We compare our implementation to GPURecSplit [23] and to single threaded PTHash [24]. We use the same experimental setup and benchmark technique for all competitors. We maintain the input size of 100 million, which we used throughout this evaluation. Recall that the theoretical lower bound of an MPHF is 1.44 bits per key. At an input size of 100 million, the space of an MPHF would be equal to the capacity of L3 cache if it used 1.34 bits per key. Thus, more compact MPHFs are rewarded with a lower query time because there is an increased probability that the required data is cached. Compare that to an input size of 10 million. In that scenario, any reasonable MPHF technique fits into L3 but L2 is still too small. Thus, there would be no reward for more space efficient representations like RecSplit.

7.7.1. Query and Space

Table 7.2 shows a comparison using different competitor configurations. It clearly shows the distinct strengths of GPURecSplit and bucket based implementations like PTHash and ours. GPURecSplit can produce MPHFs with a space consumption that is practically unreachable by our implementation. GPURecSplit is up to 0.15 bits per key more compact compared to our implementation. However, PTHash and our implementation are a clear winner in terms of query time. They only require a single memory access when using the compact encoding. Depending on the respective configurations, queries are between 2 to 6 times faster than GPURecSplit. The query times of PTHash can be faster by about 2 ns compared to our implementation. This is also the overhead we measured for retrieving the partition data.

Query	Space	PTHash [24] Configuration	Our Implement. Configuration	Construct Speedup	Financial Break Even
14 ns	2.10 bit	c = 3.0, α = 0.99, PC	A = 7.7, L = 0.65	319	-
23 ns	1.97 bit	c = 3.0, α = 0.99, EF	A = 6.8, L = 0.12	373	-
22 ns	1.90 bit	c = 2.5, α = 0.99, EF	A = 8.4, L = 0.17	948	10^{13}
15 ns	2.04 bit	c = 2.5, α = 0.99, PC	A = 8.2, L = 0.60	1170	10^{13}
15 ns	2.00 bit	c = 2.0, α = 0.99, PC	A = 8.7, L = 0.60	8456	10^{12}
22 ns	1.88 bit	c = 2.0, α = 0.99, EF	A = 8.5, L = 0.25	9926	10^{12}

Table 7.3.: Speedup of our implementation in terms of construction time compared to PTHash [24]. Configurations are chosen such that space and query time is equal. The experiments were conducted for a smaller size of N=10 million keys, to keep the construction time of PTHash feasible. This is also the reason for the much lower query times compared to the other sections of the evaluation. In our configuration, A is the average bucket size. We use orthogonal encoding. L is the relative number of compact encoders. Accordingly, (1-L) is the relative number of Golomb encoders. The "financial break even" is described in Section 7.7.4.

7.7.2. Construction Speedup

Our GPU implementation is several times faster than single threaded CPU PTHash. Table 7.3 shows speedup factors for different configurations of PTHash. The configuration of our implementation is chosen such that it matches the respective query time and space consumption of PTHash. We remark that comparisons across different hardware have to be interpreted carefully. In particular because PTHash only uses a single CPU thread. Our construction time is dominated by constant time overheads such as CPU to GPU key transfer when using small (<8) average bucket sizes. This is also why the speedup is relatively low in that case. It is only for larger average bucket sizes that the overheads are negligible and the strength of the parallel search implementation becomes apparent. Another reason why the construction speedup becomes larger for smaller space requirements is because of our improvements to the encoding step. This allows us to use a lower average bucket size than otherwise necessary to obtain the same space.

7.7.3. Energy Consumption

Table 7.4 shows measurements for power consumption of the different techniques. We measured the total power consumption of the machine using the wattmeter "Voltcraft Energy Check 3000". Power consumption of the GPU can be measured using an integrated wattmeter which is accessible by software using the "nvidia-smi" command. We measured power consumption for GPURecSplit, PTHash and our implementation as well as when the machine was idle. PTHash only uses the CPU and requires about 52 W more compared to idle. The power consumption of GPU based implementations is much higher. GPURecSplit uses 375 W and our implementation 320 W more compared to idle. GPURecSplit uses both more GPU and CPU power compared to our implementation. The increased CPU power is partially explained by the fact that GPURecSplit uses the CPU for preprocessing while the

Executing	Total	Δ Idle	GPU	Rest
Idle	75 W	0 W	28 W	47 W
PTHash	127 W	52 W	28 W	99 W
GPURecSplit	450 W	375 W	348 W	102 W
Our	395 W	320 W	320 W	75 W

Table 7.4.: We measured power consumption of the entire machine and of the GPU. The difference between total and GPU power is calculated as the rest (mostly CPU) of the system's power.

CPU in our implementation only controls the GPU. Furthermore, they use CUDA which is associated with a higher CPU overhead compared to Vulkan [38]. GPURecSplit requires about 10 % more GPU power than our implementation. Presumably, because it uses other instructions. Different GPU instructions require different amounts of power [39].

We show energy consumption per key for different competitor configurations in Table 7.2. Note that they are calculated based on construction time per key and the power consumption difference to idle. Although GPU techniques have a higher power consumption, they require less energy per key because they process a key much faster than a CPU.

7.7.4. Economic

The last criteria we discuss are financial differences. Note that the following is only meant as a side fact and is not a rigorous analysis because it heavily depends on the underlying hardware of the machine, the configurations and several other factors. First of all, the price of the hardware used in our experiments is 1715€ [40] and 252€ [41] for the GPU and CPU respectively. However, as discussed earlier the GPU requires much less energy per key compared to the CPU. The average electricity price in Germany is 101€/MWh [42]. The GPU has made up its higher hardware price by its reduced power consumption after a certain number of keys have been processed. We calculate this as the "financial break even" point in Table 7.3 for different configurations. Note that we do not provide data for small average bucket sizes because we cannot accurately measure power consumption in that scenario. Surprisingly, it only requires about 10^{13} keys before our GPU implementation is cheaper then CPU PTHash if electricity price is considered. PTHash requires in the magnitude of one year to process this many keys. Our GPU implementation is finished in a few hours.

8. Conclusion

This thesis contributes significant improvements and a GPU implementation for all steps of bucket based MPHF construction. In the first step, keys are distributed into buckets. We show that an optimal distribution of bucket sizes can be approximated as the solution of a differential equation. The optimized distribution results in a reduction of space consumption by about 0.1 bits per key when compared to state-of-the-art bucket size distributions at equal construction time. The buckets are ordered by non-increasing size in the second step. We show that space is reduced by about 0.05 bits per key when sorting buckets of the same size by increasing expected size. In the third step, the buckets are inserted by finding a pilot for each bucket that maps all of the bucket's keys to free positions. We show how to efficiently parallelize the brute force search of a pilot on a lock-step architecture. In the final step, the pilots are encoded. We apply Golomb encoding and introduce orthogonal encoding. A combination of both techniques improves space consumption by 0.12 bits per key compared to state-of-the-art Elias-Fano encoding.

Our implementation uses the parallel processing power of a GPU to accelerate MPHF construction. It can construct an MPHF of 1.73 bits per key, while maintaining fast query and construction time of 44 ns and 36 ns per key respectively. In the same construction time, our implementation can build an MPHF with an even lower query time of just 23 ns per key at the cost of an increased space of 2.19 bits per key. Those results are obtained for 100 million keys on a Nvidia RTX 3090. PTHash cannot reach 1.73 bits per key. An MPHF with a higher space consumption of 1.88 bits per key is constructed 9926 times faster by our implementation compared to PTHash.

Future Work. In the following we propose several ideas for future work. (I) The input size of our implementation is currently limited by the amount of available memory on the GPU. This issue can be solved in future work by dividing the keys into *batches* on the CPU. Each batch is small enough to fit into GPU memory. Batching also allows utilization of multi-GPU setups, because each batch can be processed independently on different GPUs. (II) We presented dynamic bucketing which is a novel construction algorithm that can be researched in future work. (III) It would be interesting to understand how the secondary order of equally sized buckets and the distribution of bucket sizes affects the entropy of the MPHF. (IV) The space consumption of our MPHF is about 0.13 bits per key off from the entropy of our theoretical model. In general, future work should bring theoretical and practical results even closer together.

A. Declaration of Means

I used ChatGPT [43] experimentally to implement the 64 bit multiplication procedure using 32 bit arithmetic in GLSL. There is no other part of the implementation or this thesis which was in any way, or even partially, written or inspired by ChatGPT. I further declare that in my implementation I used an abstraction of the Vulkan API which was written for a previous project. Finally, the local and global parallel prefix sum implementations were also from one of my earlier Vulkan projects and only required minimal modifications.

Bibliography

[1] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. "Recsplit: Minimal perfect hashing via recursive splitting". In: *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2020, pp. 175–185.

[2] Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. "ShockHash: Towards Optimal-Space Minimal Perfect Hashing Beyond Brute-Force". In: *arXiv preprint arXiv:2308.09561* (2023).

[3] Giulio Ermanno Pibiri and Roberto Trani. "PTHash: Revisiting FCH minimal perfect hashing". In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2021, pp. 1339–1348.

[4] Djamal Belazzougui and Gonzalo Navarro. "Alphabet-independent compressed text indexing". In: *ACM Transactions on Algorithms (TALG)* 10.4 (2014), pp. 1–19.

[5] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. "Perfect hashing for network applications". In: *2006 IEEE International Symposium on Information Theory*. IEEE. 2006, pp. 2774–2778.

[6] Chin-Chen Chang and Chih-Yang Lin. "Perfect hashing schemes for mining association rules". In: *The Computer Journal* 48.2 (2005), pp. 168–179.

[7] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. "Fast prefix search in little space, with applications". In: *Algorithms–ESA 2010: 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I 18*. Springer. 2010, pp. 427–438.

[8] Giulio Ermanno Pibiri and Rossano Venturini. "Efficient data structures for massive n-gram datasets". In: *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval*. 2017, pp. 615–624.

[9] Victoria G Crawford, Alan Kuhnle, Christina Boucher, Rayan Chikhi, and Travis Gagie. "Practical dynamic de Bruijn graphs". In: *Bioinformatics* 34.24 (2018), pp. 4189–4195.

[10] Andrei Broder and Michael Mitzenmacher. "Network applications of bloom filters: A survey". In: *Internet mathematics* 1.4 (2004), pp. 485–509.

[11] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. "Dynamic perfect hashing: Upper and lower bounds". In: *SIAM Journal on Computing* 23.4 (1994), pp. 738–761.

[12] Edward A Fox, Qi Fan Chen, and Lenwood S Heath. "A faster algorithm for constructing minimal perfect hash functions". In: *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*. 1992, pp. 266–273.

[13] Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. "Hash, displace, and compress". In: *European Symposium on Algorithms*. Springer. 2009, pp. 682–693.

[14] M. L. Fredman and D. E. Willard. "BLASTING through the Information Theoretic Barrier with FUSION TREES". In: *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*. STOC '90. Baltimore, Maryland, USA: Association for Computing Machinery, 1990, pp. 1–7. ISBN: 0897913612.

[15] *Nvidia Ampere GA102 GPU architecture*. URL: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf (visited on 10/01/2023).

[16] *Vulkan 1.3.265 - A Specification*. URL: https://registry.khronos.org/vulkan/specs/1.3/html/index.html (visited on 10/01/2023).

[17] *Parallel Prefix Sum*. URL: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda (visited on 10/01/2023).

[18] Paul Tarau. "Deriving a fast inverse of the generalized cantor N-tupling bijection". In: *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2012.

[19] Kimmo Fredriksson and Fedor Nikitin. "Simple compression code supporting random access and fast string matching". In: *Experimental Algorithms: 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007. Proceedings 6*. Springer. 2007, pp. 203–216.

[20] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. "Retrieval and perfect hashing using fingerprinting". In: *Experimental Algorithms: 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29–July 1, 2014. Proceedings 13*. 2014, pp. 138–149.

[21] Piotr Beling. "Fingerprinting-based minimal perfect hashing revisited". In: *ACM Journal of Experimental Algorithmics* (2023).

[22] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. "Fast scalable construction of (minimal perfect hash) functions". In: *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings 15*. Springer. 2016, pp. 339–352.

[23] Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. "High Performance Construction of RecSplit Based Minimal Perfect Hash Functions". In: *31st Annual European Symposium on Algorithms (ESA 2023)*. 2023, 19:1–19:16.

[24] Giulio Ermanno Pibiri and Roberto Trani. "Parallel and external-memory construction of minimal perfect hash functions with PTHash". In: *IEEE Transactions on Knowledge and Data Engineering* (2023).

[25] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. “Fast and scalable minimal perfect hashing for massive key sets”. In: *arXiv preprint arXiv:1702.03154* (2017).

[26] Sean Weaver and Marijn Heule. “Constructing minimal perfect hash functions using SAT technology”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 02. 2020, pp. 1668–1675.

[27] P. Erdös and A. Rényi. “On Random Graphs I”. In: *Publicationes Mathematicae Debrecen* 6 (1959), p. 290.

[28] C. E. Shannon. “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[29] Aaron Kiely. “Selecting the Golomb parameter in Rice coding”. In: *IPN progress report* 42 (2004), p. 159.

[30] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[31] Richard Arratia, Skip Garibaldi, and Joe Kilian. “Asymptotic distribution for the birthday problem with multiple coincidences, via an embedding of the collision process”. In: *Random Structures & Algorithms* 48.3 (2016), pp. 480–502.

[32] Steve Drekic and Michael Z Spivey. “On the number of trials needed to obtain k consecutive successes”. In: *Statistics & Probability Letters* 176 (2021), p. 109132.

[33] Wolfram Research Inc. *Mathematica, Version 13.3*. Champaign, IL, 2023. URL: https://www.wolfram.com/mathematica (visited on 10/01/2023).

[34] Sheehan Olver and Alex Townsend. “Fast inverse transform sampling in one and two dimensions”. In: *arXiv preprint arXiv:1307.1223* (2013).

[35] *MurmurHash3 Github*. URL: https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp (visited on 10/01/2023).

[36] Peter Sanders. “Emulating MIMD Behaviour on SIMD-Machines.” In: *EUROSIM*. 1994, pp. 313–320.

[37] Bennett Eisenberg. “On the expectation of the maximum of IID geometric random variables”. In: *Statistics & Probability Letters* 78.2 (2008), pp. 135–143.

[38] Nadjib Mammeri and Ben Juurlink. “Vcomputebench: A vulkan benchmark suite for gpgpu on mobile and embedded gpus”. In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2018, pp. 25–35.

[39] Sylvain Collange, David Defour, and Arnaud Tisserand. “Power consumption of GPUs from a software perspective”. In: *Computational Science–ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part I 9*. Springer. 2009, pp. 914–923.

[40] *Amazon Nvidia RTX 3090*. URL: https://www.amazon.de/Gigabyte-Gaming-OC-24GD-Schwarz-3304808037/dp/B09X3JXDZ8 (visited on 10/01/2023).

[41] *Amazon Intel i7-11700*. URL: https://www.amazon.de/Intel-i7-11700-Desktop-Prozessor-Basistakt/dp/B08TX3VLVQ/ (visited on 10/01/2023).

[42] *Electricity Price Germany Statista*. URL: https://www.statista.com/statistics/1267541/germany-monthly-wholesale-electricity-price/ (visited on 10/01/2023).

[43] *ChatGPT*. URL: https://chat.openai.com/ (visited on 10/01/2023).