PHOBIC: Perfect Hashing With Optimized Bucket Sizes and Interleaved Coding

Stefan Hermann
Karlsruhe Institute of Technology, Germany

Hans-Peter Lehmann
Karlsruhe Institute of Technology, Germany

Giulio Ermanno Pibiri
Ca' Foscari University of Venice, Italy
ISTI-CNR, Pisa, Italy

Peter Sanders
Karlsruhe Institute of Technology, Germany

Stefan Walzer
Karlsruhe Institute of Technology, Germany

Abstract

A minimal perfect hash function (or MPHF) maps a set of n keys to $[n] := \{1, \ldots, n\}$ without collisions. Such functions find widespread application e.g. in bioinformatics and databases. In this paper we revisit PTHash – a construction technique particularly designed for fast queries. PTHash distributes the input keys into small buckets and, for each bucket, it searches for a hash function seed that places its keys in the output domain without collisions. The collection of all seeds is then stored in a compressed way. Since the first buckets are easier to place, buckets are considered in non-increasing order of size. Additionally, PTHash heuristically produces an imbalanced distribution of bucket sizes by distributing 60% of the keys into 30% of the buckets.

Our main contribution is to characterize, up to lower order terms, an *optimal* choice for the expected bucket sizes, improving construction throughput for space efficient configurations both in theory and practice. Further contributions include a new encoding scheme for seeds that works across partitions of the data structure and a GPU parallelization.

Compared to PTHash, PHOBIC is 0.17 bits/key more space efficient for same query time and construction throughput. For a configuration with fast queries, our GPU implementation can construct an MPHF at 2.17 bits/key in 28 ns/key, which can be queried in 37 ns/query on the CPU.

2012 ACM Subject Classification Theory of computation → Data compression; Information systems → Point lookups

Keywords and phrases Compressed Data Structures, Minimal Perfect Hashing, GPU

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.69

Related Version *Full Version*: `https://arxiv.org/abs/2404.18497` [17]

Supplementary Material *Software (CPU code)*: `https://github.com/jermp/pthash/tree/phobic`
archived at `swh:1:dir:768316ce78cf600adb51747f53de4554f2810bf4`
Software (GPU code): `https://github.com/stefanfred/PHOBIC-GPU`
archived at `swh:1:dir:cc1a9df8d87fb54fced6653e4f95ae23b4c05772`
Software (Comparison with competitors): `https://github.com/ByteHamster/MPHF-Experiments`
archived at `swh:1:dir:ee4700d510c929f8acc0e49bdd256b3357d21d15`

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500). The project also received funding form the European Union's Horizon Europe research and innovation programme (EFRA project, Grant Agreement Number 101093026). This work was also partially supported by DAIS – Ca' Foscari University of Venice within the IRIDE program.

Acknowledgements This paper is based on the Master's thesis [16] of Stefan Hermann, which contains a more detailed evaluation and description of the GPU implementation.

1 Introduction

A *hash function* maps a set S of n keys to a range of integers $[m] := \{1, \ldots, m\}$, regardless of whether multiple keys collide on the same output. A *perfect* hash function (PHF) on S is a mapping *without collisions*. This requires $m \geq n$. The function does not necessarily have to store the keys explicitly. It only has to store enough information to prevent collisions, which are more likely when m is close to n. In the extreme case of $m = n$, the mapping is called a *minimal* perfect hash function (MPHF). For simplicity, we only consider the minimal case in this paper. However, our contributions are applicable for the non-minimal case as well. PHFs find widespread practical application e.g. in compressed full-text indexes [3], computer networks [23], databases [8], prefix-search data structures [1], language models [29], bioinformatics [11, 26], and Bloom filters [7]. The three main performance attributes of an MPHF are low space consumption, fast construction, and fast queries. Concerning space, the lower bound is $\log_2(e) \approx 1.44$ bits/key [24]. Practically viable approaches can get within a few percent of the lower bound, but do so with some sacrifices in running time [19, 21]. This paper is concerned with a technique that is focused on achieving fast query times. For example, this is very important when using perfect hashing to implement a static hash table that is both space-efficient and allows fast search.

Perfect Hashing Through Bucket Placement. To achieve fast query time we ideally only have to retrieve a single value from memory and directly obtain the result of the query by combining this value and the key. Perfect hashing through bucket placement is an MPHF technique which yields such a simple query algorithm. It takes the n keys and hashes them to small buckets. The algorithm *places* the buckets one after another by searching for a seed of a hash function that maps the keys to the output domain $[n]$. A seed is accepted if none of the keys in the bucket collide with another key of a previously placed bucket. We illustrate this in Figure 1. The first buckets are easier to place because the output domain is less full. Therefore, the methods insert the buckets in order of non-increasing size. While CHD [2] uses buckets of constant expected size, FCH [13] and PTHash [27, 28] set aside 30% of "heavy" buckets that receive 60% of the keys in expectation, while 70% of "light" buckets receive only 40% of the keys in expectation. This imbalance in expected bucket

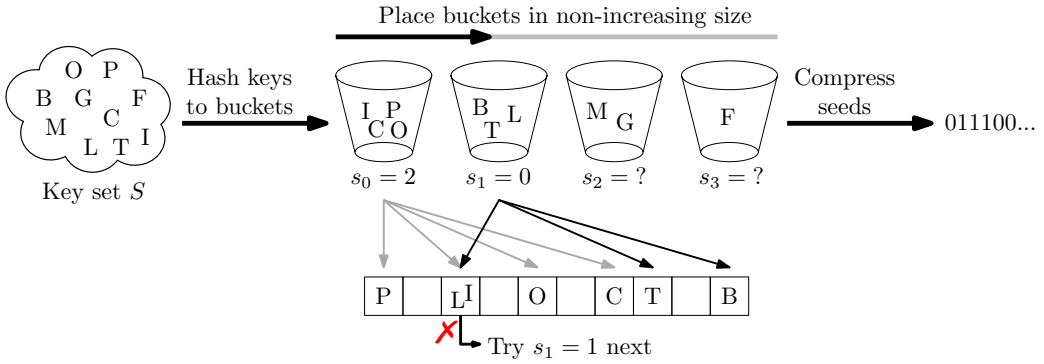


Figure 1 Example for perfect hashing through bucket placement.

sizes improves construction speed by further decreasing the size of the last, hardest to place, buckets. The resulting list of seed values are stored with various compression techniques, resulting in a variety of trade-offs between space consumption and query speed.

Contribution. This paper aims at improving the space efficiency and construction speed of PTHash, while maintaining its fast query speed. There are three ingredients. Our main contribution (in Section 4) is to characterize, up to lower order terms, an optimal distribution of expected bucket sizes, effectively taking the imbalance-trick used in FCH and PTHash to its logical conclusion. The distribution is easy-to-implement and greatly improves construction time and space efficiency in practice. Our second contribution (in Section 5.1) is to the compression of seed values. We group the seed values such that all seeds within one group follow the same statistical distribution. We can then tune a compressor for each group. Finally, we contribute (in Section 5.2) an implementation for *Graphics Processing Units* (GPUs) to speed up construction.

2 Related Work

Perfect hashing is an active area of research. In addition to perfect hashing through bucket placement, which we describe in Section 1, we now provide an overview over state-of-the-art approaches. For more details, refer to [28, Section 2].

Fingerprinting. Perfect hashing through fingerprinting [9, 25] is a technique aimed at fast construction and queries at the cost of reduced space efficiency. The idea is to map the n keys to γn positions using a hash function, where γ is a tuning parameter. A bit vector of length γn indicates positions that received exactly one key. Keys that are involved in collisions are handled recursively on another layer of the same data structure. A query operation descends through the recursive layers until it finds a 1-bit, meaning that the queried key was the only key mapping to that position. A rank operation on the bit vector for that position then gives the final MPHF value. FMPH [4] and BBHash [22] are publicly available implementations of this approach. FMPHGO [4] extends on this idea using a small number of brute-force re-tries to reduce the number of colliding keys. Fingerprinting based approaches are fast to construct but are outperformed in terms of space consumption and query time by PTHash.

Brute Force. RecSplit [12] first partitions the input into sets of equal expected size. It then recursively splits the key set of each partition until sets of small constant size (usually ≤ 16) are left. Within these sets, it finds a perfect hash function by brute force. RecSplit achieves space usage of about 1.56 bits/key. The resulting splitting tree has to be traversed during querying which implies considerably higher query costs compared to PTHash. The brute force search was later improved in SIMDRecSplit [5], which also parallelizes the construction on the GPU. To the best of our knowledge, RecSplit is the only other PHF construction technique that has a GPU implementation.

Perfect Hashing Through Retrieval. In perfect hashing through retrieval, every key has a number of candidate positions, determined by different hash functions. A retrieval data structure then stores which of the choices should be used for each key. Note that this implies some query overhead compared to PTHash. Early implementations include BPZ [6] and GOV [14]. SicHash [20] reduces space consumption using a mix of different retrieval data structures and some retries. ShockHash-RS [19, 21] combines 1-bit retrieval with the brute-force approach of RecSplit and currently is the most space-efficient approach to MPHFs with as little as 1.49 bits/key [19].

3 PHOBIC

In this section we present PHOBIC, a novel technique for perfect hashing through bucket placement. Before we dive into the actual algorithm we take a closer look at the parameters of perfect hashing through bucket placement.

3.1 Expected Bucket Sizes

It is known that larger average bucket sizes λ result in lower space consumption at the cost of an increased construction time. This was previously shown in CHD [2] for the case in which all buckets have the same expected size. However, it was previously not analyzed how the distribution of expected sizes influences space consumption. It is a simple insight that large λ already guarantees a space consumption close to the lower bound of $\log_2 e$ bits per key, *without any assumptions* on the distribution of expected sizes.

▶ **Proposition 1.** *Any specialization of perfect hashing through bucket placement requires between* $\log_2 e$ *bits per key and* $\log_2 e + \mathcal{O}(\frac{\log \lambda}{\lambda})$ *bits per key in expectation.*

Our goal therefore only needs to be to minimize construction time. Here we are faced with a lower bound for our family of approaches.

▶ **Proposition 2.** *Any specialization of perfect hashing through bucket placement has an expected construction time of* $\Omega(e^{\lambda})$ *per bucket.*

Propositions 1 and 2 are restated more formally and proved in the full version of this paper [17]. It is intuitively clear (and proved in the full version) that buckets should be processed in order from largest to smallest. The main remaining degree of freedom is to choose the expected sizes of the buckets. However, previous work has addressed this optimization problem only experimentally. Our main contribution (in Section 4) is to characterize, up to lower order terms, an optimal distribution of expected sizes. We achieve a construction time of $e^{\lambda(1+\varepsilon)}$ per bucket. Our distribution of expected sizes is easy-to-implement and greatly improves construction time and space efficiency in practice when compared to the distribution of expected sizes of related work.

3.2 The Algorithm

PHOBIC first splits the input set into disjoint *partitions* of expected equal size ρ using a pseudo-random hash function. We construct an MPHF within each partition. The various MPHFs are then logically "concatenated" into a single MPHF taking the prefix sum of the partition sizes. Partitioning has also been applied on PTHash in PTHash-HEM [28]. Like in PTHash-HEM, we use perfect hashing trough bucket placement to construct the MPHF of each partition. Each partition uses the same number $B := \lceil \rho/\lambda \rceil$ of buckets. To assign the keys to buckets we first hash them uniformly into the range $h \in [0, 1]$. Each key is then assigned to bucket $\lceil B\gamma(h) \rceil$, where $\gamma : [0, 1] \longrightarrow [0, 1]$ may be any function for now. We refer to γ as a *bucket assignment function*. Section 4 discusses how γ characterizes the expected sizes of the buckets and how to choose γ in an optimal way.

We now have all the ingredients to give pseudocode for construction and query in Algorithm 1. Our algorithm is implicitly parametrized by λ, ρ and b. We denote h_X as a pseudo-random hash function which uniformly maps into the set X.

Algorithm 1 PHOBIC construction and query.

Function construct(S, P, B, γ):
for all $k \in S$
  assign k to bucket $B \cdot \gamma(h_{[0,1]}(k))$
    of partition $h_{[P]}(k)$
offsets ← prefix sum of partition sizes
for all partitions *part*
  size ← number of keys in *part*
  free ← [*True*, ...] of size *size*
  sort buckets in non-increasing size
  for all *bucket* in *part*
    keys ← keys of *bucket*
    s ← placeBucket(*keys*, *size*, *free*)
    store seed s
return seeds, *offsets*

Function placeBucket(*keys*, *size*, *free*):
for $s = 1, 2, 3, \ldots$
  $pos \leftarrow \{h_{[size]}(k, s) : k \in keys\}$
  if $|pos| = size$ and $\forall p \in pos : free[p]$
    $free[p] \leftarrow False$ **for** $p \in pos$
    return s

Function query($key \in S, P, B, \gamma$):
$partition \leftarrow h_{[P]}(key)$
$bucket \leftarrow B \cdot \gamma(h_{[0,1]}(key))$
$offset \leftarrow offsets[partition]$
$size \leftarrow offsets[partition + 1] - offset$
$s \leftarrow$ seed of *bucket* in *partition*
return $offset + h_{[size]}(key, s)$

Our second contribution (in Section 5.1) is to improve the compression of seed values when using partitioning. Seeds are searched independently for each partition, but compressed together. We exploit that the seeds of the i-th bucket of each partition follow the same statistical distribution. This allows us to tune a compressor for each such index i. We store the seeds in an *interleaved* manner by consecutively placing the seeds for the i-th buckets from all partitions.

The construction of each of the many but small partitions requires only a tiny amount of memory. PHOBIC therefore naturally maps to the architecture of a *Graphics Processing Units* (GPU). Our final contribution is a GPU implementation (in Section 5.2) to further speed up construction compared to the CPU.

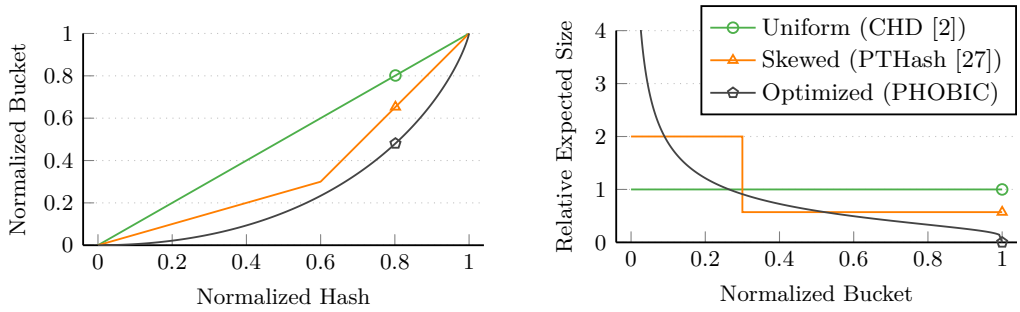
4 Optimizing Bucket Sizes

We first identify useful properties of any bucket assignment function in Section 4.1. We then identify an optimal bucket assignment function in Section 4.2 and provide an intuitive reason for its optimality. Proofs can be found in the full version [17].

4.1 Bucket Assignment Functions

Let $w_1, \ldots, w_B$ be the probability that a key hashes to bucket i for $i \in [B]$. We may assume without loss of generality that these probabilities are given in decreasing order. An equivalent view considers the prefix sums $\sigma_i := w_1 + \cdots + w_i$. A key with (normalized) hash value $x \in (0, 1]$ is then assigned to bucket i if $x \in (\sigma_{i-1}, \sigma_i]$.

We can conveniently represent this information using a *bucket assignment function* $\gamma : [0, 1] \to [0, 1]$ that: interpolates the points $\{(\sigma_i, i/B) \mid 0 \leq i \leq B\}$, is increasing and smooth on $(0, 1)$, and has non-decreasing derivative. The bucket assigned to hash value $x \in (0, 1]$ is then $\lceil \gamma(x) \cdot B \rceil$. It is a non-trivial insight of this section that a single bucket assignment function (not depending on B and n) can result in good construction times for many values of B and n simultaneously.



(a) The bucket assignment functions map a normalized hash value x to a normalized bucket index $\gamma(x)$.

(b) The expected bucket sizes relative to the average size λ are $(\gamma^{-1})'(b)$ for normalized bucket index b.

Figure 2 Comparison of bucket assignment functions $\gamma(x)$ of related work and PHOBIC ($\gamma = \beta_*$).

From now on, let $\lambda := n/B$. We summarize some useful intuitions about bucket assignment functions. These intuitions are valid for large n and B (when γ, γ^{-1}, and γ' are approximately constant on intervals of length $\frac{1}{n}$ and $\frac{1}{B}$). For now, we neglect edge cases related to γ or γ^{-1} not being smooth at 0 or not being smooth at 1 (but just smooth on $(0,1)$).

▶ **Intuition 3.** Let $x \in (0,1]$ be a normalized hash and $b = \gamma(x)$ the normalized bucket index of the bucket assigned to x. Then:

- **(i)** The expected size of the bucket assigned to x is $\lambda/\gamma'(x)$.
  (Reason: In the vicinity of x and for infinitesimal δ, a δ-fraction of the hash range (used by δn keys in expectation) is shared by a $(\gamma'(x)\cdot\delta)$-fraction of the B buckets. The quotient is $\delta n/(\gamma'(x)\delta B) = \lambda/\gamma'(x)$.)
- **(ii)** The expected size of the bucket with normalized index b is $\lambda/\gamma'(\gamma^{-1}(b)) = \lambda(\gamma^{-1})'(b)$.
  (Follows from **(i)** *and the inverse function rule.)*
- **(iii)** The expected size of a bucket is decreasing in its normalized index.
  (Follows from **(ii)** *and monotonicity of γ' and γ^{-1}.)*
- **(iv)** The expected fraction of keys with normalized hash in $(0,x]$ is x.
- **(v)** If $\mu > 0$ and $x_\mu \in (0,1)$ is such that $\lambda/\gamma'(x_\mu) = \mu$ then the expected fraction of keys in buckets of size at least μ is x_μ. *(Follows from* **(i)**, **(iii)** *and* **(iv)**.*)*

4.2 An Optimal Bucket Assignment Function

Intuitively, we identify the following bucket assignment function to be optimal, although our precise result stated below is more subtle.

$$\beta_*(x) = x + (1-x)\cdot\ln(1-x) \text{ with derivative } \beta_*'(x) = -\ln(1-x).$$

For comparison, Figure 2a shows β_* as well as the bucket assignment functions used by CHD and PTHash. In Figure 2b we see the distribution of expected bucket sizes, which is uniform for CHD, imbalanced for PTHash, and even more imbalanced for β_*.

Recall that $\lambda = n/B$ is the average bucket size. By Proposition 2 a lower bound for the expected work is $\Omega(n\cdot e^\lambda/\lambda)$. We prove, firstly, that any bucket assignment function γ that differs from β_* leads to expected work exceeding $n\cdot e^{(1+\varepsilon)\lambda}$ for some $\varepsilon = \varepsilon(\gamma)$, provided that λ is large enough. Conversely, we show that a slight perturbation β_ε of β_* leads to an expected work of essentially at most $n\cdot e^{(1+\varepsilon)\lambda}$ for any $\varepsilon > 0$, provided that $\lambda \geq \lambda_0(\varepsilon)$ is large enough. Essentially, we can get arbitrarily close to a cost of e^λ per key and only functions close to β_* can achieve this.

Our results bound the work $w_{n,\lambda}(\gamma)$ associated with γ and involve a "coupon collector term" w_{coupon}, which is the work required to place buckets of size 1.[1] We will define these more precisely below. Proofs are found in the full version [17]. We have reason to believe that our results generalize for the non-minimal case of $m > n$, as explained in the full version [17].

▶ **Theorem 4.** *Let $\gamma : [0,1] \to [0,1]$ be a continuous bucket assignment function that is smooth on $(0,1)$ with non-decreasing derivative. If $\beta_* \neq \gamma$ then*

$$\exists \varepsilon > 0 : \forall \lambda \geq \lambda_0(\varepsilon) : \forall n \geq n_0(\lambda, \varepsilon) : w_{n,\lambda}(\gamma) \geq n \cdot e^{\lambda(1+\varepsilon)} + w_{\text{coupon}} \text{ whp.}$$

While this leaves the relationship between γ and $\varepsilon(\gamma)$ open, our proof suggests that any $\varepsilon < \sup_{x \in (0,1)} \frac{\beta_*'(x)}{\gamma'(x)} - 1$ is a possible choice.

▶ **Theorem 5.** *Let $\beta_\varepsilon(x) := \varepsilon x + (1-\varepsilon)\beta_*(x)$ for some $\varepsilon > 0$. Then*

$$\forall \varepsilon > 0 : \forall \lambda \geq \lambda_0(\varepsilon) : \forall n \geq n_0(\lambda, \varepsilon) : w_{n,\lambda}(\beta_\varepsilon) \leq n \cdot e^{\lambda(1+\mathcal{O}(\varepsilon))} + w_{\text{coupon}} \text{ whp.}$$

By *with high probability* (whp) we mean probability $1 - \mathcal{O}(n^{-c})$ for some $c > 0$. Note that both theorems are phrased such that we may assume that n is much larger than λ and λ is much larger than $1/\varepsilon$. We give implementation details concerning the use of β_ε in Section 4.3.

The work associated with a bucket assignment function. To place a bucket of size $s \in \mathbb{N}$ into a hash table of size n that already has load factor $\alpha \in [0, 1 - \frac{s}{n}]$ we repeatedly try seeds for a hash function mapping keys to $[n]$, until all keys hash to free positions. The expected cost $c_n(s, \alpha)$ associated with this task under the simple uniform hashing assumption is described precisely in the full version [17]. We have to take into account self-collisions, i.e. while checking the keys one after the other, the load factor gradually increases and is $\alpha' = \alpha + \frac{s-1}{n}$ for the last key. For our purposes, the following bounds on $c_n(s, \alpha)$ suffice

$$(1-\alpha)^{-s} \leq c_n(s, \alpha) \leq s \cdot (1-\alpha')^{-s}. \tag{1}$$

This uses that $(1-\alpha)^{-s}$ and $(1-\alpha')^{-s}$ are lower and upper bounds on the number of seeds that have to be tried and that, to test a seed, at least 1 and at most s keys have to be considered. Now assume we are given a bucket assignment function γ as well as $n \in \mathbb{N}$, $\lambda \in \mathbb{R}_+$ and $B = n/\lambda$. By assigning keys to buckets according to γ and hash values in $(0,1]$ we obtain buckets. Let $s_1 \geq \ldots \geq s_B$ be their sizes in decreasing order. As we show in the full version of this paper [17] it is advantageous to process the buckets in this order. Defining $\alpha_i := \frac{1}{n} \sum_{j=1}^{i-1} s_i$, the total cost is then $w_{n,\lambda}(\gamma) = \sum_{i=1}^{B} c_n(s_i, \alpha_i)$.

Note that while this describes the *expected* cost when *given* $(s_i)_{i \in [B]}$, overall $w_{n,\lambda}(\gamma)$ is still a random variable because the numbers $(s_i)_{i \in [B]}$ are random. Assume now that there are exactly k buckets of size 1 that are placed last (we may ignore buckets of size 0). For these buckets, the upper and lower bounds in Equation (1) coincide so they incur a cost of

$$w_{\text{coupon}} := \sum_{i=1}^{k} c(1, \tfrac{n-i}{n}) = \sum_{i=1}^{k} \frac{n}{i} = n \cdot H_k.$$

[1] The name is inspired by the famous coupon collector problem. Given an urn with n balls (or coupons), the collector repeatedly draws a uniformly random ball from the urn and puts it back. The question is how many draws are needed in expectation until each ball was drawn at least once. When placing buckets of size 1, we are similarly just waiting for unused hash values to appear, when drawing them one by one.

Here H_k is the kth harmonic number, which satisfies $H_k = \Theta(\log k)$. If n is sufficiently large compared to λ then we have $k \geq n^d$ whp for some constant $d > 0$, giving a cost of $\Theta(n \cdot H_{n^d}) = \Theta(n \log n)$. This dominates overall construction time if n is sufficiently large compared to λ. Our theorems list this work for buckets of size 1 as a separate term because there are techniques to mitigate the problem: The hash function may permit to directly compute for a given key x and table position i a seed for which x is mapped to i. This is the case if the seed includes an additive displacement term, as is the case in our implementation and in FCH [13].

Intuition: What makes β_* uniquely promising. For $\mu > 0$ let $x_\mu \in (0, 1)$ be such that $\lambda/\beta'_*(x_\mu) = \mu$. By Intuition 3 (v), roughly an expected x_μ-fraction of the keys (those with hashes in $[0, x_\mu]$) land in buckets of expected size at least μ. Assume for now that a bucket of expected size μ has actual size μ (ignoring the issue that μ may not be integer). Then, since we process buckets in order of increasing size, we would process a bucket of size μ when the load factor is x_μ. The expected cost for this is, by Equation (1), around $(1-x_\mu)^{-\mu}$. Using that $\beta'_*(x) = -\log(1-x)$ gives $\mu = -\lambda/\log(1-x)$ and hence $(1-x_\mu)^{-\mu} = (1-x_\mu)^{\lambda/\log(1-x_\mu)} = e^\lambda$, i.e. a cost independent of μ. The idea behind Theorem 4 is that any bucket assignment function $\gamma \neq \beta_*$ fails to balance bucket sizes in this way, leading to significantly higher costs.

The simplification we made seems adequate for large μ since a bucket of large expected size typically has actual size close to its expectation. But consider $\mu = 1.5$ and cease to ignore rounding issues. If at load factor x_μ we would process buckets of size 1 half the time and buckets of size 2 half the time, the resulting costs are around $e^{\frac{2}{3}\lambda}$ and $e^{\frac{4}{3}\lambda}$, respectively, which does not average out to e^λ. Luckily, things are more complicated. It turns out that for small $s \in \mathbb{N}$ and assuming large λ, the *expected number* of buckets of size s is meaningfully greater than the number of buckets of *expected size* in the range $[s-1, s+1]$. This means that we get more small buckets than we seem to have called for, decreasing costs at high load factors. It seems clear that the flipside of this beneficial effect must be a detrimental effect for larger bucket sizes that a proof of Theorem 5 must quantify. When it comes to *very* large bucket sizes, we bail ourselves out by using β_ε instead of β_*: since β'_ε is lower bounded by ε, the expected bucket sizes are capped at λ/ε. It is buckets of intermediate sizes that have to pay the price.

4.3 Details on the Bucket Assignment in Practice

Our implementation of bucket assignment functions differs from our theoretical results in two minor ways.

Perturbation. Recall that $\beta_* : [0, 1] \to [0, 1]$ was, modulo certain qualifications, identified as the optimal bucket assignment function in Section 4. One of the qualifications was that we actually analyze a slightly perturbed function $\beta_\varepsilon(x) := \varepsilon x + (1 - \varepsilon)\beta_*(x)$. This limits expected bucket sizes to λ/ε, which helped with bounding construction times of large buckets. Limiting the bucket sizes is also useful in practice to reduce self-collisions inside the small partitions. Concretely we choose $\varepsilon = \frac{\lambda}{5\sqrt{\rho}}$, for an expected partition size of ρ. Thus, capping the expected bucket sizes at $5\sqrt{\rho}$. Without this perturbation ($\varepsilon = 0$), running times are noticeably worse.

Tabulating values. The functions β_* and γ_P involve expensive arithmetic operations such as a logarithm. We achieve a significant speedup by tabulating $\gamma_P(x)$ for 2048 discrete values of x and interpolating linearly.

5 Fine-Grained Partitioning

Any PHF construction can trivially be parallelized by hashing the keys into subsets of expected equal size and building a PHF for each subset in parallel. The various PHFs are then logically "concatenated" into a single PHF taking the prefix sum of the partition sizes. The respective offsets have to be looked up when querying a key, imposing some query time overhead. Partitioning is widely used for a variety of construction techniques. It was also used by PTHash in the PTHash-HEM variant [28]. In this paper, we use partitions that are several magnitudes smaller than the ones used in PTHash-HEM. In itself, reducing the partition size results in only marginal construction time improvements. However, small partitions enable a new, more efficient encoding scheme which we introduce in Section 5.1. Additionally, they enable a fast GPU parallelization which we describe in Section 5.2.

Encoding the offsets of the partitions. The offsets and sizes of the many partitions require a non-negligible amount of space. We mitigate this without incurring too much query overhead by storing sizes only implicitly as the difference of two subsequent offsets. Offsets are stored as the difference to their expectation.

Hash Function. The original PTHash hash function works by XOR-ing the hash value of the key with the hash value of the seed. Although the technique works well on large-enough partition sizes, it might fail for small sizes because of correlations in the hash values. Given the small partition sizes we use here, we have to rely on a different technique. We use the seed value p to store two numbers, namely $p = s \cdot m + d$, where m is the actual partition size and $d \in [m]$ is an additive displacement. The position of a key x is $(h(x, s) + d) \bmod m$. While searching for a seed, we calculate $h(x, s)$ for all keys of the bucket and then only have to increment the values to obtain the positions for the next seeds. If no position is found within $d \in [m]$ we continue by incrementing s, re-calculating $h(x, s)$ and setting $d = 0$. At query time, we have to calculate the position of a key x using seed p. Note that we have $d = p \bmod m$, so we can compute the position of the key x as $(h(x, \lfloor p/m \rfloor) + p) \bmod m$.

5.1 Interleaved Coding of Seeds

Once the search has finished, the seeds found for each bucket have to be stored in some compressed manner. Ideally, the seeds should be encoded such that they require little space and are quickly accessible during querying. We mainly use *Compact* and *Golomb-Rice* encoding as building blocks for our new technique. Compact encoding is also used in the original PTHash implementation. In Compact encoding, all values are stored consecutively by concatenating their binary representation. All values use the same bit length, allowing for quick access. The bit length is chosen such that the highest seed can be accommodated. Golomb-Rice [15, 30] encoding stores the b least significant bits of each seed using compact encoding. The most significant bits are stored in unary representation. A selection structure enables access to the unary part of the seeds in constant time. We apply the formula by Kiely [18] to select b. A straightforward approach would be to encode all seeds using a single encoder. However, the seeds do not follow the same statistical distributions across different buckets, hence using the same encoder for all buckets is suboptimal. It is instead beneficial to group seeds which follow the same distribution and encode them using the same encoder. PTHash does this only partially by using two encoders – one for each expected bucket size (the so-called "front-back" compression [27]).

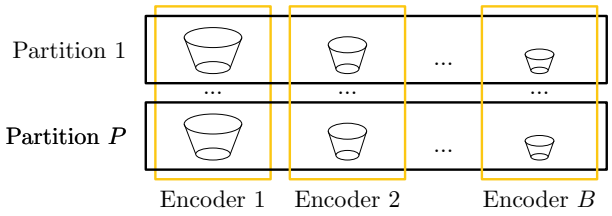


Figure 3 Interleaved coding. Encoder i stores the seed of bucket i from all partitions.

We now introduce our new technique. For each partition we hash to the same number of buckets $B := \lceil \rho/\lambda \rceil$, based on the average partition size ρ and average bucket size λ. The i-th bucket of a partition has the same expected size and the corresponding seed follows the same statistical distribution as the i-th bucket of any other partition. Although the idea of our optimized bucket assignment function is to give all buckets the same seed distribution, this is not achievable in practice. At least one reason for this are the discrete bucket sizes. This results in discrete jumps in the probability that a seed is found when processing one bucket after another. In *interleaved coding* we therefore employ B encoders and the i-th encoder stores the seeds of the i-th buckets of all partitions. Each encoder is tuned for its specific distribution (e.g., different Rice parameters). Figure 3 illustrates interleaved coding.

It is also possible to mix different encoding techniques, similar to what PTHash does. Larger buckets are accessed more often than smaller buckets because they contain more keys. Hence, it is beneficial to use an encoding technique which is optimized for fast lookup time (e.g., Compact) for the larger buckets. Conversely, the encoding for the seeds belonging to smaller buckets should be tuned for space efficiency (e.g., Golomb-Rice). To conclude this section, we point out that each of the B encoders introduces some metadata overhead (e.g., for storing its parameters). Using rather small partition sizes ρ decreases the number B of encoders and therefore the constant overhead.

5.2 GPU Parallelization

We provide a GPU implementation for even faster construction. On a GPU, each *workgroup* executes independently, typically with its own subset of data. Within each workgroup, individual *threads* execute concurrently. Threads within the same workgroup can share data and synchronize with each other through mechanisms like barriers and shared memory. Thread level parallelism allows for fine-grained parallel execution of instructions within a workgroup. However, only threads which follow the same control path and thus execute the same instruction at the same time can be executed in parallel. It is therefore crucial to avoid divergent control paths. As a first step, our parallel implementation transfers the keys to the GPU, before partitioning them. Afterwards, we sort the buckets and start the search.

Search. Our fine-grained partitioning naturally maps to the architecture of a GPU. Each partition is processed by one workgroup. The small partition sizes enable performing the search entirely using fast but small shared memory. Bucket by bucket, all threads of the workgroup cooperate to quickly find the smallest working seed. A CPU implementation would usually do so using a nested loop. The outer loop would iterate over seed values and the inner loop over keys. If a collision occurs, it would immediately leave the inner loop and continue with the first key and the next seed. However, on a GPU, leaving the inner loop would result in divergence because threads might encounter a collision after a different number of keys. This is illustrated in the left of Figure 4.

Algorithm 2 Seed search for one bucket.

```
shared sFound ← ∞
shared sNext ← threadCount
seed ← threadId, keyIndex ← 0
while sFound = ∞
    isCollision ← COLL(seed, keyIndex)
    keyIndex ← keyIndex + 1
    if isCollision
        keyIndex ← 0
        seed ← ATOMADD(sNext, 1)
    else if keyIndex = bucketSize
        sFound ← ATOMMIN(sFound, seed)
```

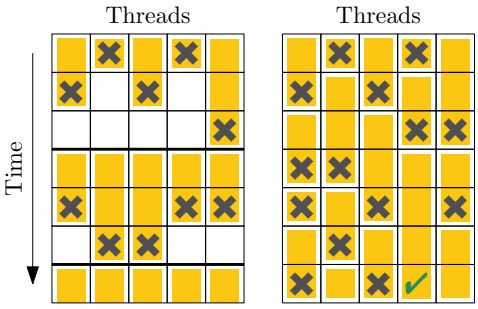


Figure 4 Each box represents one seed tested by one thread. Left: Synchronized nested loop. Right: Algorithm 2 where we continue with the first key and the next seed after a collision.

Instead, we use the technique described in [31] to emulate the behavior of the nested loop using a single loop to reduce divergence. Hence, our GPU implementation parallelizes over partitions, seeds and keys. The inner loop is emulated by incrementing the key index in each iteration. If a collision occurs, we reset the key index and emulate the behavior of the outer loop by atomically incrementing a seed counter which is shared among all threads. If the last key did not collide, we found a working seed. Multiple threads can find a seed in the same iteration. To reduce entropy, we use an atomic minimum to identify the smallest of those seeds. Note that this finds the smallest working seed overall because all threads finding a working seed must have processed each key. Therefore, if there was a smaller working seed, it would have been found in an earlier loop iteration. We give pseudocode in Algorithm 2 and illustrate the behavior on the right in Figure 4. During search, we only access shared memory and perform fast arithmetic operations. Specific optimizations for our additive displacement hash function are not shown here.

6 Experiments

In Section 6.1, we gradually integrate our improvements to show the individual effects. Then we compare our GPU and CPU implementation with the state of the art in Section 6.2. We use a machine with an Intel Core i7-11700 CPU with 64 GiB of DDR4 RAM running Ubuntu 22.04.1. Each core has 48 KiB L1 and 512 KiB L2 data cache. As a GPU, we use an Nvidia RTX 3090 and use Vulkan 1.3.236 to interface with it. We compile using GCC 11.4.0 and compiler options `-march=native` and `-O3`. All benchmarks use random strings of random length between 10 and 50 characters as input which is adopted from previous work [5, 19, 20, 21]. Note that almost all competitors first generate master hash codes of the input. This makes the construction largely independent of the input distribution. We measure the query time by querying each key once in random order. All experiments use $n = 100$ million keys, $\lambda = 8$ and an average partition size of $\rho = 2\,500$ if not stated otherwise. Our source code is public under the General Public License. You can find it through the links on the title page of this paper.

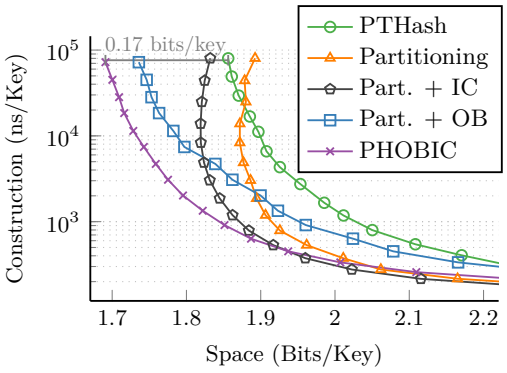


Figure 5 Beginning with **PTHash** (mono-EF, skewed bucketing), we add fine-grained **partitioning** (mono-R, skewed) with additive displacement hashing. We then add interleaved coding (**IC**-R, skewed) and optimized bucketing (mono-R, **OB**) individually. Putting it all together we arrive at **PHOBIC**. All single-threaded.

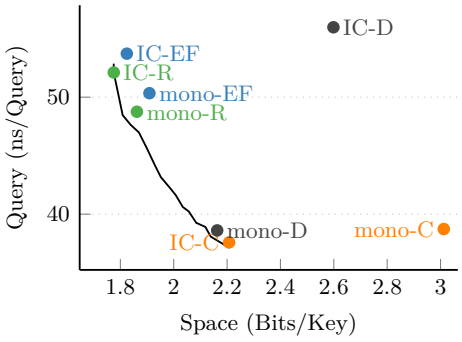


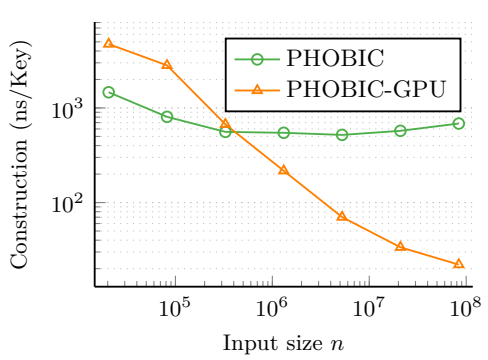
Figure 6 Query time and space consumption of **E**lias-**F**ano, **D**ictionary, **C**ompact, and **R**ice encoders. Points prefixed "mono" place all seeds into a single encoder and those prefixed "IC" use interleaved coding. The curve shows different mixtures (see Section 5) of Compact and Rice encoders. All points use optimized bucketing.

6.1 From PTHash to PHOBIC

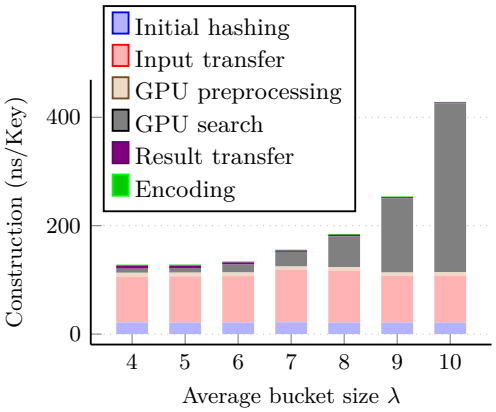
We now gradually introduce our improvements to PTHash. As basic improvements, we replace the initial hash function with xxHash [10] and implement faster parallel partitioning. In all experiments, PTHash contains these changes as well to focus on our algorithmic improvements. Figure 5 gives measurements for the different improvement steps and shows them in different combinations.

Interleaved Coding. Partitioning of PTHash is already used in PTHash-HEM for parallelization [28]. PTHash-HEM uses partitions of size $\approx 10^6$. Smaller partitions only lead to minor improvements, as we show in the full version [17]. However, smaller partition sizes shine when used with our newly introduced interleaved coding (Section 5.1). Interleaved coding uses ρ/λ encoders, where ρ is the expected partition size. Reducing the partition size can significantly reduce constant space overheads, as we also show in the full version [17]. Figure 5 compares the technique of placing all seeds into a single Rice encoder (orange curve) to placing the keys using interleaved Rice coding (black). Interleaved coding consistently improves space efficiency by 0.06 bits/key. Figure 6 compares different combinations of encoders, which were partially used in the original implementation. Interleaved coding allows for mixing of different encoding techniques. If we use this to encode the keys using different numbers of Compact and Rice encoders, we can cover the entire query time to space trade-off in our configuration. Note that the construction performance is similar for all approaches because the encoding is fast compared to the remaining construction.

Optimized Bucket Assignment. In Figure 5 we also show how using the optimized bucket function affects construction speed and space. Our optimization of the bucket assignment function is particularly helpful to construct very space efficient configurations. When compared for same construction time, the optimized function is up to 0.14 bits/key more space efficient relative to the original PTHash bucket assignment.



(a) Construction performance on CPU and GPU for $\lambda = 8.0$ and 8 threads, comparing different n. The GPU requires large n to fully utilize its computing resources.



(b) Different construction steps by values of average bucket size λ.

Figure 7 GPU performance for different input sizes (left) and λ (right).

Further Remarks. With interleaved coding, another improvement originates from the secondary bucket ordering. Primarily the buckets are sorted in non-increasing size. Secondarily sorting in increasing expected size reduces the space consumption by 0.044 bits/key compared to decreasing expected size. The reason for this behavior remains an open problem.

Original PTHash observed significant performance improvements by first calculating a non-minimal PHF and repairing the "gaps" afterwards. Refer to [27] for details. This trick does not result in an improvement when using PHOBIC.

GPU Parallelization. Our final contribution is a GPU implementation to speed up construction. Our implementation parallelizes over partitions, seeds and keys. The GPU implementation is mainly useful for large average bucket sizes λ. This is well illustrated in Figure 7b: For smaller values of λ, the construction time is dominated by the time to transfer the input data to the GPU. We also compare CPU and GPU construction speed for different input sizes in Figure 7a. The GPU requires a large number of input keys and thus a large number of partitions before its computing resources are fully utilized. Overall, the GPU outperforms the CPU for a sufficiently large λ and n. We use the GPU only to accelerate construction, while measuring all queries on the CPU.

6.2 Comparison to Other Methods

We compare our new approach to several other methods from the literature. First and foremost, we compare against the original PTHash [27, 28] implementation. The comparison also includes the fingerprinting approaches BBHash [22], FMPH [4], and FMPHGO [4]. We also compare against RecSplit [12] and approaches based on it, such as SIMDRecSplit [5], ShockHash-RS [21], and bipartite ShockHash-RS [19]. Finally, we also compare against CHD [2] and SicHash [20].

Each method has a wide range of configurations that provide a trade-off between space, construction time, and query time. To give an initial overview, we show a Pareto front for each method in Figure 8. A configuration is on the Pareto front if no other configuration of the same method is simultaneously faster and more space efficient. For this plot we use a single thread (a multithreaded measurement would mainly show what method implemented the partitioning step most efficiently instead of focusing on the algorithmic aspects). The

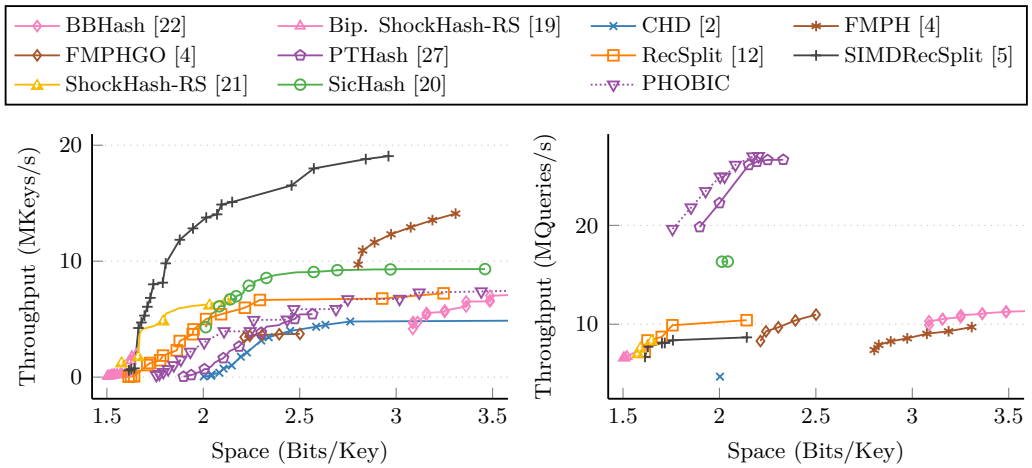


Figure 8 Construction throughput (left) and query throughput (right) for various methods on 100 million keys and using a single processing thread.

figure shows that PTHash and PHOBIC are clear winners in terms of query performance. Even though BBHash [22] and FMPH [4] are also focused on fast queries, they are significantly slower than PTHash and PHOBIC. Figure 5 shows that PHOBIC consistently saves about 0.17 bits/key for a large range of different construction times while maintaining the good query speed. We remark that this is a significant reduction in space considering the proximity to the space lower bound. The competitors achieving even lower space consumption (i.e. RecSplit [12], SIMDRecSplit [5], ShockHash-RS [21], and bipartite ShockHash-RS [19]) all have a rather slow query performance. However, somewhat surprisingly, SIMDRecSplit has the fastest construction even for less space efficient configurations. SicHash [20] takes a middle ground with faster construction than PHOBIC and query performance between PHOBIC and the RecSplit variants.

Table 1 gives a selection of configuration parameters for direct comparison, mostly taken from the corresponding papers. The full version of this paper [17] gives the same table measured on a large machine with 64 threads. Comparing configurations with the same space consumption, PHOBIC is significantly faster to construct than the original PTHash implementation. Comparing configurations that both need 1.86 bits/key and have a similar query time, PHOBIC can be constructed 83 times faster than PTHash.

On the GPU, we compare against the only available GPU construction, RecSplit-GPU [5]. We give plots in the full version of this paper [17]. Basically, we achieve the same peak construction throughput as RecSplit-GPU for the less space efficient configurations. The queries of both approaches are done on the CPU, so the fact that PHOBIC offers much faster queries applies here as well (see Figure 8). Comparing the multithreaded CPU implementation and the GPU implementation of PHOBIC, we get a construction speedup of 62 for $\lambda = 9$ with interleaved Rice coding. Note that with $\lambda = 9$, the GPU still spends a lot of its construction time on transferring the input data (see Figure 7b), but much larger values of λ are not feasible on the CPU.

Directly comparing the performance of CPU and GPU is always difficult because of the different hardware architectures. Given that the power consumption is a major cost factor in production environments, we measure it using a Voltcraft Energy Check 3000 wattmeter. For CPU-only measurements, we dismount the GPU. The machine requires about 405 W during the search step of the GPU version and 195 W for the multithreaded CPU implementation.

Table 1 Performance of various methods on 100 million keys.

Method	Space	Query	Construction (ns/key)		
	(bits/key)	(ns/query)	1 Thread	8 Threads	Speedup
Bip. SH-RS, n=64, b=2000	1.52	160	5 756	1 218	4.7
CHD, λ=3	2.27	222	352	-	-
CHD, λ=5	2.07	207	2 206	-	-
FMPH, γ=2.0	3.40	100	69	17	4.0
FMPH, γ=1.0	2.80	134	99	24	4.0
SIMDRecSplit, n=8, b=100	1.81	124	109	20	5.2
SIMDRecSplit, n=14, b=2000	1.58	143	11 062	2 360	4.7
SicHash, α=0.9, p_1=21, p_2=78	2.41	72	129	25	5.0
SicHash, α=0.97, p_1=45, p_2=31	2.08	64	179	32	5.6
PTHash, λ=4.0, α=0.99, C-C	3.19	35	314	143	2.2
PTHash, λ=5.0, α=0.99, EF	2.11	54	525	252	2.1
PTHash, λ=10.5, α=0.99, EF	1.86	49	82 721	35 048	2.4
PTHash-HEM, λ=4.0, α=0.99, C-C	3.19	39	299	45	6.6
PTHash-HEM, λ=5.0, α=0.99, EF	2.11	58	582	86	6.7
PHOBIC, λ=3.9, α=1.0, IC-C	3.18	40	197	32	6.2
PHOBIC, λ=4.5, α=1.0, IC-R	2.11	57	254	40	6.2
PHOBIC, λ=6.5, α=1.0, IC-R	1.85	52	992	176	5.6
PHOBIC, λ=9.0, α=1.0, IC-R	1.74	50	9 171	1 781	5.1
				GPU + 8 CPU Threads	
PHOBIC-GPU, λ=9.0, IC-C	2.17	37		28	
PHOBIC-GPU, λ=9.0, IC-R	1.76	52		27	
PHOBIC-GPU, λ=13.0, IC-R	1.68	50		560	
PHOBIC-GPU, λ=14.0, IC-R	1.67	49		1 470	
RecSplit-GPU, ℓ=8, b=100	1.81	126		24	
RecSplit-GPU, ℓ=14, b=2000	1.58	147		80	
RecSplit-GPU, ℓ=18, b=2000	1.55	135		1 732	

Thus, the above speedup of 62 translates to roughly 30 times lower energy consumption for constructing an MPHF on the GPU. Single-threaded CPU construction requires 74 W which is less energy efficient compared to multithreading.

7 Conclusion and Future Work

PHOBIC introduces optimized bucket sizes and interleaved encoding to PTHash. Our improvements result in 0.17 bits/key better space efficiency when compared to PTHash for similar construction and query speed. When compared for the same space consumption, PHOBIC can be constructed up to 83 times faster than PTHash, while still having the same query time. Finally, our GPU implementation improves the construction by a factor of up to 62 compared to the multithreaded CPU implementation.

Future work may explore combinations of time efficient approaches to perfect hashing and space efficient approaches. Concretely, we are hopeful that a hybrid between PHOBIC and ShockHash [21] puts further trade-offs between space and time into reach.

References

1 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *ESA (1)*, volume 6346 of *Lecture Notes in Computer Science*, pages 427–438. Springer, 2010. doi:10.1007/978-3-642-15775-2_37.

2 Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *ESA*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer, 2009. doi:10.1007/978-3-642-04128-0_61.

3 Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Trans. Algorithms*, 10(4):23:1–23:19, 2014. doi:10.1145/2635816.

4 Piotr Beling. Fingerprinting-based minimal perfect hashing revisited. *ACM J. Exp. Algorithmics*, 28:1.4:1–1.4:16, 2023. doi:10.1145/3596453.

5 Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. High performance construction of recsplit based minimal perfect hash functions. In *ESA*, volume 274 of *LIPIcs*, pages 19:1–19:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ESA.2023.19.

6 Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *WADS*, volume 4619 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2007. doi:10.1007/978-3-540-73951-7_13.

7 Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of Bloom filters: A survey. *Internet Math.*, 1(4):485–509, 2003. doi:10.1080/15427951.2004.10129096.

8 Chin-Chen Chang and Chih-Yang Lin. Perfect hashing schemes for mining association rules. *Comput. J.*, 48(2):168–179, 2005. doi:10.1093/COMJNL/BXH074.

9 Jarrod A. Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P. Schroth, and Daniel S. Rokhsar. Meraculous: De novo genome assembly with short paired-end reads. *PLOS ONE*, 6(8):1–13, August 2011. doi:10.1371/journal.pone.0023501.

10 Yann Collet. xxHash: Extremely fast non-cryptographic hash algorithm. URL: https://github.com/Cyan4973/xxHash.

11 Victoria G. Crawford, Alan Kuhnle, Christina Boucher, Rayan Chikhi, and Travis Gagie. Practical dynamic de bruijn graphs. *Bioinform.*, 34(24):4189–4195, 2018. doi:10.1093/BIOINFORMATICS/BTY500.

12 Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. RecSplit: Minimal perfect hashing via recursive splitting. In *ALENEX*, pages 175–185. SIAM, 2020. doi:10.1137/1.9781611976007.14.

13 Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *SIGIR*, pages 266–273. ACM, 1992. doi:10.1145/133160.133209.

14 Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In *SEA*, volume 9685 of *Lecture Notes in Computer Science*, pages 339–352. Springer, 2016. doi:10.1007/978-3-319-38851-9_23.

15 Solomon W. Golomb. Run-length encodings (corresp.). *IEEE Trans. Inf. Theory*, 12(3):399–401, 1966. doi:10.1109/TIT.1966.1053907.

16 Stefan Hermann. Accelerating minimal perfect hash function construction using gpu parallelization. Master's thesis, Karlsruhe Institute for Technology (KIT), 2023. doi:10.5445/IR/1000164413.

17 Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer. PHOBIC: perfect hashing with optimized bucket sizes and interleaved coding. *CoRR*, abs/2404.18497, 2024. doi:10.48550/arXiv.2404.18497.

18 Aaron Kiely. Selecting the Golomb parameter in Rice coding. *IPN progress report*, 42:159, 2004.

19 Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. Shockhash: Near optimal-space minimal perfect hashing beyond brute-force (extended version). *CoRR*, abs/2310.14959, 2023. doi:10.48550/arXiv.2310.14959.

20 Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. SicHash – small irregular cuckoo tables for perfect hashing. In *ALENEX*, pages 176–189. SIAM, 2023. doi:10.1137/1.9781611977561.CH15.

21 Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. Shockhash: Towards optimal-space minimal perfect hashing beyond brute-force. In *ALENEX*. SIAM, 2024. doi:10.1137/1.9781611977929.15.

22 Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. In *SEA*, volume 75 of *LIPIcs*, pages 25:1–25:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.SEA.2017.25.

23 Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *ISIT*, pages 2774–2778. IEEE, 2006. doi:10.1109/ISIT.2006.261567.

24 Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *FOCS*, pages 170–175. IEEE Computer Society, 1982. doi:10.1109/SFCS.1982.80.

25 Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 138–149. Springer, 2014. doi:10.1007/978-3-319-07959-2_12.

26 Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Supplement_1):i185–i194, 2022.

27 Giulio Ermanno Pibiri and Roberto Trani. Pthash: Revisiting FCH minimal perfect hashing. In *SIGIR*, pages 1339–1348. ACM, 2021. doi:10.1145/3404835.3462849.

28 Giulio Ermanno Pibiri and Roberto Trani. Parallel and external-memory construction of minimal perfect hash functions with pthash. *IEEE Trans. Knowl. Data Eng.*, 36(3):1249–1259, 2024. doi:10.1109/TKDE.2023.3303341.

29 Giulio Ermanno Pibiri and Rossano Venturini. Efficient data structures for massive N-gram datasets. In *SIGIR*, pages 615–624. ACM, 2017. doi:10.1145/3077136.3080798.

30 Robert F Rice. Some practical universal noiseless coding techniques, 1979.

31 Peter Sanders. Emulating MIMD behaviour on SIMD-machines. In *EUROSIM*, pages 313–320. Elsevier, 1994.