High Performance Construction of RecSplit Based Minimal Perfect Hash Functions

Dominik Bez ✉
Karlsruhe Institute of Technology, Germany

Florian Kurpicz ✉ ⓘ
Karlsruhe Institute of Technology, Germany

Hans-Peter Lehmann ✉ ⓘ
Karlsruhe Institute of Technology, Germany

Peter Sanders ✉ ⓘ
Karlsruhe Institute of Technology, Germany

──── **Abstract** ────

A minimal perfect hash function (MPHF) is a bijection from a set of objects S to the first $|S|$ integers. It can be used as a building block in databases and data compression. RecSplit [Esposito et al., ALENEX'20] is currently the most space efficient practical minimal perfect hash function. Its main building blocks are *splittings* and *bijections*. Using a tree-like data structure, RecSplit first splits the input set into small sets of constant size ℓ and then computes a bijection on each leaf. Both splittings and bijections heavily rely on trying multiple hash functions in a brute-force way.

We greatly improve the construction time of RecSplit using two orthogonal approaches. On the one hand, we explore the trade-off between (exponential time) brute force and more informed (polynomial time) search heuristics. *Rotation fitting* hashes the objects in each leaf to two sets and tries to combine them to a bijection by cyclically shifting one set to fill the holes in the other. *ShockHash* constructs a small cuckoo hash table in each leaf, which is overloaded to hold more objects than the asymptotic maximum.

On the other hand, we harness parallelism on the level of bits, vectors, cores, and GPUs. In combination, the resulting improvements yield speedups up to 241 on a CPU and up to 2072 using a GPU. The original RecSplit implementation needs 19 minutes to construct an MPHF for 1 Million objects with 1.56 bits per object. On the GPU, we achieve the same space usage in just 1.5 seconds. Given that the speedups are larger than the increase in energy consumption, our implementation is more energy efficient than the original implementation.

As a result, our improved RecSplit implementation is now the approach to perfect hashing with the fastest construction time over a wide range of space budgets. Surprisingly, this even holds for rather high space budgets where asymptotically faster methods are available.

1 Introduction

A *Perfect Hash Function* (PHF) is a hash function that does not have collisions, i.e., is injective, on a given set S of objects. Evaluating the PHF on any object not in S can return an arbitrary value. A *Minimal Perfect Hash Function* (MPHF) maps the objects in S to the first $|S|$ integers, so it is bijective. MPHFs are useful in many applications, for

example, to implement hash tables with guaranteed constant access time [20]. By storing only fingerprints in the hash function cells [14, 3], we obtain an *approximate membership data structure*. Storing payload data in the cells, we obtain an updatable retrieval data structure [28]. Finally, the perfect hash function values can be used as small identifiers of the input objects [5], which are easier to handle and more space efficient than, for example, strings.

MPHFs can be very compact – the theoretically minimal space usage is 1.44 bits per object [2]. Currently, the most space-efficient practical MPHF is RecSplit [13]. It provides various tradeoffs between the space consumption, construction time, and query time. For example, RecSplit can construct an MPHF with 1.56 bits per object in less than 2 ms per object. This, however, is too slow for large inputs.

In this paper, we provide several improvements inside the RecSplit framework. We first describe RecSplit and other preliminaries in Section 2, and briefly review related work in Section 3. As a core step during construction, RecSplit tries out hash functions on a small set of objects until one hash function is a bijection. We introduce two new bijection search mechanisms in Section 4, which reduce the search space of the brute force algorithm compared to the original method. *Rotation fitting* hashes the objects to two sets and tries to fit one set into the "holes" of the other set by rotating (cyclically shifting) it. As a side-effect, this approach makes good use of bit parallelism. *ShockHash* stores, for each object, a choice between two possible hash functions in a 1-bit retrieval data structure. This choice (when it exists) can be efficiently computed using cuckoo hashing as in SicHash [26]. This greatly reduces the number of necessary brute force trials.

We then parallelize RecSplit (with and without rotation fitting) using the vector parallelism available with *Single Instruction Multiple Data* (SIMD) instructions and the thread parallelism available with multicore CPUs and GPUs. Given that hash function construction here is mostly compute bound and can be done in parallel for a huge number of small subproblems, the GPU is an ideal hardware. Utilizing GPUs for evaluating hash functions is known from mining of cryptocurrencies with proof-of-work approach (e.g., Bitcoin). Our extensive evaluation in Section 6 shows speedups of up to 50 using SIMD, 241 using multi threading, and 2072 using a GPU. Because GPUs are so much faster at constructing MPHFs, they lead to a better energy efficiency than the CPU, as we show in the experiments. Finally, in Section 7, we summarize the results and give directions for future research.

Our Contributions.   We introduce two new methods for searching for bijections that can be used in RecSplit: *Rotation fitting* and the use of retrieval data structures. We significantly accelerate the construction by four kinds of parallelism (bits, vectors, multicores and GPU). This accelerates RecSplit constructions by a factor up to 2072 and even makes its construction performance competitive to significantly less space efficient minimal perfect hash functions.

2   Preliminaries

In Section 2.1, we first shortly describe basic techniques needed by our implementation. We then continue with describing RecSplit in detail in Section 2.2. Finally, we describe SIMD in Section 2.3 and GPUs in Section 2.4.

2.1   Basics

Words and Bit Vectors.   An important operation in RecSplit is `popcount`, which returns the number of bits in a word that are set to 1. Given a bit vector, the $select_1(x)$ operation
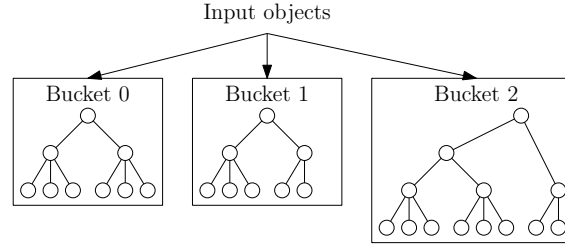
Figure 1 Illustration of the overall RecSplit data structure.

returns the position of the x-th 1-bit in the vector. The operation can be executed in constant time [9] and has very fast implementations [24]. An additional operation we need in this paper is $\text{rot}_k^i(x)$ which rotates (i.e., cyclically shifts) the k least significant bits of x by i bit positions. This can be implemented in a bit parallel way using shifting and masking.

Golomb-Rice. Golomb-Rice codes are space-efficient variable-length codes that allow for storing arbitrarily large numbers. Golomb codes [21] are optimal for geometric distributions, and Golomb-Rice codes [35] are a faster special case, which is almost as space efficient. Given a parameter τ and the number x to store, the τ least significant bits of x are stored directly and the remaining most significant bits of x are encoded in unary. The first part is called *fixed part* and the second is called *unary part*. The unary part consists of $\lfloor x/2^\tau \rfloor$ 0-bits and a final 1-bit. To access one element, we can get the lower bits from the array of fixed parts and the upper bits through two $select_1$ queries.

Elias-Fano. An Elias-Fano representation [12, 15] can be used to store a monotonic sequence of integers. Similar to Golomb-Rice codes, the least significant bits of each value are stored directly in the lower-bits array and can be accessed directly. Using this representation, a monotonic sequence $p_1, \dots, p_k$ with $p_k \leq U$ can be stored using $k(2 + \log(U/k))$ bits. The remaining most significant bits u at index i are encoded as a 1-bit in a bit vector at position $i + u$. This means that by executing a $select_1$ query on the upper bits and looking up the lower bits, we can restore any value in constant time.

Prefix Sums. Let x_i for $i \in [k]$ be a sequence of k numbers. We define the inclusive prefix sum as $\hat{X}_j = \sum_{i=0}^{j} x_i$ and the exclusive prefix sum as $\hat{X}_j = \sum_{i=0}^{j-1} x_i$. This means for each j, they indicate the sum of all previous elements (the prefix). Because prefix sums are monotonic, they can be stored with Elias-Fano coding.

2.2 RecSplit

We now describe RecSplit [13], the MPHF that this paper is based on. Before diving into the details of the most important steps, *splittings* and *bijections*, we give a broad overview over the data structure. The first step of the construction is to apply an initial hash function on every object of the input to generate objects of uniform distribution. These objects are mapped to different buckets of expected size b, where b is a tuning parameter. For each bucket, RecSplit constructs a *splitting tree*, which we describe in more detail later. Each inner node of the tree uses brute force to search for a hash function that distributes the objects to the child nodes such that the children have a specific size. The leaf nodes of the tree (except possibly the last) then have exactly ℓ objects, where a bijection is searched also

using brute force. The hash function indices for splittings and bijections, as well as prefix sums of bucket sizes and bucket encoding lengths are stored in compressed form. Figure 1 illustrates the overall data structure.

The combination of brute force splitting and bijections is highly space efficient from an information-theoretical point of view – disregarding overheads due to encoding and metadata, optimal space consumption can be achieved. Consequently, as the leaf size ℓ gets larger, optimal space is approached [13].

A RecSplit hash function can be evaluated by first applying the initial hash function on the object. Then the bucket is determined and the encoding of the bucket is located. The splitting tree in the bucket is traversed from the root to a leaf by applying the splittings stored at each node. The number of objects in earlier buckets and for each splitting the number of objects left to the current node are accumulated. This value is added to the result of applying the bijection in the leaf.

Splitting Trees.   In each bucket, RecSplit constructs an independent splitting tree. The tree partitions the objects into smaller and smaller sets until the individual sets are small enough such that a bijection on them can be found in reasonable time. At each inner node, RecSplit tries random hash functions to find one that distributes the objects to the child nodes such that each child node gets a specific number of objects. The number of child nodes of an inner node is called fanout. The fanout is optimized in such a way that the expected amount of work to find the splitting is roughly equal to the amount of work in all children combined. The fanouts of the two bottom-most levels are $\max\{2, \lceil 0.35\ell + 0.55 \rceil\}$ and $\max\{2, \lceil 0.21\ell + 0.9 \rceil\}$. In the terminology of the RecSplit paper, these levels are called *lower aggregation levels*. The levels above, also called *upper aggregation levels*, simply use a fanout of 2. The splitting tree has a well-defined shape, depending only on the leaf size ℓ and the number of objects in the bucket.

Bijections.   The lowest level of the splitting tree is called leaf level. Each leaf, except for possibly the last, contains ℓ objects. On all of these small sets of objects, RecSplit can try random hash functions until one of the functions happens to map the m objects bijectively to the first m integers. While the technique is impractical for large sets, the splitting tree ensures that the sets are small enough. A larger leaf size leads to a longer construction time since searching for bijections is expensive for large leaves. On the flip side, larger ℓ make the data structure more space efficient and faster to query. The inner loop of the bijection search applies a hash function modulo ℓ on each object. It converts the value to a bit by taking two to the power of it, and sets the corresponding bit in a bit vector of length ℓ using a logical OR operation. After hashing all objects, if the resulting bit vector has all its bits set to 1, it means that the hash function is a bijection on the leaf. If it is not, RecSplit tries the next hash function. To speed up the case of unsuccessful trials, RecSplit uses a *bijection midstop*. *Bijection midstop* is a way to immediately check for a collision, allowing to possibly skip hashing all objects when earlier objects already have a collision. When the probability of a collision is about 90%, RecSplit executes a `popcount` instruction on the bit vector. If the number of bits does not match the number of objects already hashed, it can already stop and continue trying the next hash function.

Representation.   The shapes of the splitting trees are not stored explicitly. Only the hash function identifiers are stored in preorder, i.e., first the root, then the whole subtree of the first child, then the whole subtree of the second child and so forth. The numbers are encoded

with Golomb-Rice code (see Section 2.1), where all unary parts and all binary parts of a tree are stored together. The optimal Golomb parameter τ depends on the probability that a given hash function is a valid splitting or bijection, respectively. Because the tree has a well-defined shape, the optimal parameters can be pre-calculated. During a query, the tree needs to be traversed. When traversing into a subtree, the encoding of the children left of it need to be skipped. The number of bits to skip in the fixed parts is known because of the well-defined shape. The number of bits to skip in the unary parts can be determined with a $select_1$ query.

The encodings of all splitting trees from all buckets are concatenated in a single bit vector. To evaluate the hash function at a bucket, we need to know the number of objects in previous buckets (prefix sum) and the position in the bit vector where the encoding of each bucket starts. These are two monotonic sequences that can be stored with Elias-Fano codes (see Section 2.1). Because both sequences are strongly correlated, RecSplit uses a modification of Elias-Fano codes to store both of them together in a more space efficient way.

2.3 SIMD

It is common, especially in perfect hashing, that the same operation needs to be executed on different data. This can be achieved with a simple loop, which means that the corresponding instructions must be decoded by the hardware for every element. This can be improved by using *Single Instruction, Multiple Data* (SIMD) [16]. A single instruction is used to apply the same operation on a *vector* of several elements. This reduces the number of instructions that need to be decoded and allows the hardware to process the whole vector in parallel.

We refer to a single element within a SIMD vector as a *lane*. A lane is a word with a specified number of bits. For example, a vector may contain 16 lanes with 32 bits each, i.e., the vector contains 512 bits overall. It is advisable to use as many lanes as possible to maximize throughput.

SIMD is not restricted to simple operations like addition. It may also provide operations to permute the elements in the vector, load and store non-contiguous data (gather and scatter), or other advanced operations. The exact set of operations depends on the concrete implementation of the SIMD model. The Advanced Vector Extensions (AVX) [22] are extensions to the x86 instruction set which is used by many Intel and AMD processors. AVX-512 [23] extends these operations to 512-bit vectors and adds many new instructions, e.g., for using masks to mask out specific lanes. AVX-512 is divided in many smaller subsets, where each processor may only support some of them. One of these subsets which is useful for our implementation is AVX512VPOPCNTDQ which provides `popcount` on 512-bit vectors with lanes of size 32 and 64 bits. The rot_k^i function that cyclically shifts bits (see Section 2.1) can be implemented in a SIMD parallel way.

2.4 GPUs

Graphics Processing Units (GPUs) are specialized processors initially designed for computer graphics applications. Over the last decades, GPUs evolved to general purpose processors for highly parallelizable tasks. We now describe the hardware and programming interface in the following paragraphs. To provide a grasp of the dimensions of a current GPU, we give metrics of the NVIDIA RTX 3090 [30], which is also used for our experiments (see Section 6).

Compute Hardware. A GPU consists of several streaming multiprocessors (SMs) (RTX 3090: 82). Each SM contains many arithmetic logic units (ALUs) to perform computations

(RTX 3090: 64 integer ALUs). Several threads (RTX 3090: 32) operate in *lock-step*, i.e., they execute the same instruction at the same time. Such a bundle of threads is called *warp*. Threads are masked out for instructions they should not execute. This means that in loops, each thread in a warp has to iterate as many times as the thread with the largest number of iterations. To hide latencies, e.g., for memory access, each SM is oversubscribed with more threads than ALUs, and the GPU schedules the threads efficiently. Multiple warps of threads form a *thread block*. Thread blocks are guaranteed to reside on the same SM, which enables them to cooperate. In particular, they can synchronize, and they have access to the same shared memory.

Memory. The *global memory* is the largest and slowest memory on the GPU (RTX 3090: 24 GB). When multiple threads of a warp access the memory simultaneously, the hardware serves the requests with as few memory transactions as possible. To improve performance, the memory access pattern should lead to as few transactions as possible and these transactions should contain as few unused bytes as possible.

Shared memory is a fast memory placed on each SM. It can be used by the treads within the same thread block. On the RTX 3090, shared memory is part of a unified data cache, where the memory that is not reserved for shared memory is used as the L1 cache. The data in shared memory is partitioned into 32 memory banks, and the i-th 32-bit word is stored in bank i mod 32. When n threads of the same warp simultaneously access different words within the same bank, an n-way bank conflict occurs. The operations now have to be serialized and the whole warp has to wait.

CUDA. An efficient way to develop applications on NVIDIA GPUs is CUDA [31]. Functions which can be executed on the GPU (also often called *device*) are called *kernels*. Each kernel is executed on a *grid* of thread blocks. The grid size and the number of threads per block can be selected by the user. The user can create several *streams* on the host and launch kernels and data transfers into streams. The operations launched into a specific stream are executed in order, but operations in different streams can arbitrarily overlap if no explicit synchronization is done.

3   Related Work

Perfect Hashing is an active area of research [10, 37, 2, 26, 33, 27, 7, 19, 6, 8, 29]. Due to a lack of space, we only describe the most recent and fastest algorithms here. For a more detailed overview of recent methods, refer to Ref. [26]. To the best of our knowledge, there is no technique that constructs MPHFs on the GPU yet. Lefebvre and Hoppe [25] describe the GPU evaluation of MPHFs that were constructed on CPUs.

FiPHa/BBHash. A fast and simple approach to minimal perfect hashing uses fingerprinting and bumping [8, 29, 27]. BBHash [27] is a publicly available parallel implementation. The set S of input objects is hashed using a hash function $h \to \beta n$ for a tuning parameter β. The set S' of objects that have a collision is handled recursively. Consider the bit vector b with $b[i] = 1$ iff $|\{s \in S : h(s) = i\}| = 1$. Then $\mathrm{rank}(h(s))$ defines an MPHF on $S \setminus S'$ where $\mathrm{rank}(h(s))$ counts the number of one-bits in b up to $h(s)$ and can be implemented in constant time [9, 24]. This approach needs at least e bits per object (when $\beta = 1$) and provides efficient queries when about 4 or more bits per object are available (using larger values of β). An advantage is very simple and easily parallelizable construction.

PTHash. PTHash [33] is based on FCH [19] which can be considered a predecessor of the hash-and-displace technique [2]. The objects are first distributed into different buckets using a hash function, but the distribution is not uniform. Specifically, about 60% of the objects are mapped to 30% of the buckets. The buckets are then processed in order of decreasing size. For each bucket, a hash function is searched such that each object can be placed in the output domain without colliding with the objects of an earlier bucket. The hash function identifiers are searched linearly and then stored in compressed form with several possible compression schemes. The proclaimed goal of PTHash is fast query times. Using an appropriate compression scheme, only a single memory access is required to find the hash value, and the remaining operations are simple hash function evaluations and arithmetic. Compared to the original implementation of RecSplit, PTHash consumes 0.5 to 2.5 more bits per object, but has two to four times faster queries and can be constructed in less time.

SicHash. SicHash [26] is based on the simple idea to store the index of the hash function to be used in a retrieval data structure. It can capitalize on recent progress on fast and nearly space optimal retrieval [11]. Computing the right index amounts to constructing a cuckoo hash table [32, 18]. In contrast to brute force methods like PTHash and RecSplit, this can be done in near linear time even on large tables. SicHash refines this basic approach using a mix of several fixed precision retrieval data structures and by using many small(ish) cuckoo hash tables rather than a single large table. Through *overloading*, it can fit more objects into each cuckoo hash table than the asymptotic maximum, exploiting that the tables are small. Roughly, SicHash allows faster construction than PTHash while offering similar query time and space consumption.

4 Improved Bijection Search

The general idea of RecSplit consists of two independent steps, bijections and splittings (see Section 2.2). In this section, we now introduce two new methods for searching for bijections. As a reminder, given m objects, we are looking for a way to quickly find a mapping of the objects to the numbers in $[m]$ without any collisions. The original implementation tries out hash functions using brute force until one of them is a bijection. Let us imagine different bijection methods on a scale between plain brute force and a deterministic, linear-time algorithm. Our new methods provide two more data points on the scale that move away from plain brute force methods.

4.1 ShockHash

Our first technique, ShockHash (small, heavily overloaded cuckoo hash tables), is based on an idea introduced in SicHash [26]. In a (binary) cuckoo hash table [32], each object can be placed at two different positions, determined by two hash functions. When we create a cuckoo hash table of size m and then also insert m objects, the object positions implicitly describe a bijection. We can then use a 1-bit retrieval data structure that maps each object to a bit indicating which of the hash functions was used to place it. The retrieval data structure can be stored with space close to 1 bit per object [11]. A reader familiar with cuckoo hash tables will notice that cuckoo hash tables have a *load threshold* of 50%. This means that the probability of successful construction of a table that is filled more than 50% tends to 0 for $m \to \infty$. As observed in Ref. [26], though, small cuckoo hash tables have not only a higher variance in their load factors, but also a higher median load factor. This means that when we only look at very small cuckoo hash tables, we can get away with filling them

completely, requiring only a small number of retries. ShockHash needs more storage space than the original brute force bijection implementation, but is significantly faster. It reduces the need for undirected, brute force searching significantly and replaces it with the more directed construction of cuckoo hash tables. In order to adapt to the faster bijection search, we also modify the fanout to be 2 for the entire splitting tree.

4.2   Rotation Fitting

We now introduce a second approach for bijection search, which we call *rotation fitting*. The method is mainly based on brute force, but ensures that we need significantly fewer hash function evaluations. From the result of one evaluation, we derive additional candidates that are very fast to compute. Rotation fitting is efficient for finding a bijection on m objects, where $m \leq w$ (the word size of the machine). We randomly distribute the objects into two sets A and B by using a 1-bit hash function. Like in the original RecSplit implementation, given a hash function h, we calculate the hash value of all objects in A and set the respective bits in the word a to 1. The function h may be ruled out as a valid bijection by calculating the `popcount` of a. Analogously, the set B is mapped to the word b. Let us now rotate (i.e., cyclically shift) the bits in b. If we can find a rotation value such that the bits in b fit exactly onto the zeroes in a, we have found a bijection. More formally, this is the case if there is an $r \in [m]$, such that $a|\text{rot}_m^r(b)$ has the m least significant bits all set. To efficiently store r, only every m-th hash function is tried, which means the hash function index is congruent to zero modulo m. This number plus r is stored for each leaf. We can restore r later by calculating modulo m and restore the hash function index by rounding down to the next multiple of m. At query time, a rotation corresponds to an addition modulo m to each object in the set B. Given that all except one leaf have $m = \ell$, we only use rotation fitting for these leaves and let the compiler optimize the modulo operation.

Lookup Tables.   It is possible to avoid trying out all m rotations by using a lookup table t. For all possible values of a, this table contains a rotation parameter $t[a]$ such that $\text{rot}_m^{t[a]}(a)$ is minimal. If a value x can be rotated to get the value y, then $\text{rot}_m^{t[x]}(x) = \text{rot}_m^{t[y]}(y)$. Let $c = 2^m - 1$ be the word where the m least significant bits are set. The value $\hat{b} = b \oplus c$ is b with the m least significant bits flipped. Note that b can fill the holes in a if and only if $\hat{b}$ can be rotated to match a. Thus, the necessary rotation of b can be calculated as $r = (t[\hat{b}] - t[a]) \mod m$ using two table lookups. Rotation r is valid if $a|\text{rot}_m^r(b) = c$.

Because rotation is a very cheap operation, preliminary experiments show no improvement by lookup tables. Especially in the case of GPUs, the shared memory is a scarce resource and the global memory is too slow. Our implementation therefore does not use lookup tables, even though we find the idea to normalize random permutations like this an interesting and novel concept. Applying this idea to other permutations is left for future research.

5   Parallelization

We describe the SIMD implementation in Section 5.1 and, on top of it, a multi-threaded implementation in Section 5.2. Finally, we describe our implementation for GPUs in Section 5.3. Because rotation fitting seems more promising than ShockHash in preliminary experiments, we parallelize only the original brute force method and rotation fitting.

5.1 SIMD

For the SIMD parallelization, we mainly focus on the description of bijections and splittings, which take most time of the construction. While we do accelerate additional parts of the code, the ideas are more straight forward and are omitted due to space constraints. The main idea of our SIMD implementation is to try multiple hash function seeds simultaneously.

Bijections. For the bijections, each SIMD lane is responsible for trying one hash function. For this, we load consecutive hash function identifiers and the same input object to each lane of a SIMD vector, and evaluate the hash function of range m, where m is the size of this leaf (usually $= \ell$). The resulting hash value in each lane is converted to a single bit by taking two to the power of it. After calculating the logical OR of these bits for all objects in the set, we check for a bijection by comparing each lane with a constant that has all m lower bits set to 1.

A *Bijection midstop* (see Section 2.2) is a way to check for collisions before hashing all objects. In contrast to the sequential implementation, we cannot simply stop trying one hash function because that would leave one SIMD lane unused. This can be avoided in the SIMD implementation by using a backlog. After the midstop, we store the collision-free hash function identifiers and the respective bit vectors holding already hashed object positions in a backlog. When there are enough objects in the backlog to fill a complete SIMD vector, then this vector is processed as usual to find a bijection. This way, no vector operations are wasted on vectors where most lanes are irrelevant.

For the rotation fitting technique, remember that the number we store as a seed is the hash function identification plus the rotation. This number should be as small as possible to avoid wasting space. To achieve this and to ensure that the resulting MPHF is equal to the one produced by the sequential implementation, caution must be taken when trying out the rotations. When trying out one rotation after the other, another lane with a smaller lane ID might find a bijection for a higher rotation number, which leads to an overall smaller number. Therefore, all rotations are tried out and only at the end it is checked whether a bijection was found.

Splittings. For the splittings, the original implementation uses small arrays of counters. Each counter contains the number of objects hashed to each split section. Using an array for the counts is problematic in the SIMD version. Each SIMD lane would need its own array and expensive gather and scatter instructions are necessary to increment the counts. We therefore use two different methods.

For the upper aggregation levels with fanout 2, we use a single counter for the number of objects hashed to the left child. The number of objects in the right child can then be determined by subtraction.

For leaf size $\ell \leq 24$, each counter of a valid lower level splitting in the current RecSplit implementation fits into a single byte. Because an overflowing counter for one child would then just add 1 to the next counter, such overflows cannot make an invalid splitting look valid. With this, we can then use a lookup table to increment one of the packed counters after hashing an object. For almost all practically relevant leaf sizes ($\ell \leq 21$), this lookup uses a VGATHER instruction. VGATHER performs an array lookup for all lanes in parallel, which is fast starting with Intel Skylake processors.

When a seed for a valid splitting is found, we need to redistribute the objects. We now use SIMD to apply the same hash function to several objects at once, and store the results in an array. We then redistribute the objects without SIMD parallelism.
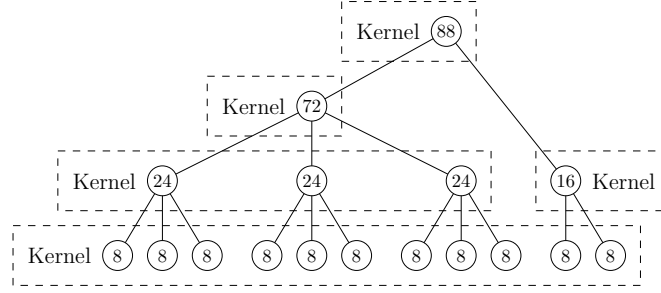
Figure 2 Illustration of how each splitting tree is handled on the GPU.

5.2 Multi-Threading

The original RecSplit implementation only uses a single thread. This leaves a lot of processing power unused since most modern processors contain several processing cores. As stated in the original RecSplit paper, parallelizing RecSplit is fairly easy because the buckets are completely independent of each other. On top of the SIMD parallelization, we therefore now use multiple threads to accelerate the construction even further. We parallelize the implementation by spawning several threads and assigning a consecutive portion of the buckets to each thread. Because the number of buckets is large and the input objects are hashed to buckets uniformly, the load of all threads is reasonably balanced. Each thread needs to know the beginning of its first bucket which is already available after sorting the input.

Until they are finished processing their buckets, all threads work independently. After a splitting or bijection is found, it must be stored in the Golomb-Rice coded sequence. To avoid synchronization, each thread uses its own local sequence and treats its input as if it was the complete input. This means it also stores the pointers to the start of each bucket encoding locally. After all threads are done, we sequentially concatenate the Golomb-Rice sequences and build the combined Elias-Fano data structure holding the prefix sum of bucket sizes and pointers to the bucket encodings.

5.3 GPU

For the GPU implementation, which we describe in the following, we use ideas very similar to the SIMD implementation. First, we reserve enough global memory to store the input and output data for constructing 128 buckets concurrently. Each bucket is constructed individually with its own kernel calls. The reason is that the shape of each tree in a bucket depends on the number of objects hashed to it, which makes it hard to run a single kernel for multiple buckets. Using 128 streams, we still construct 128 buckets concurrently. The buckets are distributed round-robin to the streams. This approach is mainly aimed at large leaf sizes ℓ, where the overhead of starting a kernel is negligible. Better performance for smaller leaves is left for future work as our initial goal was to obtain good performance for the most space efficient configurations. The corresponding kernel calls and asynchronous memory transfers are initiated from the host system.

Each node in each splitting tree is handled by one thread block. For the three lowest levels (bijections and two aggregation levels), all thread blocks are started together using one kernel call (see Figure 2). Note that on these three levels, the size of a node and the starting seed is constant for all nodes on the level (except for possibly the last node for which we launch an extra kernel if necessary). Therefore, these three levels are very homogeneous.

Conversely, the higher levels with fanout $s = 2$ are heterogeneous. The size may be different for different nodes on the same level. This information would need to be provided to every single thread block if we used kernels with more than one thread block. In fact, it is not even clear which nodes could be seen as part of the same level since the splitting tree may not be perfectly balanced, i.e., the length of the shortest path from the root to a leaf may be different for different leaves. Moreover, the number of nodes in the higher levels is generally small compared to the lower levels since each inner node has at least two child nodes. This is especially true for high leaf sizes, which also means high fanouts in the aggregation levels. Therefore, only a single block per splitting is used in the higher levels.

We use the GPU only for splittings and bijections. Because the kernels are launched per level, the results are stored in BFS order. For the final data structure, we need to store them in preorder. For multiple buckets in parallel, the CPU unpacks the resulting seeds recursively and writes them to an encoded sequence. Finally, the already encoded sequences from multiple threads are concatenated sequentially.

Bijections. We load all objects relevant for that node into the shared memory. Then each thread tries out a distinct hash function index. If it finds a bijection, it updates an atomic variable. After k tries, the threads synchronize, check if a bijection was found and if it was, load the hash function index into global memory. The value k is dependent on the size m of that leaf. For $m < 14$ or when using rotation fitting, then $k = 1$.

Since a warp works in lock-step, a bijection midstop (see Section 2.2) is only effective if all threads in a warp detect a conflict. If at least one thread does not detect a conflict, the whole warp has to continue processing the remaining objects. We therefore modify the midstop to again provide a probability of about 90% that all threads in the warp find a collision. Bijection midstop is only used for $m \geq 14$, and not for our new rotation fitting technique.

Splittings. Finding a valid splitting works similar to the SIMD implementation. A word c is initialized to zero, where each thread has its own c. All counts are packed in c, one byte per count. Other than in the SIMD implementation, no lookup table is used for incrementing individual bytes since memory is too slow for this use. Instead, we use multiplication and shifts. An alternative variant that stores counters in shared memory is slower in preliminary experiments, even when padding the counters to reduce the probability of bank conflicts. We therefore use the packed variant in our implementation. To redistribute the objects, we store one counter for each each child bucket. Each thread then takes one object, increments the bucket's counter atomically, and writes the object to the respective position in the bucket.

6 Experiments

We first describe the experimental setup and general improvements. We then continue with the bijection techniques and parallelization. Finally, we compare our implementation with competitors from the literature. The code and scripts needed to reproduce our experiments are available on GitHub: <https://github.com/ByteHamster/GpuRecSplit>. The code for the comparison with competitors is available on GitHub as well: <https://github.com/ByteHamster/MPHF-Experiments>.

Experimental Setup. We run our experiments on an Intel i7 11700 processor with 8 cores (16 hardware threads) and a base clock speed of 2.5 GHz. The machine runs Ubuntu 22.04

with Linux 5.15.0. We use the GNU C++ compiler version 11.2.0 with optimization flags `-O3 -march=native`.

We use the Vector Class Library (VCL) by Fog [17] for most SIMD operations. The SIMD implementation only supports x86 CPUs and is optimized towards AVX2 and AVX-512. The GPU implementation uses CUDA 11. As a reminder, only the *construction* is using SIMD, multithreading, and/or the GPU. The query implementation is identical for the SIMD and GPU implementation and almost equal to the original implementation [13].

For the comparison of different configurations of our data structure, we use random 128-bit integers as input data, following the approach used in the original implementation [13]. For the comparison with competitors, we use strings of uniform random length $\in [10, 50]$ containing random characters except for the zero byte. Note that, as a first step, all competitors generate a *master hash code* (MHC) of each object using a high quality hash function. This makes the remaining computation largely independent of the input distribution. For RecSplit, all subsequent hashing is derived from the MHC using the *remix* function from MurmurHash3, an efficient and simple way to scramble the bits in a hash value.

6.1   Our Implementation

While the original implementation [13] uses std::sort, we use counting sort [36], which is up to ten times faster. It also gives us the prefix sum of bucket sizes directly, which we can use for parallelization. The focus of our implementation are the most space efficient configurations with large ℓ and b. It turns out that the implementation is even competitive for less space efficient configurations, but in these cases sequential sorting is a bottle-neck, limiting the scalability. Parallelizing this step is left for future work.

Because 64-bit multiplications are inefficient on GPUs [1] and SIMD units, we replace the 64-bit multiplicative remix function by a 32-bit version which suffices as buckets are small.

Bijection Search Techniques.   Figure 3 demonstrates the effect of different bijection search methods using a single-threaded, non-vectorized CPU implementation. Given that the methods' parameters are not directly comparable, we plot a Pareto front[1] for space usage versus construction time. The construction time refers to the entire MPHF construction, including the time used for splittings.

Using the same leaf size ℓ, ShockHash is significantly faster than brute force, but it also needs more space. ShockHash achieves speedups of up to 2 with the same space usage, but usually the method is slower at the same space usage.

The rotation fitting technique is consistently faster than the brute force method, making the entire MPHF construction up to 3 times faster. The space overhead of rotation fitting becomes negligible to the noise for large ℓ. Additionally, it is easier to vectorize than a cuckoo hash table construction. Therefore, all following experiments are conducted using only rotation fitting. This also makes the plots easier to read because we only have one set of parameters to deal with.

Dependence on Input Parameters.   In Figure 4, we plot the performance of the SIMD, GPU and non-vectorized versions for different leaf sizes ℓ and bucket sizes b. For better comparability with the original paper [13], we include a wide range of configurations, even

---

[1]A configuration is on the Pareto front if it is not dominated by any other configuration with respect to both construction time and space consumption.
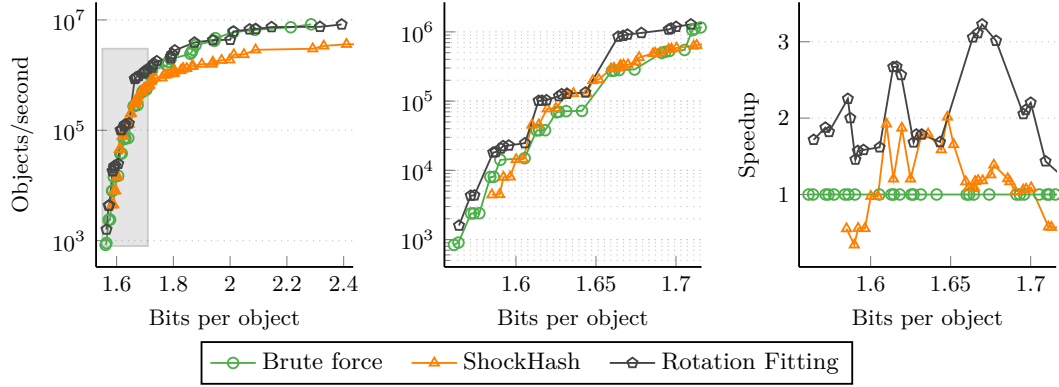
Figure 3 Pareto plot over the construction throughput of different variants of searching for bijections in the leaves. Single-threaded, non-vectorized measurements with $N = 5$ Million objects. The middle plot enlarges the most space efficient area. The plot on the right gives speedups relative to the brute force method.[2]
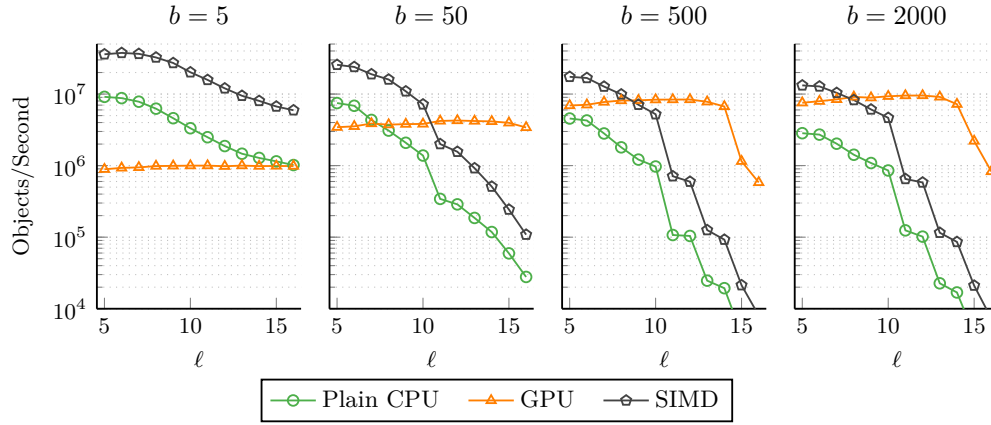

Figure 4 Construction throughput with different hardware architectures based on different input parameters. $N = 5$ Million objects, 1 CPU thread.

ones that are not very competitive. The SIMD version is consistently up to 4.5 times faster than the non-vectorized version and shows the same scaling behavior. The plot indicates that there is no configuration where one would prefer the non-vectorized version. While the GPU has significant speedups for space efficient configurations, it does not speed up the fast, space-inefficient configurations. Reasons for this are overhead due to kernel launches and data transfer. Given that it is more competitive than we expected even for the smaller leaf sizes, we plan to reduce the overhead caused by a large number of kernel calls in future work.

Scaling. Figure 5 shows how the SIMD and GPU versions scale when selecting a different number of CPU threads. The configurations are adopted from the RecSplit paper [13] and

---

[2] Note that giving speedups is non-trivial here because there might not be a configuration that achieves the same space usage that we could compare with. We therefore calculate the speedup relative to an interpolation of the next larger and next smaller data points. This is reasonable since RecSplit instances can be interpolated as well by hashing a certain fraction of objects into data structures with different configurations.
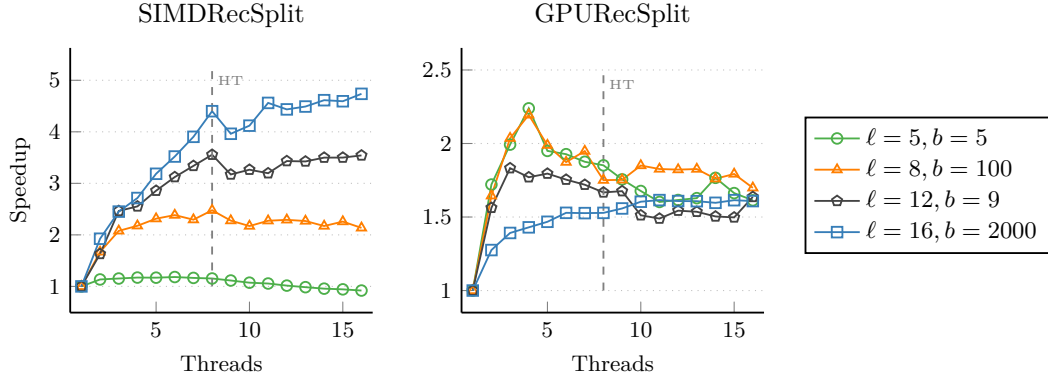
Figure 5 Construction speedup by number of threads used, on a machine with 8 cores and 16 hardware threads (HT), for different configurations. We scale both the number of processors and N (weak scaling). The number of input objects N for the single-threaded measurements is selected such that a construction takes about 3 seconds. Configurations are the examples that are highlighted in the RecSplit paper [13].

each have significantly different scaling behavior.

On the CPU, the most space efficient variant achieves significant speedups. The variants that have rather small buckets spend more time building the data structures holding per-bucket metadata, which is a sequential operation in our implementation. Therefore, the space-inefficient configurations do not profit much from parallelization. This is also visible in the Pareto fronts comparing SIMDRecSplit with competitors from the literature (see Figure 6). Note that our implementation is mainly focused on the most space-efficient configurations. Achieving good scaling behavior with space-inefficient variants is left for future work.

On the GPU, the picture changes drastically. The most space efficient method barely profits from using more CPU threads for kernel submission because the GPU is busy anyway. For the less space efficient variants, using 4–6 CPU threads can be beneficial, with a speedup of about 2. The decrease in performance when using more threads for the less space efficient configurations shows that the approach of starting many small kernels is not profitable. Optimizations for these less space efficient configurations are left for future work.

Overall Speedup.   Our rotation fitting technique leads to a speedup of up to 3 (see Figure 3), with the same space usage. SIMD parallelism improves the construction speed by up to a factor of 4.5 (see Figure 4). Finally, multi-threading for highly space-efficient configurations shows a speedup of close to 5 (see Figure 5). Table 1 shows the overall improvement of our implementation when compared to the original RecSplit implementation [13] The original RecSplit paper says that MPHF construction at 1.56 bits per object is possible. This configuration with $N = 1$ Million objects takes about 19 minutes using the original implementation. Our implementation achieves the same space usage in just 28 seconds on the CPU and 1.5 seconds on the GPU. Investing about 10 minutes of GPU time, our implementation achieves a space usage of only 1.52 bits per object. This is about 30% closer to the lower bound [2] of 1.44 bits, and simultaneously twice as fast as the original implementation.

■ **Table 1** Overall construction times compared to the original RecSplit implementation. $N = 1$ Million objects (weak scaling). Construction times are given in μs/object.

ℓ	b	Method	Threads	B/Object	Construction	Speedup
16	2000	RecSplit [13]	1	1.561	1152.6	
16	2000	SIMD	1	1.561	139.2	8
16	2000	SIMD	16	1.562	28.1	40
16	2000	GPU	4	1.562	1.5	763
18	50	RecSplit [13]	1	1.711	2919.5	
18	50	SIMD	1	1.707	58.0	50
18	50	SIMD	16	1.709	12.1	241
18	50	GPU	4	1.708	1.4	2072
24	2000	GPU	4	1.524	633.9	

■ **Table 2** Energy consumption of different configurations with $\ell = 18, b = 50$. Construction duration and energy usage are given for $N = 1$ Million objects. Energy consumption is both given as difference to the idle power of 78 W, as well as total energy consumption of the whole system. For the total consumption of CPU-only measurements, we subtract the 16 W GPU idle power.

Method	Threads	Duration	Total system		Difference to idle	
			Power	Energy	Power	Energy
RecSplit [13]	1	19 min	99 W	112 860 J	37 W	42 180 J
SIMDRecSplit	1	58 sec	108 W	6 264 J	46 W	2 668 J
SIMDRecSplit	16	12 sec	124 W	1 488 J	62 W	744 J
GPURecSplit	4	1.4 sec	408 W	571 J	330 W	462 J

Energy Consumption. Of course, directly comparing CPU and GPU implementations is unfair. A sensible metric to compare them is the energy consumption, which can be a major cost factor. Additionally, the energy consumption is not influenced by market prices. We therefore measure the energy consumption of our system, specifically, the difference to the idle power. Table 2 gives measurements for different configurations and hardware architectures. The energy consumption is homogeneous throughout most of the execution time, except for a short ramp-up in the beginning. We do not count the ramp-up to the energy consumption. Measurements are performed using a Voltcraft 870 Multimeter with USB interface and power adapter.

Even though SIMD instructions need slightly more power, the total energy consumption of constructing one MPHF is about 18 times lower. The GPU, even though it needs significantly more power, is so much faster that the resulting energy usage of constructing one MPHF is close to 200 times lower than the original single-threaded CPU implementation.

6.2 Comparison with Competitors

In our evaluation, we compare our parallelization of RecSplit to SicHash [26], PTHash [33], BBHash [27], CHD [2], and the original RecSplit [13]. For the scaling plots, we include the partitioning-based parallelization PTHash-HEM [34] and a simple partitioning-based parallelization of SicHash and SIMDRecSplit. The methods then first hash input objects to different partitions, which can then be constructed independently in parallel. The partitioning
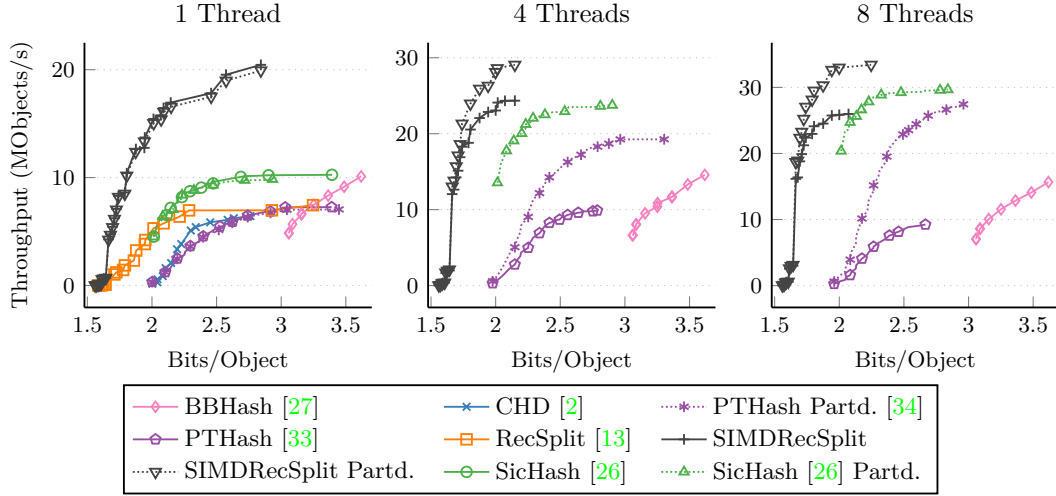
Figure 6 Competitor trade-off of construction time vs space usage. Weak scaling, $N/p = 10$ Million objects. The methods labeled with "Partd." do not provide a native parallelization of the construction algorithm itself, but partition the input objects to independent perfect hash functions. This approach can be used for all PHF construction algorithms but increases the query time. For SicHash and PTHash, we plot all Pareto optimal data points but only show markers for every fourth point to increase readability. Therefore, the lines might bend on positions without markers.

idea can universally be applied to all perfect hashing methods to reduce sequential bottlenecks, but it introduces some query overhead. Note that our focus is still on parallelizing the native methods because it is transparent to the queries.

Space usage trade-off.   Figure 6 plots a space versus construction time Pareto front for each approach. Looking at a single thread first, we make the surprising observation that SIMDRecSplit not only wins for the most space efficient configurations for which we designed it but, by far, dominates all the other methods also for less space-efficient cases. For parallel construction, SIMDRecSplit scales well for the space-efficient cases but less so for the remaining ones, resulting in a steeper Pareto front. Nevertheless, SIMDRecSplit still dominates the other methods although by a lesser margin. It seems that all the current methods run into scalability bottlenecks unrelated to the particular approach used.

Table 3 lists construction and query time of a selection of competitor configurations. When looking at the query time, PTHash is a clear winner. While BBHash can achieve the same query speed and good construction speed, its space usage is large. SicHash has a query time close to PTHash's most compact representation, but is faster to construct and more space efficient. All RecSplit variants can achieve significantly lower space than other competitors but require considerably more query time. However, our single-threaded SIMD implementation dominates most competitors with respect to both space and construction time. The use of rotations makes the queries about 10% slower than the original RecSplit implementation. The main goal of RecSplit is to achieve extremely small representation, and queries are not very fast to begin with, so this seems acceptable.

Construction Scaling.   You can find strong scaling and weak scaling experiments in Figure 7. As described in Section 6.1, the scaling behavior of SIMDRecSplit depends on the parameters selected. Because our main focus is on space efficient configurations, the scaling of configu-

Table 3 Query and construction measurements of different competitor configurations, using $N = 10$ Million objects. Query time is given in ns per query, construction time in ns per object.

| Method	B/Object	Construction	Query
BBHash [27], $\gamma = 1.0$	3.059	208 ns	51 ns
BBHash [27], $\gamma = 5.0$	6.871	50 ns	36 ns
PTHash [33], $c = 7.0$, $\alpha = 0.99$, C-C	3.313	199 ns	20 ns
PTHash [33], $c = 11.0$, $\alpha = 0.88$, D-D	4.379	138 ns	25 ns
PTHash [33], $c = 6.0$, $\alpha = 0.99$, EF	2.345	248 ns	35 ns
SicHash [26], $\alpha = 0.97$, $p_1 = 44$, $p_2 = 30$	2.081	172 ns	40 ns
SicHash [26], $\alpha = 0.9$, $p_1 = 20$, $p_2 = 77$	2.412	119 ns	41 ns
RecSplit [13], $\ell = 5$, $b = 5$	2.928	145 ns	65 ns
RecSplit [13], $\ell = 8$, $b = 100$	1.793	709 ns	75 ns
RecSplit [13], $\ell = 14$, $b = 2000$	1.584	126534 ns	96 ns
SIMDRecSplit, $\ell = 5$, $b = 5$	2.96	49 ns	71 ns
SIMDRecSplit, $\ell = 8$, $b = 100$	1.806	107 ns	80 ns
SIMDRecSplit, $\ell = 14$, $b = 2000$	1.585	11742 ns	110 ns

rations with small ℓ and b is not as good. For large ℓ, the speedups of our SIMDRecSplit implementation are close to speedups that competitors only achieve through partitioning.

7 Conclusion and Future Work

We have shown that by harnessing parallelism at all available levels – bits, vectors, cores, and GPUs – one can dramatically accelerate the construction of highly space efficient minimal perfect hash functions (MPHFs) using the brute-force RecSplit approach [13]. This leads to speedups of up to 241 on SIMD and 2072 on the GPU and also dramatically reduces energy consumption. Surprisingly, this even turns out to be the fastest available approach for constructing less space-efficient MPHFs. This is not what we expected. Our initial hypothesis was that there would be trade-off with asymptotically faster approaches winning for less requirements on space consumption. We therefore also explored further approaches to "fast" search like ShockHash. Only a small step in this direction was successful – rotation fitting is still brute force but reduces the work needed per tried hash function while adding a tiny little bit of space requirement. But even the asymptotically "obvious" improvement of replacing ℓ rotations/checks by two table lookups are not productive on current architectures. So, brute-force, simplicity (in the inner loops), and parallelism currently wins against any attempt at sophistication. We believe that this will be further amplified by further engineering the parallelization of RecSplit (e.g., hashing, sorting, and data structure assembly).

If we accept this conclusion, the main open problem is to improve query time. Traversing an aggressively compressed tree for each hash function evaluation is inherently more expensive than the simple constant time operations needed in PTHash [33] or SicHash [26] but there should be more efficient ways to break down MPHF construction into small subproblems that can be solved with brute-force. We believe that the techniques developed here will turn out to be useful in that respect.

There may also be a renaissance of clever search. Perhaps the approach from rotation fitting to use a lookup table for normalizing bit patterns could be generalized to a richer set of mappings than just rotations. Also, rotation fitting could be generalized by splitting
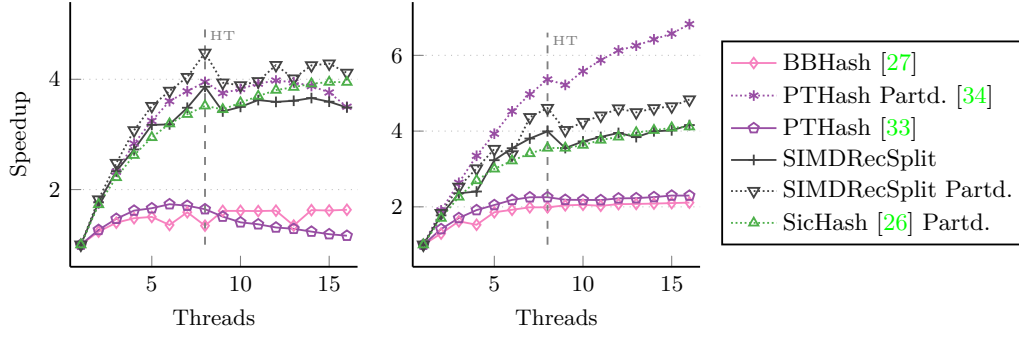
(a) Weak scaling: $N = t \cdot 10$ Million. **(b)** Strong scaling: $N = 50$ Million.

■ **Figure 7** Construction time speedups when using multiple threads, on a machine with 8 cores and 16 hardware threads (HT). Speedups are given relative to each method's single threaded performance. The methods labeled with "Partd." do not provide a native parallelization of the construction algorithm itself, but partition the input objects to independent perfect hash functions. This approach can be used for all PHF construction algorithms but increases the query time. Note that the suboptimal weak scaling behavior of PTHash can be explained with the fact that on PTHash, different N leads to significantly different space efficiency.

into more than two parts. The resulting search for several rotations gives more room for sophistications like search space pruning.

Finally, we can look for generalizations of RecSplit for computing non-minimal PHFs which allows us to further reduce space consumption of the hash function itself.

References

1 Nvidia developer forums - question about 64 bit integer performance. Technical report, 2018.

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 Michael A Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 182–193. IEEE, 2018.

4 Dominik Bez. Perfect hash function generation on the GPU with RecSplit. Master's thesis, 2022. doi:10.5445/IR/1000152719.

5 Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Perfect hashing for data management applications. *arXiv preprint cs/0702159*, 2007.

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 Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Inf. Syst.*, 38(1):108–131, 2013. `doi:10.1016/J.IS.2012.06.002`.

8 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, 08 2011. `doi:10.1371/journal.pone.0023501`.

9 David Clark. Compact pat trees. 1997.

10 Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Inf. Process. Lett.*, 43(5):257–264, 1992. `doi:10.1016/0020-0190(92)90220-P`.

11 Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. In *20th Symposium on Experimental Algorithms (SEA)*, 2022.

12 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974. `doi:10.1145/321812.321820`.

13 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`.

14 Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

15 Robert Mario Fano. On the number of bits required to implement an associative memory. Technical report, MIT, Computer Structures Group, 1971. Project MAC, Memorandum 61".

16 Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21(9):948–960, 1972. `doi:10.1109/TC.1972.5009071`.

17 Agner Fog. C++ vector class library. Technical report, 2013.

18 D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Comp. Syst.*, 38(2):229–248, 2005.

19 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`.

20 Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

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

22 Intel. Advanced vector extensions programming reference. Technical report, 2011.

23 Intel. Avx-512 instructions. Technical report, 2013.

24 Florian Kurpicz. Engineering compact data structures for rank and select queries on bit vectors. In *SPIRE*, volume 13617 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2022. `doi:10.1007/978-3-031-20643-6_19`.

25 Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Trans. Graph.*, 25(3):579–588, 2006. `doi:10.1145/1141911.1141926`.

26 Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer. Sichash - small irregular cuckoo tables for perfect hashing. *CoRR*, abs/2210.01560, 2022. `doi:10.48550/ARXIV.2210.01560`.

27 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`.

28 Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *International Symposium on Experimental Algorithms*, pages 138–149. Springer, 2014.

29 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`.

30 Nvidia. Nvidia ampere GA102 GPU architecture. Technical report, 2020.

31 Nvidia. CUDA C++ programming guide. Technical report, 2022.

32 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

33 Giulio E. Pibiri and Roberto Trani. PTHash: Revisiting FCH minimal perfect hashing. In *SIGIR*, pages 1339–1348. ACM, 2021. `doi:10.1145/3404835.3462849`.

34 Giulio Ermanno Pibiri and Roberto Trani. Parallel and external-memory construction of minimal perfect hash functions with pthash. *CoRR*, abs/2106.02350, 2021.

35 Robert F. Rice. Some practical universal noiseless coding techniques. *Jet Propulsion Laboratory, JPL Publication*, 1979.

36 Harold Herbert Seward. *Information sorting in the application of electronic digital computers to business operations*. PhD thesis, Massachusetts Institute of Technology. Department of Electrical Engineering, 1954.

37 Sean A. Weaver and Marijn Heule. Constructing minimal perfect hash functions using SAT technology. In *AAAI*, pages 1668–1675. AAAI Press, 2020.