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) bijectively maps a set S of objects 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. It heavily relies on trying out hash functions in a brute force way.

We introduce *rotation fitting*, a new technique that makes the search more efficient by drastically reducing the number of tried hash functions. Additionally, we greatly improve the construction time of RecSplit by harnessing parallelism on the level of bits, vectors, cores, and GPUs.

In combination, the resulting improvements yield speedups up to 239 on an 8-core CPU and up to 5438 using a GPU. The original single-threaded RecSplit implementation needs 1.5 hours to construct an MPHF for 5 Million objects with 1.56 bits per object. On the GPU, we achieve the same space usage in just 5 seconds. Given that the speedups are larger than the increase in energy consumption, our implementation is more energy efficient than the original implementation.

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

Keywords and phrases compressed data structure, parallel perfect hashing, bit parallelism, GPU, SIMD, parallel computing, vector instructions

Digital Object Identifier 10.4230/LIPIcs.ESA.2023.19

Related Version *Extended Version*: `https://arxiv.org/abs/2212.09562` [5]

Supplementary Material *Software (Source Code)*: `https://github.com/ByteHamster/GpuRecSplit` archived at `swh:1:dir:1245e6eaef109ce4eb9f24080a2e9bdad7baf6d1`
Software (Comparison with Competitors): `https://github.com/ByteHamster/MPHF-Experiments` archived at `swh:1:dir:890e76e03dd70e63eb57f3f62e466a4ee825cee4`

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).

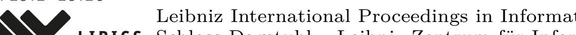
Acknowledgements This paper is based on and has text overlaps with Dominik Bez' Master's thesis [4]. We refer readers to that thesis for a detailed evaluation of the effects of low-level decisions like the choice of different similar SIMD instructions.

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

© Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders;
licensed under Creative Commons License CC-BY 4.0

31st Annual European Symposium on Algorithms (ESA 2023).
Editors: Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman; Article No. 19; pp. 19:1–19:16



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$n = |S|$ integers, so it is bijective. MPHFs are useful in many applications, for example, to implement hash tables with guaranteed constant access time [21]. By storing only fingerprints in the hash table cells [3, 15], we obtain an *approximate membership data structure*. Storing payload data in the cells, we obtain an updatable retrieval data structure [32]. Finally, the perfect hash function values can be used as small identifiers of the input objects [6], 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 [14]. It provides various trade-offs between the space consumption, construction time, and query time.

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 a new bijection search mechanism in Section 4, which reduces 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 positive side effect, this approach makes good use of bit parallelism.

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, 239 when additionally using multi-threading with 16 threads, and 5438 using a GPU, compared to the original single-threaded implementation without rotation fitting. 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. With *rotation fitting*, we introduce a new method for searching for bijections that can be used in RecSplit. We significantly accelerate the construction by four kinds of parallelism (bits, vectors, multicores, and GPU). Together, this accelerates RecSplit constructions by a factor up to 5438 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 1-bits in a word. Given a bit vector, the $\mathit{rank}_1(x)$ operation returns the number of 1-bits before position x, and the $\mathit{select}_1(x)$ operation returns the position of the x-th 1-bit. The operation can be executed in constant time [10, 27] and has very fast and space-efficient implementations [28, 39]. An additional operation we need in this paper is $\mathrm{rot}_k^i(x)$ 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.

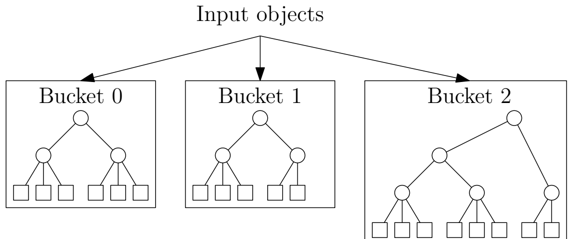


Figure 1 Illustration of the overall RecSplit data structure. Circular nodes of the trees represent splittings, squares represent bijections.

Golomb-Rice. The Golomb code [24] is a variable length code that is optimal for geometric distributions. Golomb-Rice [38] is 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 form the *fixed part* which is stored directly. The remaining bits are encoded in unary, consisting 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 [13,16] can be used to store a monotonic sequence of integers $p_1, \ldots, p_k$ with $p_k \leq U$. 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. 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. Using this representation, the sequence can be stored using $k(2 + \log(U/k))$ bits.

2.2 RecSplit

We now describe RecSplit [14], the MPHF that this paper is based on. Figure 1 illustrates the overall 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.

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 have a small configurable size ℓ. The splitting tree has a well-defined shape, depending only on the leaf size ℓ and the number of objects in the bucket. At each inner node, RecSplit tries random hash functions to find one that distributes the objects to the child nodes according to the tree structure. 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.

Bijections. The lowest level of the splitting tree is called *leaf level*. Each leaf, except for possibly the last, contains ℓ objects. This is small enough that it is feasible to search for a bijective mapping by trying random hash functions using brute force. 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.

Representation. Because the splitting trees have a well-defined shape, it is enough to store the hash function identifier at each node in preorder. These numbers are encoded with Golomb-Rice code, where all unary parts and all binary parts of a tree are stored together. The optimal Golomb parameter τ is different based on the layer in the tree and can be pre-calculated. The encodings of all splitting trees from all buckets are concatenated in a single bit vector. An additional sequence with encoding based on Elias-Fano encodes both the prefix sums of the number of objects in each bucket and the positions where the encoding of each bucket starts.

Query. A RecSplit hash function can be evaluated by determining the bucket of an object and locating its encoding. The splitting tree in the bucket is traversed from the root to a leaf by applying the splitting hash function, which determines the child to descend into. Finding the encoding of a subtree is possible by executing a $select_1$ query on the upper bits of the Golomb-Rice coded hash function identifiers. During traversal, the number of objects stored in children left to the one descended into are accumulated. The final hash value is then the sum of the value of leaf bijection, the number of objects to the left in the splitting tree, and the total size of previous buckets.

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 [14].

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) [17]. A single instruction is used to apply the same operation on a *vector* of several elements. We refer to a single element within a SIMD vector as a *lane*. For example, a vector may contain 16 lanes with 32 bits each, i.e., the vector contains 512 bits overall. The exact set of operations depends on the concrete implementation of the SIMD model. On many Intel and AMD processors, SIMD operations are available through the Advanced Vector Extensions (AVX) [25]. AVX-512 [26] extends these operations to 512-bit vectors and is divided into many smaller subsets that offer additional operations. A subset that 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 parallel using SIMD.

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 [33], 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.

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. *Shared memory* is a fast memory placed on each SM. It is shared between the threads of the same thread block. On the RTX 3090, shared memory and L1 cache are allocated on the same memory areas. The data in shared memory is partitioned into 32 memory banks, and the i-th 32-bit word is stored in bank $i \bmod 32$. When multiple threads simultaneously access different words within the same bank, the access operations have to be serialized.

CUDA. An efficient way to develop applications on NVIDIA GPUs is CUDA [34]. Functions which can be executed on the GPU 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*. The kernels and data transfers launched into a specific stream are executed in order, but operations in different streams can arbitrarily overlap.

3 Related Work

Perfect Hashing is an active area of research [2, 7, 8, 9, 11, 20, 30, 31, 32, 37, 40]. 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. [30]. To the best of our knowledge, there is no technique that constructs MPHFs on the GPU yet. Lefebvre and Hoppe [29] 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 [9, 31, 32]. BBHash [31] 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 $rank_1(h(s))$ defines an MPHF on $S \setminus S'$. 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 the very simple and easily parallelizable construction.

PTHash. PTHash [37] is based on FCH [20] 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 other objects that are already placed. 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 more space, but has faster queries and faster construction time. PTHash-HEM [36] is an implementation that first partitions the input and then constructs each partition independently in parallel.

SicHash. SicHash [30] 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 [12]. Computing a valid index for all objects amounts to constructing a cuckoo hash table [19, 35]. In contrast to the brute force methods at the core of 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. Roughly, SicHash allows faster construction than PTHash while offering similar query time and space consumption.

4 Rotation Fitting

The general idea of RecSplit consists of two independent steps, bijections and splittings (see Section 2.2). In this section, we introduce a new method for searching for bijections in RecSplit's leaf nodes. As a reminder, given m objects, we are looking for a way to quickly find a mapping of the objects to the first m integers without any collisions. The original implementation tries out hash functions using brute force until one of them is a bijection.

Rotation fitting 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 when $m \leq w$, where w is the size of a machine word. We randomly distribute the objects into two sets A and B by using a 1-bit hash function. The 1-bit hash function is the same for all leaf nodes and does not ensure that A and B have the same size. Now we search for a hash function h that gives a bijection on the leaf. Like in the original RecSplit implementation, 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 using the same hash function h. Let us now rotate (i.e., cyclically shift) the bits in b. If we can find a rotation value such that the 1-bits in b fit exactly onto the 0-bits in a, we have found a bijection on the leaf. More formally, this is the case if there is an $r \in \{0, ..., m-1\}$, such that $a \,|\, \text{rot}_m^r(b)$ has the m least significant bits all set. In the extended version [5], we show that for large m the probability of finding a bijection using rotation fitting is about m times higher than the probability when using RecSplit's brute force approach.

To efficiently store r, we only try hash function identifiers which are multiples of 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. The space overhead per object introduced by rotation fitting tends to 0 for large m [5].

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 $r = (t[\hat{b}] - t[a]) \bmod m$ using two table lookups. Rotation r is valid if $a | \mathrm{rot}_m^r(b) = c$.

Because rotation is a very cheap operation, preliminary experiments show no improvement by lookup tables. Especially on GPUs, shared memory is a scarce resource and global memory is too slow. Our implementation therefore does not use lookup tables. Nonetheless, rotation fitting with lookup tables provides an asymptotic improvement of the running time by a factor of m. We also 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, the multi-threaded implementation in Section 5.2. Finally, we describe the GPU implementation in Section 5.3.

5.1 SIMD

For the SIMD parallelization, we focus on the description of bijections and splittings, which (in most configurations) take most time of the construction. While we additionally accelerate the construction of the Elias-Fano data structure, the ideas are more straight forward and are omitted due to space constraints. The main idea of our SIMD parallelization is to try multiple hash function seeds simultaneously. Depending on the operation, we use SIMD lanes with a width of either 32 bits or 64 bits.

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. 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. For rotation fitting, 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, so caution must be taken when trying out the rotations. If one lane finds a bijection, it might be possible that a higher rotation leads to a bijection on a lane with a smaller hash function index. Because this gives a smaller overall number to store, we always try all rotation values, even if a bijection is found.

Splittings. For the splittings, the original implementation uses small arrays of counters. Each counter contains the number of objects hashed to the respective split section. Instead, we 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 all practical leaf sizes ($\ell \leq 24$), each counter of a valid *lower* level splitting 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. 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.

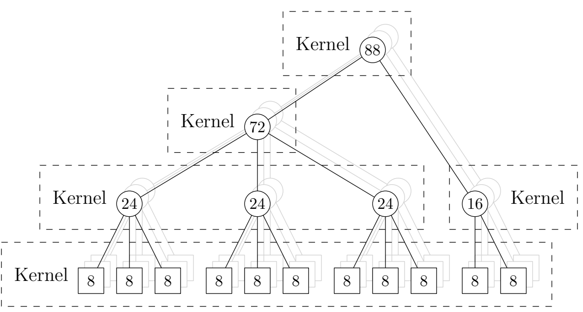


Figure 2 Illustration of how all equally-shaped splitting trees are handled together on the GPU.

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 multiple processing cores. As stated in the original RecSplit paper [14], parallelizing RecSplit is fairly easy because the buckets are completely independent of each other. First, we sort the input objects by their bucket index in parallel, and then determine the bucket borders. We then start several threads and assign 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.

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

In the GPU implementation, we first partition the objects to their buckets and partition the buckets by their respective size. We then use the GPU to determine the splittings and bijections within the buckets. Buckets with the same size have splitting trees with the same shape and can therefore be handled efficiently within the same set of kernel calls. This keeps the number of kernel calls small and is important for scalability. Using CUDA's streams, we additionally construct different bucket shapes concurrently, to utilize the GPU in case the number of buckets having a specific shape is small. For an overview, see Figure 2.

Bijections. All leaf nodes[1] of all trees with the same shape are constructed with a single kernel call. For each leaf node, we start one block of threads. First, the threads in each block cooperate to load all objects relevant for that leaf node into the shared memory. Similar to the SIMD implementation, where each lane tried a different hash function, now each thread tries a different hash function. After each hash function, the threads synchronize, check if a bijection was found and if it was, store the hash function index into global memory.

---

[1] All leaf nodes except possibly the last of each tree, which might have fewer objects.

Splittings. Like for the bijections, each splitting is handled by a thread block. The threads cooperate to load the objects into the shared memory and then each thread tries a different hash function index. For the two lowest aggregation levels, the thread blocks of all nodes in that level are started together using one kernel call (see Figure 2). Note that on these levels, the size of a node and the starting seed is constant. Therefore, the levels are very homogeneous. Conversely, the higher levels with fanout $s = 2$ are more heterogeneous. In particular, the number of objects on a specific level may be different for different nodes on the same level. Therefore, we launch individual kernels for each of those splittings, which contain the thread block for all trees with the same shape. We use multiplication and shifts to increment the counters of how many objects ended up in each lane. 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. After a valid splitting is found, the threads in a block cooperate to reorder the objects in that node accordingly.

Assembly. 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. The CPU unpacks the resulting seeds recursively and writes them to an encoded sequence.

6 Experiments

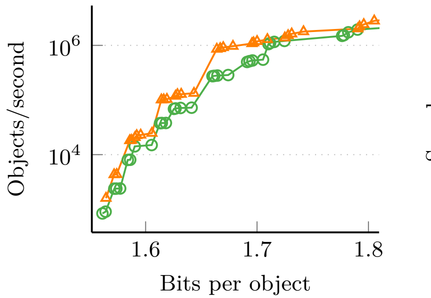
We first describe the experimental setup and general improvements. We then continue with a comparison of different techniques of our implementation before comparing the implementation with competitors from the literature. The code and scripts needed to reproduce our experiments are available on GitHub under the General Public License [22, 23].

Experimental Setup. We ran most of our experiments on an Intel i7 11700 processor with 8 cores (16 hardware threads (HT)) and a base clock speed of 2.5 GHz, supporting AVX-512. The machine runs Ubuntu 22.04 with Linux 5.15.0 and contains an NVIDIA RTX 3090 GPU. For additional experiments, we used a machine with an AMD EPYC 7702P processor with 64 cores (128 hardware threads) and a base clock speed of 2.0 GHz. The machine runs Ubuntu 20.04 with Linux 5.4.0 and supports only AVX2. Unless otherwise noted, all experiments were run on the Intel machine. We used the GNU C++ compiler version 11.2.0 with optimization flags `-O3 -march=native`. The SIMD implementation only supports x86 CPUs and is optimized towards AVX-512 using the Vector Class Library [18]. The GPU implementation uses CUDA 11. As a reminder, only the *construction* is using SIMD, multi-threading, and/or the GPU. The query implementation is identical for the SIMD and GPU implementation and almost equal to the original implementation [14]. We therefore did not compare the query performance of SIMD and GPU implementation.

For the comparison with competitors, we used 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. When only comparing different configurations of our own data structure, we used random 128-bit integers directly as MHC, which follows the approach of the original implementation [14].

6.1 Our Implementation

While the original implementation [14] uses std::sort to partition objects into buckets, we use IPS^{2}Ra [1]. For the less space efficient configurations ($\ell < 5, b < 100$), constructing the buckets is fast, so significant time is spent on sorting objects to buckets. For these



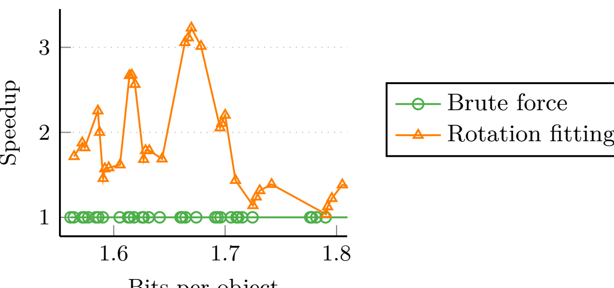


Figure 3 Pareto front over the construction throughput of different variants of searching for bijections in the leaves. Single-threaded, non-vectorized measurements with $n = 5$ Million objects. The plot on the right gives speedups relative to the brute force method.[3]

configurations, IPS^{2}Ra both speeds up the sequential case and also enables sorting in parallel. For more space-efficient configurations ($\ell > 8$), the partitioning step needs less than 1% of the total construction time, both in the parallel and the sequential case. In this section, we compare against a slight adaption of the original implementation, using IPS^{2}Ra and supporting parallel construction.

Rotation Fitting. In order to compare rotation fitting with the brute force variant, we give a Pareto front[2] of space usage versus construction time in Figure 3. The construction time refers to the entire MPHF construction, including the time used for splittings. Rotation fitting is consistently faster, making the entire MPHF construction up to 3 times faster. The space overhead of rotation fitting becomes negligible for moderately large ℓ (see extended version [5]). Unless otherwise noted, all following experiments use rotation fitting.

Dependence on Input Parameters. In Figure 4, we plot the throughput of the SIMD, GPU and non-vectorized versions for different leaf sizes ℓ and bucket sizes b. For better comparability with the original paper [14], we include a wide range of configurations, even 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 in ℓ. The plot indicates that there is no configuration where one would prefer the non-vectorized version. While the GPU offers significant speedups for space efficient configurations, it performs not as good for the space inefficient configurations. A reason for this is data transfers to and from the GPU.

Multi-Threading. Table 1 shows that the parallel construction is up to 5 times faster on an 8-core machine. In the extended version [5], we give more detailed measurements of how different RecSplit configurations scale in the number of CPU threads. Rather unusual configurations with extremely small buckets ($b = 5$) do not scale as well, because they spend a lot of time partitioning the objects to buckets – even though we already use the highly optimized parallel sorter IPS^{2}Ra [1].

[2] 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.

[3] 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.

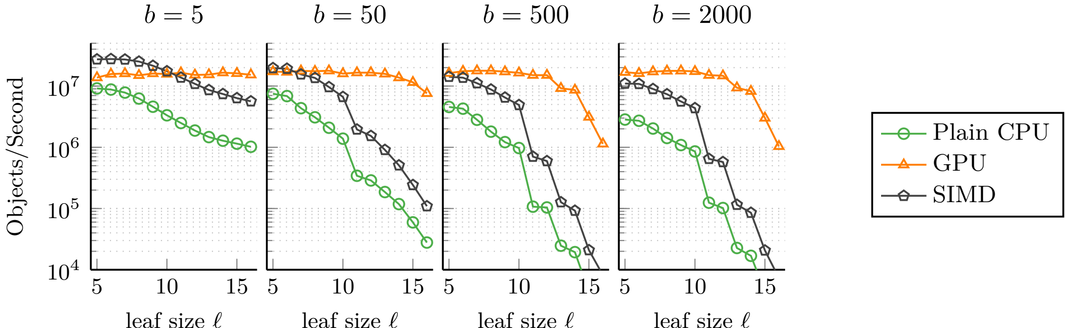


Figure 4 Construction throughput with different hardware architectures based on different input parameters. $n = 5$ Million objects, 1 CPU thread.

Overall Speedup. Our rotation fitting technique leads to a speedup of up to 3 (see Figure 3) and SIMD parallelism improves the construction speed by up to a factor of 4.5 (see Figure 4). Multi-threading for highly space-efficient configurations shows a speedup of close to 5. Table 1 shows the overall improvement of our implementation on CPU and GPU when compared to the original RecSplit implementation. The original RecSplit paper says that MPHF construction at 1.56 bits per object is possible. This configuration with 5 Million objects takes about 1.5 hours using the original implementation. Our SIMD implementation achieves the same space usage in just 2 minutes on the CPU and 5 seconds on the GPU. Investing about 40 minutes of GPU time, our implementation achieves a space usage of only 1.495 bits per object. This is about 40% closer to the lower bound [2] of 1.44 bits, and simultaneously more than twice as fast as the original implementation.

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. Table 2 gives energy consumption 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.

Even though SIMD instructions need slightly more power, the total energy consumption of constructing one MPHF is lower. The GPU, even though it needs significantly more power, is so much faster that the resulting energy usage is about 1000 times lower than the original single-threaded CPU implementation. For basic RecSplit, the AMD machine needs about 1.5 times more time than the Intel machine. This can be readily explained by a lower clock frequency. This performance gap grows to a factor 4.6 for sequential SIMDRecSplit. The likely main reason is that the AMD machines lacks the AVX-512 vector units of the Intel machine. Still, since both processors have two 256-bit AVX2 units per core, it seems that better performance might be achievable with careful tuning for the AMD architecture. On the contrary, the AMD machine shows good scalability so that the energy consumption when using the entire machine is only a factor 1.3 larger than on the Intel machine – despite the fact that our implementation was tuned for the Intel architecture.

Table 1 Construction time of the GPU implementation compared to our multi-threaded adaption of the original RecSplit implementation. $n = 5$ Million objects (strong scaling). Construction times are given in μs/object. We do not report speedups for $\ell = 24$ because the CPU baseline takes too long for this configuration.

Configuration	Method	Bijections	Threads	B/Obj	Constr.	Speedup
$\ell = 16, b = 2000$	RecSplit [14]	Brute force	1	1.560	1175.4	1
	RecSplit	Brute force	16	1.560	206.5	5
	SIMDRecSplit	Rotation fitting	1	1.560	138.0	8
	SIMDRecSplit	Rotation fitting	16	1.560	27.9	42
	GPURecSplit	Brute force	GPU	1.560	1.8	655
	GPURecSplit	Rotation fitting	GPU	1.560	1.0	1173
$\ell = 18, b = 50$	RecSplit [14]	Brute force	1	1.707	2942.9	1
	RecSplit	Brute force	16	1.713	504.0	5
	SIMDRecSplit	Rotation fitting	1	1.709	58.3	50
	SIMDRecSplit	Rotation fitting	16	1.708	12.3	239
	GPURecSplit	Brute force	GPU	1.708	5.2	564
	GPURecSplit	Rotation fitting	GPU	1.709	0.5	5438
$\ell = 24, b = 2000$	GPURecSplit	Brute force	GPU	1.496	2300.9	—
	GPURecSplit	Rotation fitting	GPU	1.496	467.9	—

6.2 Comparison with Competitors

We now compare our implementation to the sequential codes RecSplit [14], SicHash [30], and CHD [2] as well as the parallel codes PTHash [37], PTHash-HEM [36] and BBHash [31].

Space usage trade-off. Figure 5 shows 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 even strengthens its margin to the competing approaches.

Construction Scaling. Figure 6 compares scaling behavior of the parallel codes. We see that BBHash scales poorly while both PTHash-HEM and SIMDRecSplit scale well on the 8-core Intel machine. However, SIMDRecSplit scales better than PTHash-HEM on the 64-core AMD machine.

Queries. Table 3 shows that 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. 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.

Table 2 Energy consumption with $\ell = 18, b = 50$ and $n = 5$ Million objects. Energy consumption is both given as difference to the idle power, as well as total energy consumption of the whole system. For CPU-only measurements of the 8-core Intel machine, we dismount the GPU.

				Total system		Δ to idle	
Machine	Method	Threads	Constr. Seconds	Power Watt	Energy Joule	Power Watt	Energy Joule
8-core Intel	RecSplit [14]	1	14 714.5	78	1 147 731	37	544 436
	SIMDRecSplit	1	291.5	87	25 360	46	13 409
	SIMDRecSplit	16	61.5	104	6 396	63	3 874
	GPURecSplit		2.5	457	1 142	380	950
64-core AMD	RecSplit [14]	1	21 620.8	223	4 821 438	91	1 967 492
	SIMDRecSplit	1	1 328.7	224	297 629	92	122 240
	SIMDRecSplit	128	23.6	364	8 590	232	5 475

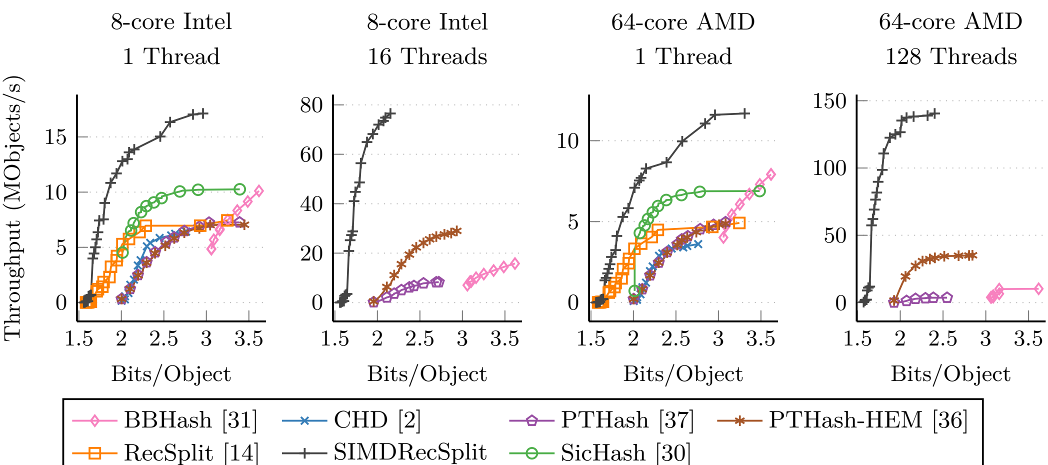


Figure 5 Trade-off of construction time vs space usage. Weak scaling, $n/p = 10$ Million objects. For SicHash and PTHash, we plot all Pareto optimal data points but only show markers for every fourth point to increase readability. Therefore, the lines might bend on positions without markers.

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 [14]. This leads to speedups of up to 239 on SIMD and 5438 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 a trade-off with asymptotically faster approaches winning for fewer requirements on space consumption. Our new technique *rotation fitting* reduces the work needed per tried hash function while adding a tiny bit of space requirement. 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 algorithmic sophistication.

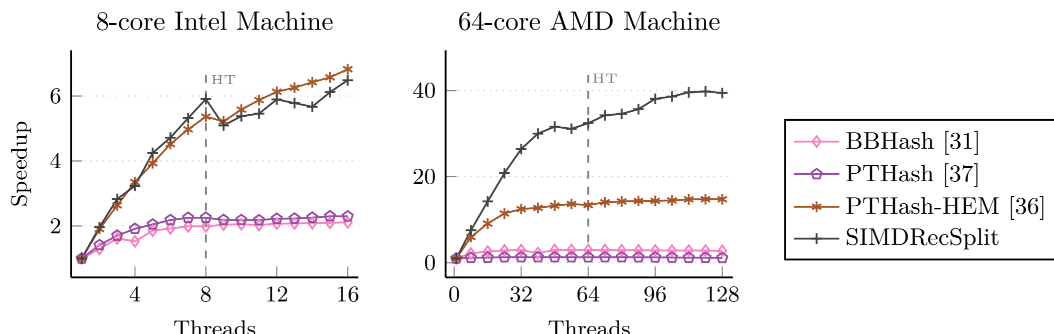


Figure 6 Construction time speedups when using multiple threads t. Strong scaling, $n = 50$ Million. Speedups are given relative to each method's single threaded performance. For a comparison of absolute performance, refer to Figure 5. Configurations used are BBHash: $\gamma = 2.0$; PTHash/PTHash-HEM: $c = 6.0$, $\alpha = 0.95$, EF; SIMDRecSplit: $\ell = 10$, $b = 2000$.

Table 3 Query and construction time of different competitor configurations on 10 Million objects.

Method	Bits/Obj.	Constr./Obj.	Query/Obj.
BBHash [31], γ=5.0	6.871	50 ns	36 ns
BBHash [31], γ=1.0	3.059	208 ns	51 ns
PTHash [37], c=11.0, α=0.88, D-D	4.379	138 ns	25 ns
PTHash [37], c=7.0, α=0.99, C-C	3.313	199 ns	20 ns
PTHash [37], c=6.0, α=0.99, EF	2.345	248 ns	35 ns
SicHash [30], α=0.9, p_1=20, p_2=77	2.412	119 ns	41 ns
SicHash [30], α=0.97, p_1=44, p_2=30	2.081	172 ns	40 ns
RecSplit [14], ℓ=5, b=5	2.928	145 ns	65 ns
RecSplit [14], ℓ=8, b=100	1.793	709 ns	75 ns
RecSplit [14], ℓ=14, b=2000	1.584	126 534 ns	96 ns
SIMDRecSplit, ℓ=5, b=5	2.96	49 ns	71 ns
SIMDRecSplit, ℓ=8, b=100	1.806	107 ns	80 ns
SIMDRecSplit, ℓ=14, b=2000	1.585	11 742 ns	110 ns

Another attempt at sophistication that so far failed is to combine brute force RecSplit with the retrieval approach of SicHash [30]. The idea of this *ShockHash* approach is to allow retrieval of a single bit of information for each element. The brute force part then tries pairs of random hash functions until they define a pseudo-forest – a collection of components consisting of a tree plus one additional edge. While ShockHash seems to allow space efficient perfect hashing, initial experiments indicated that performance-wise ShockHash is also inferior to pure brute force (see the extended version [5] for details). More efficient implementations of ShockHash may change this picture in the future.

Also, rotation fitting could be generalized by splitting into more than two parts. The resulting search for several rotations gives more room for sophistications like search space pruning. Furthermore, 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.

Everything discussed so far is mainly concerned with construction time. However, an equally important problem is to improve query time. Traversing an aggressively compressed tree for each query is inherently more expensive than the simple constant time operations

needed in PTHash [37] or SicHash [30] 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.

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. Better tuning of the SIMD variant for AMD or perhaps even a portable implementation that also works on ARM or RISC-V would be relevant for widespread application.

References

1 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering in-place (shared-memory) sorting algorithms. *ACM Trans. Parallel Comput.*, 9(1):2:1–2:62, 2022. doi:10.1145/3505286.

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 *FOCS*, pages 182–193. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00026.

4 Dominik Bez. Perfect hash function generation on the GPU with RecSplit. Master's thesis, Karlsruhe Institute for Technology (KIT), 2022. doi:10.5445/IR/1000152719.

5 Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders. High performance construction of recsplit based minimal perfect hash functions, 2022. doi:arXiv:2212.09562.

6 Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Perfect hashing for data management applications. *CoRR*, abs/cs/0702159, 2007. arXiv:0702159.

7 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.

8 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.

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 David Richard Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.

11 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.

12 Peter C. Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. Fast succinct retrieval and approximate membership using ribbon. In *SEA*, volume 233 of *LIPIcs*, pages 4:1–4:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SEA.2022.4.

13 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.

14 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.

15 Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT*, pages 75–88. ACM, 2014. doi:10.1145/2674005.2674994.

16 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.

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

18 Agner Fog. C++ vector class library. http://www.agner.org/optimize/vectorclass.pdf, 2013.

19 Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005. doi:10.1007/S00224-004-1195-X.

20 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.

21 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

22 GpuRecSplit – GitHub. https://github.com/ByteHamster/GpuRecSplit, 2023.

23 MPHF-Experiments – GitHub. https://github.com/ByteHamster/MPHF-Experiments, 2023.

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

25 Intel. Advanced vector extensions programming reference. https://www.intel.com/content/dam/develop/external/us/en/documents/36945, 2011.

26 Intel. Avx-512 instructions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html, 2013.

27 Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

28 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.

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

30 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.

31 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.

32 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.

33 Nvidia. Nvidia ampere GA102 GPU architecture. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2020.

34 Nvidia. CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2022.

35 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. doi:10.1016/j.jalgor.2003.12.002.

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

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

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

39 Sebastiano Vigna. Broadword implementation of rank/select queries. In *WEA*, volume 5038 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2008. doi:10.1007/978-3-540-68552-4_12.

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