Karlsruhe Institute of Technology

Perfect Hash Function Generation on the GPU with RecSplit

Master's Thesis of

Dominik Bez

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Peter Sanders
	Jun.-Prof. Dr. Thomas Bläsius
Advisors:	Dr. Florian Kurpicz
	M.Sc. Hans-Peter Lehmann

Time Period: April 1, 2022 – September 30, 2022

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, September 30, 2022

Abstract

Minimal perfect hash functions (MPHFs) map a static set S of arbitrary keys bijectively into the first $|S|$ natural numbers, i.e., each hash value is used exactly once. They are useful in many applications, for example, to implement hash tables with guaranteed constant access time. MPHFs can be very compact — less than 2 bits per key are possible. On the other hand, MPHFs are not able to decide whether a given key is in S or not. Currently, the most space-efficient practical MPHF is RecSplit. 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 key in less than 2 ms per key. This, however, is too slow for large inputs. In this thesis, we present new implementations of RecSplit that use multithreading, SIMD, and the power of GPUs to improve the construction time. Paired with our new *bijection rotation* technique, we achieve speedups of up to 333 for our SIMD implementation on an 8-core CPU, and up to 1873 for our GPU implementation compared to the original, sequential implementation of RecSplit. This enables us to construct an MPHF with 1.56 bits per key in less than 1.5 µs per key.

Deutsche Zusammenfassung

Minimale perfekte Hashfunktionen (MPHFs) bilden eine statische Menge S von beliebigen Schlüsseln auf die Menge der ersten $|S|$ natürlichen Zahlen bijektiv ab, d. h., jeder Hashwert wird exakt einmal verwendet. Sie sind in vielen Anwendungen hilfreich, zum Beispiel, um Hashtabellen mit garantiert konstanter Zugriffszeit zu implementieren. MPHFs können sehr kompakt sein — weniger als 2 Bit pro Schlüssel sind möglich. Andererseits sind MPHFs nicht in der Lage zu entscheiden, ob ein gegebener Schlüssel zu S gehört. Zurzeit ist RecSplit die speichereffizienteste MPHF. RecSplit bietet verschiedene Kompromisse zwischen Platzverbrauch, Konstruktionszeit und Anfragezeit an. RecSplit kann zum Beispiel eine MPHF mit 1.56 Bits pro Schlüssel in weniger als 2 ms pro Schlüssel konstruieren. Das ist jedoch zu langsam für große Eingaben. Diese Arbeit präsentiert neue RecSplit-Implementierungen, die Multithreading, SIMD und die Leistung von GPUs nutzen, um die Konstruktionszeit zu verbessern. Gemeinsam mit unserer neuen *bijection-rotation*-Methode erreichen wir Beschleunigungen um Faktoren bis zu 333 für unsere SIMD-Implementierung auf einer 8-Kern CPU und bis zu 1873 für unsere GPU-Implementierung verglichen mit der originalen, sequenziellen RecSplit-Implementierung. Dadurch können wir MPHFs mit 1.56 Bits pro Schlüssel in weniger als 1.5 µs pro Schlüssel konstruieren.

Acknowledgments

I thank my advisors Dr. Florian Kurpicz and M.Sc. Hans-Peter Lehmann for providing this interesting topic and giving me detailed feedback on my thesis. I thank them and Prof. Dr. Peter Sanders for suggesting fruitful ideas. For reviewing this thesis, I thank Prof. Dr. Peter Sanders and Jun.-Prof. Dr. Thomas Bläsius. Finally, I thank Lucas Alber, Julius Häcker, Anne Herrmann, and Wendy Yi for proofreading my thesis.

This work was performed on the computational resource bwUniCluster funded by the Ministry of Science, Research and the Arts Baden-Württemberg and the Universities of the State of Baden-Württemberg, Germany, within the framework program bwHPC.

Contents

1. Introduction

Given a set S of m arbitrary keys, a *minimal perfect hash function (MPHF)* $h\colon S \to \{0, 1, \ldots, m-1\}$ maps each key in S to a distinct number between 0 and $m-1$, i.e., h is a bijective function. It can be used to implement a hash table as a simple array since no conflicts of the hash values can occur. The MPHF h is a static data structure which is constructed using S. The crux of minimal perfect hash functions is that they can be made very space efficient. In fact, practical MPHFs only need 1.56 bits per key whereas the theoretical lower bound is $\log_2(e) \approx 1.44$ bits per key [46, 51].

This advantage comes at a price. The MPHF h cannot be used to check whether a given key x belongs to S since that would require to store the set S. If $x \notin S$, it will just return an arbitrary number. Therefore, MPHFs are only applicable in situations where it is either already known whether $x \in S$ or it is not a problem if $x \notin S$ and h is used anyway. The latter situation might be improved by using an *approximate membership query (AMQ) filter* (e.g., a Bloom filter [34]) to filter out most keys not belonging to S. AMQ filters can be very space efficient as well.

A typical application of MPHFs is to exploit the memory hierarchy. Assume you want to store a large static hash table of the keys S which does not fit in the main memory. This may happen for example if the keys in S are long strings. By constructing an MPHF h of S which fits in main memory, the hash table can be stored on the disk as a simple array A. At each index i of the array, the key x in S with the hash value $h(x) = i$ is stored alongside the corresponding information which is meant to be stored in the hash table.

Given an arbitrary key y, h can be used to calculate the correct index j in A before loading the corresponding information $A[j]$ from the disk. It can then be checked whether y equals the key stored at $A[j]$. If not, then $y \notin S$. By using an AMQ filter, most disk accesses for keys not in S can be avoided. Using an MPHF may also be useful for other levels of the memory hierarchy where h fits in a smaller but faster memory than A. For example, h might fit in the cache and A in main memory or h fits on the local disk whereas A can only be accessed over a network.

Using a complicated AMQ filter on top of the MPHF is not necessary. In fact, the MPHF itself in conjunction with an additional array can be used as an AMQ filter. The hash value j of the key y can be used as an index into an array B which contains *fingerprints* of all keys in S. A fingerprint is just a word with ω bits produced by a random hash function. The key y is not in S if the fingerprint stored at $B[j]$ is different from the fingerprint of y. The false positive rate ϵ is the probability that the two fingerprints match, but $y \notin S$. If

each fingerprint contains ω bits and assuming the used hash function is indistinguishable from a random function, then $\epsilon = 2^{-\omega}$. This is equal to the theoretical lower bound [36]. In conclusion, MPHFs can be used to implement static AMQ filters, and if an MPHF is required anyway, this is optimal in terms of the tradeoff between ω and ϵ.

To fulfill the aforementioned purposes efficiently, it is usually required that the MPHF can be queried in constant time and constructed in linear time. These requirements are often relaxed to include expected running times. The space requirement, query time and construction time are the main metrics to evaluate and compare different minimal perfect hash function techniques.

1.1 Contribution

Currently, the most space-efficient practical MPHF is RecSplit [46]. It can reach 1.56 bits per key while being competitive for query and construction time compared to other techniques. RecSplit needs less than 2 milliseconds per key to construct such a space-efficient MPHF. However, this amounts to about 50 hours for 100 million keys which stands to question the practicability of RecSplit for large inputs if such space efficiency is desired. Fortunately, the RecSplit construction is an embarrassingly parallel problem. The keys are first divided into buckets which can be processed independently of each other. This opens the door for parallelization. A tree is constructed for each bucket. For every node in this tree, several normal hash functions are tried on the keys in the node to find a hash function which satisfies a specific property. Trying out the hash functions can be accelerated using *single instruction, multiple data (SIMD)*.

Nevertheless, there is no public parallel implementation of RecSplit yet. This thesis is concerned with implementing the RecSplit construction for modern parallel hardware to fully utilize its processing power. This is achieved by creating two new implementations. The first uses multithreading and SIMD to speed up the construction on *Central Processing Units (CPUs)*. We refer to it as the *SIMD implementation* or *SIMDRecSplit*. The second implementation uses the processing power of *Graphics Processing Units (GPUs)*. This requires the presence of such a device but can speed up the construction significantly. The second implementation is called *GPU implementation* or *GPURecSplit*. Both are compared to the *original implementation* which is provided by the Sux library by Vigna [13].

1.2 Outline

In the next chapter, we introduce necessary foundations, including the functionality of GPUs. Chapter 3 presents important perfect hash function techniques. However, it is not necessary to understand them to comprehend this thesis. RecSplit, the MPHF technique this thesis parallelizes, is explained in detail in Chapter 4. We describe the implementation details of our new RecSplit implementations in Chapter 5. The result of our endeavor is examined experimentally in Chapter 6. Finally, this thesis is summarized in Chapter 7 along with some hints for future work.

2. Preliminaries

We provide the foundation to understand the rest of this thesis in this chapter.

2.1 Notation

This section introduces notions and notation that are used throughout this thesis.

Word RAM

In this thesis, the word random-access machine (word RAM) model is used [52]. Such a machine operates on words of w bits. Arithmetic and bitwise operations on these words can be executed in constant time.

General Notation

The notation for the set of natural numbers from including 0 to excluding n is $[n] = \{0, \ldots, n-1\}$. Rounding a up to the next integer is denoted as $\lceil a \rceil$, rounding down is denoted as $\lfloor a \rfloor$. The operator $\leftarrow$ is used to assign a value to a variable, i.e., $a \leftarrow a + b$ assigns the value $a + b$ to a. The old value of a is used for the addition and then overwritten. On overflow, values are reduced modulo 2^w where w is the number of bits in a word, i.e., only the w least significant bits are stored while the remaining most significant bits are discarded (the value "wraps around").

Shift Operator

The bits of a word a can be shifted n bits to the left with the syntax $a << n$ and n bits to the right with $a >> n$. The empty spots are filled with 0-bits. If no overflow occurs, a left shift by n bits is equivalent to a multiplication with 2^n. Similarly, $a >> n$ is equivalent to $\lfloor \frac{a}{2^n} \rfloor$.

Bitwise Operators

There are various bitwise operations that manipulate the bits of words. The bitwise OR-operator `|` sets a bit at a given position if and only if at least one of the operands has a 1-bit at the same position. For example, $3 \mid 6 = \texttt{011}_2 \mid \texttt{110}_2 = \texttt{111}_2 = 7$, where $_2$ denotes binary numbers. Similarly, $\oplus$ computes the bitwise exclusive OR (XOR) and `&` computes the bitwise AND.

Rotation Operator

The least significant m bits of a word can be rotated left n bits with the rotation operator $\texttt{rot}_m^n$. By assuming the bits other than the lower m bits of a are zero, it can be implemented as

$$\texttt{rot}_m^n(a) = ((a << n) \mid (a >> (m-n))) \,\&\, ((1 << m) - 1).$$

Popcount

The popcount operation counts the number of 1-bits in a word. For example, the popcount of $25 = 11001_2$ is 3. The SSE4.2 instruction set extension of the x86 instruction set contains a popcount instruction for 64-bit values and is available on most modern x86 CPUs [21, 24].

Find Least Significant 1-Bit / Count Trailing Zeros

The number of trailing zeros of a word is equivalent to the index of the least significant 1-bit. We denote the number of trailing zeros of a as $\rho(a)$. The x86 instruction set contains instructions to calculate this value in constant time [21].

Delete Least Significant 1-Bit

The operation to set the least significant 1-bit to zero is called `clear_rho`. The Bit Manipulation Instruction Set 1 (BMI1), an instruction set extension of x86, contains an instruction to perform this operation directly [21]. Alternatively, it can be implemented as $\texttt{clear\_rho}(a) = a \,\&\, (a-1)$.

Find N-th Least Significant 1-Bit / Selection

The operation to find the n-th least significant 1-bit of a word is called ρ_n. Note that $\rho_0 = \rho$. This operation can be implemented by calling `clear_rho` n times before calling ρ:

$$\rho_n(a) = \rho(\texttt{clear\_rho}^n(a))$$

However, it can be implemented with just three instructions using a bit shift, the parallel bits deposit instruction (PDEP) from the Bit Manipulation Instruction Set 2 (BMI2) [21] and ρ [46]. It is not necessary to understand details of this implementation in the context of this thesis. Thus, we do not explain it further.

Speedup

An important concept to compare the performance of two algorithms is called *speedup*. Let $T(A)$ be the time algorithm A needs and $T(B)$ the time algorithm B needs for the same task. The speedup $S(A,B)$ of B compared to A is calculated as $S(A,B) = \frac{T(A)}{T(B)}$. This means algorithm A needs a factor of $S(A,B)$ longer than B. The speedup can be smaller than 1, in which case A is faster. Speedups are used especially for parallel algorithms, i.e., to compare a parallel algorithm with a given number of threads to a sequential algorithm.

2.2 SIMD

A lot of code operates on *scalar* values. For example, a piece of code may add the two integers a and b. Sometimes, the same operation must be done on a set of values, e.g., adding the values of two arrays pointwise. This can be achieved with a simple loop. However, the corresponding instructions must be decoded by the hardware for every element and only one element is processed at a time. This can be improved by using single instruction, multiple data (SIMD) [48]. A single instruction is used to apply the same operation on

several elements. This means instead of adding two scalar values, two *vectors* containing multiple integers each are added. This reduces the number of instructions that need to be decoded and allows the hardware to process the whole vector in parallel. Using SIMD can yield high speedups in many situations, e.g., when adding two arrays.

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), blending the contents of two vectors or other advanced operations. The exact set of operations depends on the concrete implementation of the SIMD model.

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 32 lanes with 16 bits each, i.e., the vector contains 512 bits overall. It is advisable to use as many lanes as possible to maximize throughput. This may include decreasing the size of each lane if it allows for using more lanes. The act of utilizing SIMD is also called vectorization. There are several possibilities to vectorize a given piece of code [25]:

- Write it directly in assembly language to utilize SIMD instructions of the given hardware. This is tedious and error prone since many low-level details must be considered. However, it provides direct control over the hardware. An experienced assembly programmer may produce faster code than any of the other ways of vectorization can. The assembly can be used as the input of an assembler or be embedded in a high-level language as inline assembly.
- Use intrinsics. They act like a function in a high-level programming language but often compile to a single specific instruction. This frees the programmer of considering low-level details like register allocation while enabling the use of specific instructions. Moreover, the compiler can fully optimize a piece of code containing intrinsics which it cannot do with inline assembly.
- Rely on auto vectorization. An optimizing compiler may be able to vectorize a simple loop to improve its performance. For example, a compiler may optimize a loop which adds two arrays pointwise. However, this is not always possible and often results in suboptimal code. In the example, the compiler may not know the size of the arrays. It has to generate code to process some elements in a scalar form since the number of lanes in a SIMD vector may not divide the number of elements in each array. This can produce a large amount of machine code. On the other hand, the programmer may know that it divides without a remainder — possibly because a padding was introduced specifically for this purpose.
- Using a library. This may be an optimized library for a specific purpose which happens to use SIMD (e.g., for matrix multiplication) or a more general library which simplifies vectorization with a more readable syntax than intrinsics and by providing additional functionality (e.g., the Vector Class Library [26]; see below).

2.2.1 Advanced Vector Extensions (AVX)

The Advanced Vector Extension (AVX) [14] are instruction set extensions to the instruction set architecture x86 which is used by many Intel and AMD processors. These extensions provide many instructions for vectors containing 256 or 512 bits, depending on the version. There are several versions of AVX where each version adds more instructions to the last version or increases the vector size. For most floating-point instructions, 32-bit and 64-bit lanes can be used. Many integer instructions allow for lanes of size 8, 16, 32 and 64 bits.

The first version of AVX does not contain integer instructions and is therefore not relevant for our SIMD implementation. Instead, our SIMD implementation is optimized for CPUs

supporting AVX2 and AVX-512 [15]. AVX2 allows many integer operations on 256-bit vectors. AVX-512 extends these operations to 512-bit vectors and adds many new instructions, e.g., for using masks to mask out specific lanes. However, not every CPU supporting AVX-512 supports all instructions. 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. Without this extension, popcount is not available for vectors of any bit size.

2.2.2 Vector Class Library

The Vector Class Library (VCL) is a C++ library written primarily by Fog [26, 27]. It encapsulates SIMD vectors as objects which simplifies the syntax compared to intrinsics since it avoids the necessity to provide the size of the vector and the lanes for each operation. Furthermore, it provides an infix notation for many typical operations like addition and XOR. The Vector Class Library is especially useful for advanced purposes like per-element branches, finding the first lane which satisfies a specific property or looking up values in a lookup table.

Perhaps the most compelling feature compared to writing intrinsics is the portability. The Vector Class Library contains vectors of size 128, 256 and 512 bits and specifies operations on them. If no instruction set is available which includes vectors of a given size, VCL will emulate it using smaller vectors. For these reasons, we use the Vector Class Library for our SIMD implementation. As a minimum, VCL requires the SSE2 instruction set extension of x86 and can utilize newer x86 extensions, including AVX-512. Unfortunately, other instruction sets like ARM are not supported.

The Vector Class Library has no mean of querying the native vector size, i.e., 256 bits for AVX2 and 512 bits for AVX-512. We have implemented such a functionality at compile time. This is useful in our case since we want to use as large vectors as possible to maximize throughput but with as little overhead as possible. We have to try out hash functions in a SIMD manner until a hash function is found which satisfies a specific property. This discourages us from always using 512-bit vectors and relying on the emulation of VCL if no AVX-512 is available because a fitting hash function may have already been found in the first half, but the emulation continues processing the second half. Therefore, we use 512-bit vectors if AVX-512 is available and 256-bit vectors otherwise. If even AVX2 is not available, we rely on the emulation of VCL.

Not every instruction is supported by the Vector Class Library. This is especially true for instructions which are only available in newer instruction set extensions and are hard to emulate on older extensions. Fortunately, the VCL vector objects can be converted to the intrinsic vector types which are used for the intrinsics, both implicitly and explicitly without any overhead. This allows for the use of intrinsics directly on VCL objects. We make use of this technique to use popcount if AVX512VPOPCNTDQ is available and left bit shift, where the number of bits to shift is different for each lane and provided in a second vector, if at least AVX2 is available.

2.3 GPUs

Graphics Processing Units (GPUs) are specialized processors initially designed for computer graphics applications. Many operations in computer graphics need to be done to a large number of elements, e.g., for every pixel of an image or every vertex of a polygon mesh. This allows for parallelization using SIMD. Over the last decades, GPUs evolved to general purpose processors for highly parallelizable tasks. They can be programmed using special

programming languages and *Application Programming Interfaces (APIs)*. One such API is CUDA [71, 3]. It is developed by Nvidia and can be used to program Nvidia GPUs. We use CUDA to implement GPURecSplit and use CUDA terminology throughout this thesis.

The information in this section is based on the CUDA C++ Programming Guide [3] and the Nvidia Ampere GA102 GPU Architecture white paper [17].

2.3.1 Structure of a GPU

In this section, the structure of a typical modern GPU is explained. The exact structure may differ between different models. To provide a grasp of the dimensions of a modern GPU, the exact numbers for many metrics of the Nvidia RTX 3090 [17] are mentioned. This GPU is used in the experiments in Chapter 6. A GPU consists of several *streaming multiprocessors (SMs)* (RTX 3090: 82). Each SM contains many *arithmetic logic units (ALUs)* to perform computations. The RTX 3090 has 128 ALUs for 32-bit floating-point math per SM, but only 64 ALUs for 32-bit integer math per SM. This amounts to a large number of ALUs compared to a CPU.

Single Instruction, Multiple Threads (SIMT)

GPUs are SIMD machines. More specifically, they are *single instruction, multiple threads (SIMT)* machines. The main difference is the notion of a thread. In SIMD terminology, a thread is a stream of operations which may include SIMD operations that operate on vectors containing several lanes. In SIMT terminology, there is a single thread for each SIMD lane without a "superior" thread responsible for a whole vector. To ensure the same advantages as SIMD, 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*.

Masking is used to enable the execution of different instructions for different threads of a warp, e.g., to implement control flow like the `if` statement. Masking means every thread of the warp executes each instruction, but they are *inactive (masked out)* for instructions they should not execute, e.g., because the condition in the `if` was false. It is called *divergence* when some threads of a warp are inactive. Divergence should be avoided as often as possible since it reduces performance. Note that for an `if-else` where at least one thread in the warp has a condition which is true and another thread has a condition which is false, each thread of the warp has to execute both cases and is inactive in the wrong case. Even worse, in loops each thread in the warp has to iterate as many times as the thread with the largest number of iterations.

Hardware Multithreading

Since a GPU has many ALUs (RTX 3090: 5248 32-bit integer ALUs), many threads are necessary to utilize them. But a number of threads matching the number of ALUs is not sufficient. Each operation has a latency, which is especially high for memory operations. While a thread waits for the completion of its last operation, it cannot utilize an ALU. Modern CPUs try to decrease the latencies as much as possible and optimize the time a single thread needs by using low-latency memory, fast and large caches, deep pipelines, superscalar techniques, branch prediction and out-of-order architectures. GPUs, on the other hand, do not have to decrease the time each thread takes. Their goal is to maximize throughput, i.e., process as many threads as possible in a given amount of time, irrespective of the time a single thread takes.

So instead of decreasing latencies, GPUs overcome latencies by scheduling another warp while a warp waits on the previous operation to finish. This is called *latency hiding* and means each SM is oversubscribed with many more threads than ALUs. To avoid any

overheads when switching between different warps, each SM contains many 32-bit registers (RTX 3090: 65 536) and each thread keeps its values in its allocated registers until it is finished. In each clock cycle, several (RTX 3090: up to four) warps are scheduled. These warps must be ready to be scheduled, i.e., their next operation cannot depend on a previous unfinished operation.

The maximum number of threads per SM can be very high (RTX 3090: 1536, i.e., 12 times the number of 32-bit floating-point ALUs). To hide high memory latencies, the number of threads should optimally reach this number. Some modern CPUs have a similar oversubscription of threads as well to better utilize the resources of each CPU core in a multithreading context. This is often called *simultaneous multithreading (SMT)* or *hyper-threading*. In many cases, only two threads per core are possible.

Global Memory

The global memory is the largest and slowest memory on the GPU. It is usually several gigabytes large on modern GPUs (RTX 3090: 24 GB). Since GPUs use latency hiding, global memory is not optimized to minimize latency but to maximize bandwidth instead (RTX 3090: 936 GB/s bandwidth). This is important to serve the many threads that can be resident on the GPU at the same time (RTX 3090: 125 952).

To use this bandwidth as efficiently as possible, each memory access should be *coalesced*. Consider a warp of 32 threads where each thread accesses a word in global memory. The hardware will serve these requests with as few memory transactions as possible where each transaction transfers 32, 64 or 128 bytes (the exact specifics depend on the model). These transfers are aligned, i.e., the address of the first byte is a multiple of the transaction amount. 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.

Caches

A GPU may contain several caches to reduce latency and increase bandwidth for frequently accessed data. The RTX 3090, for example, has a total of 6144 KiB of L2 cache. It also has a faster L1 cache which is placed on each SM, i.e., the threads resident on the same SM share an L1 cache. The size of the L1 cache is up to 128 KiB per SM, but a part of it may be used as shared memory (see below).

Shared Memory

Shared memory is a fast memory usually placed on each SM. A group of threads that are resident on the same SM can use shared memory as a manually managed cache or to share data. It can also be used as a mean to store data which should be accessed with an index (i.e., an array, which is not possible with registers) without using the slow global memory. A typical application is to load the data on which a group of threads operates on into shared memory. On the RTX 3090, shared memory is part of a unified data cache containing 128 KiB. The amount of shared memory is automatically configured as 0, 8, 16, 32, 64, or 100 KiB, and the remainder is used as the L1 cache.

Bank Conflicts

The data in shared memory is partitioned into 32 memory banks. This number may be different for other GPU models, but it is correct for all CUDA devices of the past 10 years. The memory is divided into the banks in 32-bit chunks. This means the first 32 bits are in the first bank, the next 32 bits are in the second bank and so forth. The 33rd 32-bit chunk is again in the first bank. When n threads of the same warp access different words

within the same bank in the same instruction, an *n-way bank conflict* occurs. This means the accesses must be serialized and the whole warp has to wait. This has an impact on performance. If several threads read the same word, it is not a bank conflict since the word can be broadcast to the different threads.

Bank conflicts happen especially for *strided* memory access. Consider a 32x32 matrix stored row by row in shared memory. Assume each thread of a warp of 32 threads iterates over one row, where the first thread iterates over the first row, the second thread over the second row and so forth. Let each element in the matrix be a 32-bit word. Since the size of each word equals the size of each word in a bank and the width of the matrix equals the number of banks, each column of the matrix is stored in one bank. In every iteration, each thread accesses a different word in the same bank — a 32-way bank conflict! The name *strided* comes from the fact that the accesses of each thread in the warp are a specified number of bytes (here 128) apart which may lead to bank conflicts, especially if the stride is a power of two.

2.3.2 CUDA

CUDA is an API to program Nvidia GPUs for general-purpose computations. It offers integration into programming languages like C++ to simplify GPU computations. The CUDA extension of C++ is called CUDA C++ and is used in our GPU implementation. The GPU is called *device* in CUDA and the system containing the GPU (the CPU, main memory) is referenced as *host*.

Kernels

Functions which can be executed on the device are named *kernels*. They are defined and called similar to normal C++ functions by the host. To call the kernel `kernelTest` with the function arguments `arg0` and `arg1`, the syntax `kernelTest<<<b, t>>>(arg0, arg1)` can be used. This creates a *grid* of `b` *thread blocks* with `t` threads per thread block. Every thread in the grid executes the code in `kernelTest`. Grids and thread blocks can also be two- or three-dimensional, but this is not relevant for this thesis.

Every thread has an ID which it can use as a constant, for example as an array index. This is what allows different threads to operate on different data. A thread block is divided into warps in the natural order of the threads. Assume a warp size of 32. The first 32 threads of the thread block belong to the same warp, the next 32 thread to another warp and so forth. The size of a thread block should be a multiple of the warp size to avoid unused threads within a warp.

The threads within a thread block are guaranteed to reside on the same SM. This enables them to cooperate. In particular, they can *synchronize*, and they have access to the same shared memory. Synchronizing is achieved by calling a special synchronizing function. After calling the function, the calling thread can only continue after all threads in its thread block have called the same function. The function guarantees that all memory operations to shared or global memory before the synchronization are visible to all threads in the thread block after synchronization.

Calling a kernel is also called *launching* and is an asynchronous operation. This means that after a kernel is launched by the host, the host can immediately continue with its work without waiting for the kernel to finish. The CUDA runtime takes care of transferring the code and parameters to the device. To use the results produced by the kernel, the host can synchronize with the device to wait for the kernel to complete.

Streams

Similar to kernel launches, transferring data between host and device memory can be done asynchronously. When launching several kernels and data transfers before synchronizing with the device, all the operations are done in order. For example, when launching kernel `kernelA` before `kernelB`, `kernelB` will only start execution after `kernelA` is completely finished. This is important to ensure correctness if `kernelB` depends on the results of `kernelA`.

However, sometimes different kernels and data transfers are completely independent and could be executed at the same time if sufficient resources are available. This can be achieved using *streams* in CUDA. 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.

Asynchronous Data Copies

As stated previously, data can be transferred asynchronously between host and device memory. Another possible asynchronous memory copy is between global memory and shared memory using `memcpy_async`. The threads of a thread block can cooperate to kick off an asynchronous transfer to their shared memory. While the data is transferred, the threads can continue with other computations. The threads have to use a special synchronization function before using the loaded data. This function blocks until the data is available. Using `memcpy_async` can hide the high latency of the data transfer and avoids using an extra register per thread as an intermediary between global and shared memory.

Compute Capability

Different GPU models support different CUDA *compute capabilities*. Newer models usually support higher compute capabilities. Higher compute capabilities provide more advanced functions and may have larger caches, shared memory, more registers per SM, etc. For example, the `memcpy_async` API explained in the last paragraph only has hardware acceleration for devices with compute capability 8.0 or higher. Other devices emulate the behavior by loading the data from global memory into registers before storing it in shared memory. The threads can only continue after all data has been loaded, i.e., the memory copy is not asynchronous. The RTX 3090 has compute capability 8.6.

2.4 Prefix Sum

Let x_i for $i \in [k]$ be a sequence of numbers. We define the *inclusive prefix sum* $\hat{X}_j$ as $\hat{X}_j = \sum_{i=0}^{j} x_i$ and the *exclusive prefix sum* X_j as $X_j = \sum_{i=0}^{j-1} x_i$. This means for each j, they indicate the sum of all previous elements (the prefix), where x_j is only included in the inclusive prefix sum. Note that $X_0 = 0$. The inclusive prefix sum can be calculated from the exclusive prefix sum by shifting the values to the left, i.e., $\hat{X}_j = X_{j+1}$.

Prefix sums are used frequently in algorithms and data structures. We store part of the data for our MPHF in a prefix-sum form. If the distinction is irrelevant or clear from the context, we omit whether the prefix sum is inclusive or exclusive.

2.5 Pareto Front

We use the notion of *Pareto fronts* in our visualizations. To understand the concept, consider the following definitions. Let $A \subseteq \mathbb{R}^n$ be a subset of the n-dimensional Euclidean

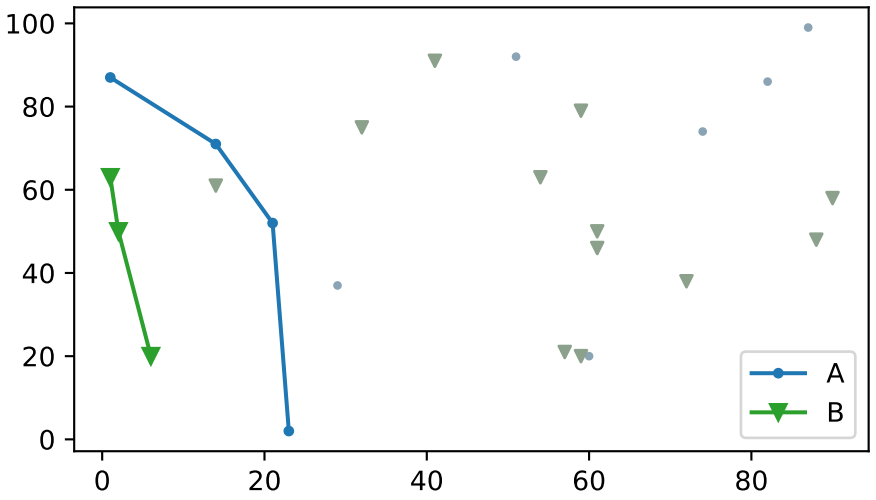


Figure 2.1: This plot shows the Pareto fronts of two sets "A" and "B". All blue points which are not on the blue line are dominated by the four points on the blue line, and the green triangles not on the green line are dominated by the green line.

space and $x, y \in A$. The point x is said to *dominate* the point y if $x_i \leq y_i$ for all $i \in [n]$. We call x *Pareto optimal* if no other point in A dominates x. The *Pareto front* is the set of all Pareto-optimal points of A. A visualization of Pareto fronts is shown in Figure 2.1. The Pareto front is basically the bottom-left border of all points belonging to the same set. We use Pareto fronts to show the tradeoff between two metrics, such as time versus space consumption.

3. Related Work

This chapter explains several important minimal perfect hash functions. This is meant as an overview to place RecSplit, the MPHF this thesis is concerned about and which is explained in the next chapter, in the landscape of minimal perfect hash functions. We also present some applications of MPHFs in Section 3.6.

3.1 GOV

Like other authors before them [68, 38, 35], Genuzio et al. (referred to as GOV) [54] use random linear systems to construct minimal perfect hash functions. Let n be the number of keys in S, $m = (1+\epsilon)n$ for $\epsilon > 0$, $\mathbb{F}_2$ the finite field with two elements, and $h_\theta \colon S \to \mathbb{F}_2^m$ a hash function from a family of hash functions (indexed with θ) such that $h_\theta(x)$ contains exactly r ones and $m - r$ zeros for each $x \in S$. The number r is a constant. Write the hash values of all keys in S as the rows of a matrix $H \in \mathbb{F}_2^{n \times m}$. If ϵ is large enough, H can be transformed into row echelon form with high probability, e.g., with Gaussian elimination. In this case, each row (i.e., key in S) can be assigned a unique column (i.e., value in $[m]$). We just have to store which of the r ones in $h_\theta(x)$ is the correct one. If the transformation into row echelon form fails, try again with a different θ.

To store these values in a compact form which allows for constant-time access, a similar idea is used. Generally, to store the static function $f \colon S \to [2^b]$, the linear equation

$$HW = V \text{ with } H \in \mathbb{F}_2^{n \times m}, W \in \mathbb{F}_2^{m \times b}, V \in \mathbb{F}_2^{n \times b},$$

where V contains the values $f(x)$ in its rows, is solved for W. The matrix W is stored directly. To compute $f(x)$, $h_\theta(x)$ reveals the r rows of W that must be combined with XOR to get $f(x)$.

To summarize, H is transformed into row echelon form to compute a number $j \in [r]$ for each key x that tells us which 1 in $h_\theta(x)$ is at the position of the final hash value. The matrices H and V (which contains j as a row) are used in a linear equation to solve for W which is stored. It can be used to retrieve j, which then tells us the final hash value. However, the final hash value is in $[m]$, not $[n]$. This means that the structure so far is a perfect hash function, but not a *minimal* perfect hash function. To make the perfect hash function minimal, a ranking structure can be used. Given a number $k \in [m]$, the ranking structure can be used to calculate the number of actually used hash values smaller than k in constant time. This calculated number is then used as the minimal perfect hash value.

In previous work, ϵ is chosen such that the linear equation can be solved in linear time using hypergraph peeling. Genuzio et al. [54] use a smaller ϵ, but have to deal with the fact that Gaussian elimination has running time $O(n^3)$. The first step is to use a hash function to distribute the input into smaller sets and compute the MPHF on each set separately. This reduces the running time to $O(n)$. For practical performance, their contribution is using *broadword programming* which is similar to SIMD, but within a single word, and an improved Gaussian elimination algorithm they call *lazy Gaussian elimination*.

The GOV MPHF can achieve 2.24 bits per key. As the authors of RecSplit have shown [46], RecSplit can produce a smaller MPHF with faster query time, at the cost of a longer construction time. Using the improved implementations of RecSplit proposed in this thesis, RecSplit can outperform GOV in all three major metrics.

3.2 CHD

The compressed hash-and-displace (CHD) algorithm [31] is based on hash-and-displace by Pagh [73]. First, the keys are distributed into buckets of small expected size using a hash function. These buckets are then handled in decreasing order of size. For each bucket, a family of hash functions is used to brute-force search for a hash function that is injective on the current bucket and does not use a hash value that is already used by a previous bucket. The index of the hash function of each bucket is stored in a compressed array using a technique by Fredriksson and Nikitin [53]. To ensure expected linear construction time, the codomain of the hash functions is $[m]$ with $m = (1 + \epsilon)n$ and $\epsilon > 0$. That means the result is only a perfect hash function, not an MPHF. Similar to GOV, a ranking data structure is used to obtain an MPHF.

CHD can construct MPHFs with about 2 bits per key in practical construction times and can theoretically reach the lower bound of 1.44 bits per key. The authors of RecSplit [46] have shown that RecSplit dominates CHD in the main metrics construction time, query time, and space consumption with large margins. From a practical point of view, CHD can be considered obsolete.

3.3 BBHash

The authors of BBHash [66] provide an efficient, parallel implementation of the fingerprinting idea [69]. Given n_0 keys S_0, a hash function $h_0 \colon S_0 \to [n_0]$ is used on all keys, and only those n_1 keys that have the same hash value as another key are put into the set S_1. The remaining keys are finished processing. For each finished key, a bit in a bit array of n_0 bits is set to 1. The keys in S_1 are then processed in the same way using the hash function $h_1 \colon S_1 \to [n_1]$, and a bit array of n_1 bits. This procedure is continued until all keys are finished.

To query BBHash with a key x, $h_0(x)$ is used to find the bit in the first bit array. If this bit is 1, then the final hash value is $h_0(x)$. Otherwise, $h_1(x)$ is used to find the bit in the second bit array, and if it is 1, then the final hash value is $n_0 + h_1(x)$, and so forth. Similar to GOV and CHD, the result is only a perfect hash function, not a minimal perfect hash function. To get an MPHF, the bit arrays are concatenated, and a ranking data structure is used on them.

The advantage of BBHash over the other algorithms is that construction and queries are fast. However, BBHash needs more than 3 bits per key which is significantly more than the other algorithms considered in this thesis. The authors argue that in many use cases, the difference between 1.44 and 3 bits per key is insignificant compared to the space consumption of the actual data that is indexed by the MPHF. RecSplit [46] can

achieve a smaller MPHF with a similar construction time and slower queries than the most space-efficient configuration of BBHash. The construction of RecSplit is slightly slower for $n < 10^8$, but faster otherwise. With the improvements in this thesis, especially fixing a performance bug of the RecSplit query in Section 6.2, we believe that RecSplit can dominate this configuration of BBHash by choosing the right parameters.

3.4 PTHash

PTHash [76] is based on FCH [50] which can be considered a predecessor of the hash-and-displace technique mentioned in Section 3.2. As in the other techniques discussed so far, the keys are first distributed into different buckets using a hash function, but unlike before, the distribution is not uniform. Specifically, about 60% of the keys are mapped to 30% of the buckets. First, a seed s is chosen such that there is no hash collision in any bucket using the pseudorandom hash function h_s. The buckets are then processed in order by decreasing size. For each bucket B_i, an integer k_i is searched such that the position

$$(h_s(x) \oplus h_s(k_i)) \bmod n$$

for each $x \in B_i$ is yet unused. The found k_i is stored in an array indexed by i. This array is compressed using one (or two, because the buckets are partitioned into two arrays) of several possible compression schemes [45, 47, 53], giving different tradeoffs of query time versus space consumption.

The proclaimed goal of PTHash is fast query times. Using an appropriate compression scheme, only a single memory access is required to find the value k_i, and the remaining operations are simple hash function evaluations and arithmetic. The authors compared PTHash [76] to FCH [50], CHD [31], GOV [54], BBHash [66], RecSplit [46], and EMPHF, a technique based on hypergraph peeling [29]. The only technique with comparable query times is FCH, but PTHash can be constructed significantly faster and can be configured to require less space. Compared to RecSplit, PTHash consumes 0.5 to 2.5 more bits per key, but has two to four times faster queries and can be constructed in less time. According to more recent results [75], PTHash seems to dominate BBHash in all three major metrics. While RecSplit needs fewer bits per key and can be accelerated a lot using the techniques in this thesis, it cannot reach the query times of PTHash.

3.5 GPU Techniques

To the best of our knowledge, there is no technique that constructs an MPHF on the GPU yet. The literature we could find that combine GPUs and perfect hashing are either an application of MPHFs or provide an MPHF to be queried on GPUs. For example, perfect spatial hashing [63] is a perfect hashing technique for two- or three-dimensional spatial data. Points that are close to each other are queried coherently to allow for memory coalescing and therefore efficient queries on GPUs. The construction, however, is performed on the CPU.

Weaver and Heule proposed a SAT-based MPHF construction [81] which achieves about 1.83 bits per key with practical construction time. It is dominated by RecSplit [46] in the three major metrics, but it is still theoretically interesting. By combining the SAT-based approach with SAT solvers capable of using the GPU [72, 74], it should be possible to construct an MPHF using the GPU. Nonetheless, we expect that RecSplit would dominate the SAT-based approach even in this case.

3.6 Applications

As discussed in Chapter 1, a typical application of MPHFs is to exploit the memory hierarchy for efficient hash tables by storing the MPHF in a faster but smaller memory than the actual hash table. We have also seen that approximate membership query filters can be implemented using perfect hash functions. This has been evaluated in practice by Graf and Lemire [58]. There are also more specialized applications of perfect hash functions such as hypergraph algorithms [33], compressed text indexing [32], prefix searching [30], databases [37, 39], networks [67], machine learning [77], and bioinformatics [40].

4. RecSplit

In this chapter, we describe the MPHF *RecSplit* [46]. The first step of the RecSplit construction is applying an *initial hash function* on every element of the input to receive keys of the same length (see Section 4.1). These keys are distributed into different buckets of constant expected size using a *bucket-assignment function* (see Section 4.2). For each bucket, a *splitting tree* is constructed (see Section 4.3). Each inner node of this tree is brute-force searching for a *splitting* — a hash function which distributes the keys to the child nodes in a specific manner. Each leaf node only has a few keys. An MPHF on m keys of a leaf can be computed by brute-force searching for a *bijection* from the keys into $[m]$.

A RecSplit instance can be queried for the hash value of an element by first applying the initial hash function to get the corresponding key x. The bucket-assignment function is used on this key to find the correct bucket. The splitting tree in this bucket is traversed from the root to the leaf which contains x by applying the splittings stored at each node. In each step, the number of keys with a smaller hash value are accumulated, i.e., the number of keys in all buckets with smaller IDs and for each splitting the number of keys in child nodes which are left from the child node which contains x. This accumulated value is added to the result of applying the bijection in the leaf on x. The obtained value is the correct hash value of the given element if the element was in the set which was used to construct this RecSplit instance.

All the required data for the RecSplit instance must be stored in a compact way which still allows for fast access. The splittings and bijections are stored in a bit vector called *Rice Bit Vector*. Each splitting/bijection is encoded as a *Golomb-Rice* code [57, 78, 79] (see Section 4.4). The number of keys in each bucket are stored as a prefix sum. This is necessary to retrieve the number of keys in buckets with smaller IDs in constant time. Finally, the bit position where the splitting tree of each bucket begins must be stored since all splitting trees are stored contiguously in the Rice Bit Vector. Note that this is the same as the exclusive prefix sum of the number of bits of each bucket. Both, the number of keys in prefix-sum form and the bit positions, are stored together in a customized *Elias-Fano* representation [45, 47, 46]. The RecSplit authors call this customization *Double Elias-Fano*, and we describe it in Section 4.5.

4.1 Initial Hash Function

The input elements are completely arbitrary. For example, the elements could be long strings and in particular, the length may vary between the elements. To obtain more handy

values, an initial hash function is applied to each element. The resulting *keys* have a fixed length. By using an initial hash function with good statistical properties, the keys can be treated like random values. The original implementation of RecSplit in the Sux library [13] uses SpookyHash V2 by Jenkins [60] for this purpose. It is fast and has good statistical properties. The resulting hash values consist of 128 bits. This is important since in a set of 5 billion 64-bit random values the probability of a collision (two equal values) is about 50% (birthday paradox [65]). This makes it impossible to construct large RecSplit instances if only 64-bit keys are used.

SpookyHash is not a cryptographic hash function. An adversary could find two different elements with the same hash value produced by SpookyHash. In this case, RecSplit cannot produce an MPHF since both keys are indistinguishable. In the original implementation of RecSplit, this would lead to an endless loop. This is effectively a denial-of-service attack. Therefore, SpookyHash is not suitable if the input elements are from an untrustworthy source (at least not without checking for collisions first). An alternative is SipHash [28]. It is not a cryptographic hash function either, but it uses private keys to prevent attackers from finding hash collisions. We do not care for adversaries in this thesis and use SpookyHash regardless of the dangers. But this is important to keep in mind if RecSplit is used in practice.

For the remainder of this thesis, we only consider the 128-bit keys which are the result of the hash function and not the original input elements. The keys are treated like random elements.

4.2 Bucket Assignment

The user specifies an expected bucket size b. The n keys are distributed into $\lceil \frac{n}{b} \rceil$ buckets according to the bucket-assignment function. This function operates on the 64 most significant bits of the 128-bit keys. Let u be the 64 most significant bits of the key x. The key x is assigned to the bucket with the ID

$$\left(u \cdot \left\lceil \frac{n}{b} \right\rceil \right) >> 64$$

where 128-bit multiplication is used. This leads to a practically unbiased distribution of the keys between the buckets since u is considered random. Conceptually, this is an *inversion* [44] and has been used in other projects as well [64]. The 64-bit key can be interpreted as a real number between 0 (inclusive) and 1 (exclusive) by treating it as the decimal part. Multiplying it with the number of buckets $\lceil \frac{n}{b} \rceil$ and rounding down results in a number between 0 (inclusive) and $\lceil \frac{n}{b} \rceil$ (exclusive). Instead of rounding down, the decimal part is removed here by shifting the result 64 bits to the right.

The expected bucket size b is restricted to the interval $[1, 2000]$. Experiments show that a larger b is not useful [46]. Due to the random distribution, bucket sizes can differ. To be exact, the bucket size of a given bucket follows a binomial distribution with parameters n and $p = \frac{1}{\lceil n/b \rceil} \approx \frac{b}{n}$. For $n \to \infty$, the bucket size is distributed according to the Poisson distribution with the parameter b [49]. Even for $b = 2000$ and a large amount of keys, the probability that 3000 or more keys are assigned to the same bucket is almost zero. This is taken as a fact in the original implementation [13], and we will use it as well for our implementations.

For the remainder of the construction, only the 64 least significant bits of the 128-bit keys are used. This ensures that the 64-bit keys within each bucket are completely independent of each other.

4.3 Splitting Trees

For each bucket, an MPHF is constructed independent of the other buckets. This MPHF is a splitting tree which partitions the keys into smaller and smaller sets until the individual sets are small enough such that a bijection on them can be found in reasonable time. The *leaf size* ℓ is a user-supplied compile-time parameter of RecSplit and specifies the maximum number of keys in a leaf. A larger leaf size leads to a longer construction time since searching for bijections is expensive for large leaves. On the flip side, splitting trees are more space efficient and can be queried faster for larger leaf sizes. The splitting tree has a well-defined shape to enable fast queries. This shape is defined only by the leaf size and the number of keys in the bucket.

A splitting tree comprises several *levels*. The tree is traversed from top to bottom. The lowest level is called *leaf level* and consists of the leaves of the splitting tree. Each leaf, except for possibly the last, contains ℓ keys. The level above the leaves is called *first aggregation level*. The number of child nodes of an inner node is called *fanout*. The maximum fanout s_1 in the first aggregation level is $s_1 = \max\{2, \lceil 0.35\ell + 0.5 \rceil\}$. Every node in the first aggregation level, except for possibly the last, has fanout s_1 and in particular contains $s_1\ell$ keys. 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. For details, see the original paper [46]. A larger fanout leads to a more shallow tree and therefore to faster queries and, as we will see later (see Section 4.4.1), to a more space-efficient MPHF.

Above the first aggregation level follows the *second aggregation level* which is similar. The fanout s_2 is again optimized such that the expected amount of work is equal to the combined amount of work of all children. The result is $s_2 = 2$ for $\ell < 7$ and $s_2 = \lceil 0.21\ell + 0.9 \rceil$ else. Again, each node in this level except the last has fanout s_2 and contains $s_2 s_1 \ell$ keys.

All remaining nodes are considered part of the *higher levels*. The fanout in the higher levels is always 2. A similar optimization as in the aggregation levels is not done since this would need another branch in the query. This is not really worth it because the amount of nodes in the higher levels is relatively small and the fanout would be smaller than in the lower levels anyway since more work is necessary for more keys. For a node in the higher levels, the number of keys in the left child is chosen as a multiple of $s_2 s_1 \ell$. As an example, the splitting tree with 202 keys for leaf size $\ell = 8$ is shown in Figure 4.1.

The corresponding level for a node can be calculated based on the number of keys using the formula below. A tree may not have every level. For example, the splitting tree of a bucket with ℓ or fewer keys consists of only a single node.

$$\text{level of a node with } m \text{ keys} = \begin{cases} \text{higher level} & \text{if } m > s_2 s_1 \ell \\ \text{second aggregation level} & \text{else if } m > s_1 \ell \\ \text{first aggregation level} & \text{else if } m > \ell \\ \text{leaf level} & \text{else} \end{cases}$$

The leaf size is restricted to satisfy $2 \leq \ell \leq 24$. Larger leaves are too expensive for practical purposes since bijections on many keys are hard to find. The restriction to this interval has advantages for the implementation.

Hash Functions

The brute-force searches for splittings and bijections utilize a family of random and independent hash functions $h_i^m \colon [2^{64}] \to [m]$, $i \in \{0, 1, 2, \dots\}$. The index i (which we also call *hash function identifier*) of the first hash function which satisfies the requirements of the splitting/bijection is stored in the Rice Bit Vector.

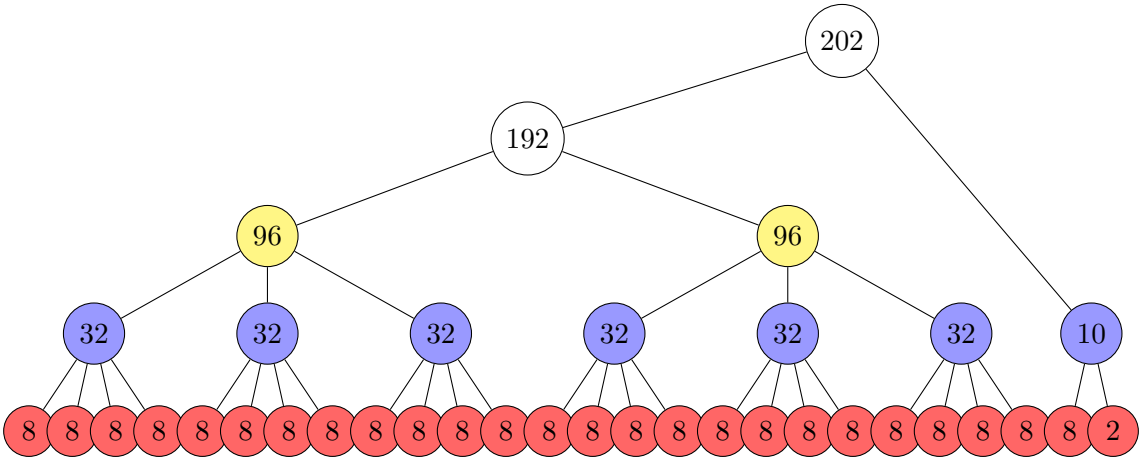


Figure 4.1: The splitting tree with 202 keys for leaf size $\ell = 8$. Each node is labeled with the number of keys it contains. The red nodes at the bottom are the leaves. The first aggregation level consists of the blue nodes above. Except for the last, all nodes in this level have fanout 4. The two yellow nodes above make up the second aggregation level. They have fanout 3. The two white nodes at the top constitute the higher levels. As always, the fanout in the higher levels is 2.

4.3.1 Analysis

In the following, we analyze RecSplit mathematically. Not all details and derivations are shown. See the original RecSplit paper [46] for more information.

Bijections

Let X be a set of m keys and $h\colon X \to [m]$ a random hash function on X. There are m^m possible hash functions of this form. Each of these functions has equal probability of being h. Out of these functions, $m!$ are bijections because there are $m!$ permutations of m elements. Consequently, the probability that h is a bijection is $\frac{m!}{m^m}$. The algorithm is trying out hash functions from the family of random hash functions until a bijection is found, beginning with the hash function with index 0. The index of the first bijection is distributed geometrically [49]. Therefore, the expected number of hash functions that must be tried out is $\frac{m^m}{m!}$. Using Stirling's approximation, this is approximately $\frac{e^m}{\sqrt{2\pi m}}$. In particular, the amount of work is exponential in the number of keys.

As a side note, this means finding a single bijection on the complete input of n keys (i.e., bucket size and leaf size are both n) is optimal in terms of space consumption of the MPHF. The hash function identifier that must be stored is on average about $\frac{e^n}{\sqrt{2\pi n}}$ which needs

$$\log_2\left(\frac{e^n}{\sqrt{2\pi n}}\right) = \log_2(e)n - \frac{1}{2}\log_2(2\pi n) \approx 1.44n$$

bits to be stored, i.e., 1.44 bits per key — the theoretical lower bound [51]. Of course, this is not practical. The construction time is exponential and the query time is linear in n since every bit of the MPHF must be used to query the hash value of any key.

Splittings

For a splitting of m keys with fanout s where child $i \in \{0, \ldots, s-1\}$ contains k_i keys, the expected number of hash functions that need to be tried out is

$$\sqrt{\frac{(2\pi)^{s-1}\prod_{i=0}^{s-1} k_i}{m}}.$$

This number is maximized if all children are equally large, i.e., balanced splits are more expensive to find than unbalanced splits.

4.4 Rice Bit Vector

The hash function identifiers are stored in a space-efficient way which still allows for fast access. Importantly, the shapes of the splitting trees are not stored explicitly. Instead, they 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. Only the hash function identifiers are stored. The encodings of all splitting trees are concatenated in a single bit vector named *Rice Bit Vector*.

4.4.1 Golomb-Rice Codes

Each hash function identifier is encoded as a Golomb-Rice code [57, 78, 79]. Given a Golomb parameter τ and the hash function identifier d, the τ least significant bits of d are stored directly and the remaining most significant bits of d are encoded in unary. The first part is called *fixed* part and the second is called *unary* part. The unary part consists of $(d >> \tau)$ 0-bits and a final 1-bit. For example for $\tau = 3$, $d = \texttt{1010}_2$ has the fixed part $\texttt{010}_2$ and the unary part $\texttt{01}_2$.

Golomb-Rice codes allow for storing arbitrarily large numbers which is important since the hash function identifiers can theoretically get arbitrarily large. Furthermore, using Golomb-Rice codes is useful since the hash function identifiers follow a geometric distribution. Golomb codes [57] are optimal for geometric distributions, and Golomb-Rice codes [78] are a faster special case which is almost as space efficient. The optimal Golomb parameter τ depends on the probability that a given hash function is a valid splitting or bijection, respectively. For the probability p, the optimal Golomb parameter is

$$\tau(p) = \max\left\{0, \left\lceil \log_2\left(-\frac{\log_2 \varphi}{\log_2(1-p)}\right)\right\rceil\right\} \tag{4.1}$$

with the golden ratio $\varphi = \frac{\sqrt{5}+1}{2}$ [61].

As was shown in the original paper [46], the space consumption of RecSplit tends to the optimum for large n, except for about 2 bits which are lost per splitting/bijection. In order to optimize the space consumption, the overall number of splittings and bijections should be reduced. This can be achieved by increasing the leaf size ℓ. This leads to a smaller number of bijections and to a larger fanout in the aggregation levels and therefore to fewer splittings. Viewed differently, there are more keys per splitting/bijection to amortize the cost of 2 bits. Larger leaf sizes also lead to faster queries because the splitting trees are more shallow due to the larger fanouts in the aggregation levels.

4.4.2 Fast Queries

The splitting tree must be traversed fast from the root to a leaf to calculate the hash value of a key. This is made possible by storing the fixed and unary parts of each splitting tree separated; first the fixed part and then the unary part of all Golomb-Rice codes of the splitting tree. Both are stored in preorder. The crucial part is that given a node v of the splitting tree, the shape of the subtree rooted at v depends only on the number of keys in v. This includes the number of nodes in the subtree and the number of bits stored in the fixed part of all nodes in the subtree. Both values, together with the Golomb parameter, can be stored in a lookup table indexed by the number of keys.

After evaluating a splitting in a query, it is clear which child node c must be evaluated next. To find the hash function identifier of c, it may be necessary to *skip* one or more subtrees. In the case where c is the first child of the splitting, no subtree must be skipped. The next values in the fixed and the unary part decode the correct hash function identifier. Otherwise, the subtrees of all children of the same splitting left from c must be skipped. In the higher levels, this is only one subtree. In the aggregation levels, it may be more than one subtree, but all these subtrees have the same number of keys and therefore the same shape.

To skip a subtree in the fixed part, the number of bits that the fixed part of the subtree occupies is looked up in the lookup table. If there is more than one left child, the number of bits to skip is multiplied with the number of left children.

Skipping a subtree in the unary part can be achieved by using a selection operation. Remember that the unary part of each hash function identifier contains exactly one 1-bit. The number of nodes in a subtree is therefore equal to the number of 1-bits in its unary part. Since the number of nodes is stored in a lookup table indexed by the number of keys, we only have to skip the number of 1-bits specified in the lookup table. Skipping m 1-bits is equal to finding the m-th 1-bit (zero-indexed). This is implemented by calling popcount on 64-bit words representing the unary part until at least m 1-bits are found. Then, the efficient selection operation described in Section 2.1 is used to find the exact bit position of the m-th 1-bit in the unary part.

As described earlier, the fixed and unary parts of each splitting tree are stored concatenated. To initially find the start of the unary part of the whole splitting tree without storing it explicitly, the same idea as skipping a subtree in the fixed part is used. Just look up the size of the fixed part of the whole splitting tree in the lookup table and add this to the start of the fixed part (which is stored in the Double Elias-Fano representation, see Section 4.5).

Example

As an example, consider $\ell = 8$ like in Figure 4.1. Assume the current node contains 90 keys. It can be deduced from this number that it is part of the second aggregation level. The lookup table at index 90 tells us that the Golomb parameter of this node is 6, i.e., the next 6 bits in the fixed part of the Rice Bit Vector are the least significant bits of the hash function identifier, and we need to count the number of 0-bits until the next 1-bit in the unary part to get the most significant bits. The hash function identifier is used to calculate the hash value of the input key, a value in [90]. For example, the hash value may be 72, i.e., the child at index $\left\lfloor \frac{72}{32} \right\rfloor = 2$ contains the input key. The divisor is 32 since this is the number of keys per node in the first aggregation level.

We have to skip two subtrees. The lookup table at index 32 says that each subtree contains five nodes. The bit position in the unary part is advanced until 10 1-bits are found, i.e., the new position is the bit after the 10th 1-bit. Using the lookup table again, each subtree contains 39 bits in the fixed part. Thus, the bit position in the fixed part is advanced 78 bits.

4.4.3 Lookup Table

For the construction algorithm and especially for the query algorithm, the Golomb parameter of a splitting/bijection with a given number of keys must be computed quickly. Moreover, the query algorithm requires the number of nodes and the number of bits in the fixed part of any splitting tree given its number of keys. For fast access, all three quantities are stored in a lookup table. It is important that this lookup table does not consume too much space since this would counteract the space efficiency of the MPHF.

All three quantities fit in a single 32-bit word per node size. The Golomb parameter only needs 5 bits because of the requirement $\ell \leq 24$ and Equation (4.1). The number of nodes uses 11 bits and the remaining 16 bits are used for the size of the fixed part. Since we assumed no bucket has 3000 or more keys, the lookup table only needs 3000 entries. With 32 bits each, the complete lookup table takes 12 kilobytes. For a million keys, this amounts to about 0.1 bits per key. However, we do not count this towards the required space of the MPHF as it is not really a part of the MPHF (the lookup table can be calculated again at any time independent of the keys), but it needs space in the main memory and cache.

4.5 Double Elias-Fano

RecSplit uses a customized Elias-Fano representation [45, 47] to store the prefix sum of the bucket sizes and the bit position where the splitting tree of each bucket starts in the Rice Bit Vector. This customization is called *Double Elias-Fano*. It enables constant-time access with little space requirements. We first describe the general Elias-Fano representation before explaining the customization done for RecSplit.

4.5.1 Elias-Fano Representation

An Elias-Fano representation can be used to store a monotonic sequence of integers

$$0 \leq x_0 \leq x_1 \leq \cdots \leq x_\kappa$$

with $\kappa \geq 0$. Similar to Golomb-Rice codes, the least significant bits of each value are stored directly. To be exact, the $\mu = \max\left\{0, \left\lfloor \log_2\left(\frac{x_\kappa+1}{\kappa+1}\right)\right\rfloor\right\}$ least significant bits of each value are stored contiguously in the *lower-bits array* and can be accessed directly.

The remaining most significant bits are encoded as unary differentials in the *upper-bits array*. This means that the upper-bits array contains a 1-bit for every value in the sequence and the number of 0-bits between the 1-bits of two values is exactly the difference of the most significant bits of the two values. For example for $0 \leq i < j \leq \kappa$, the number of 0-bits between the i-th and j-th 1-bit (zero-indexed) in the upper-bits array is $(x_j >> \mu) - (x_i >> \mu)$. Put differently, the corresponding 1-bit of x_i is at the position $(x_i >> \mu) + i$. A selection data structure is used to find the position ν of the i-th 1-bit of the upper-bits array in constant time. Since $\nu = (x_i >> \mu) + i$, the upper bits of x_i can be calculated as $\nu - i$.

4.5.2 Customization

RecSplit stores the prefix sum of the bucket sizes and the bit position of the start of each splitting tree in a single combined Elias-Fano representation, hence the name *Double Elias-Fano*. Let λ_i be the prefix-sum sequence and σ_i the bit-position sequence for $i \in [\kappa+1]$ with the number of buckets $\kappa = \lceil \frac{n}{b} \rceil$. It is $\lambda_0 = \sigma_0 = 0$, $\lambda_\kappa = n$ the total number of keys, and σ_κ the total number of bits of the Rice Bit Vector. The sequence σ_i is not stored directly. Instead, the sequence $\psi_i = \sigma_i - \beta\lambda_i$ is stored where $\beta = \frac{\sigma_\kappa}{\lambda_\kappa}$ is the average number of bits per key the splitting trees require. This utilizes the dependency between both sequences to compress σ_i. Note that ψ_i may not be monotone which is resolved in the following.

As the next step, both sequences are rescaled by reducing the differences between successive elements by the minimum difference. This means,

$$\delta_\lambda = \min_{i \in \{1,\ldots,\kappa\}} (\lambda_i - \lambda_{i-1}) \quad \text{and} \quad \delta_\psi = \min_{i \in \{1,\ldots,\kappa\}} (\psi_i - \psi_{i-1})$$

are calculated and λ_i is replaced by the sequence $\phi_i = \lambda_i - i\delta_\lambda$ and ψ_i is replaced by $\chi_i = \psi_i - i\delta_\psi$. This reduces the range of λ_i, especially if b is large. For example if $b = 2000$,

the probability that a bucket contains less than 1500 keys is almost zero. Therefore, $\delta_\lambda \geq 1500$ with high probability, which reduces each value on average by a factor of four and saves about two bits per bucket. If ψ_i is not monotone, then δ_ψ is negative and the rescaled sequence χ_i is again monotone.

The lower-bits arrays of both sequences are interleaved, i.e., for a given $i \in [\kappa + 1]$, χ_i is stored directly after ϕ_i in a single lower-bits array. This can potentially save a cache miss. The upper-bits arrays are stored separately. The authors of RecSplit note that both sequences are very regular. This allows for a simple selection data structure. This data structure provides the position of each 2^{14}-th 1-bit and each 256th 1-bit in between. The first position is stored as a 64-bit word, the second position is relative to the first and only requires 16 bits. As before, the positions for both upper-bits arrays are interleaved to potentially save a cache miss.

Let $\zeta_\phi(\gamma)$ be the position of the γ-th 1-bit in the upper-bits array of ϕ_i, i.e., the 1-bit corresponding to ϕ_γ. Analogously, $\zeta_\chi(\gamma)$ is defined for χ_i. The array containing the positions is called *jump array*. A single entry of this array at index j contains several values. First, it contains the position $\xi_\phi = \zeta_\phi(2^{14}j)$, i.e., the position of the 1-bit corresponding to $\phi_{2^{14}j}$ in the upper-bits array of ϕ_i. It is followed by the position $\xi_\chi = \zeta_\chi(2^{14}j)$. Both values are 64-bit words. Finally, the position of every 256th 1-bit is stored relative to the corresponding 64-bit position as a 16-bit word, alternating between both sequences. These are the values

$$\zeta_\phi(2^{14}j) - \xi_\phi, \zeta_\chi(2^{14}j) - \xi_\chi, \zeta_\phi(2^{14}j + 256) - \xi_\phi, \zeta_\chi(2^{14}j + 256) - \xi_\chi, \zeta_\phi(2^{14}j + 2 \cdot 256) - \xi_\phi, \ldots$$

where the first two entries are always zero. This concludes the description of the selection data structure.

We proceed to explain how to compute ϕ_i, ϕ_{i+1} and χ_i in expected constant time using the stored data. These three values can then be used to compute λ_i, λ_{i+1} and σ_i by reverting the steps explained in the previous paragraphs. The first value (λ_i) represents the number of keys which are in buckets with an index smaller than i. This number must be added to the hash value computed by the splitting tree to get the correct hash value of the whole MPHF. The second value (λ_{i+1}) is necessary to calculate the size of the bucket $\lambda_{i+1} - \lambda_i$, and χ_i denotes where the encoding of this buckets splitting tree begins in the Rice Bit Vector.

The least significant bits of all three values can be retrieved directly with a lookup in the lower-bits array. To calculate the most significant bits of ϕ_i and χ_i, the positions of the respective 1-bits in the upper-bits arrays need to be found. The selection data structure is used to find the nearest 1-bit which is represented by the structure, i.e., every 256th 1-bit. The correct position is finally searched linearly by using popcount and selection on 64-bit words (see Section 2.1) in both upper-bits arrays. The 1-bit corresponding to ϕ_{i+1} follows the 1-bit of ϕ_i and can therefore be found fast. The most and least significant bits only need to be combined to get the correct values.

Analysis

In the following, we analyze the customized Elias-Fano representation in greater detail than in the original RecSplit paper [46]. Let $\mu_\phi = \max\left\{0, \left\lfloor \log_2\left(\frac{\phi_\kappa + 1}{\kappa + 1}\right)\right\rfloor\right\}$ and $\mu_\chi = \max\left\{0, \left\lfloor \log_2\left(\frac{\chi_\kappa + 1}{\kappa + 1}\right)\right\rfloor\right\}$. The space consumption of the lower-bits array is $(\kappa + 1)(\mu_\phi + \mu_\chi)$ bits. The upper-bits arrays need $\kappa + 1 + (\phi_\kappa >> \mu_\phi)$ or $\kappa + 1 + (\chi_\kappa >> \mu_\chi)$ bits, respectively. The number of bits of both upper-bits arrays is at most $3\kappa + 2$. We show this by showing

$\phi_\kappa >> \mu_\phi \leq 2\kappa + 1$, the other case is analogous. If $\mu_\phi = 0$, then $\phi_\kappa + 1 < 2\kappa + 2$ and therefore $\phi_\kappa >> \mu_\phi = \phi_\kappa \leq 2\kappa$. Otherwise, using substitution with $\Phi = \frac{\phi_\kappa + 1}{\kappa + 1}$,

$$\begin{aligned}
\phi_\kappa >> \mu_\phi &= \left\lfloor \frac{\phi_\kappa}{2^{\lfloor \log_2(\frac{\phi_\kappa+1}{\kappa+1}) \rfloor}} \right\rfloor \\
&= \left\lfloor \frac{(\kappa+1)\Phi - 1}{2^{\lfloor \log_2(\Phi) \rfloor}} \right\rfloor \\
&\leq \left\lfloor \frac{\Phi}{2^{\lfloor \log_2(\Phi) \rfloor}} (\kappa + 1) \right\rfloor \\
&\leq 2\kappa + 1
\end{aligned}$$

since $\frac{\Phi}{2^{\lfloor \log_2(\Phi) \rfloor}} < 2$ for all $\Phi > 0$.

The jump array needs $\frac{\kappa}{2^{14}} \cdot 2(64 + 16 \cdot \frac{2^{14}}{256}) + O(1) = \frac{17}{128}\kappa + O(1)$ bits. The overall space consumption of the Double Elias-Fano representation is dominated by the lower-bits array. It needs $O(\kappa \log b) = O(\frac{n \log b}{b})$ bits. This means we can decrease the amount of space the Double Elias-Fano representation occupies by increasing b. However, this also increases construction and query time as we will see in the Evaluation (Chapter 6).

The only part of querying Double Elias-Fano which does not obviously need constant time is the linear search in the upper-bits arrays after the selection data structure has been used. The required running time for this step is linear in the distance between the 1-bit which was found by the selection data structure and the 1-bit representing the queried bucket. Since both upper-bits arrays contain $O(\kappa)$ bits and exactly $\kappa + 1$ 1-bits each, this distance is *on average* $O(1)$. Here it is important that both stored sequences are very regular to justify the *expected* constant running time from this fact.

5. Implementation

In this chapter, we describe how the different implementations of RecSplit work in more detail. Each separate step is first explained generally before discussing details of the implementations for specific hardware architectures. Finally, an overview is given how the separate steps are used to construct a RecSplit instance. The GPU implementation uses CUDA 11 [71]. The SIMD implementation uses the vector class library by Fog [26, 27], only supports x86 CPUs and is optimized towards AVX2 and AVX-512.

5.1 Sorting

The original RecSplit implementation in the Sux library [13] uses the sorting function of the C++ standard library to sort the keys according to their bucket numbers. For example, the libstdc++ standard library used by the GCC compiler implements introsort [70, 56] which is based on quicksort and has a running time of $O(n \log n)$. This means that, strictly speaking, this implementation of RecSplit does not have expected linear running time. However, for most useful parameter combinations, the sorting time is negligible compared to the remainder of the algorithm.

Since we only want to distribute the keys according to their respective bucket numbers and do not need to sort the keys inside each bucket, a rather natural choice for a faster algorithm is bucket sort. More specifically, we use a variant known as counting sort [80]. Our custom implementation of this out-of-place algorithm has three phases:

1. Count the number of keys that are hashed to each bucket in an array A.
2. Compute the inclusive prefix sum of the bucket sizes A.
3. Place the keys in the right buckets by iterating over the input, computing the bucket number for each key, placing it in the output position given by A and decreasing the position in A. Only the last 64 bits of the keys are placed in the bucket since the other half is only used to determine the bucket number. After this phase, A contains the exclusive prefix sum of the bucket sizes.

Counting sort has linear running time for keys with constant size. However, for inputs exceeding the cache sizes it may have an inferior performance compared to introsort since counting sort does several random accesses per key. Introsort, contrarily, traverses the input linearly and can therefore utilize the cache more efficiently [62]. This problem can be alleviated by using prefetching in phases 1 and 3 to hide the latency of cache misses.

For this, the `_mm_prefetch` [22] intrinsic from the `immintrin.h` header is used, which emits a prefetch instruction on the x86 architecture. In the first phase, the input data is processed in batches of size 32. First, the addresses of the respective counters are calculated, prefetched, and subsequently incremented. The third phase is handled similarly, but has two prefetching loops, one for the indices in A and one for the output.

On top of the better performance, counting sort has another advantage. After counting sort has finished, the array A contains the exclusive prefix sum of the bucket sizes, i.e., the starting index of each bucket. This is required later for the Elias-Fano representation and had to be calculated extra in the original implementation. Furthermore, knowing the bucket positions beforehand is necessary for the parallelization of SIMDRecSplit (see Section 5.8).

A disadvantage of this implementation of counting sort is that it is not in-place and consumes about 50% additional memory to store the sorted output.

5.2 Starting Seeds

The probabilistic analysis done in the original RecSplit paper [46] is only applicable if all evaluations of hash functions are independent of each other. For different keys, and in particular for different buckets, this is ensured since the keys are the result of the initial hash function and therefore are decorrelated (see Section 4.1). However, the same key is used on different levels within its bucket. We need to make sure that the used hash functions are different each time the key is used. The original implementation [13] tracks the level of the split or bijection that is currently examined and looks up a randomly generated but fixed starting seed depending on the level. This starting seed is added to the hash function ID (which is later stored in the Rice Bit Vector) and the key to be hashed. The level of the highest split is 0.

In the GPU version, all leaves (except possibly the last one if it has another size) are processed with one kernel call. The same is true for the two aggregation levels above the leaves. Note that the leaves do not need to have the same level since the level is counted beginning at the top and the tree is not necessarily perfectly balanced. To avoid passing the starting seed to each leaf of the kernel, a constant starting seed for all leaves is chosen randomly, but fixed at compile time. Identically, different starting seeds are generated for the two aggregation levels above the leaf level. The higher levels still use the same starting seed lookup table as the original version. Note that this modification does not introduce any correlation since the splits and bijections that use the same starting seeds use different keys.

The query function must be changed slightly to incorporate these changes. In the last three levels, the constant starting seeds are used instead of the lookup table. This should not be slower than the previous version. In fact, it is expected to be slightly faster since the constant is known at compile time and does not need to be looked up in a table. Moreover, two increments of the level can be avoided. For this reason and to ensure both versions produce the exact same resulting MPHF, the SIMD version also uses this technique.

5.3 32-Bit Hash Function

The original RecSplit implementation [13] uses a 64-bit remix function [8, 41] to implement the hash functions. This remix function $f\colon [2^{64}] \to [2^{64}]$ takes a 64-bit word and returns a 64-bit word by mixing the input bits using shifts, XORs, and multiplications with constants. It can be viewed as a pseudorandom permutation of all 64-bit words. RecSplit uses it to implement hash functions of the form $h\colon [2^{64}] \to [m]$ with $m < 2^{16}$ by using the remix

function on the input, setting the 16 most significant bits to zero, multiplying the result with m and shifting it 48 bits to the right:

$$h(x) = ((f(x) \,\&\, ((1 << 48) - 1)) \cdot m) >> 48$$

Effectively, this interprets $f(x)$ as a fixed-point decimal number between including zero and excluding one, where the decimal point is after the 16 most significant bits [46]. By multiplying with m, the result is a fixed-point decimal number between zero (inclusive) and m (exclusive). The shift by 48 bits to the right extracts the decimal part before the decimal point, which is an integer between including 0 and excluding m.

The remix function f uses two 64-bit integer multiplications, i.e., three 64-bit multiplications are necessary per evaluation of the hash function h. This is a problem for the GPU implementation since most GPUs are optimized for 32-bit operations (floating-point and integer). Shifts, XOR, and addition of 64-bit integers can be implemented by using only a few 32-bit instructions. However, 64-bit multiplications take a lot more instructions and are therefore very expensive on GPUs [12].

This problem can be solved by using a 32-bit remix function $g\colon [2^{32}] \to [2^{32}]$. Fortunately, MurmurHash3 — from which the 64-bit remix function originates — also contains a 32-bit remix function [8]. This function does the same operations as f on 32-bit words with other constants. The function g can be used to implement h in the same way as above by replacing the constant 48 by 16 and using 32-bit operations. This way, h can be implemented without any 64-bit integer multiplications. This does not only profit the GPU version but also the SIMD version since it can effectively double the throughput of the remix function. The advantage for the GPU implementation is shown in Section 6.5.1.

However, the input of h is always a 64-bit word and must first be transformed to a 32-bit word before using g. It is not good to ignore the 32 most significant bits and only use the 32 least significant bits. For example, this would be a problem for $\ell = 24$. Recall that the probability that a random hash function is a bijection on 24 keys is $\frac{24!}{24^{24}} \approx \frac{\sqrt{2\pi \cdot 24}}{e^{24}}$. Therefore, the probability that more than 2^{32} hash functions must be tried is $\left(1 - \frac{\sqrt{2\pi \cdot 24}}{e^{24}}\right)^{2^{32}} \approx 0.137$. If the 32 most significant bits of the input are ignored, there are effectively only 2^{32} different hash functions. If more are needed, there is no possible solution. In our implementation, this problem would result in an endless loop.

A more severe problem is that effectively the 32 most significant bits of the keys are ignored. Therefore, if two keys with the identical 32 least significant bits end up in the same bucket, they are always hashed to the exact same hash code. This makes it impossible to find a solution again since both keys end up in the same leaf and there is no bijection possible. The probability that there are at least two identical keys in a bucket of 2000 random 32-bit keys is $1 - \frac{(2^{32})!}{(2^{32}-2000)! \cdot (2^{32})^{2000}} \approx 0.000465$. Given ten million keys, i.e., 5000 buckets, the probability to create this situation is $1-(1-0.000465)^{5000} \approx 0.902$. This is an unacceptable risk.

These problems can be solved by using the XOR of the 32 most and the 32 least significant bits of the input. Note that the input of h in the example is the sum of the hash function ID d (beginning with 0) and the key to hash y (we ignore the starting seed here). Both, d and y, are 64-bit words. If the 32 most significant bits of $d + y$ are ignored, $d + t2^{32}$ results in the same hash function for every integer t. This is not the case if the XOR of the 32 most and the 32 least significant bits is used since the addition carries information to the more significant bits. This carry depends on the key y. Note that it is inevitable that information is lost on the transition from 64 to 32 bits. However, the crucial part is that the transformation looks different for different keys y. For this to work it is import that d

and y are mangled using addition (not XOR or other bitwise operations!) and that the XOR happens *after* the addition.

The constants of the 64-bit remix function were optimized by Strafford such that the result of two consecutive words are as decorrelated as possible [41]. This is useful for RecSplit since two hash functions that are tried out consecutively use consecutive words (d is increased by one for each new hash function). Something similar could be done for the 32-bit hash function. However, it would be more complicated since the words are not consecutive because of the XOR and the starting seed. Furthermore, the experiments show no significant difference of the size of the MPHFs between the original version (which uses the 64-bit remix function) and the SIMD/GPU versions (which use the 32-bit function); see Section 6.5.1. This does not promise much to gain by optimizing the 32-bit remix constants.

5.3.1 SIMD Implementation

Let us now discuss the specifics of the remix function for the SIMD implementation. The SIMD version works on vectors of a fixed number of bits. Consider the x86 instruction set extension AVX-512. Each vector consists of 512 bits. Therefore, a vector can contain eight 64-bit words or 16 32-bit words. A vector operation performs the same operation on all words in the vector. Preferably, 32-bit words should be used to increase throughput. To use the 32-bit hash function, the 64-bit input must first be transformed to 32 bits using XOR as explained above. This means the input consists of two vectors containing 64-bit words. There are several possible implementations to transform these two vectors into one vector containing the proper 32-bit words which can be used as the input of the hash function.

The first implementation uses the `compress` function of the Vector Class Library [27]. This function takes two vectors containing 64-bit words and packs the 32 least significant bits of each word into one vector containing 32-bit words. Given the input vectors x and y, the transformation can be implemented as

$$\texttt{compress}(x >> 32, y >> 32) \oplus \texttt{compress}(x, y).$$

The shift operation operates on the individual 64-bit words in the vector.

The second implementation uses the `blend16` function [27]. It takes two vectors and 16 template parameter indices. The result is the concatenation of the 32-bit words selected by the indices. An index of 16 or more means the respective word in the second input vector is meant. This function can be used to pack the 32 least or most significant bits of each 64-bit word into one vector. Let `blend16even` be the function with all the even indices from 0 to 30 as template parameters and `blend16odd` the function with all the odd indices. The transformation then looks like

$$\texttt{blend16even}(x, y) \oplus \texttt{blend16odd}(x, y).$$

According to the experiments (see Section 6.4.1), the second implementation is faster and is therefore used. If AVX2 is available, but no AVX-512, 256 bit vectors containing four 64-bit words or eight 32-bit words are used. Instead of `blend16`, `blend8` is used. If no AVX2 is available, 256 bit vectors are simulated by the Vector Class Library utilizing SSE instruction sets if available.

5.4 Leaf Level

For each leaf, a bijection on the leaf keys $X = \{x_1, \ldots, x_m\}$ must be found. Let $h\colon X \to [m]$ be a hash function. The following algorithm is used to determine whether h is a bijection:

1. Initialize $a \leftarrow 0$.
2. For each $x \in X$, set the bit of a at index $h(x)$ to one, i.e., $a \leftarrow a \mid (1 << h(x))$.
3. The function h is a bijection if and only if the m least significant bits of a are all set to one, i.e., if $a = (1 << m) - 1$.

5.4.1 Bijection Midstop

For large m, this procedure can be accelerated. After the first $p = p(m)$ ($p < m$) keys are processed, there may already be a collision, i.e., two or more keys with the same hash value. Note that in this case, there are less than p bits set in a. Therefore, collisions can be detected after processing the first p keys using the popcount instruction. If a collision is found, the algorithm proceeds with the next hash function. Otherwise, the remaining keys are hashed and step 3 is done as usual. We call this algorithm *bijection midstop*.

In the original RecSplit implementation [13], bijection midstop is used for $m \geq 9$. The parameter $p = p(m) = \lceil 2 \cdot \sqrt{m} \rceil$ is chosen such that the probability that a collision is found is about 90%.

5.4.2 GPU Implementation

One kernel call per bucket is used to find the bijections of all leaves with the same size. Each thread block handles exactly one leaf. First, the leaf keys X are loaded into shared memory using `memcpy_async` as explained in Section 2.3.2. The threads use the time it takes the data to arrive for initialization. The memory access is not perfectly coalesced since proper alignment is not guaranteed because the keys of all leaves are stored contiguously without padding. Nonetheless, almost every byte loaded from global memory by the kernel is used by one of the thread blocks which improves caching.

A hash function identifier d in shared memory is initialized to the maximum possible value. Each thread tries a distinct hash function defined by the thread index inside the block. If a thread finds a bijection h, it atomically sets d to the minimum of d and the identification of h. Then, each thread tries the next hash function which is defined as the last plus the number of threads in the block. After k tries, the threads synchronize, check if a bijection was found and if it was, load d into global memory. The value k is dependent on $m = |X|$. The more keys X contains, the higher is k since more tries are necessary on average to find a bijection. This reduces the amount of synchronizations for large leaves. For $m < 14$ or when using bijection rotation (see Section 5.5), then $k = 1$.

Since a warp works in lock-step, bijection midstop 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 keys. Therefore, using the same midstop parameter as in the original version [13] is not advisable. Instead, $p = p(m) = \lceil 3 \cdot \sqrt{m} \rceil$ is used, which again provides a probability of about 90% that all threads in the warp find a collision. This parameter also proved almost optimal in experiments, see Section 6.5.2. Bijection midstop is only used for $m \geq 14$.

5.4.3 SIMD Implementation

Similar to the GPU implementation, several hash functions are tried out at the same time by loading consecutive hash function identifiers in a vector of 64-bit words. For each key in X, the key is first added to the vector. A second vector is produced by adding the number of 64-bit words in a vector. Both vectors are combined into one vector containing 32-bit words before applying the hash function as explained in Section 5.3.1. The result for each SIMD lane is a number in $[m]$. As in the other implementations, a bit at this number interpreted as an index must be set to 1.

AVX2 and AVX-512 contain an instruction named VPSLLVD to shift the contents of a vector by the number of bits specified in another vector [14, 21]. Unfortunately, the Vector Class Library does not support this instruction and only permits shifting the contents of a vector by an amount that is the same for all lanes [27]. However, the instruction can still be used by using the intrinsic `_mm256_sllv_epi32` or `_mm512_sllv_epi32`, respectively [22]. We use it to shift a 1-bit to the specified index. As a fallback if neither AVX2 nor AVX-512 is available, a lookup in a table containing the powers of two is used. Note that shifting the number 1 by x bits to the left is the same as computing 2^x.

After all elements are processed, checking for a bijection is done by comparing the contents of the SIMD lane with the word where the m least significant bits are set to 1. The result is a boolean vector which contains a boolean for each lane indicating whether a bijection was found. To check whether a bijection was found in any SIMD lane, the `horizontal_or` function of the Vector Class Library is used. It returns true if and only if at least one of the lanes contains "true". If it returns true, the `horizontal_find_first` function is used to calculate the index of the first lane containing true which is used to compute the first hash function identifier which results in a bijection.

Bijection Midstop

Bijection midstop works a little different in the SIMD implementation than in the other implementations. Remember that in the GPU implementation if any thread in a warp does not find a collision, the whole warp has to continue processing the remainder of the leaf. This can be avoided in the SIMD implementation by using a backlog. After the midstop, all words which do not exhibit a collision are stored in a backlog. When there are enough keys in the backlog to fill a complete 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.

There are two different implementations to calculate the popcounts depending on the instruction set. If the instruction set extension AVX512VPOPCNTDQ is available, we can use popcount on 512-bit vectors containing 32-bit words [21]. The popcounts are then compared to the midstop parameter. The function `to_bits` of the Vector Class Library is used to obtain a bitmask of 16 bits with a 1-bit at every index where no collision was found. As long as the bitmask is not zero, we can use ρ (see Section 2.1) to find the index of the first word without a collision, clear the respective bit with `clear_rho`, and store the word in the backlog.

If AVX512VPOPCNTDQ is not available, popcount cannot be used on vectors. Instead, all words are dumped in an array and then processed sequentially.

5.5 Bijection Rotation

A new and faster approach to speed up the search for bijections than bijection midstop is called *bijection rotation*. Essentially, bijection rotation is a method to find an MPHF on

m keys with $m \leq w$ (the word size of the machine). The idea is similar to FCH [50] (see Section 3.4). We distribute the keys randomly into two sets A and B by calculating the parity of each key. Note that in all other places the key is used in conjunction with a hash function, therefore we can use the key here directly without causing correlations.

Given a hash function h, we calculate the hash value of all keys in A and set the respective bits in the word a to 1. Like bijection midstop, h may be ruled out as a valid bijection by calculating the popcount of a. The set B is processed identically with the bits set in the word b. For all possible m rotations of b, we test if we have found a bijection. This is the case if and only if $a \mid \texttt{rot}_m^r(b)$ has the m least significant bits all set for a rotation $r \in [m]$. To efficiently store r, only every m-th hash function is tried, which means the number of the hash function is congruent to zero modulo m. This number plus r is stored in the Rice Bit Vector. We can restore r later by calculating modulo m and restore the hash function by subtracting r.

Bijection rotation is only used for full leaves, i.e., if $m = \ell$. This leads to an extra branch in the query but avoids using bijection rotation for small leaves. Furthermore, ℓ is a compile-time constant which can accelerate the modulo operations.

5.5.1 Analysis

For a simple analysis, assume that m is even and $|A| = |B|$. Moreover, assume that after A is processed, a has m distinct rotations, i.e., for all $r \in [m] \setminus \{0\}$, $\texttt{rot}_m^r(a) \neq a$. We define the following events:

- P means "h is a bijection on X"
- Q means "h is a bijection on $X = A \cup B$ using bijection rotation"
- R means "h is injective on A"
- S means "Given a subset $T \subseteq [m]$ with $|T| = \frac{m}{2}$, $h(B) = T$"

Theorem 5.1. $\mathbb{P}(Q) = m\mathbb{P}(P)$

Proof. The function h is a bijection on X using bijection rotation if it is injective on A and fills the remaining holes with B. Therefore,

$$\mathbb{P}(Q) = \mathbb{P}(R) \cdot m\mathbb{P}(S \mid R) = \mathbb{P}(R) \cdot m\mathbb{P}(S)$$

since S and R are independent and there are m possible rotations of b to fill the holes left in a. To be specific, there are m different events with the form of S where each event has a different set T. The set T contains one of the m possible rotations of the holes left in a. Note that rotating b has the same effect as rotating a, hence this view is equivalent. These m events are disjoint because at most one of them can occur. This is only true because of the assumption that a has m distinct rotations. The probability that any of these m events occur is equal to the sum of the probabilities. Each individual probability is $\mathbb{P}(S)$, i.e., the sum is $m\mathbb{P}(S)$.

Since there are $m^{m/2}$ possible functions $f\colon A \to [m]$ of which $\frac{m!}{(m/2)!}$ are injective functions, $\mathbb{P}(R) = \frac{m!}{(m/2)!m^{m/2}}$. Similarly, $(m/2)!$ of the possible functions satisfy the condition in S: $\mathbb{P}(S) = \frac{(m/2)!}{m^{m/2}}$. We can conclude that

$$\mathbb{P}(Q) = \frac{m!}{(m/2)!m^{m/2}} \cdot \frac{(m/2)!}{m^{m/2}} = \frac{m! \cdot m}{m^m} = m\mathbb{P}(P).$$

□

This means that if the assumptions are satisfied, the expected number of hash functions that need to be tried is reduced by a factor of m, which can speed up the construction significantly. In practice, the gains are smaller since $|A|$ and $|B|$ are not guaranteed to be equal and a may have less than m distinct rotations. For example using binary notation, $\texttt{rot}_4^2(1010_2) = 1010_2$.

5.5.2 Lookup Table

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 $\texttt{rot}_m^{t[a]}(a)$ is minimal. If a value x can be rotated to get the value y, then $\texttt{rot}_m^{t[x]}(x) = \texttt{rot}_m^{t[y]}(y)$. Let $c = (1 << m) - 1$ 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. The necessary rotation of b can now be calculated with just two lookups. We have found a bijection if and only if

$$a \mid \texttt{rot}_m^{((t[\hat{b}]-t[a]) \bmod m)}(b) = c.$$

However, this lookup technique is not used in our implementations. Often, especially for large leaves, it is not necessary to compute the rotations since h is not injective on A or B, which can be detected with popcount to stop early. Moreover, the lookup table consists of 2^ℓ bytes and is accessed pseudorandomly. This can cause cache misses, especially for $\ell \geq 16$ since many processors have an L1 data cache size smaller than 2^{16} B = 64 KiB. For example, the Intel Core i7-11700 which we use in our experiments has 48 KiB of L1 data cache per core as reported by the `lscpu` command [20]. We could not measure improvements in the sequential and scalar implementation using the lookup table, see Section 6.3.2

For the GPU and SIMD implementations, the rotations must be computed more often since the early stopping does not work as well (see below). Nevertheless, the lookup technique is not promising. In the SIMD case, gather instructions are necessary to load the values from the lookup table to the vector registers. These instructions are relatively expensive compared to the simple instructions used to rotate. In the GPU case, global memory operations are too expensive to justify saving some instructions. Constant memory is optimized for the use case where all threads in a warp access just a few or only one single element. Shared memory is a scarce resource and still significantly slower than registers. Therefore, the lookup technique is not used for the GPU implementation either.

5.5.3 SIMD Implementation

We proceed to explain specifics of bijection rotation for the SIMD implementation. In the other implementations, popcount is used after processing each set to find collisions early as explained before. Unfortunately, this is not as straightforward for the SIMD implementation. After processing the first set, some SIMD lanes may already find a collision. But if at least one lane does not contain a collision, we need to process the other set as well. Similarly, there may be lanes which do not contain a collision for the second set. We can only stop early if for each lane a collision was found for one or both sets. Therefore, collisions are only checked after processing both sets, but before trying out the rotations. Since the popcount instruction on vectors are necessary, this optimization is only used if the AVX512VPOPCNTDQ instruction set is available.

Remember that the number that is stored in the Rice Bit Vector 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 GPU implementation, caution must be taken when trying out the rotations. This goal is not achieved by trying out one rotation after the other and checking if a bijection was found after each rotation. 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.

Note that several rotations of the same word can lead to a bijection. For example, consider $\ell = 4$ and the two words $a = \texttt{0101}_2$ and $b = \texttt{1010}_2$. We need to rotate b such that the 1-bits in b match the 0-bits in a. The word b can be rotated either zero or two bits to the left to find a match. To make sure that always the smallest rotation is found, we start at the highest rotation. To be exact, a right rotation instead of a left rotation is used to rotate b by one bit to the right in every iteration. The `select` function of the Vector Class Library is used to store the current rotation in the result if a bijection was found. This function takes a boolean vector and two additional data vectors as input and selects the value of one of the two data vectors per lane based on the value in the boolean vector.

5.6 Aggregation Levels

Above the leaf level in the splitting tree, there are two so-called aggregation levels. Each node of the first aggregation level splits a subset of the bucket into the leaves. Each node in the second aggregation level splits a larger subset into the nodes of the first aggregation level. The size of the subsets (called unit) is fixed at compile time, except the last one per level which can be smaller. The same is true for the fanout, i.e., the number of parts the subset is split into. Consider a node in the aggregation levels with the keys X with $|X| = m$, unit u, and fanout s. To check whether a given hash function $h\colon X \to [m]$ is a valid splitting, the following algorithm is used:

1. Initialize array A with s words that are each equal to zero. The values in A are called counts.
2. For each $x \in X$, increment $A[\lfloor h(x)/u \rfloor]$ by one.
3. The function h is a valid splitting if and only if $A[i] = u$ for all $i \in [s-1]$.

The hash function does not produce a value in $[s]$ directly since this would mean the probability that a key is hashed to a child node is the same for all child nodes. However, the last child node may be smaller than u which should be reflected by the probabilities to maximize the probability of a given hash function to be a splitting. This is achieved by hashing to $[m]$ and then dividing by u in step 2. We do not need to check the last count in A because the value is determined after checking all other counts.

5.6.1 Integer Division

Usually, integer division is a much slower operation than integer multiplication. However, it can be accelerated by using shifts, additions and multiplications, especially if the divisor is known at compile time [59]. The simplest case is if the divisor is a power of two, i.e., 2^j. It can be replaced by a shift of j bits to the right. For other divisors, the formula $\left\lfloor \frac{x}{y} \right\rfloor \approx \left(x \cdot \left\lfloor \frac{2^q}{y} \right\rfloor\right) >> q$ for a suitable q can be used. If y is a compile-time constant, finding a suitable q, computing $\left\lfloor \frac{2^q}{y} \right\rfloor$ and finding corrections to circumvent rounding errors can be done at compile time. At running time, only the multiplication, right shift and the necessary corrections are executed [27].

Fortunately, the only division needed in our implementations is the division by the unit u in the algorithm above, which is a compile-time constant. A good optimizing compiler

will replace this division by such an optimized series of instructions [59]. This is also true for the GPU implementation as inspecting the assembly revealed. In the SIMD implementation, the calculations are done by the Vector Class Library [26]. To ensure that most steps are done at compile time, the divisor must be wrapped by the `const_uint` macro. Otherwise, the Vector Class Library will do all steps at running time since the available vector extension may not contain a division instruction [27].

The Vector Class Library does not support 64-bit integer division, but we only need 32-bit division. Furthermore, unsigned division is faster than signed division and 16-bit division is faster than 8- or 32-bit division. In our case, the dividend is a 32-bit unsigned integer. However, 16-bit division can be utilized since the dividend is bounded by the maximum bucket size which is 3000. To do this, the result of the hash function is first converted to a vector containing 16-bit words by cutting off the 16 most significant bits. This vector is used for the division and must be extended to 32 bits per lane afterward to be processed further. Using 16-bit division did not clearly improve the construction time (see Section 6.4.5) and is therefore not used in the final implementation.

5.6.2 SIMD Implementation

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. This problem can be alleviated by storing the counts in vector registers. To use this technique efficiently, we first show that a single byte per count is sufficient.

Theorem 5.2. *Given $u < 256$, 8 bits per count are sufficient to correctly verify whether h is a valid splitting.*

Proof. If h is a valid splitting, the first $s-1$ counts are correctly u since no overflow occurs.

Let $A[i] = u$ for all $i \in [s-1]$ where each word in A has 8 bits. We need to show that the number of keys that are hashed to each of the first $s-1$ child nodes is really u. The only way how each count is u but the real numbers are different is if a value overflowed. Assume there is an overflow in count $j \in [s-1]$. Since values wrap around on overflow and $A[j] = u$, the real number of count j is $u + 256z$ for an integer $z \geq 1$. These $256z$ extra keys cannot contribute to the other counts. This means another count has fewer keys than a valid splitting should have. However, each of the first $s-1$ counts is u, i.e., the real number is at least u. The only child node which could potentially miss some keys is the last count since it is never checked. However, its number of keys in a valid splitting is at most u which is smaller than 256. □

The fanout s ($2 \leq s \leq 9$) depends on the leaf size, the aggregation level and the size of the node. Remember that we use 32 bits for the hash function. Since one byte is sufficient for each count, we can store four counts per lane in a single vector by packing them in a 32-bit word. We do not need the last count, therefore a single vector is sufficient for $s \leq 5$, otherwise two vectors are sufficient. Given $j \in [s]$, we need to be able to increment the count $A[j]$. For this, we use the array

$$B = [0, 0, 0, 0, 1, 2^8, 2^{16}, 2^{24}, 0, 0, 0, 0, 0].$$

Each value in B is a 32-bit word. If $s \leq 5$, the count $A[j]$ is incremented by adding $B[4+j]$ to the packed counts. As usual, this happens in parallel for all SIMD lanes and j can be different for each lane. Note that $B[4+j]$ has a single bit set at the position where the count j begins. If $s = 5$ and $j = 4$, no count is incremented since the last count is irrelevant.

If $s > 5$, we need two vectors for the counts. The first vector contains the first four counts, the second vector the remaining counts. For $j \in [s]$, the first vector is treated as before by adding $B[4 + j]$. If $j \geq 4$, nothing changes. The value $B[j]$ is added to the second vector. For $j < 4$ or $j = 8$, nothing changes. Otherwise, the correct count is incremented.

Note that Theorem 5.2 is not enough to prove the correctness of this approach. If a count j overflows (i.e., reaches the value 256), the 256 increments are not lost completely. If the overflowing count is not the fourth count of the lane in one of the vectors (i.e., $j \notin \{3, 7\}$), a single bit is carried to the next count. However, this means 256 increments are lost for j and one extra increment is added to the count $j + 1$. Using the slightly stricter requirement $u < 255$, it is clear that the same proof can be applied to this situation since 255 increments are lost. Fortunately, we require $\ell \leq 24$, and from this requirement and the formulas to calculate the fanout given the leaf size we can conclude $s \leq 9$. Therefore, $u \leq 24$ for the first aggregation level and $u \leq 9 \cdot 24 = 216$ for the second aggregation level.

For the lookup in the array B, the `lookup` function of the Vector Class Library is used. Since $j \leq 8$, the values of B are stored in one or two vectors for faster lookup than in an array.

After all keys are processed, we need to check whether all counts (except the last) are u. For this, the value(s) of the 32-bit word(s) in the vector(s) that signify a valid splitting are calculated beforehand. Then, only one or two equality checks and a `horizontal_or` are necessary to decide whether at least one of the lanes has found a valid splitting.

Key Redistribution

After a valid splitting h is found, the keys must be sorted such that the first u keys are hashed to the first child node by h, the next u keys are hashed to the second child node and so forth. This is a form of bucket sort where the size of each bucket (child node) is known beforehand. First, the array C which indicates the position for the next key in each child node is initialized with the values $0, u, 2u, \ldots, (s - 1)u$. Afterward, two vectors are loaded with the next keys to redistribute. These vectors are treated as in Section 5.3.1 to calculate the corresponding child nodes. Note that while we use the same function to handle the vector, something different happens conceptually than when searching for splittings. Here, the same hash function is applied to several keys at the same time whereas different hash functions are applied to the same key at the same time when searching for splittings.

The resulting hash values are stored in an array to be processed further sequentially. Each key is stored in a temporary array at the position $C[i]$ where i is the hash value of the key. The position $C[i]$ is incremented before the next key is processed. This process is sequential since several keys in the vector may be hashed to the same child node. In this case, the same position is incremented several times and the keys need to be stored at different positions. This is not easy to achieve with SIMD.

When loading keys into the two vectors, both vectors are filled completely even if there are not enough keys remaining. In this case, keys from other nodes or a padding is loaded. This is not a problem, the process is aborted after the last real key is processed. The temporary array is then copied into the bucket.

5.6.3 GPU Implementation

Like on the leaf level, the aggregation levels on the GPU use one thread block per splitting. Finding a valid splitting works similar to the SIMD implementation. A word c is initialized to zero, where each thread has its own c. If the fanout s is at most five, c contains 32 bits, otherwise 64 bits. All counts are packed in c, one byte per count. Other than in the SIMD

implementation, no lookup table is used since memory is too slow for this use. Instead, count j ($j \in [s]$) is incremented by multiplying it first with eight before shifting a 1-bit by this value to the left and adding it to c:

$$c \leftarrow c + (1 << (j << 3)).$$

If $s = 5$ and $j = 4$ or $s = 9$ and $j = 8$, the shift amount is 32 or 64 bits, respectively. In this case, the computed value that is added to c should be zero. However, this is not guaranteed by the CUDA specification. To achieve this behavior, the cases $s = 5$ and $s = 9$ are treated slightly different than each other and the other cases. If $s = 5$, a funnel shift is used, in particular the intrinsic `__funnelshift_lc` [4]. It takes three 32-bit arguments. The first two are concatenated to form a 64-bit word, where the first argument are the least significant bits and the second argument the most significant bits. The third argument describes the number of bits this 64-bit value is shifted to the left. The result are the 32 most significant bits of the shifted value. A shift by 32 bits or more is well-defined to be equal to a shift by 32 bits, i.e., the first argument is returned. Therefore, the desired result can be achieved in the following way:

$$c \leftarrow c + \texttt{\_\_funnelshift\_lc}(0, 1, j << 3).$$

For $s = 9$, there is an extra branch to check if the shift amount equals 64 bits.

Alternative Implementation using Shared Memory

A different approach is to store the counts in shared memory. Each thread gets a portion of shared memory it can use as the array A containing the counts. Using Theorem 5.2, one byte per count is used. Remember that shared memory is partitioned into 32 banks. Each bank slot contains 32 bits. We include padding between the counts of different threads such that the first count of each thread begins at the start of a bank. Therefore, if $s \leq 4$, all counts of a given thread are within one bank. This means no bank conflicts are possible.

If $5 \leq s \leq 8$, each thread uses two banks to store the counts. This can cause bank conflicts on several occasions. First, each time the counts are initialized to zero there are s two-way bank conflicts. Note that the thread with index i within the warp ($i < 16$) and the thread with index $i + 16$ use the same two banks and therefore cause a bank conflict each time a count is set to zero. The number of bank conflicts and overall work can be reduced by treating each bank slot as a 32-bit integer to set it to zero instead of setting each byte to zero individually. This reduces the number of bank conflicts to two per initialization.

Another conflict occurs when checking if a valid splitting is found. Similarly to the initialization, s two-way bank conflicts occur each time. Again, the number of bank conflicts and overall work can be reduced by treating each bank as a 32-bit integer and computing the values of the banks which represent a valid splitting beforehand. The remaining number of bank conflicts is again two.

The last conflict occurs each time a counter is incremented. This is not guaranteed to cause a bank conflict every time. However, the probability that there are two threads i and $i + 16$ of a warp such that both access the first or both access the second bank is quite high. For example, if $s = 8$, the chance is equal for both banks. Remember that it is pseudorandom for each thread which count is accessed. Therefore, the probability of a given thread pair i and $i + 16$ to cause a bank conflict is 50%. Since there are 16 such pairs per warp, the probability that a warp causes a bank conflict when loading or storing a count is more than 99.99%. To increment a counter, it must first be loaded from shared memory, incremented and then stored to shared memory. This means there are two two-way bank conflicts (almost) each time a counter is incremented.

0	1	2	3	4	5	6	7
0-0	0-1	1-0	1-1	2-0	2-1	3-0	3-1
PAD	4-0	4-1	5-0	5-1	6-0	6-1	7-0
7-1							

(a) Two banks per thread with the padding in red.

0	1	2	3	4	5	6	7
0-0	0-1	0-2	1-0	1-1	1-2	2-0	2-1
2-2	3-0	3-1	3-2	4-0	4-1	4-2	5-0
5-1	5-2	6-0	6-1	6-2	7-0	7-1	7-2

(b) Three banks per thread.

Figure 5.1: Shows the assignment of the counts to the banks. The left figure depicts the situation if two banks per thread are used, the right figure if three banks per thread are used. For illustration purposes, only 8 threads per warp and 8 banks are assumed. Each column is a bank with the bank number at the top. Every bank slot is labeled with the thread ID followed by the bank number of this thread, i.e., 2-1 is the bank slot of the second bank of thread 2 (zero-indexed). There is no bank conflict if each thread accesses its first (or second or third) bank at the same time since the second number in each column is different from the other numbers in the same column.

Fortunately, bank conflicts can be avoided completely for the first two occasions by using a padding after the first 16 threads per warp. The padding consists of a single unused bank. This has the effect that the first bank of a thread in the first half of a warp is the same bank as the second bank of a thread in the second half and vice versa. When each thread accesses its first bank, every thread accesses a different bank of the 32 total banks. For example, thread 0 accesses bank 0, thread 16 accesses bank 1, thread 1 accesses bank 2 and so forth. Therefore, no bank conflicts occur at initialization and when checking if a splitting was found. See Figure 5.1a for a visualization.

The padding can also decrease the likelihood of a bank conflict when incrementing a count. For example, consider $s = 5$. We perform an analysis similar to two paragraphs above. If no padding is used, the probability that a given thread pair $(i, i + 16)$ within a warp cause a bank conflict is $\frac{4}{5}^2 + \frac{1}{5}^2 = 0.68$. The reason is that each thread accesses its first bank with a probability of $\frac{4}{5}$ and its second bank with a probability of $\frac{1}{5}$. Therefore, the probability that any of the 16 thread pairs in a warp cause a bank conflict is more than 99.99999%.

With padding, the odds are changed. The probability that a given thread accesses its first bank is still $\frac{4}{5}$, but for the other thread that uses the same bank, this is its second bank and is therefore accessed with a probability of $\frac{1}{5}$. The probability that a given thread i with $i < 16$ experiences a bank conflict is therefore $2\left(\frac{4}{5} \cdot \frac{1}{5}\right) = 0.32$. Consequently, the probability of a warp causing a bank conflict is $1 - (1 - 0.32)^{16} \approx 0.998$. Arguably, the difference to the version without padding is negligible. However, it shows that the padding at least does not worsen the situation.

If $s = 9$, each thread uses three banks to store the counts. No padding (apart from the padding which ensures that the first count of each thread begins at the start of a bank) is necessary to avoid bank conflicts. This has to do with the fact that 32 (the total number of banks) and 3 (the number of banks used by each thread) are relatively prime. See Figure 5.1b for a visualization. The only situation where bank conflicts can occur is when incrementing a counter. This, however, cannot be avoided. Unfortunately, even three-way bank conflicts are possible in this situation, but using no padding is the best we can do since it ensures that for each of the 32 banks there is exactly one of the 32 threads whose third bank it is.

Overall, after applying all aforementioned optimizations, the only place in the whole construction where bank conflicts can occur are when incrementing a counter in the aggregation levels (if $s > 4$), atomically storing the hash function identification in the result if a valid splitting or bijection was found (which rarely happens) and when accessing positions atomically to redistribute the keys after a valid splitting or bijection was found (see next paragraph). Nonetheless, the experiments show that storing the counts in shared memory is slower than using a single 32- or 64-bit word which contains all counts in a packed form; see Section 6.5.3. This is due to the unavoidable bank conflicts and the fact that shared memory is generally slower than registers. Therefore, the counts are not stored in shared memory in our final implementation.

Key Redistribution

After a valid splitting h is found, the keys are written back to global memory sorted by their child node number. For this, an array $C \leftarrow [0, u, 2u, \ldots, (s-1)u]$ of size s is initialized. Each thread loads the key at the position of the bucket indicated by its thread index (if available) from shared memory, calculates the child node number, reads and increments the according counter in C atomically and writes the key to the global memory at the position indicated by the counter. This process is repeated for the next keys by adding the number of threads in a thread block to the last index until the end of the bucket is reached.

Since all threads of a thread block access the same s counters in C concurrently, there is a lot of contention expected. The problem can be alleviated by using warp-aggregated atomics [10]. Using this technique, the threads of a warp calculate how many of the threads increment each counter. Then, only one of the threads per counter atomically increases the counter by the calculated number. This decreases the number of atomic operations needed significantly. The correct position for each thread is then computed by broadcasting the old value to each thread corresponding to the counter and adding the index of the thread within the group of threads corresponding to the counter.

This technique can be implemented quite easily using cooperative groups [9]. Each thread iterates over the values from 0 to $s-1$ and checks if the value is equal to the child node number of the current key. In the iteration where this is the case, all threads in the same warp with the same child node number are coalesced, i.e., are active while all other threads in the warp are inactive.[1]

However, warp-aggregated atomics do not improve the construction time; see Section 6.5.4. The reason is that, as noted in the Nvidia developer blog [10], current versions of the NVCC compiler automatically apply warp-aggregated atomics in some cases if applicable. The code generated by NVCC is faster than the hand-written code. Therefore, the warp-aggregated atomics technique is not used explicitly in our implementation. As a side note, testing showed that the hand-written code seems to be faster if the counters are 64-bit words instead of 32 bits. But this is not relevant for our implementation.

5.7 Higher Levels

The higher levels, i.e., the splittings above the two aggregation levels, are relatively simple. The fanout is always two. Therefore, no array is necessary to store the counts. Nonetheless, the original RecSplit implementation [13] uses an array of size two. Given the keys X with $|X| = m$, the unit u (the number of keys which should be hashed to the first child node) and a hash function $h\colon X \to [m]$, the following algorithm is used to check whether h is a valid splitting:

[1] Strictly speaking, not all threads with the same child node number must be active. They could be divergent, such that the same atomic is increased several times by the same warp within one iteration [11].

1. Initialize array A with two words that are each equal to zero. The values in A are called counts.
2. For each $x \in X$, increment $A[h(x) \geq u]$ by one, where $h(x) \geq u$ is one if $h(x)$ is greater than or equal to u and zero otherwise.
3. The function h is a valid splitting if and only if $A[0] = u$.

Instead of using an array for the two counts, the SIMD and GPU implementation use a single counter c which is incremented if and only if $h(x)$ is smaller than u.

5.7.1 SIMD Implementation

Usually, the boolean value "true" is encoded as a one, whereas the boolean value "false" is encoded as a zero. This is used to conditionally increment the counter c based on the result of the comparison of $h(x)$ and u without using a branch. In the Vector Class Library, however, the boolean value "true" is encoded as the word containing only 1-bits. This represents a -1 in two's complement. Therefore, the hash function h is a valid splitting if and only if $c = -u$ after processing all keys in the node.

To avoid confusions, the addition of a negative value instead of the usual one is made explicit by using the `if_add` function of the Vector Class Library [27]. It takes the condition as first argument and adds the third argument to the second if the condition is true and returns the second argument unaltered otherwise. It is used to increment the counter as follows:

$$c \leftarrow \texttt{if\_add}(h(x) < u, c, -1).$$

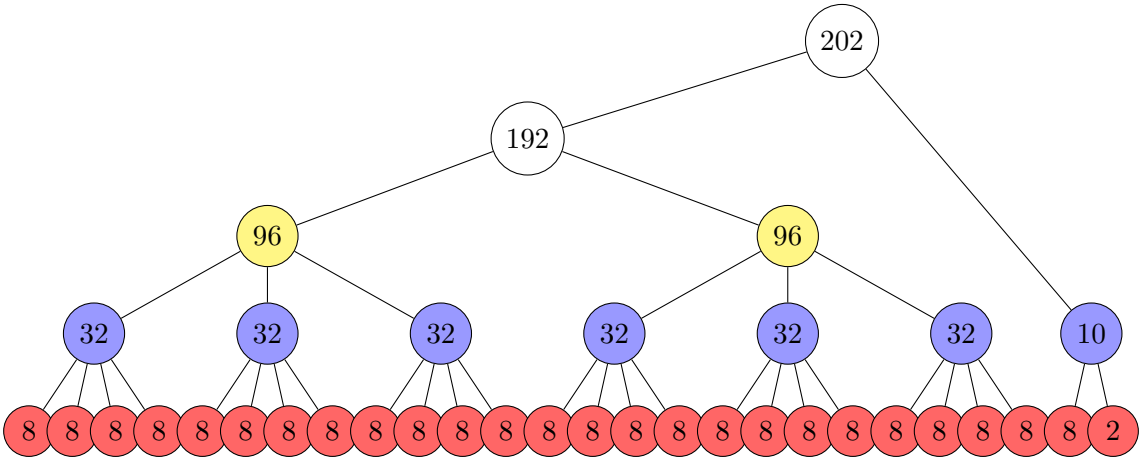
The resulting machine code is identical to adding the condition directly to c. Other implementations, like using `if_sub` to subtract one or using `if_add` to add a one and later comparing with u resulted in more complicated machine code.

The key redistribution works similar to the aggregation levels. However, after applying the hash function and comparing the value to u, the result is not zero or one, but zero or minus one. This must be considered when using this value to store the key at the correct position in the temporary array. Instead of storing the complete result of the comparison in an array to be further processed, the values are reduced to a single bit per lane using the function `to_bits` of the Vector Class Library. This is necessary because the Vector Class Library does not permit storing a boolean vector in an array. The result is a word d containing a bit for every lane which is one if and only if the condition is true. We iterate over the bits of this word to get a zero or one for each key in the currently processed vector, which indicates whether it is hashed to the first or the second child node. In iteration j, the bit is calculated as $(d >> j)$ `&` 1.

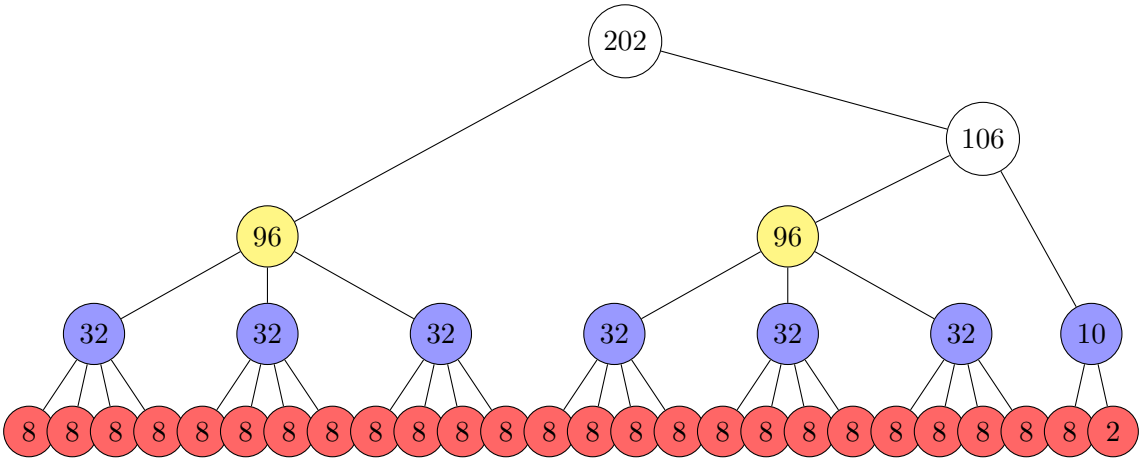
5.7.2 Balanced Splittings

Section 4.3 describes the shape of a splitting tree and provides Figure 4.1 as an example, which is depicted again in Figure 5.2a. Looking closely, it is revealed that the tree is not as balanced as possible. The right child of the root splitting could receive 96 additional keys without violating any properties. The result is shown in Figure 5.2b. Neither the number of splittings nor anything in the lower levels has changed. The only thing that has changed are the two splittings in the higher levels. In the following, we analyze this particular example to show why balanced splittings are useful, which makes sense intuitively.

We begin with the space consumption. As discussed in Section 4.4.1, the space consumption of splitting trees is optimal apart from about two bits which are lost per splitting/bijection. Since the number of splittings and bijections has not changed, the expected number of



(a) The original splitting tree as used by the original RecSplit implementation [13]. This is the same illustration as Figure 4.1.



(b) The balanced splitting tree as used by the GPU and SIMD implementation.

Figure 5.2: The unbalanced (top) and balanced (bottom) splitting tree with 202 keys for leaf size $\ell = 8$. Each node is labeled with the number of keys it contains.

used bits should not change significantly. Indeed, using the formulas in Section 4.3.1, the expected number of hash functions that must be tried to find a valid splitting for the root is 7.728 for the original splitting tree and 17.791 for the balanced tree, whereas the other splitting in the higher level requires on average 17.366 or 7.544 trials, respectively. This means the balanced tree needs about 2.302 times as many trials for the root splitting, but the original tree needs about 2.302 times as many trials for the other splitting. Since the number of trials dictate the space consumption, the balanced tree should need about the same space.

When looking at construction time, the balanced tree is better. We use the expected number of hash function evaluations as the cost metric. The original tree needs about $202 \cdot 7.728 + 192 \cdot 17.366 \approx 4895$ evaluations, the balanced tree $202 \cdot 17.791 + 106 \cdot 7.544 \approx 4393$. Therefore, the balanced tree can be constructed faster. However, this difference is practically negligible. A single full leaf of this tree needs about $8 \cdot \frac{8^8}{8} \approx 3329$ hash function evaluations and there are 25 such leaves.

The largest improvement is the query time. The balanced splitting can reduce the average number of splittings that must be evaluated. Consider a random key from a bucket containing such a splitting tree. For the original tree, there are on average $\frac{192 \cdot 4 + 10 \cdot 2}{202} \approx 3.90$ splittings that must be evaluated to compute the hash value. Contrarily, the balanced tree only requires $\frac{96 \cdot 4 + 106 \cdot 3}{202} \approx 3.48$ splitting evaluations. This means we can save on average 0.4 hash function evaluations for this particular splitting tree. Of course, the concrete numbers depend on the splitting tree, i.e., on the number of keys and ℓ. The experiments confirm the analysis, see Section 6.3.3.

Balanced splittings in the higher levels can be implemented without any additional computations. Consider a splitting in the higher levels with m keys. Let η be the number of keys of a full node in the second aggregation level, i.e., 96 in the example before. In the original implementation of RecSplit [13], the number of keys of the left child is calculated as

$$\eta \left\lceil \frac{\lfloor \frac{m}{2} \rfloor}{\eta} \right\rceil = \eta \left\lfloor \frac{(m >> 1) + \eta - 1}{\eta} \right\rfloor$$

where rounding down is achieved by using regular integer division. Essentially, $\lfloor \frac{m}{2} \rfloor$ is rounded up to the next multiple of η. If $\lfloor \frac{m}{2} \rfloor$ is just slightly larger than a multiple of η, the splitting is rather unbalanced. A balanced splitting can be achieved by rounding to the next multiple of η, be it down or up. This is possible with the formula

$$\eta \left\lfloor \frac{(m >> 1) + (\eta >> 1)}{\eta} \right\rfloor.$$

Note that η is a compile-time constant, therefore the exact same operations are done as in the original formula, just another constant is added to $m >> 1$.

5.8 CPU Parallelization

We have seen all the required steps to find the splittings and bijections of a splitting tree, which is usually the most time-consuming part of the construction. The original RecSplit implementation [13] 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 construction of splitting trees is completely independent of each other for different buckets. We parallelize SIMDRecSplit by spawning several threads and assigning a consecutive portion of the buckets to each thread. Each thread needs to know the beginning of its first bucket in the input which is fortunately provided after sorting the input with counting sort (see Section 5.1).

Until they are finished processing their buckets, all threads work completely independent of each other; no synchronization is required. However, after a splitting or bijection is found it must be stored in the Rice Bit Vector. To avoid synchronization, each thread uses its own local Rice Bit Vector and treats its input as it was the complete input. This means it also stores the prefix of the number of bits of each of its buckets in the corresponding array P. The values in P must be fixed before computing the Double Elias-Fano representation.

After a thread has processed all of its buckets, it tries to enter a critical section. Only one thread is allowed in the critical section at the same time. Furthermore, it is ensured that the threads enter the critical section in the order of their thread ID. The first thread only stores its local Rice Bit Vector in the global Rice Bit Vector by setting the respective pointers correctly. The other threads append their local Rice Bit Vector to the global Rice Bit Vector in the critical section. After leaving the critical section, they fix their values in

P by adding the last value of the previous thread (which can be retrieved from the Double Elias-Fano representation) to all their values.

The critical section is implemented by using a mutex and a condition variable. To enter the critical section, the mutex is locked using a `std::lock_guard` [18] of the C++ Standard Library. This operation blocks the calling thread until no other thread is in the critical section, locks the mutex and automatically unlocks the mutex after the `std::lock_guard` goes out of scope. Apart from the first thread, all threads use the condition variable after locking the mutex to check whether it is their turn, i.e., whether all previous threads have already finished the critical section. This is simply implemented as a check whether a shared integer o contains the own thread ID. If it is not the threads turn yet, it gets blocked by the condition variable until it gets notified. After the Rice Bit Vector is stored/appended in the critical section, each thread increments o and notifies the waiting threads using the condition variable.

The GPU implementation uses this form of CPU parallelization as well. This is useful if the bottleneck is mostly on the CPU side. The CPU is responsible for preparing the data, launching the kernels and encoding the results in the Rice Bit Vector. The CPU may be the bottleneck for small leaf size, small bucket size or the combination of several powerful GPUs with a slow CPU. However, the overall effect on performance seems small. The GPU implementation is mainly useful for relatively high leaf and bucket sizes; see Section 6.6.1. Therefore, the effect of CPU parallelization on the construction time of the GPU implementation is not formally evaluated using experiments.

5.9 Double Elias-Fano

After all buckets are finished, the Double Elias-Fano representation [45, 47, 46] is constructed (see Section 4.5). It contains the prefix of the number of keys in each bucket and the bit position where each bucket begins in the Rice Bit Vector. It contains four loops, all of which can be significantly improved. The first loop computes the minimum of the differences between two consecutive values in the sequences (δ_λ and δ_ψ). It can be accelerated using SIMD.

The second loop stores the values in three different bit arrays, the lower-bits array and the two upper-bits arrays. Using SIMD for this loop is a lot more complicated since for each value only a part of a word must be manipulated. Therefore, the same word may be accessed for consecutive values. By vectorizing the loop in a straightforward way, information could be lost. However, the loop does contain other operations which can profit from SIMD (shifts, applying masks, calculating indices, etc.). Only the parts which cannot be vectorized are serialized.

The third loop computes the jump array for the prefix sum of the number of keys. This array is used as the selection data structure to accelerate access to the upper-bits arrays. Let κ be the number of buckets. The loop iterates over each bit of the upper-bits array for the prefix sum. This array contains about 2κ to 3κ bits, empirically and using the formulas derived in Section 4.5.2. Exactly κ bits are set to 1. If a bit is not set, nothing needs to be done. This can cause a lot of branch mispredictions. Moreover, only for every 256th 1-bit any real work (apart from checking the bit and increasing a counter) must be done.

The unnecessary work of checking every bit can be avoided by using popcount on 64-bit values until 256 1-bits are found. Then, the exact position of the 256th bit must be computed. The selection algorithm (see Section 2.1) can be used for this by applying it on the 64-bit word that contains the 256th 1-bit. The computed value can be used to set the jump array. The fourth loop has exactly the same structure and computes the jump array for the bit positions. It can be optimized in the same fashion. The result of these optimizations is shown in Section 6.3.4.

5.10 GPU Implementation - Overview

We now have all the building blocks we need to describe the procedure to compute a minimal perfect hash function of the input using the GPU. First, the input is sorted as explained in Section 5.1 to receive an array A of 64-bit keys and another array which indicates where each bucket begins in A. We then reserve enough space on the host as well as on the device to temporarily store 128 buckets and the resulting hash function identification numbers. The memory on the host side must be allocated as *pinned* (also called *page-locked*) memory in order to allow asynchronous data transmission between host and device. Additionally, the memory which is used as a buffer to be transferred to the device is allocated as *write-combining* memory. It is expected to provide faster transfers by avoiding bus snooping. It is applicable here since the host only writes to this memory [3].

The memory on the device is allocated as one chunk. Like the original RecSplit implementation [13], we assume the maximum number of keys in a bucket is 3000. This number is also an upper bound of the number of splittings and bijections in a single bucket, i.e., the number of hash functions to store. The largest number of splittings and bijections is achieved for 3000 keys in a bucket and a leaf size of 2. Note that in this case, the splitting tree is a binary tree where each node (except the leaf nodes) has two child nodes. Such a tree with k leaf nodes has exactly $2k - 1$ nodes, which can be proven using induction. For 3000 keys and leaf size 2, the number of leaves is 1500 which means 2999 splittings and bijections. Since each splitting and bijection is stored as a 64-bit value before encoding it in the Rice Bit Vector later, we reserve the same amount of space as for the bucket keys.

Overall, the global memory space consumption on the device is $2 \cdot 128 \cdot 3000 \cdot 8$ bytes $= 6\,144\,000$ bytes. Funnily enough, the amount of L2 cache on the Nvidia RTX 3090 (which we use in our experiments) is 6144 KiB [17]. Because a KiB refers to 1024 bytes, the available amount of L2 cache is slightly larger than the overall consumption. Moreover, usually only a fraction of the memory is used since buckets are mostly smaller than 3000 keys and the number of splittings and bijections is much smaller for higher leaf sizes due to higher fanouts in the aggregation levels.

The implementation uses 128 CUDA streams, i.e., up to 128 buckets are in the pipeline to being sent to the device, being processed by the device and the results sent back to the host. For each stream, there is an associated pinned and write-combined memory region on the host to buffer the bucket, a pinned memory region on the host to buffer the results and a memory region on the device to store the bucket and the results. Each thread on the host uses a portion of the streams to process its buckets. It iterates through its streams in a round robin fashion. If a stream has not been used yet, it is created using the CUDA runtime. Otherwise, the thread synchronizes with the stream, i.e., waits until all previously launched operations in the stream are finished. Then, the results (the hash identification numbers of all splitting and bijections in the splitting tree of the bucket) are in the respective buffers and can be encoded in the Rice Bit Vector.

After creating or synchronizing with the stream and encoding the results, the thread copies the next bucket into the buffer — if there is a next bucket. It then launches the necessary operations to the stream. These operations are all asynchronous, i.e., they immediately return control to the calling thread and are executed by the CUDA runtime when the necessary resources are available. The first operation is transferring the buffer containing the bucket to the device memory. The following operations launch the different kernels. At first, the higher levels are launched. This is done with a recursive function. If the size of the bucket is greater than the node size of the second aggregation level, a higher level kernel is started before recursively calling the same function again twice. The kernel is started with a single thread block which computes a single splitting. Streams help to still

utilize the hardware. For each recursive call, the size, starting index of the input, index of the result and the level (for looking up the starting seed) are adapted.

In the original implementation [13], such a recursive function was used to create the complete splitting tree. However, the leaf level and the two aggregation levels are treated differently than the higher levels. Instead of launching a kernel for every single splitting/bijection, one kernel per level is launched with as much thread blocks as there are splittings/bijections on the level. This decreases launch overhead and can increase utilization since the number of concurrent kernels is limited on CUDA GPUs [3]. 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 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 would use kernels with more than one thread block on the higher levels where each thread block computes a different splitting. 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.

As explained above, the three lowest levels are processed by starting one or two kernels per level. Note that the exact number of launched kernels depends on the bucket size and the leaf size. For a small bucket, there may only be a single leaf and therefore only a single kernel launched. However, this can usually not utilize the device completely. For maximum utilization and minimal overhead, the bucket size and the leaf size should be as large as possible. This also results in the most space-efficient MPHF. After all kernels are launched into the stream, the result is scheduled to be transferred to the host.

The sequence in which the hash function identifiers are stored in the result is equal to the sequence in which the kernels are launched into the stream and for each kernel the thread blocks from first to last. This is a different sequence than what the Rice Bit Vector expects. To achieve a correct result, the values must be encoded in the Rice Bit Vector in preorder, i.e., first the root, then its left child and its left child and so forth until the first leaf, after which the second leaf is encoded, the second node on the first aggregation level, its first child and so forth. This is exactly the sequence we get when calculating the splittings and bijections in a recursive function. However, we only use a recursive function for the higher levels.

The remaining levels are stored following the results of the higher levels in a breadth-first order, i.e., first all splittings of the second aggregation level from first to last node, then all splittings from the first aggregation level and finally all bijections of the leaves. To correctly unpack the results and encoding them in the Rice Bit Vector, the beginning of each of these segments is documented in an array for later use. After synchronizing with the stream to wait for the results, a recursive function is used to unpack the results. This time, the recursion ends at the leaves and not earlier. This function uses the stored beginnings of each segment to encode the results in the correct order.

After a thread has processed all of its buckets, it synchronizes with the other threads as explained in Section 5.8. After all threads are finished, the allocated memory on the CUDA device is freed. Finally, the Double Elias-Fano representation is constructed as explained in Section 5.9 using the calculated arrays containing the number of keys and the number of bits in the Rice Bit Vector for each bucket in a prefix-sum form.

5.10.1 Batched Memory Transfer

As explained earlier, the input is transferred bucket by bucket to the device. The usual advice is to batch smaller memory transfers into one larger memory transfer to reduce overheads associated with each transfer [3]. Depending on the bucket size, each transfer to the device only contains a few bytes to a few kilobytes. Batching these transfers could reduce the overhead significantly.

This can be achieved by starting one memory transfer for all streams before the kernels of the first stream are launched. All buckets are packed contiguously to avoid transferring unused bytes. This means the starting address of each bucket in global memory must be adapted. Without batched transfers, each bucket just starts $3000 \cdot 8$ bytes after the bucket before it. To make sure that the kernels in each stream only start processing after the memory transfer is completed, a CUDA event [3] is used. This event is recorded in the same stream as the memory transfer and signals whether the transfer has finished. For each bucket, a synchronization call is inserted before the first kernel launch to wait for the event.

When the thread arrives at the first stream again, it processes the results of all streams. This means the thread iterates over the streams, synchronizes with each stream and encodes its results in the Rice Bit Vector. After all results are encoded, the next batch can be transferred. Since the thread synchronizes with all streams before starting the next transfer and launching new kernels, the device and the bus between host and device are not fully utilized. A possible solution is to use two transfers per batch, i.e., starting a transfer for the first half of the streams and a transfer for the second half. When encoding the results, only the first half is considered at first before starting the next transfer and launching new kernels. In the meantime, the second half of the streams can continue utilizing the resources. In our case, this is not necessary because we use several CPU threads (see Section 5.8) which has the same effect.

Testing showed no significant improvements using batched memory transfers. For relatively large leaf and bucket sizes, the bottleneck is finding the splittings and bijections. The memory transfers are insignificant, which was also confirmed using the profiling tool “Nvidia Nsight Systems” [23]. Configurations with smaller leaf and bucket sizes could theoretically profit more from batched transfers, but they still have high overheads and cannot utilize the GPU fully due to many small kernel calls. In these cases, SIMDRecSplit is faster than GPURecSplit; see Section 6.6.1. Thus, the GPU implementation is not optimized further for small leaf and bucket sizes if the benefit is not very clear. This means batched memory transfer is not used in our final implementation. In particular, batching the memory transfers of the opposite direction — transferring the results from the device to the host — was not implemented in the first place.

5.10.2 Work Queues

The maximum number of kernels using the same GPU at the same time is limited. For the Nvidia RTX 3090, which we use in our experiments, this number is 128 [3]. However, the number of work queues (concurrent connections) is by default only 8. All operations launched into a given stream are appended to one work queue. Since the operations in the stream are processed sequentially and there are several streams per work queue, the maximum number of concurrent kernels is effectively limited to 8. This severely limits the utilization of the GPU since we rely heavily on the fact that several kernels can use the same GPU at the same time. A single kernel, especially of the higher levels, may only utilize a small portion of the GPU since the number of thread blocks is rather small.

Fortunately, the number of work queues can be adjusted by setting the environment variable CUDA_DEVICE_MAX_CONNECTIONS to the desired value [3]. A higher value may

increase utilization but has higher overheads. The experiments showed that 32 work queues provide good performance for the most relevant parameters of the GPU implementation, i.e., relatively high leaf and bucket sizes since for smaller values the overheads are high anyway and utilization is low; see Section 6.5.5. Therefore, 32 work queues are used in our experiments by setting the environment variable using the `export` command [19]. It is the responsibility of the user to set the environment variable as convenient.

5.11 SIMD Implementation - Overview

The structure of the SIMD implementation is similar to the GPU implementation. First, the input is sorted. Some threads are started which immediately begin to process their buckets. For each bucket, a recursive function is called which itself calls the functions of the respective levels based on the current size before recursively calling itself with adapted parameters. Contrary to the GPU implementation, the recursion ends at the leaves. This is much simpler and also resembles the construction in the original implementation [13]. After a thread has processed all of its buckets, it synchronizes with the other threads as explained in Section 5.8. After all threads are finished, the Double Elias-Fano representation is constructed.

6. Evaluation

We evaluate the performance of our new RecSplit implementations and compare them to the original RecSplit implementation in the Sux library [13]. We begin with the experimental setup in Section 6.1, followed by us fixing a performance bug in the original RecSplit implementation in Section 6.2. Then, we conduct various experiments concerning the SIMD and GPU implementation in Section 6.3. The goal is to experimentally verify the usefulness of different approaches. The next section shows experiments specifically for the SIMD implementation (Section 6.4), followed by experiments for the GPU implementation in Section 6.5. We compare both implementations to the original RecSplit implementation in Section 6.6. Finally, we show a small comparison with PTHash (see Section 3.4).

6.1 Experimental Setup

Hard- and Software

If not said otherwise, the machine used for the experiments features an Intel Core i7-11700 CPU with 8 cores and 16 threads (hyper-threading activated), 2.5-4.9 GHz, 48 KiB L1 data cache per core, 32 KiB instruction cache per core, 512 KiB L2 cache per core and 16 MiB L3 cache overall [5, 20]. It has access to AVX-512 instructions, including AVX512VPOPCNTDQ. The machine has 64 GiB of dual-channel DDR4-3200 RAM and an Nvidia RTX 3090 GPU [17]. The experiments are performed using Ubuntu 21.10. All RecSplit implementations are implemented in C++; the GPU implementation uses CUDA C++. GCC 11.2.0 [55] is used as the C++ compiler and NVCC from CUDA 11.7 [71] as the CUDA compiler. The compiler options `-march=native` and `-O3` are used.

General Setup

As a reminder, only the RecSplit 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. The times are measured by wall clock using the `chrono::high_resolution_clock` from the C++ standard library. Generally, each configuration tested in this chapter is repeated five times and the shown data points depict the average of these five repetitions with the standard deviation as a vertical line. The inputs for the different repetitions are different, but the input is identical for the same configuration and repetition of different implementations in the same plot. This is done to measure the effect of the input on the running time, while ensuring fairness between different implementations, i.e., different lines in the same plot.

The input consists of pseudorandom 128-bit keys, and the initial hash function (see Section 4.1) is skipped. This is reasonable because the result of the initial hash function is treated as a 128-bit random key anyway.

When measuring query times, 100 million random queries are performed and the total time is divided by 100 million to receive the average query time. The reported standard deviation is therefore not the standard deviation of single queries, but the standard deviation of five data points which are the average of 100 million queries each. Note that the input of a query may not be part of the original set used in the construction. This should not have an impact on performance since the distribution is equal. The input of each query depends on the output of the previous queries (the input is XORed with a variable that is the XOR of all previous outputs). This avoids the overlapping of instructions for different queries (pipelines, out-of-order execution, superscalar processors, etc.) such that the resulting average is a more realistic representation of the time a single query takes, i.e., its latency. The query benchmark is adopted from the original RecSplit implementation in the Sux library [13].

If not said otherwise, the implementations are configured as described in the previous chapter. An exception are balanced splittings (see Section 5.7.2). Since those were only discovered after most experiments have already been performed and it was not feasible to redo each experiment, they are not used in most experiments. Per default, SIMDRecSplit uses as many CPU threads as the CPU has hardware threads (in our case 16), and GPURecSplit uses 8 CPU threads. The number of threads per thread block is 64 in the higher levels, 256 in the aggregation levels, and 512 in the leaf level. Those numbers were chosen through testing.

Nomenclature

The names shown in the legends follow a specific naming scheme:

- Prefix:
  - "orig": the original implementation
  - "simd": the SIMD implementation
  - "gpu": the GPU implementation
- Infix (or suffix):
  - "Rot": bijection rotation is used, see Section 5.5
  - "NoRot": bijection rotation is not used
- Optional suffix: depends on the plot and is used to distinguish different configurations of the same RecSplit implementation

For example, "origRot" is our modified version of the original implementation with bijection rotation.

6.2 Performance Bug in the Original Implementation

As explained in Section 2.1, selection on a 64-bit word can be implemented with just three instructions if the instruction set extension BMI2 is available. If not, the Sux library contains a slower fallback which is used by RecSplit [13]. Unfortunately, there is a bug in the compile-time detection of the availability of the required PDEP instruction. The code checks if the macro `__haswell__` is defined. The rationale is that Intel Core CPUs support the BMI2 instruction set for Haswell and newer architectures. However, GCC

only defines this macro if compiled for the Haswell architecture specifically, e.g., with the compiler option `-march=haswell`. If compiled for newer Intel Core architectures, AMD architectures, or if just the option `-mbmi2` is used to activate BMI2, the macro `__haswell__` is not defined and thus only the slower fallback is used.

The existence of macros defined by the compiler can be checked for example with `echo | g++ -dM -E -march=skylake - | grep haswell -i`, which returns nothing for GCC 9.3.0 [55] and Clang 10.0.0 [2] (by replacing `g++` with `clang`) on Ubuntu 20.04.1 LTS. In fact, Clang does not even define `__haswell__` if the option `-march=haswell` is used. The experiments in the original RecSplit paper [46] were compiled with GCC 8.8.1 and performed on an Intel Core i7-7770. We are not aware of any CPU with this name, but due to the name similarity we assume it has the same architecture as the Intel Core i7-7700 [6] (it may just be a typo). The makefile in the Sux library uses `-march=native` to set the architecture at compile time. If the authors of RecSplit used this makefile and compiled their benchmarks on the same machine as the experiments were performed, the fast selection algorithm was probably not used.

We fix this bug by checking for the macro `__BMI2__` instead, which is defined if BMI2 is available and activated, e.g., with `-march=native` on a machine where BMI2 is available or `-mbmi2`. The only place in the RecSplit construction where the fast selection algorithm is used is our improved construction of the Double Elias-Fano representation 5.9. This is such a minor part of the whole construction that this bug fix has no significant effect on the running time. However, the query algorithm uses the fast selection algorithm twice when querying the Double Elias-Fano representation and once every time a subtree is skipped, see Section 4.5. Therefore, using the fast selection algorithm has a significant impact on the performance of the query algorithm.

Figure 6.1 shows the improvement by fixing the performance bug. The "originalImplementation" is the unaltered implementation from the Sux library [13]. The "origNoRot" implementation is the original implementation integrated into our framework, with the performance bug fixed and with the option to use bijection rotation (in this case deactivated). The fixed version is 10–20% faster. In all following measurements, we refer to the fixed version as the original implementation.

A similar performance bug is in the implementation of `clear_rho`. We use the correct instruction to delete the least significant 1-bit instead of the fallback `clear_rho`$(a) = a$ `&` $(a-1)$ by replacing the macro `__haswell__` by `__BMI__`. However, this makes no difference using current versions of GCC and Clang since those are able to transform the fallback to the single instruction automatically if it is available.

6.3 Evaluating Different Techniques

This section evaluates different techniques concerning both of our implementations, i.e. SIMDRecSplit and GPURecSplit.

6.3.1 Sorting

We have seen how to improve the sorting method to distribute the keys into the buckets in Section 5.1. As the plots in 6.2 show, the speedup of our faster sorting method compared to the original method is up to 12. Even with this faster method, the sorting can still take more than 70% of the total SIMDRecSplit construction time when using multithreading for the construction of the splitting trees. However, this is only the case for small leaf sizes (in 6.2 leaf size $\ell = 5$) and small bucket sizes. The percentage of the total construction time is smaller for larger leaf sizes since the sorting time is unchanged whereas computing splitting

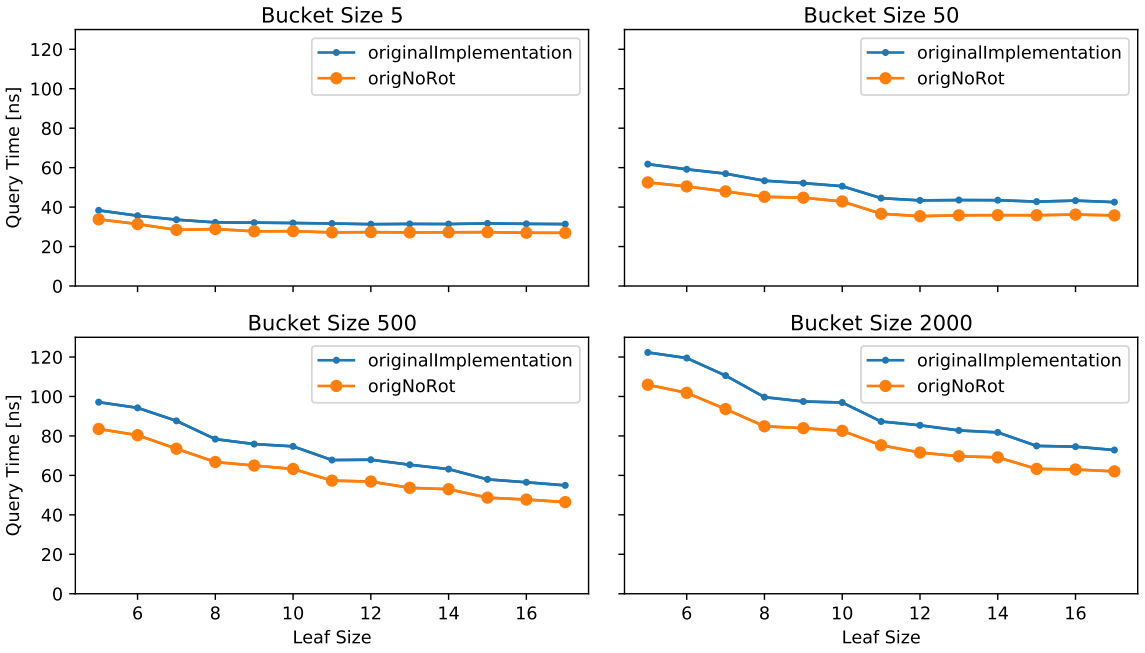


Figure 6.1: Query times of the original implementation before and after fixing the performance bug of the selection algorithm. The lines of "originalImplementation" are before fixing the bug and "origNoRot" after fixing the bug. The MPHF was constructed with one million keys.

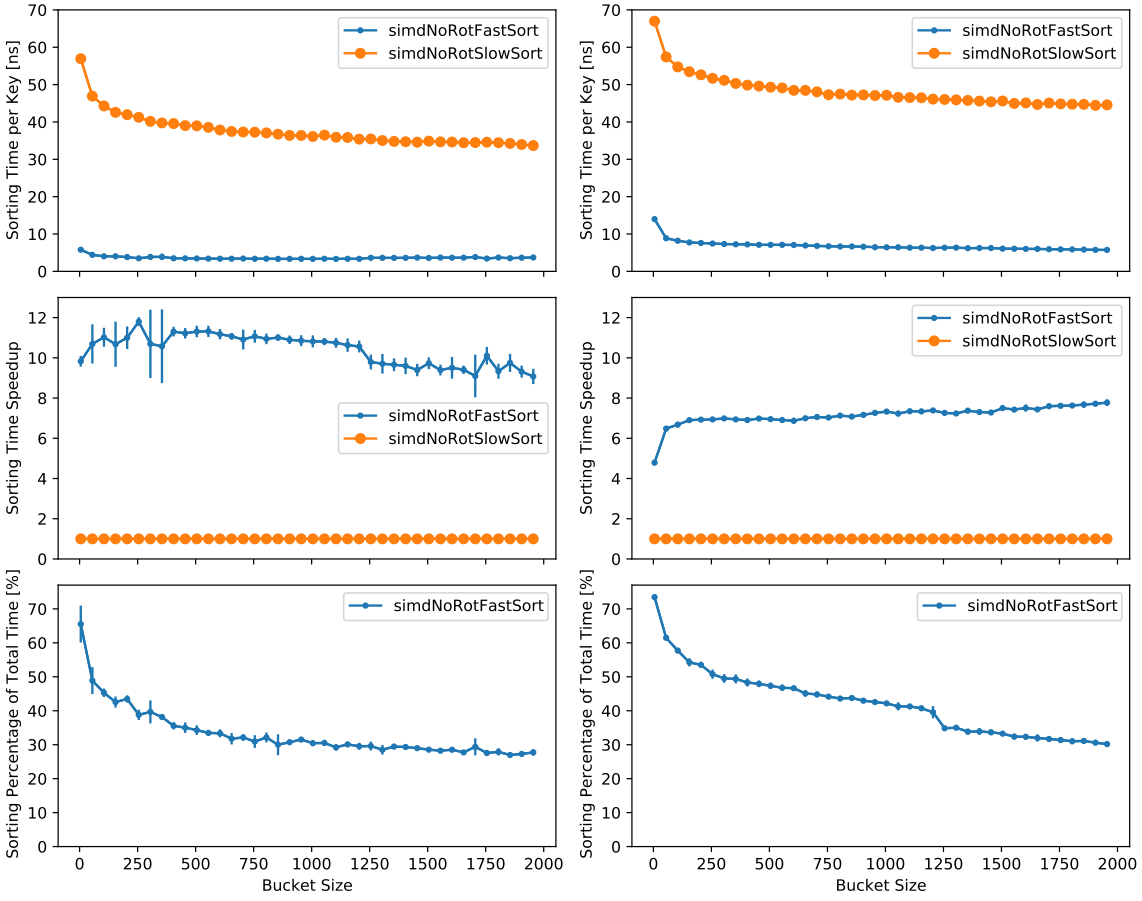
trees becomes more expensive. Similarly, larger bucket sizes decrease the percentage because sorting gets faster due to better cache efficiency and a smaller array which holds the prefix sum of the bucket sizes. The percentage in the original implementation is generally smaller since the construction of the splitting trees takes more time.

The speedup is smaller for ten million keys than for one million keys. This is probably due to caching effects. One million keys occupy 16 MB. The output of the counting sort algorithm only needs about half of that since only the least significant bits of the keys are stored in the output, plus the array containing the prefix sum of the bucket sizes. It fits completely in the 16 MiB L3 cache of the used CPU. This is not the case for ten million keys. The SIMD implementation was used for the experiments because the GPU implementation has large overheads and is generally slower than the SIMD implementation if leaf and bucket sizes are small. Therefore, the sorting time has a smaller influence on the construction time of GPURecSplit.

6.3.2 Bijection Rotation - Lookup Table

In Section 6.6.1, we will see that bijection rotation can improve the construction time. We have seen the possibility of using a lookup table for bijection rotation in Section 5.5.2. We have implemented it for the original (sequential) implementation because it is more promising there as for the GPU and SIMD version. As already discussed, using a lookup table in SIMD or GPU code can be much slower than in sequential code. The resulting speedups of using a lookup table are depicted in Figure 6.3. As can be seen, no significant improvement of the construction time is achieved. In fact, the construction takes significantly longer for larger leaf sizes due to the cache usage of the large lookup table. Consequently, we do not use the lookup table in the final implementations.

Out of the four tested bucket sizes, the negative impact on performance for leaf size $\ell = 17$ is the largest for bucket size 50. This is because for bucket size 5, the lookup table is rarely ever used since bijection rotation is generally only used for "full" leaves, i.e., leaves of size



(a) One million keys

(b) Ten million keys

Figure 6.2: Sorting time, speedup of the faster sorting method compared to the original method, and the percentage of the total RecSplit construction time for leaf size $\ell = 5$. The bucket sizes are $\{5 + 50i \mid i \in [40]\}$. There is no plot for the percentage of the original sorting method of the total construction time because this would require changing other parts of the algorithm or computing the prefix sum of the bucket sizes, see Section 5.1. However, the percentage would definitely be larger.

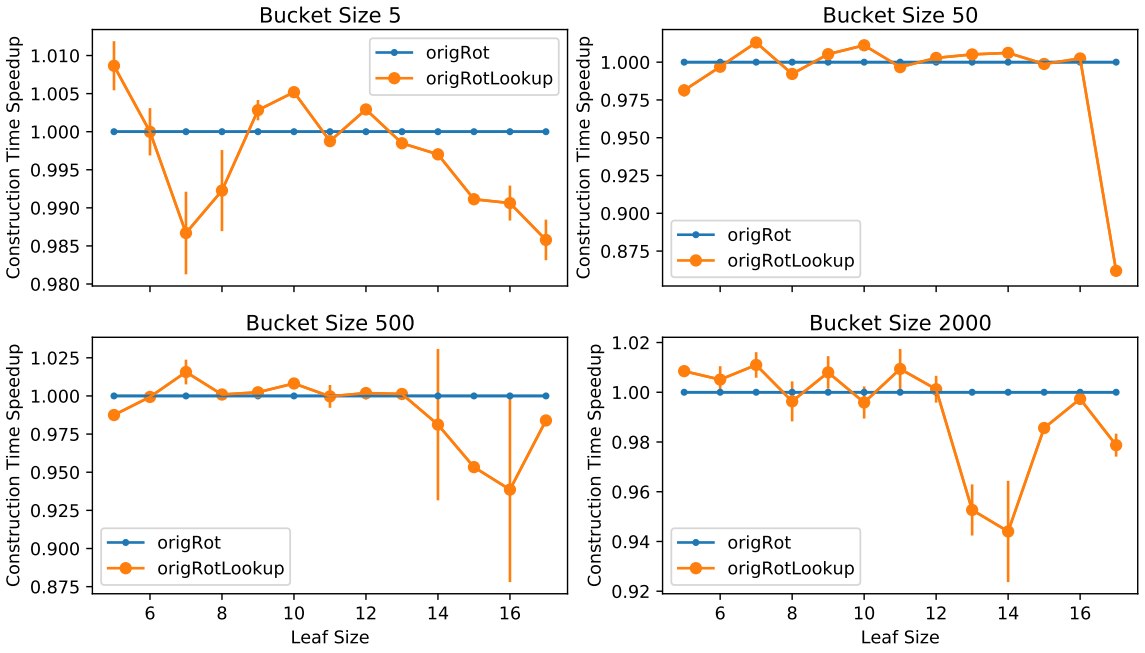


Figure 6.3: Speedup of using a lookup table for bijection rotation ("origRotLookup") compared to using no lookup table. The input consists of one million keys.

ℓ. For buckets of size 500 and 2000, the required amount of work in the aggregation and higher levels is much larger than for bucket size 50. The lookup table has no effect on these levels, thus the negative effect is smaller.

6.3.3 Balanced Splittings

In Section 5.7.2, we have proposed making the splittings in the higher levels more balanced. The mathematical analysis of the exemplary splitting tree has shown that we expect no significant difference of the construction time and the number of bits the MPHF consumes. The experiments validate the analysis. We measured less than 5% difference in the construction time using balanced splittings, sometimes it is slower and sometimes faster. This difference is not significant given the standard deviation. Similarly, the difference of the space consumption is less than 0.001 bits per key (more or less) which is negligible as well.

As expected, the queries are faster if balanced splittings are used. This is shown for the SIMD version in Figure 6.4. Note that the resulting MPHF of the GPU implementation is identical to the result of SIMDRecSplit if the configuration is equal. Hence, evaluating the GPU implementation here is redundant. The effect of balanced splittings is the largest for bucket size 50. The higher levels are almost irrelevant for bucket size 5, which means the effect is minor. For large bucket sizes, the effect shrinks since the overall query times increase while the effect of balanced splittings do not increase significantly. After the first splitting (the root node), the number of keys in the left child is a multiple of the number of keys in a node of the second aggregation level. This means this splitting will always be balanced, and we have changed nothing. Therefore, the only splittings which can profit from our change are the outermost right splittings, which become relatively fewer for larger bucket sizes.

6.3.4 Double Elias-Fano

We explained how to improve the construction time of the Double Elias-Fano representation in Section 5.9. As we can see in Figure 6.5, this is only really relevant for small leaf and

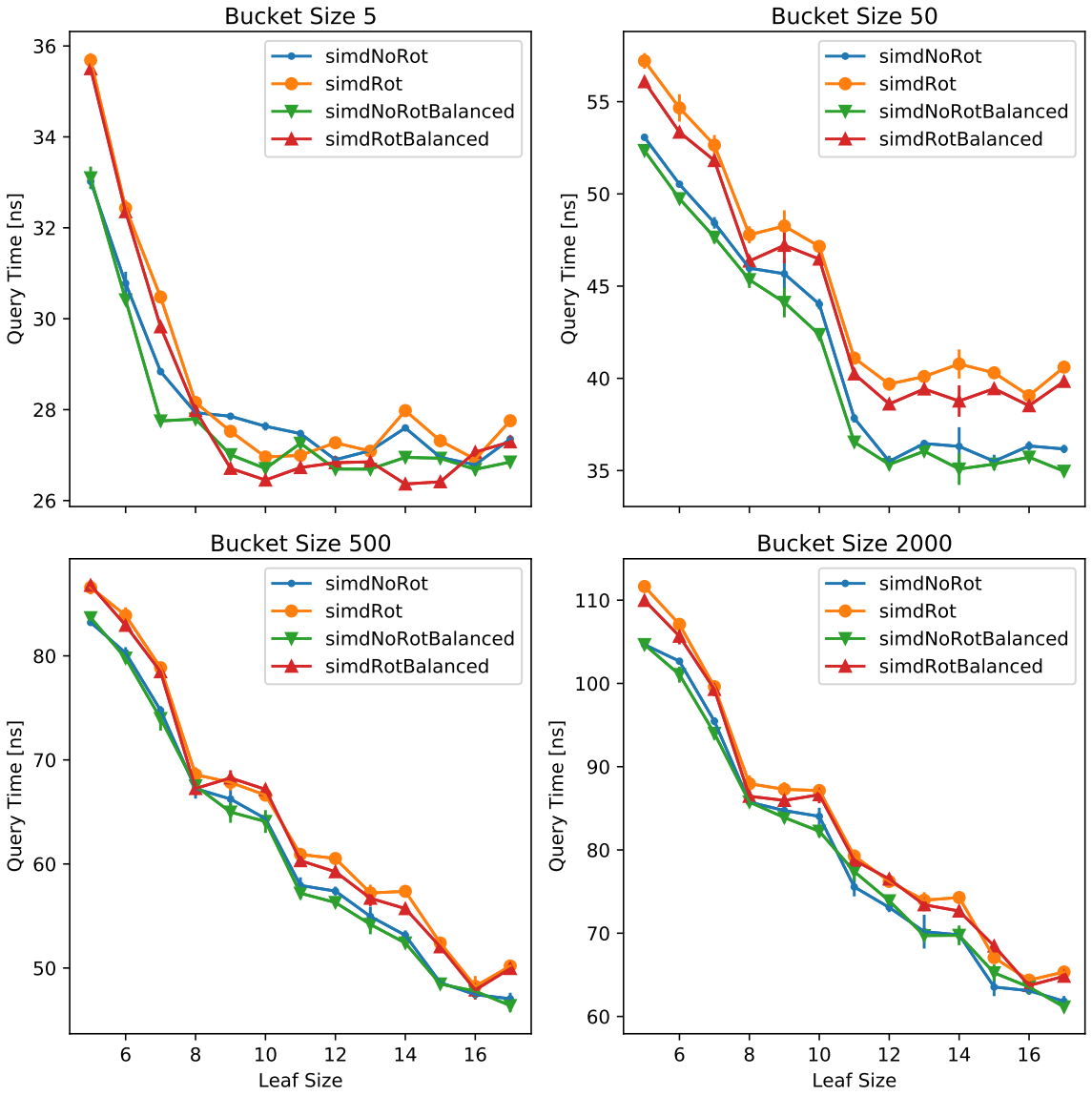


Figure 6.4: Query times with and without balanced splittings for various bucket sizes, key sizes, and with and without bijection rotation. The measurements with balanced splittings were conducted on Ubuntu 22.04 instead of Ubuntu 21.10. The MPHF was constructed with one million keys.

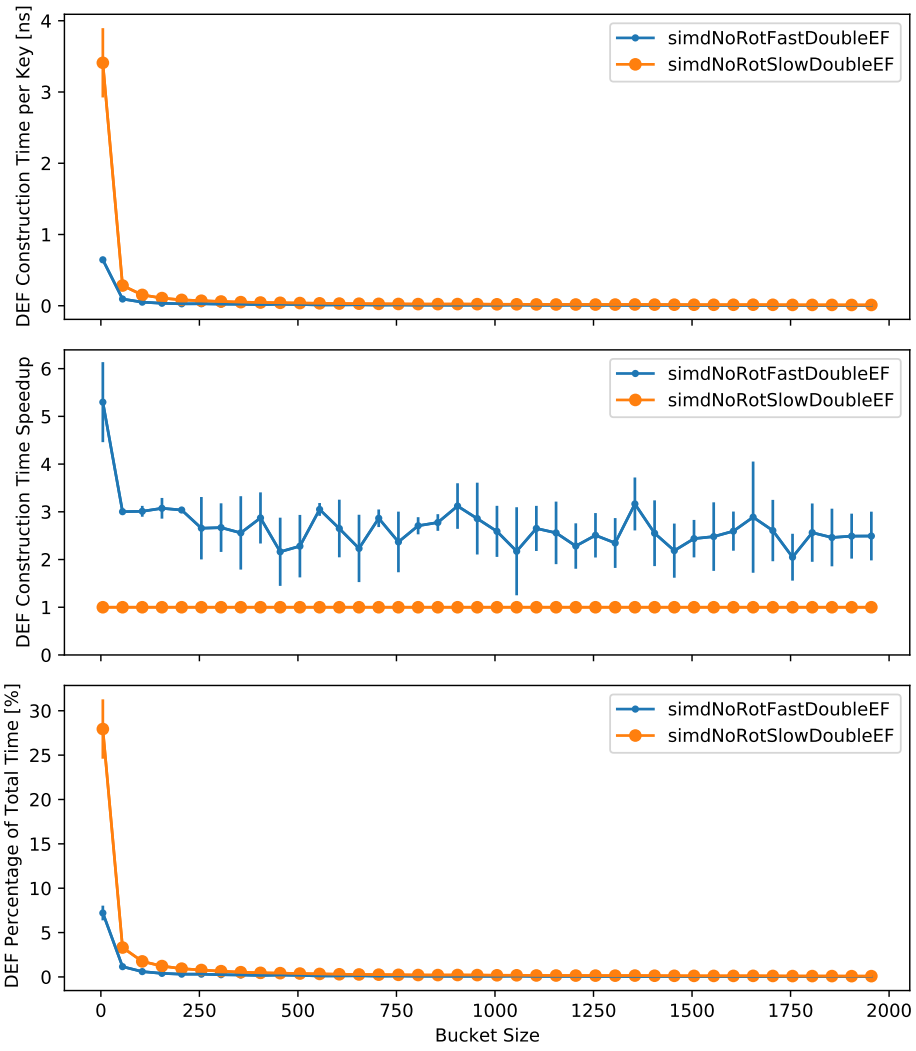


Figure 6.5: Construction time, speedup of the faster method compared to the original method, and the percentage of the total RecSplit construction time of the Double Elias-Fano representation for leaf size $\ell = 5$. The bucket sizes are $\{5 + 50i \mid i \in [40]\}$. The input consists of one million keys.

bucket sizes, e.g., both 5, where the slower Double Elias-Fano construction takes 28% of the total SIMDRecSplit construction time. For larger leaf sizes, the construction of Double Elias-Fano takes the same amount of time, but the computation of the splitting trees takes much longer. For larger bucket sizes, the input arrays of the Double Elias-Fano representation are smaller since both input arrays contain $\kappa + 1$ elements for κ buckets. This means the time to construct it decreases.

Generally, the achieved speedup compared to the original implementation of the Double Elias-Fano representation is between 2 and 6. Like the sorting method (see Section 6.3.1), we only evaluated the SIMD implementation here. For the configurations where the faster construction has a significant impact on the overall running time, i.e., small leaf and bucket size, the overhead of GPURecSplit is too large for any useful measurements.

6.4 Evaluating Different Techniques - SIMD Implementation

We evaluate different techniques concerning the SIMD implementation in this section. As said before, SIMDRecSplit uses 16 threads if not said otherwise.

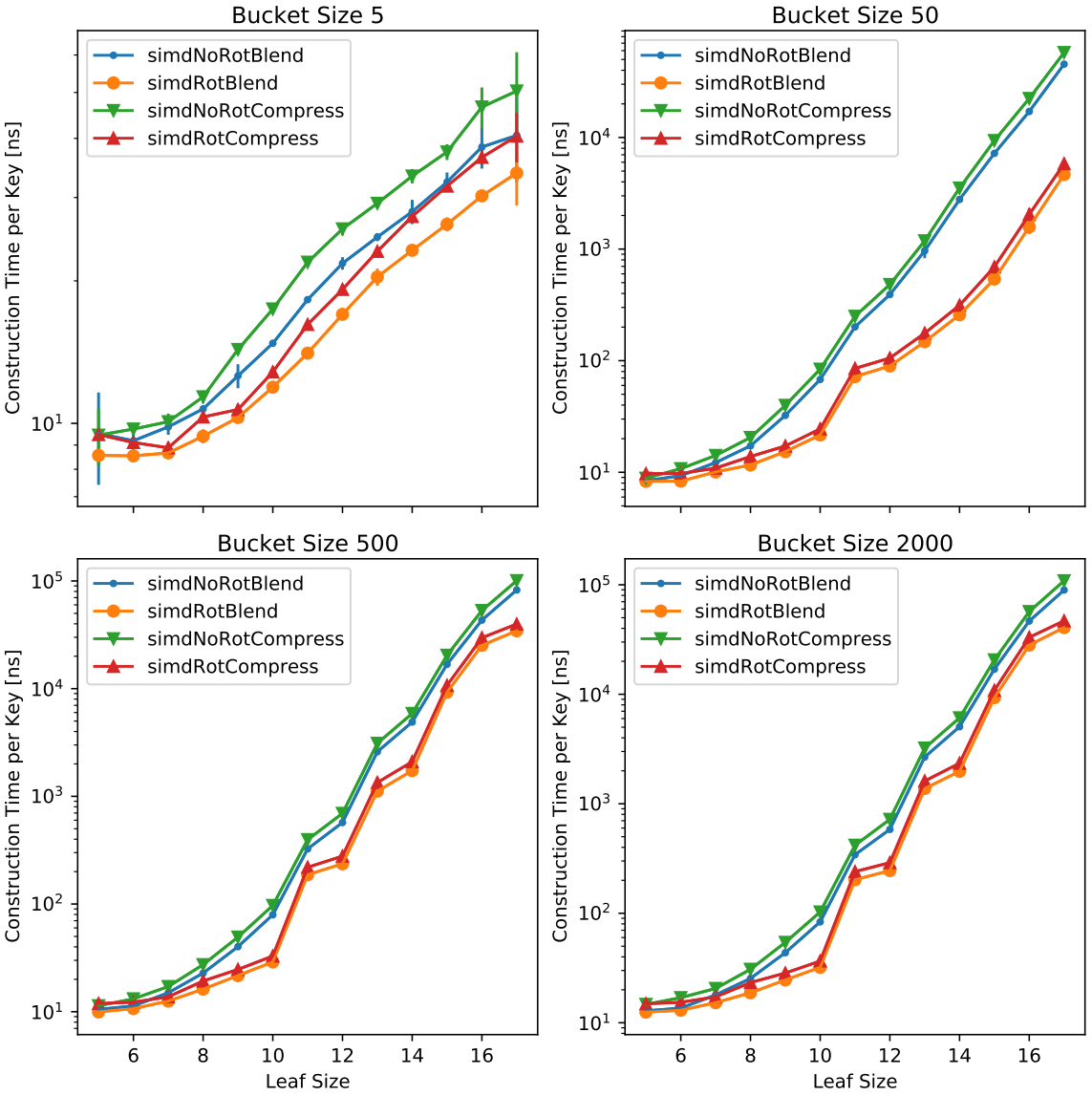


Figure 6.6: Construction time of SIMDRecSplit, comparing both implementations of the XOR step before applying the remix function. The input consists of one million keys.

6.4.1 32-Bit Hash Function

Section 5.3.1 presents two different possible implementations of how to compute the XOR of the 32 most and 32 least significant bits as the input of the remix function. One implementation uses the `compress` function whereas the other uses the `blend16` function. The plots in Figure 6.6 show clearly that the implementation that uses `blend16` is faster. We use it for the final implementation of SIMDRecSplit.

The use of the original 64-bit remix function was not evaluated for the SIMD implementation as this would require changing many parts of the code. However, we evaluate it for the GPU implementation in Section 6.5.1.

6.4.2 Bijection Midstop Parameter

The bijection midstop *factor* α is used to determine the bijection midstop *parameter* $p = p(m) = \lceil \alpha\sqrt{m} \rceil$ which denotes the number of keys that are processed before checking for a collision; see Section 5.4.1. For the original implementation, $\alpha = 2$ and for GPURecSplit

$\alpha = 3$. The speedup achieved by using different values of α compared to using no bijection midstop is shown in Figure 6.7. Note that bijection midstop is used even for small leaves in these plots. In the final implementation, bijection midstop is only used for leaves of size at least 12. This seems reasonable since most bijection midstop factors achieve a speedup greater than 1 for $\ell \geq 12$. Note that this means bijection midstop performs not as bad as it seems in the plot for bucket size 5.

All other measurement in this chapter use $\alpha = 2.6$. Looking at Figure 6.7, this does not seem optimal. This value was chosen because it performed best in preliminary experiments. Many other experiments in this chapter were performed before the measurements in Figure 6.7, so we decided to retain it for the remaining experiments as well.

6.4.3 Bijection Midstop Popcount

As discussed in Section 5.4.3, popcount can only be used on vectors if the instruction set extension AVX512VPOPCNTDQ is available. Otherwise, the preliminary words are dumped in an array before calculating popcount one word after the other to decide which hash functions already exhibit a collision. The speedup of using the vector popcount instruction relative to the sequential fallback is depicted in Figure 6.8. As expected, a significant difference is only measured for $\ell \geq 12$ since bijection midstop is not even used otherwise. Then, the speedup is about 10%.

As we have seen in other experiments, the greatest speedup is achieved for bucket size 50. For bucket size 5, most leaves are smaller than 12 and therefore bijection midstop is not used at all. For large bucket sizes, the aggregation and higher levels become more relevant and are thus decreasing the effect of improving the search for bijections.

6.4.4 Bijection Rotation Midstop

After processing both sets when using bijection rotation, the SIMD implementation checks whether any collisions has been found. This is not to be confused with bijection midstop (bijection midstop and bijection rotation are mutually exclusive), but the idea is the same; see Section 5.5.3. This optimization can only be used if the instruction set extension AVX512VPOPCNTDQ is available and is only used for $\ell \geq 11$. The results can be seen in Figure 6.9. The speedup is up to 30%. Like in other plots before, the largest speedup is measured for bucket size 50. For buckets of size 5, bijection rotation is rarely used because bijection rotation is only used for full leaves. For large buckets, the aggregation and higher levels make up a larger part of the construction time, thus decreasing any effect only affecting leaves.

6.4.5 16-Bit Integer Division

As explained in Section 5.6.1, 16-bit integer division can be used in the aggregation levels which could theoretically be faster. However, Figure 6.10 shows no clear benefit of using 16-bit division. Perhaps, the time the division takes is insignificant compared to the total construction time. Since we do not profit from 16-bit division, we take the simpler implementation and use 32-bit integer division.

6.4.6 Multithreading

The construction of the splitting trees is parallelized by distributing the buckets to different threads, see Section 5.8. This is fairly straightforward with almost no communication between different threads required. The resulting speedups without bijection rotation are shown in Figure 6.11. With bijection rotation, the corresponding plots look very similar and are therefore omitted. The achieved speedups are up to 7. For bucket size 5, the

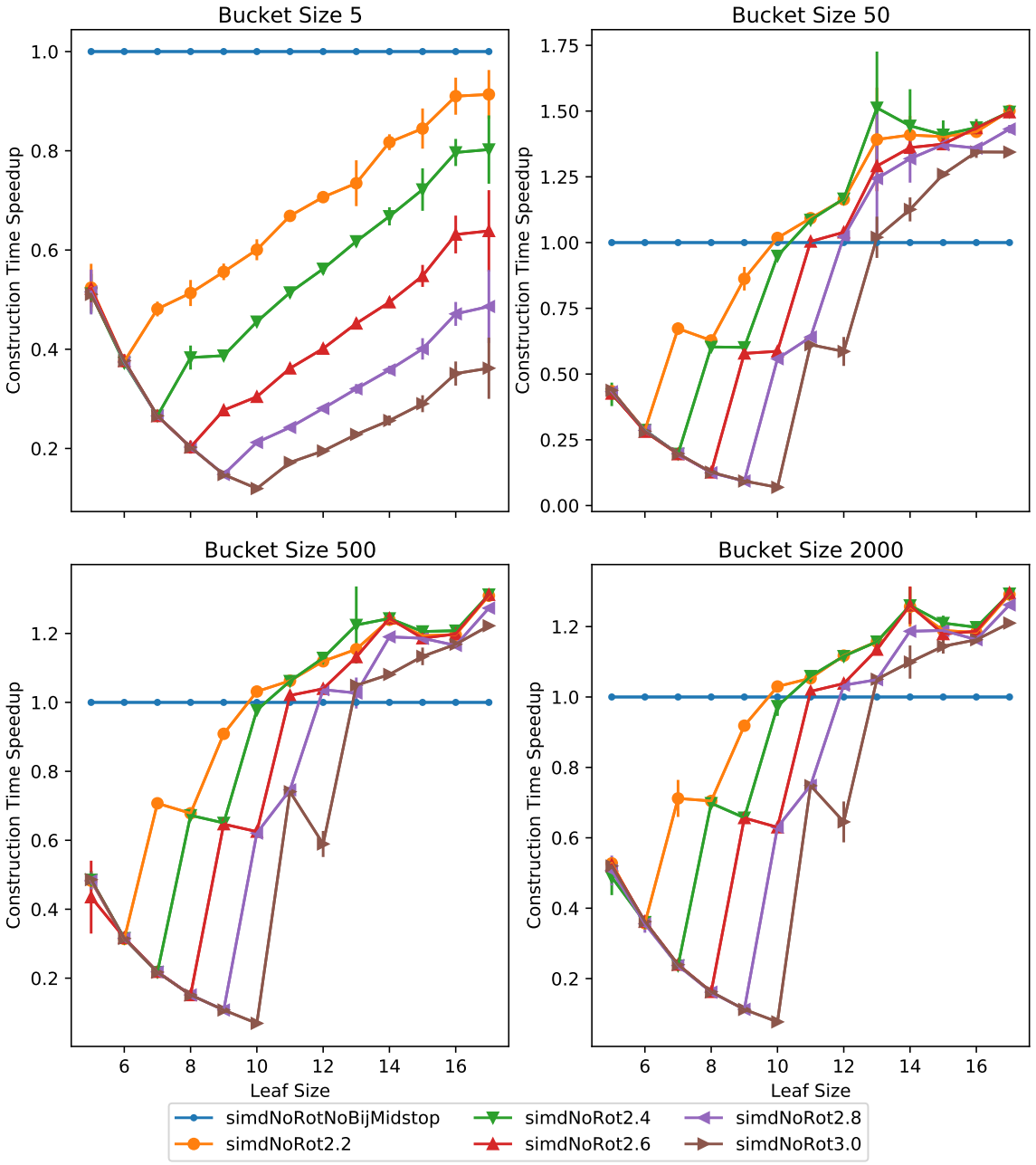


Figure 6.7: Construction time speedup of SIMDRecSplit using different bijection midstop factors relative to the construction time using no bijection midstop at all. The implementations using bijection midstop use it for all leaves, even very small leaves. The input consists of one million keys.

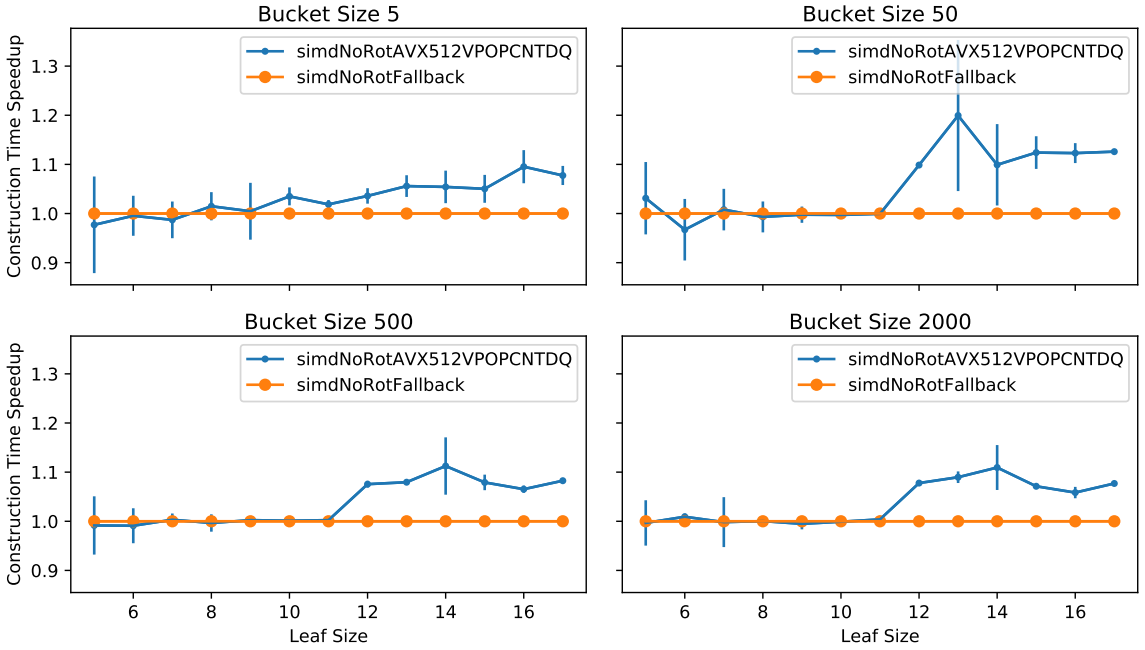


Figure 6.8: Construction time speedup of SIMDRecSplit using the vector popcount instruction for bijection midstop relative to the construction time using the fallback. The input consists of one million keys.

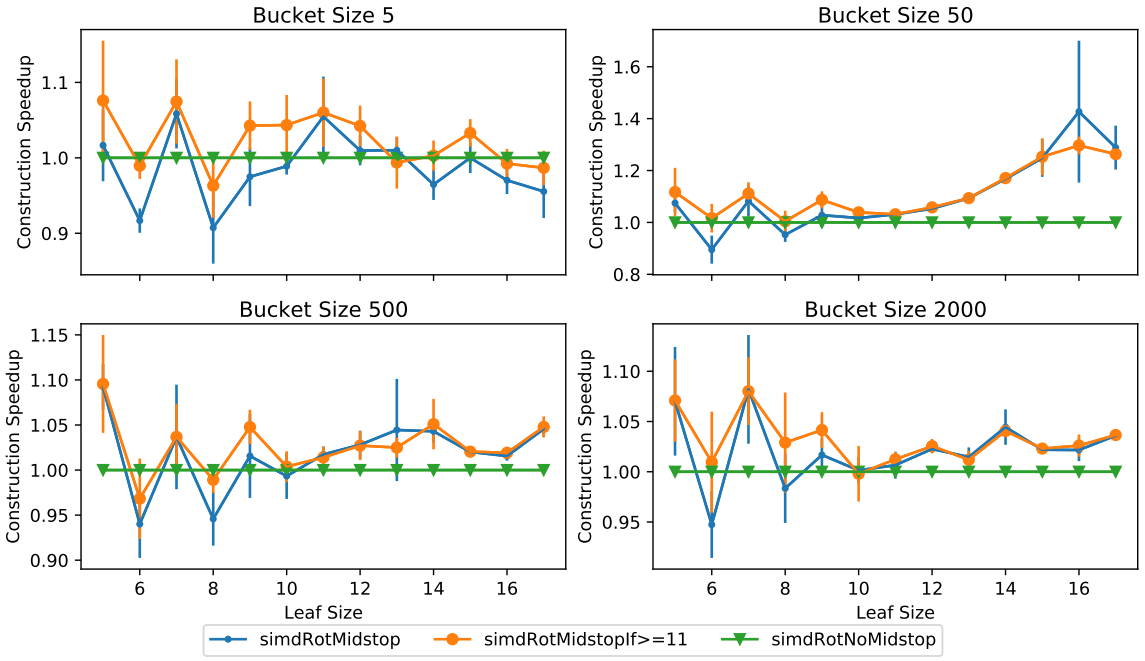


Figure 6.9: Construction time speedup of SIMDRecSplit using bijection rotation midstop (not to be confused with bijection midstop) for all leaf sizes and for $\ell \geq 11$ relative to the construction time without bijection rotation midstop. The input consists of one million keys.

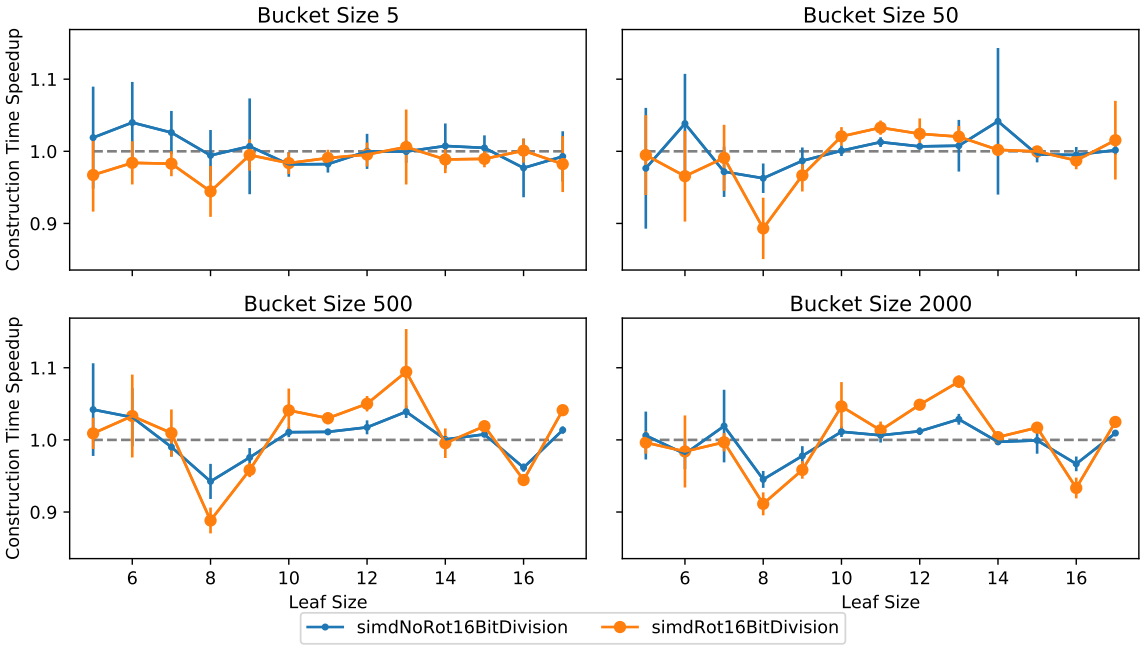


Figure 6.10: Construction time speedup of SIMDRecSplit using 16-bit integer division instead of 32-bit integer division. The speedup of "simdNoRot16BitDivision" is relative to the version with 32-bit division and deactivated bijection rotation, and the speedup of "simdRot16BitDivision" is relative to the version with 32-bit division and with bijection rotation activated. The input consists of one million keys.

speedups are smaller because sorting and the Double Elias-Fano construction take more time, and they do not use multithreading. Even for a very large leaf size ℓ, most buckets will be processed quickly since they are smaller than ℓ if the bucket size is 5. This means the time it takes to construct the splitting trees does not increase exponentially with ℓ as for larger bucket sizes.

Interestingly, the speedup declines for $\ell > 12$ and relatively large bucket sizes. Unfortunately, we do not have a good explanation for this behavior. A conjunction is that the proportion of vector instructions increases for larger leaf sizes. A single vector instruction may need more energy than a single scalar instruction since more transistors are active. This means a single core may draw more power if many vector instructions are used. It may be necessary to decrease the clock frequency to avoid overheating of the CPU. In this case, the use of more threads has also decreased utility because they contribute to the overall power consumption.

A similar conjunction is that the total construction time increases for larger leaf sizes which means the CPU may not be able to hold the clock frequency during the whole construction. It may throttle down after a while. Between the different measurements, the MPHF is always queried 100 million times. This is done sequentially, i.e., the CPU may be able to cool down before the next construction.

The plots also show that the SIMD implementation can profit from hyper-threading. Using 16 threads gives a slightly larger speedup than using only 8 threads although the used CPU only has 8 physical cores.

6.5 Evaluating Different Techniques - GPU Implementation

This section is dedicated to evaluating different techniques concerning the GPU implementation.

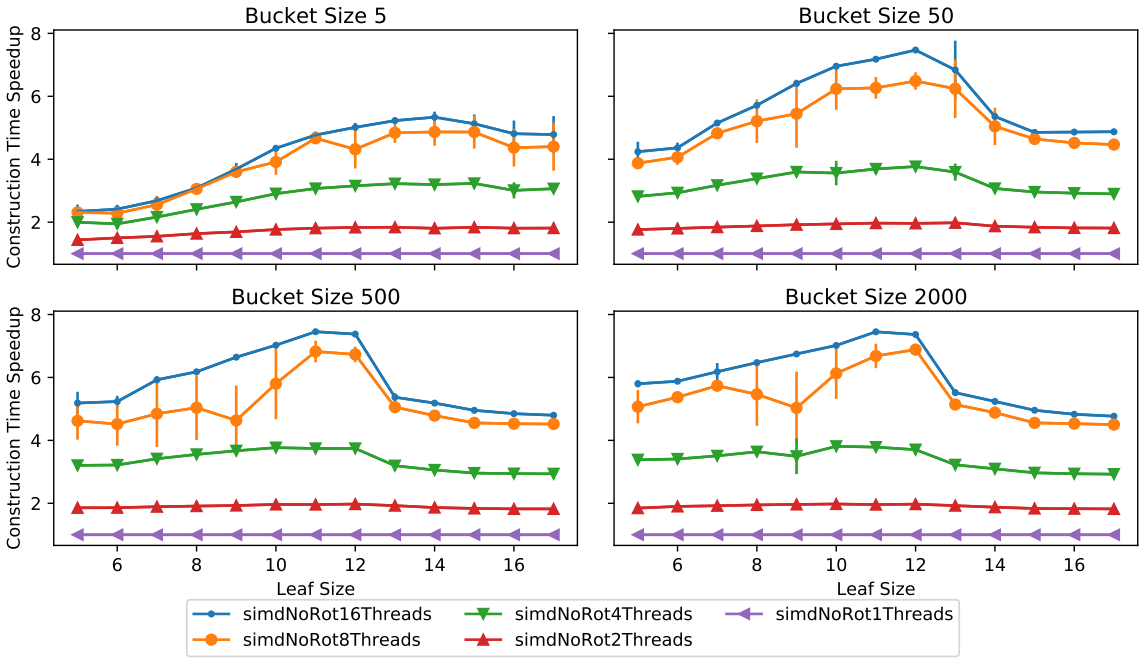


Figure 6.11: Construction time speedup of SIMDRecSplit using various number of threads relative to using only a single thread. The input consists of one million keys.

6.5.1 32-Bit Hash Function

We have engineered a 32-bit hash function in Section 5.3 to combat slow 64-bit integer multiplications. As Figure 6.12 shows, this can accomplish a speedup of up to 1.7 for large leaf sizes. Contrary to most other techniques evaluated so far, this also has an effect on the resulting MPHF and the query algorithm. The query algorithm must also use the 32-bit hash function instead of the original 64-bit hash function. Fortunately, there is no negative impact on the number of bits the MPHF takes. The difference is less than 0.002 bits per key in both directions. This is indistinguishable from noise.

This is not the case for the query times. As can be seen in Figure 6.13, the 32-bit hash function has a negative impact on the query time. Queries take about 3% longer with the 32-bit hash function. The reason is that there is an extra step required. Before applying the 32-bit *remix* function, the 32 most and 32 least significant bits must be combined with XOR. The 32-bit remix function itself should not have a negative impact on the performance since it does the same operations, just with 32 bits instead of 64 bits. If it has any impact, it should be positive because the 32-bit constant used for multiplication must not first be moved into a register unlike the 64-bit constant in the 64-bit remix function.

Using the 32-bit hash function decreases the construction time without an effect on the space requirements, but it increases the query time. This means using it is useful if the query time is not a major concern, e.g., if the space consumption is the most important metric. If the query time is important, the picture is not as clear. To get a better overview of the tradeoff between construction and query time using the 32- vs the 64-bit hash function, Figure 6.14 pictures the *Pareto fronts* (see Section 2.5) of GPURecSplit with both hash functions. This means from all the tested combinations of leaf and bucket size, only those points are shown for which no other point has a less than or equal construction and query time and is strictly better in at least one of those metrics.

The outcome of Figure 6.14 is that neither the 32- nor the 64-bit hash function is clearly better in this tradeoff. If for example the Pareto front of “gpuNoRot32Bit” was below-left of the Pareto front of “gpuNoRot64Bit”, it would mean that the 32-bit hash function is

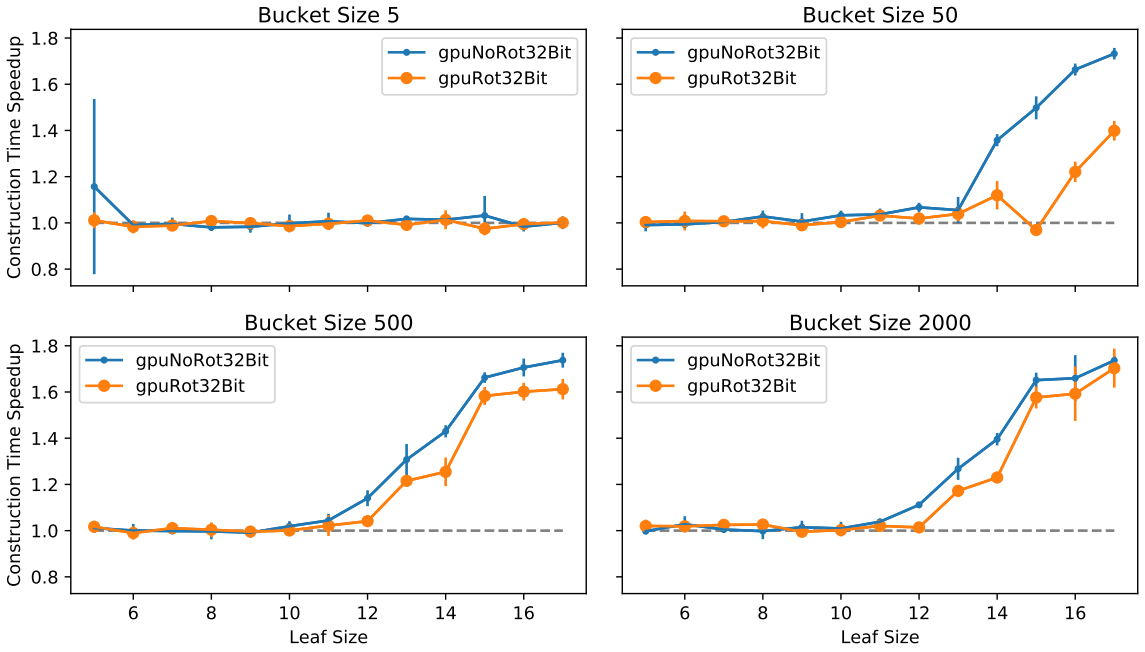


Figure 6.12: Construction time speedup of GPURecSplit using the 32-bit hash function relative to using the original 64-bit hash function. The speedup of "gpuNoRot32Bit" is relative to the version without bijection rotation, and the speedup of "gpuRot32Bit" is relative to the version with bijection rotation. The input consists of one million keys.

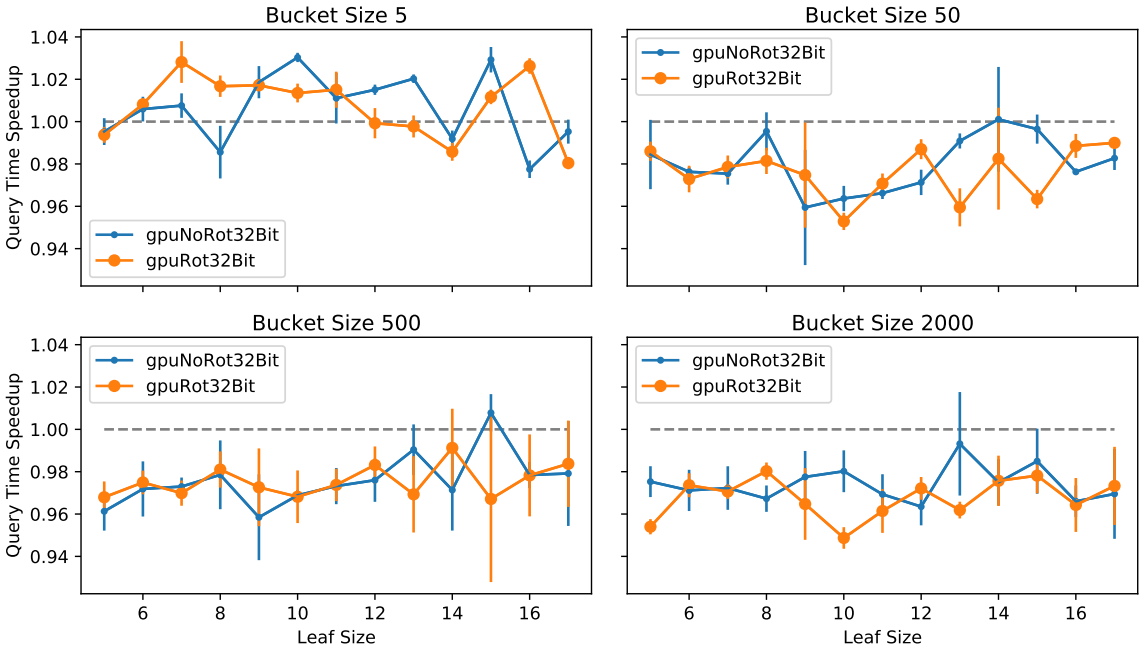


Figure 6.13: Query time speedup of GPURecSplit using the 32-bit hash function relative to using the original 64-bit hash function. The speedup of "gpuNoRot32Bit" is relative to the version without bijection rotation, and the speedup of "gpuRot32Bit" is relative to the version with bijection rotation. The input consists of one million keys.

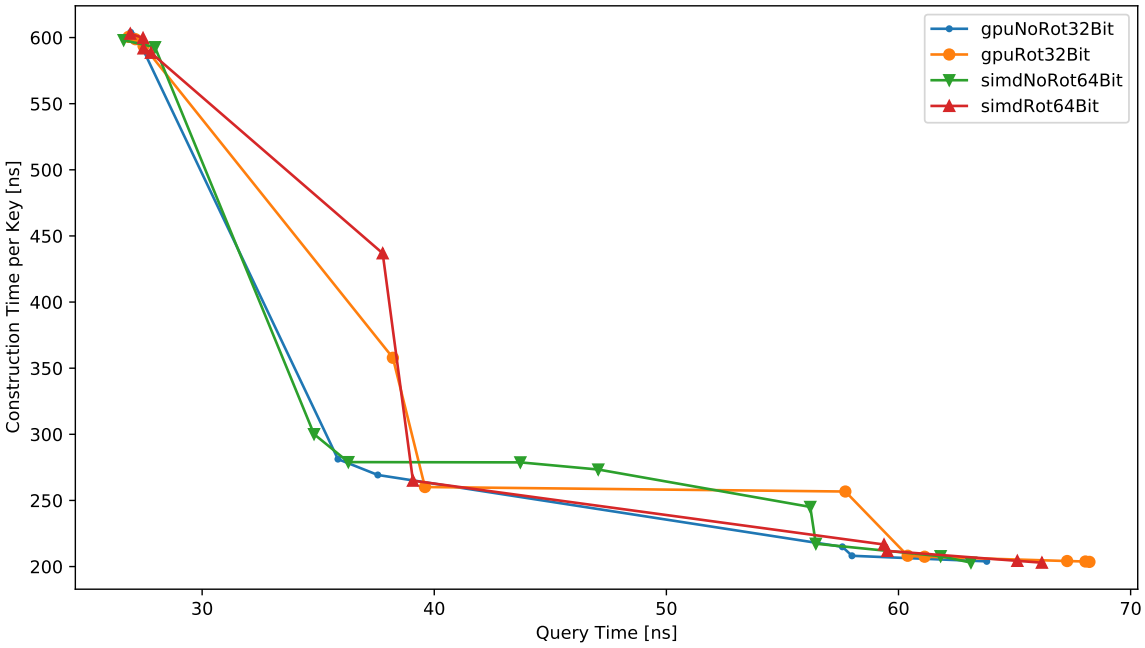


Figure 6.14: Pareto fronts (construction time versus query time) of using the 32-bit hash function or the original 64-bit hash function. The input consists of one million keys.

better in the case when bijection rotation is not used. This is because for every combination of leaf size and bucket size that "gpuNoRot64Bit", one could find a combination such that "gpuNoRot32Bit" has faster construction and query time. However, since the lines cross several times and the same is true for "gpuRot32Bit" and "gpuRot64Bit", there is no clear winner.

6.5.2 Bijection Midstop Parameter

We measure the influence of the bijection midstop factor α on the construction time as in Section 6.4.2. Remember that the resulting bijection midstop parameter is $p = p(m) = \lceil \alpha\sqrt{m} \rceil$. The speedup achieved by using different values of α compared to using no bijection midstop is shown in Figure 6.15. Note that bijection midstop is used even for small leaves in these plots. In the final implementation, bijection midstop is only used for leaves of size at least 14. Like in many other experiments before, the effect is the greatest for bucket size 50. Bucket size 5 is dominated by the overhead of using a GPU, and for larger bucket sizes the aggregation and higher levels take a larger portion of the total construction time.

All other measurements in this chapter use $\alpha = 3$. Looking at Figure 6.15, $\alpha = 2.8$ could be a bit better. However, the value 3 was chosen because it performed best in preliminary experiments. Many other experiments in this chapter were performed before the measurements in Figure 6.15, so we decided to retain it for the remaining experiments as well.

6.5.3 Using Shared Memory in Aggregation Levels

We have proposed an alternative approach to store the counts in shared memory instead of in a packed form in registers, see Section 5.6.3. In fact, this was how the aggregation level was implemented in the beginning. Only after the SIMD implementation enforced a different approach due to expensive gather instructions, it became clear that the same approach should improve the performance of GPURecSplit. This is validated by Figure 6.16. As expected, there is no significant difference for bucket sizes 5 and 50 since the aggregation

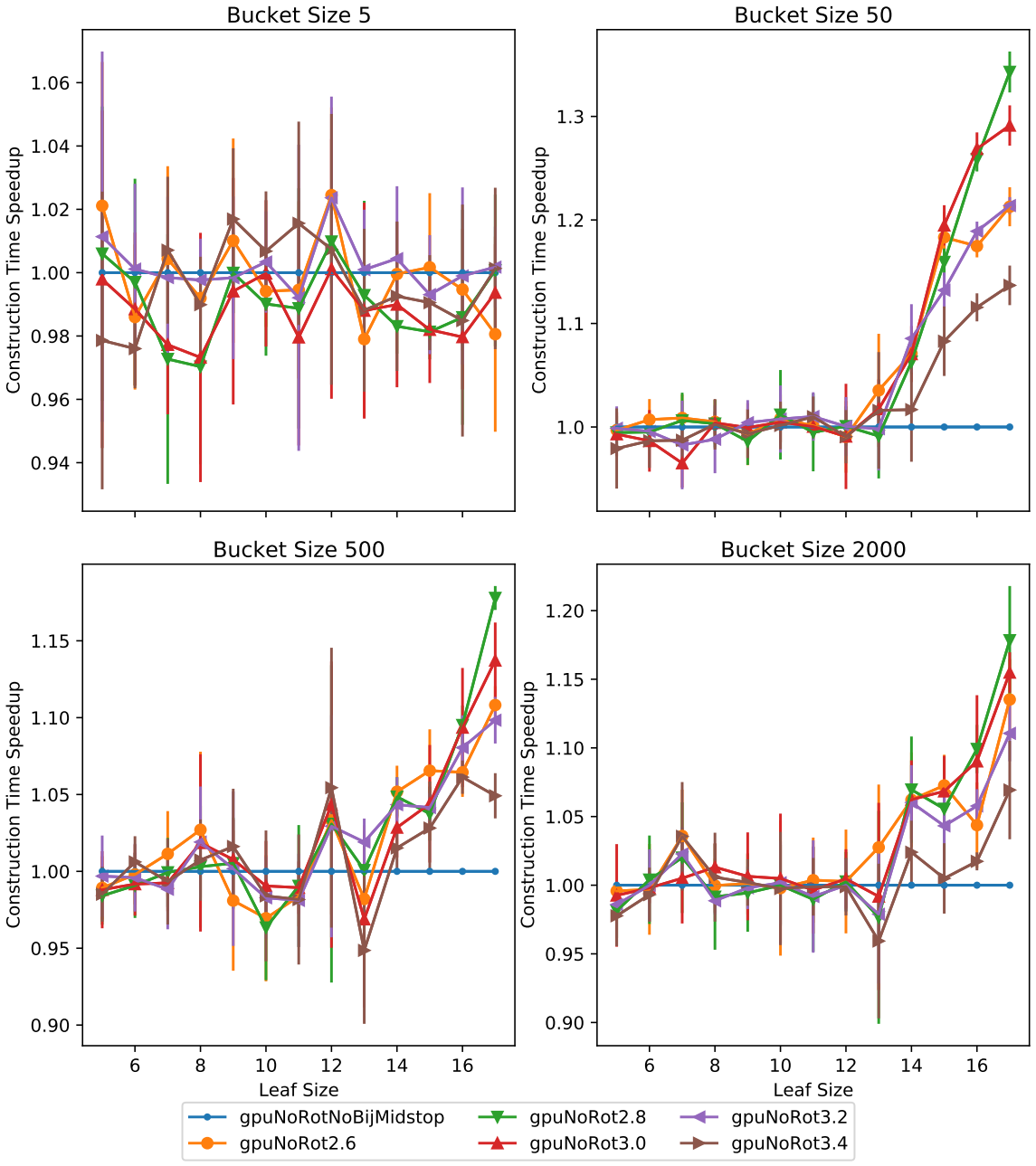


Figure 6.15: Construction time speedup of GPURecSplit using different bijection midstop factors relative to the construction time using no bijection midstop at all. The implementations using bijection midstop use it for all leaves, even very small leaves. The input consists of one million keys.

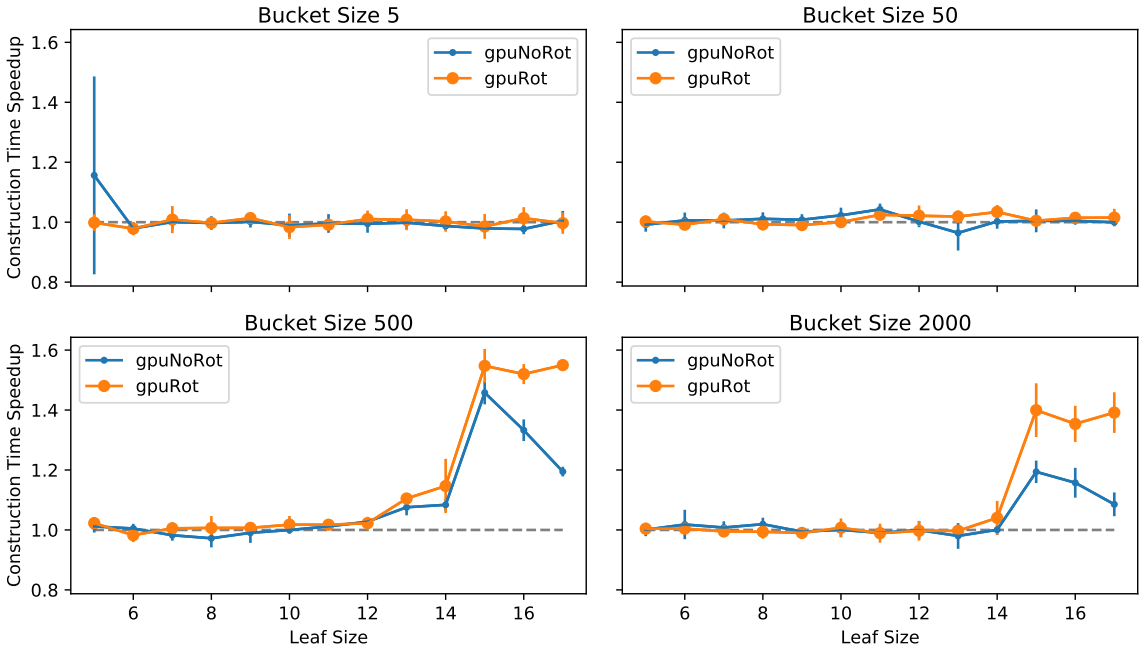


Figure 6.16: Construction time speedup of GPURecSplit by storing the counts of the aggregation level in packed form in registers relative to the alternative approach of storing the counts in shared memory. The speedup of "gpuNoRot" is relative to the version without bijection rotation and the speedup of "gpuRot" is relative to the version with bijection rotation. The input consists of one million keys.

levels do not play an important role for such small buckets. For larger bucket sizes and relatively large leaf sizes, the speedup is up to 1.5. The construction time is dominated by the overhead of using a GPU for small leaf sizes, therefore no significant improvement can be measured there.

6.5.4 Key Redistribution

Section 5.6.3 describes how to use warp-aggregated atomics to efficiently distribute the keys into the right child nodes after a valid splitting has been found. It was already stated that manually using warp-aggregated atomics should not improve the construction time since the compiler is able to optimize this automatically. This is confirmed by Figure 6.17. The difference is indistinguishable from noise as the standard deviations indicate.

6.5.5 Work Queues

We have seen in Section 5.10.2 that the number of work queues is per default only 8. This decreases the GPU utilization because the GPU implementation relies on the concurrent execution of many kernels to utilize the complete processing power of the GPU. The effect of increasing the number of work queues is shown in Figure 6.18. The speedup is larger than 3 for some configurations. As many times before, the speedup is the highest for bucket size 50. For bucket size 5 or if the leaf size is small, the construction takes actually longer. The reason is that the overhead of a kernel launch increases if more work queues are used and the number of kernel launches is high if bucket size or leaf size is small. The larger overhead is why the number of work queues is per default only 8.

For larger bucket sizes, the speedup is smaller because the implementation relies less on concurrent kernels. In a large bucket, the aggregation levels and the leaf level contain many thread blocks and are therefore able to utilize the GPU better than smaller buckets. Figure 6.18 only shows the speedups without bijection rotation because the plots with

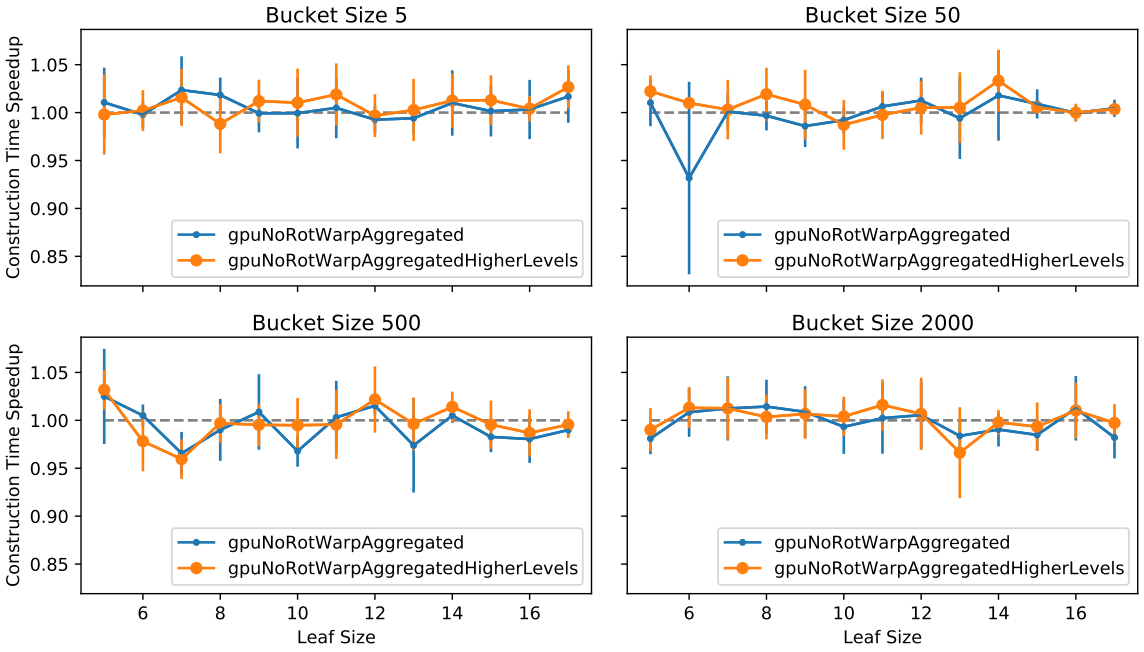


Figure 6.17: Construction time speedup of GPURecSplit by using warp-aggregated atomics manually. The values of "gpuNoRotWarpAggregated" are measured with warp-aggregated atomics in the higher levels and the aggregation levels, whereas "gpuNoRotWarpAggregatedHigherLevels" uses warp-aggregated atomics manually only in the higher levels. The input consists of one million keys.

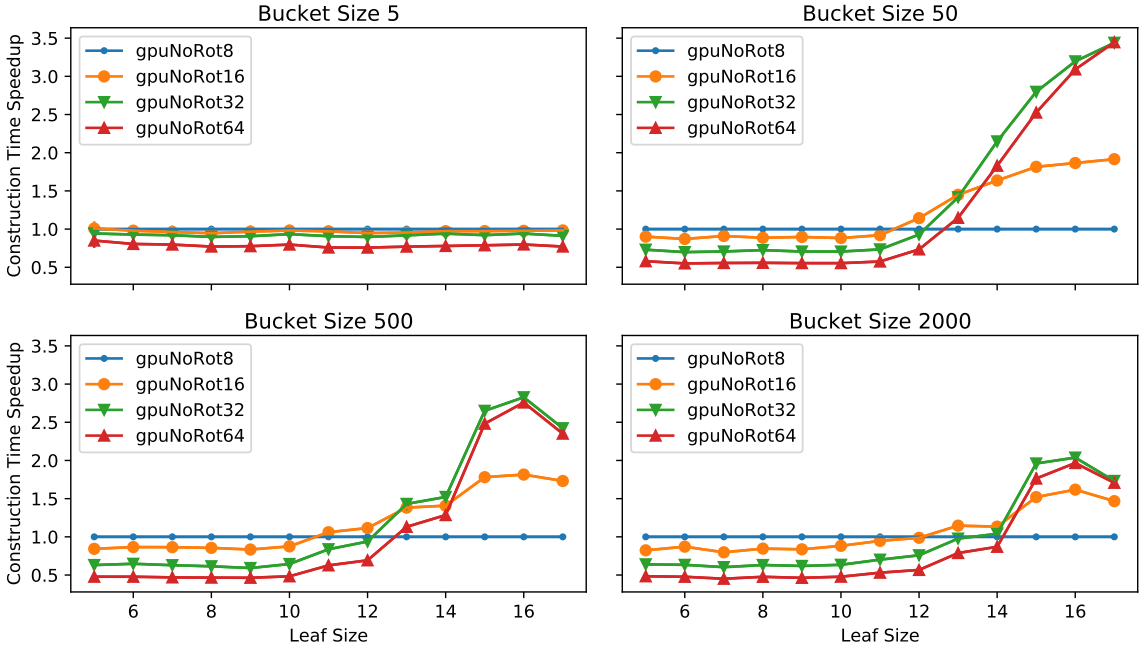


Figure 6.18: Construction time speedup of GPURecSplit by using more than the default 8 work queues. The numbers in the legend denote the number of work queues. The input consists of one million keys.

bijection rotation look almost identical. The only difference is that the speedups are a bit smaller for bucket size 50, only up to 2.5 instead of 3.5. This is because finding bijections is the most expensive part for bucket size 50, but they are a lot faster using bijection rotation. Therefore, the increased overhead of using more work queues has a larger impact than without bijection rotation.

6.5.6 Multiple GPUs

We have also measured how well GPURecSplit scales with more than one GPU. Only a few slight changes to the code are necessary to enable the use of more than one GPU. Each CPU thread creates its streams on one GPU and launches all its kernels on this GPU. Theoretically, it should be able to scale perfectly with more GPUs since no synchronization between different threads is required. Unfortunately, this is not the case in practice as Figure 6.19 shows. There is only a speedup greater than 1 for very large leaf sizes and an at least moderately large bucket size or for a very large input.

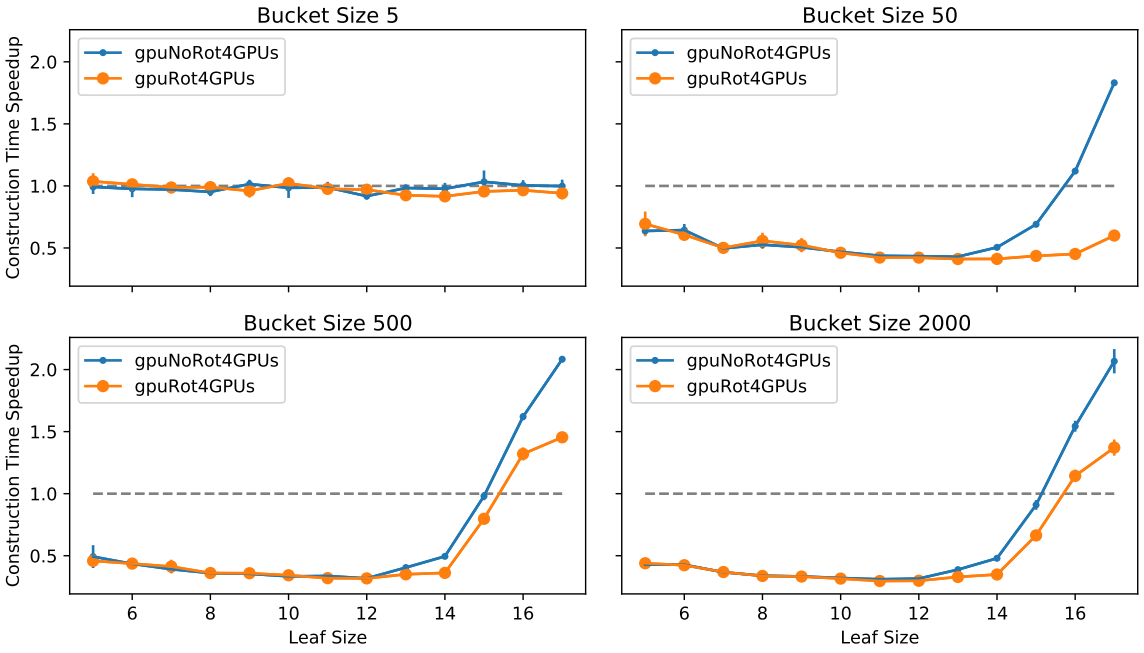
The machine used for this experiment is part of the bwUniCluster. Specifications of the machine can be found at [1] under the name "GPU x4". It features two Intel Xeon Gold 6230 CPUs [7] with 20 cores each, 384 GB of main memory and four Nvidia Tesla V100 GPUs [16]. This particular GPU is one generation older than the RTX 3090, but the number of 32-bit integer ALUs is almost equal with 5120 ALUs instead of 5248. The newest CUDA version available on this machine is 11.4, so this is the version we use. CUDA 11.4 only officially supports GCC up to version 10. GCC 10.3 produced an internal compiler error upon compiling the project, thus we use GCC 10.2. The operating system is Red Hat Enterprise Linux 8.4.

To ensure full utilization of all GPUs, we doubled the number of CPU threads to 16 and the total number of streams to 256. This means there are four CPU threads per GPU and 64 streams. These numbers were not specifically optimized, i.e., there is perhaps room for improvement. The reason for the less-than-expected improvement is probably the large overhead associated with using several GPUs. Memory must be allocated and later freed on all devices, more streams and threads must be created and destroyed, there is more synchronization overhead after the threads have finished, etc. The bus between the CPU and GPUs may also increasingly become a bottleneck the more GPUs are used. This particular problem could be alleviated by using batched memory transfer as explained in Section 5.10.1.

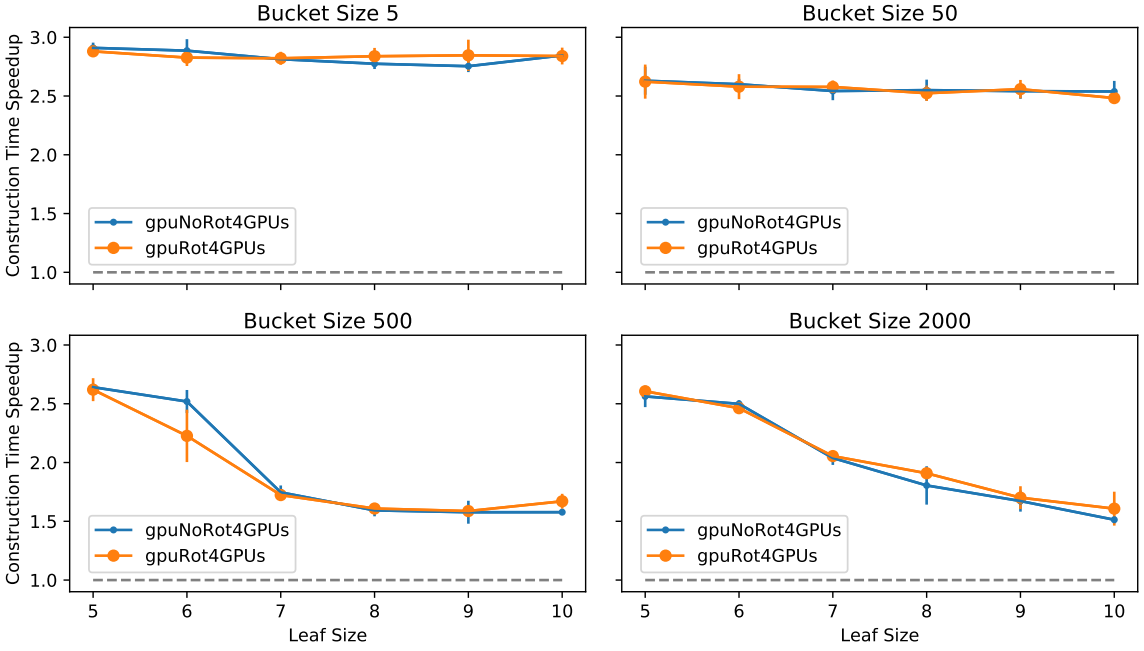
Many overheads are constant. Therefore, they are less relevant for a billion input keys than for a million. This explains why the speedups are generally bigger for a larger input. Figure 6.19 also shows that larger bucket sizes lead to smaller speedups. This is the case because the utilization is generally higher for larger bucket sizes. The speedup increases for very large leaf sizes since the overheads become increasingly irrelevant due to very long-running kernels. The leaf kernels are shorter with bijection rotation, hence the speedup is smaller.

6.6 Comparison with the Original RecSplit Implementation

In this section, we finally compare our new implementations with the original RecSplit implementation. We begin with the construction time, followed by the space consumption and the query times. In the end, we provide some plots showing the Pareto fronts of all implementations to get an overview which implementations are better suited in which situations. The measurements in this section were performed with one million keys and with one billion keys. For the latter, the maximum leaf size is only 10 instead of 17 since the construction would take too long for larger leaf sizes. The SIMD and GPU implementations



(a) One million keys



(b) One billion keys

Figure 6.19: Construction time speedup of GPURecSplit by using four Nvidia Tesla V100 GPUs relative to using only one. The speedup of "gpuNoRot4GPUs" is relative to the version without bijection rotation, "gpuRot4GPUs" is relative to the version with bijection rotation.

Table 6.1: Maximum speedups of our new implementations compared to the original implementation without bijection rotation for one million keys. The bucket and leaf size that can achieve this speedup are depicted as well.

Implementation	Maximum speedup	Bucket size b	Leaf size ℓ
origRot	13	50	17
simdNoRot1Thread	7	5	9
simdRot1Thread	50	50	17
simdNoRot16Threads	46	50	12
simdRot16Threads	333	50	15
gpuNoRot	581	2000	17
gpuRot	1873	50	17

use balanced splittings (see Section 5.7.2) in this section. To make the difference between the original implementation ("origNoRot") and our implementations more clearly, the original version is drawn with dashed lines.

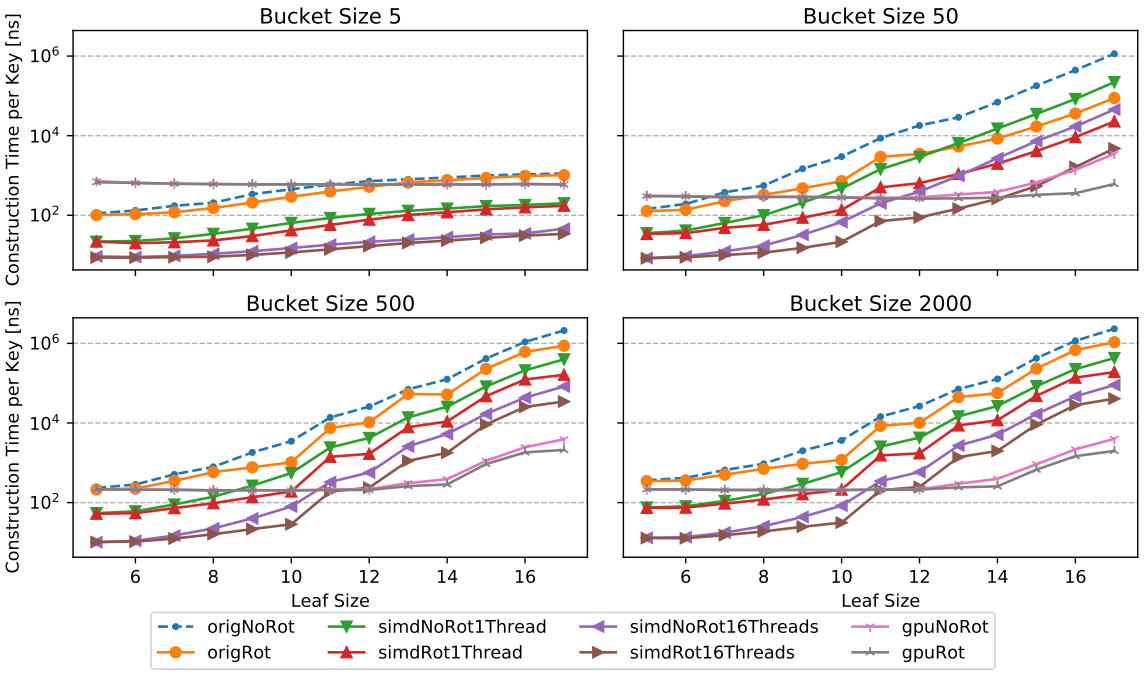
6.6.1 Construction Time

Figure 6.20 depicts the construction time of our implementations. For bucket size 5, SIMDRecSplit is generally faster than the other implementations. Using a GPU has too much overhead for such small buckets. Therefore, the GPU implementation is only faster than the original implementation for very large leaf sizes where the overheads play an decreased role. Even then, the difference is not large since GPU utilization is low for small buckets. This changes drastically for bucket size 50. With leaf size 17 and one million input keys, the speedup of GPURecSplit with bijection rotation relative to the original implementation without bijection rotation is more than 1800 as can be seen in Figure 6.21a and Table 6.1.

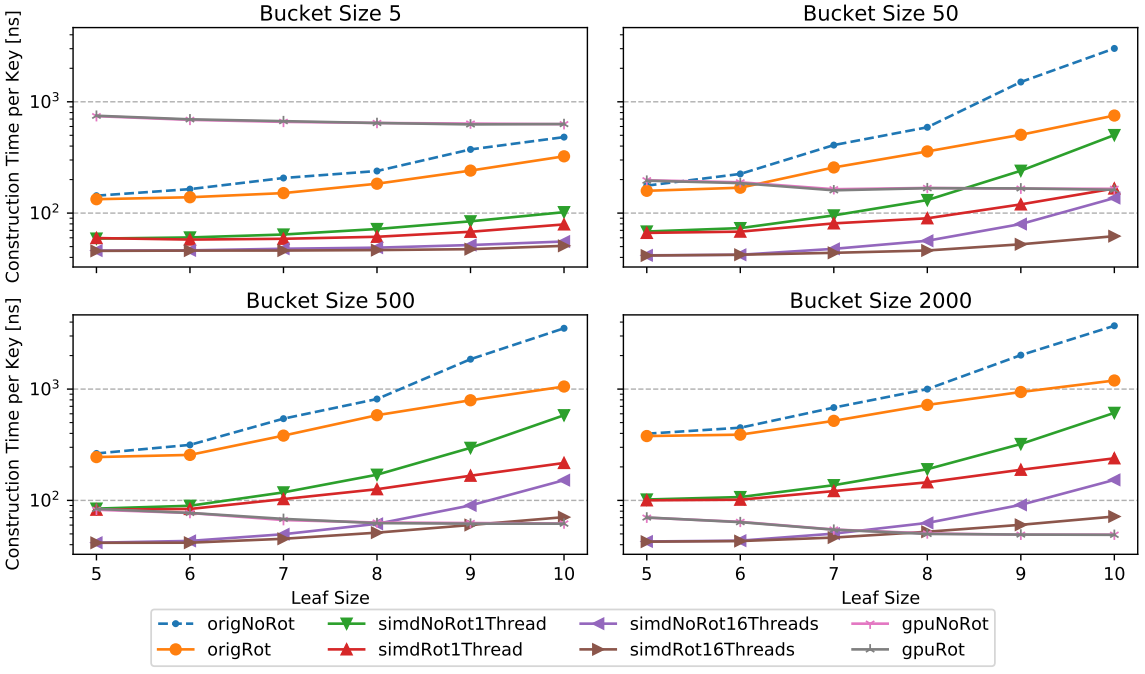
The speedup is higher for bucket size 50 than for larger bucket sizes since bijection rotation has the biggest effect for this bucket size. For bucket size 5, most leaves are very small and do not profit much from bijection rotation. For larger buckets, the aggregation and higher levels make up a larger portion of the running time, and they do not profit at all from bijection rotation. Generally, bijection rotation leads to smaller construction time. As expected, the difference is greater for larger leaf sizes; see Section 5.5.1 for an analysis.

Other than the other implementations, the GPU implementation gets faster for larger bucket sizes, as is quite obvious in Figure 6.20b. The reason is that the GPU can be better utilized and the overhead is reduced since the kernel calls of the leaf and aggregation levels contain more thread blocks. The large overheads are also the reason why the construction time of GPURecSplit does not increase for larger leaf sizes until about leaf size 12. For smaller leaf sizes, the overhead dominates the construction time. As a corollary, it is never really useful to use GPURecSplit with $\ell < 12$ since we get a faster and more compact MPHF with the same construction time by using $\ell = 12$.

For small leaf or bucket sizes, SIMDRecSplit is the fastest implementation, but the GPU implementation overtakes it if both parameters are large. In this case, GPURecSplit can reach very large speedups. Unfortunately, the speedup of the SIMD implementation decreases for large leaf sizes as visible in Figure 6.20a. We have seen something similar in Section 6.4.6, but the same explanation is not applicable here. If the reason was CPU throttling due to more vector instructions, then the CPU was actually better utilized since vector instructions are doing more work per instruction. Perhaps a better explanation is

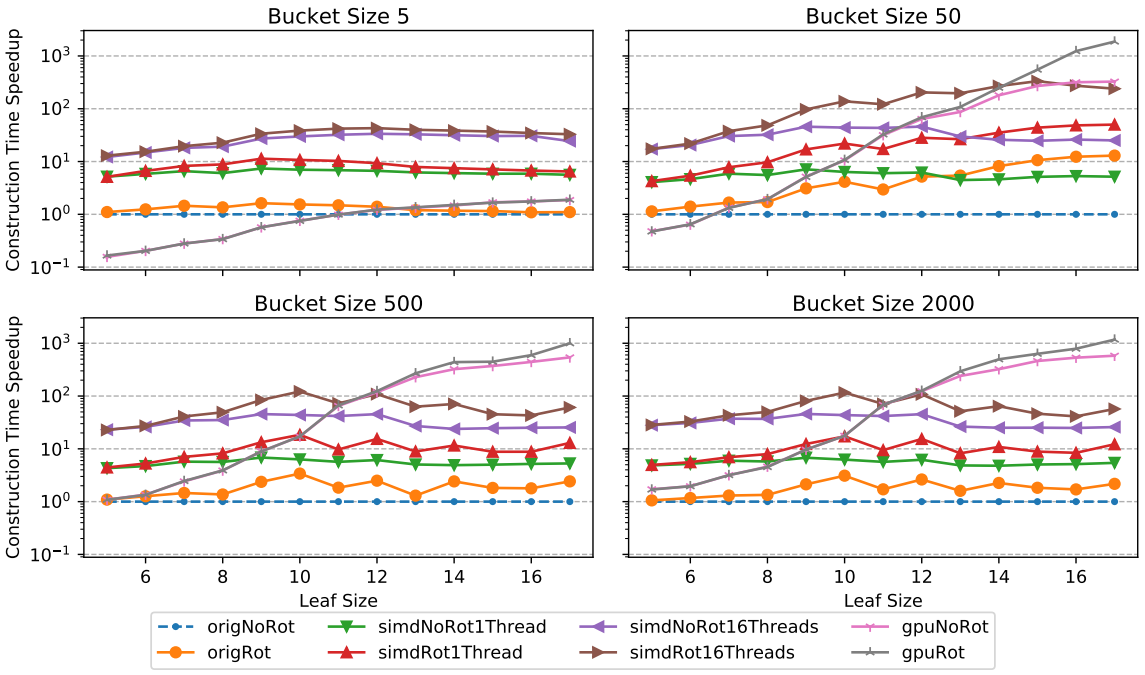


(a) One million keys

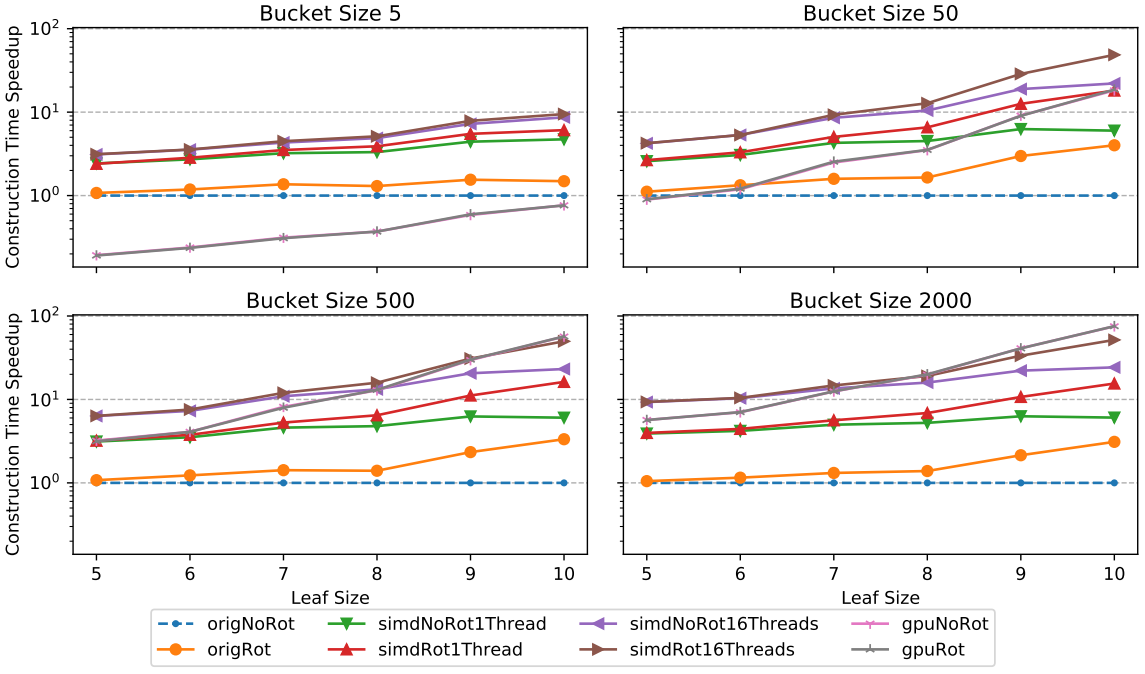


(b) One billion keys

Figure 6.20: Construction time of all our RecSplit implementations for different bucket and leaf sizes.



(a) One million keys



(b) One billion keys

Figure 6.21: Construction time speedup of all our RecSplit implementations relative to the original implementation without bijection rotation.

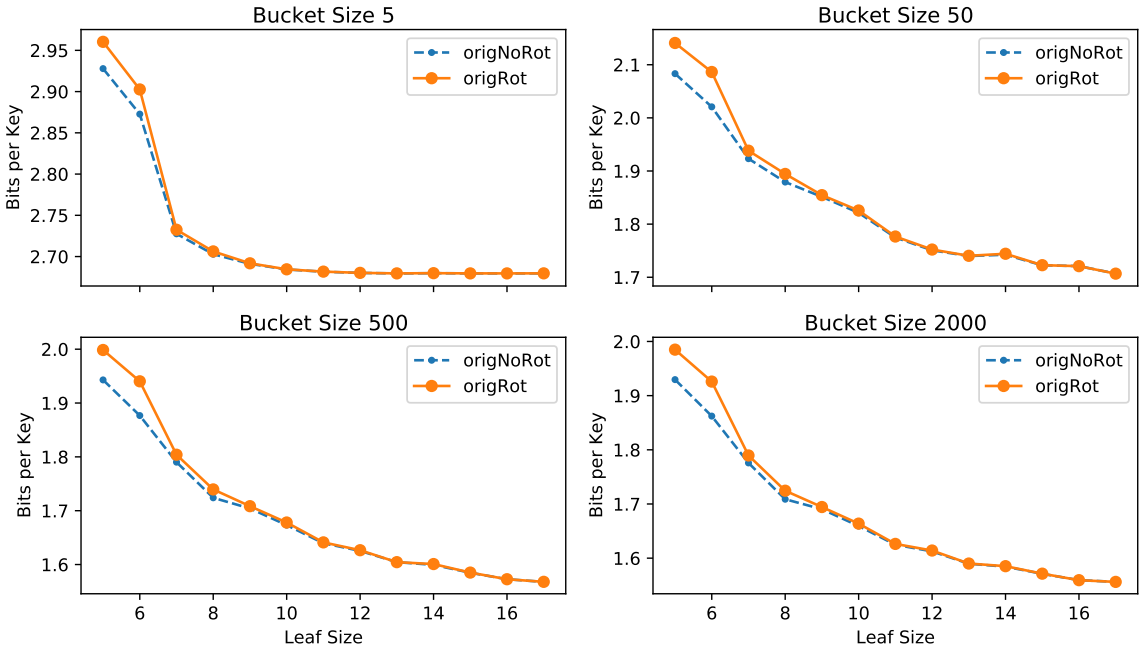


Figure 6.22: Space consumption of the original implementation and our version with bijection rotation. The input consists of one million keys. The other implementations are not shown since the space consumption is nearly identical. For the same reason, the plots for a billion input keys are omitted.

that both, bijection midstop and bijection rotation, work better in scalar code than in SIMD code, and these techniques become increasingly powerful for large leaf sizes. We have to do quite some extra work for bijection midstop which is not necessary in the original implementation, see Section 5.4.3. The bijection rotation midstop is also not as effective, and if at least one SIMD lane has not found a collision, the whole vector has to be finished processing; see Section 5.5.3.

6.6.2 Space Consumption

The space consumption of RecSplit with and without bijection rotation is shown in Figure 6.22. The SIMD and GPU implementations have virtually the same space consumption as the respective version of sequential RecSplit with or without bijection rotation. The difference is less than 0.002 bits per key in both directions for a million keys, and nearly nonexistent for a billion keys. This means the differences in the final MPHF, in particular the use of the 32-bit hash function, have no significant effect on the space requirements.

As expected, the space consumption decreases for larger bucket and leaf sizes. For small buckets, the Double Elias-Fano representation takes up a lot of space such that the total space consumption is between 2.6 and 3.0 bits per key. However, for bucket size 50 it is already very small compared to the space consumption of the Rice Bit Vector. The total space consumption is between 1.7 and 2.2 bits per key. Increasing the bucket size further to 2000 only saves about 0.1 bits per bucket. This comes at the cost of much slower queries as we will see later. Increasing the leaf size has a greater effect. Of course, this leads to longer construction time, but the query times are improved as well.

Using bijection rotation increases the space consumption by about 0.05 bits per key for leaf size 5, but the difference shrinks for larger leaf sizes and becomes insignificant for fairly large leaf sizes. To understand this behavior, take a look back at Theorem 5.1 in Section 5.5.1. It tells us that given a leaf of size m, we need to try out on average a factor of m fewer hash function. To be able to store the correct rotation, only every m-th hash

function is tried. This means that the number stored in the Rice Bit Vector is on average the same, i.e., the space consumption should be identical.

However, we made some assumptions to prove Theorem 5.1. If those are not true, the factor by which the number of trials are reduced is smaller than m. Since we still only try every m-th hash function, the space consumption is larger than without bijection rotation. For larger leaf sizes, the assumptions are approximately met, and consequently the reduction factor is approximately m. For example, $|A| \approx |B|$ if m is large according to the law of large numbers. Similarly, the number of distinct rotations of a is on average closer to m the larger m is. This is not sufficient to fulfill the assumptions of the theorem, but the calculations are closer to the truth the more close $|A|$ and $|B|$ are, and the more close the distinct number of rotations of a is to m. In conclusion, the difference of using bijection rotation versus not using bijection rotation decreases for larger leaf sizes.

6.6.3 Query Time

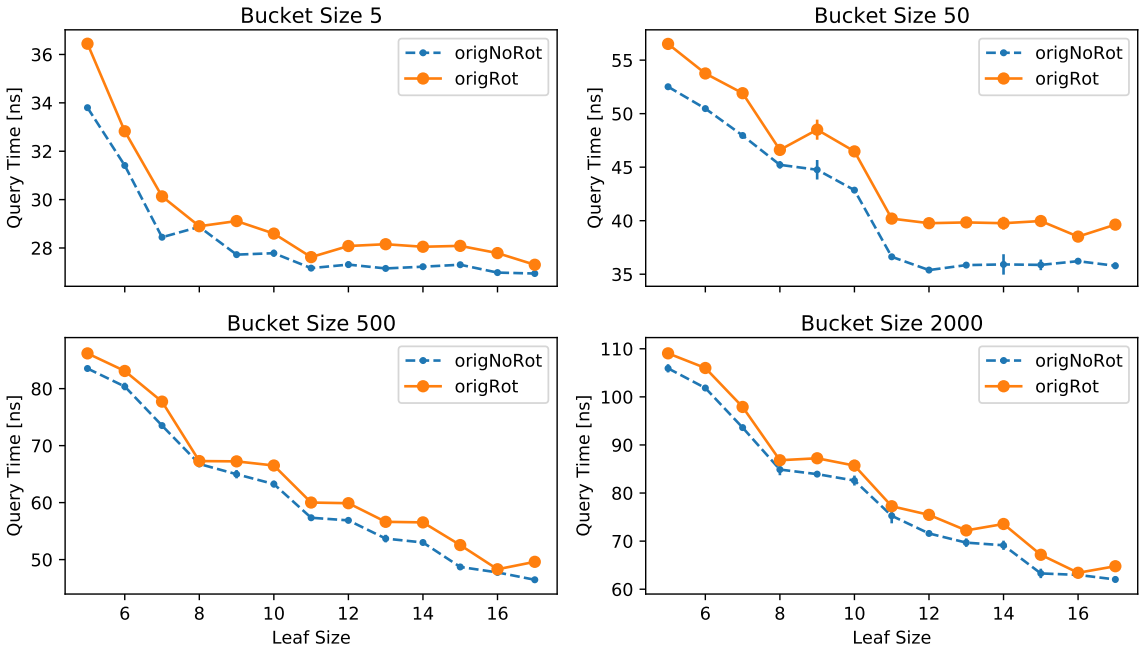
To avoid overcrowded plots, the plots for the query time are split up into plots only containing the query time of the original implementations (Figure 6.23) and plots containing the speedups of the remaining implementations relative to the original implementations (Figure 6.24). As expected, the query times generally increase for larger bucket sizes but decrease for larger leaf sizes. This is because the query time is fundamentally dependent on the height of the splitting trees, and the height is larger for bigger buckets but smaller for large leaf sizes due to higher fanouts in the aggregation levels.

The query times are significantly larger for a billion keys than for a million keys. The reason is not algorithmic, as the query time is $O(1)$, but simply cache related. Even for bucket size 5 and leaf size 5, the MPHF of a million keys takes up less than 3 MB (see Figure 6.22) which fits easily in L3 cache. For a billion keys, the space consumption is three orders of magnitude larger. Caching can also explain an interesting anomaly in Figure 6.23b. The query times of a billion keys are smaller for bucket size 500 than for bucket size 50. The reason is probably that the Double Elias-Fano representation is smaller for bucket size 500, hence a larger part of it fits in the cache. For example using the formulas in Section 4.5.2, the jump array needs about 266 kB and fits in L2 cache while the upper-bits array together take up at most 12 MB and therefore fit in L3 cache. For bucket size 50, both are ten times larger.

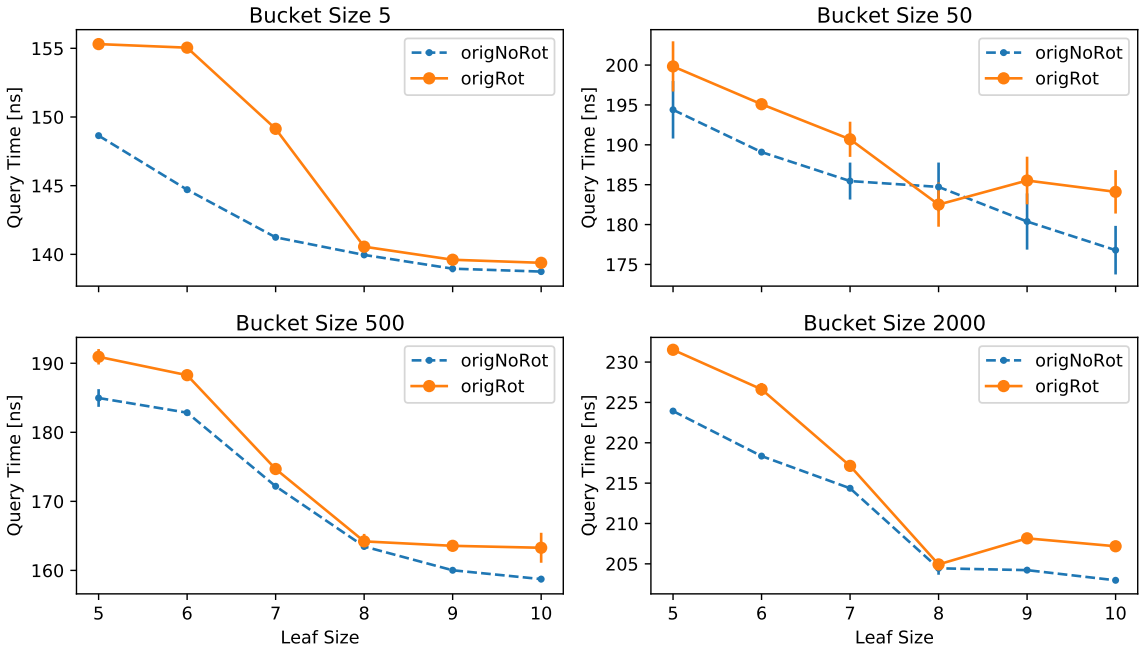
Using bijection rotation, the query times are a few percent larger. This is expected since more work is necessary. First, an additional case distinction is required to check whether the found leaf is full (i.e., $m = \ell$) because only full leaves use bijection rotation. If it is full, a modulo operation is used to calculate the rotation. In the case where the key was in the set B which was rotated (see Section 5.5), we have to apply the rotation which needs another modulo operation. Fortunately, the modulus is in both cases the compile-time constant ℓ. Therefore, the compiler is able to avoid expensive modulo operations similar to Section 5.6.1. In the simplest case, ℓ is a power of two. Then, modulo can be replaced by a single inexpensive AND instruction. This is the reason why the difference between the two implementations in Figure 6.23 is smaller for $\ell = 8$ and $\ell = 16$.

Looking at Figure 6.24, the SIMD and GPU implementations produce neither generally faster nor generally slower MPHFs. In most cases, the difference is less than 2%. However, for a million keys and bucket size 5, the speedup is up to 8%. The reason could be that the starting seeds are compile-time constants (see Section 5.2), but it is unclear whether this can really explain such great differences. At least it is reasonable that the influence is greater for bucket size 5 than for larger buckets as the overall work is smaller.

Interestingly, the chart is flipped for a billion keys. The GPU implementation produces an MPHF that is up to 11% slower than the original implementation. Given the fact that the

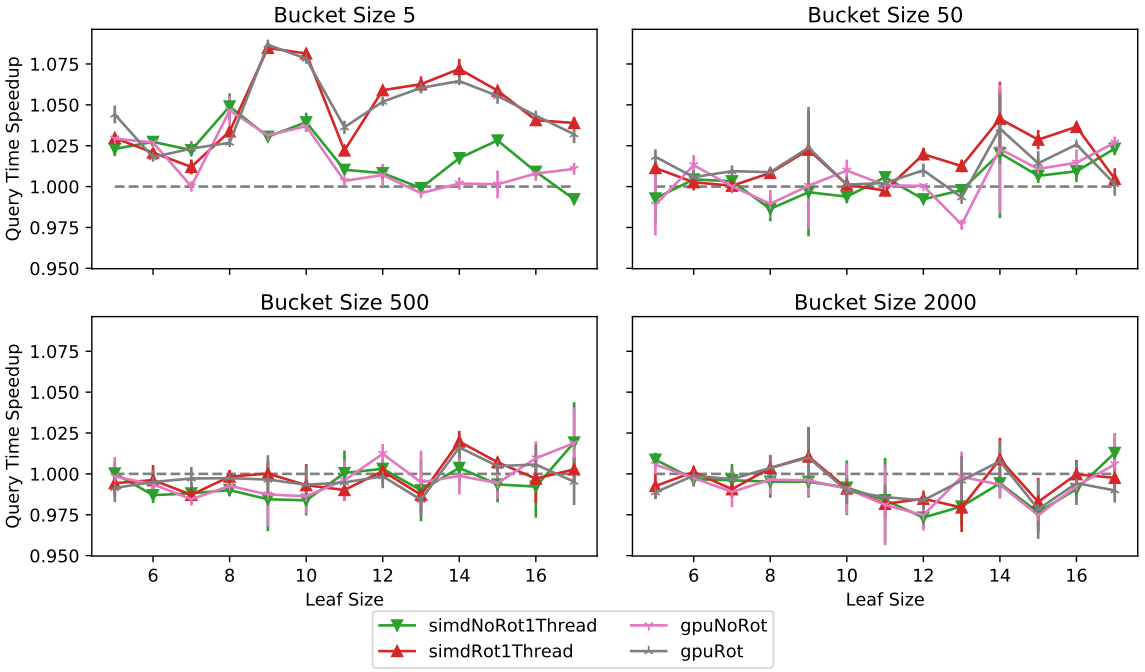


(a) One million keys

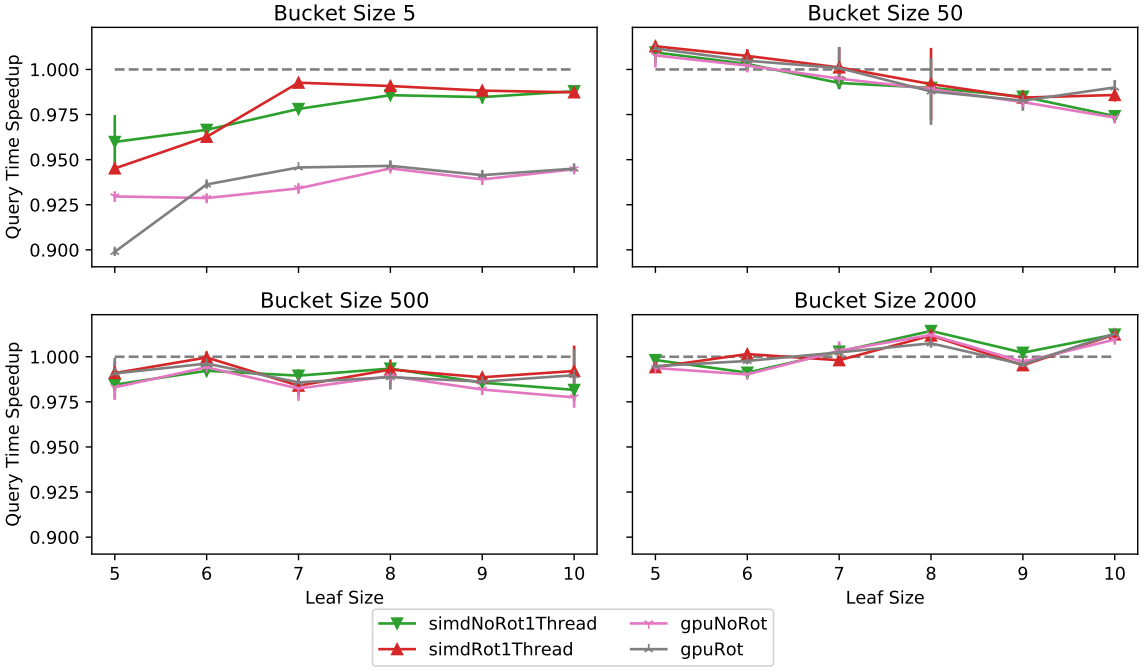


(b) One billion keys

Figure 6.23: Query time of all our RecSplit implementations for different bucket and leaf sizes. Only the original implementations are shown for a better visualization.



(a) One million keys



(b) One billion keys

Figure 6.24: Query time speedup of all our RecSplit implementations relative to the original implementation. The speedups with bijection rotation are relative to the original implementation with bijection rotation, and the speedups without bijection rotation are relative to the original implementation without bijection rotation.

GPU implementation produces the exact same MPHF as the SIMD implementation which is much faster in this case, this difference is perhaps better explained with the measuring method than an actual difference in the result. The queries are timed directly after the MPHF is constructed. Maybe there is a difference in the clock frequency of the GPU. Unfortunately, we can only speculate at this point.

6.6.4 Pareto Fronts

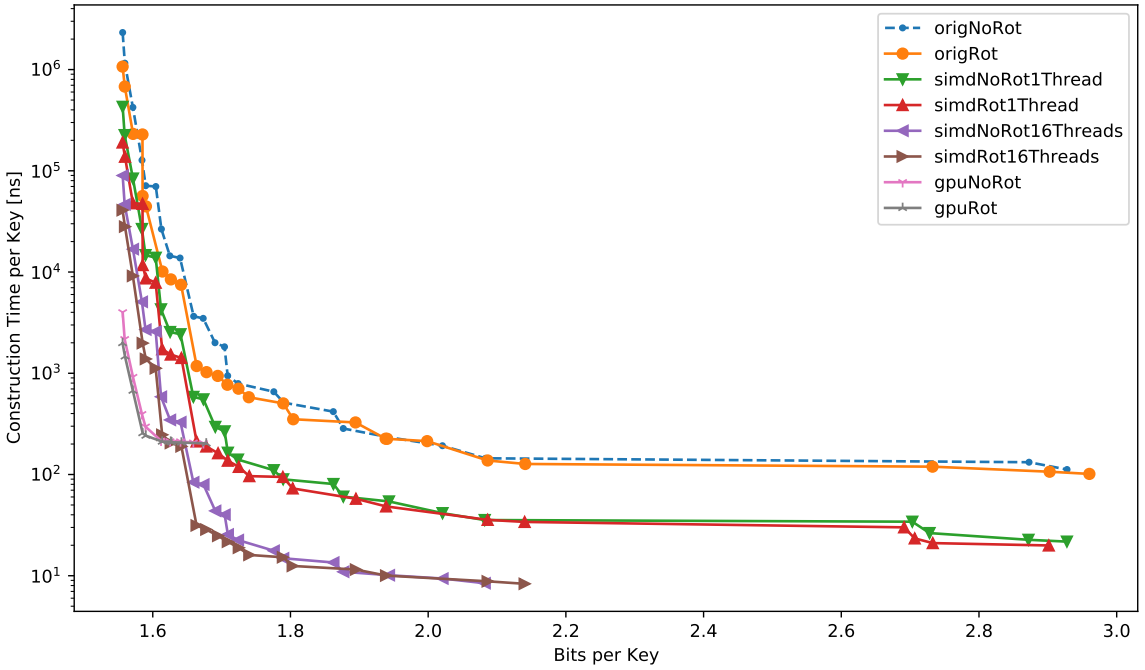
If the user does not care for the query time, Figure 6.25 tells us that GPURecSplit with bijection rotation is the best choice for very compact MPHFs, whereas SIMDRecSplit with bijection rotation is better suited for very fast construction at the cost of more bits per key. The original implementation is never useful as it is completely dominated by the SIMD implementation. For a billion keys, GPURecSplit with bijection rotation, bucket size 2000 and leaf size 17 dominates almost everything. Only the SIMD implementation can construct the MPHF slightly faster at the expense of significantly more space. Figure 6.25 also shows us that bijection rotation is indeed useful, i.e., the slightly higher space consumption can be offset by tweaking the parameters such that the construction is still faster and produces a more compact MPHF.

If the user only cares about query and construction time, Figure 6.26 clearly states that SIMDRecSplit is the preferred choice. This is expected since it is generally a lot faster than the original implementation, and GPURecSplit is slow for small bucket sizes which are necessary for fast queries. Bijection rotation seems to be a bit better for the SIMD implementation, but this is not the case for the other implementations.

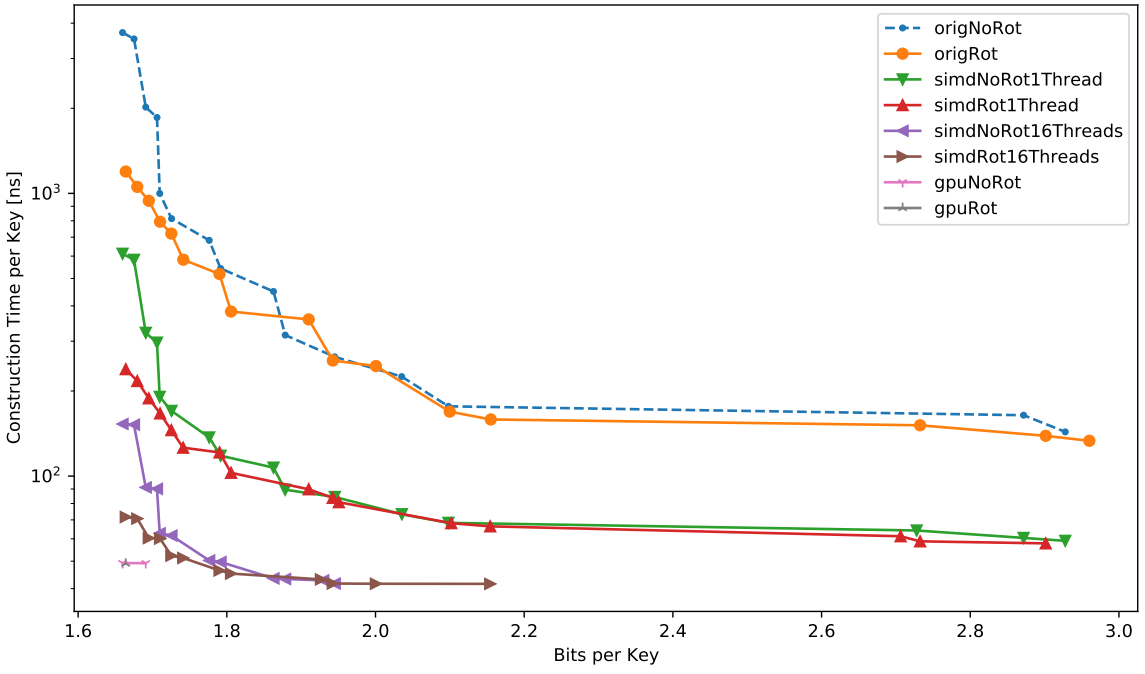
For the sake of completeness, we show the Pareto fronts of query time versus space consumption in Figure 6.27. There are no big differences between the implementations since the goal of our implementations is to improve the construction time while retaining the query time and space consumption as good as possible. That said, bijection rotation is counterproductive if the construction time is irrelevant since it slightly increases space consumption and query time.

6.7 Comparison with PTHash

To the best of our knowledge, our implementations of RecSplit are the best methods to construct very space-efficient MPHFs (below 2 bits per key) at practical construction and query times. Currently, the best method for very fast queries at reasonable construction time and space consumption is PTHash [76, 75]; see Section 3.4. We have tested PTHash with the parameters used in a paper by the authors [75]. As evident in Table 6.2, SIMDRecSplit with bucket size 7 and leaf size 11 has better construction time and space consumption than PTHash. However, PTHash has up to 2.6× faster queries. No RecSplit implementation can come close to the query times of PTHash. Even by increasing the leaf size further, only minuscule improvements can be measured since with $b = 7$ and $\ell = 11$ most splitting trees only consist of a single leaf.

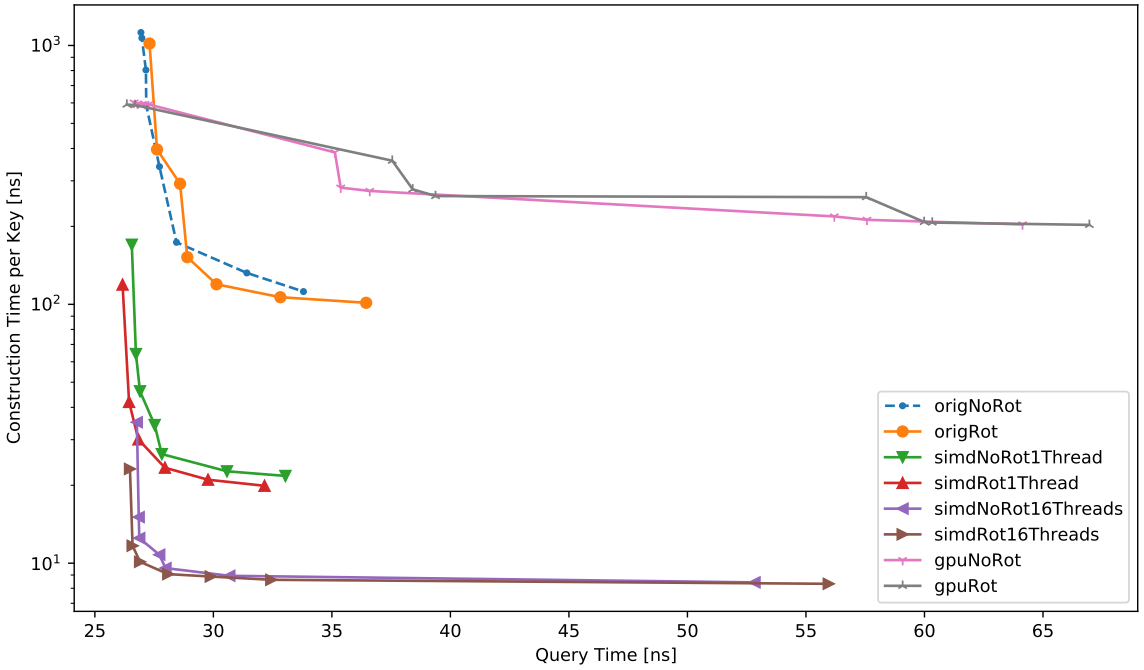


(a) One million keys

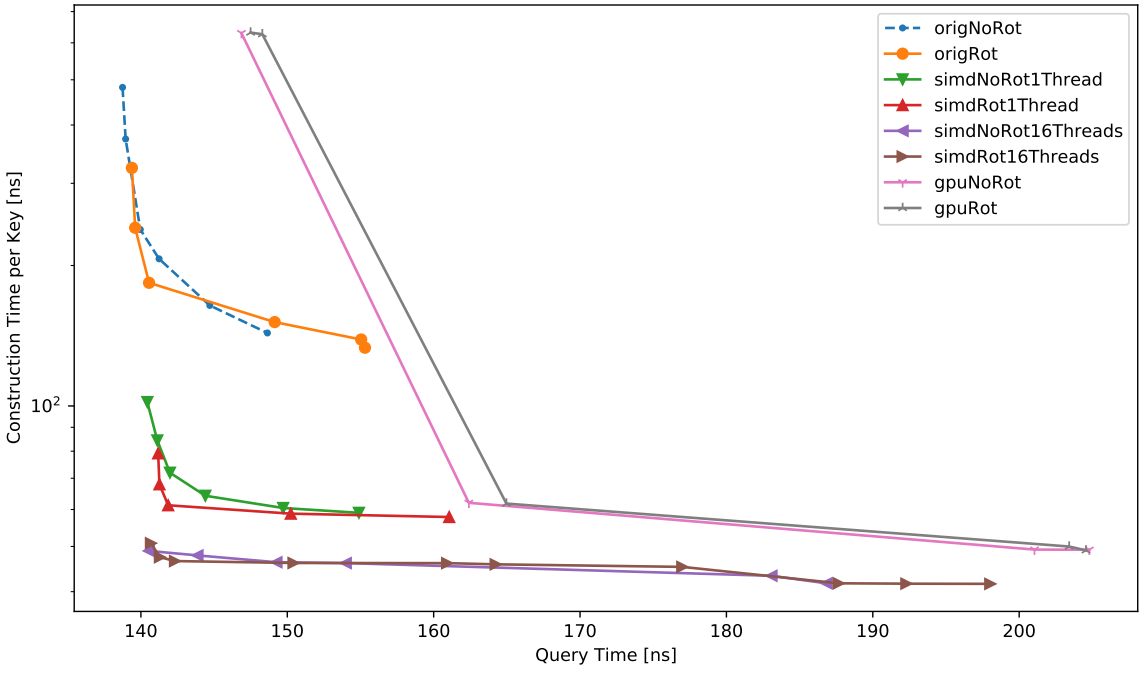


(b) One billion keys

Figure 6.25: Pareto fronts (construction time versus space consumption) for the different implementations.

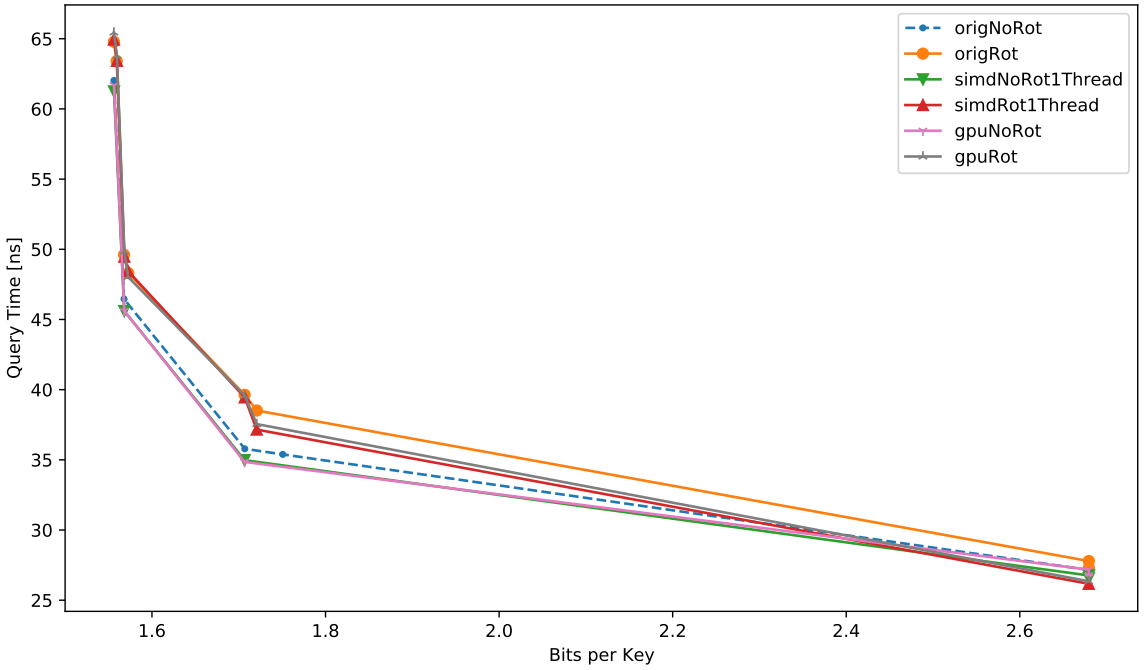


(a) One million keys

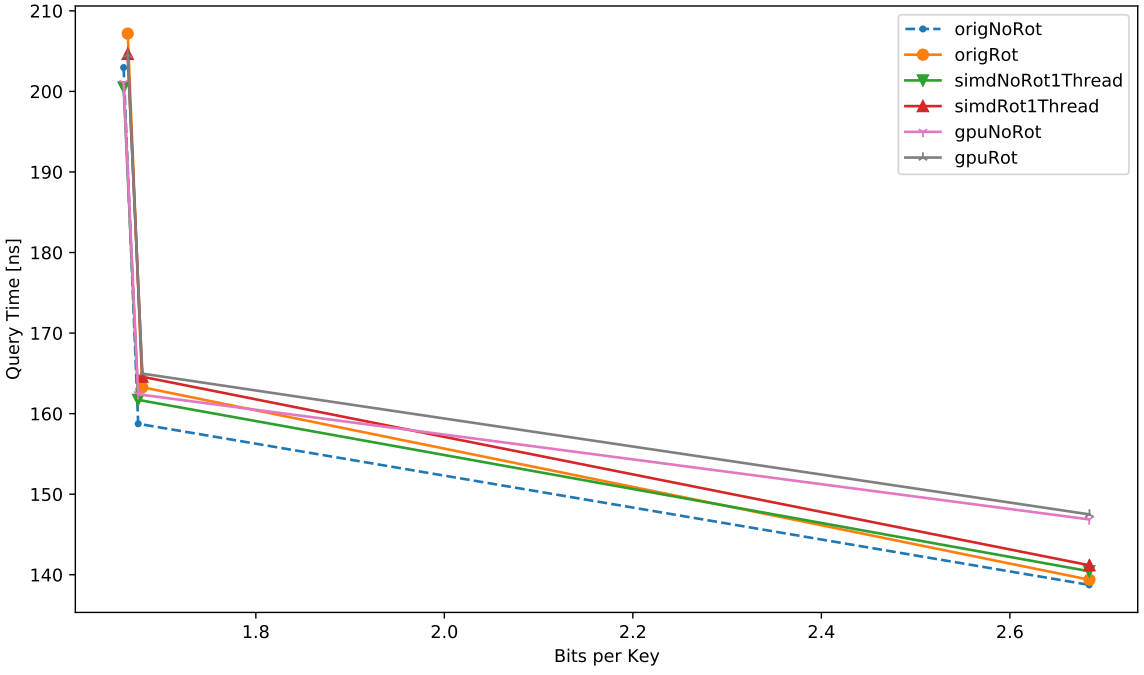


(b) One billion keys

Figure 6.26: Pareto fronts (construction time versus query time) for the different implementations.



(a) One million keys



(b) One billion keys

Figure 6.27: Pareto fronts (query time versus space consumption) for the different implementations.

Table 6.2: Comparison with internal-memory PTHash. SIMDRecSplit and PTHash both use 16 threads. For SIMDRecSplit, bucket size $b = 7$ and leaf size $\ell = 11$. Bijection rotation is not used. As in a paper by the authors of PTHash [75], it uses $\alpha = 0.94$ and $c = 7$ with the encoder mentioned in parentheses; see [76] for details.

Method	1 million keys			10 million keys			100 million keys		
	Constr. [ns/key]	Space [bits/key]	Query [ns]	Constr. [ns/key]	Space [bits/key]	Query [ns]	Constr. [ns/key]	Space [bits/key]	Query [ns]
SIMDRecSplit	46	2.39	29	52	2.39	37	83	2.39	73
PTHash (D-D)	102	3.48	12	98	3.34	14	112	3.23	26
PTHash (PC)	95	3.11	13	99	2.91	16	110	2.76	29
PTHash (EF)	101	2.70	21	99	2.54	27	108	2.45	45

7. Conclusion

In this thesis, we present new techniques to construct minimal perfect hash functions (MPHFs) faster. In particular, we provide new implementations of the RecSplit MPHF [46] that make use of multithreading, SIMD, and GPUs. We also propose a new technique called *bijection rotation* to construct MPHFs on small key sets of size m with almost optimal space consumption and an up to m times faster construction than the usual brute-force approach. Using bijection rotation, we achieve speedups of up to 333 for our SIMD implementation on an 8-core CPU, and up to 1873 for our GPU implementation compared to the original, sequential implementation in the Sux library [13]. This makes it possible, for example, to construct an MPHF with 1.56 bits per key in less than 1.5 µs per key.

7.1 Future Work

There are many ways to improve our work or RecSplit in general. We propose some ideas in this section for possible future work.

7.1.1 Bijection Rotation

We introduced bijection rotation in Section 5.5 and provided a rudimentary analysis in Section 5.5.1. For future work, it would be interesting to analyze bijection rotation more thoroughly, in particular the average savings without using assumptions that are not always true. Based on this analysis, it would be wise to adapt the fanouts in the aggregation levels. The fanouts were chosen in the original RecSplit paper [46] such that the expected work in both aggregation levels is approximately equal to the expected amount of work in the leaf level. With bijection rotation, the expected amount of work in the leaf level is reduced. Therefore, the fanouts should be decreased as well. We have not yet done this, which is the reason why bijection rotation is currently the most useful for bucket size about 50.

7.1.2 Combine RecSplit with SAT-Based MPHF

We mentioned a SAT-based MPHF [81] in Section 3.5. The authors proposed two techniques: one that achieves about 1.83 bits per keys in practical time, and one that is nearly optimal, albeit slow and only feasible for small inputs of at most about 40 keys. Future work could try to use this second technique to compute the bijections in RecSplit. Perhaps, this is faster, especially for large leaves. If it is indeed faster, then it is interesting to see whether SAT-based techniques can also be used to accelerate the search for splittings.

7.1.3 Scalability

We discussed the use of multiple GPUs for GPURecSplit in Section 6.5.6. The authors of RecSplit have already denoted that RecSplit can easily make use of distributed computing (e.g., with MapReduce [42]) and external storage. The latter can be achieved by using an external sorting algorithm [43], the remaining steps only need one bucket in main memory at a time. This allows for RecSplit implementations that scale well beyond a few billion keys.

However, the more input keys, the higher the risk for hash collisions in the initial hash function, see Section 4.1. If a collision appears, RecSplit cannot determine an MPHF (the implementation runs into an endless loop). To diminish the chance of a collision, a 128-bit hash function is used. Unfortunately, not every bit is used. The 64 least significant bits of the 128-bit key are used to find splittings and bijections inside the bucket, but the 64 most significant bits are only used to calculate the bucket of the key. Since there are only $\lceil \frac{n}{b} \rceil$ buckets, the extracted information contains only $\log_2 \lceil \frac{n}{b} \rceil$ bits. This is obvious if the number of buckets is a power of two: the bucket-assignment function (see Section 4.2) just extracts the $\log_2 \lceil \frac{n}{b} \rceil$ most significant bits of the key.

The result is that RecSplit has not the collision resistance one would expect from 128-bit keys. For example, consider $n = \left\lceil 1.18\sqrt{2^{64}} \right\rceil \approx 5$ billion keys and bucket size $b = 2000$. According to the birthday paradox [65], there is at least one collision in the 64 least significant bits of the keys with a probability of more than 50%. There are 2.5 million buckets, and the construction fails if two colliding keys end up in the same bucket. It follows that the probability of failure is at least $0.5 \cdot \frac{1}{2\,500\,000} = \frac{1}{5\,000\,000}$. It is up to the reader whether a chance of 1 in 5 million is an acceptable risk. Of course, this also depends on the situation. The valuation is different if the construction is run once with a human supervisor versus if it is used in a consumer application that is used by millions of users.

It is important to note that this problem results in a practical limitation of the input size. For roughly 13 quadrillion keys ($13 \cdot 10^{15} \approx 1.44 \cdot 2^{53}$, or 208 petabytes of data), the probability of failure is more than 50%. According to the birthday paradox, that is a factor of 1674 smaller than the $1.18\sqrt{2^{128}}$ keys that would provide the same limit if the whole 128 bits were used.

Solutions

A simple solution is checking for collisions in every bucket. If there is a collision in any bucket, use another seed for the initial hash function and try again. This is reasonable if the chance of failure is not too large. Another solution is to use the whole 128-bit keys for the construction of the splitting trees. This way, RecSplit has the same collision resistance as expected from 128-bit keys, but this comes at the cost of significantly increased construction time (presumably at least a factor of 2). One solution should be implemented for a scalable and practical implementation of RecSplit.

7.1.4 Generalizations

The authors of RecSplit remark that RecSplit can be generalized to perfect hash functions that are not minimal, i.e., the codomain is not $[n]$, but $[m]$ for $m > n$ and the perfect hash function is only required to be injective instead of bijective. Such perfect hash functions can be even more compact than MPHFs. Similarly, RecSplit can be generalized to k-perfect hashing to save space, where up to k keys per hash value are allowed. This has been implemented by the author of this thesis, but the result is (yet) unpublished. A future production-ready implementation of RecSplit could offer both generalizations and decide automatically whether a SIMD or GPU implementation is used based on the parameters (bucket and leaf size) and the presence of a suitable GPU.

Bibliography

[1] BwUniCluster 2.0 Hardware and Architecture - 2 Components of bwUniCluster. https://wiki.bwhpc.de/e/BwUniCluster_2.0_Hardware_and_Architecture#Components_of_bwUniCluster. Accessed: September 11, 2022.

[2] Clang: a C language family frontend for LLVM. https://clang.llvm.org/. Accessed: September 10, 2022.

[3] CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: August 9, 2022.

[4] CUDA Toolkit Documentation - Integer Intrinsics - Funnel Shift. https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html#group__CUDA__MATH__INTRINSIC__INT_1g6afcc1126ca2bf68f74c34812cbde57d. Accessed: August 8, 2022.

[5] Intel Core i7-11700 Processor. https://ark.intel.com/content/www/us/en/ark/products/212279/intel-core-i711700-processor-16m-cache-up-to-4-90-ghz.html. Accessed: September 8, 2022.

[6] Intel Core i7-7700 Processor. https://ark.intel.com/content/www/us/en/ark/products/97128/intel-core-i77700-processor-8m-cache-up-to-4-20-ghz.html. Accessed: September 10, 2022.

[7] Intel Xeon Gold 6230 Processor. https://ark.intel.com/content/www/us/en/ark/products/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz.html. Accessed: September 11, 2022.

[8] MurmurHash3 Github. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp. Accessed: July 18, 2022.

[9] Nvidia Developer Blog - Cooperative Groups: Flexible CUDA Thread Programming. https://developer.nvidia.com/blog/cooperative-groups/. Accessed: August 8, 2022.

[10] Nvidia Developer Blog - CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics. https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/. Accessed: August 8, 2022.

[11] Nvidia Developer Blog - Using CUDA Warp-Level Primitives. https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/. Accessed: August 8, 2022.

[12] Nvidia Developer Forums - Question about 64 Bit Integer Performance. https://forums.developer.nvidia.com/t/question-about-64-bit-integer-performance/64147. Accessed: August 18, 2022.

[13] Sux Github. `https://github.com/vigna/sux`. Accessed: July 11, 2022.

[14] Intel Advanced Vector Extensions Programming Reference. `https://www.intel.com/content/dam/develop/external/us/en/documents/36945`, 2011. Accessed: August 16, 2022.

[15] Intel AVX-512 Instructions. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html`, 2017. Accessed: August 16, 2022.

[16] Nvidia Tesla V100 GPU Architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017. Accessed: September 11, 2022.

[17] Nvidia Ampere GA102 GPU Architecture. `https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf`, 2020. Accessed: August 16, 2022.

[18] cppreference.com - std::lock_guard. `https://en.cppreference.com/w/cpp/thread/lock_guard`, 2021. Accessed: August 19, 2022.

[19] Linux manual page - export(1p). `https://man7.org/linux/man-pages/man1/export.1p.html`, 2021. Accessed: August 19, 2022.

[20] Linux manual page - lscpu(1). `https://man7.org/linux/man-pages/man1/lscpu.1.html`, 2021. Accessed: August 19, 2022.

[21] Intel 64 and IA-32 Architectures Software Developer Manuals. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`, 2022. Accessed: August 18, 2022.

[22] Intel Intrinsics Guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`, 2022. Accessed: August 18, 2022.

[23] Nvidia Nsight Systems. `https://developer.nvidia.com/nsight-systems`, 2022. Accessed: August 19, 2022.

[24] Agner Fog. Instruction tables - Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. `https://www.agner.org/optimize/instruction_tables.pdf`. Accessed: August 18, 2022.

[25] Agner Fog. Optimizing software in C++ - An optimization guide for Windows, Linux, and Mac platforms. `https://www.agner.org/optimize/optimizing_cpp.pdf`. Accessed: August 16, 2022.

[26] Agner Fog. Vector Class Library. `https://github.com/vectorclass/version2`. Accessed: July 11, 2022.

[27] Agner Fog. Vector Class Library Manual. `https://www.agner.org/optimize/vcl_manual.pdf`. Accessed: July 11, 2022.

[28] Jean-Philippe Aumasson and Daniel J. Bernstein. Siphash: a fast short-input prf. Cryptology ePrint Archive, Paper 2012/351, 2012. `https://eprint.iacr.org/2012/351`.

[29] Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. Cache-oblivious peeling of random hypergraphs. In *2014 Data Compression Conference*, pages 352–361, 2014.

[30] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In Mark de Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, pages 427–438, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[31] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, pages 682–693, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[32] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Trans. Algorithms*, 10(4), August 2014.

[33] Jules Bertrand, Fanny Dufossé, Somesh Singh, and Bora Uçar. Algorithms and data structures for hyperedge queries. Research Report RR-9390, Inria Grenoble Rhône-Alpes, 2022. Revised version April 2022.

[34] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[35] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Inf. Syst.*, 38(1):108–131, March 2013.

[36] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 59–65, New York, NY, USA, 1978. Association for Computing Machinery.

[37] Chin-Chen Chang and Chih-Yang Lin. Perfect hashing schemes for mining association rules. *The Computer Journal*, 48(2):168–179, 2005.

[38] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, page 30–39, USA, 2004. Society for Industrial and Applied Mathematics.

[39] Chuang-Kai Chiou and Judy C. R. Tseng. An incremental mining algorithm for association rules based on minimal perfect hashing and pruning. In Hua Wang, Lei Zou, Guangyan Huang, Jing He, Chaoyi Pang, Hao Lan Zhang, Dongyan Zhao, and Zhuang Yi, editors, *Web Technologies and Applications*, pages 106–113, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[40] Victoria G Crawford, Alan Kuhnle, Christina Boucher, Rayan Chikhi, and Travis Gagie. Practical dynamic de Bruijn graphs. *Bioinformatics*, 34(24):4189–4195, June 2018.

[41] David Strafford. David Strafford Blogspot. `http://zimbry.blogspot.com/2011/09/better-bit-mixing-improving-on.html`. Accessed: July 18, 2022.

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[43] Roman Dementiev and Peter Sanders. Asynchronous parallel disk sorting. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, page 138–148, New York, NY, USA, 2003. Association for Computing Machinery.

[44] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer Verlag, 1986.

[45] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, April 1974.

[46] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. RecSplit: Minimal perfect hashing via recursive splitting. In *2020 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185, 2020.

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

[48] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[49] Catherine Forbes, Merran Evans, Nicholas Hastings, and Brian Peacock. *Statistical Distributions, Fourth Edition.* John Wiley & Sons, Ltd, 2010.

[50] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '92, page 266–273, New York, NY, USA, 1992. Association for Computing Machinery.

[51] 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, June 1984.

[52] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[53] Kimmo Fredriksson and Fedor Nikitin. Simple compression code supporting random access and fast string matching. In Camil Demetrescu, editor, *Experimental Algorithms*, pages 203–216, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[54] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms*, pages 339–352, Cham, 2016. Springer International Publishing.

[55] GNU Project. GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`. Accessed: September 8, 2022.

[56] GNU Project. libstdc++ Introsort Implementation. `https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01347.html#l02245`. Accessed: June 3, 2022.

[57] S. Golomb. Run-length encodings (corresp.). *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[58] Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *ACM J. Exp. Algorithmics*, 25, March 2020.

[59] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. *SIGPLAN Not.*, 29(6):61–72, June 1994.

[60] Bob Jenkins. SpookyHash: a 128-bit noncryptographic hash. `https://burtleburtle.net/bob/hash/spooky.html`, 2012. Accessed: August 22, 2022.

[61] Aaron B. Kiely. Selecting the Golomb Parameter in Rice Coding. *Interplanetary Network Progress Report*, 42-159:1–18, 2004.

[62] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66–104, 1999.

[63] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Trans. Graph.*, 25(3):579–588, July 2006.

[64] Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1), January 2019.

[65] Arjen K. Lenstra. Birthday paradox. In *Encyclopedia of Cryptography and Security*, pages 147–148. Springer US, Boston, MA, 2011.

[66] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[68] B. S. Majewski, N. C. Wormald, G. Havas, and Z. J. Czech. A Family of Perfect Hashing Methods. *The Computer Journal*, 39(6):547–554, January 1996.

[69] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *Proceedings of the 13th International Symposium on Experimental Algorithms - Volume 8504*, page 138–149, Berlin, Heidelberg, 2014. Springer-Verlag.

[70] David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[71] Nvidia. CUDA Developer Website. `https://developer.nvidia.com/cuda-zone`. Accessed: July 11, 2022.

[72] Muhammad Osama, Anton Wijs, and Armin Biere. Sat solving with gpu accelerated inprocessing. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 133–151, Cham, 2021. Springer International Publishing.

[73] Rasmus Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In Frank Dehne, Jörg-Rüdiger Sack, Arvind Gupta, and Roberto Tamassia, editors, *Algorithms and Data Structures*, pages 49–54, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[74] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. CUD@SAT: Sat solving on gpus. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3):293–316, 2015.

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

[76] Giulio Ermanno Pibiri and Roberto Trani. PTHash: Revisiting FCH minimal perfect hashing. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '21, page 1339–1348, New York, NY, USA, 2021. Association for Computing Machinery.

[77] Giulio Ermanno Pibiri and Rossano Venturini. Efficient data structures for massive n-gram datasets. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '17, page 615–624, New York, NY, USA, 2017. Association for Computing Machinery.

[78] R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology*, 19(6):889–897, 1971.

[79] D. Salomon, G. Motta, and D. Bryant. *Data Compression: The Complete Reference*. Molecular biology intelligence unit. Springer London, 2007.

[80] Harold H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Massachusetts Institute of Technology, 1954.

[81] Sean Weaver and Marijn Heule. Constructing minimal perfect hash functions using sat technology. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(02):1668–1675, April 2020.