Universität Bielefeld
Technische Fakultät

Bachelor-Arbeit

SIMD Implementation of MSB Radix Sort with Separate Key and Payload Datastreams

im Studiengang
Informatik

Jonas Richard Keller

14. Juli 2022

Betreuer
Prof. Dr.-Ing. Ralf Möller, AG Technische Informatik
M.Sc. Jan O'Sullivan, AG Technische Informatik

Abstract

In this thesis a SIMD (Single Instruction Multiple Data) implementation of the MSB Radix Sort algorithm with separate key and payload datastreams is presented. The implementation makes use of AVX-512 `mask_compresstoreu` instructions and is based on the implementation with combined key and payload datastreams developed by Möller (2021).

This thesis explores whether a separation of key and payload datastreams provides an improvement in the performance of the SIMD implementation of MSB Radix Sort algorithm. The separation of key and payload datastreams also enables the algorithm to be generic, supporting arbitrary key and payload data types of arbitrary size with an arbitrary number of payloads.

Besides the separation of the key and payload datastreams, other improvements to the algorithm are implemented as well.

Additionally, a thorough analysis of the performance of the algorithm and comparison with other algorithms for sorting different types and distributions of data is performed.

The experiments show a significant improvement over the implementation by Möller (2021) for most distributions of data up to a factor of 2. A portion of the speedup for datasets with a payload is shown to be due to the separation of the key and payload datastreams. The presented algorithm is however slower than other SIMD implementations of sorting algorithms for many distributions of data.

Hiermit versichere ich, dass ich diese Bachelor-Arbeit selbständig bearbeitet habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und entsprechende Zitate kenntlich gemacht.

Bielefeld, den 14. Juli 2022

Jonas Keller

Contents

1 Introduction **1**

2 Vectorization using SIMD instructions **3**
2.1 SIMD instructions by Intel 4
2.1.1 AVX-512 4
2.1.2 Vector intrinsics 5
2.1.3 SSE/AVX* instructions relevant for this thesis 6

3 Related Work **9**

4 MSB Radix sort **11**
4.1 Sorting different data types 12
4.2 Comparison sorter for small subarrays 13

5 C++ implementation **15**
5.1 Template-wrapper for SIMD instructions 17

6 SIMD Implementation of MSB Radix Sort **21**
6.1 SIMD bit sorter 21
6.2 Example for the SIMD bit sorter 24
6.3 Difference to previous implementation 26
6.4 Handling of Payloads 27
6.4.1 Previous Method 27
6.4.2 Handling different numbers of payloads 28
6.4.3 Handling different sized key and payloads and choosing `numElemsPerVec` 29
6.5 Comparison sorter for small subarrays 32

7 Experiments **33**
7.1 Key and Payload Data Types 33
7.2 Tested sorting algorithms 34

7.3 Input data . . . 35
7.4 Test environment . . . 36
7.5 Time measurements . . . 36
7.6 Checks . . . 37

8 Results **39**
8.1 Determining the optimal comparison sorter threshold . . . 39
8.2 Comparison between RadixSIMD and MoellerCompress . . . 42
8.3 Comparison between RadixSIMDOneReg and RadixSIMD . . . 52
8.4 Comparison with other sorting algorithms . . . 53
8.5 Runtime with respect to the number of elements . . . 58
8.6 Runtime of RadixSIMDNoCmp with respect to the threshold . . . 61

9 Discussion **65**
9.1 Discussion of the results . . . 65
9.2 Possible improvements . . . 66
9.2.1 Caching more vectors . . . 66
9.2.2 Faster sorting algorithm for small subarrays . . . 66
9.2.3 Keeping track of smallest and largest element for early termination . . . 67
9.2.4 AVX/AVX2 version . . . 67
9.2.5 Multithreading . . . 67
9.2.6 Quicksort instead of MSB Radix Sort . . . 68
9.2.7 Additional experiments . . . 68

A C++ code for the SIMD bit sorter class **69**

B Comparison graphs **73**
B.1 `int32` . . . 74
B.2 `int32-int32` . . . 78
B.3 `float` . . . 82
B.4 `float-int32` . . . 86
B.5 `double` . . . 90
B.6 `uint8` . . . 94
B.7 `int16` . . . 98

Bibliography **106**

1

Introduction

Sorting is a very well-known and fundamental problem in computer science. Sorting algorithms are used to solve many problems which include enabling binary search, finding duplicates or finding the k-th largest element. Thus, sorting algorithms are an essential part of many algorithms (Cormen et al., 2009, p. 148; Skiena, 2008, pp. 104-106).

Because of the wide range of applications where sorting is used, the speed of sorting algorithms is important.

There are many ways to improve the speed of an algorithm. Some do not require any changes to the algorithm itself, such as higher CPU clock frequencies or architectural improvements of CPUs. Other improvements do require changes to the sorting algorithm itself, such as the use of multiple CPU cores or theoretical improvements (such as using an algorithm with better complexity).

Another way to speed up some algorithms is to use vector instructions. With vector instructions, operations can be executed for all elements of a vector in parallel, instead of just one element at a time.

Some sorting algorithms have already been improved using SIMD instructions. Möller (2021) has implemented MSB Radix Sort using AVX-512 `mask_compressstoreu` instructions. However, this implementation only supports datasets with one key and at most one payload which has to have the same size as the key. The payloads are also handled in a way that decreases the speedup compared to key-only sorting significantly.

This thesis aims to improve upon Möller's implementation by separating the key and payload datastreams and supporting any number of payloads whose sizes can be different from the key.

The source code of the algorithm developed in this thesis is available at `https://github.com/jonicho/simd-radix-sort` and is licensed under the MIT license. This repository contains the algorithm developed in this thesis as a header-only library `radixSort.hpp`. It also contains the source code of the programs used to test the algorithm and to generate the data sets used for the results.

2

Vectorization using SIMD instructions

Since the algorithm developed in this thesis makes use of SIMD instructions, this chapter provides a short introduction to SIMD instructions.

The abbreviation SIMD stands for *Single Instruction Multiple Data* and is part of Flynn's taxonomy. Flynn's taxonomy is a classification proposed by Flynn (1967), which defines the classes SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data) and MIMD (Multiple Instruction Multiple Data).

SIMD (or vector) instructions operate on (SIMD) vectors, which can be thought of as fixed length arrays of elements of the same length. Usually these vectors are loaded into SIMD (vector) registers. For example, a SIMD register with a length of 512 bit might contain 8 elements of 64 bits each. SIMD instructions operate on these (SIMD) vectors and perform an operation on each element of the vectors (or datastreams) simultaneously (hence the name *Single Instruction Multiple Data*). This is in contrast to conventional SISD (Single Instruction Single Data) instructions, which apply an operation to only a single element (or datastream) at a time.

Figure 2.1 shows a vector addition of two 4-element vectors using a SIMD add instruction as an example. The same operation (in this case an addition) is performed on each element of the vectors simultaneously.

The idea of computing using vector instructions is more than 50 years old with the first computer to feature SIMD instructions being the ILLIAC IV, which was a supercomputer completed in 1966 (Hord, 1982).

Today, SIMD instructions are featured in most modern CPUs.

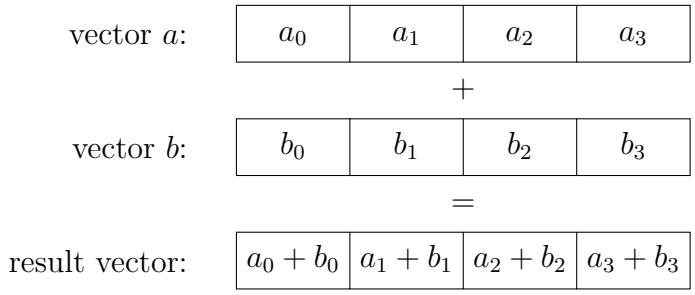


Figure 2.1: Addition of two 4-element vectors using a SIMD add instruction

2.1 SIMD instructions by Intel

The first SIMD instruction set that Intel introduced for its processors was the MMX instruction set (Ch and rasekaran, 1997). It features eight 64 bit wide SIMD registers (`mm0` through `mm7`) which however re-used floating-point registers and only operated on integers (Wikipedia contributors, 2022b). This prevented programs from using floating-point numbers and MMX instructions at the same time.

To fix these shortcomings, Intel introduced the SSE (Streaming SIMD Extensions) instruction set in 1999, which later was expanded to SSE2, SSE3, SSSE3 and SSE4. The SSE instruction sets introduced 16 independent 128 bit wide SIMD registers `xmm0` through `xmm15` and operations for these registers which included floating-point operations (Wikipedia contributors, 2022e).

In 2008, Intel proposed the AVX (Advanced Vector Extensions) instruction set, which, among other things, expands the SSE registers to 256 bits, calling them `ymm0` through `ymm15`. Later, AVX was extended to AVX2, which introduced further new instructions (Wikipedia contributors, 2022a).

2.1.1 AVX-512

AVX-512 is the newest vector instruction set proposed by Intel. It was introduced in 2013 as the successor to AVX and AVX2 (Intel, 2013).

The AVX-512 instruction set expands the SSE and AVX vector registers even more to 512 bits and also doubles the number of vector registers. The AVX-512 registers are called `zmm0` through `zmm31` where the bottom 256 bits of each register are called `ymm0` to `ymm31` and the bottom 128 bits of each register are called `xmm0` to `xmm31`.

AVX-512 also introduced new mask registers (`k0` through `k7`), which can be used to mask operations on the vector registers. For example when adding

two vectors, a mask can be used to set certain elements of the result vector to zero instead of the sum.

AVX-512 actually consists of several instruction sets. The most important of these is AVX-512F, the foundation, which every CPU supporting AVX-512 must implement. The AVX-512 instruction sets relevant for this thesis are

- AVX-512BW (Byte and Word Instructions), which adds instructions for 8-bit and 16-bit integer operations,
- AVX512DQ (Doubleword and Quadword Instructions), which adds additional 32-bit and 64-bit instructions,
- AVX-512VL (Vector Length Extensions), which extends most instructions to also operate on `xmm` (128-bit) and `ymm` (256-bit) registers and
- AVX512VBMI2 (Vector Byte Manipulation Instructions 2) which adds additional instructions for loading and storing 8-bit and 16-bit wide elements.

2.1.2 Vector intrinsics

To use SIMD instructions in higher level languages like C, C++ or Rust, for example, one could use the assembly instructions directly with the use of inline assembly. However, this defeats the purpose of higher level languages as it is difficult to write, difficult to maintain, error-prone and might prevent the compiler from optimizing the code as it does not know what the inline assembly code does (DavidWohlferd, 2016).

For this reason, many high level languages provide vector intrinsics. These are pseudo-functions that provide the convenience and type safety of regular functions but are replaced by SIMD instructions by the compiler and thus do not have any calling overhead.

To use Intel vector intrinsics in C or C++, the file `immintrin.h` must be included with `#include <immintrin.h>`. This file provides the data types `__m128`, `__m256`, `__m512`, `__m128d`, `__m256d`, `__m512d`, `__m128i`, `__m256i` and `__m512i`. The numbers in the type names indicate the number of bits the vector can hold, while the suffix indicates the type of the elements contained in that vector. No suffix stands for single-precision floating-point values, the suffix `d` stands for double-precision floating-point values, and the suffix `i` stands for integer values (Intel, 2022, pp. 3-12 - 3-14).

The intrinsic functions that file provides are prefixed with `_mm_`, `_mm256_` or `_mm512_`, indicating that the function operates on a vector of 128, 256 or 512 bits, respectively. Additionally, the functions are suffixed with an

indicator of the type of the vector element the functions operate on. For example, the intrinsic `_mm256_add_ps` which has the signature

```
__m256 _mm256_add_ps (__m256 a, __m256 b)
```

adds two 256 bits wide single-precision floating-point vectors element-wise and returns the resulting vector (Intel, 2022, pp. 3-12 - 3-14).

Additionally, the file `immintrin.h` provides types for the AVX-512 mask registers, which are called `__mmask8`, `__mmask16`, `__mmask32` and `__mmask64`, where the number indicates the number of bits the mask can hold. Functions on these registers are prefixed with `_k` and suffixed with `_mask8`, `_mask16`, `_mask32` or `_mask64`, indicating that the function operates on a mask of 8, 16, 32 or 64 bits, respectively (Intel, b).

2.1.3 SSE/AVX* instructions relevant for this thesis

The following instruction families are relevant for this thesis (Intel, b):

`set1`

Sets all elements of a vector to a given value.
Example intrinsic: `_mm512_set1_ps`

`loadu`

Loads all elements of a vector from memory. In contrast to the `load` instruction, the memory address passed to `loadu` does not have to be aligned to any particular boundary.
Example intrinsic: `_mm512_loadu_ps`

`maskz_loadu`

Loads all elements of a vector from memory, but only those elements where the corresponding bit in a given mask is set to 1. If the bit in the mask is set to 0, the element is set to 0.
Example intrinsic: `_mm512_maskz_loadu_ps`
Figure 2.2 shows an example for the `maskz_loadu` instruction.

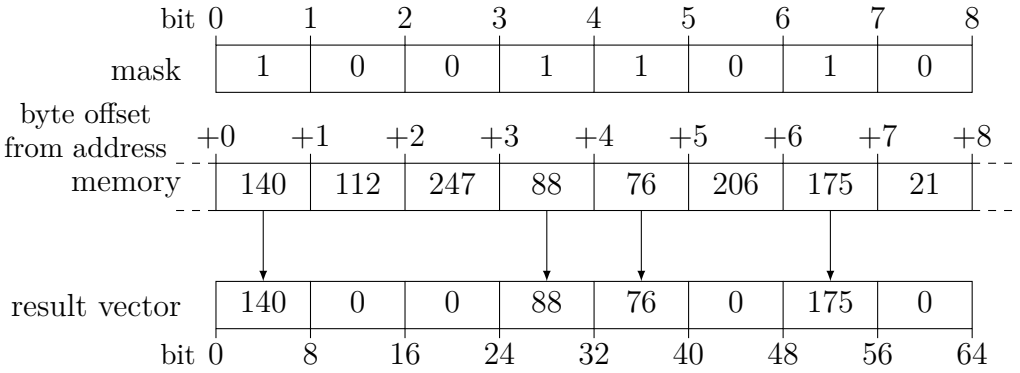


Figure 2.2: Example for the `maskz_loadu` instruction. A vector of 8 8-bit integers is loaded from memory into a vector register using the `maskz_loadu` instruction with an 8-bit mask.

`mask_compressstoreu`

Stores all elements of a vector where the corresponding bit in a given mask is set to 1 contiguously in memory.
Example intrinsic: `_mm512_mask_compressstoreu_ps`
Figure 2.3 shows an example for the `mask_compressstoreu` instruction.

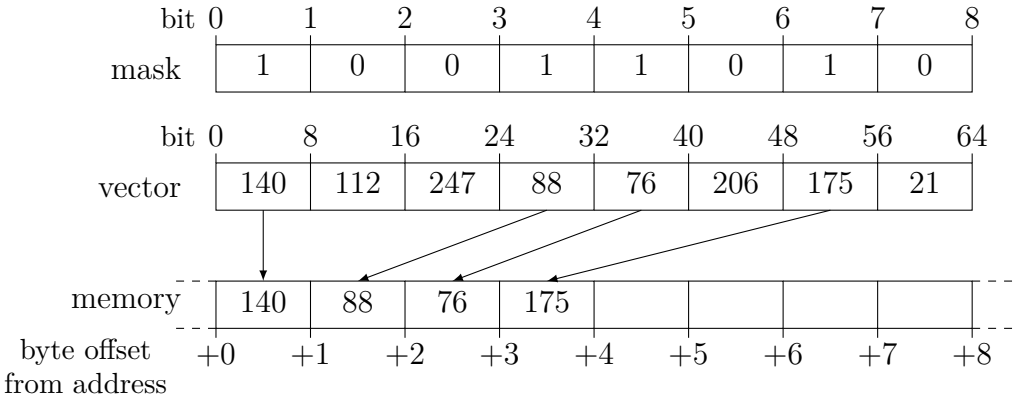


Figure 2.3: Example for the `mask_compressstoreu` instruction. A vector of 8 8-bit integers is stored into memory using the `mask_compressstoreu` instruction with an 8-bit mask.

`test_mask`

Computes the bitwise AND of two vectors, and returns a mask where a bit is set if the corresponding element in the result vector is non-zero.
Example intrinsic: `_mm512_mask_test_epi32_mask`

`knot`

Computes the bitwise NOT of a mask.
Example intrinsic: `_knot_mask32`

`kand`

Computes the bitwise AND of two masks.
Example intrinsic: `_kand_mask32`

`kpopcnt`

Counts the number of set bits in a mask (**pop**ulation **c**ou**nt**).
Strictly speaking, this instruction is not part of the SSE or one of the AVX instruction sets, but it is needed for the algorithm presented in this thesis, so it is included in this list anyway.

3

Related Work

Vectorized sorting algorithms are an extensively researched topic resulting in vast amounts of literature. Thus only some of the most relevant research can be presented in this section.

One of the first sorting algorithms designed for parallel computers was bitonic sort developed by Batcher (1968) which was implemented using SIMD instructions for supercomputers by Siegel (1977). Bitonic sort is a sorting network which is a type of sorting algorithm where the order of comparisons is set in advance that sorts a fixed number of elements (Wikipedia contributors, 2022d).

Zagha and Blelloch (1991) developed a vectorized radix sort algorithm for the CRAY Y-MP supercomputer. They achieve a speedup of 3 to 5 over the vectorized implementation of Quicksort by Levin (1990). The speedup is at least partly due to the fact that the CRAY Y-MP does not have a cache which results in all memory being accessible with equal cost.

More recently, Gueron and Krasnov (2015) implemented a vectorized Quicksort using AVX2, which switches to insertion sort for small subarrays. Their partitioning function uses precomputed permutation masks and requires $\mathcal{O}(n)$ additional memory. They achieved a speedup of 4 compared to STL Sort, the sorting algorithm included in the C++ standard library (Wikipedia contributors, 2022c).

Using AVX-512, Bramas (2017) designed a Quicksort which makes use of AVX-512 `mask_compressstoreu` instructions for the Quicksort partitioning. In contrast to the implementation of Gueron and Krasnov (2015), his partitioning does not require any additional memory or precomputed permutation masks. He achieved this by buffering the outermost vectors in registers

and using the `mask_compressstoreu` instructions. For small subarrays, his implementation switches to a bitonic sort.

Another Quicksort implementation was developed by Blacher et al. (2021) using AVX2. For the Quicksort partitioning they used the vector buffering by Bramas (2017) to avoid the need for additional memory and the precomputed permutation masks by Gueron and Krasnov (2015). For small subarrays, their implementation uses sorting networks. Thiemicke et al. (2021) port this algorithm to AVX-512, which however is slower than the original implementation for large arrays.

Around the same time, Möller (2021) proposed a SIMD implementation of bitwise MSB radix sort using AVX-512 instructions. He used the AVX-512 `mask_compressstoreu` instructions to implement a vectorized in-place bit sorter algorithm, which is used to sort an array of elements according to a specific bit. For small subarrays, his implementation, similar to the implementation by Gueron and Krasnov (2015), switches to insertion sort. His bit sorter algorithm buffers vectors in SIMD registers to make the algorithm in-place, similar to the partitioning function by Bramas (2017).

This thesis builds on the implementation by Möller (2021).

4

MSB Radix sort

The algorithm presented in this thesis uses in-place MSB Radix Sort. MSB Radix Sort will first be described for sorting unsigned integers into ascending order and then be generalized to be able to sort signed integers and floating point numbers as well as being able to sort an array into descending order.

Unlike many other sorting algorithms, MSB Radix Sort does not sort by comparing keys with each other. Instead, it sorts an array by examining the bits of the binary representation of the keys (Knuth, 1998, pp. 122-123).

Algorithm 1 MSB Radix sort

```
 1: procedure MsbRadixSort(array, length)
 2:     RadixRecursion(array, 0, length − 1, number of most significant bit)
 3: procedure RadixRecursion(array, left, right, bitNo)
 4:     if right − left ≤ 0 then return
 5:     split ← BitSort(array, left, right, bitNo)
 6:     if bitNo − 1 ≥ 0 then
 7:         RadixRecursion(array, left, split − 1, bitNo − 1)
 8:         RadixRecursion(array, split, right, bitNo − 1)
 9: procedure BitSort(array, left, right, bitNo)
10:     while true do
11:         i ← index of leftmost element between left and right that has bit
                bitNo set to 1
12:         j ← index of rightmost element between left and right that has
                bit bitNo set to 0
13:         if i ≥ j then return i
14:         swap array[i] and array[j]
```

Algorithm 1 shows the MSB Radix Sort algorithm. It works in the following way:

First, the array is sorted according to the most significant bit of the keys. This results in two subarrays, one containing the keys with the most significant bit set to 0 and one containing the keys with the most significant bit set to 1. The two subarrays are then recursively sorted according to the next most significant bit and so on. The recursion stops when a subarray is empty or all bits were sorted (Knuth, 1998, pp. 122-128).

The bit sorter works by finding the leftmost element with the bit set to 1 and the rightmost element with the bit set to 0, i.e. the first elements from the left and right, which are at the 'wrong' location. Those two elements are swapped so that they are in the 'right' position. This is then repeated until the elements are sorted by the bit (this can be detected by the fact that the leftmost element whose bit is set to 1 is right of the rightmost element whose bit is set to 0). This results in two subarrays, where all of the elements in the first subarray are smaller than all of the elements in the second subarray (Knuth, 1998, pp. 122-128).

4.1 Sorting different data types

So far, the MSB Radix Sort algorithm has only been described for sorting unsigned integers in ascending order. But, one may also want to sort other types of data or sort data in descending order. Luckily, the algorithm can very easily be extended to also sort in descending order and handle other data types, as long as the bits of the data type are ordered by significance, i.e. the most significant bit is the highest bit (bit 31 in case of a 32-bit data type) and the least significant bit is the lowest bit (bit 0). In this thesis only unsigned integers, signed integers and IEEE-754 floating point numbers are considered.

Conveniently, unsigned integers, signed integers and IEEE-754 floating point numbers have the property that the bits are ordered by significance

bit index	$n-1$	$n-2, \ldots, 0$
unsigned integer	up	up
signed integer	down	up
IEEE-754 floating point number	down	up if bit $n-1$ is 0, down otherwise

Table 4.1: Bit sort directions for sorting different n-bit data types in ascending order (Herf, 2001; Möller, 2021)

(Herf, 2001), thus only the direction the individual bits have to be sorted in differs for these three data types. To sort a specific bit in descending instead of ascending order, only the order of the two tests on line 11 and 12 in algorithm 1 needs to be changed.

Table 4.1 shows the direction the bits of different data types have to be sorted. Using this table the MSB Radix Sort algorithm can be extended to sort unsigned integers, signed integers and IEEE-754 floating point numbers.

To sort an array descending instead of ascending, the bit sort directions can simply be inverted. This inverts the sort direction of every bit and thus the sort direction of the whole data type.

4.2 Comparison sorter for small subarrays

For smaller subarrays the cost of the recursive function calls per element increases, since for smaller subarrays less elements are sorted per call. To combat this, the algorithm uses a comparison sorter for subarrays with a size below the threshold `cmpSortThreshold`. The comparison sorting algorithm used is insertion sort, which despite its runtime complexity of $\mathcal{O}(n^2)$ is faster than most other sorting algorithms for small arrays (Tang, 2012).

The optimal value for `cmpSortThreshold` is explored in section 8.1.

5

C++ implementation

The sorting algorithms presented in this thesis are implemented in C++. The language was chosen because of its widespread use and because of its template features, which are used to implement the sorting algorithm in a generic fashion.

Similar to the MSB Radix Sort implementation by Möller (2021), the different versions of MSB Radix Sort implemented for this thesis share the same recursion function (in this thesis called radix recursion).

Listing 1 C++ implementation of the radix recursion

```
template <bool Up, typename BitSorter, typename CmpSorter,
          bool IsRightSide = false, bool IsHighestBit = true,
          typename K, typename... Ps>
void radixRecursion(int bitNo, SortIndex cmpSortThreshold, SortIndex left,
                    SortIndex right, K *keys, Ps *...payloads) {
  if (right - left <= 0) { return; }
  if (right - left < cmpSortThreshold) {
    CmpSorter::template sort<Up, K, Ps...>(left, right, keys, payloads...);
    return;
  }
  SortIndex split = BitSorter::template sortBit<Up, IsHighestBit, IsRightSide,
    K, Ps...>(bitNo, left, right, keys, payloads...);
  if (bitNo > 0) {
    radixRecursion
      <Up, BitSorter, CmpSorter, IsHighestBit ? false : IsRightSide, false>
        (bitNo - 1, cmpSortThreshold, left, split - 1, keys, payloads...);
    radixRecursion
      <Up, BitSorter, CmpSorter, IsHighestBit ? true : IsRightSide, false>
        (bitNo - 1, cmpSortThreshold, split, right, keys, payloads...);
  }
}
```

Listing 1 shows the C++ implementation of the radix recursion as a template function. The implementation as a template function makes the algorithm generic, so that it can be used to sort data sets with arbitrary key and payload data type combinations and also provides a means of configuring different bit sorter and comparison sort algorithms at compile time.

For configuring the bit sorter and comparison sort algorithms, the template parameters `BitSorter` and `CmpSorter` are used. The different algorithms are passed as classes that contain a template function `sort` or `sortBit`, respectively. The `BitSorter` sorts a specific bit and the `CmpSorter` sorts a whole subarray with a comparison based sorting algorithm. The function parameter `cmpSortThreshold` controls the threshold for using the comparison based sorting algorithm, as described in section 4.2.

The next two template parameters `IsRightSide` and `IsHighestBit` are used to keep track whether the algorithm is currently on the root of the sorting tree (parameter `IsHighestBit`) and, if not, on which side of the sorting tree the algorithm is currently working on (parameter `IsRightSide`). These two parameters, together with the parameter `Up` (which controls the sort direction), are used to determine the sort direction of the current bit according to table 4.1 in chapter 4.

The last template parameters `K` and `Ps...` determine the data type of the keys and payloads. Most of the time these can be automatically deduced by the compiler (Vandevoorde et al., 2017).

Listing 2 C++ implementation of the sequential bit sorter

```
1  struct BitSorterSequential {
2    template <bool Up, bool IsHighestBit, bool IsRightSide,
3              typename K, typename... Ps>
4    static INLINE SortIndex sort(int bitNo, SortIndex left, SortIndex right,
5                                 K *keys, Ps *...payloads) {
6      SortIndex l = left; SortIndex r = right;
7      while (l <= r) {
8        while (l <= r && (bitDirUp<K, Up, IsHighestBit, IsRightSide>() !=
9                          isBitSet(bitNo, keys[l])))
10         { l++; }
11       while (l <= r && (!bitDirUp<K, Up, IsHighestBit, IsRightSide>() !=
12                         isBitSet(bitNo, keys[r])))
13         { r--; }
14       if (l < r) {
15         std::swap(keys[l], keys[r]); (std::swap(payloads[l], payloads[r]), ...);
16       }
17     }
18     return l;
19   }
20 };
```

Listing 2 shows the C++ implementation of the sequential bit sorter as described in chapter 4. This class can be passed as the `BitSorter` template

parameter to the radix recursion, to use the sequential bit sorter as the bit sorter algorithm. The template function `bitDirUp` used in this class provides an implementation of table 4.1 for the bit sort direction.

5.1 Template-wrapper for SIMD instructions

As explained in chapter 2, SIMD instructions can be accessed from C++ via vector intrinsics. The SIMD version of MSB Radix Sort could be implemented using those intrinsics. However, vector intrinsics are specific to the vector size and the data type of the vector elements. This makes it impractical to develop a generic algorithm that can be used for arbitrary key and payload data type combinations, since it would have to be implemented for each key and payload data type combination separately.

To overcome this, a template wrapper library can be used, which wraps the vector registers and intrinsics in template classes and functions. This allows the algorithm to be implemented in a generic fashion for arbitrary key and payload data type combinations where the appropriate vector registers and intrinsics are chosen at compile time.

There are such template wrapper libraries available, for example the T-SIMD library developed by Möller (2016). However, the T-SIMD library does not provide support for 64-bit wide data types (`uint64_t`, `int64_t` and `double`) and only provides 16, 32 and 64 byte wide vectors.

But the MSB Radix Sort implementation for this thesis requires vectors with byte widths of 8, 16, 32, 64, 128, 256 and 512 and also for 64-bit wide data types.

Therefore, for this thesis, a separate template wrapper was implemented based on the T-SIMD library. Listing 3 shows the implementation of the SIMD vector class of that library, where `MMRegType<T, Bytes>` is the appropriate vector register type (i.e. one of `__128`, `__m256`, etc.) for the type `T` and number of bytes `Bytes`. Listing 4 shows the implementation of the SIMD mask class, where `MaskType<Size>` is the appropriate mask register type (i.e. one of `__mmask64`, `__mmask32`, etc.). Listing 5 shows is the implementation of the `loadu` instruction of that library as an example.

Since the AVX* instruction sets only provide 16, 32 and 64 byte wide vector registers, the vectors of sizes 8, 128, 256 and 512 are emulated. For the 8 byte wide vectors, a 16 byte vector register is used, with the upper 8 bytes simply being ignored. Operations on vectors of this size use the AVX-512 masked instructions with a mask where the upper 8 bits are set to 0 to emulate operations on the 8 byte wide vectors, as can be seen on lines 15 to 20 in listing 5. To emulate the 128, 256 and 512 byte wide vectors, multiple

64 byte wide vector registers are combined into a single vector, as shown in listing 3. Operations on these emulated vectors are then repeated for each of the 64 byte wide vector registers, emulating the operations on the 128, 256 and 512 byte wide vectors, as can be seen on lines 21 to 32 in listing 5.

Listing 3 C++ implementation of the SIMD vector class

```
1  template <typename T, int Bytes = 64, typename = void> struct Vec;
2
3  template <typename T, int Bytes>
4  struct Vec<T, Bytes, std::enable_if_t<(Bytes <= 64) && is_power_of_two<Bytes>>> {
5    MMRegType<T, Bytes> mmReg;
6    static constexpr int numElems = Bytes / sizeof(T);
7    Vec() = default;
8    Vec(const MMRegType<T, Bytes> x) : mmReg(x) {}
9    Vec &operator=(const MMRegType<T, Bytes> x) { mmReg = x; return *this; }
10   operator MMRegType<T, Bytes>() const { return mmReg; }
11 };
12
13 template <typename T, int Bytes>
14 struct Vec<T, Bytes, std::enable_if_t<(Bytes > 64) && is_power_of_two<Bytes>>> {
15   MMRegType<T, 64> mmReg[Bytes / 64];
16   static constexpr int numElems = Bytes / sizeof(T), numRegs = Bytes / 64;
17   Vec() = default;
18   MMRegType<T, 64> &operator[](int i) { return mmReg[i]; }
19   const MMRegType<T, 64> &operator[](int i) const { return mmReg[i]; }
20 };
```

Listing 4 C++ implementation of the SIMD mask class

```
1  template <int Size> struct Mask {
2    MaskType<Size> k;
3    Mask() = default;
4    Mask(const MaskType<Size> &x) : k(x) {}
5    Mask &operator=(const MaskType<Size> &x) { k = x; return *this; }
6    operator MaskType<Size>() const { return k; }
7  };
```

Listing 5 Generic implementation of `loadu` as a C++ template function (shortened)

```
template <int Bytes = 64, typename T>
static INLINE Vec<T, Bytes> loadu(const T *p) {
  if constexpr (Bytes == 64) {
    if constexpr (std::is_integral_v<T>) { return _mm512_loadu_si512(p); }
    else if constexpr (std::is_same_v<T, float>) { return _mm512_loadu_ps(p); }
    else if constexpr (std::is_same_v<T, double>) { return _mm512_loadu_pd(p); }
  } else if constexpr (Bytes == 32) {
    if constexpr (std::is_integral_v<T>) { return _mm256_loadu_si256((__m256i_u *)p); }
    else if constexpr (std::is_same_v<T, float>) { return _mm256_loadu_ps(p); }
    else if constexpr (std::is_same_v<T, double>) { return _mm256_loadu_pd(p); }
  } else if constexpr (Bytes == 16) {
    if constexpr (std::is_integral_v<T>) { return _mm_loadu_si128((__m128i_u *)p); }
    else if constexpr (std::is_same_v<T, float>) { return _mm_loadu_ps(p); }
    else if constexpr (std::is_same_v<T, double>) { return _mm_loadu_pd(p); }
#ifdef __AVX512VL__
  } else if constexpr (Bytes == 8) {
    if constexpr (std::is_integral_v<T>) {
      if constexpr (sizeof(T) == 1) { return _mm_maskz_loadu_epi8(0xff, p); }
    }
#endif
  } else if constexpr (Bytes == 128 || Bytes == 256 || Bytes == 512) {
    Vec<T, Bytes> result;
    for (int i = 0; i < Vec<T, Bytes>::numRegs; i++) {
      if constexpr (std::is_integral_v<T>)
        { result[i] = _mm512_loadu_si512(p + i * Vec<T, 64>::numElems); }
      else if constexpr (std::is_same_v<T, float>)
        { result[i] = _mm512_loadu_ps(p + i * Vec<T, 64>::numElems); }
      else if constexpr (std::is_same_v<T, double>)
        { result[i] = _mm512_loadu_pd(p + i * Vec<T, 64>::numElems); }
    }
    return result;
  }
}
```

6

SIMD Implementation of MSB Radix Sort

As mentioned in chapter 5, the radix recursion function is shared by the sequential and SIMD implementations, so it is not discussed in detail here. The part that is actually vectorized in the SIMD implementation is the bit sorter.

The algorithm described in this chapter is called 'RadixSIMD' in the following.

6.1 SIMD bit sorter

For the SIMD implementation of the bit sorter, the bit sorter developed by Möller (2021) with a few modifications is used. It is actually quite similar to the SIMD partitioning algorithm developed by Bramas (2017) with the main difference being the following: In the algorithm developed by Möller (2021) and extended for this thesis, the elements to be sorted are not compared to a pivot element, but instead a specific bit of the elements is tested.[1]

The algorithm is first described without considering payloads and then the handling of payloads is discussed.

[1] Testing a specific bit can actually be interpreted as comparing the elements to a 'virtual' pivot element, making MSB Radix Sort quite similar to Quicksort, as noted by Knuth (1998, p. 128).

In the following, `numElemsPerVec` denotes the number of elements contained in one SIMD vector. How the value of `numElemsPerVec` and the actual sizes of SIMD vectors are chosen is explained in chapter 6.4.3.

The C++ implementation of the SIMD bit sorter can be found in appendix A.

Algorithm 2 SIMD bit sorter

```
 1: procedure SIMDBitSort(array, left, right, bitNo)
 2:     if there are at least numElemsPerVec elements then
 3:         vecStore ← load numElemsPerVec elements from the left
 4:     while there are at least numElemsPerVec unread elements do
 5:         vec ← vecStore
 6:         sortMask ← test if bit bitNo is set for each element of vec
 7:         if there are enough elements free on the left then
 8:             vecStore ← load numElemsPerVec elements from the right
 9:         else
10:             vecStore ← load numElemsPerVec elements from the left
11:         store the elements in vec with the corresponding bit not set in sortMask
            to the left of array
12:         store the elements in vec with the corresponding bit set in sortMask
            to the right of array
13:     numElemsRest ← number of remaining elements in array
14:     if numElemsRest ≠ 0 then
15:         vecRest ← load the remaining numElemsRest elements
16:     if there were at least numElemsPerVec elements in the beginning then
17:         sortMask ← test if bit bitNo is set for each element of vecStore
18:         store the elements in vecStore with the corresponding bit not set in
            sortMask to the left of array
19:         store the elements in vecStore with the corresponding bit set in
            sortMask to the right of array
20:     if numElemsRest ≠ 0 then
21:         sortMaskRest ← test bit bitNo for each element of vecRest
22:         store the elements in vecRest with the corresponding bit not set in
            sortMaskRest to the left of array
23:         store the elements in vecRest with the corresponding bit set in
            sortMaskRest to the right of array
24:     return index of first element with bit bitNo set
```

Algorithm 2 shows the pseudocode for the vectorized version of the bit sorter algorithm. The basic idea is the same as in the sequential version shown in chapter 4: searching for elements where the bit to sort is set to 1

from the left and moving them to the right and searching for elements where the bit to sort is set to 0 from the right and moving them to the left (when sorting the bit descending this would be the other way around), essentially finding elements that are on the 'wrong' side and moving them to the 'right' side. However, instead of handling the elements one by one, multiple elements in an entire vector are sorted at once. This is done in the following way:

First, one SIMD vector is preloaded into `vecStore` from the left end of the array using the `loadu` instruction, if there are at least `numElemsPerVec` elements. This frees[2] `numElemsPerVec` elements on the left of the array for sorted elements to be written to. After that, a while loop is executed until there are less elements left for sorting than fit into a SIMD vector.

At the beginning of each while loop iteration there are always `numElemsPerVec` elements free in total, possibly not all on one side.

In the while loop, first the vector in `vecStore` is moved into `vec`, freeing `vecStore`. Then the bit `bitNo` is tested in each element in `vec` using the `test_mask` instruction, resulting in the mask `sortMask`. In the mask `sortMask`, a set bit indicates that the corresponding element should be moved to the right and an unset bit indicates that the corresponding element should be moved to the left. If the bit `bitNo` should be sorted in descending instead of ascending order, this should be the other way around, so `sortMask` would be inverted in that case.

There now are `numElemsPerVec` elements in `vec` to be written back to the array and, as noted before, there are always `numElemsPerVec` elements free at the beginning of a while loop iteration. Despite this, the elements in `vec` can not yet be moved to their correct side. This is because it is not guaranteed how these `numElemsPerVec` free elements are distributed to the left and right sides. Thus, there might not be enough elements free on one of the two sides. However, on at least one side there are enough elements free, since otherwise there would not be `numElemsPerVec` elements free in total.

So, to have enough elements free on both sides, first the number of set bits in the mask `sortMask` is counted using the `kpopcnt` instruction and compared to the number of free elements on the left and right to determine on which side there are not enough elements free. Then, from that side, another vector of `numElemsPerVec` elements is loaded into `vecStore` using the `loadu` instruction (if there are enough elements free on both sides, the vector is still loaded, but the side from which it is loaded does not matter).

This frees another `numElemsPerVec` elements making $2 \cdot$ `numElemsPerVec` elements free in total.

[2] Note that a 'free' element does not mean that there is no element, but simply represents the fact that the element can be overwritten without losing any information.

Since now there are enough elements free on both sides, the elements in `vec` can be stored to the left and right, according to the mask `sortMask`. This is done with the `mask_compressstoreu` instructions introduced in chapter 2. Specifically, one `mask_compressstoreu` instruction is used with the mask `sortMask`, storing the elements with the corresponding bit set in `sortMask` contiguously to the right and another `mask_compressstoreu` instruction with the inverted mask `sortMask`, storing the elements with the corresponding bit not set in `sortMask` contiguously to the left.

Now, there are `numElemsPerVec` elements free in total again and the next while loop iteration is started.

After the while loop finished executing, there might still be some elements left to sort in the array because they were not enough to fill an entire SIMD vector. Let `numElemsRest` be the number of these elements (then `numElemsRest` $<$ `numElemsPerVec`). Additionally, if there were at least `numElemsPerVec` elements in the array at the beginning of the while loop, there are `numElemsPerVec` elements in `vecStore` left to sort, too.

To sort these remaining elements, first the `numElemsRest` elements that are left in the array are loaded into `vecRest`. This is done using the `maskz_loadu` instruction with a mask where only the lower `numElemsRest` bits are set. This frees `numElemsRest` elements in the array resulting in `numElemsPerVec` + `numElemsRest` consecutive free elements in the middle of the array.

Then, if there were at least `numElemsPerVec` elements in the array at the beginning of the while loop, the elements in `vecStore` are stored back into the array exactly the same way as in the while loop, except that there are no additional elements loaded from one side, since the `numElemsPerVec` + `numElemsRest` free elements are consecutive and thus there are enough elements free on both sides.

Finally, the remaining `numElemsRest` elements in `vecRest` are stored back into the array, again exactly the same way as in the while loop, except that there are no additional elements loaded from one side.

6.2 Example for the SIMD bit sorter

To aid with the understanding of the SIMD bit sorter algorithm, this section provides an example.

In the example, an array of length 15 is sorted according to the `bitNo`-th bit of the key. Instead of whole elements, only the `bitNo`-th bit of the key of each element is used to represent the whole element. Free elements are represented by '/'. Note that, as above, 'free' does not mean that there is no

element, but simply represents the fact that the element can be overwritten without losing any information. For the example, `numElemsPerVec` is 4.

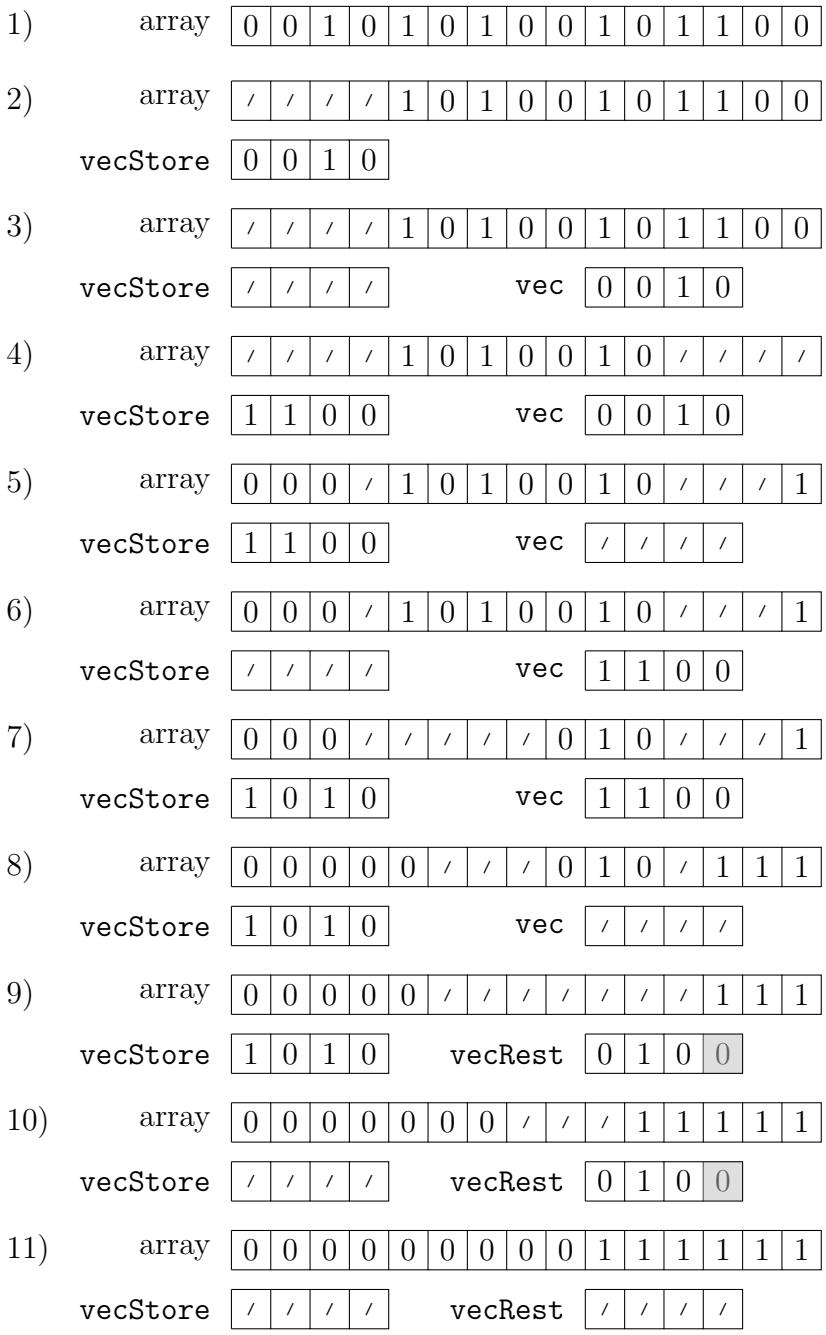


Figure 6.1: Example of the SIMD bit sorter

Figure 6.1 shows the example of the SIMD bit sorter. The steps are as follows:

1) The initial state. The array is not sorted according to bit `bitNo` yet.

2) A vector is preloaded from the left into `vecStore`.

3) Beginning of first while loop iteration: the vector in `vecStore` is moved to `vec`.

4) 3 elements in `vec` should go to the right, but there are no elements free on the right, so a vector is loaded into `vecStore` from the right.

5) The elements in `vec` are stored to the left and right, according to bit `bitNo` using the `mask_compressstoreu` instruction.

6) Beginning of second while loop iteration: the vector in `vecStore` is moved to `vec`.

7) 2 elements in `vec` should go to the left, but there is only one element free on the left, so a vector is loaded into `vecStore` from the left.

8) The elements in `vec` are stored to the left and right, according to bit `bitNo` using the `mask_compressstoreu` instruction.

9) There are now `numElemsRest` $= 3 < 4 =$ `numElemsPerVec` remaining elements in the array left to read, so the while loop is exited and the remaining elements are loaded into `vecRest` using the `maskz_loadu` instruction. The greyed out 0-elements in `vecRest` represent the elements that were set to 0 by the `maskz_loadu` instruction.

10) The elements in `vecStore` are stored to the left and right, according to bit `bitNo` using the `mask_compressstoreu` instruction.

11) The elements in `vecRest` are stored to the left and right, according to bit `bitNo` using the `mask_compressstoreu` instruction.
The array is now sorted according to bit `bitNo`.

6.3 Difference to previous implementation

The implementation of the bit sorter presented in this thesis is different than the implementation by Möller (2021). If the number of elements in the subarray is an integer multiple of `numElemsPerVec`, the two implementations

are similar. However if it is not, the two implementations handle the remaining elements differently.

The implementation by Möller (2021) leaves the remaining elements on the right of the subarray and sorts the part that is an integer multiple of `numElemsPerVec` using SIMD instructions. It then sorts the remaining elements into the 'middle' of the already sorted part by using a modified sequential bit sorter. So the SIMD bit sorter by Möller (2021) essentially 'sweeps' the array from the left and almost the right into the middle and then accesses the right end of the array again, which might cause a cache miss and thus suboptimal performance. The implementation developed in this thesis instead leaves the remaining elements in the 'middle' and then uses masked instructions to sort them. It essentially 'sweeps' the array from the left and right into the middle and then only accesses the middle to sort the remaining elements, which prevents a potential cache miss and thus might increase performance. Additionally, using masked SIMD instructions instead of a sequential bit sorter probably also increases performance.

6.4 Handling of Payloads

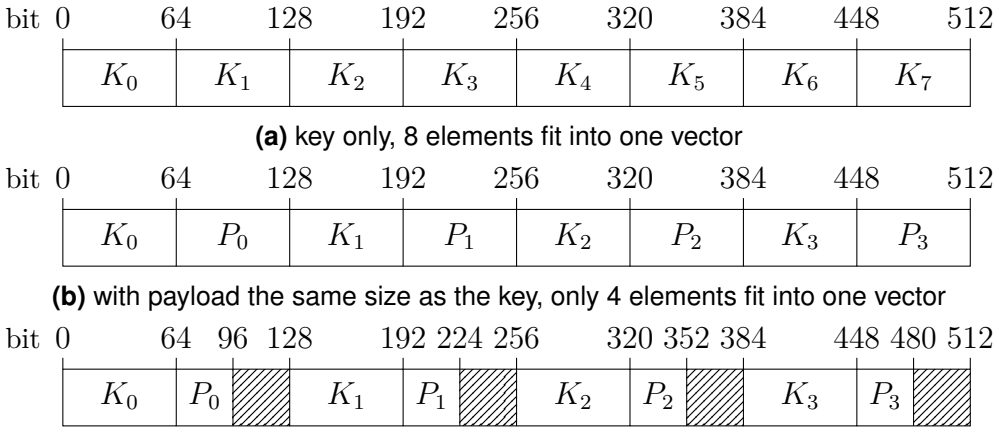
So far, the sorting has only been discussed without considering payloads. Therefore, so far the SIMD implementation of MSB Radix Sort can only sort datasets which do not contain payloads. But one may also want to sort a dataset where a single element consists of a key and one or more payloads rather than just one key.

6.4.1 Previous Method

The implementation by Möller (2021) handles payloads by combining key and payload into a single element, where the key occupies the low bytes and the payload occupies the high bytes. This results in a single array where keys and payloads are interleaved, also called array of structs (AoS) (Intel, 2012). The sorting is then done by loading the keys and payloads into a vector register and only considering the bits of the element where the key is located, i.e. the low bits.

This however means that, in contrast to key-only sorting, only half of one SIMD vector instead of a full vector is filled with keys, as can be seen in figures 6.2a and 6.2b. This reduces the number of elements per vector, which reduces the speedup gained by the SIMD implementation compared to key-only sorting significantly (Möller, 2021).

Additionally, the implementation by Möller (2021) only supports at most one payload, which also has to have the same size as the key. His implementation can also be used to sort a dataset where the payload is smaller than the key by padding the payload to the size of the key. However this wastes parts of the vector registers used for sorting as shown in figure 6.2c.



(a) key only, 8 elements fit into one vector

(b) with payload the same size as the key, only 4 elements fit into one vector

(c) with payload half the size of the key, only 4 elements fit into one vector and there is unused space (striped area)

Figure 6.2: Usage of a 512-bit AVX-512 vector register when sorting a 64-bit key (represented with K_i) with different payloads (represented with P_i) using the algorithm by Möller (2021).

To remove these limitations and improve performance when sorting datasets with payloads, a new method is proposed.

6.4.2 Handling different numbers of payloads

To be able to always sort the same number of elements in one vector regardless of the number of payloads, the proposed method uses separate arrays instead of arranging the data in one interleaved array. Specifically, one array is used for the keys and an additional array is used for each payload. This data structure is called SoA (Structure of Arrays) (Intel, 2012).

The sorting is then done on the key array exactly as in the key-only case except that every load and store operation that is performed on the key array is mirrored for every payload array. For the `mask_compressstoreu` instruction, the same sort masks that were obtained from the keys are used. So the key and every payload are loaded into separate vector registers, resulting in a full SIMD vector being filled with keys as shown in figure 6.3. This way, the same number of elements can be sorted at the same time as in the key-only case, independent of the number of payloads.

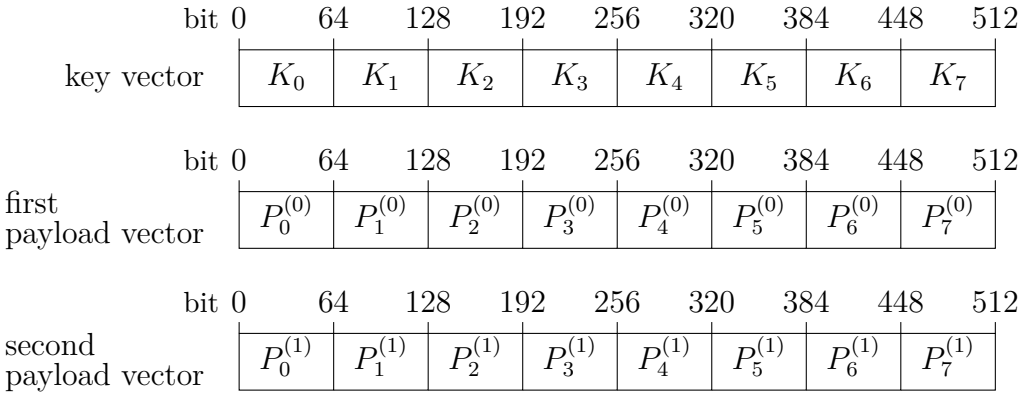


Figure 6.3: Usage of 512-bit AVX-512 vector registers when sorting a 64-bit key (represented with K_i) with two 64-bit payloads (represented with $P_i^{(0)}$ and $P_i^{(1)}$) using the proposed method.

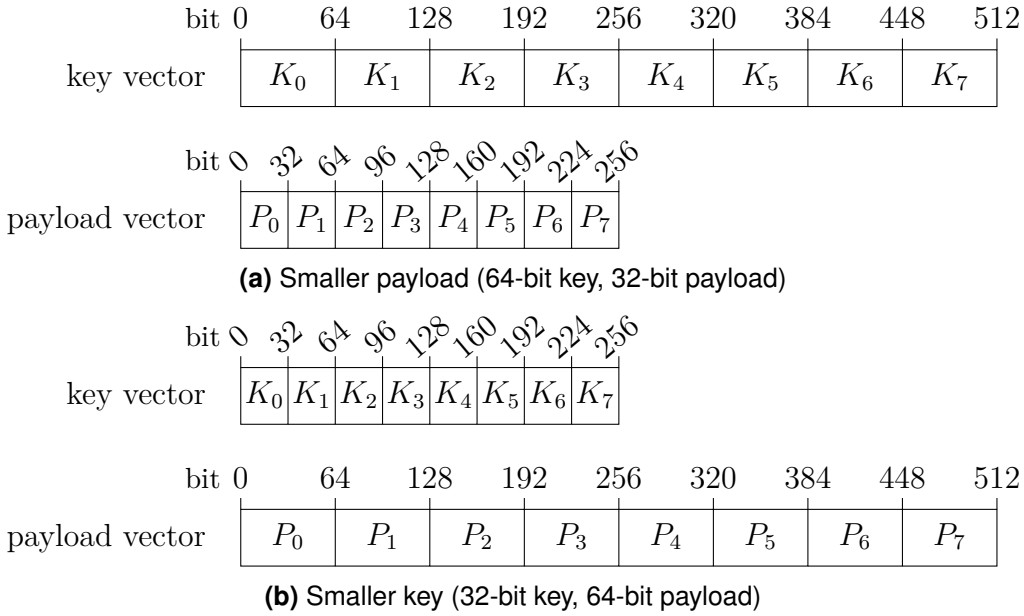
Thus, the only slowdown when sorting a dataset with more or larger payloads is due to the cost of the additional load and store operations for the payloads, instead of also increasing the element-size and thereby reducing the number of elements that fit into a single SIMD vector.

This is the same way the sorting algorithm by Bramas (2017) handles payloads. However, his algorithm only supports one `int32` payload and only if the key also has the type `int32` as well.

6.4.3 Handling different sized key and payloads and choosing `numElemsPerVec`

So far, only the sorting of datasets where all payloads have the same size as the key was discussed. However, one might also want to sort datasets where some or all payloads have a size that differs from the size of the key. One simple way to achieve this would be to pad all smaller types to the size of the largest type. However, this wastes space both in memory and in the vector registers as seen in section 6.4.1, where a smaller payload was padded to the size of the key to be sorted by the algorithm by Möller (2021).

To avoid wasted space, instead of padding the data type, one could simply use smaller vector registers as shown in figure 6.4. When all payloads are smaller than or as big as the key, this is exactly what is done in the algorithm developed for this thesis. Here, the vectors with sizes below 512 bits that were mentioned in section 5.1 are necessary. The smallest vector with size 64 bits (or 8 bytes) that had to be emulated is necessary when sorting a dataset with 64-bit keys and 8-bit payloads.



(a) Smaller payload (64-bit key, 32-bit payload)

(b) Smaller key (32-bit key, 64-bit payload)

Figure 6.4: Use of smaller vector registers to avoid wasted space.

However, when there is a payload larger than the key, one might notice that the vector register used for the keys is no longer the largest available one, as can be seen in figure 6.4b.

To avoid this, a second option is proposed: The second method always uses the largest available vector register for the keys. However, that means that the payloads that are larger than the key are loaded into a vector register which is larger than the largest available vector register. For this, the emulated vector registers of sizes 128, 256 and 512 bytes (1024, 2048 and 4096 bits) introduced in section 5.1 are used. By emulating larger vector registers, more keys can be loaded into a single vector register, as shown in figure 6.5, and thus more elements can be sorted in parallel. However, since the larger vector registers are emulated, multiple SIMD instructions need to be executed to execute a single operation, which may negate the performance gained by using the largest available (non-emulated) vector register for the keys.

In the following, the version of RadixSIMD that uses the first option (that does not use the larger emulated vector registers) is called 'RadixSIMDOneReg'. The version that uses the second option (that uses the larger emulated vector registers) is simply called 'RadixSIMD'.

Which one of these two possible options for handling datasets with larger payloads than keys is faster is explored in chapter 8.

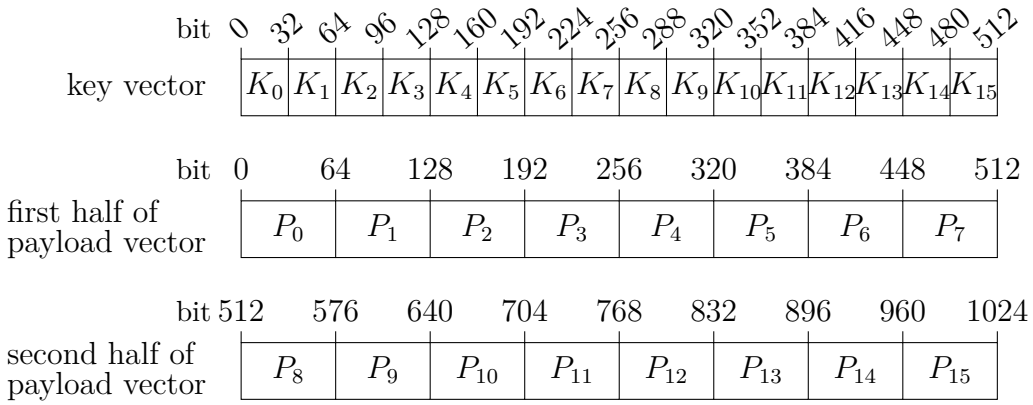


Figure 6.5: Use of emulated, larger vector register for payloads to be able to use a full 512-bit AVX-512 vector register for the keys, when sorting a 32-bit key with a 64-bit payload. In this case, a 1024-bit vector register is emulated using two 512-bit vector registers.

To make the determination of the vector sizes easier, the parameter `numElemsPerVec` is used. This parameter determines the number of elements in each vector. The sizes of the vectors are then calculated by multiplying this parameter with the size of the data type the vector is supposed to hold.

Listing 6 shows the C++ implementation of the parameter `numElemsPerVec` as part of the `BitSorterSIMD` class.

Listing 6 Implementation of the parameter `numElemsPerVec` as a variable template as part of the `BitSorterSIMD` class.

```
1 template <bool OneReg = false> struct BitSorterSIMD {
2
3   template <typename K, typename... Ps>
4   static constexpr SortIndex
5       numElemsPerVec = OneReg ? 64 / std::max({sizeof(K), sizeof(Ps)...})
6                               : 64 / sizeof(K);
7
8   //...
9 };
```

For the first of the above mentioned options for choosing the vector register sizes, which is used by the algorithm RadixSIMDOneReg, the parameter `numElemsPerVec` is set to 64 divided by the size of the largest data type in bytes. This leads to the largest available (non-emulated) vector register being used for the largest data type and smaller vector registers for smaller data types.

For the second of the options mentioned above, which is used by the algorithm RadixSIMD, the parameter `numElemsPerVec` is set to 64 divided by the size of the key data type. This leads to the largest available (non-emulated) vector register being used for the key data type and different sized, potentially emulated, vector registers for payloads of a different size.

6.5 Comparison sorter for small subarrays

As mentioned in section 4.2, the MSB Radix Sort implementation developed for this thesis switches to a comparison sorter for small subarrays. The threshold `cmpSortThreshold` controls the size below which the comparison sorter is used.

For the algorithms RadixSIMD and RadixSIMDOneReg, insertion sort is used as the comparison sorter.

There are two further comparison sorters examined.

One of them is the small sort used by the SIMD Quicksort implementation by Bramas (2017), which is a bitonic sort. It can sort arrays that fit into 16 512-bit vectors, meaning arrays with a size of up to `16*64/sizeof(K)` elements, where `sizeof(K)` is the size of the data type used for the keys in bytes. The algorithm using this small sort is called 'RadixSIMDBramSmall' in the following.

The other comparison sorter is not actually a sorting algorithm, as it simply does nothing. So the algorithm using this comparison sorter, which is called 'RadixSIMDNoCmp' in the following, does not actually fully sort the array. Instead, it is used to provide an upper bound for performance improvements that can be gained by just optimizing the comparison sorter.

Additionally, even though RadixSIMDNoCmp does not fully sort the array, it still sorts the array partly, meaning if an array was sorted with the algorithm RadixSIMDNoCmp, the elements in the array are guaranteed to be at most `cmpSortThreshold` places away from the position they would be in if the array was sorted. There could be applications where this is enough and a fully sorted array is not actually needed. The parameter `cmpSortThreshold` would then be used to control the 'degree of sortedness', where a larger value decreases computation time but also decreases the 'degree of sortedness'.

What value of `cmpSortThreshold` is optimal is explored in section 8.1.

7

Experiments

To evaluate the performance of the SIMD-Implementation of MSB Radix Sort developed for this thesis, the runtime of this algorithm as well as other algorithms for comparison was measured.

In order to develop a thorough understanding of the performance characteristics of the algorithm in comparison to the other algorithms, the experiments were conducted with different input sizes, different combinations of key and payload types as well as different distributions of the input data.

7.1 Key and Payload Data Types

The following data types are used for testing:

- `int8`: 8-bit signed integer
- `int16`: 16-bit signed integer
- `int32`: 32-bit signed integer
- `int64`: 64-bit signed integer
- `float`: 32-bit floating point number
- `double`: 64-bit floating point number

These types are then used to form different combinations of key and payload data types with different numbers of payloads. These are written in the form '`keyType-payloadType1-payloadType2-...`'. For example, the key payload data type combination where the key is a `float` and the payloads are an `int32` and an `int64` is written as `float-int32-int64`.

Note that unsigned integer types are not tested. This is because when sorting, they only differ in the direction the highest bit (the sign) is sorted and thus almost certainly do not provide additional insight.

7.2 Tested sorting algorithms

The following algorithms were analyzed:

- **Radix:** The MSB Radix Sort implementation developed for this thesis. There are multiple versions analyzed:
  - **RadixSeq:** The sequential version of the algorithm.
  - **RadixSIMD:** The SIMD version of the algorithm, using the second option introduced in section 6.4.3 (always using a 512-bit AVX-512 vector register for the keys).
  - **RadixSIMDOneReg:** The SIMD version of the algorithm, using the first option introduced in section 6.4.3 (using a smaller vector register for key/payload data types that are smaller than the largest key/payload data type).
  - **RadixSIMDBramSmall:** RadixSIMD, but using the small sort algorithm developed by Bramas (2017), which uses bitonic sort instead of insertion sort.
  - **RadixSIMDNoCmp:** RadixSIMD, but with a pseudo comparison sorter which does nothing. This version does not actually completely sort the data, but is used to provide an upper bound for the performance improvements that could be achieved by just optimizing the comparison sorter.
- **Moeller:**
  - **MoellerSeq:** The sequential implementation of MSB Radix Sort developed by Möller (2021).
  - **MoellerCompress:** The SIMD implementation of MSB Radix Sort using AVX-512 `mask_compressstoreu` instruction developed by Möller (2021).
- **STLSort:** The sorting algorithm included in the C++ standard template library (STL) (Wikipedia contributors, 2022c).
- **IPPRadix:** The Radix Sort implementation included in the Intel Performance Primitives library (Intel, a).

- **BramasSort:** The AVX-512 Quicksort implementation developed by Bramas (2017).
- **BlacherSort:** The AVX2 Quicksort implementation developed by Blacher et al. (2021).

Note that some of the sorting algorithms analyzed do not support all key and payload data type combinations. MoellerSeq and MoellerCompress only support up to one payload which has to have the same size as the key. IPPRadix for some reason does not support any payload data type and the key data type cannot be `int8` (however all other key data types are supported, including `uint8`). BramasSort only supports the combinations `int32`, `double` and `int32-int32`. BlacherSort only supports the combination `int32`. RadixSIMDBramSmall supports the same combinations as BramasSort, so it supports only the combinations `int32`, `double` and `int32-int32`.

7.3 Input data

The input data for each test is randomly generated. The keys are generated using different distributions while the payloads were randomly generated using the corresponding key as a seed to make checking the payloads later more easy.

The distributions used for generating the keys are:

- **Uniform:** A uniform distribution. For integer key data types a uniform distribution over the whole range of the integer is used. For floating point key data types a uniform distribution in $[-1, 1[$ was used.
- **Gaussian:** A gaussian distribution with mean $\mu = 0$. For floating point data types a standard deviation of $\sigma = 1$ was used and for integer data types a standard deviation of $\sigma = 100$ was used. For integer data types the distribution is generated from a gaussian distribution of `double`s and then rounded to the nearest integer.
- **Zero:** All keys of the input data are zero.
- **ZeroOne:** All keys of the input data are either zero or one, with equal probability. This provides a distribution with only two different values but with high entropy.
- **Sorted:** An already in ascending order sorted dataset. This is generated by first generating a Uniform distribution for integers and a Gaussian distribution for floating point numbers and then sorting the data.

- **ReverseSorted:** The same as the Sorted distribution but in reverse order.
- **AlmostSorted:** The same as the Sorted distribution, but with $\lfloor 2^{\log_{10}(n)} \rfloor$ pairs of elements swapped, where n is the number of elements.
- **AlmostReverseSorted:** The same as the ReverseSorted distribution, but with $\lfloor 2^{\log_{10}(n)} \rfloor$ pairs of elements swapped, where n is the number of elements.

7.4 Test environment

The tests are run on a machine with an Intel© Core™ i7-12700K CPU running at 5 GHz and 32 GB of main memory. The operating system used for the tests is Manjaro Linux with kernel version 5.17.15. The code was compiled with gcc version 12.1.0.

The processor provides the following AVX-512 instruction set extensions: AVX-512F, AVX-512DQ, AVX-512BW, AVX-512IFMA, AVX-512CD, AVX-512BW, AVX-512VL, AVX-512BF16, AVX-512VBMI, AVX-512VBMI2, AVX-512VNNI, AVX-512BITALG, AVX-512VPOPCNTDQ, AVX-512VP2INTERSECT and AVX-512FP16.

Note that Intel does not officially support AVX-512 on Alder Lake processors (which includes the Intel© Core™ i7-12700K processor) and fused off the AVX-512 circuitry entirely a few months after the processor family was released (Alcorn, 2022). AVX-512 had to be enabled for the Intel© Core™ i7-12700K processor in the BIOS to run the tests for this thesis. This should, however, not have an effect on the results of the tests.

7.5 Time measurements

The sorting time of each algorithm is measured by a call to `clock_gettime` with the `CLOCK_PROCESS_CPUTIME_ID` clock id right before and after the call to the sorting function. Since some of the sorting algorithms that support payloads require the data in Array of Structures (AoS) layout, and some require the data in Structure of Arrays (SoA) layout, the data is converted to the required layout before the sorting is performed. This conversion is not included in the time measurement.

To ensure reliable results even for small datasets, the sorting is performed multiple times and the average time is taken. The number of runs is

$\max\left(1, \lfloor\frac{2^{22}}{n}\rfloor\right)$ where n is the number of elements in the input data. Additionally, to avoid potential warmup effects, warmup runs are performed before the actual measurement. The number of warmup runs is $\max\left(1, \lfloor\frac{2^{18}}{n}\rfloor\right)$ where n is the number of elements in the input data.

For each run, a new input data set is generated in a new memory location to avoid cache effects.

7.6 Checks

After each sorting run there are checks performed confirming that the data is actually sorted. It is not checked whether the elements that are in the array before the sorting are the same as the elements that are in the array after the sorting. The payloads are not checked either. This is because these additional checks have a runtime complexity of at least $\mathcal{O}(n^2)$ and thus would increase the time needed to perform the time measurements significantly.

However, there was a dedicated test program written for the algorithms developed for this thesis (RadixSeq, RadixSIMD, RadixSIMDOneReg and RadixSIMDBramSmall), which tests many key payload data type combinations for all the input data distributions with different dataset sizes. It checks that the data is sorted, that each payload is associated with the correct key, that there is no key missing and that there is no new key introduced by the sorting algorithm. The number of duplicate keys is not checked, meaning that the sorting algorithm could hypothetically overwrite one instance of a duplicate key with another key without the checks noticing this. However if the sorting algorithm is actually incorrect, this is very unlikely to happen only to duplicate keys and thus the checks would almost certainly catch the error in the algorithm.

8

Results

8.1 Determining the optimal comparison sorter threshold

The MSB Radix Sort algorithm developed in this thesis as well as the implementation by Möller (2021) switch to a comparison sorter when the number of elements to be sorted is below the threshold `cmpSortThreshold`, as described in section 4.2 and 6.5.

It is very difficult to determine the threshold by just analyzing the code of the algorithm because it depends on many non-obvious factors. So the threshold is determined experimentally by running the algorithm with multiple different thresholds and multiple different array sizes and then comparing the results. This is done for different key and payload data types.

The optimal comparison sorter threshold is only determined for the distributions Uniform and Gaussian.

However, only the data for the distribution Uniform is shown, since for floating point key types, the runtimes for Uniform and Gaussian distributions are very similar and did not seem to affect the optimal threshold at all and for integer key types, the threshold did not seem to affect the runtime when sorting with Gaussian distributions.

Additionally, only the data where the array size was $2^{18} = 262144$ is shown, since the size of the array did not seem to have an impact on which threshold was optimal either.

For the RadixSIMDOneReg algorithm, only key payload data type combinations were tested, where RadixSIMDOneReg actually differs from RadixSIMD,

i.e. only where there is a payload data type that is larger than the key data type.

Also, the RadixSIMDBramSmall algorithm is only tested for the key payload data type combinations that are supported, which are `int32`, `double` and `int32-int32`.

Figures 8.1, 8.2, 8.3, 8.4, 8.5 and 8.6 show the runtime of MoellerSeq, RadixSeq, MoellerCompress, RadixSIMD, RadixSIMDOneReg and RadixSIMDBramSmall respectively, for different thresholds and key payload data type combinations.

For all tested algorithms there is a clear optimum visible.

For the two sequential algorithms MoellerSeq and RadixSeq, a threshold of $2^6 = 64$ seems to be a reasonable choice and for the algorithms MoellerCompress, RadixSIMD and RadixSIMDOneReg, a threshold of $2^4 = 16$ seems to be acceptable.

For the algorithm RadixSIMDBramSmall, the performance seems to be better the larger the threshold is. As mentioned in section 6.5, the small sort algorithm by Bramas (2017) supports arrays with a size of up to `16*64/sizeof(K)` elements, where `sizeof(K)` is the size of the data type used for the keys in bytes. For the key data type `int32` this is $16 * 64/4 = 256 = 2^8$ elements. So, for the algorithm RadixSIMDBramSmall, the threshold is chosen as large as possible while not exceeding the above mentioned limit, i.e. it is set to `16*64/sizeof(K)`. This however means that the sorting tree of RadixSIMDBramSmall is much shallower than the sorting trees of the other algorithms, possibly affecting the comparability.

The above mentioned values for `cmpSortThreshold` are used in the following tests. For the algorithm RadixSIMDNoCmp the same value as for RadixSIMD is used.

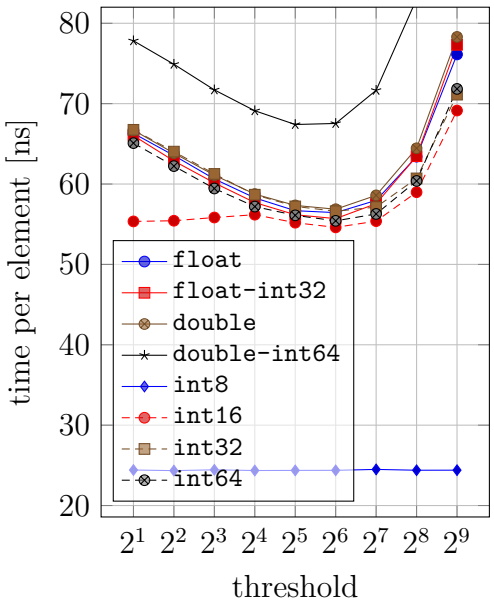


Figure 8.1: Runtime of MoellerSeq for different values for `cmpSortThreshold`, 2^{18} elements, Uniform distribution

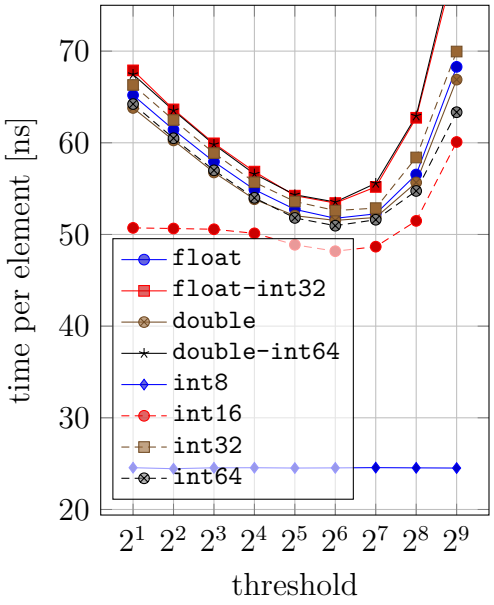


Figure 8.2: Runtime of RadixSeq for different values for `cmpSortThreshold`, 2^{18} elements, Uniform distribution

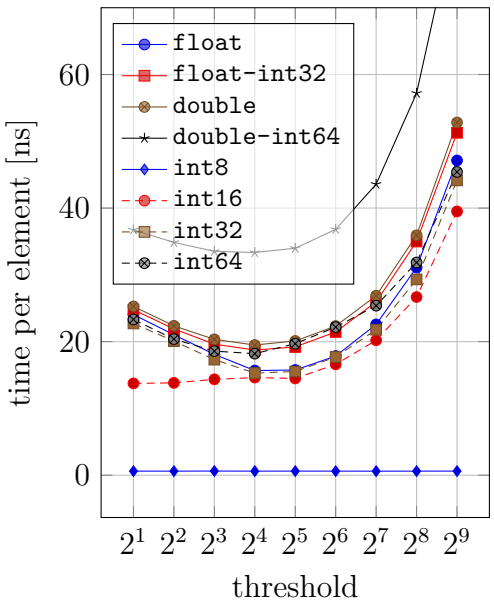


Figure 8.3: Runtime of MoellerCompress for different values for `cmpSortThreshold`, 2^{18} elements, Uniform distribution

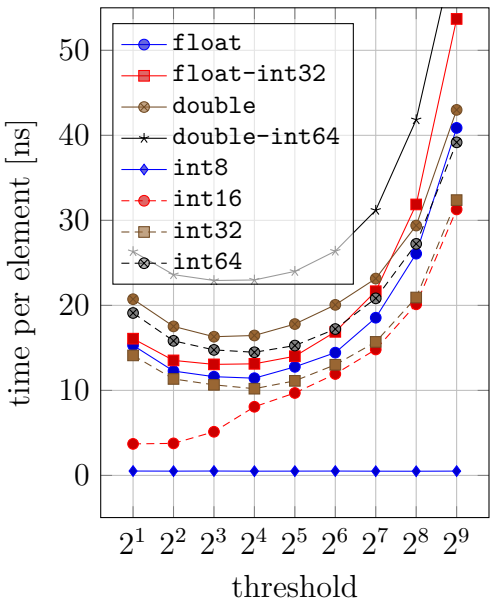


Figure 8.4: Runtime of RadixSIMD for different values for `cmpSortThreshold`, 2^{18} elements, Uniform distribution

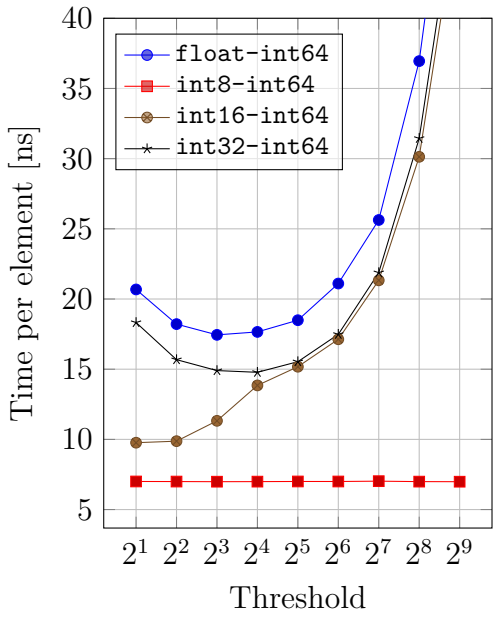


Figure 8.5: Runtime of RadixSIMD-OneReg for different values for `cmpSortThreshold`, 2^{18} elements, Uniform distribution

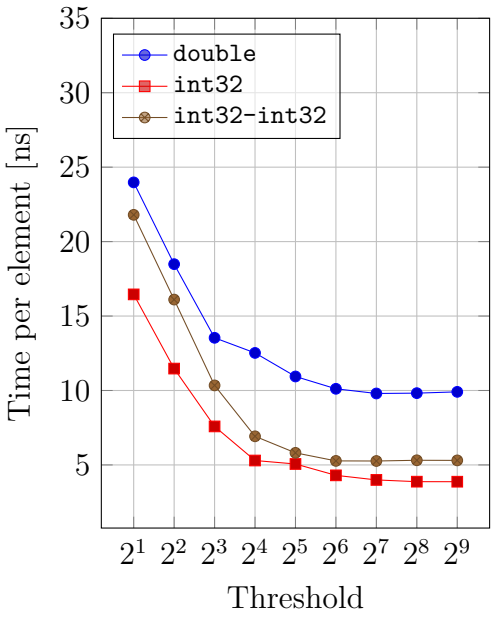


Figure 8.6: Runtime of RadixSIMD-BramSmall for different values for `cmpSortThreshold`, 2^{18} elements, Uniform distribution

8.2 Comparison between RadixSIMD and MoellerCompress

One of the main goals of this thesis was a more efficient payload handling by separating key and payload datastreams. To evaluate whether this goal was achieved, RadixSIMD is compared to MoellerCompress.

However, there are also other differences between RadixSIMD and MoellerCompress besides the way payloads are handled and thus any performance difference between the two algorithms may not be due to the difference in payload handling. Therefore the algorithms are compared without payloads and with payloads where the comparison without payloads provides a baseline and the comparison with payloads can then be used to determine the performance impact of the different payload handling.

The comparison is done for the key data types `float`, `double`, `int8`, `int16`, `int32` and `int64`. For each key data type, the comparison is done for four different key payload data type combinations: without any payload, with payload of the same as the size of the key data type, with payload with half the size of the key data type and with payload with a quarter of the size

of the key data type (in this section, these combinations are called ‘without payload’, ‘with payload’, ‘with half payload’ and ‘with quarter payload’, respectively). For example for a `float` key data type, the comparison is done for the combinations `float` (‘without payload’), `float-int32` (‘with payload’), `float-int16` (‘with half payload’) and `float-int8` (‘with quarter payload’). For tests of MoellerCompress where the payload is smaller than the key, the payload is padded to the size of the key data type. Also, only combinations where a required data type for the payload exists are tested. For example when the key data type is `int8`, there does not exist a data type of half or quarter the size of the key data type.

The comparisons are done for all distributions introduced in section 7.3.

The number of elements is $2^{18} = 262144$ for all tests in this section.

Gaussian distribution

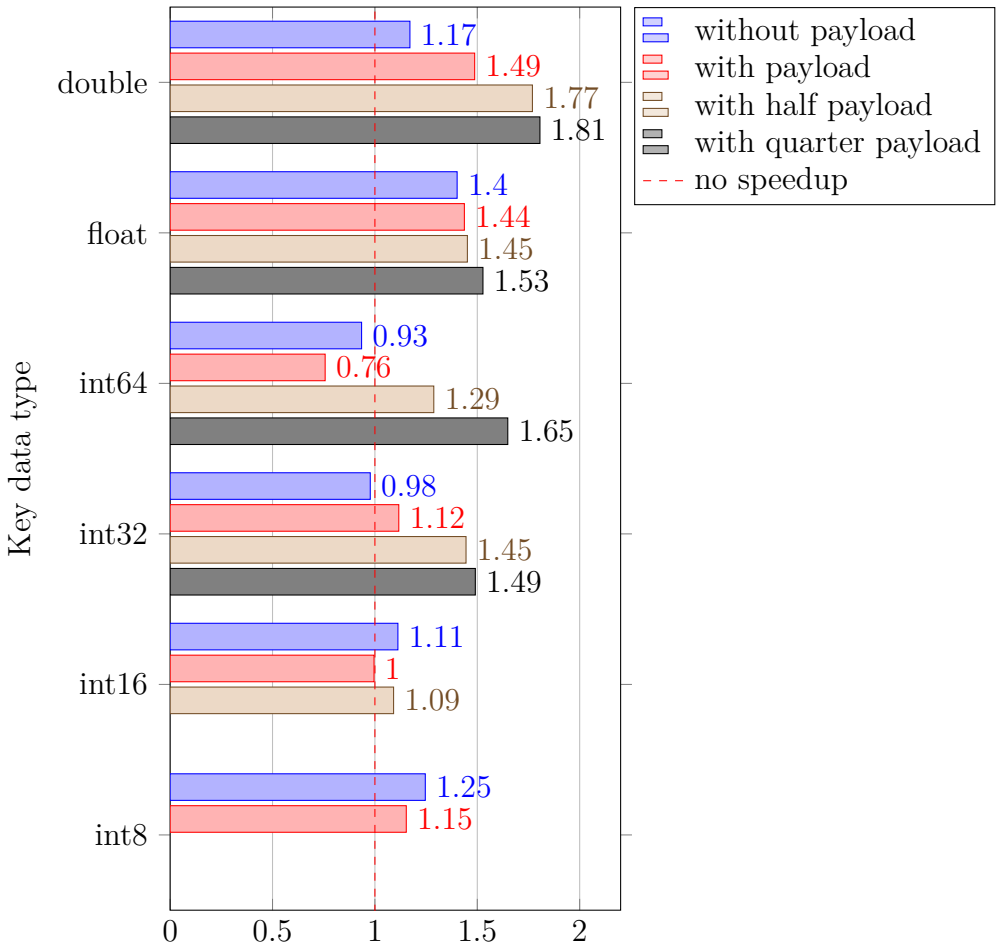


Figure 8.7: Speedup of RadixSIMD over MoellerCompress, Gaussian distribution

Figure 8.7 shows the speedup of RadixSIMD over MoellerCompress with the Gaussian distribution. For almost all key payload data type combinations, RadixSIMD is faster than MoellerCompress up to a speedup of about 1.8. The speedup is greater with smaller payloads and most of the time it is greater for combinations with payload than for combinations without payload.

The exceptions are the key data type `int64`, where the speedup is slightly less than 1 without payload and less than 1 when the payload has the same size as the key and `int16` and `int8`, where the speedup for the different combinations is very similar.

Uniform distribution

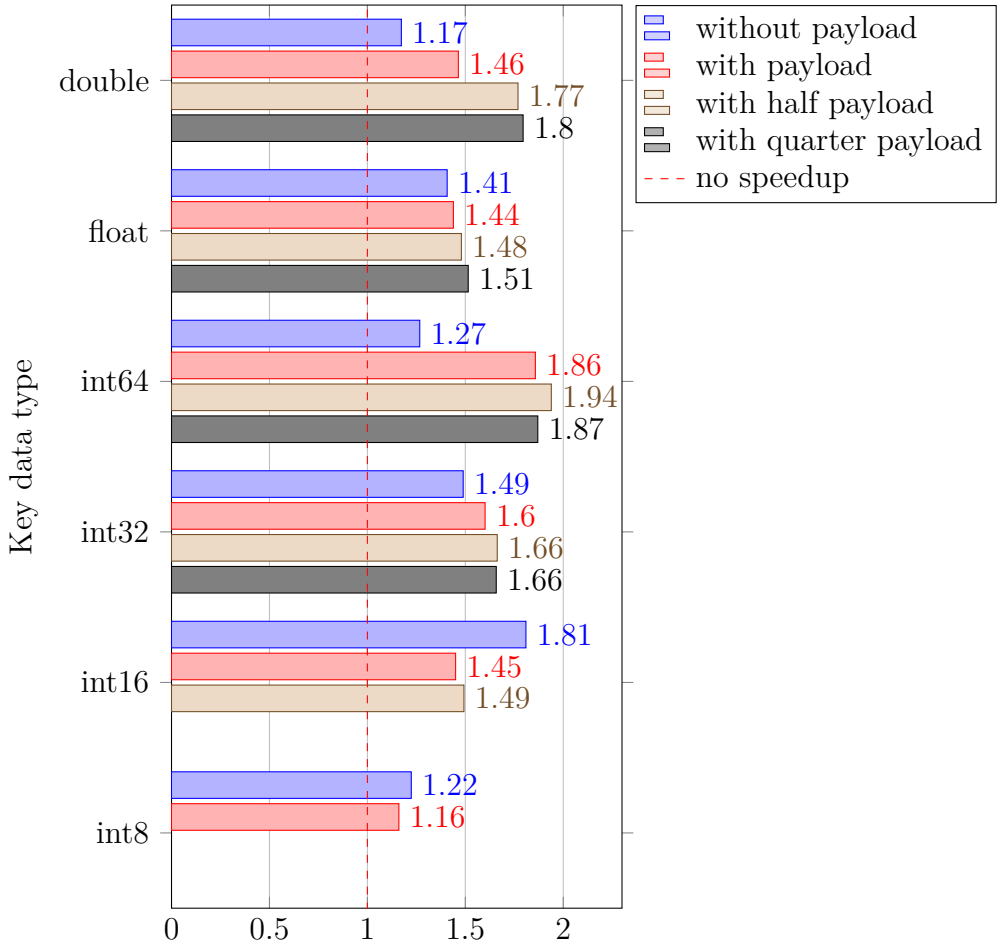


Figure 8.8: Speedup of RadixSIMD over MoellerCompress, Uniform distribution

Figure 8.8 shows the speedup of RadixSIMD over MoellerCompress with the Uniform distribution. For this distribution, RadixSIMD is faster than MoellerCompress for all key data types up to a speedup of 1.94. Similar to the Gaussian distribution, for datasets with a payload, the speedup of RadixSIMD over MoellerCompress is greater than without a payload for all key data types except for `int16` and `int8`. Also there is a slight increase in speedup as the payload gets smaller, however it is not as pronounced as for the Gaussian distribution. For example for the key data types `int64` and `int32`, the speedup is approximately equal for all tested payload sizes.

Zero distribution

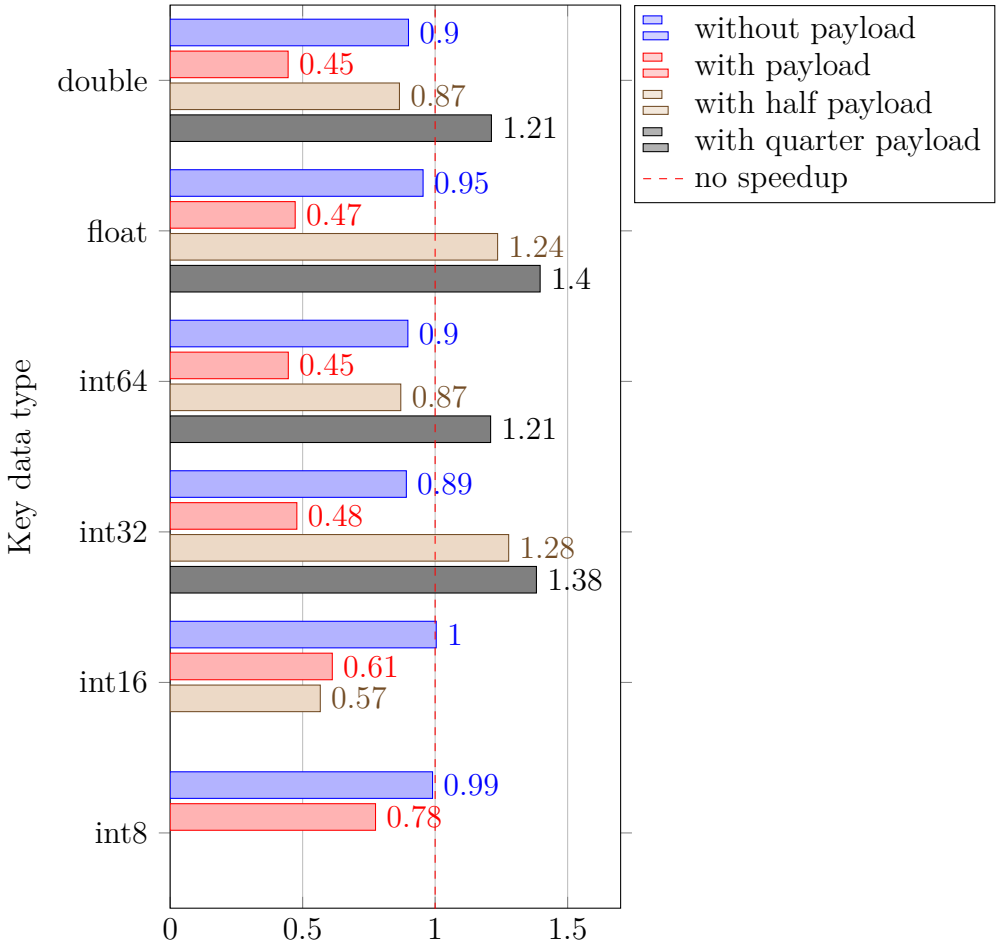


Figure 8.9: Speedup of RadixSIMD over MoellerCompress, Zero distribution

Figure 8.9 shows the speedup of RadixSIMD over MoellerCompress with the Zero distribution. Here, RadixSIMD is almost exactly as fast as MoellerCompress for all key data types without payload. With payload however, RadixSIMD is only half as fast as MoellerCompress for the key data types `double`, `float`, `int64`, and `int32`. The speedup is higher for the key data types `int16` and `int8`, but still below 1. With half payload, the speedup is sometimes above 1 and sometimes below 1. With quarter payload, the speedup is always between 1.2 and 1.4.

There currently is no explanation for this behavior.

ZeroOne distribution

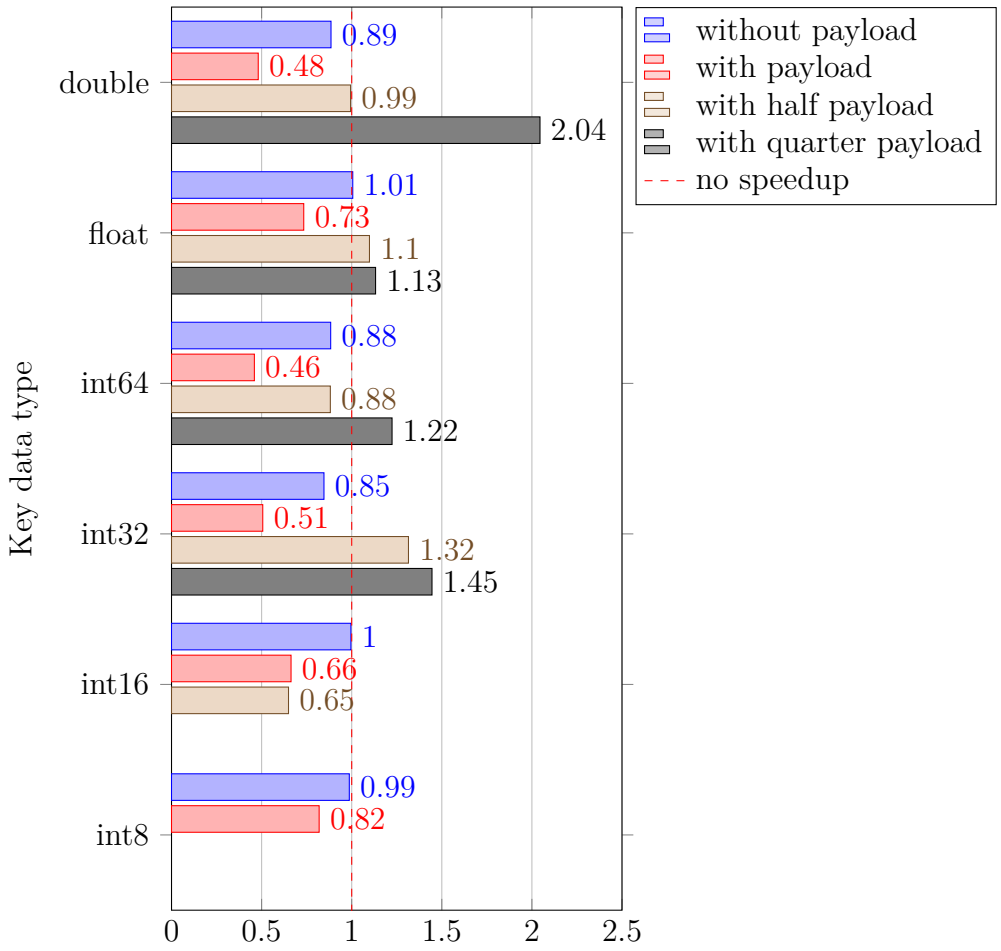


Figure 8.10: Speedup of RadixSIMD over MoellerCompress, ZeroOne distribution

Figure 8.10 shows the speedup of RadixSIMD over MoellerCompress with the ZeroOne distribution. The behavior is very similar to the Zero distribution, with the biggest difference being that the speedup for the key data types `double` with quarter payload is 2 instead of 1.2. As with the Zero distribution, this behavior can currently not be explained.

Sorted distribution

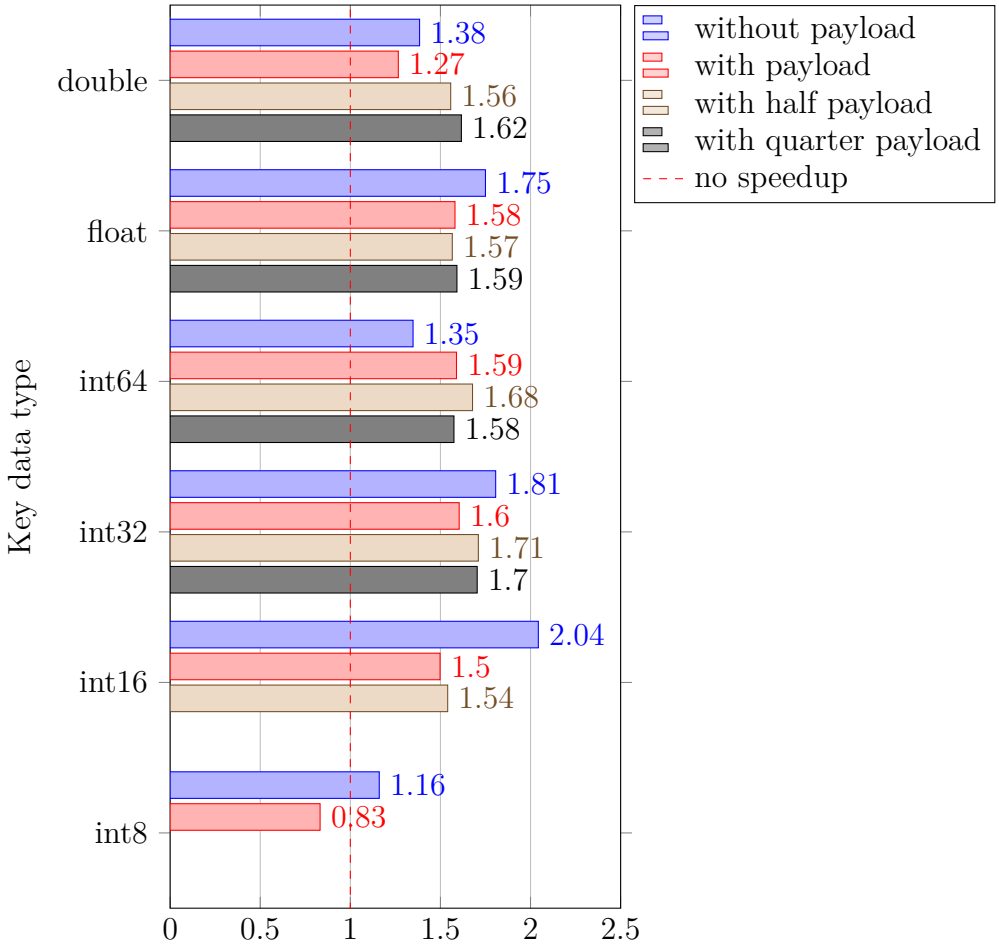


Figure 8.11: Speedup of RadixSIMD over MoellerCompress, Sorted distribution

Figure 8.11 shows the speedup of RadixSIMD over MoellerCompress with the Sorted distribution. In this case, RadixSIMD is faster than MoellerCompress for all key data types except `int8`, where the speedup is less than 1 with payload. There is no clear pattern as to how the speedup depends on the size or existence of a payload.

ReverseSorted distribution

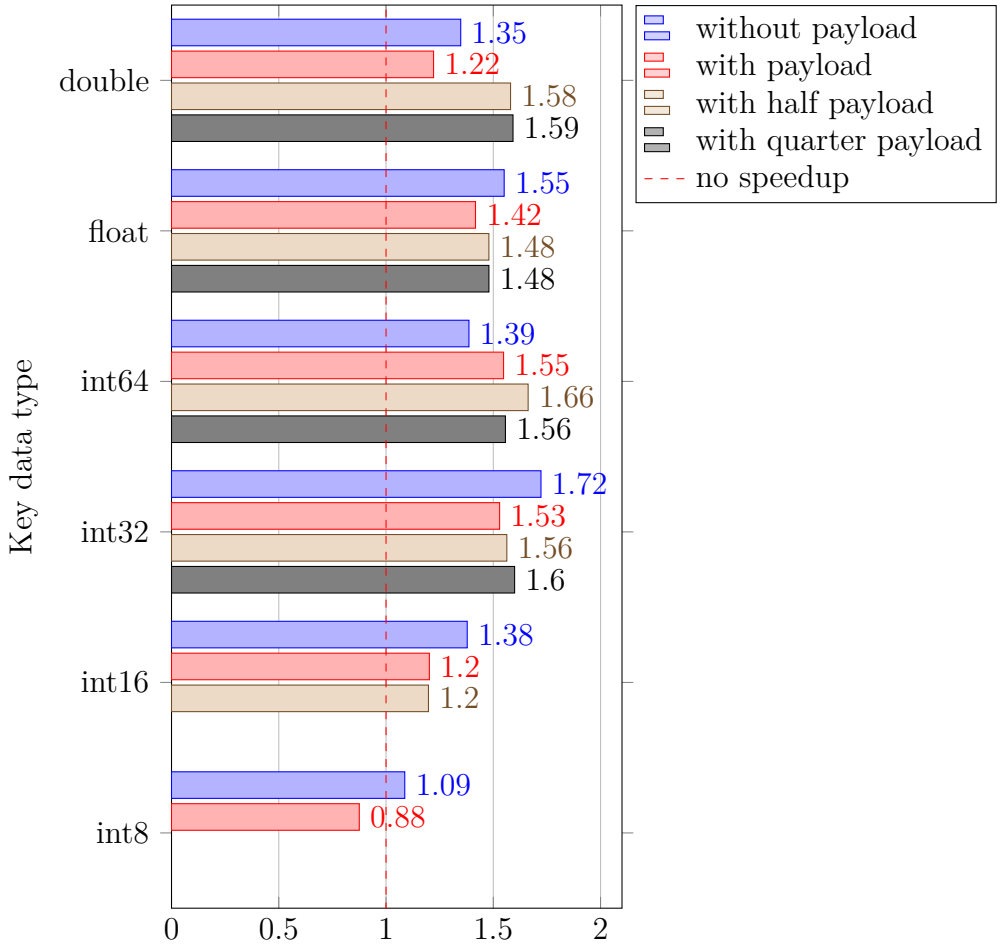


Figure 8.12: Speedup of RadixSIMD over MoellerCompress, ReverseSorted distribution

Figure 8.12 shows the speedup of RadixSIMD over MoellerCompress with the ReverseSorted distribution. The behavior is very similar to the Sorted distribution. RadixSIMD is faster than MoellerCompress for all key data types with and without payload except for `int8`, with no pattern visible. Why the fact that the data is reversed compared to the Sorted distribution does not have an impact on the speedup is not clear.

AlmostSorted distribution

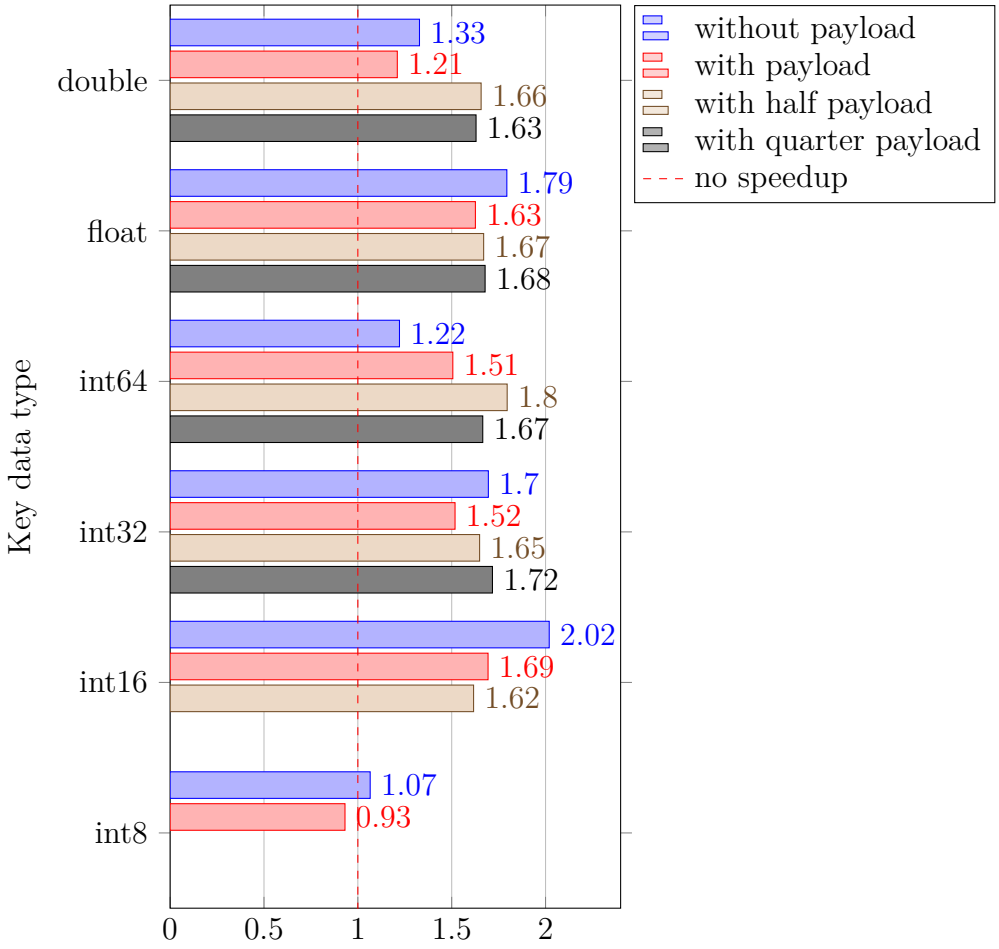


Figure 8.13: Speedup of RadixSIMD over MoellerCompress, AlmostSorted distribution

Figure 8.13 shows the speedup of RadixSIMD over MoellerCompress with the AlmostSorted distribution. The graph for this distribution is almost identical to the graph for the Sorted and ReverseSorted distribution.

This is probably because to generate the AlmostSorted distribution, the Sorted distribution is used, as explained in section 7.3. Why the fact that the data from the AlmostSorted distribution is almost sorted instead of completely sorted does not have an impact on the speedup, is not clear however.

AlmostReverseSorted distribution

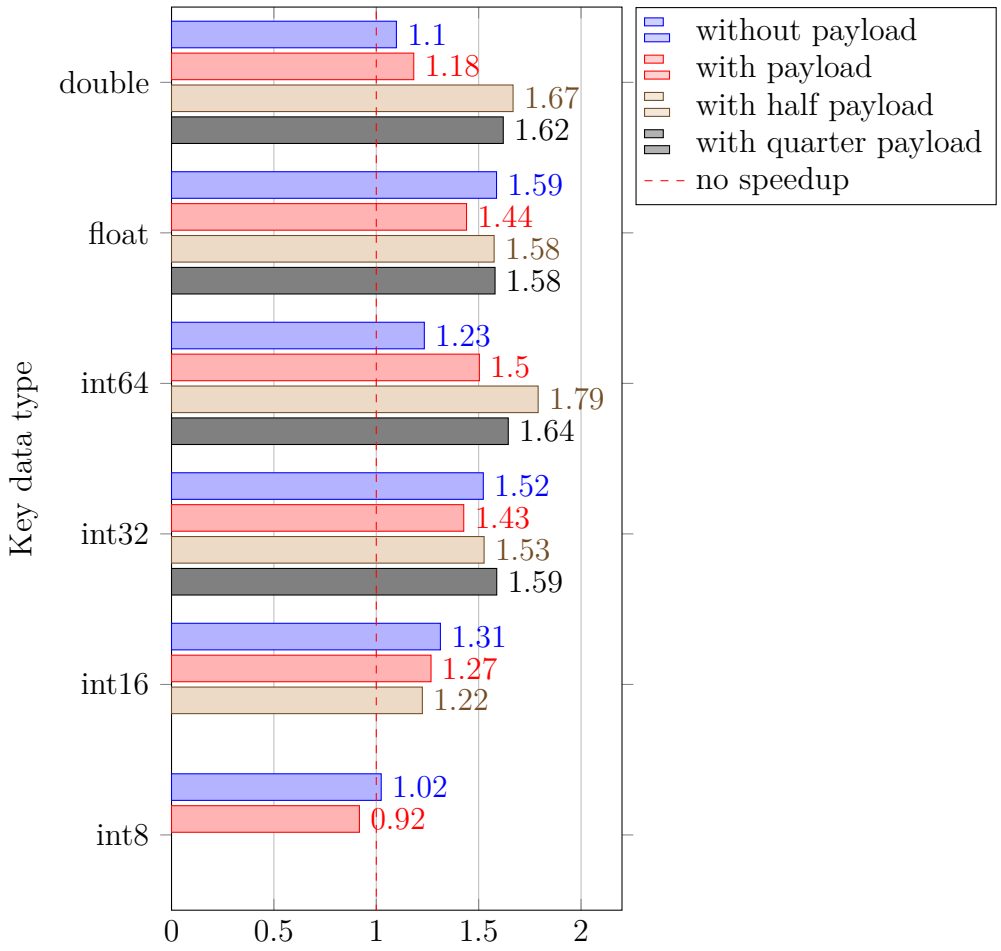


Figure 8.14: Speedup of RadixSIMD over MoellerCompress, AlmostReverseSorted distribution

Figure 8.14 shows the speedup of RadixSIMD over MoellerCompress with the AlmostReverseSorted distribution. Similar to the AlmostSorted distribution, the graph for this distribution is almost identical to the graph for the Sorted and ReverseSorted distribution. The reason for this is not clear in this case either.

8.3 Comparison between RadixSIMDOneReg and RadixSIMD

For handling key payload data type combinations where there is a payload data type larger than the key data type, there are two methods proposed in chapter 6.4.3. One is used by RadixSIMD and the other is used by RadixSIMDOneReg.

Since the results for different distributions are very similar, only the results for the Uniform distribution are shown in figure 8.15.

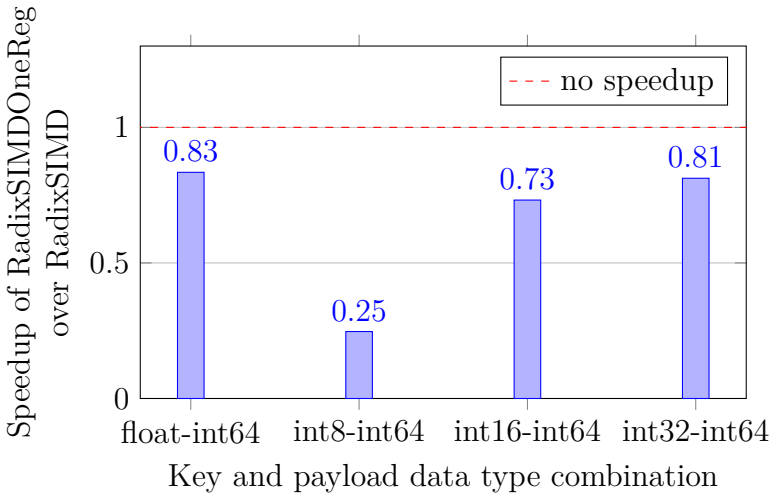


Figure 8.15: Speedup of RadixSIMDOneReg over RadixSIMD, 2^{18} elements, Uniform distribution

RadixSIMDOneReg is slower than RadixSIMD for all key data types. Also, RadixSIMDOneReg gets slower in comparison to RadixSIMD as the key data type size decreases. The reason for this is probably that RadixSIMD fills the largest available SIMD register completely with keys and thus sorts more elements simultaneously as the size of the key data type decreases whereas RadixSIMDOneReg always only sorts in this case 8 elements at a time.

So the emulation of larger SIMD vectors described in section 5.1 does provide a performance benefit compared to avoiding the emulation and sorting less elements at a time.

8.4 Comparison with other sorting algorithms

For the comparison with other sorting algorithms, all algorithms were measured sorting a dataset with 2^{18} elements and the time per element is shown. This was done for all distributions and several key payload data type combinations.

Only some of the comparisons are shown in this section, the comparison graphs for all distributions and all tested key payload data type combinations can be found in appendix B.

Note that, as mentioned in section 7.2, not all tested algorithms support all key payload data type combinations and are therefore not shown in all of the graphs.

The bars in the following figures (8.16-8.20) are colored to make the graphs easier to read. The bars for the sequential algorithms are colored black, RadixSIMD is colored red and RadixSIMDBramSmall is colored brown. The bars for the rest of the algorithms are colored blue. The bar for RadixSIMDNoCmp is dashed as a reminder that it does not actually fully sort the data.

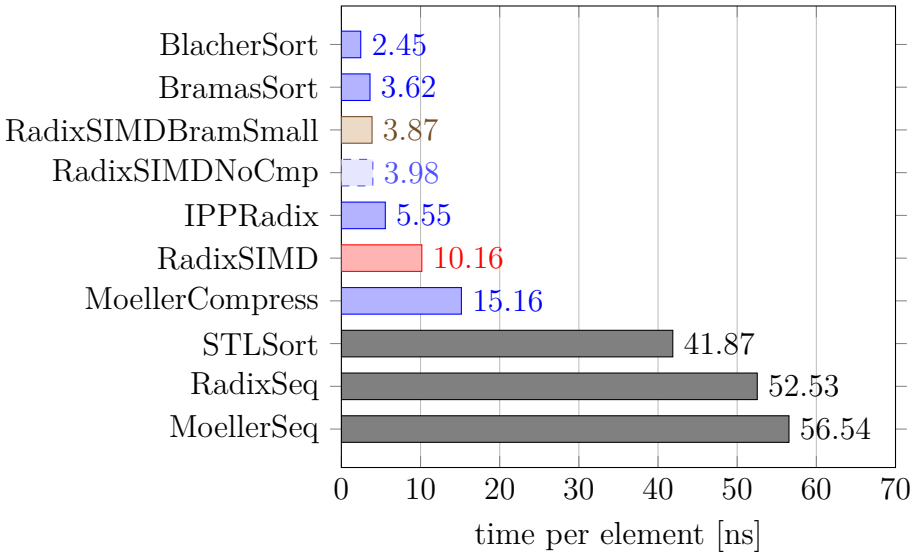


Figure 8.16: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution Uniform

Figure 8.16 shows the time per element for the Uniform distribution and the key payload data type combination `int32` for different sorting algorithms. The sequential sorting algorithms MoellerSeq, RadixSeq and STLSort are

the slowest with approximately 42 to 57 nanoseconds per element. MoellerCompress is significantly faster than the sequential algorithms, the runtime is approximately 15 nanoseconds per element. RadixSIMD is faster than MoellerCompress, with about two thirds of the runtime, but slower than IPPRadix, BramasSort and BlacherSort. BlacherSort is the fastest of the tested sorting algorithms, with a runtime of 2.45 nanoseconds per element. The algorithms RadixSIMDBramSmall and RadixSIMDNoCmp are with 3.87 and 3.98 nanoseconds per element only slightly slower than BramasSort. This indicates that the RadixSIMD algorithm spends most of its time in the comparison sorter and can thus be accelerated significantly by improving the comparison sorter.

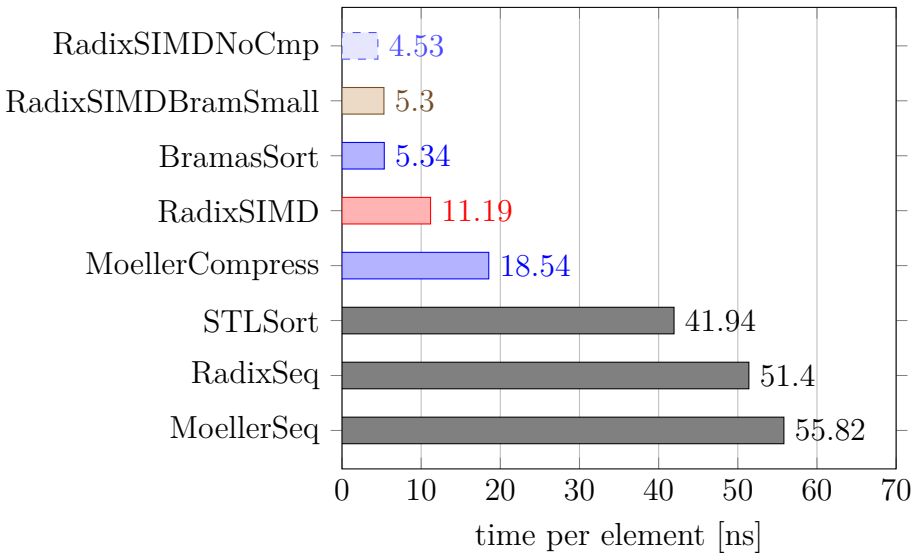


Figure 8.17: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32-int32` and distribution Uniform

Figure 8.17 shows the time per element for the Uniform distribution and the key payload data type combination `int32-int32` for different sorting algorithms. The results are very similar to the ones for `int32`. The runtimes for the sequential algorithms are within 3% of the ones for `int32`. MoellerCompress is again significantly faster than the sequential algorithms and RadixSIMD is almost 40% faster than MoellerCompress. BramasSort and RadixSIMDBramSmall again have an almost equal runtime of approximately 5.3 nanoseconds per element. In this case BramasSort and RadixSIMDBramSmall are the fastest of the tested sorting algorithms, which might only be due to the fact that BlacherSort only supports the `int32` combination. RadixSIMDNoCmp is slightly faster than RadixSIMDBramSmall and Bra-

masSort. This again indicates that the RadixSIMD algorithm spends most of its time in the comparison sorter and can be accelerated significantly by improving the comparison sorter.

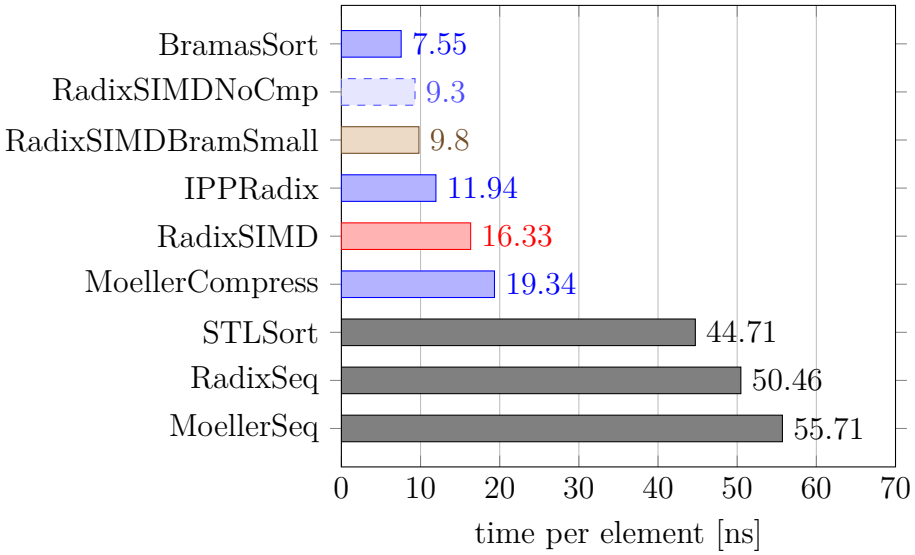


Figure 8.18: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution Gaussian

Figure 8.18 shows the time per element for the Gaussian distribution and the key payload data type combination `double` for different sorting algorithms. The results are similar to the ones for `int32` and `int32-int32` for the Uniform distribution. The sequential algorithms MoellerSeq, RadixSeq and STLSort are again the slowest. MoellerCompress is significantly faster than the sequential algorithms and RadixSIMD is faster than MoellerCompress but does not beat IPPRadix and BramasSort.

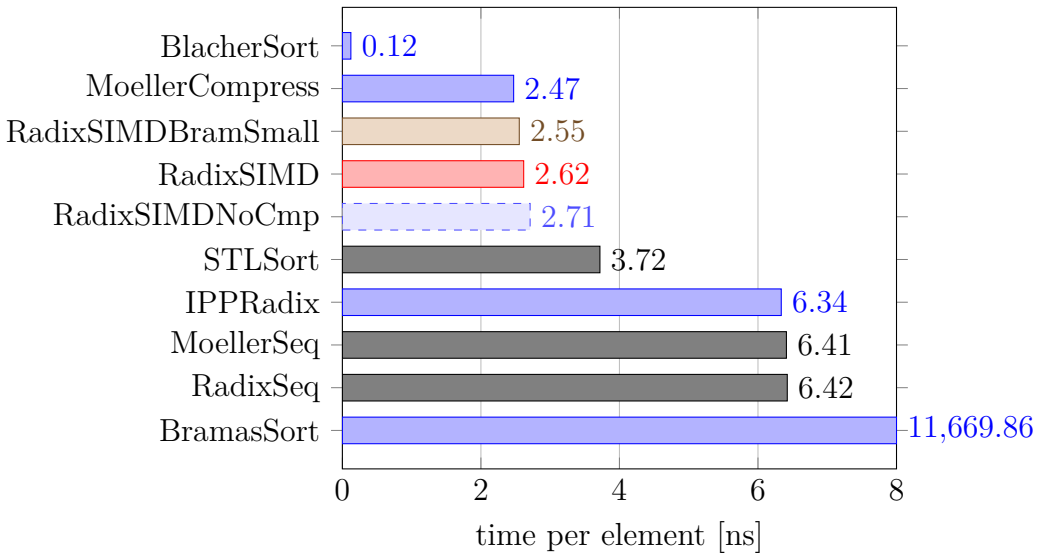


Figure 8.19: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution Zero

Figure 8.19 shows the time per element for the Zero distribution and the key payload data type combination `int32` for different sorting algorithms. BlacherSort is the fastest of the tested sorting algorithms, in this case it is more than an order of magnitude faster than the next fastest sorting algorithm. This is probably because BlacherSort keeps track of the smallest and largest value while partitioning a subarray enabling it to terminate early when the smallest and largest value are equal and thus the subarray does not need to be sorted further.

The next fastest algorithms for this distribution are RadixSIMD, RadixSIMD-BramSmall, RadixSIMDNoCmp and MoellerCompress, which are all almost exactly equally fast. STLSort is a bit slower and IPPRadix, MoellerSeq and RadixSeq are almost twice as slow.

Interestingly, IPPRadix is almost exactly as fast as RadixSeq and MoellerSeq.

Very noticeable in this graph is the runtime of BramasSort, which is several orders of magnitude slower than the other algorithms. This is because BramasSort uses Quicksort without any early termination, which for constant input leads to a 'fully unbalanced' sorting tree (i.e. every non-leaf node has exactly one child), leading to a very deep recursion depth and thus a very slow runtime.

For this data type combination, the graph for the ZeroOne distribution is very similar.

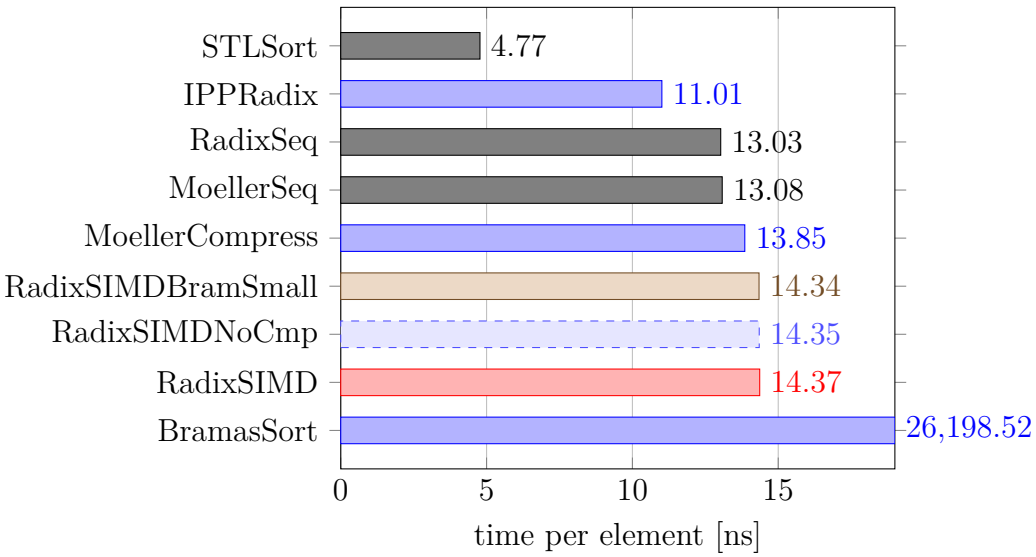


Figure 8.20: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution Zero

Figure 8.20 shows the time per element for the Zero distribution and the key payload data type combination `double` for different sorting algorithms. One might expect the results in this case to be similar to the ones for `int32`, since in both cases the array is filled with exactly the same content (apart from the size of the key, which however should not change the relative runtimes much).

However, the results are completely different. For this data type combination, STLSort is the fastest of the tested sorting algorithms by over a factor of 2 and the sequential algorithms RadixSeq and MoellerSeq are slightly faster than most of the SIMD algorithms.

The reason for this is currently not clear, but it might somehow be caused by a difference in how integers and floating point numbers are compared.

Note that BlacherSort does not support the `double` combination and thus cannot be shown in this graph, otherwise it might be the fastest of the tested sorting algorithms in this case too.

Summary of the comparisons with other sorting algorithms

As mentioned above, more comparison graphs can be found in appendix B.

Most of the time, RadixSIMD is faster than MoellerCompress, except for the distributions Zero and ZeroOne for almost all key payload data type combinations. This was already seen in section 8.2.

RadixSIMD is also faster than the sequential algorithms except for the distributions Zero, ZeroOne, Sorted and ReverseSorted most of the time. This is probably due to the fact that RadixSIMD stores every element that was read from memory back to memory, whereas the sequential algorithms only write the elements back to memory when necessary, which for example for the Zero distribution is never the case.

The algorithms RadixSIMDNoCmp and RadixSIMDBramSmall are almost always significantly faster than RadixSIMD, indicating that the RadixSIMD algorithm spends most of the sorting time in the comparison sorter. Thus, by improving the comparison sorter, there could be significant speedup achieved.

The algorithm BlacherSort is the fastest or only slightly slower than the fastest of the tested sorting algorithms for all distributions. It however only supports the `int32` combination.

8.5 Runtime with respect to the number of elements

In this section, the runtime per element with respect to the number of elements for different distributions, key payload data type combinations and sorting algorithms is analyzed. Only some graphs are shown in this section, since most of the graphs are very similar, apart from the ordering of the algorithms by runtime, which was already discussed in the previous section.

As an example, figure 8.21 shows the runtime per element with respect to the number of elements for the Uniform distribution and the key payload data type combination `int32` for different sorting algorithms. When sorting small arrays (less than $2^4 = 16$ elements), the runtime per element is largest and approximately the same for all sorting algorithms except for IPPRadix where the runtime is significantly higher than the runtime of the other algorithms for small arrays.

As the number of elements gets larger, the runtime per element decreases for every algorithm up until about $2^6 = 64$ elements where the runtime per element plateaus for most of the algorithms. Between approximately $2^{10} = 1024$ and $2^{12} = 4096$ elements, the runtime per element gets larger as the number of elements increases for almost all of the algorithms. From that point on, the runtime per element gets larger still, but the slope of the curve is less steep than between $2^{10} = 1024$ and $2^{12} = 4096$ elements.

The initially high and then decreasing runtime per element is probably due to the fact that the calling overhead is constant and thus very high in relation to the number of elements for small arrays.

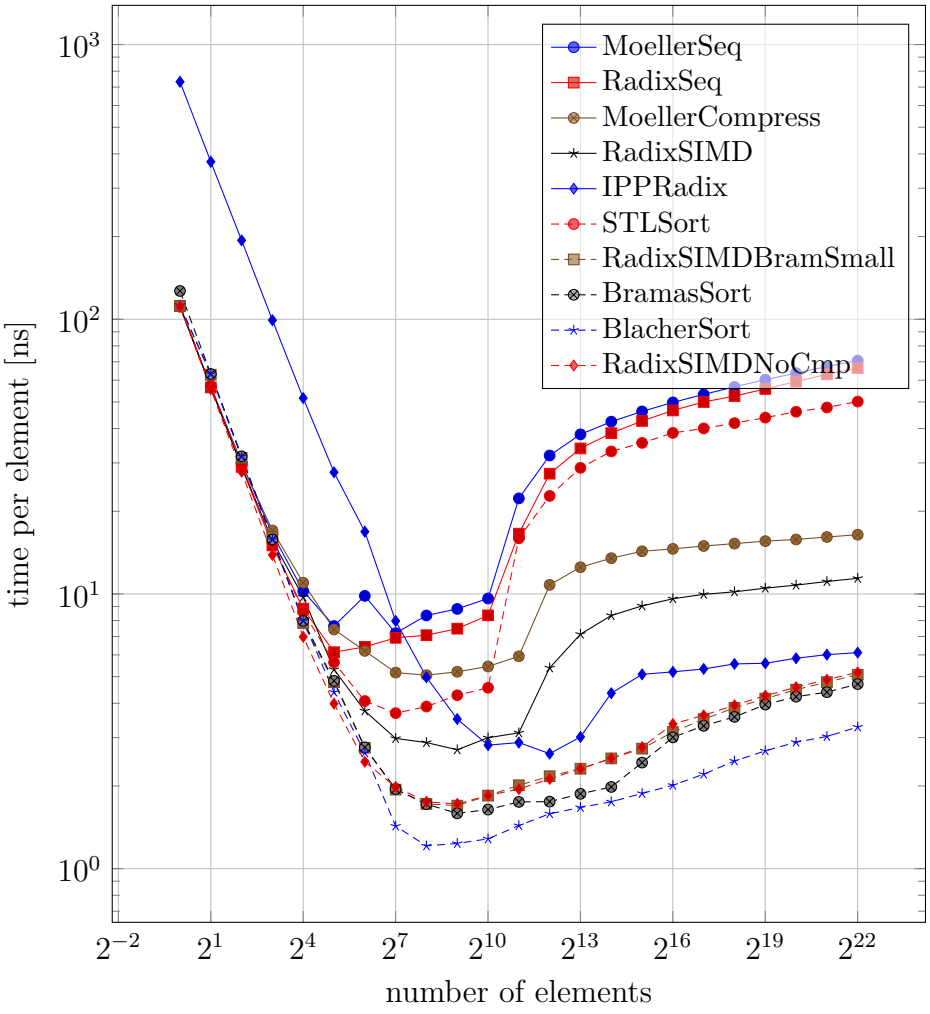


Figure 8.21: Runtime of different algorithms with respect to the number of elements for the combination `int32` with distribution Uniform.

The plateau and then rise in runtime per element is probably caused by caching effects.

Figure 8.22 shows the runtime per element of different sorting algorithm for the Zero distribution and the key payload data type combination `int32` as another example.

Here one can clearly see the runtime complexity of $\mathcal{O}(n^2)$ of BramasSort for constant input, which becomes visible for more than 2^8 elements, since below that threshold BramasSort uses bitonic sort with better runtime complexity for constant input.

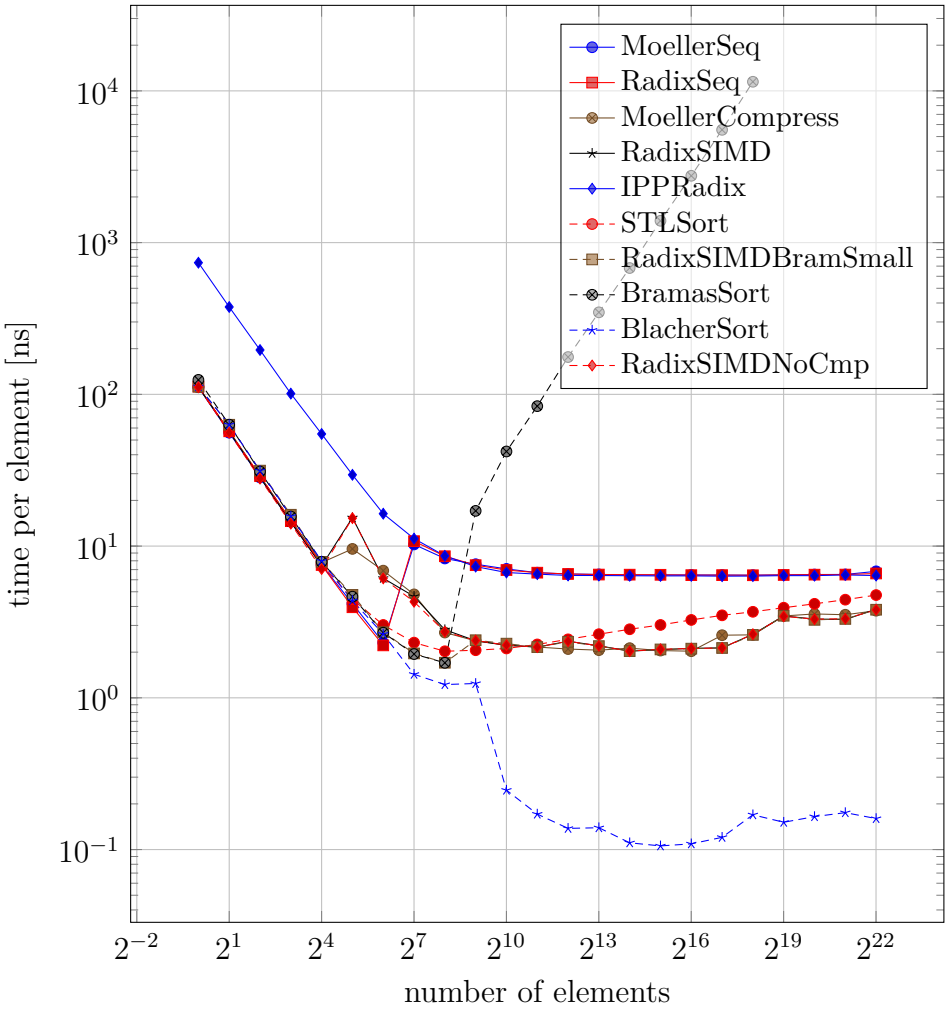


Figure 8.22: Runtime of different algorithms with respect to the number of elements for the combination `int32` with distribution Zero.

Additionally, BlacherSort has a very low runtime for more than 2^9 elements. This is very likely due to the early termination enabled by keeping track of the smallest and largest value while partitioning, as mentioned above.

8.6 Runtime of RadixSIMDNoCmp with respect to the threshold

In this section, the runtime of RadixSIMDNoCmp with respect to the threshold `cmpSortThreshold` for different distributions and key payload data type combinations is analyzed.

As in section 8.1, the size of the array did not seem to have an impact on the behavior (apart from the absolute runtimes). Thus, only the data where the array size is $2^{18} = 262144$ is shown. Additionally, the graphs for the distributions Sorted, ReverseSorted, AlmostSorted and AlmostReverseSorted are not shown either, since they are qualitatively the same as the graph for the Uniform distribution.

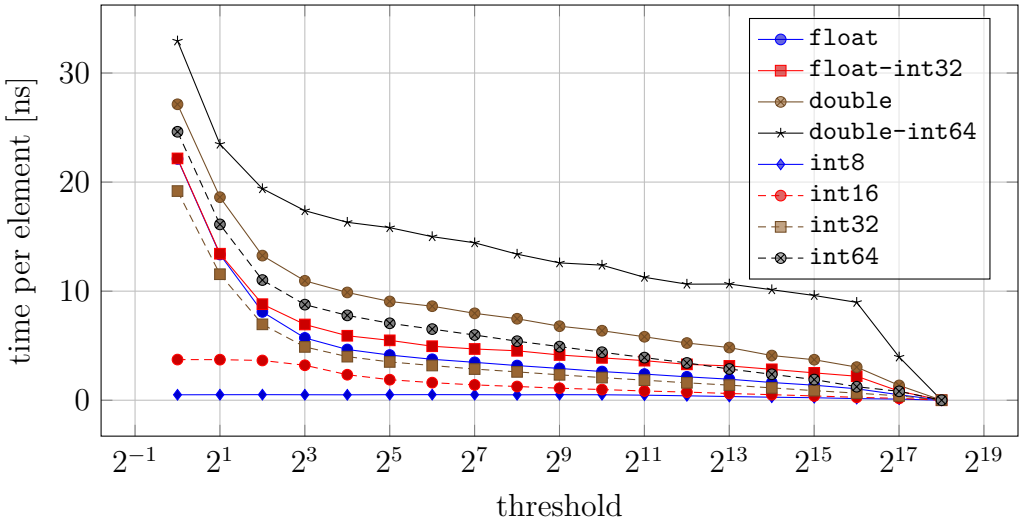


Figure 8.23: Runtime of RadixSIMDNoCmp for different values for `cmpSortThreshold`, 2^{18} elements, Uniform distribution

Figure 8.23 shows the runtime of RadixSIMDNoCmp for the Uniform distribution. As one might expect, the runtime per element decreases with increasing threshold, since the sorting tree gets less deep. When the threshold is set to the size of the array (2^{18}), the runtime per element is almost 0, as array is already at the threshold at the beginning of the sorting and the sorting algorithm immediately terminates.

For small thresholds below about 2^3, the decrease in runtime per element as the threshold increases is larger than for thresholds about 2^3, where the decrease looks linear (with a logarithmic x-axis). This is probably because the cost for each recursive call is constant, and for smaller subarrays there

are less elements sorted per recursive call, making the cost for each recursive call larger per element.

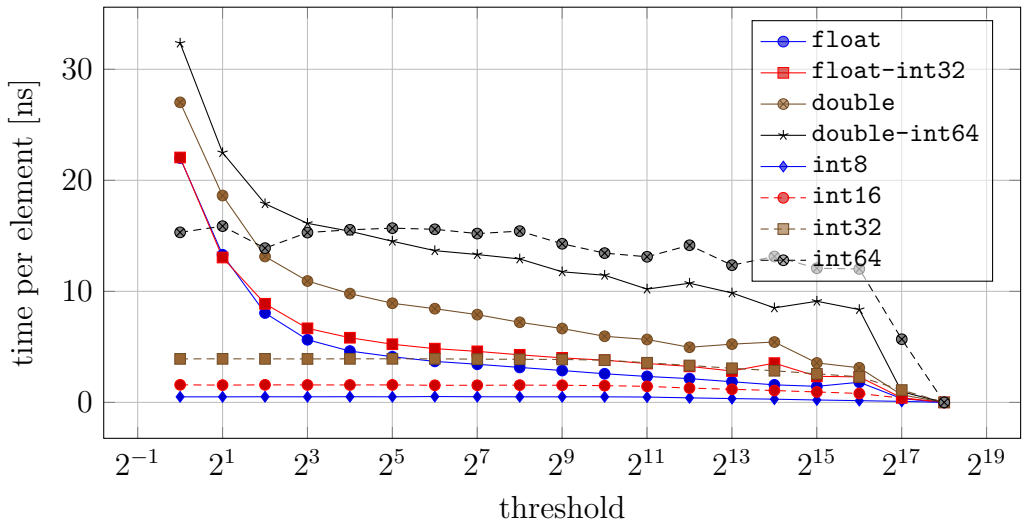


Figure 8.24: Runtime of RadixSIMDNoCmp for different values for `cmpSortThreshold`, 2^{18} elements, Gaussian distribution

Figure 8.24 shows the runtime of RadixSIMDNoCmp for the Gaussian distribution. The results are similar to the Uniform distribution.

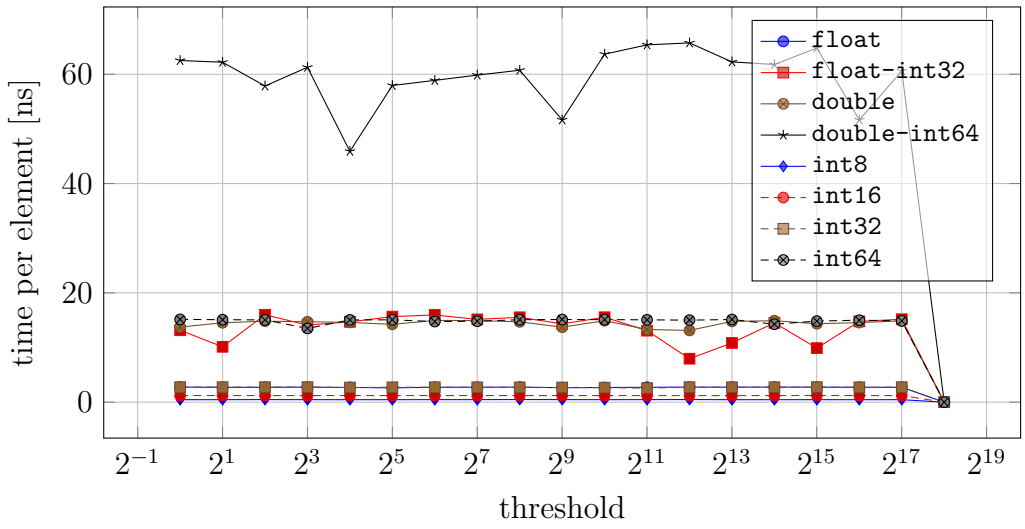


Figure 8.25: Runtime of RadixSIMDNoCmp for different values for `cmpSortThreshold`, 2^{18} elements, Zero distribution

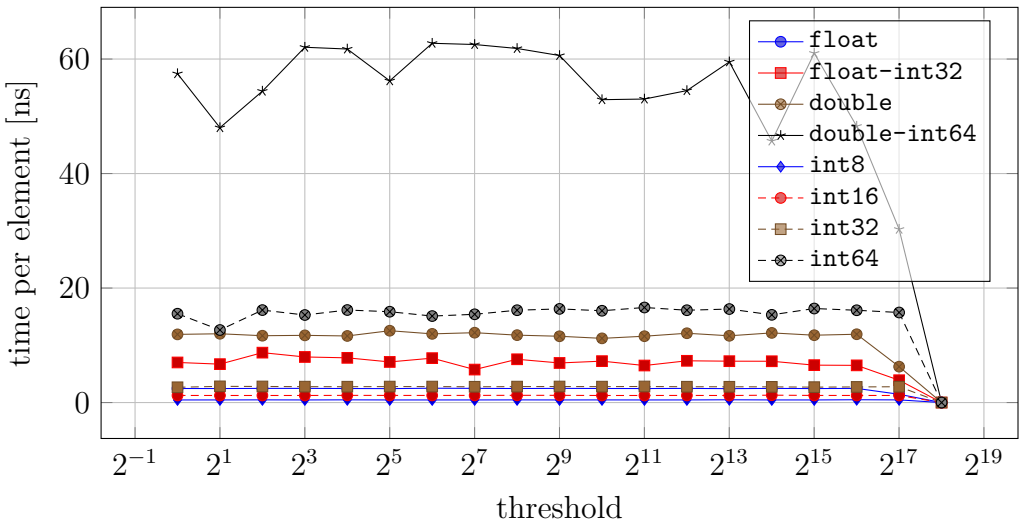


Figure 8.26: Runtime of RadixSIMDNoCmp for different values for `cmpSortThreshold`, 2^{18} elements, ZeroOne distribution

Figures 8.25 and 8.26 show the runtime of RadixSIMDNoCmp for the Zero and ZeroOne distributions. Apart from the case where the threshold is equal to the size of the array, the threshold does not seem to influence the runtime per element for these distributions. This is because the elements are all equal (Zero distribution) or only differ in the lowest bit (ZeroOne distribution). This causes one of the two subarrays produced by the bit sorter to be empty, while the other one is the same size as the input array. Thus, the subarrays never get smaller and therefore never reach the threshold, independent of the actual value of the threshold.

9

Discussion

9.1 Discussion of the results

The results presented in chapter 8 show that the goal of this thesis was achieved to a large extent. For most of the key payload data type combinations and most distributions, the presented implementation of MSB Radix Sort with separate key and payload datastreams (RadixSIMD) is faster or equally as fast as the implementation of MSB Radix Sort with combined key and payload datastreams by Möller (2021) (MoellerCompress). As the comparison between the two algorithms with and without payloads shows, this is at least partly due to the separation of the key and payload datastreams when sorting datasets with payload. Another reason for the speedup is likely due to the difference in the handling of remaining elements in the bit sorter.

For datasets with payload with Zero or ZeroOne distribution, RadixSIMD is slower than MoellerCompress. The reason for this is not clear and might be worth investigating.

The algorithm MoellerCompress by Möller (2021) was already significantly faster than STLSort (the sorting algorithm included in the C++ standard template library (Wikipedia contributors, 2022c)), which was verified in this thesis. The algorithm RadixSIMD developed in this thesis provides an additional speedup over STLSort.

The comparisons with the sorting algorithms RadixSIMDBramSmall and RadixSIMDNoCmp show that the algorithm RadixSIMD spends a large portion of the sorting time in the comparison sorter for small subarrays and thus can be accelerated significantly with just improving the comparison sorter or using a better one.

The algorithms IPPRadix and BramasSort (Bramas, 2017) outperformed the algorithms developed by Möller (2021) and in this thesis significantly for many tested scenarios while BlacherSort (Blacher et al., 2021) outperformed all tested algorithms for almost all tested scenarios. However, IPPRadix does not support payloads, BramasSort only supports the combinations `int32`, `double` and `int32-int32` and BlacherSort only supports the combination `int32`, while RadixSIMD supports arbitrary keys and payloads.

9.2 Possible improvements

9.2.1 Caching more vectors

For datasets where the key is smaller than one of the payloads, the use of larger emulated vector registers for the payloads and thus the caching of more vectors as described in section 6.4.3 (the emulated vector registers consist of multiple physical vector registers, see section 5.1) provides a significant speedup, as seen in section 8.3.

It may be beneficial to cache even more vectors by using the emulated vectors for the keys as well, not just for datasets where the key is smaller than one of the payloads. This would require larger emulated masks. However, the speedup for the method described in section 6.4.3 is probably only caused by the fact that the keys are still stored in only a single physical register and thus the bit tests can be executed with one instruction, but the vector register is larger, allowing more keys to be sorted simultaneously. Caching even more vectors and using the emulated vectors for the keys as well would require the bit testing of the keys to be split into multiple instructions (one for each physical part of the emulated vector register) and thus might not provide additional speedup.

Nevertheless, caching more vectors might accelerate the sorting algorithm because of other effects (such as more efficient cache usage or enabling better instruction-level parallelism).

It would be interesting to see what happens when the algorithm uses more vector registers than are available on the processor and thus has to swap vector into memory.

9.2.2 Faster sorting algorithm for small subarrays

As seen in section 8.4, RadixSIMD can be accelerated greatly by using a faster comparison sorter for small subarrays. There may be an algorithm that performs even better than the small sort by Bramas (2017) used by

RadixSIMDBramSmall. A faster comparison sorter might also allow the threshold `cmpSortThreshold` to be increased and thus making the sorting tree of the MSB Radix Sort shallower, which may further increase performance.

9.2.3 Keeping track of smallest and largest element for early termination

The AVX2 Quicksort implementation by Blacher et al. (2021) keeps track of the smallest and largest element in the subarray when partitioning. By comparing these two elements with the pivot, it is possible to detect if all the elements in either of the two resulting subarrays are the same. If this is the case, the sorting of the respective subarray can be skipped. This improves the sorting time for arrays with many duplicate elements significantly, as can be seen in section 8.4 or the graphs in appendix B.

It should be possible to implement this for the MSB Radix Sort as well, which would probably accelerate the sorting of arrays with many duplicate elements.

When sorting arrays with few or without any duplicates however, this modification should not lead to early termination, but the algorithm still has to keep track of the smallest and largest element. So, if this were to be implemented, the performance when sorting arrays with no duplicates should be analyzed as well.

9.2.4 AVX/AVX2 version

The implementation of MSB Radix Sort presented in this thesis uses the AVX-512 instruction set and for some key payload data type combinations it requires some AVX-512 extensions that not all CPUs with AVX-512 support have.

For CPUs that do not support AVX-512, it might be possible to implement an AVX/AVX2 version of the algorithm by simulating `mask_compressstoreu` instructions in a similar way Blacher et al. (2021) did with his AVX2 implementation of Quicksort.

9.2.5 Multithreading

The current implementation of the algorithm only runs on a single thread. But the two subarrays the bit sorter produces can be sorted independent of each other which allows them to be sorted simultaneously by multiple threads. On CPUs with multiple cores this very likely improves the performance of the algorithm significantly.

However, to produce the two subarrays the bit sorter has run first, so the two subarrays are not available immediately to be sorted by multiple threads and only one thread is used in the beginning. There might be a way to parallelize the bit sorting to benefit from multiple threads early in the sorting process.

9.2.6 Quicksort instead of MSB Radix Sort

MSB Radix Sort can be interpreted as a variant of Quicksort, where the pivot is a 'virtual' element with exactly one bit set (Knuth, 1998, p. 128). By actually using Quicksort instead of MSB Radix Sort, a different pivot could be chosen such that the sorting tree is more balanced. For example, the implementation by Blacher et al. (2021) seems to choose a pivot very well, as this sorting algorithm outperforms all other tested algorithm with almost all distributions (see section 8.4).

9.2.7 Additional experiments

In this thesis it was not analyzed how the runtime of the algorithm depends on the number and size of the payloads.

One might expect, that the runtime increases linearly with the number as well as the size of the payloads at least until there are not enough vector registers available to hold a key vector and a vector for each payload. It would be interesting to see what happens if there are not enough vector registers available and the vectors have to be swapped into memory.

Additionally, multiple smaller payloads could be arranged into multiple payload arrays (SoA) but could also be combined into one bigger payload, essentially interleaving the multiple smaller payloads into one array (AoS). It would be interesting to analyze which of these two options is faster.

The behavior of the algorithm for distributions with no duplicate keys was not analyzed in this thesis either, as most of the distributions used in the experiments in this thesis likely had duplicate keys. It might be worthwhile to test if there is a difference in the runtime of the algorithms when the distributions have no duplicates.

A

C++ code for the SIMD bit sorter class

```
template <bool OneReg = false> struct BitSorterSIMD {
  template <typename K, typename... Ps>
  static constexpr SortIndex
      numElemsPerVec = OneReg ? 64 / std::max({sizeof(K), sizeof(Ps)...})
                              : 64 / sizeof(K);

  template <bool Up, bool IsHighestBit, bool IsRightSide, typename K,
            typename... Ps>
  static INLINE SortIndex sort(int bitNo, SortIndex left, SortIndex right,
                               K *keys, Ps *...payloads) {
    static constexpr SortIndex _numElemsPerVec = numElemsPerVec<K, Ps...>;

    SortIndex numElems = right - left + 1;

    SortIndex readPosLeft = left;
    SortIndex readPosRight = right - _numElemsPerVec + 1;
    SortIndex writePosLeft = left;
    SortIndex writePosRight = right;

    simd::Vec<K, _numElemsPerVec * sizeof(K)> keyVecStore;
    std::tuple<simd::Vec<Ps, _numElemsPerVec * sizeof(Ps)>...> payloadVecStore;
    if (numElems >= _numElemsPerVec) {
      keyVecStore =
          simd::loadu<_numElemsPerVec * sizeof(K)>(&keys[readPosLeft]);
      payloadVecStore = std::make_tuple(
          simd::loadu<_numElemsPerVec * sizeof(Ps)>(&payloads[readPosLeft])...);
      readPosLeft += _numElemsPerVec;
    }

    while (readPosLeft <= readPosRight) {
      auto keyVec = keyVecStore;
      auto payloadVec = payloadVecStore;
      auto [sortMaskLeft, sortMaskRight] =
          getSortMasks<Up, IsHighestBit, IsRightSide, K, Ps...>(keyVec, bitNo);
      SortIndex numElemsToLeft = simd::kpopcnt(sortMaskLeft);
```

```
36        SortIndex numElemsToRight = _numElemsPerVec - numElemsToLeft;
37        bool areEnoughElemsFreeLeft =
38            (readPosLeft - writePosLeft) >= numElemsToLeft;
39        if (areEnoughElemsFreeLeft) {
40          keyVecStore =
41              simd::loadu<_numElemsPerVec * sizeof(K)>(&keys[readPosRight]);
42          payloadVecStore =
43              std::make_tuple(simd::loadu<_numElemsPerVec * sizeof(Ps)>(
44                  &payloads[readPosRight])...);
45          readPosRight -= _numElemsPerVec;
46        } else {
47          keyVecStore =
48              simd::loadu<_numElemsPerVec * sizeof(K)>(&keys[readPosLeft]);
49          payloadVecStore =
50              std::make_tuple(simd::loadu<_numElemsPerVec * sizeof(Ps)>(
51                  &payloads[readPosLeft])...);
52          readPosLeft += _numElemsPerVec;
53        }
54        compress_store_left_right(
55            writePosLeft, writePosRight - numElemsToRight + 1, sortMaskLeft,
56            sortMaskRight, keyVec, payloadVec, keys, payloads...);
57        writePosLeft += numElemsToLeft;
58        writePosRight -= numElemsToRight;
59      }
60
61      SortIndex numElemsRest = readPosRight + _numElemsPerVec - readPosLeft;
62
63      simd::Mask<_numElemsPerVec> restMask;
64      simd::Vec<K, _numElemsPerVec * sizeof(K)> keyVecRest;
65      std::tuple<simd::Vec<Ps, _numElemsPerVec * sizeof(Ps)>...> payloadVecRest;
66      if (numElemsRest != 0) {
67        restMask = (simd::knot(simd::Mask<_numElemsPerVec>(0))) >>
68                   (_numElemsPerVec - numElemsRest);
69        keyVecRest = simd::maskz_loadu<_numElemsPerVec * sizeof(K)>(
70            restMask, &keys[readPosLeft]);
71        payloadVecRest =
72            std::make_tuple(simd::maskz_loadu<_numElemsPerVec * sizeof(Ps)>(
73                restMask, &payloads[readPosLeft])...);
74        readPosLeft += numElemsRest;
75      }
76
77      if (numElems >= _numElemsPerVec) {
78        auto [sortMaskLeft, sortMaskRight] =
79            getSortMasks<Up, IsHighestBit, IsRightSide, K, Ps...>(keyVecStore,
80                                                                  bitNo);
81        SortIndex numElemsToLeft = simd::kpopcnt(sortMaskLeft);
82        SortIndex numElemsToRight = _numElemsPerVec - numElemsToLeft;
83        compress_store_left_right(
84            writePosLeft, writePosRight - numElemsToRight + 1, sortMaskLeft,
85            sortMaskRight, keyVecStore, payloadVecStore, keys, payloads...);
86        writePosLeft += numElemsToLeft;
87        writePosRight -= numElemsToRight;
88      }
89
90      if (numElemsRest != 0) {
91        auto [sortMaskLeftRest, sortMaskRightRest] =
92            getSortMasks<Up, IsHighestBit, IsRightSide, K, Ps...>(keyVecRest,
93                                                                  bitNo);
94        sortMaskLeftRest = simd::kand(sortMaskLeftRest, restMask);
95        sortMaskRightRest = simd::kand(sortMaskRightRest, restMask);
96        SortIndex numElemsToLeftRest = simd::kpopcnt(sortMaskLeftRest);
97        SortIndex numElemsToRightRest = numElemsRest - numElemsToLeftRest;
98        compress_store_left_right(writePosLeft, writePosLeft + numElemsToLeftRest,
99                                  sortMaskLeftRest, sortMaskRightRest, keyVecRest,
```

```
100                                 payloadVecRest, keys, payloads...);
101       writePosLeft += numElemsToLeftRest;
102       writePosRight -= numElemsToRightRest;
103     }
104     return writePosLeft;
105   }
106
107 private:
108   template <bool Up, bool IsHighestBit, bool IsRightSide, typename K,
109             typename... Ps>
110   static INLINE std::tuple<simd::Mask<numElemsPerVec<K, Ps...>>,
111                            simd::Mask<numElemsPerVec<K, Ps...>>>
112   getSortMasks(simd::Vec<K, numElemsPerVec<K, Ps...> * sizeof(K)> keyVec,
113                int bitNo) {
114     auto bitMaskVec = simd::reinterpret<K>(
115         simd::set1<UInt<sizeof(K)>, numElemsPerVec<K, Ps...> * sizeof(K)>(
116             UInt<sizeof(K)>(1) << bitNo));
117     if constexpr (bitDirUp<K, Up, IsHighestBit, IsRightSide>()) {
118       auto sortMaskRight = simd::test_mask(keyVec, bitMaskVec);
119       auto sortMaskLeft = simd::knot(sortMaskRight);
120       return std::make_tuple(sortMaskLeft, sortMaskRight);
121     } else {
122       auto sortMaskLeft = simd::test_mask(keyVec, bitMaskVec);
123       auto sortMaskRight = simd::knot(sortMaskLeft);
124       return std::make_tuple(sortMaskLeft, sortMaskRight);
125     }
126   }
127
128   template <typename K, typename... Ps>
129   static INLINE void compress_store_left_right(
130       SortIndex leftPos, SortIndex rightPos,
131       simd::Mask<numElemsPerVec<K, Ps...>> leftMask,
132       simd::Mask<numElemsPerVec<K, Ps...>> rightMask,
133       simd::Vec<K, numElemsPerVec<K, Ps...> * sizeof(K)> keyVec,
134       std::tuple<simd::Vec<Ps, numElemsPerVec<K, Ps...> * sizeof(Ps)>...>
135           payloadVec,
136       K *keys, Ps *...payloads) {
137
138     simd::mask_compressstoreu(&keys[leftPos], leftMask, keyVec);
139     std::apply(
140         [&](auto... payloadVecs) {
141           (simd::mask_compressstoreu(&payloads[leftPos], leftMask, payloadVecs),
142            ...);
143         },
144         payloadVec);
145
146     simd::mask_compressstoreu(&keys[rightPos], rightMask, keyVec);
147     std::apply(
148         [&](auto... payloadVecs) {
149           (simd::mask_compressstoreu(&payloads[rightPos], rightMask,
150                                      payloadVecs),
151            ...);
152         },
153         payloadVec);
154   }
155 };
```

B

Comparison graphs

The bars in the following figures (B.1-B.56) are colored to make the graphs easier to read. The bars for the sequential algorithms are colored black, RadixSIMD is colored red and RadixSIMDBramSmall is colored brown. The bars for the rest of the algorithms are colored blue. The bar for RadixSIMD-NoCmp is dashed as a reminder that it does not actually fully sort the data.

B.1 int32

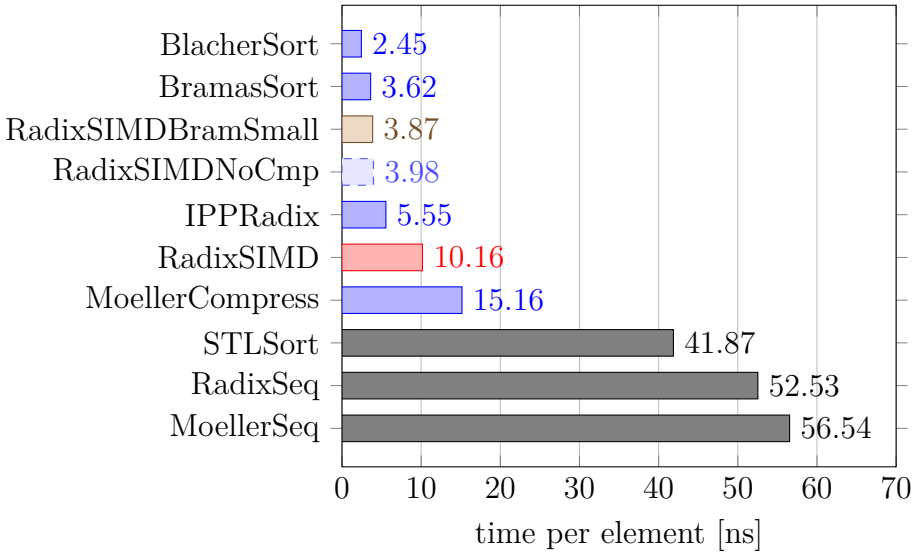


Figure B.1: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution Uniform

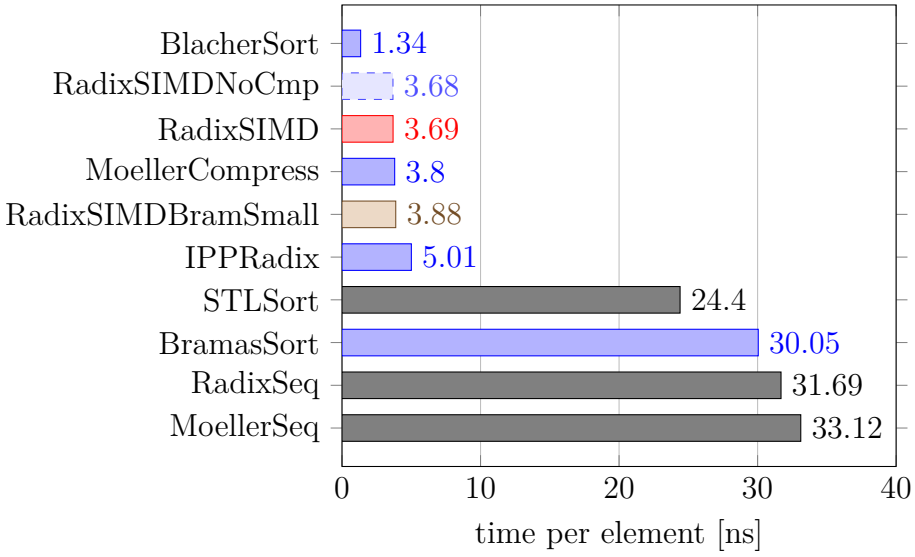


Figure B.2: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution Gaussian

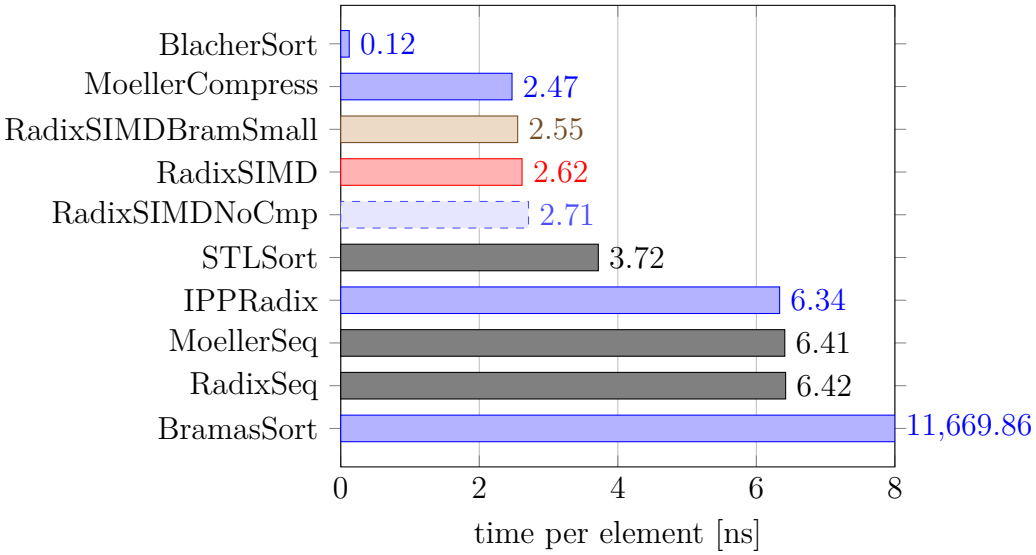


Figure B.3: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution Zero

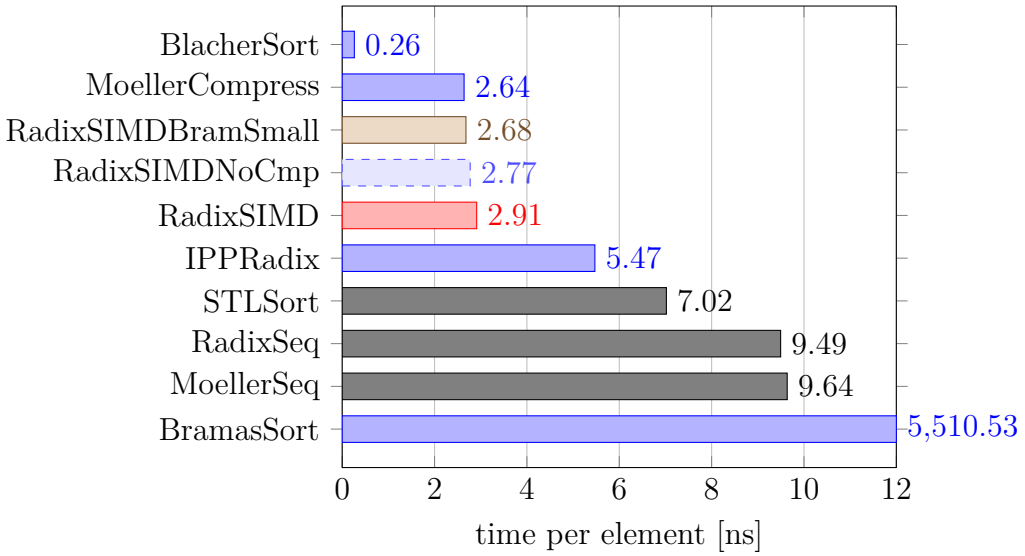


Figure B.4: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution ZeroOne

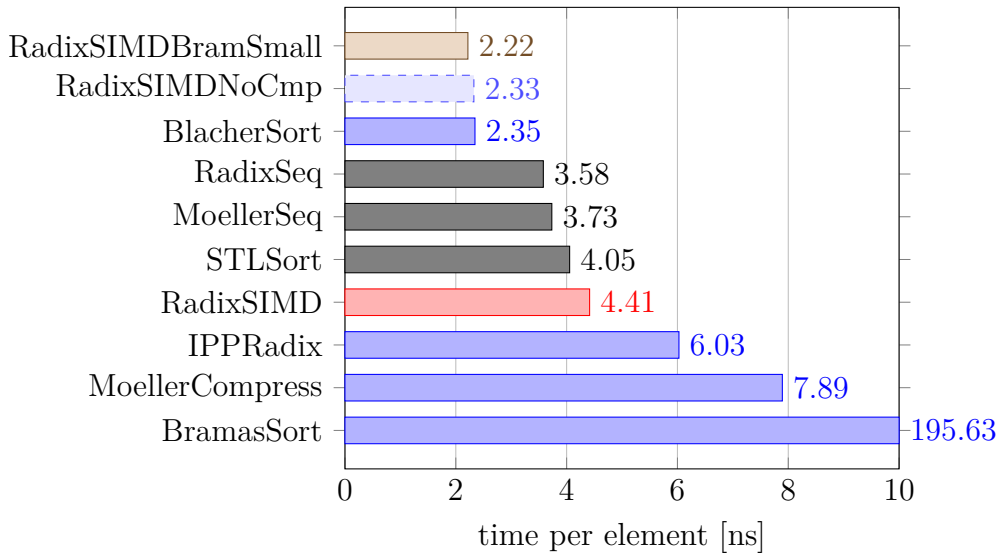


Figure B.5: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution Sorted

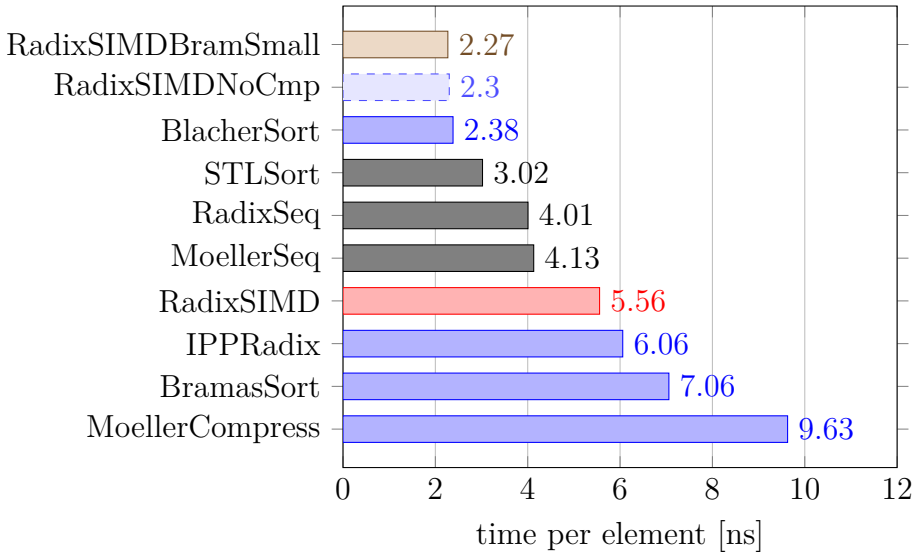


Figure B.6: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution ReverseSorted

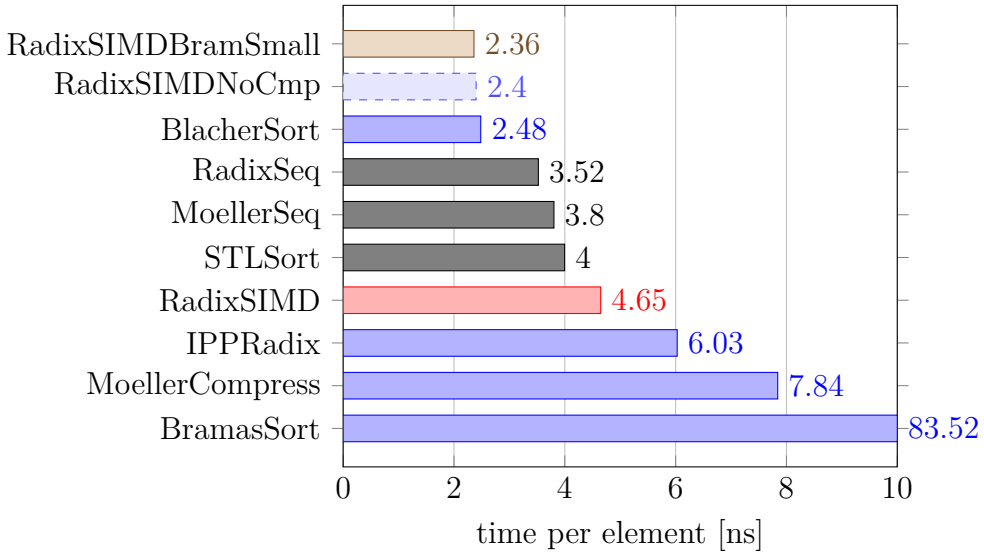


Figure B.7: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution AlmostSorted

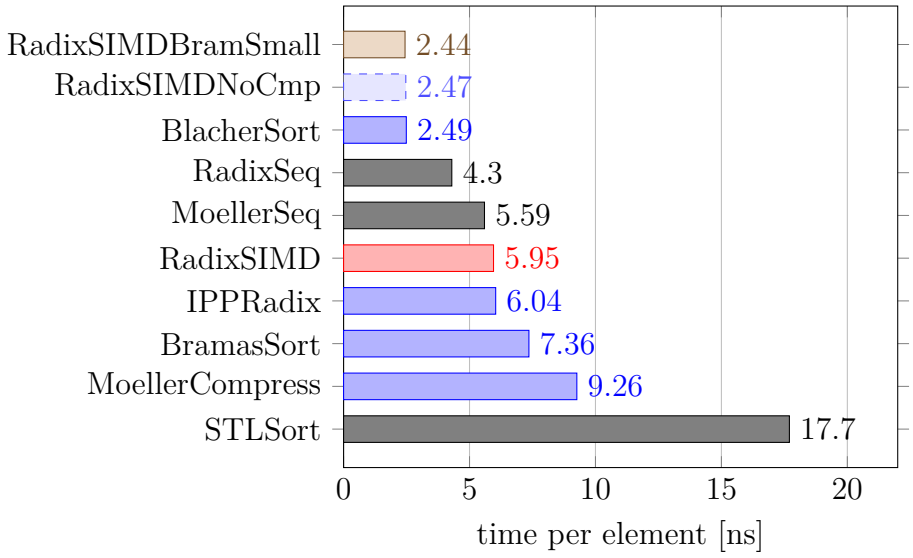


Figure B.8: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32` and distribution AlmostReverseSorted

B.2 int32-int32

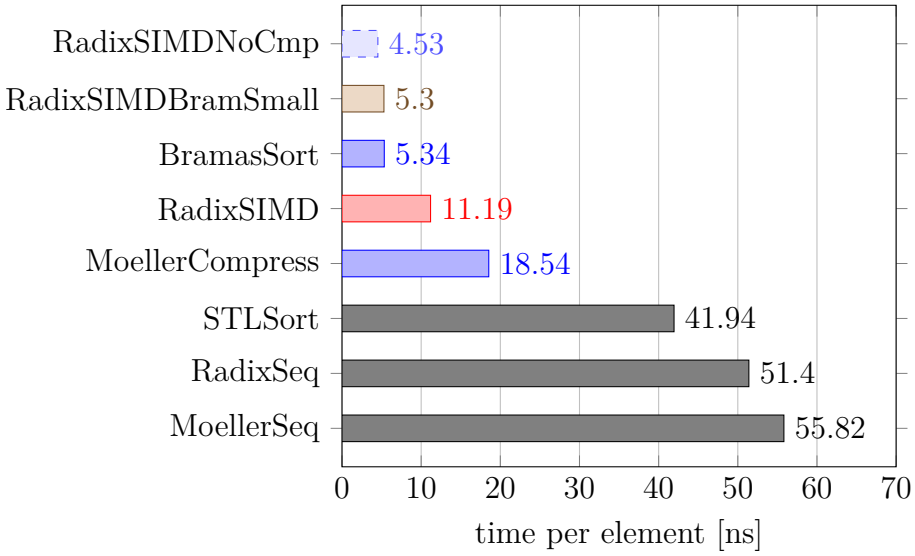


Figure B.9: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32-int32` and distribution Uniform

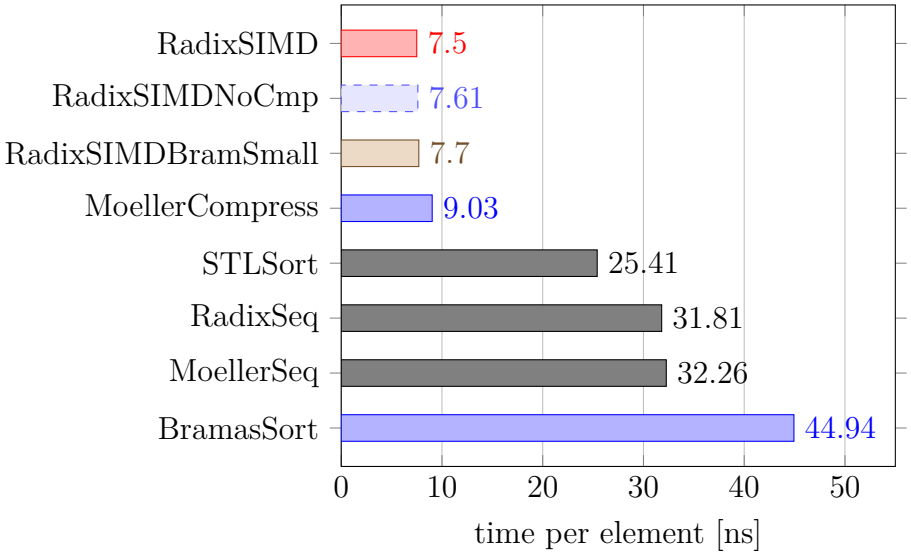


Figure B.10: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32-int32` and distribution Gaussian

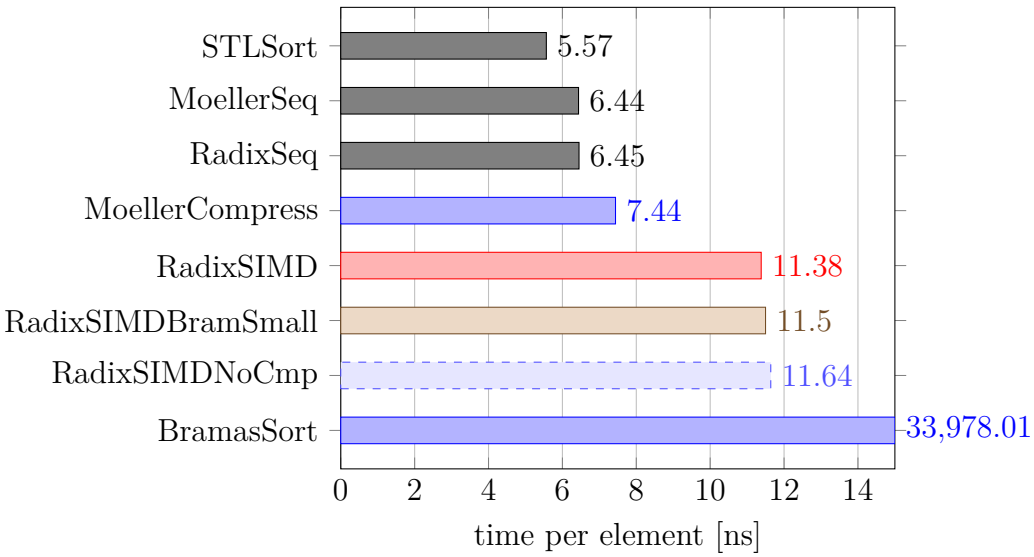


Figure B.11: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32-int32` and distribution Zero

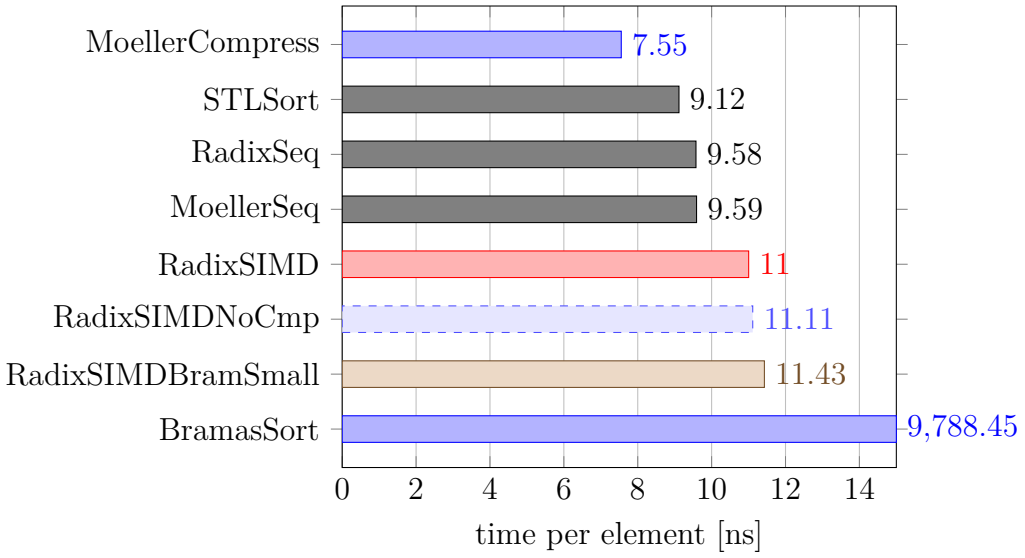


Figure B.12: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32-int32` and distribution ZeroOne

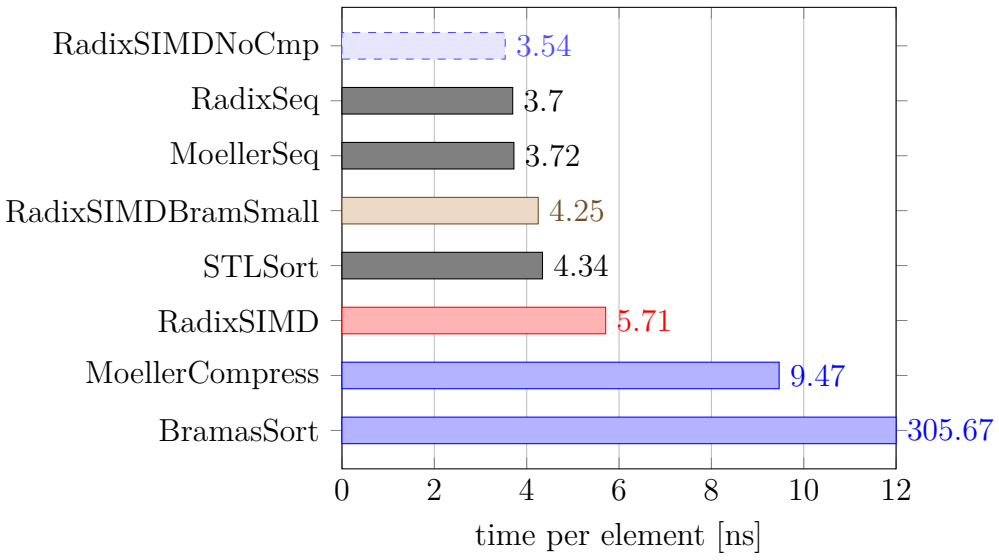


Figure B.13: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32-int32` and distribution Sorted

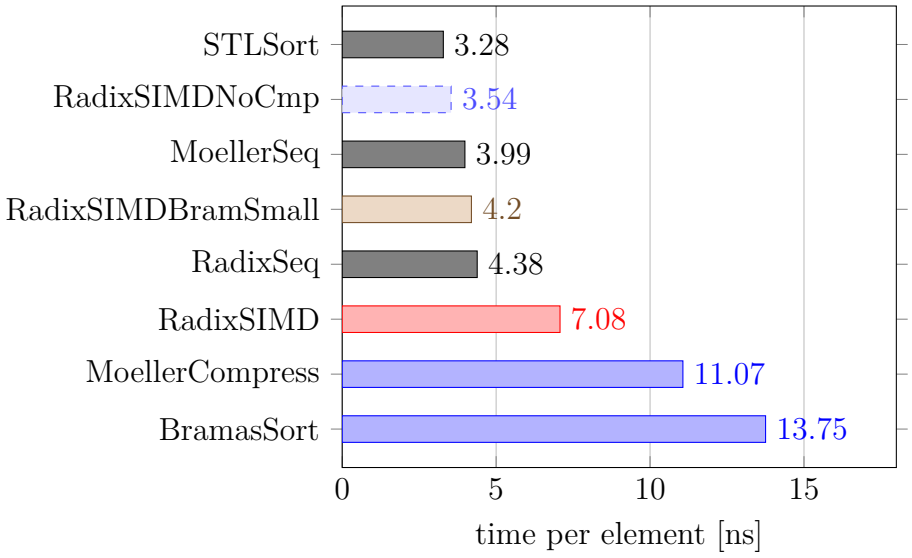


Figure B.14: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32-int32` and distribution ReverseSorted

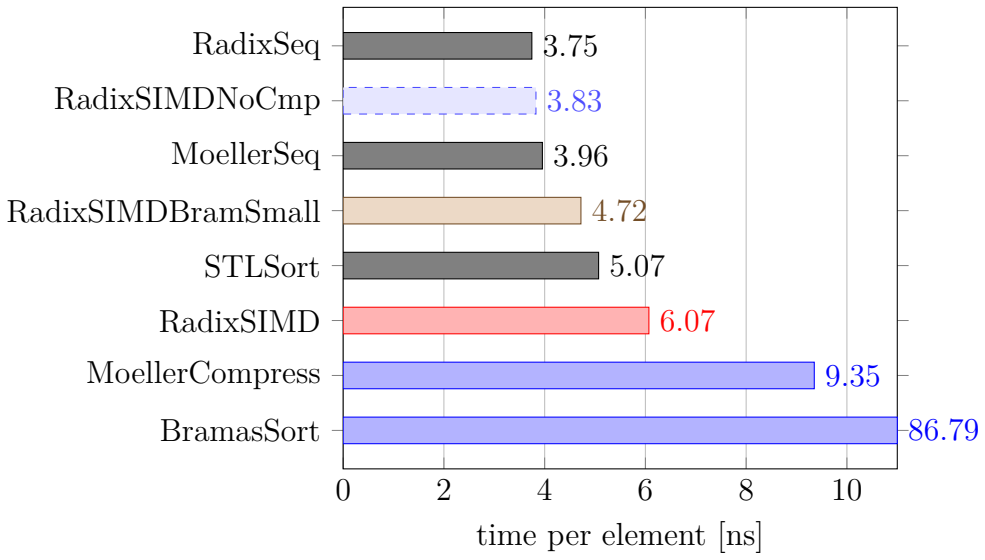


Figure B.15: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32-int32` and distribution AlmostSorted

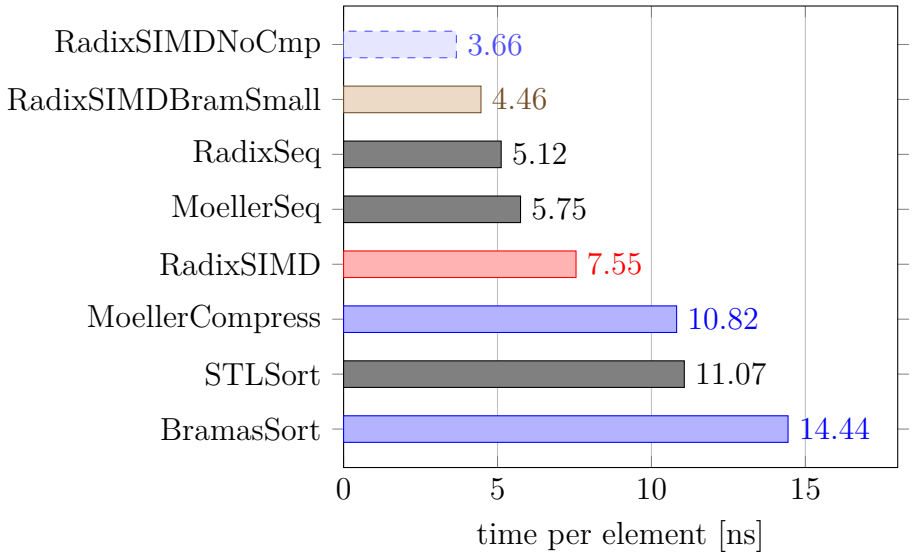


Figure B.16: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int32-int32` and distribution AlmostReverseSorted

B.3 `float`

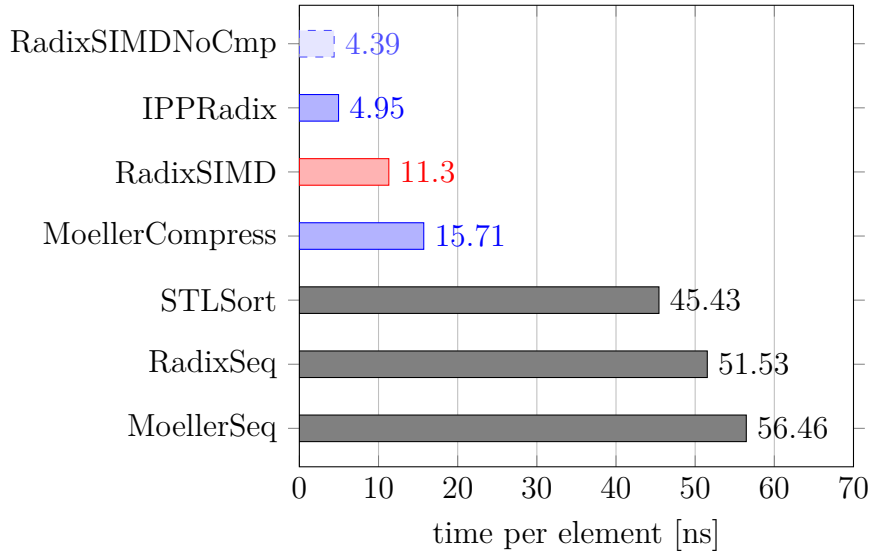


Figure B.17: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float` and distribution Uniform

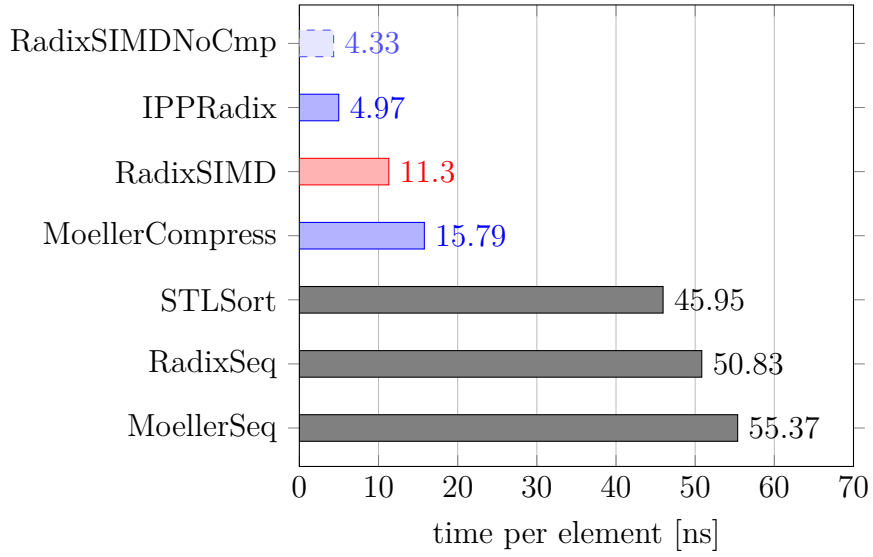


Figure B.18: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float` and distribution Gaussian

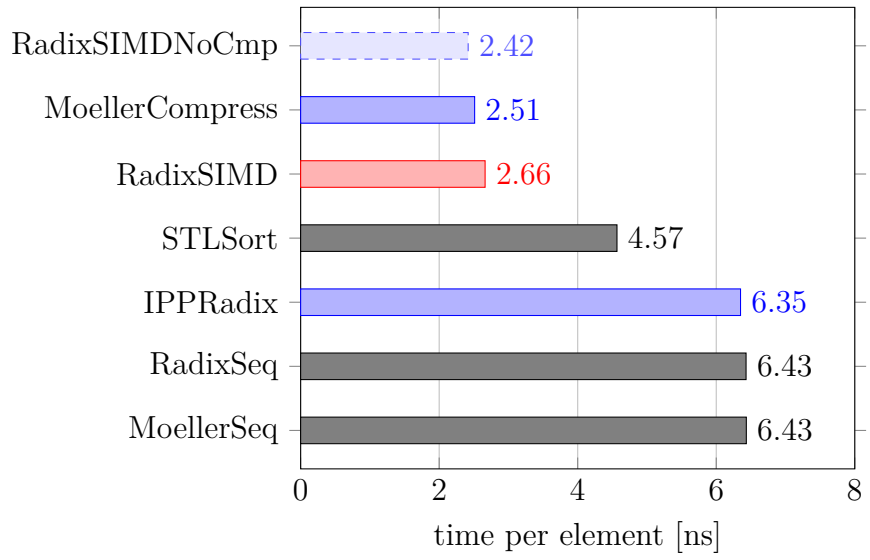


Figure B.19: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float` and distribution Zero

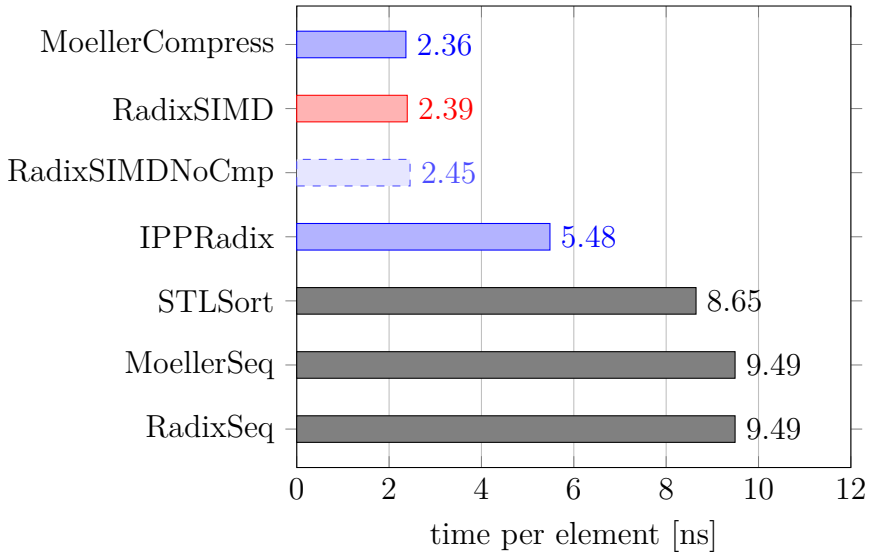


Figure B.20: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float` and distribution ZeroOne

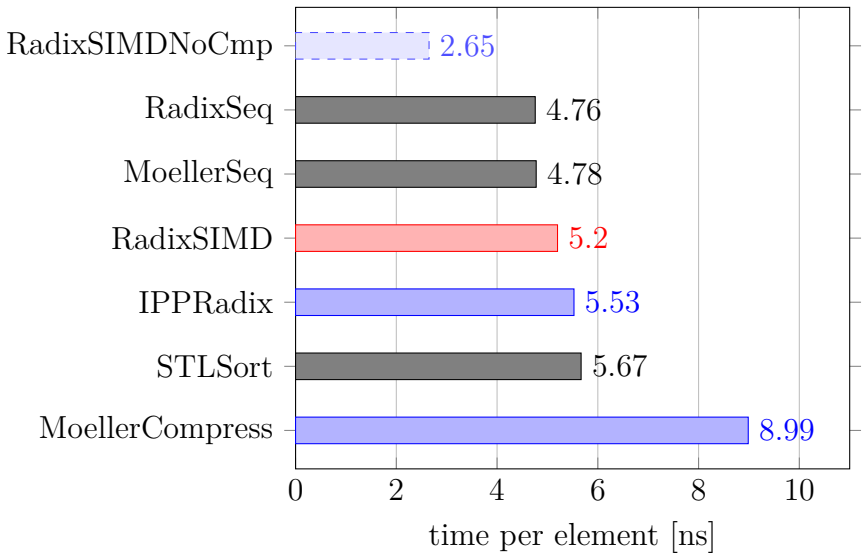


Figure B.21: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float` and distribution Sorted

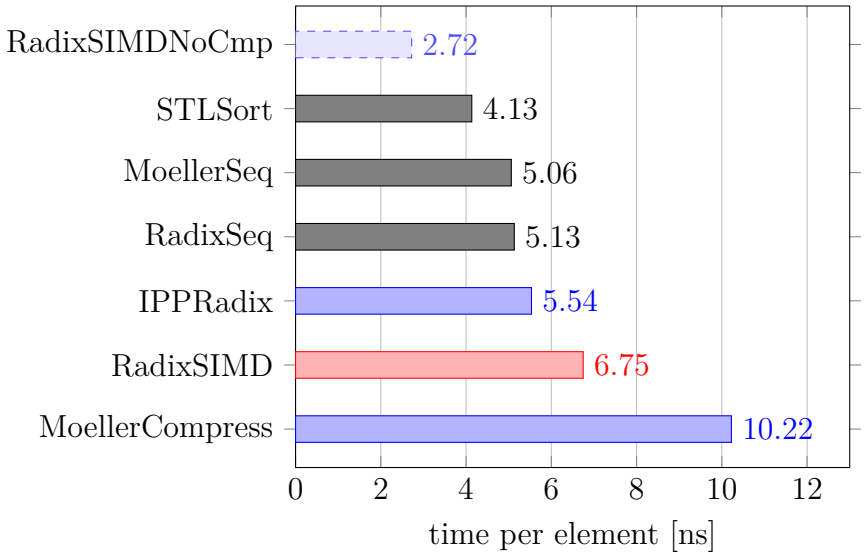


Figure B.22: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float` and distribution ReverseSorted

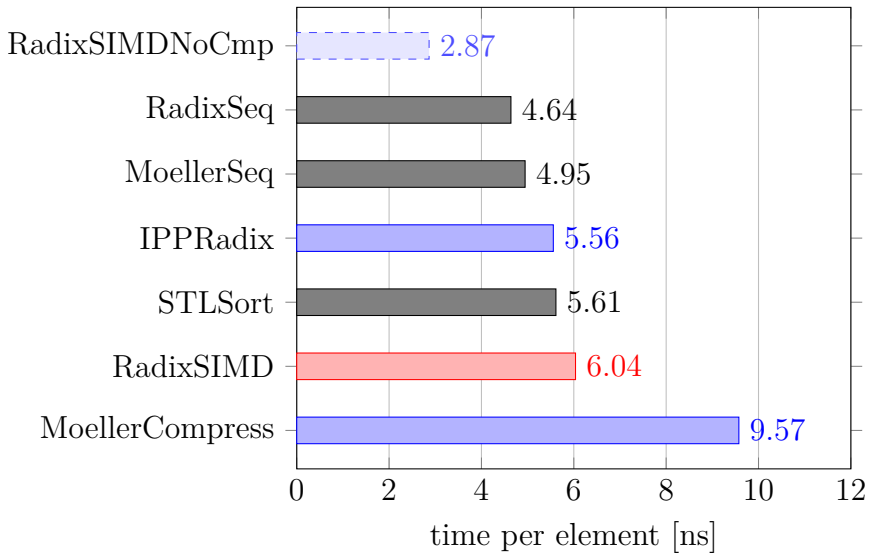


Figure B.23: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float` and distribution AlmostSorted

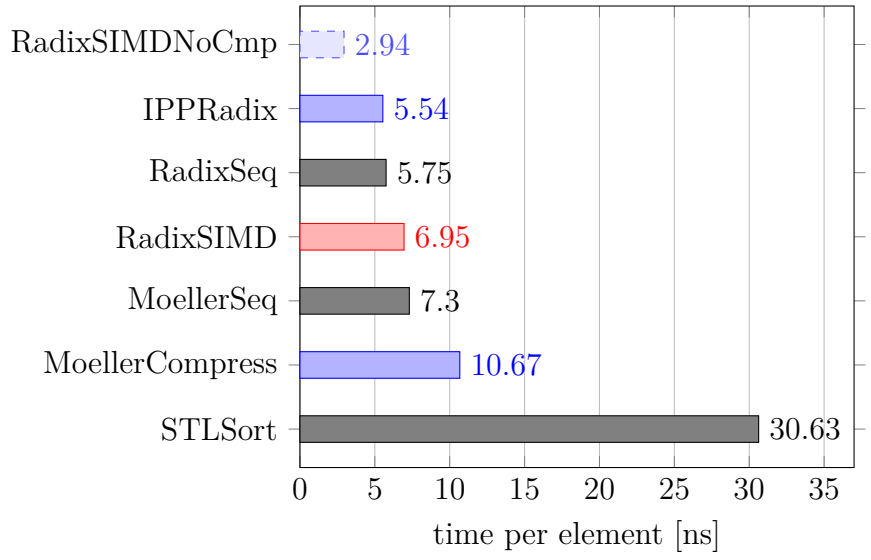


Figure B.24: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float` and distribution AlmostReverseSorted

B.4 `float-int32`

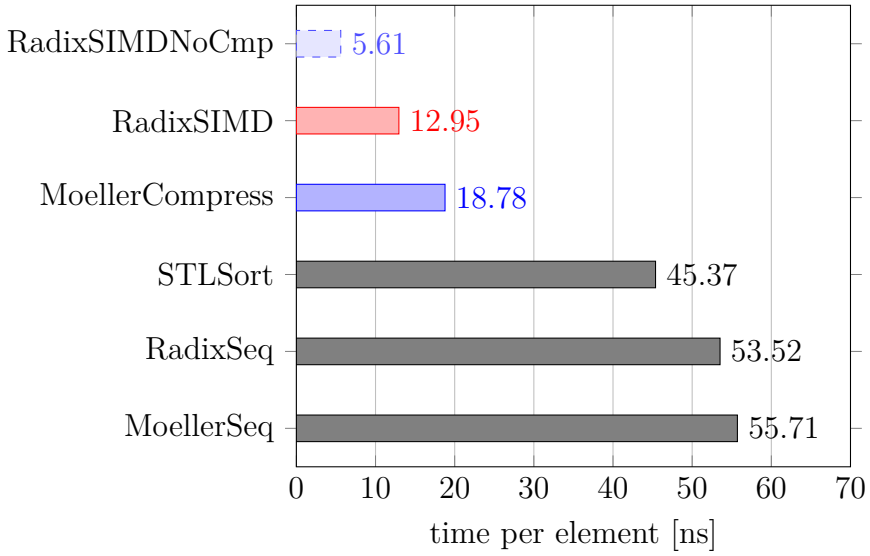


Figure B.25: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float-int32` and distribution Uniform

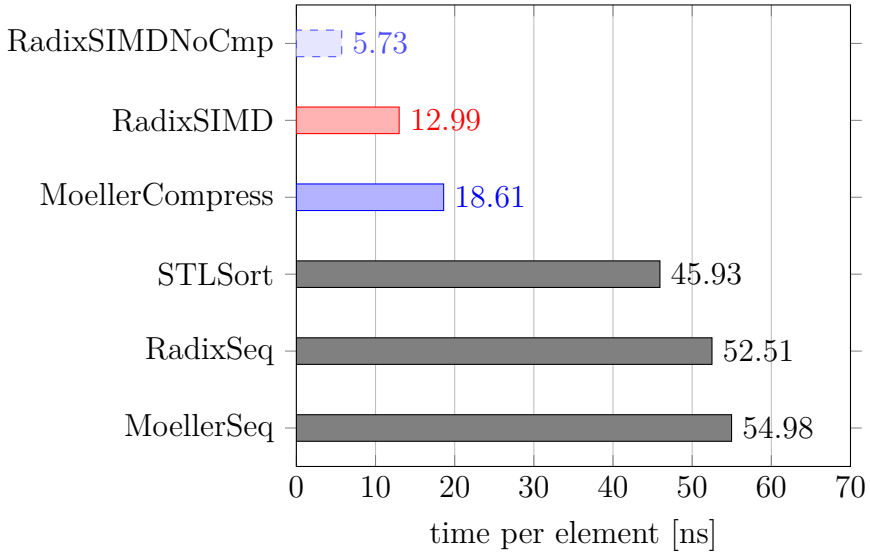


Figure B.26: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float-int32` and distribution Gaussian

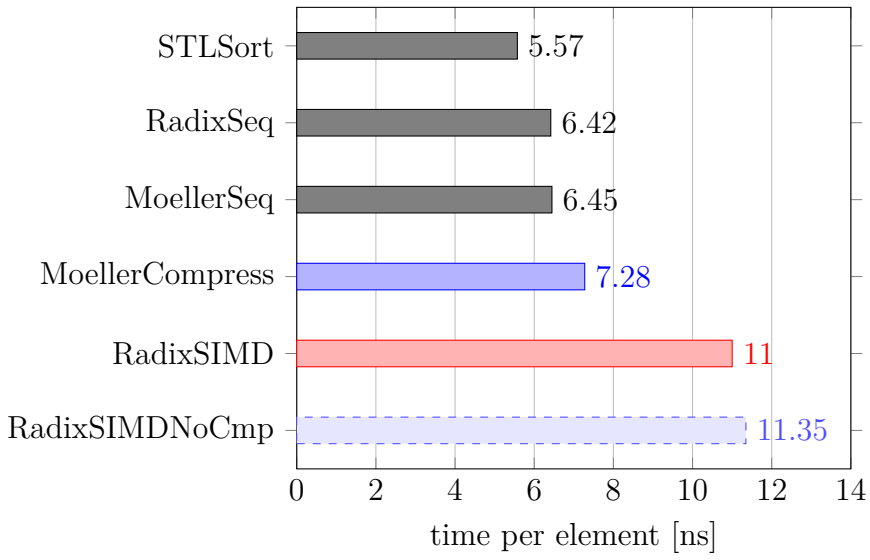


Figure B.27: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float-int32` and distribution Zero

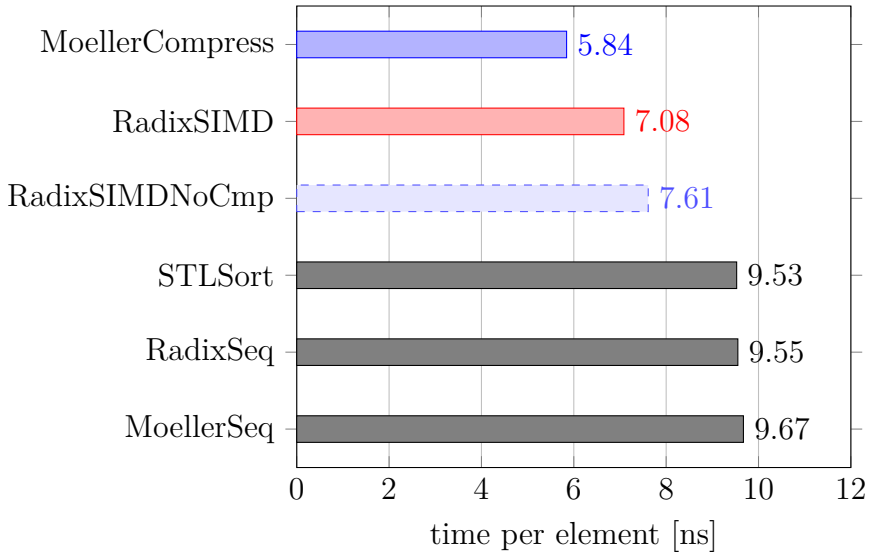


Figure B.28: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float-int32` and distribution ZeroOne

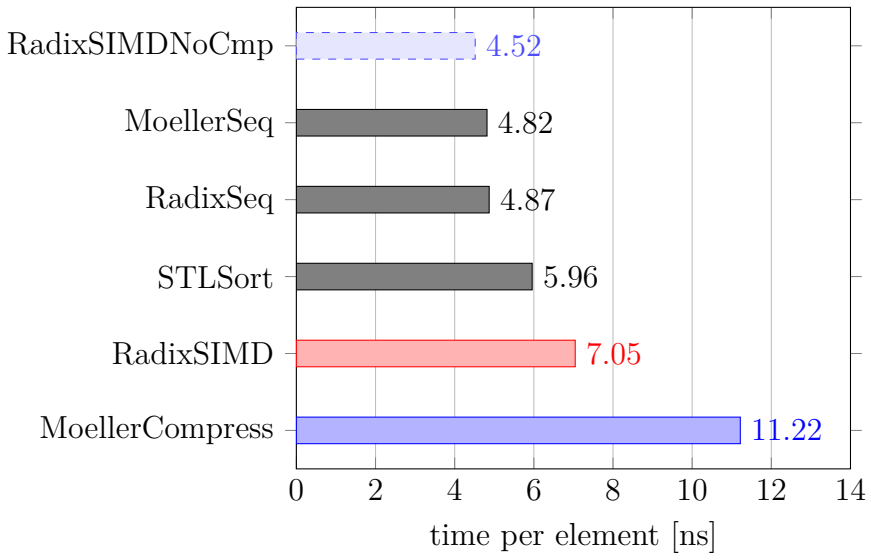


Figure B.29: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float-int32` and distribution Sorted

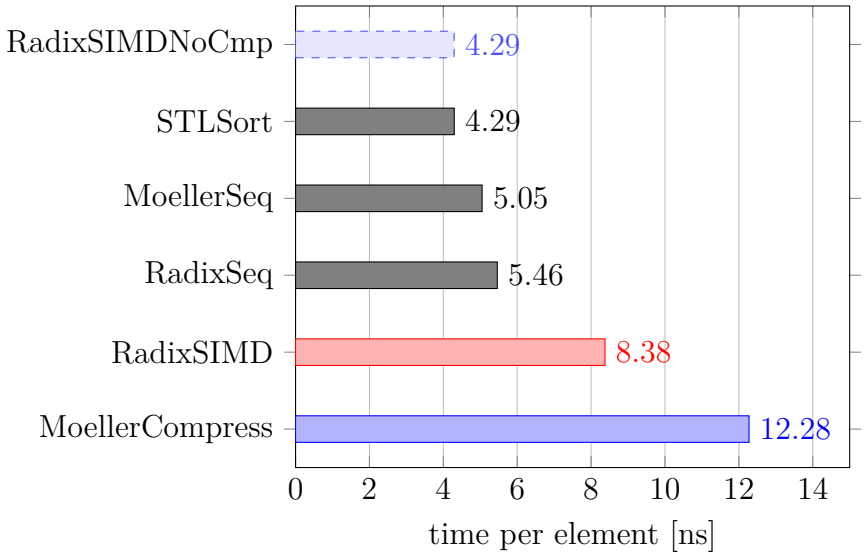


Figure B.30: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float-int32` and distribution ReverseSorted

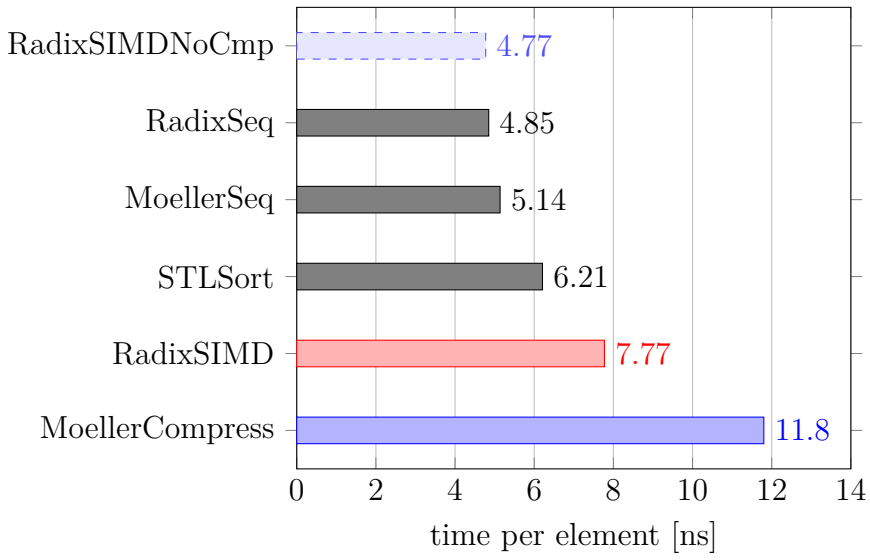


Figure B.31: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float-int32` and distribution AlmostSorted

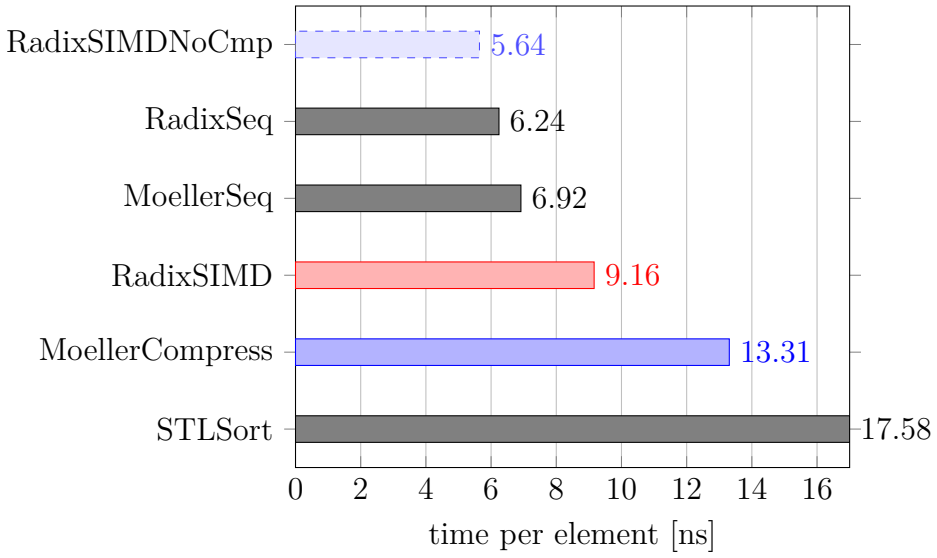


Figure B.32: Comparison of the runtime of different algorithms for 2^{18} elements with combination `float-int32` and distribution AlmostReverseSorted

B.5 `double`

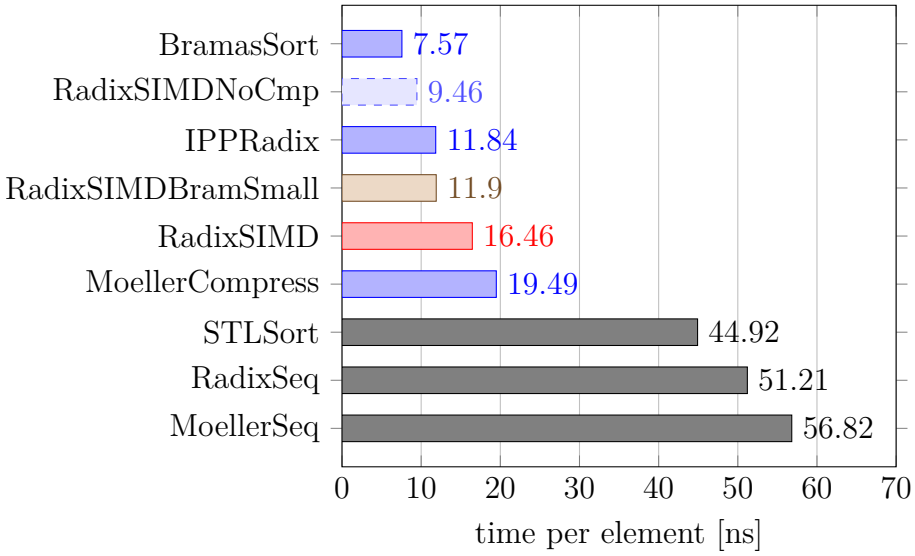


Figure B.33: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution Uniform

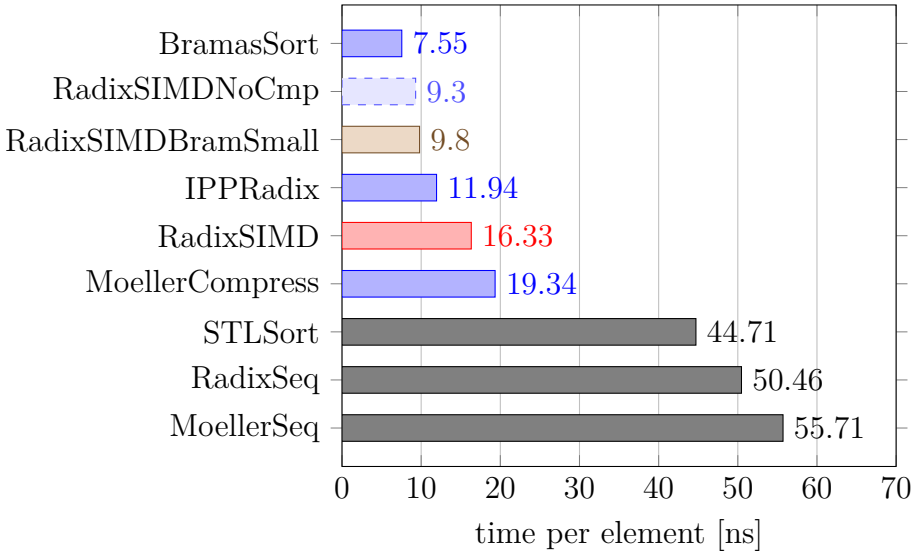


Figure B.34: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution Gaussian

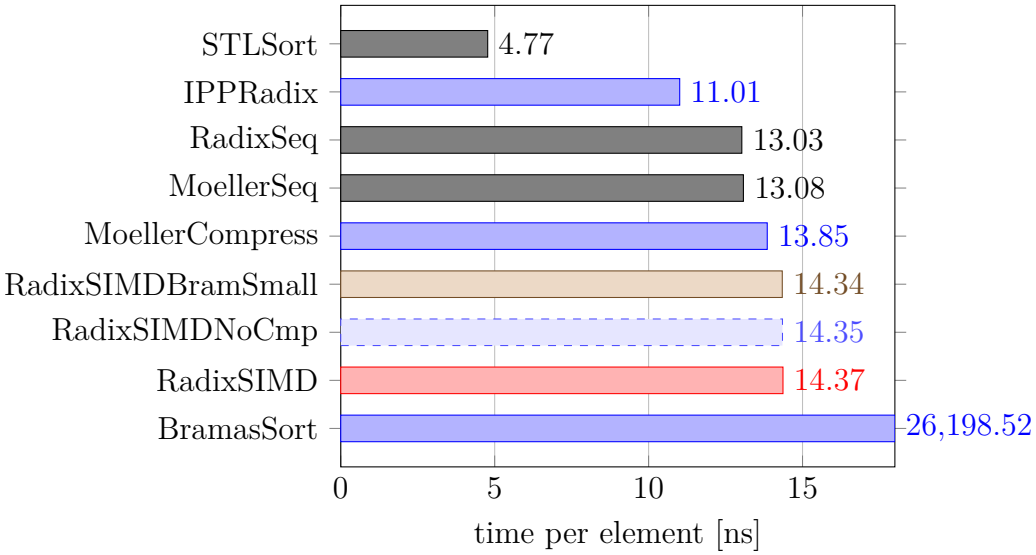


Figure B.35: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution Zero

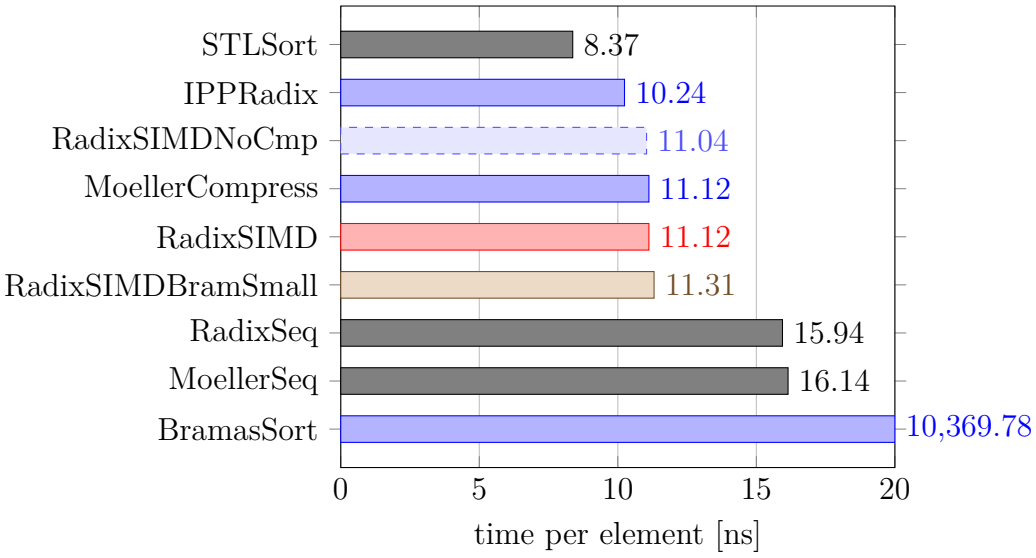


Figure B.36: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution ZeroOne

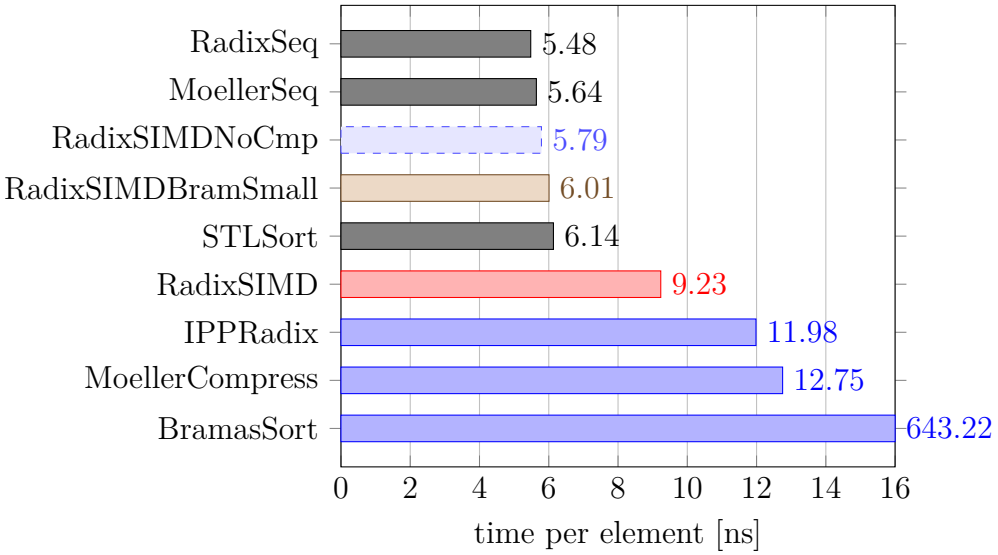


Figure B.37: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution Sorted

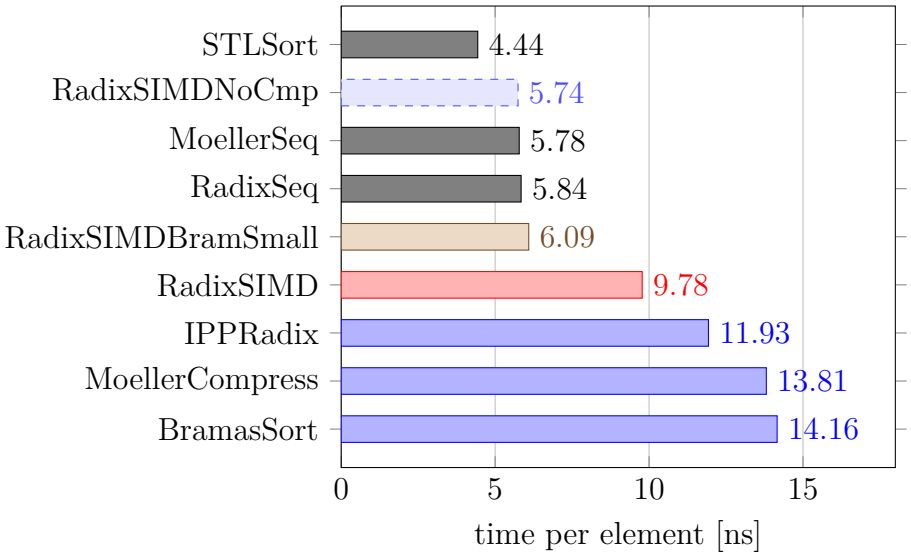


Figure B.38: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution ReverseSorted

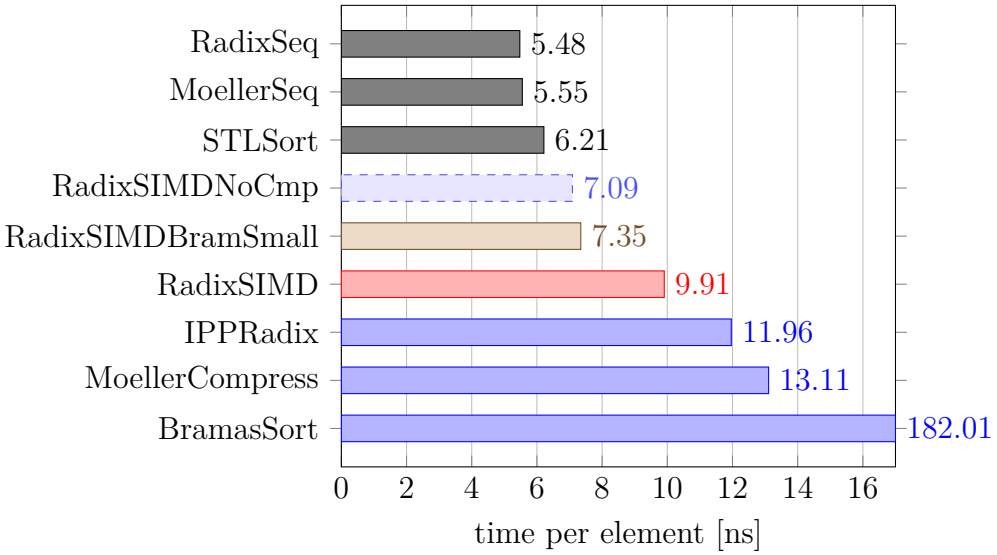


Figure B.39: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution AlmostSorted

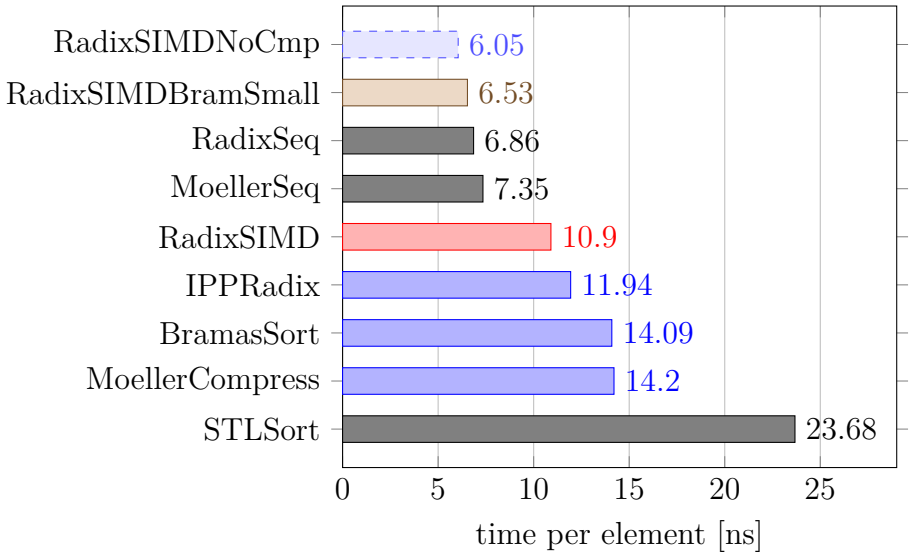


Figure B.40: Comparison of the runtime of different algorithms for 2^{18} elements with combination `double` and distribution AlmostReverseSorted

B.6 uint8

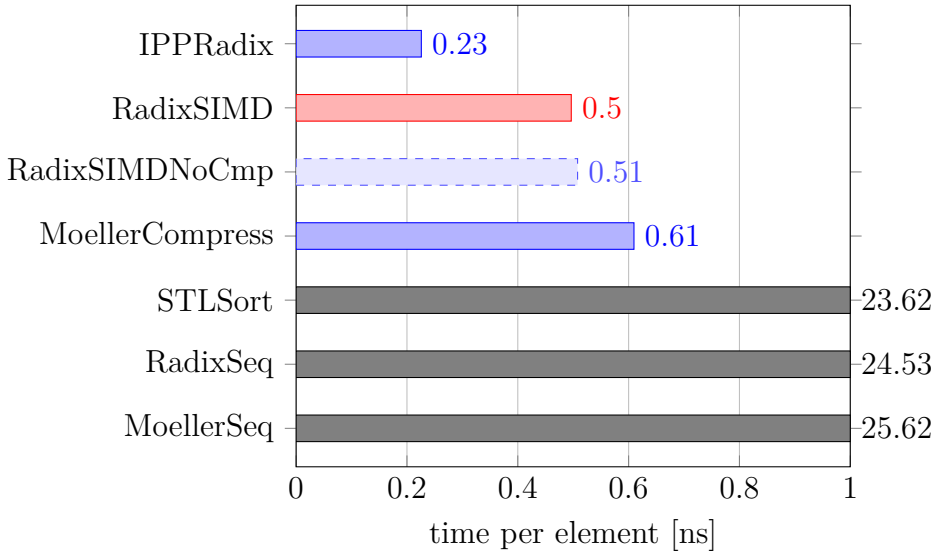


Figure B.41: Comparison of the runtime of different algorithms for 2^{18} elements with combination `uint8` and distribution Uniform

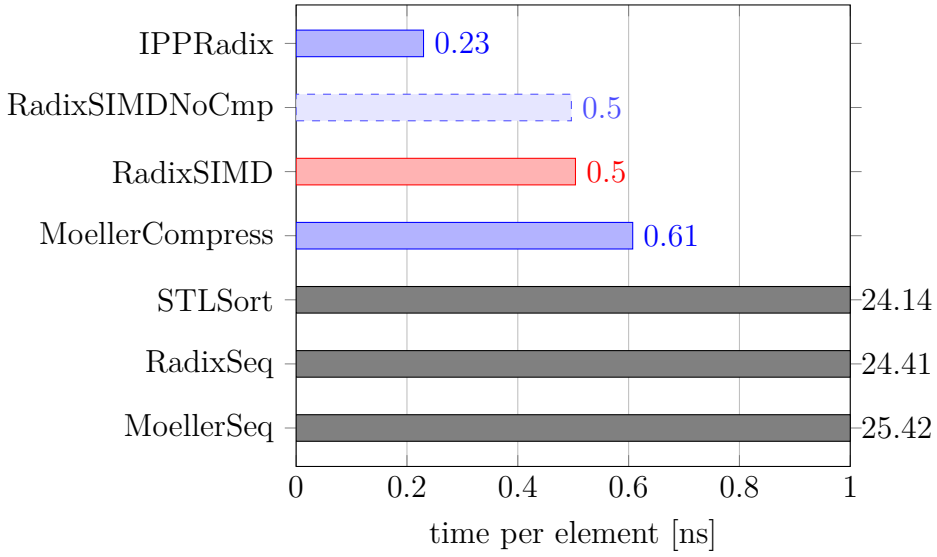


Figure B.42: Comparison of the runtime of different algorithms for 2^{18} elements with combination `uint8` and distribution Gaussian

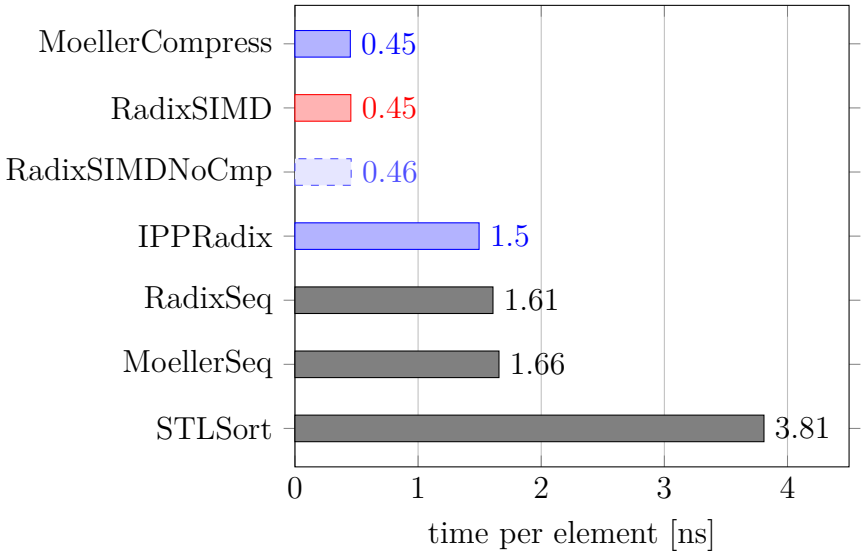


Figure B.43: Comparison of the runtime of different algorithms for 2^{18} elements with combination `uint8` and distribution Zero

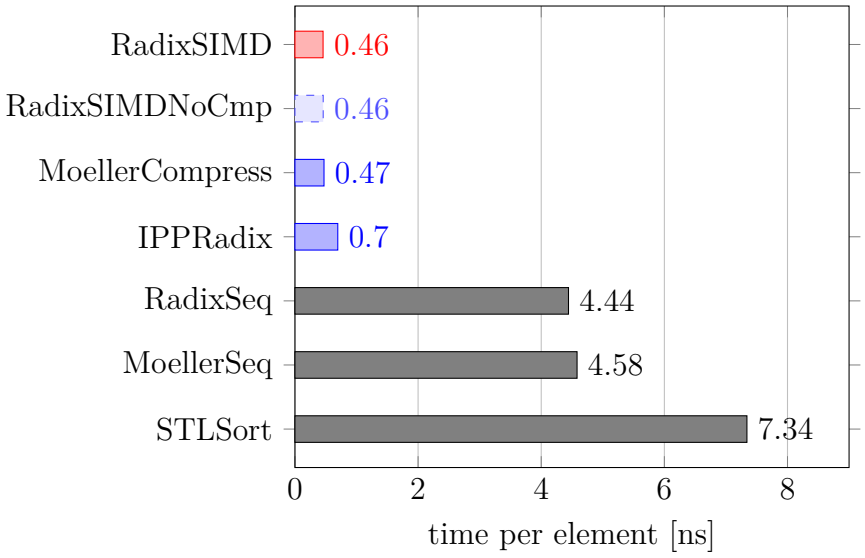


Figure B.44: Comparison of the runtime of different algorithms for 2^{18} elements with combination `uint8` and distribution ZeroOne

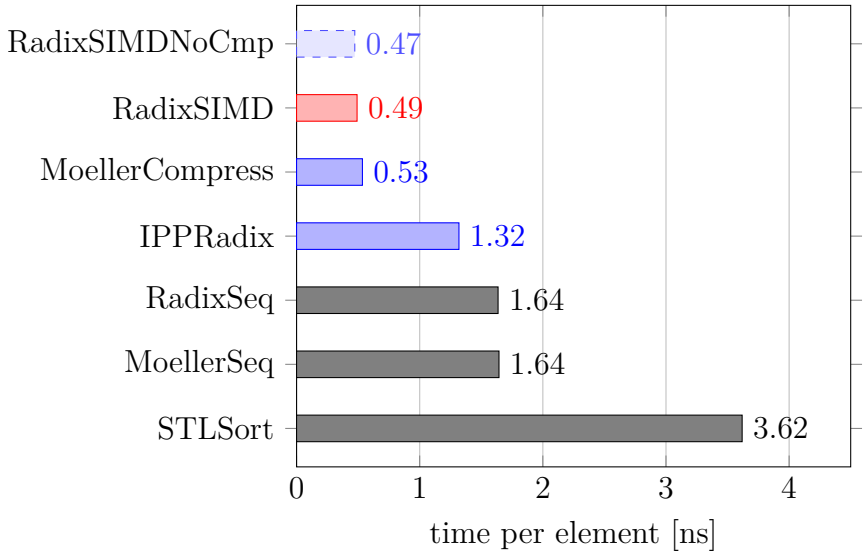


Figure B.45: Comparison of the runtime of different algorithms for 2^{18} elements with combination `uint8` and distribution Sorted

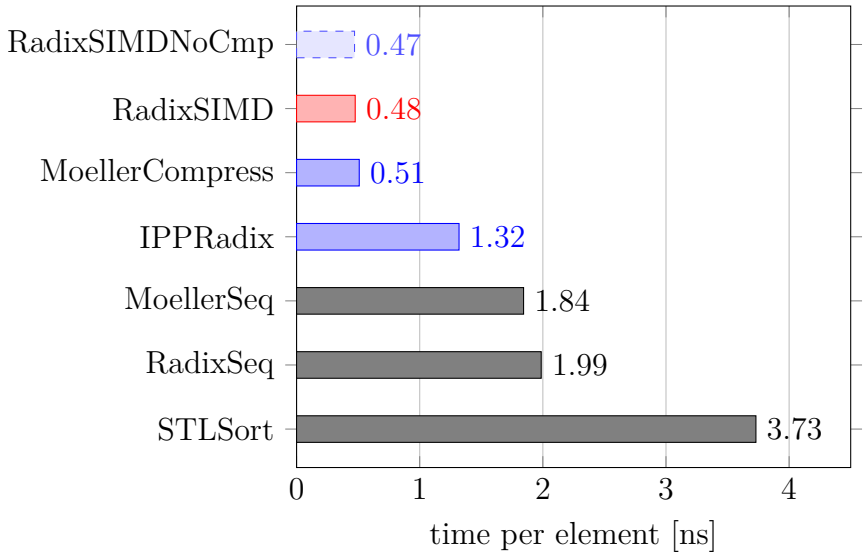


Figure B.46: Comparison of the runtime of different algorithms for 2^{18} elements with combination `uint8` and distribution ReverseSorted

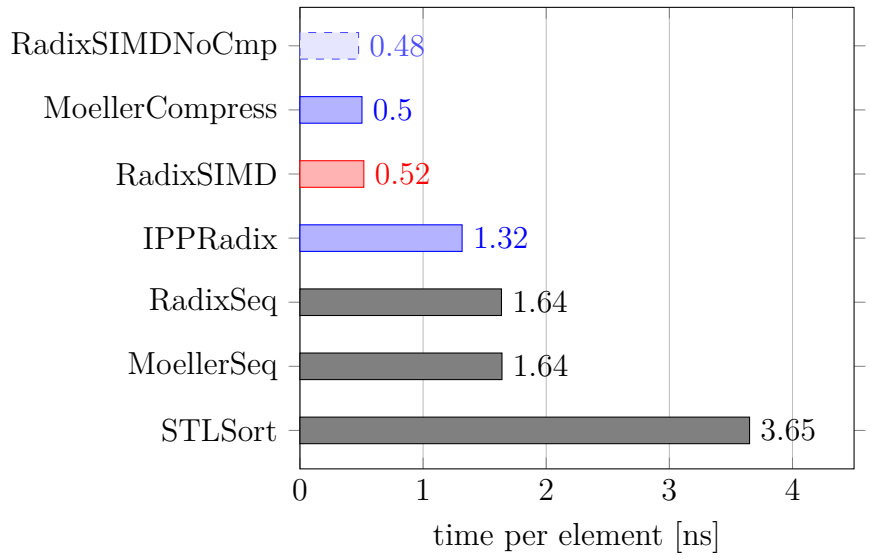


Figure B.47: Comparison of the runtime of different algorithms for 2^{18} elements with combination `uint8` and distribution AlmostSorted

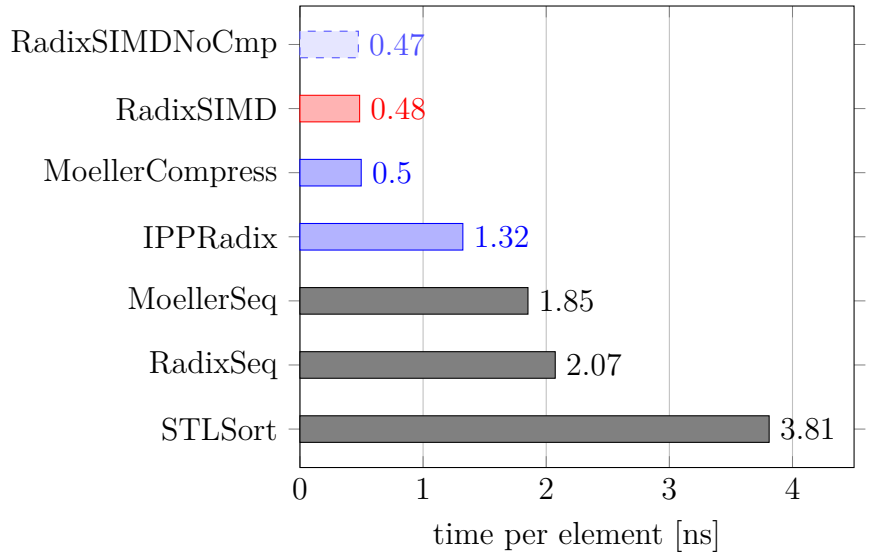


Figure B.48: Comparison of the runtime of different algorithms for 2^{18} elements with combination `uint8` and distribution AlmostReverseSorted

B.7 int16

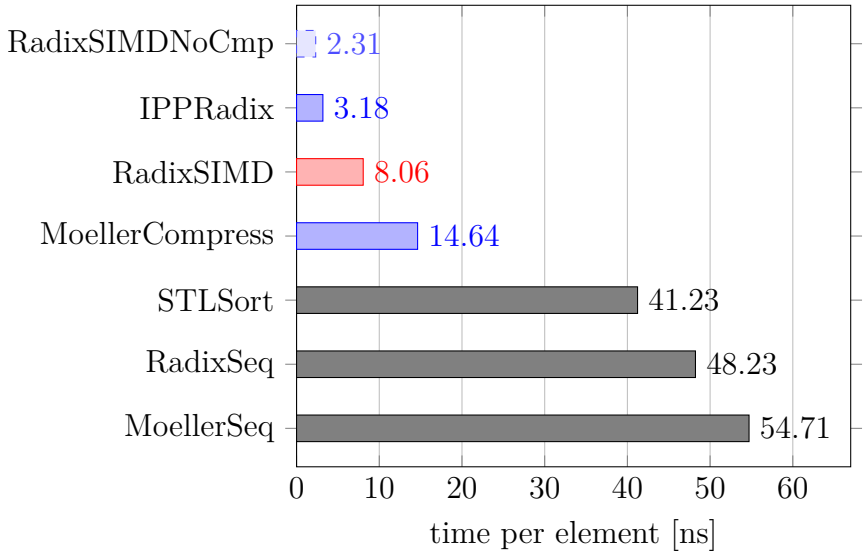


Figure B.49: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int16` and distribution Uniform

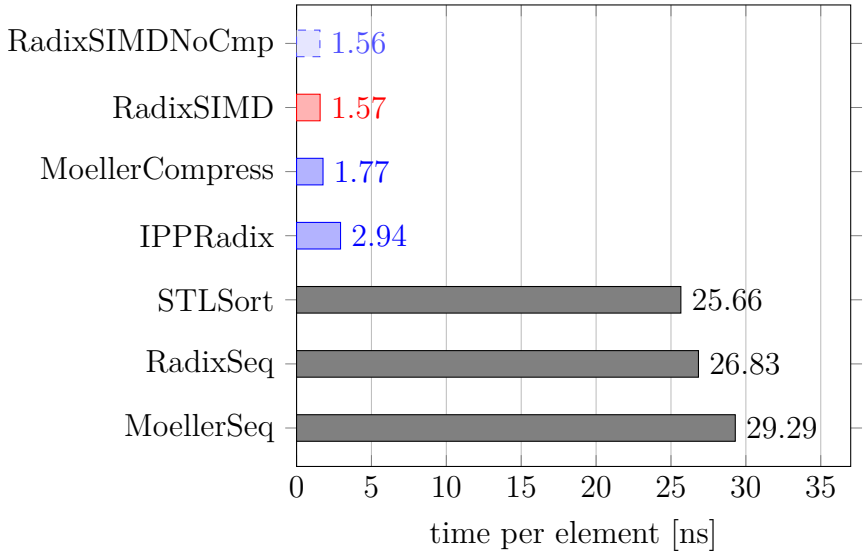


Figure B.50: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int16` and distribution Gaussian

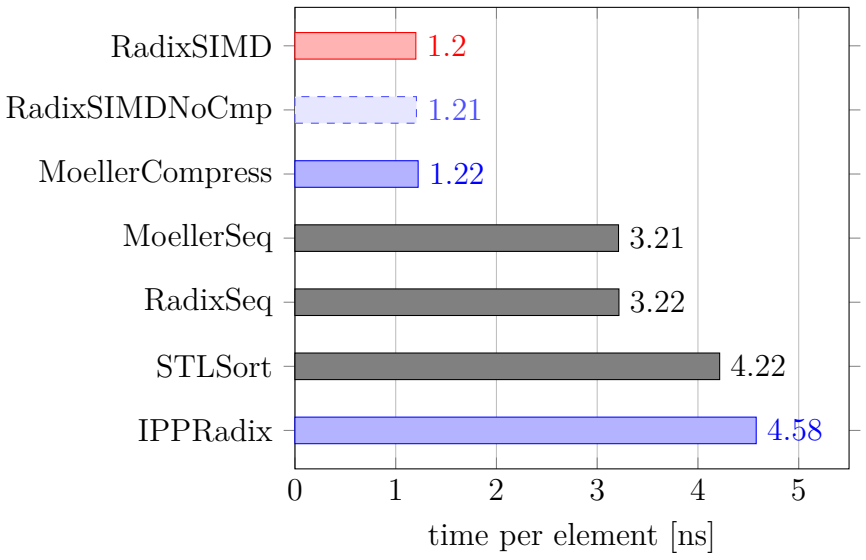


Figure B.51: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int16` and distribution Zero

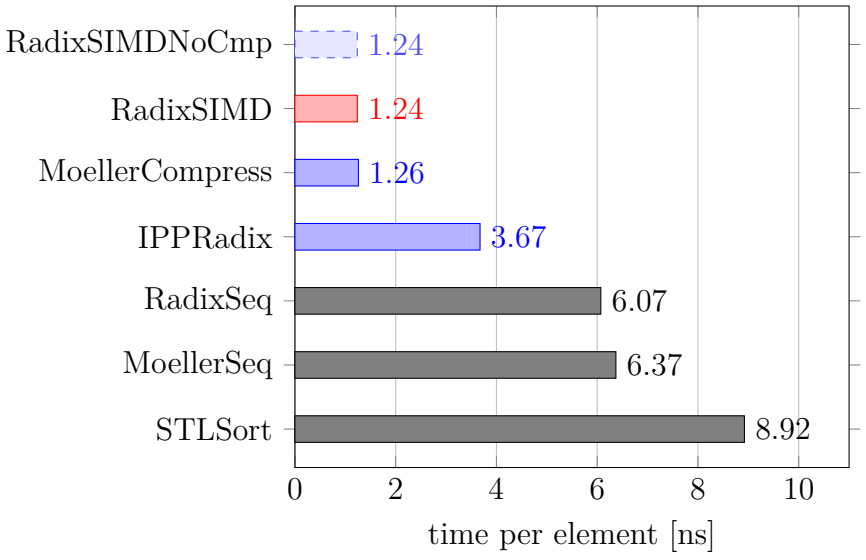


Figure B.52: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int16` and distribution ZeroOne

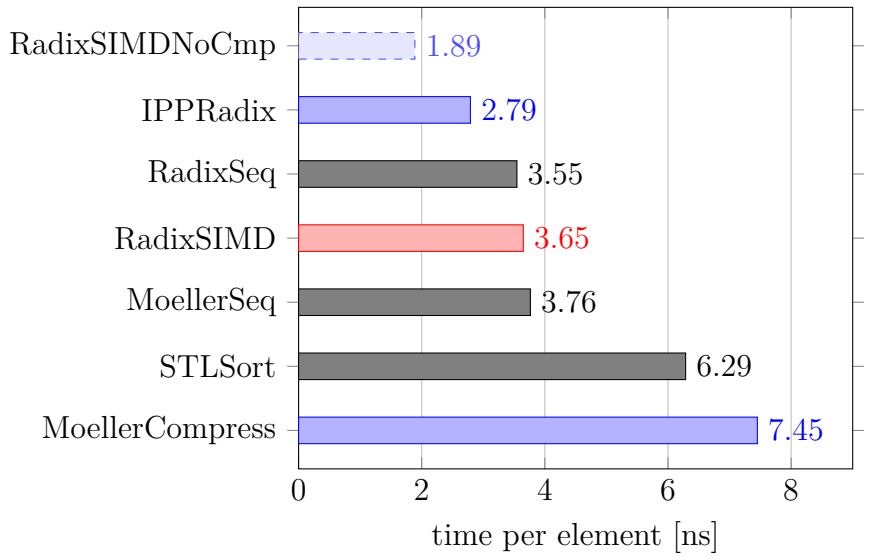


Figure B.53: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int16` and distribution Sorted

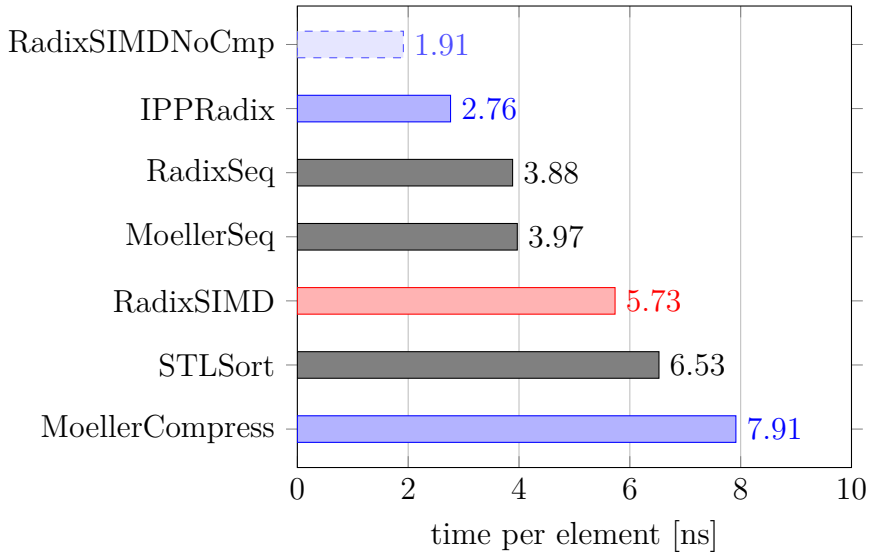


Figure B.54: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int16` and distribution ReverseSorted

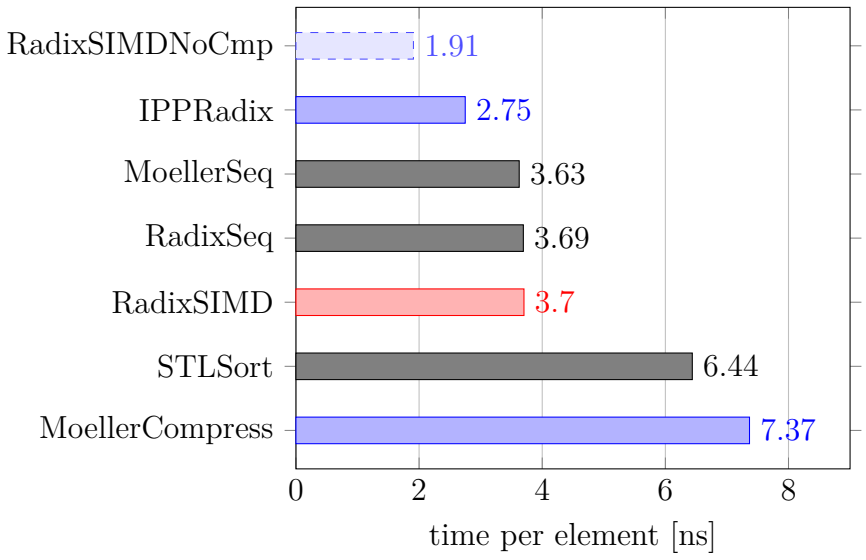


Figure B.55: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int16` and distribution AlmostSorted

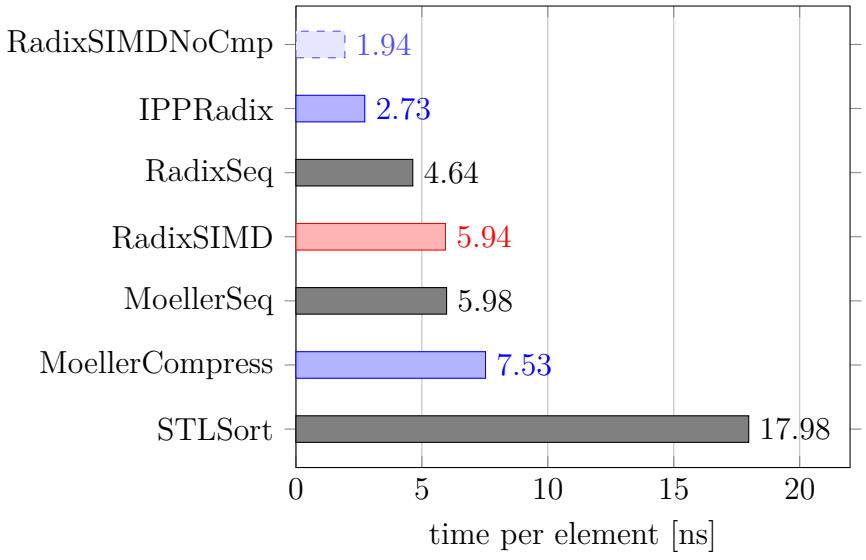


Figure B.56: Comparison of the runtime of different algorithms for 2^{18} elements with combination `int16` and distribution AlmostReverseSorted

Bibliography

Alcorn, P. *Intel Nukes Alder Lake's AVX-512 Support, Now Fuses It Off in Silicon*. Tom's Hardware, 2022. URL https://www.tomshardware.com/news/intel-nukes-alder-lake-avx-512-now-fuses-it-off-in-silicon. (Retrieved July 14, 2022).

Batcher, K. E. *Sorting Networks and Their Applications*. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery. ISBN 9781450378970. URL https://doi.org/10.1145/1468075.1468121.

Blacher, M. ; Giesen, J. ; Kühne, L. *Fast and Robust Vectorized In-Place Sorting of Primitive Types*. In *19th International Symposium on Experimental Algorithms (SEA 2021)*, volume 190 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:16. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, Juni 2021. URL https://elib.dlr.de/142624/.

Bramas, B. *A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake*. International Journal of Advanced Computer Science and Applications, 8(10), 2017. URL http://dx.doi.org/10.14569/IJACSA.2017.081044.

Ch, R. ; rasekaran. *Intel to unveil faster Pentium chip*. The Washington Post, 1997. URL https://www.washingtonpost.com/archive/business/1997/01/08/intel-to-unveil-faster-pentium-chip/9d6bdbd2-51b0-4a56-b24a-f4b1ee8b795e/. (Retrieved July 14, 2022).

Cormen, T. H. ; Leiserson, C. E. ; Rivest, R. L. ; Stein, C. *Introduction to Algorithms*. MIT Press, 3rd Edition, 2009. ISBN 9780262033848.

DavidWohlferd. *Reasons you should NOT use inline asm*. GCC Wiki, 2016. URL https://gcc.gnu.org/wiki/DontUseInlineAsm. (Retrieved July 14, 2022).

Flynn, M. *Very High-Speed Computing Systems*. Proceedings of the IEEE, 54:1901 – 1909, 01 1967. URL http://doi.org/10.1109/PROC.1966.5273.

Gueron, S. ; Krasnov, V. *Fast Quicksort Implementation Using AVX Instructions*. The Computer Journal, 59(1):83–90, 08 2015. ISSN 0010-4620. URL https://doi.org/10.1093/comjnl/bxv063.

Herf, M. *Radix Tricks*, 2001. URL http://stereopsis.com/radix.html. (Retrieved July 14, 2022).

Hord, R. M. *The Illiac IV: The First Supercomputer*. Springer-Verlag Berlin Heidelberg, 1982. ISBN 9783662103456.

Intel Corporation. *Intel® Integrated Performance Primitives. Part of Intel oneAPI Toolkits.*, a. URL https://software.intel.com/en-us/intel-ipp.

Intel Corporation. *Intel® Intrinsics Guide*, b. URL https://software.intel.com/sites/landingpage/IntrinsicsGuide/. (Retrieved July 14, 2022).

Intel Corporation. *How to Manipulate Data Structure to Optimize Memory Use on 32-Bit Intel® Architecture*, 2012. URL https://www.intel.com/content/www/us/en/developer/articles/technical/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.html. (Retrieved July 14, 2022).

Intel Corporation. *Intel® AVX-512 Instructions*, 2013. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html. (Retrieved July 14, 2022).

Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2*, 2022. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. (Retrieved July 14, 2022).

Knuth, D. E. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. ISBN 0201896850.

Levin, S. A. *A fully vectorized quicksort*. Parallel Computing, 16(2): 369–373, 1990. ISSN 0167-8191. URL https://doi.org/10.1016/0167-8191(90)90074-J.

Möller, R. *Design of a low-level C++ template SIMD library*. Technical report, Bielefeld University, Faculty of Technology, Computer Engineering Group, 2016. URL https://www.ti.uni-bielefeld.de/html/people/moeller/DOCUMENTS/templateSIMD.pdf.

Möller, R. *Bitwise MSB Radix Sort on AVX-512*. Technical report, Bielefeld University, Faculty of Technology, Computer Engineering Group, 2021. URL https://www.ti.uni-bielefeld.de/html/people/moeller/DOCUMENTS/simdRadixSort.pdf.

Siegel, H. J. *The Universality of Various Types of SIMD Machine Interconnection Networks*. SIGARCH Comput. Archit. News, 5(7):70–79, mar 1977. ISSN 0163-5964. URL https://doi.org/10.1145/633615.810655.

Skiena, S. S. *The Algorithm Design Manual*. Springer London, 2nd Edition, 2008. ISBN 978-1-84800-070-4. URL https://doi.org/10.1007/978-1-84800-070-4.

Tang, D. *CS241 – Lecture Notes: Sorting Algorithm*, 2012. URL https://www.cpp.edu/~ftang/courses/CS241/notes/sorting.htm. (Retrieved July 14, 2022).

Thiemicke, F. ; Blacher, M. ; Kühne, L. *Implementation of a vectorized Quicksort using AVX-512 intrinsics*, August 2021. URL https://elib.dlr.de/145402/.

Vandevoorde, D. ; Josuttis, N. M. ; Gregor, D. *C++ Templates - The Complete Guide, 2nd Edition*. Addison-Wesley, 2017.

Wikipedia contributors. *Advanced Vector Extensions — Wikipedia, The Free Encyclopedia*, 2022a. URL https://en.wikipedia.org/w/index.php?title=Advanced_Vector_Extensions&oldid=1097298131. (Retrieved July 14, 2022).

Wikipedia contributors. *MMX (instruction set) — Wikipedia, The Free Encyclopedia*, 2022b. URL https://en.wikipedia.org/w/index.php?title=MMX_(instruction_set)&oldid=1067835148. (Retrieved July 14, 2022).

Wikipedia contributors. *Sort (C++) — Wikipedia, The Free Encyclopedia*, 2022c. URL https://en.wikipedia.org/w/index.php?title=Sort_(C%2B%2B)&oldid=1070273046. (Retrieved July 14, 2022).

Wikipedia contributors. *Sorting network — Wikipedia, The Free Encyclopedia*, 2022d. URL https://en.wikipedia.org/w/index.php?title=Sorting_network&oldid=1063325047. (Retrieved July 14, 2022).

Wikipedia contributors. *Streaming SIMD Extensions — Wikipedia, The Free Encyclopedia*, 2022e. URL https://en.wikipedia.org/w/index.php?title=Streaming_SIMD_Extensions&oldid=1088269872. (Retrieved July 14, 2022).

Zagha, M. ; Blelloch, G. E. *Radix Sort for Vector Multiprocessors*. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, page 712–721, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897914597. URL https://doi.org/10.1145/125826.126164.