Fast and Robust Vectorized In-Place Sorting of Primitive Types

Mark Blacher
Institute for Theoretical Computer Science, Friedrich Schiller Universität Jena, Germany

Joachim Giesen
Institute for Theoretical Computer Science, Friedrich Schiller Universität Jena, Germany

Lars Kühne
German Aerospace Center (DLR), Jena, Germany

Abstract

Modern CPUs provide *single instruction-multiple data* (SIMD) instructions. SIMD instructions process several elements of a primitive data type simultaneously in fixed-size vectors. Classical sorting algorithms are not directly expressible in SIMD instructions. Accelerating sorting algorithms with SIMD instruction is therefore a creative endeavor. A promising approach for sorting with SIMD instructions is to use sorting networks for small arrays and Quicksort for large arrays. In this paper we improve vectorization techniques for sorting networks and Quicksort. In particular, we show how to use the full capacity of vector registers in sorting networks and how to make vectorized Quicksort robust with respect to different key distributions. To demonstrate the performance of our techniques we implement an in-place hybrid sorting algorithm for the data type `int` with AVX2 intrinsics. Our implementation is at least 30% faster than state-of-the-art high-performance sorting alternatives.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases Quicksort, Sorting Networks, Vectorization, SIMD, AVX2

Digital Object Identifier 10.4230/LIPIcs.SEA.2021.3

Supplementary Material *Software (Source Code)*: `https://github.com/simd-sorting/fast-and-robust` archived at `swh:1:dir:945e0841401092b83647202f46e8a60b084619f9`

Funding This work has been supported by the German Research Foundation (DFG) grant GI-711/5-1 within the Priority Program 1736 Algorithms for Big Data.

1 Introduction

Sorting is a fundamental problem in computer science, since sorting algorithms are part of numerous applications in both commercial and scientific environments. Among others, sorting plays an important role in combinatorial optimization [28], astrophysics [29], molecular dynamics [15], linguistics [39], genomics [33] and, weather forecasting [36]. Sorting is used in databases for building indices [30], in statistics software for estimating distributions [13], and in object recognition for computing convex hulls [2]. Sorting enables binary search, simplifies problems such as checking the uniqueness of elements in an array, or finding the closest pair of points in the plane [10]. Faster sorting implementations can thus reduce the computing time in a wide area of applications.

Some of the performance-critical applications mentioned above extensively sort arrays of numbers [36], that is, primitive types like `int` or `float`. High-performance libraries like Intel Performance Primitives (IPP) [21] or Boost.Sort [35] therefore offer fast sorting procedures for primitive types. But, these implementations do not take advantage of the available vector instruction sets, which are an integral part of modern CPUs. In computational domains such as linear algebra [4, 14], image processing [25], or cryptography [23] vector instructions are ubiquitous, however, vector instructions are rarely used in practice for sorting. One reason is that the vectorization of sorting algorithms is cumbersome, since existing sorting algorithms

cannot be directly expressed in SIMD instructions. Another reason is that current vectorized sorting algorithms often cannot outperform an optimized radix sort. Furthermore, current vectorized Quicksort implementations [5, 16] are not robust to different key distributions.

The idea of using vector instructions for sorting is not new. The field of vectorized sorting has been explored since the 1970s, first on vector supercomputers with their specialized instruction sets and later on modern architectures (see our discussion of related work below). A large part of the work on vectorized sorting algorithms has been done on old hardware from today's point of view. Since then, SIMD instruction sets continued to evolve and vector registers became wider. Also, new vectorization techniques such as vectorized in-place partitioning in Quicksort [5] have been discovered. Due to these changes, a new look at vectorized sorting algorithms seems worthwhile.

Related Work. Vectorized sorting is a topic that has been extensively researched on vector supercomputers [31, 37, 38, 41]. However, the knowledge gained there cannot be transferred uncritically to modern hardware. For example, Zagha and Blelloch [41] vectorize Radix Sort on the vector supercomputer CRAY Y-MP. Compared to a vectorized hybrid algorithm of Quicksort and Odd-Even Transposition Sort [31], they achieve speedup factors of three to five with their Radix Sort. The speedup is partly due to the fact that the CRAY Y-MP has no hardware prefetcher, which means accessing data elements in RAM in random order takes the same time as accessing them sequentially.

Studies on vectorized sorting, which take into account modern hardware, that is, vector instructions, instruction-level parallelism, CPU caches, and multicore parallelism, focus primarily on the vectorization of Mergesort [8, 19, 20]. Essentially, a vectorized Mergesort is a hybrid of merging networks and the usual Mergesort or some multiway variant of it. Despite efficient utilization of the underlying hardware, single-threaded vectorized Mergesort variants on mainstream CPUs do not achieve the speed of a hardware optimized Radix Sort such as Intel's platform-aware Radix Sort (IPP Radix Sort) [21]. For special hardware like the Xeon Phi 7210 processor, which features AVX-512 instructions and high bandwidth memory, a fast vectorized Mergesort variant exists, but unfortunately without publicly available source code for testing [40].

Gueron and Krasnov [16] show that Quicksort can be vectorized on modern hardware using the AVX and AVX2 instruction sets. Their partitioning function requires $\mathcal{O}(n)$ additional memory. The pivot value is always the last element of the partition. After the vectorized compare operation with the pivot value, the elements in the vector register are shuffled using shuffle masks that are stored in a lookup table. Gueron and Krasnov miss the opportunity to sort even small arrays with vector instructions. For sub-partitions with fewer than 32 elements, they use insertion sort, similarly to the C++ Standard Template Library (STL). The running times of their vectorized Quicksort are higher than those of IPP radix sort when sorting randomly distributed 32-bit integers.

Using AVX-512 instructions, Bramas [5] designs a vectorized hybrid algorithm based on Quicksort and Bitonic Sort that outperforms IPP radix sort on random input data by a factor of 1.4. Using the new compress instructions in AVX-512, Bramas implements Quicksort's partitioning function without the use of pre-computed permutation masks. Compress instructions arrange elements contiguously in a vector according to a bitmask. Compress instructions are not new per se. Both Stone [38] and Levin [31] have already used compress instructions on vector supercomputers to implement Quicksort's partitioning function. The true contribution of Bramas is that he implements the partitioning function without additional memory. He does this by buffering the two SIMD vectors at the outer ends

of the array. The pivot value for partitioning the array is determined with the median-of-three strategy. For sorting small sub-partitions with less than or equal to 256 integers Bramas uses a branch-free vectorized Bitonic Sort.

Practically Important Features for Sorting. Among the sorting implementations mentioned above, the fastest are sort AVX-512 [5] and Intel's IPP radix sort [21]. In addition to speed, the following features are often equally important in practice: low memory usage, robustness against different key distributions, portability, and efficient parallelization. Table 1 summarizes the features of our and other sorting implementations for primitive types. In this paper we consider only the single-threaded case, since our goal is to show that *pure* vectorized sorting is indeed the fastest available option for sorting primitive types. Vectorization and parallelization are orthogonal concepts. Parallelization would make the results highly dependent on the parallelization technique. The efficient combination of our vectorization techniques with parallelization is a topic that we plan to explore in future work.

Table 1 Features of various high performance sorting implementations for primitive types. The state-of-the-art general purpose sorting algorithm IPS^{4}o [1] is included here for comparison, although it is not optimized for sorting primitive types. We use the term in-place if the sorting algorithm requires no more than $\mathcal{O}(\log n)$ additional non-constant memory on average.

	this paper	sort AVX-512 [5]	IPP radix sort [21]	IPS^{4}o [1]
in-place	✓	✓	✗	✓
robust (distributions)	✓	✗	✓	✓
robust (duplicate keys)	✓	✗	✓	✓
portable ($\leq$ AVX2)	✓	✗	✓	✓[a]
fast ($n \leq 1000$)	✓	✓	✗	✗
fast ($n > 1000$)	✓	✓	✓	✗
parallelized efficiently	✗	✗	✗	✓

[a] Code does not compile on Windows.

Our Contributions. The main contributions of this paper are vectorization techniques for sorting networks and Quicksort that allow to design faster, more robust and memory efficient sorting algorithms for primitive types. In particular, we introduce the following techniques:

- For sorting small to medium-sized arrays we show how to use the full capacity of vector registers in sorting networks.
- To implement the vectorized partitioning function of Quicksort in-place with AVX2 instructions, we combine the lookup table strategy of Gueron and Krasnov [16] with the in-place partitioning of Bramas [5].
- To make vectorized Quicksort robust to different key distributions, we design an alternative pivot selection strategy that is used for unbalanced sub-arrays during partitioning. With this strategy, which is related to Radix Exchange Sort [7, 11], the worst case running time of our Quicksort is $\mathcal{O}(kn)$, where k is the number of bits in the primitive type.

Based on these techniques we implement a sorting algorithm with AVX2 vector instructions that outperforms general purpose and competing high-performance sorting implementations for primitive types. The source code of our implementation is available at `https://github.com/simd-sorting/fast-and-robust`. Additionaly, we provide an efficiently vectorized implementation of Quickselect [36], which uses the same vectorization techniques to find the kth smallest element in an unordered array.

2 Preliminaries

Vector Instructions. Vector instructions (SIMD instructions) in the x86 instruction set exist since 1997 [6]. The register width was steadily increased since then. Larger registers allow more elements of a primitive type to be processed simultaneously. Starting with 64 bits in MMX (Multi Media Extension), the register width increased to 128 bits for SSE (Streaming SIMD Extensions) and again doubled to 256 bits in AVX (Advanced Vector Extensions). Current CPUs for high-performance computing have 512-bit registers and support the AVX-512 instruction set [22]. Modern mainstream CPUs, however, support at most AVX2. Like its predecessor AVX, AVX2 uses vectors of up to 256 bits, but has a larger instruction set. Since we want our implementation to run on a large variety of available CPUs we use AVX2 and not AVX-512 to implement our ideas. Instead of writing the vectorized part of the algorithms in assembly language, we use intrinsic functions. Intrinsic functions have the advantage that explicit assignments of registers are omitted. Intrinsic functions are also more portable between different compilers and operating systems.

Latency and Throughput. Besides the vector width, the two metrics latency and throughput provide further information about the performance of an instruction. Latency is the number of cycles that an instruction takes until its result is available for another instruction. Throughput, on the other hand, indicates how many cycles it takes on average until an identical independent instruction can begin its execution [12]. If the throughput of an instruction is smaller than the latency, executing several identical independent instructions in sequence can lower the accumulated latency of the computation, because instructions can begin before the results of previous instructions are available. We exploit this heavily by simulating larger vectors to increase instruction-level parallelism of our implementation.

3 The Algorithm

In this section we describe the design and implementation of our algorithm for sorting arrays of data type `int` (32-bit signed integer). Most state-of-the-art sorting implementations combine several sorting algorithms for achieving good performance on different array sizes and key distributions. Here, we also take this approach and combine vectorized Quicksort with vectorized sorting networks into a fast and robust in-place sorting algorithm. The overall design of our algorithm is shown in the simplified C++ Listing 1.

The remainder of this section is organized as follows: In Subsection 3.1 we discuss our techniques to vectorize sorting networks for sorting small and medium-sized arrays, respectively. In Subsections 3.2 and 3.3 we describe our combined pivot strategy that makes Quicksort robust with respect to different key distributions and also makes it robust with respect to duplicate keys. As will be discussed in Subsection 3.4, our algorithm is in-place in the sense that the only non-constant memory overhead comes from the recursive function calls. This Section finishes with a brief worst and average running time analysis in Subsection 3.5.

3.1 Sorting Networks

The building blocks of sorting networks are **Compare-and-exchange (COEX) modules**. A COEX module consists of two inputs and two outputs. Each input receives a value. The two values are sorted in the module and transferred to the two outputs. In our representation of a COEX module, the upper output track contains the smaller and the lower output track the larger value. See Figure 1 for different options of visualizing a COEX module.

Listing 1 Our vectorized Quicksort algorithm partitions arrays with vector instructions until the sub-arrays become small enough ($\leq$ 512 elements) and then switches to vectorized sorting networks. For achieving higher performance, two different techniques are used to vectorize sorting networks (not shown in Listing 1). One for small arrays ($n < 128$) and one for medium-sized arrays ($128 \leq n \leq 512$). To detect sub-arrays in which all elements are equal, the *smallest* and *largest* values in the array are computed during the partitioning phase. We combine two pivot strategies to make our Quicksort robust with respect to various distributions. After partitioning, we do not set the pivot element to its correct position in the sorted array. This allows us to use values as pivots that are not contained in the array.

```
void qs_core(int *arr, int left, int right, bool choose_avg = false, int avg = 0) {
  if (right - left <= 512) { // sorting networks for array sizes <= 512
    sort_with_sorting_networks(arr + left, right - left + 1);
    return;
  }
  int pivot = choose_avg ? avg : get_pivot_median_medians(arr, left, right);
  int smallest = INT32_MAX; // smallest value in the array after partitioning
  int largest = INT32_MIN;  // largest value in the array after partitioning
  int bound = partition(arr, left, right + 1, pivot, &smallest, &largest);
  // the ratio of the size of the smaller partition to the size of the array
  double ratio = min(right - (bound - 1), bound - left) / double(right - left + 1);
  if (ratio < 0.2) // if unbalanced sub-arrays, change pivot strategy
    choose_avg = !choose_avg;
  if (pivot != smallest) // if values not identical in left sub-array, recurse
    qs_core(arr, left, bound - 1, choose_avg, average(smallest, pivot));
  if (pivot + 1 != largest) // if values not identical in right sub-array, recurse
    qs_core(arr, bound, right, choose_avg, average(largest, pivot));
}
```

In a sorting network, the COEX modules are combined in such a way that they always sort a fixed-length sequence of values. The amount of modules and their execution order are fixed for a network and do not depend on the input values. Sorting networks are therefore data-oblivious algorithms. A sorting network for eight elements is shown in Figure 2. The number of parallel steps and the number of COEX modules are the two key parameters that are used to optimize sorting networks. A sorting network with the minimum number of COEX modules does not necessarily have the fewest parallel steps and vice versa [9].

There are **regular** [3, 34] and **irregular sorting networks** [9, 27]. For a regular sorting network, an algorithmic rule exists that generates the network and can also generate larger networks of the same type. For irregular sorting networks it is not known how the construction of these networks can be generalized to create larger networks [26].

(a) Detailed. **(b)** Simplified.

Figure 1 Compare-and-exchange module visualizations. To represent modules in sorting networks we use the simplified visualization depicted in (b).

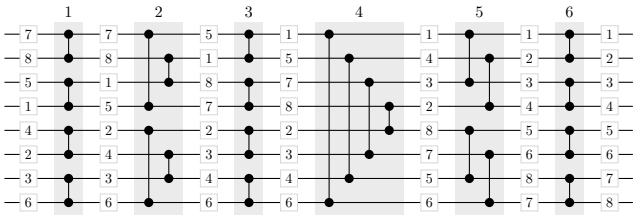


Figure 2 Bitonic sorting network for eight elements (24 modules, six parallel steps). The input comes from the left and runs to the right through the network. The numbers above the gray boxes index the **parallel steps**. Each parallel step is characterized by the fact that there are no dependencies between individual modules. All COEX modules within a gray box can thus be executed simultaneously.

In the following two paragraphs we show how to efficiently vectorize sorting networks for sorting small ($n < 128$) and medium-sized ($128 \leq n \leq 512$) arrays. We define $n = 128$ as the boundary between small and medium-sized arrays, since we use different types of sorting networks. The technique we use for sorting small arrays is especially efficient for Bitonic Sort. Therefore, we apply it to small Bitonic sorting networks. In contrast, the technique we use for sorting medium-sized arrays can efficiently vectorize irregular sorting networks that have a minimum amount of COEX modules.

Sorting Small Arrays With Sorting Networks. To efficiently vectorize small Bitonic sorting networks, it is important to distinguish between modules and nodes. A COEX module schematically consists of two nodes and a vertical connecting line between them. (see Figure 1b). The elements passing through the two nodes are compared within a module. Here, we represent a COEX module as a tuple of two comma separated numbers. For example, $(1, 2)$ stands for a module in which the element at the first node is compared to the second. The numbers in brackets represent the nodes to be compared and not the values to be sorted. Nodes correspond only to the positions of elements in a sorting network.

The aim of vectorizing sorting networks is to execute as many modules as possible at once. To perform a vectorized COEX operation, the two nodes of a module must be at the same index in two different vectors. By computing the pairwise minimum and maximum between these two vectors, the vectorized COEX operation is executed. Before we perform a vectorized COEX operation, we use shuffle instructions to place the two nodes of a module under the same index in the two vectors. We use two different shuffle instructions. The first shuffles the elements only within one vector. The second called shuffle*, receives two vectors as input and mixes them according to a mask to create a new vector.

In our approach nodes are considered only virtually, while the values to be sorted are actually stored in the vector registers. When a COEX operation is executed, two types of exchanges are possible. On the one hand, values can be exchanged between the vector registers, on the other hand, nodes within a module can also be swapped. Since the nodes are virtual, the swaps of the nodes are also only virtual, and thus do not consume CPU cycles. Here, in order to simplify the presentation of our approach, the capacity of vectors is limited to four elements. Figure 3 shows our technique for sorting eight elements with the bitonic sorting network from Figure 2.

Our technique distributes nodes within modules to different vectors and thus uses the full capacity of vector registers. Bramas [5] uses, unlike our approach, only half the capacity of a vector register because the two nodes of a module are held within one vector. Furthermore, Bramas uses only permutation instructions for implementing sorting networks. We use shuffle instructions with lower latency wherever possible.

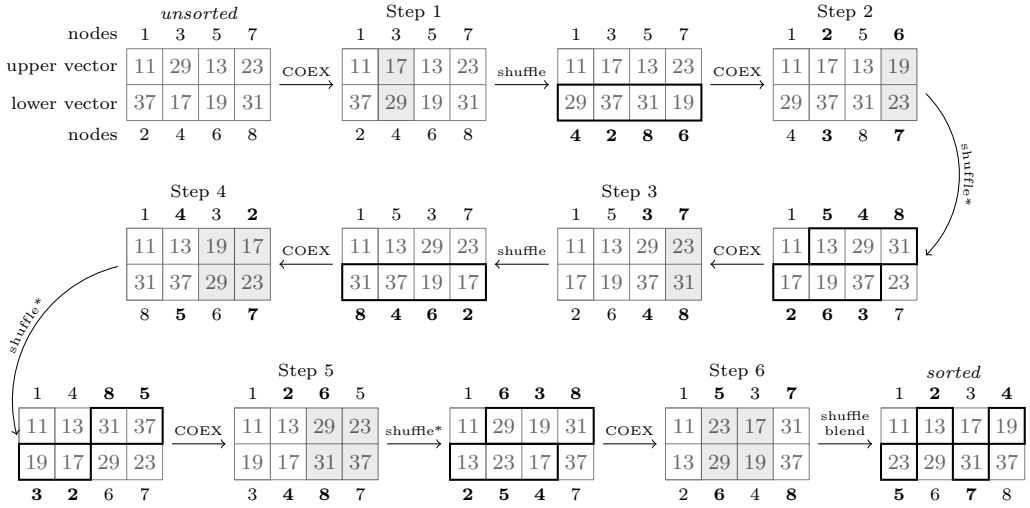


Figure 3 Bitonic Sort of 8 elements with two vectors. Each vector has a capacity of 4 elements. We call the six parallel steps in which the COEX operations are executed Step 1, ..., Step 6. In the first parallel step, Bitonic Sort has the four COEX modules $\{(1,2),(3,4),(5,6),(7,8)\}$. The association of the nodes and indices in the vectors at the beginning of the sorting operation is freely selectable. We start under the assumption that the upper vector represents the Nodes $1,3,5,7$ and the lower vector represents the Nodes $2,4,6,8$. This initial distribution of nodes prevents unnecessary shuffles of elements before executing the first parallel step because the nodes within modules are at the same index in the two vectors. In the vectorized COEX operation of the first parallel step only the values 29 and 17 are exchanged, which is indicated by the gray box in Step 1. The second parallel step requires the modules $\{(1,4),(2,3),(5,8),(6,7)\}$. Before executing Step 2, the lower vector is shuffled so that the nodes to be compared face each other. The COEX operation in Step 2 exchanges the values 23 and 19. The nodes within two modules must also be swapped (3 with 2, and 7 with 6). These swaps are necessary, because the minimum of two nodes after executing the vectorized COEX operation is always in the upper vector and the minimum is assigned to the node with the smaller index. The next four parallel steps are performed in a similar way. After executing Step 6, the upper vector always contains the elements at the Nodes 1, 5, 3 and 7, and, the lower vector contains the elements at the Nodes 2, 6, 4 and 8, respectively. The elements are sorted, but must be rearranged according to the indices of the nodes. After rearranging the elements with two shuffle and two blend instructions, the eight elements are properly sorted.

Sorting Medium-Sized Arrays With Sorting Networks. For sorting medium-sized arrays ($128 \leq n \leq 512$) we interpret the array as a matrix in row-major order, where the number of columns corresponds to the number of elements that can be placed in a vector. In an AVX2 vector there is space for eight values of data type `int`. An array with 128 values of datatype `int` thus corresponds to a matrix with eight columns and 16 rows.

To sort column-wise, the vectorized COEX operation is sufficient. Only pairwise minima and maxima between the vectors are computed, and no permutations or shuffles of the elements within the vectors are required for sorting the columns of a matrix. During a vectorized COEX operation, the same COEX module is executed in all columns. The number of vectorized COEX operations therefore depends on the number of modules in a sorting network. The advantage of sorting the elements inside columns is that even sorting networks with an irregular distribution of modules can be used, since each vectorized COEX operation uses only one module of the sorting network.

Sorting networks with a minimum number of COEX modules are particularly suitable for sorting the values in columns. To sort eight columns each containing 16 values of data type `int`, we use Green's irregular sorting network [27], which consists of only 60 COEX

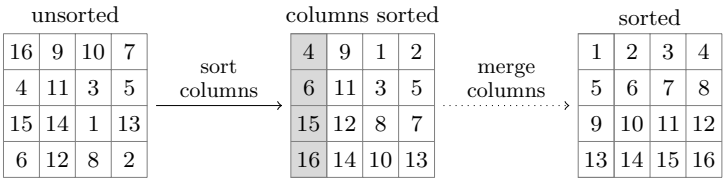


Figure 4 Sorting medium-sized arrays. We interpret the array as a matrix in row-major order. For sorting columns we use sorting networks with a minimum number of COEX modules. To merge the sorted columns, we apply Bitonic Merge directly, without transposing the columns first.

modules. To fully sort the 128 values, the eight sorted columns must be merged. Instead of transposing the 16×8 matrix and merging the sorted rows with a Bitonic Merge network, we apply Bitonic Merge directly to the sorted columns of the matrix. The technique for merging sorted columns is similar to the technique we use for sorting small arrays. Before we execute a COEX operation, we shuffle or permute elements of vectors such that the nodes of each module are placed in different vectors at the same index. Figure 4 summarizes our approach for sorting medium-sized arrays with vector instructions.

3.2 Pivot Selection

To make our vectorized Quicksort robust with respect to various distributions, we combine two different pivot selection strategies and switch between them when one strategy leads to unbalanced sub-arrays (see Listing 1).

First Strategy. The first strategy determines the pivot with a heuristic similar to the median of medians. Without the necessity to place the pivot to its final position in the array, determining the median of medians becomes faster since the index of the pivot is not needed. With AVX2 vector instructions we can compute eight medians simultaneously. To compute eight times the median-of-nine, we gather 72 random elements from the array to be partitioned and store them in nine vectors. The 72 random indices for the nine gather instructions are computed with the random number generator xoroshiro128+ [26]. We implement xoroshiro128+ with vector instructions to speed up the computation of random numbers. The nine vectors with its 72 elements are interpreted as a 9×8 matrix in row-major order. We compute the medians column-wise with a median network [32], which we implement with min and max vector instructions. After applying a median network to the columns of the matrix, the fifth vector contains all eight medians. We sort the eight medians with vector instructions and choose as pivot the average of the fourth and fifth largest medians. If the resulting average is not an integer, we round down to the next smaller integer.

Second Strategy. If the vectorized median of medians heuristic leads to unbalanced sub-arrays, we switch to the second pivot strategy, where we choose the average of the *smallest* value and the *current pivot* as the pivot in the left sub-array, and, in the right sub-array, the average of the *largest* value and the *current pivot* as the pivot. The quadratic worst case of Quicksort is avoided, since choosing the average of the lower and upper bounds of the sub-array as pivot guarantees that the range of possible values is halved with each recursive call. The difference to Radix Exchange Sort [7, 11] in our second pivot strategy is that instead of bits, actual values are used to halve the range of possible values at each recursive call. If the second pivot strategy leads to unbalanced sub-arrays, the first pivot selection strategy is used in the next recursive call.

To realize the second pivot strategy, we always compute the *smallest* and the *largest* values in the array with min and max vector instructions when partitioning the array. We take advantage of instruction-level parallelism of modern processors to hide the execution time of the minimum and maximum computations. The latency of min and max vector instructions is only one cycle and the throughput is half a cycle [24], which means that their execution time can almost be hidden if they are called while waiting for the results of high latency instructions.

3.3 Many Duplicate Keys

The knowledge about the *smallest* and the *largest* values in an array also allows us to detect sub-arrays where all values are equal. If the *smallest* value in an array and the *current pivot* are identical, no further partitioning of the left sub-array is required, or, if the *largest* value and the *current pivot* + 1 in an array are identical, no further partitioning of the right sub-array is necessary. Applying these checks before partitioning sub-arrays makes our vectorized Quicksort robust against arrays with many duplicate keys.

3.4 In-place Partitioning

We start partitioning the array without vector instructions until the number of unpartitioned elements corresponds to a multiple of the number of elements in a vector. Next, we cache two vectors with unpartitioned elements from the outer ends of the array. This creates space to perform the partitioning in-place with Bramas' [5] technique. To partition a single vector we use, similar to Gueron and Krasnov [16], permutation masks that are stored in a lookup table. Only one permutation instruction is sufficient for partitioning a vector, since we fill, like Bramas, the array from left to right with values smaller than or equal to the pivot, and, from right to left with values greater than the pivot. After storing a partitioned vector on the left and right side of the array we choose the next vector to partition from the side of the array with fewer already partitioned elements. Overwriting of not yet partitioned values of the array is impossible, because there is space for storing a partitioned vector on each side of the array due to the two initially cached vectors. This space is only used up as soon as the vectors cached at the beginning are partitioned and stored in the array. See Appendix A for a worked example of our vectorized in-place partitioning algorithm.

To reduce the total latency of partitioning big arrays and thus increase instruction-level parallelism, we load 64 elements into eight vector registers, partition the eight vectors individually and save them back to the array. In order to realize the partitioning of big arrays in-place we cache 16 vectors instead of two, at the beginning of the partitioning.

3.5 Running Time Analysis

The running time of our vectorized sorting algorithm is determined by the two employed pivot strategies, namely, the (vectorized) median of the medians strategy and the strategy that is inspired by Radix Exchange Sort. While we can control the worst case behaviour of our vectorized sorting algorithm with the latter strategy, the median of medians strategy allows to control the average case behaviour. In particular, we can give the following guarantees.

▶ **Lemma 1.** *The worst case running time of our vectorized sorting algorithm is $\mathcal{O}(kn)$ where n is the number of keys and k is the number of bits per key.*

Proof. (Sketch) The claim follows since we always switch from the median of the medians pivot strategy to the second pivot strategy whenever the first strategy leads to unbalanced sub-arrays. The second pivot strategy uses the average of the smallest and the largest values in the array as pivot element. This is similar to Radix Exchange Sort that uses the most significant bit to subdivide the keys in an array. It follows that the worst case complexity of our algorithm when using the second pivot strategy is the same as for Radix Exchange Sort, namely $\mathcal{O}(kn)$. Hence, our algorithm runs in worst case time $\mathcal{O}(kn)$ if $k \geq \log n$. If $k < \log n$, then the array needs to contain duplicate keys that our algorithm detects such that the worst case running time still is in $\mathcal{O}(kn)$. ◀

▶ **Lemma 2.** *The average case running time of our vectorized sorting algorithm is $\mathcal{O}(n \log n)$, where the average is with respect to the uniform distribution over the set of permutations of n different keys.*

Proof. (Sketch) This follows immediately from the analogous results for Quicksort with median of the medians pivot strategy. Remember that we only switch to the second pivot strategy when the median of medians strategy leads to unbalanced sub-arrays. Hence, switching the pivot strategy cannot degrade the asymptotic average running time. ◀

For putting our running time analysis into perspective, note that our vectorized sorting algorithm is not a comparison based algorithm, since we use an arithmetic operation in the pivot strategy that is inspired by Radix Exchange Sort.

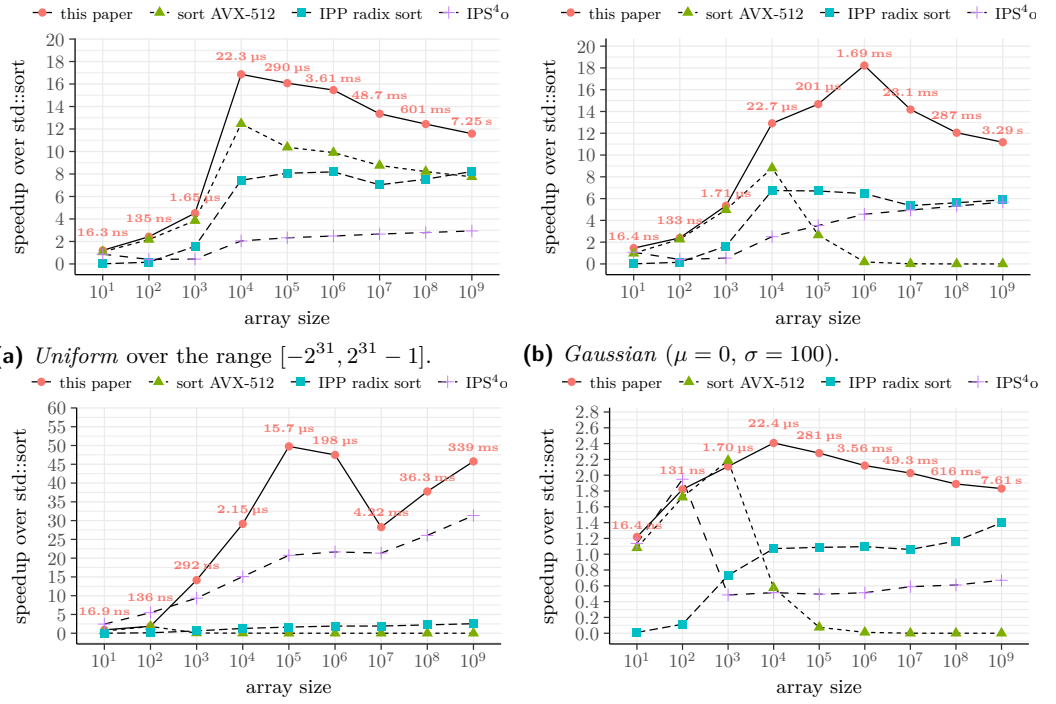
4 Experiments

Single-Threaded Performance. We compare the single-threaded performance of our implementation with its high-speed competitors, sort AVX-512 [5], Intel's platform-aware radix sort (IPP radix sort) [21], and IPS^{4}o [1]. IPP radix sort is generally accepted as a benchmark for high-performance sorting of primitive data types [5, 16, 40]. IPS^{4}o is one of the fastest general purpose sorting algorithms. We report the performance of each algorithm in terms of speedup factors over std::sort from the C++ STL for different array sizes and distributions of integers. For computing a single speedup factor we divide the running time of std::sort by the running time of the respective algorithm. In our measurements we consider array sizes in the range of 10^1 to 10^9, whereby the next larger array size contains ten times more elements than the previous one. To evaluate the speed and robustness of our sorting algorithm, we fill the arrays with integers according to the following four distributions:

(a) *Uniform.* Random integers in the range $[-2^{31}, 2^{31} - 1]$.
(b) *Gaussian.* Normally distributed random numbers with expected value $\mu = 0$ and standard deviation $\sigma = 100$. We round the numbers to the nearest integer, resulting in many duplicate values in large arrays.
(c) *Zero.* Every value is set to a constant: an input distribution with zero entropy [17].
(d) *Almost Sorted.* Array of size n with $\lfloor \frac{1}{2} \cdot 2^{\log_{10} n} \rfloor$ misplaced numbers. For example, an array of the size 10^6 that contains 32 integers in wrong positions.

For our experiments, we use a machine with an Intel i9-10980XE 18-core processor running Ubuntu 20.04.1 LTS with 128 GB of RAM. Each core has a base frequency of 3.0 GHz and a max turbo frequency of 4.6 GHz and supports the AVX-512 vector instruction set. We compile our experiments with the Intel C++ compiler version 19.1.2.254 for 64 bit using the optimization flags `-Ofast` and `-march=native`.

Figure 5 shows how the speedup factors of our implementation over std::sort compared to speedup factors of state-of-the-art high-speed sorting algorithms. Starting from an array length of 10^4 our implementation is always significantly faster for the considered distributions of integers. We are on par with sort AVX-512, when sorting small arrays. Both our implementation and sort AVX-512 use vectorized sorting networks for small arrays, but our implementation only uses the more portable AVX2 vector instructions, but not AVX-512. In the AVX-512 instruction set the vectors are twice as wide than in AVX2, and in AVX-512 there are more suitable instructions for sorting available. Furthermore, Figure 5b and Figure 5c show the in general poor performance of Sort AVX-512, when sorting arrays with many duplicate keys, while our vectorized Quicksort implementation can handle this case efficiently. It also becomes apparent that a general purpose sorting algorithm like IPS^{4}o cannot keep up with the performance of an efficient vectorized sorting algorithm when sorting primitive data types, at least for the single-threaded case. For the multi-threaded case, the results may not be as clear-cut, since memory bandwidth becomes the ultimate performance limit for sorting algorithms.



(a) *Uniform* over the range $[-2^{31}, 2^{31} - 1]$.

(b) *Gaussian* ($\mu = 0$, $\sigma = 100$).

(c) *Zero* (all values are equal).

(d) *Almost Sorted* ($\lfloor \frac{1}{2} \cdot 2^{\log_{10} n} \rfloor$ misplaced numbers).

Figure 5 Speedup factors of our implementation, sort AVX-512 [5], IPP radix sort [21], and, IPS^{4}o [1] over std::sort for four different distributions of 32-bit signed integers. In addition, the graphs also contain the absolute running times of our implementation.

Performance Indicators. For finding out the reasons why our implementation performs better than state-of-the-art high-speed sorting algorithms, we look at performance indicators such as cycles required per sorted integer, cache misses, and instructions per cycle of the various implementations. Table 2 contains performance indicators for the evaluated sorting algorithms. Additionally, we also show performance indicators for std::sort. Each algorithm sorts an array of 10^9 random integers. Our implementation needs on average 73 instructions

per integer, while IPP radix sort requires only about 44 instructions. Also sort AVX-512 requires on average fewer instructions per integer (≈ 46) than our implementation. But both IPP radix sort and sort AVX-512 need more cycles to execute these fewer instructions than our implementation. This means the CPU utilization (instruction-level parallelism) of our implementation is higher than that of IPP radix sort or sort AVX-512. At the same wall-clock time, our implementation sorts more than 30% more integers than IPP radix sort or sort AVX-512. In addition, it should be noted that our implementation is in-place, while IPP radix sort uses $\mathcal{O}(n)$ extra memory.

Table 2 Performance indicators per sorted integer. Each algorithm sorts an array of length 10^9 populated with random integers from the range $[-2^{31}, 2^{31} - 1]$. For example, sorting an integer with std::sort, on average, takes 377.14 CPU cycles and requires 226.25 instructions. The number of instructions per cycle (IPC) is defined as the quotient of the number of instructions and the number of cycles. A high IPC indicates efficient CPU utilization in terms of instruction-level parallelism. GHz denotes the average clock speed while sorting. When the integer is not in the cache, a cache miss occurs. The average number of cache misses per sorted integer is shown for the first level cache (L1) and the last level cache (LLC). The performance indicator branch-misses denotes the average number of mispredictions of the CPU branch-predictor per sorted integer.

	cycles	instructions	L1-misses	LLC-misses	branch-misses	IPC	GHz
std::sort	377.14	226.25	1.41	0.75	13.04	0.60	4.76
this paper	34.44	72.68	1.25	0.59	0.06	**2.11**	4.76
sort AVX-512	43.32	46.44	1.42	0.71	0.34	1.07	3.97
IPP radix sort	48.83	43.68	5.53	0.46	0.00	0.89	4.71
IPS^{4}o	137.96	273.83	1.78	0.28	1.46	1.98	4.76

5 Conclusions

In this paper we presented a highly efficient single-threaded in-place sorting algorithm that we implemented for the data type `int` with AVX2 intrinsics in C++. To make our implementation fast and robust, we improved vectorization techniques for sorting networks and Quicksort. In particular, we showed how to utilize the full capacity of vector registers for executing the modules of sorting networks and developed a pivot selection technique to efficiently sort arrays of different key distributions. Furthermore, we presented a vectorized in-place partitioning technique for vectorized Quicksort that has a high degree of instruction-level parallelism. With our implementation we achieve a speedup of at least 30% over state-of-the-art high-performance sorting algorithms. The worst case running time of our sorting algorithm is $\mathcal{O}(kn)$, where k is the number of bits in the primitive type. Our implementation is easily extensible to other primitive types like `unsigned int` or `float`, and can also be adapted to sort primitive 64-bit types. Actually, there is no need to consider the sorting of floating-point numbers separately, since the IEEE format was designed in such a way that with some additional bit-twiddling floating-point numbers can be sorted with integer sorting routines [6, 18]. On future processors with larger vector registers and more diverse vector instructions, the speed difference in favor of vectorized sorting algorithms for primitive types is likely to increase further. Hence, we see a vectorized in-place hybrid algorithm based on sorting networks and Quicksort as a possible replacement for current sorting implementations in high-performance libraries.

References

1. Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.9.
2. Jakob Andreas Bærentzen, Jens Gravesen, François Anton, and Henrik Aanæs. *Guide to Computational Geometry Processing*. Springer, 2012. doi:10.1007/978-1-4471-4075-7.
3. Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 307–314. ACM, 1968. doi:10.1145/1468075.1468121.
4. BLAS. Basic linear algebra subprograms. URL: http://www.netlib.org/blas/.
5. Berenger Bramas. A novel hybrid quicksort algorithm vectorized using avx-512 on intel skylake. *International Journal of Advanced Computer Science and Applications*, 8(10), 2017. doi:10.14569/IJACSA.2017.081044.
6. Randal Bryant. *Computer Systems: A Programmer's Perspective*. Pearson, 3rd edition, 2016.
7. Shi-Kuo Chang. *Data Structures and Algorithms*. World Scientific, 2003.
8. Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008. doi:10.14778/1454159.1454171.
9. Michael Codish, Luís Cruz-Filipe, Thorsten Ehlers, Mike Müller, and Peter Schneider-Kamp. Sorting networks: To the end and back again. *Journal of Computer and System Sciences*, 2016. doi:10.1016/j.jcss.2016.04.004.
10. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 3rd edition, 2009.
11. Amjad M. Daoud, Hussein Abdel-jaber, and Jafar Ababneh. Efficient non-quadratic quick sort (nqquicksort). In Ezendu Ariwa and Eyas El-Qawasmeh, editors, *Digital Enterprise and Information Systems*, pages 667–675. Springer, 2011.
12. Developer Zone. Intel® Software Developer Zone, 2008. URL: https://software.intel.com/en-us/articles/measuring-instruction-latency-and-throughput.
13. Jay Devore and Kenneth Berk. *Modern Mathematical Statistics With Applications*. Springer, 2012.
14. Jack J. Dongarra, Fred G. Gustavson, and Alan H. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, 1984. doi:10.1137/1026003.
15. Michael Griebel. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Springer, 2007.
16. Shay Gueron and Vlad Krasnov. Fast quicksort implementation using avx instructions. *The Computer Journal*, 59(1):83–90, 2016. doi:10.1093/comjnl/bxv063.
17. David R. Helman, David A. Bader, and Joseph JáJá. A randomized parallel sorting algorithm with an experimental study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998. doi:10.1006/jpdc.1998.1462.
18. Michael Herf. Radix tricks, 2001. URL: http://stereopsis.com/radix.html.
19. Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 189–198, 2007. doi:10.1109/PACT.2007.4336211.
20. Hiroshi Inoue and Kenjiro Taura. Simd- and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment*, 8(11):1274–1285, 2015. doi:10.14778/2809974.2809988.
21. Intel Corporation. Developer Reference for Intel® Integrated Performance Primitives. URL: https://software.intel.com/en-us/ipp-dev-reference.

22 Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2016.

23 Intel Corporation. Intel® Integrated Performance Primitives Cryptography: Developer Guide, 2020. URL: https://software.intel.com/sites/default/files/ippcp-devguide.pdf.

24 Intrinsics Guide. Intel intrinsics guide. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

25 Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: Computer Vision in C++ With the OpenCV Library*. O'Reilly Media, 2016.

26 Ronald Kneusel. *Random Numbers and Computers*. Springer, 1st edition, 2018.

27 Donald E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

28 Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 2006.

29 Ibrahim Küçük. *Astrophysics*. IntechOpen, 2012.

30 Tapio Lahdenmäki and Michael Leach. *Relational Database Index Design and the Optimizers*. John Wiley & Sons, 2005.

31 Stewart A. Levin. A fully vectorized quicksort. *Parallel Computing*, 16:369–373, 1990. doi:10.1016/0167-8191(90)90074-J.

32 Peng Li, David J. Lilja, Weikang Qian, Kia Bazargan, and Marc D. Riedel. Computation on stochastic bit streams digital image processing case studies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(3):449–462, 2014. doi:10.1109/TVLSI.2013.2247429.

33 Michal Ozery-Flato and Ron Shamir. Sorting by translocations via reversals theory. In *Comparative Genomics*, pages 87–98. Springer, 2006. doi:10.1007/11864127_8.

34 Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2:205–211, 1992. doi:10.1142/S0129626492000337.

35 Steven Ross. Boost.sort. URL: https://www.boost.org/doc/libs/1_67_0/libs/sort/doc/html/index.html.

36 Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

37 Howard Jay Siegel. The universality of various types of simd machine interconnection networks. *SIGARCH Comput. Archit. News*, 5(7):70–79, 1977. doi:10.1145/633615.810655.

38 Harold S. Stone. Sorting on star. *IEEE Transactions on Software Engineering*, SE-4(2):138–146, 1978. doi:10.1109/TSE.1978.231484.

39 Martin Weisser. *Essential Programming for Linguistics*. Edinburgh University Press, 2009.

40 Zekun Yin, Tianyu Zhang, André Müller, Hui Liu, Yanjie Wei, Bertil Schmidt, and Weiguo Liu. Efficient parallel sort on avx-512-based multi-core and many-core architectures. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 168–176. IEEE, 2019. doi:10.1109/HPCC/SmartCity/DSS.2019.00038.

41 Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 712–721. ACM, 1991. doi:10.1145/125826.126164.

A Worked Example of Vectorized In-Place Partitioning

Our vectorized partitioning technique is a hybrid of the techniques of Gueron and Krasnov [16], and, Bramas [5] with additional optimizations. Our technique for partitioning an array is demonstrated in Figure 6 using vectors that can hold four elements.

In the example in Figure 6, the *array* to be partitioned consists of 20 elements. The pivot is 49. An arrow, with the text *partition vector* above, indicates the partitioning of a single vector using the pivot vector *v_pivot*. While partitioning a vector, the elements smaller than or equal to the pivot are arranged contiguously at the beginning of the vector, and the elements greater than the pivot are arranged thereafter. The function *partition vector* returns the number of elements greater than the pivot. Based on this information the indices

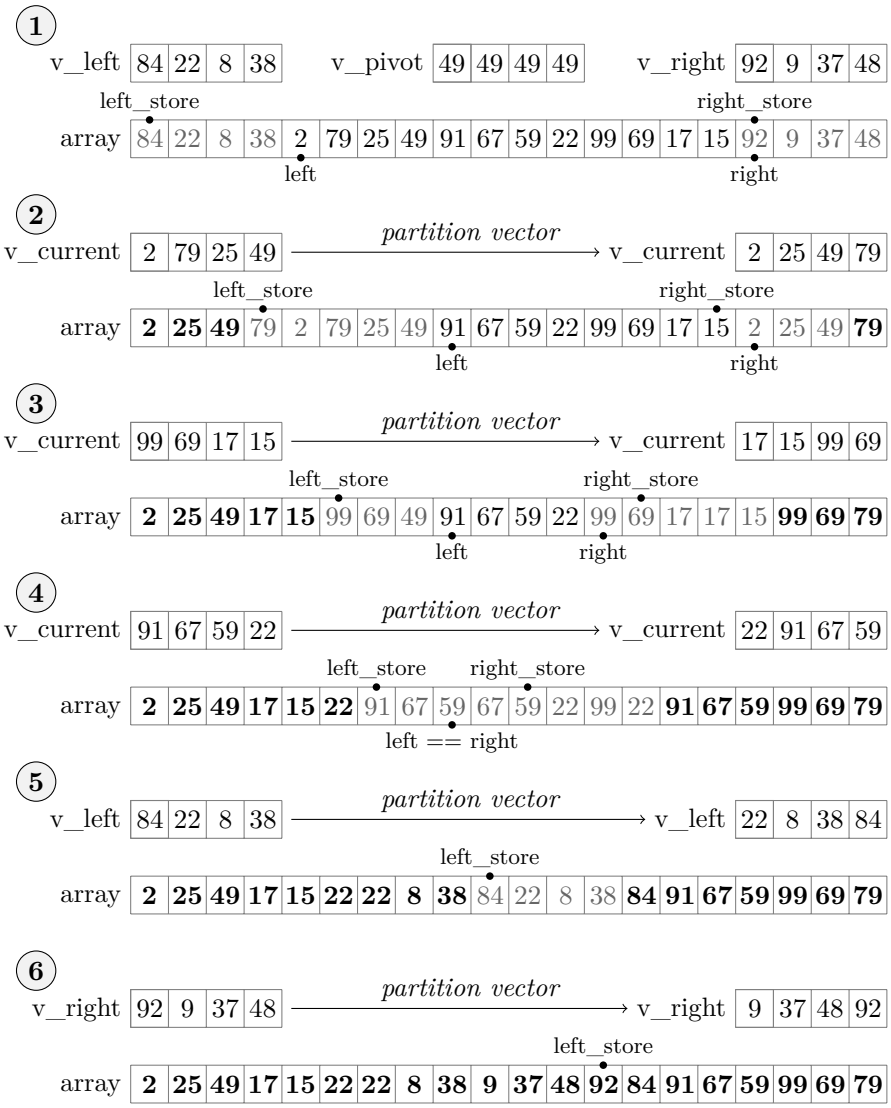


Figure 6 Vectorized partitioning example.

left_store and *right_store* are updated. The indices *left_store* and *right_store* indicate the store points in the *array* for a partitioned vector. The *left* and *right* indices, on the other hand, are used to determine the next elements to load for partitioning. Already partitioned elements are printed in bold. Elements that can be overwritten are grayed out. The details of the procedure in Figure 6 are as follows:

① At the beginning of partitioning, the first four elements of the *array* are stored in the vector *v_left* and the last four elements in the vector *v_right*. The vectors *v_left* and *v_right* are partitioned only after all the other elements of the *array* have been processed.

② The four elements at the index *left* are loaded to *v_current*. The vector *v_current* is partitioned and stored at the outer ends of the *array*. The store point *left_store* moves three index positions to the right, because in *v_current* three elements are smaller than or equal to the pivot. Equivalently, *right_store* moves one index position to the left, since in *v_current* one element is greater than the pivot.

③ Since the *array* contains fewer partitioned elements on the right side than on the left (one element on the right side and three on the left), the four elements are loaded into *v_current* from the right side of the *array*. After the partitioning of *v_current* and its storage on both sides of the *array* according to the store points *left_store* and *right_store* a total of five elements are partitioned to the left and three to the right side of the *array*. The store points *left_store* and *right_store* are updated and *right* is moved four index positions to the left.

④ The vector *v_current* is again loaded from the index *right*, since the right side of the *array* contains fewer partitioned elements than the left side. The vector *v_current* is partitioned and stored according to the store points *left_store* and *right_store*. The index *right* and the store points are updated. The indices *left* and *right* are the same, which means that the vectors *v_left* and *v_right* must be partitioned before the complete *array* is fully partitioned.

⑤ The initially created vector *v_left* is partitioned and stored on both sides of the *array* according to *left_store* and *right_store*. Three elements are smaller than the pivot, so *left_store* is moved three index positions to the right. The store point *right_store* can be ignored because it is no longer needed.

⑥ The initially created vector *v_right* is partitioned and stored according to *left_store*. Three elements are smaller than the pivot, so *left_store* is moved three index positions to the right.