juliacon

ChipSort: a SIMD and cache-aware sorting module

Nicolau Leal Werneck[1]

[1]TomTom NV

ABSTRACT

Sorting is a fundamental programming problem with many important applications. While there are well-known sorting algorithms for conventional computers, special techniques are required to take the most out of real hardware offering features such as cache memory and instruction-level parallelism. ChipSort.jl is a Julia module for SIMD and cache-aware sorting. It implements sorting networks and bitonic merge networks with SIMD instructions, with configurable vector sizes. It also implements Combsort, which lends itself easily to vectorization and can achieve good performance depending on the memory access cost. Insertion sort is used to finalize. Large arrays are approached with a multi-way Mergesort. The implementation of ChipSort itself is of interest from a programming languages perspective due to its use of metaprogramming, more specifically Julia's generated functions. This enables custom code generation for different tasks and hardware at run-time, a feat owed to Julia. This article presents the implemented techniques as well as experiments that demonstrate speed gains compared to multiple standard libraries.

Keywords

Julia, Sorting, SIMD, Parallelism, Metaprogramming

1. Introduction

Sorting has enjoyed an important place within computer programming topics for a long time. It has great importance to practical applications, being very useful to organize data for fast retrieval, and also attracts attention as a theoretically interesting problem in itself [16]. While sorting is a well-understood problem in general, with a few classic algorithms available, achieving optimal performance for specific problems and architectures may require a careful algorithm selection rather than a simple parameter tuning [7, Part II, Introduction].

Many factors can influence the running time of a computer program, starting with algorithmic complexity and the base clock speed of the processor. The performance observed in real computers has been increasingly depending on processor features that are not taken into account by the simplest models, though. These features include cache memory and parallelism at the processor and instruction levels. Cache memory has become popular in consumer processors in the past couple of decades, and as long as memory access patterns exhibit temporal and spatial locality, it allows programs to attain a higher performance than would be possible with a simpler memory architecture [10, Chapter 3]. Writing software with better support for parallelism has also become necessary to utilize the full potential offered by modern and future processors [31].

One particular form of parallelism that concerns our work is SIMD (single instruction, multiple data) instructions, where the processor applies a same operation to multiple independent memory elements simultaneously. These operations are said to act on *vectors*, where each parallel *lane* carries out the scalar version of the program. SIMD parallelism is only suitable to specific problems. It may introduce data access constraints and overheads, and even clock speed reductions. SIMD performance can be affected negatively by branching and scattered memory access. It is mostly suited for calculating many replicas of a long arithmetic expression, preferably with little IO. It can still pay off in appropriate settings and, as will be demonstrated, sometimes the implementation can be simple too. More information about SIMD can be found in [4].

Processors offering SIMD or other forms of instruction-level parallelism started to become widely available in the early 2000s. One example is the introduction of SSE instructions in the x86 architecture [17, Chapter 1]. The current availability of processors with 512 bit registers, along with modern GPUs, underscore the demand for software-building tools that support vector operations. One programming language feature relevant to that goal is polymorphism, allowing programmers to abstract over data types and vectors. Another is *metaprogramming*, the ability of writing programs that manipulate or generate other programs [1][27, Ch.25]. Metaprogramming enables code generation for complex algorithms customized to problem parameters and platform resources.

The interest in sorting and the great relevance of cache memory and parallelism to modern computing eventually led to research on sorting algorithms that account for these factors, following research in other topics where SIMD is more clearly beneficial. At the time SIMD CPUs became widely available there was already a trend to employ specialized hardware like GPUs for scientific applications [19, 28]. Although SIMD was used mostly for multimedia applications at first [5, 25], some early alternative uses include 3D graphics [20] and machine learning [26]. Following earlier work on parallel sorting in other architectures, the late 2000s saw the first practical demonstrations of sorting with modern consumer SIMD chips [13, 6]. Most of the techniques utilized then are still employed in more recent works [2, 14].

This article presents `ChipSort.jl` [30], a Julia module that implements SIMD and cache-aware sorting utilizing some of the techniques found in the literature. These include sorting and merging networks, the use of Combsort with Insertion sort finalization for medium sized arrays and of multi-way merging for large arrays. These techniques require not only tuning parameters such as vector and array sizes, but also generating specific code when implementing different size networks. `ChipSort` seeks to generate efficient custom code for each task, relying on Julia metaprogramming capabilities for that.

Section 2 ahead presents the techniques employed in `ChipSort`. Section 3 presents experimental results assessing the performance of the module and Section 4 brings some concluding remarks.

2. SIMD sorting techniques

`ChipSort` offers four high-level functions that can be used in applications as replacements for other sorting functions such as `sort` in the Julia standard library. The `chipsort!` function does not require any configuration and offers a great performance on a wide range of cases. The remaining functions are suitable for arrays of specific size ranges, and require choosing some numeric parameters. The function `chipsort_large` offers a good overview of all the techniques implemented in the module, and this section is structured according to its stages. Some of these techniques may also be useful in other applications instead of sorting, one clear example being the in-place matrix transpose.
The way `ChipSort` works mostly follows [14], except it does not yet support registers with key and payload and there are no measures to guarantee a stable sort, such as ensuring unique keys or performing a post-processing to enforce the original order of colliding keys. Another main difference is that `ChipSort` always employs Combsort with an Insertion sort finalization.
The function `chipsort_large` achieves sorting by first splitting the input array in chunks that are sorted separately with `chipsort_medium!` and then merged simultaneously. This inner sort is performed in multiple steps:

(1) sorting small blocks of data with a sorting network;
(2) reordering vectors in memory according to a matrix transpose;
(3) vectorized Combsort with a limited number of iterations;
(4) regular insertion sort until the whole data is sorted.

The remainder of this section details the multiple techniques employed in this process.

2.1 Sorting networks

Sorting networks [16, Sec. 5.3.4] are graphs composed by comparator modules, units that take two numbers in, and output the smallest and largest at specific outputs. By properly arranging comparators we can create a procedure to take an array of elements and output them in sequence. The design of a sorting network can minimize the total number of comparisons, or if they can be done in parallel, minimize the number of steps to complete the process.
A sorting network can be represented by a Knuth graph such as Figure 1. Each vertical line is a comparator. The graph represents a network that takes an array of four values (A_1, A_2, A_3, A_4) and outputs the sorted sequence (S_1, S_2, S_3, S_4).

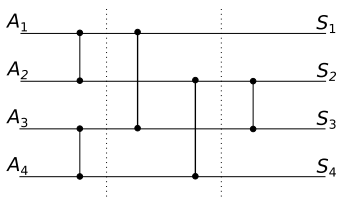

Fig. 1. A sorting network for 4 elements. The dashed lines delimit each step where operations are independent of each other.

In `ChipSort`, as in the related work, parallelism is mainly exploited by carrying out the comparisons of the sorting network over vectors of size V contained in SIMD registers, effectively running V networks simultaneously. Other forms of instruction-level parallelism can also be in play, though. It may be possible to parallelize instructions such as `min` and `max`, or the loading of data into registers. This is often done implicitly by the microarchitecture, and programmers can only make sure their code is suitable for this by properly ordering operations and avoiding branching, for instance. Part of this work is expected to be performed by the compiler, and in particular LLVM in the case of Julia.
`ChipSort` supports sorting networks of different sizes, currently only powers of 2, which are predefined as data structures in the file `sorting-network-parameters.jl`. The generated function `sort_net` assigns each element of the sequence at each step to a variable, and the values from the next step are calculated according to the network specification. The comparisons are performed by the `min` and `max` functions, and the function is generic on the element types. Figure 2 shows one example of generated code.

```
4 => (((1,2), (3,4)), ((1,3), (2,4)), ((2,3),))
```

```
input_0_1 = input[1]
input_0_2 = input[2]
input_0_3 = input[3]
input_0_4 = input[4]
input_1_1 = min(input_0_1, input_0_2)
input_1_2 = max(input_0_1, input_0_2)
input_1_3 = min(input_0_3, input_0_4)
input_1_4 = max(input_0_3, input_0_4)
input_2_1 = min(input_1_1, input_1_3)
input_2_3 = max(input_1_1, input_1_3)
input_2_2 = min(input_1_2, input_1_4)
input_2_4 = max(input_1_2, input_1_4)
input_3_1 = input_2_1
input_3_4 = input_2_4
input_3_2 = min(input_2_2, input_2_3)
input_3_3 = max(input_2_2, input_2_3)
return (input_3_1, input_3_2, input_3_3, input_3_4)
```

Fig. 2. The parameters from a four-element sorting network consisting of three steps, and the corresponding code generated by `sort_net`.

This function is a good first example of how `ChipSort` utilizes Julia's metaprogramming features. Multiple implementations of sorting networks are available in the literature [6, 11, 9], however they implement networks of different sizes straight as code. In `ChipSort` they are represented as data structures that guide a generated function to produce equivalent code. While this data structure is currently hard-coded, it could be potentially produced only when necessary by an algorithm such as Bose-Nelson, what remains a future plan for the module.
The `sort_net` generated function expects elements to which `max` and `min` are defined. It can be called with simple data types, *e.g.* `sort_net(4,2,5,3)` works. In practice `ChipSort` often calls this function with vectors, completely relying on `SIMD.jl` [24]. This module maps a vector in Julia similar to tuples or static arrays straight to LLVM vector types, ensuring SIMD code can be produced when possible and also allowing the flexibility of handling logic vectors that are larger than the actual registers, or spilling data to the stack if all registers are occupied.

2.2 SIMD vectors transpose

Consider a group of stacked SIMD vectors. Each column in this representation is termed a *lane*. The result from `sort_net` with vectors is that each lane contains a sorted sequence. This data must be transposed to produce sorted vectors. This is done in `ChipSort` through `transpose_vecs`, which is a generated function that supports rectangular matrices, but with dimensions constrained to powers of 2. The generated code consists mostly of calls to `SIMD.shufflevec`. This is another instance of an operation that other projects [2, 9] implement with multiple low-level functions contemplating each different case, while in `ChipSort` there is only a single Julia generated function that produces equivalent code.
Figure 3 displays an example of 16 random integers loaded into SIMD registers as 4 vectors. After `sort_net` each lane contains a sorted sequence, and `transpose_vecs` turns the lanes into vectors (registers). The `chipsort_small!` function is essentially a sorting network followed by a transpose and a merge procedure.

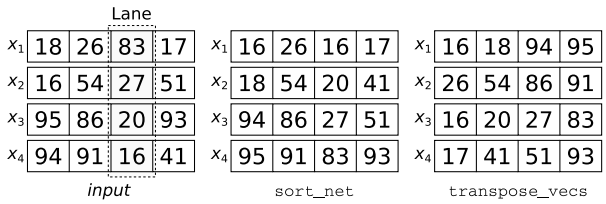

Fig. 3. Four vectors with four values, the result from a vectorized four-elements sorting network, and the transpose.

2.3 Bitonic Merge Networks

A bitonic sequence starts either increasing or decreasing and contains at most one change of direction. Bitonic merge networks allow the creation of merge networks of different sizes, and they are again implemented in `ChipSort` with a single generated function, `bitonic_merge`. Merging two sorted vectors requires first reversing one of them, and the result is stored in two vectors containing the first and second halves of the complete sorted sequence.
Apart from the use in sorting small sequences and in the multi-way merge tree, bitonic merge networks can also be used to implement a regular merge sort. This technique was investigated in `ChipSort` only superficially, and while there were no promising results in terms of performance, `ChipSort` also contains the generated function `bitonic_merge_interleaved` to illustrate how metaprogramming can help implement multiple simultaneous merges with interleaved execution, as done in other projects [6].

2.4 Vectorized Combsort

One of the most peculiar aspects of `ChipSort`, taken directly from AA-sort [13, 14], is the use of the Combsort algorithm. Combsort essentially generalizes Bubble sort in the same way that Shell sort generalizes Insertion sort [12, 8, 18, 16, 7]. The array is swept multiple times, ordering pairs of values at a given distance from each other. The distance is reduced each time by $\frac{4}{3}$. Parallelism is achieved either by parallelizing the operations from a single array or by processing independent arrays of the same size.
`ChipSort` contains two functions that utilize Combsort, employing these two distinct forms of parallelism. The first is `chipsort!`, a serial implementation of the algorithm which is vectorized implicitly by the optimizing compiler. `chipsort!` starts with Combsort until the interval size is 1, when it switches to Insertion sort.
The other function is `chipsort_medium!`, an explicitly vectorized implementation utilizing `SIMD.jl`. This second function also performs a number of operations before and after Combsort. It consists of the following steps:

(1) Apply a sorting network on K blocks of J vectors of size V.

(2) Vectorized Combsort until the interval size is 1.

(3) Transpose blocks.

(4) Transpose data in-place from K blocks of J vectors into J blocks of K vectors.

(5) Vectorized Combsort again.

(6) Sort blocks again.

(7) Insertion sort.

After the first two steps the result is essentially a matrix where the $K \times J$ columns are the vectors, and each of the V rows contains an approximately sorted sequence. The transpose steps reshape each row into a new block. Each row is processed independently at first, and the transpose allows elements from distinct groups to interact. The last step is necessary to ensure the whole array is sorted, not viewed anymore as a matrix with independent rows.
In both implementations switching to Insertion sort presents a better performance than sticking to Combsort until the end. Once the interval becomes 1 Combsort essentially becomes Bubble sort, thus finding an alternative seems beneficial. Insertion sort should fare well because the array is now approximately sorted, and any out-of-order elements should be already close to their destinations.
Finishing a sorting method with Insertion sort is a common practice, present at least in Musser's Introsort [21]. This is also akin to the classic method used in the Julia standard library, switching to Insertion sort when the array becomes too small. The difference is that Combsort is not a divide-and-conquer approach, and therefore Insertion sort must be applied once on the whole array.
The utilization of Insertion sort as a final stage to Combsort has been considered before [23, 12], and Insertion sort was even employed by [14] for dealing with partial keys. The main contribution of the present work may be to extend AA-sort [13, 14] by giving a larger role to Insertion sort and also relying more on automatic vectorization by the compiler.

2.5 In-place matrix transpose

In-place matrix transposition can be attained by moving a number to its destination, then moving away the number that was found there, and so on, until a cycle is completed and a new number is selected to be moved [32]. While carrying out this procedure is simple, it is not trivial to find out what are the cycles to transpose a matrix of any given dimensions, and even finding out the number of cycles turns out to be a difficult problem [15, 1.3.3-12].
`ChipSort` contains the generated function `transpose!` to perform in-place matrix transposition. The cycle seeds are computed at the time of code generation, and at run-time the function only moves the data, computing the index sequences and detecting when each cycle finishes. Therefore even though the function is completely generic, supporting any shape, the most complex part of the problem is tackled programmatically during code generation, resulting in a very simple run-time code.

2.6 Vectorized multi-way merging

Merging multiple arrays simultaneously requires the creation of a *merge tree* to keep some intermediate data. Each node in this tree is essentially an iterator traversing the merger of two arrays in the lower level. To employ the vectorized bitonic merge network in this process it is necessary to keep a whole vector at each node of the tree. To produce the next vector of elements one of the two source arrays is selected according to their smallest next element, and a whole vector is pulled from that array. This vector is merged with the intermediate one, and the first half of the resulting sequence can be taken to the next level of the tree, while the second half is kept as the new intermediate data.
Given a set of sorted arrays, once the tree is initialized data is taken from it one vector at a time, producing the final sorted sequence. Each time a new vector is requested the tree is traversed following the smallest element among the node children, until one of the input arrays is reached. The data from the array is then loaded from memory and merged with the intermediate data, producing new updates moving through the tree until the root is reached.
The intermediate data from the tree is kept by `ChipSort` in a contiguous array, treated as a heap. Assuming 1-based indexing, each node ancestor can be simply found by the Euclidean division of its index by 2. This data is assumed to fit some level of cache memory during operation, as in the related work [13, 6, 14].

3. Experiments

This section reports experiments with the `ChipSort` methods, as well as alternative ones. The term *method* can be understood as both the algorithms and their implementations in the Julia language. The specific methods tested were:

—`chipsort!` Combsort finishing with Insertion sort, written as a serial program and vectorized by the compiler.

—`chipsort_small!` in-register, based on sorting networks.

—`chipsort_medium!` vectorized Combsort, plus additional steps, finishing with Insertion sort.

—`chipsort_large` merge-sort with a multi-way merge tree, starting from chunks processed by `chipsort_medium!`.

—`insertion_sort!` Insertion sort taken from the Julia library.

—`sort!` the Julia standard sort, Quicksort finishing with Insertion sort for small arrays (20 elements).

The `!` implies in-place operation. The allocation time for `chipsort_large` was not discounted in our analyses.
While our experiments focused on 32-bit integer values, similar results were observed with other types, although with possibly smaller speedups with 64-bits. Further details are omitted for the sake of brevity. For floating-point numbers, while in theory every method should perform comparably to integers, issues can appear in practice due to peculiarities such as `NaN`s. This interferes with the ability of the compiler to optimize `chipsort!`, although the other methods with explicit vectorization are less affected. There is ongoing work in Julia and LLVM to improve the performance of floating-point comparisons [22, 29, 3].

3.1 Small arrays

In our first experiment we evaluate the sorting of small arrays, with 64 32-bit values. The `chipsort_small!` method based on sorting and bitonic merge networks was compared to serial Insertion sort, the Julia sort and `chipsort!`. The short running time makes it challenging to perform proper measurements, therefore the task was to sort 128 sequences of 64 contiguous values in memory, forming a single block of 2^{13} uniform random entries. Sorting was performed in-place, in a for-loop. The results are in Table 1.

Table 1. Sorting 128×64 `UInt32` elements.

Method	Median time
`chipsort_small!`	57.206 μs
`chipsort!`	133.510 μs
Insertion sort	194.679 μs
Julia standard	222.910 μs

The `chipsort_small` function achieved a significant acceleration of over 3 times relative to Insertion sort or the Julia standard function, and twice over `chipsort!`. Even though the in-register function was clearly faster, this first experiment already demonstrates the potential of `chipsort!`, which presented a speedup of at least 46% compared to the other alternatives.

3.2 Medium and large arrays

Our second experiment compared many different methods on inputs of exponentially increasing sizes, from 2^6 to 2^{20}. Figure 4 displays the resulting measurements. Although the curves look too concentrated in this visualization, it is possible to notice that `chipsort!` consistently outperforms the Julia standard library by a factor of 80% to 100%, or up to twice as fast. We can also observe that Insertion sort soon becomes too inefficient.

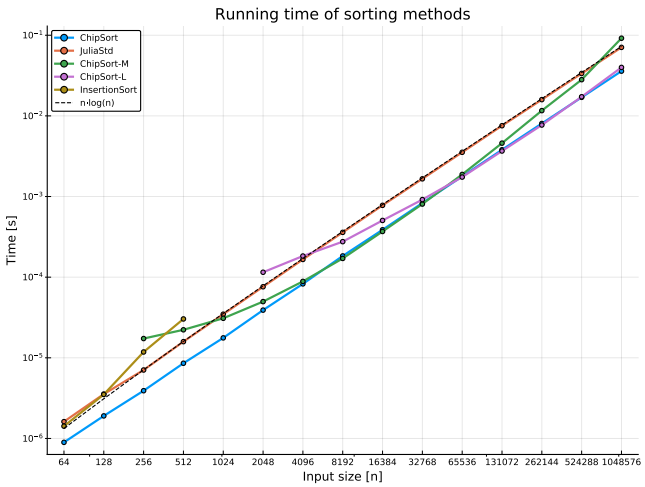

Fig. 4. Running time of the different sorting methods studied.

Figure 5 shows a different visualization from the same data with the time divided by the input size. The vertical dotted lines indicate the size of each cache level. Here we can see more clearly how `chipsort_medium!` and `chipsort_large` slightly outperform `chipsort!` in a few cases.
Figure 6 displays the measured times for 2^{13} and 2^{18} elements. All the tested `ChipSort` methods surpass the Julia standard library in these cases. In the 2^{13} case `chipsort_medium!` exhibits a slight speedup of 7.3% over `chipsort!`, and on the 2^{13} case it is `chipsort_large!` with 4.4%.

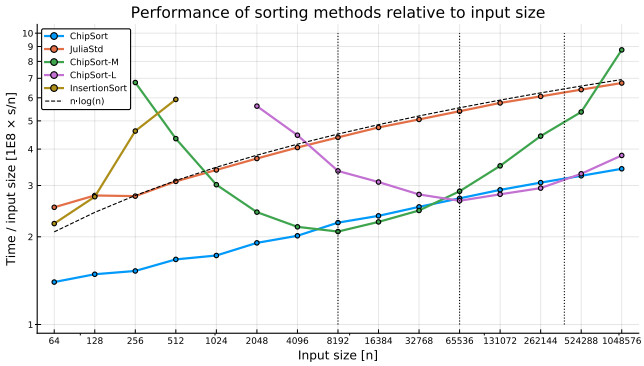

Fig. 5. Curves displaying the relative running time from each studied method. The ordinate represents the median of the measured running times divided by the input size at each test. Lower is faster.

3.3 Other platforms

One last experiment was carried out with different programming languages in the same machine and with Julia in different machines, to provide more context. Table 2 displays statistics collected from Julia, Python and C++ programs. The experiment was performed with a Ubuntu 18.04 system using Julia 1.3.0-DEV.431, Python 3.6.8 with Numpy 1.14.3 and GCC 8.3.0 with `-O3 -march=native` and libstdc++ 6.0.25. For the case of Python the measurements obtained by calling the function through PyCall were better than using the native Python shell, therefore the best value was kept. Results obtained for C++ with Julia interoperation using `Cxx.jl` were slightly worse than native, thus discarded. Numpy, C++ `sort` and Julia all presented similar performance, unmatched by C++ `qsort`. These methods make little to no use of SIMD, enabling `ChipSort` to surpass them.

Table 2. Sorting arrays of 8k and 1M `UInt32` elements in different software platforms.

Method	8k	1M
`ChipSort`	172.35 μs	35.28 ms
Julia standard	337.03 μs	68.39 ms
Numpy via PyCall	354.55 μs	70.01 ms
C++ `std::sort`	374.20 μs	68.10 ms
C++ `std::qsort`	589.52 μs	108.2 ms

Table 3 contains the results for `chipsort!` and the Julia standard `sort!` on an Intel processor with AVX512 instructions, and an ARM processor with 128-bit NEON instructions. No change in performance can be seen on the AVX512 machine, and on the ARM machine the speedup was smaller than before: only 60%. The most notable fact in this experiment is that the same high-level and serial code successfully produced a vectorized sort on machines with different vector sizes and even instruction sets, always with a beneficial outcome.

Table 3. Sorting arrays of 8k and 1M `UInt32` elements in different hardware platforms.

Chip	Method	8k	1M
AXV512	`ChipSort`	177.4 μs	35.20 ms
AXV512	Julia standard	347.5 μs	69.07 ms
NEON	`ChipSort`	350.9 μs	73.36 ms
NEON	Julia standard	573.2 μs	113.68 ms

4. Conclusion

We have presented `ChipSort`, a Julia module for SIMD and cache-aware sorting. The main sorting method utilized in `ChipSort` is Combsort finalized with Insertion sort. This method is based on previous proposals for SIMD sorting [13, 14] and other practical sorting methods [12, 21]. This method was implemented as a serial program vectorized by the Julia compiler and LLVM, resulting in a sorting function up to twice as fast as the Julia standard library sort. This performance holds for at least a million 32-bit integers in a personal computer.

Other techniques available in `ChipSort` are sorting networks, bitonic merge networks, in-place matrix transpose and a multi-way merge tree. Vectorized in-register sorting, although limited to small powers of 2, proved to deliver speedups of up to 3 times relative to the standard library.

Results were limited for the method dedicated to large arrays, only narrowly surpassing the main `ChipSort` method in few cases. Merge-based techniques may be more advantageous when sorting large records, though.

While the main `ChipSort` method relies on compiler optimizations, other methods utilize explicit vectorization trough `SIMD.jl`. `ChipSort` makes extensive use of metaprogramming, more specifically Julia generated functions. This allows functions like sorting networks to be implemented in an abstract and generic way. The networks are represented as data structures, used by a higher-level program to assemble the final network code. This contrasts with other libraries where different networks are implemented straight as final functions. Julia also has the opportunity to perform custom optimizations since `ChipSort` is implemented in pure Julia with abstract methods that can be specialized.

It is a testament to the quality of the Julia compiler infrastructure that the most successful method implemented in `ChipSort` has no explicit vectorization or intrinsics. It is just a simple, scalar implementation of Combsort that is optimized by the compiler to generate code adapted to different problem settings and to the hardware architecture at run-time. And in the cases where explicit vectorization is necessary Julia still offers a suitable framework that allowed `ChipSort` to implement sophisticated techniques with concise code. This project can illustrate how well Julia implements traditional language features such as parametric types, and its metaprogramming features make it stand out from other languages even more.

Some future prospects for `ChipSort` are to implement missing features such as buffers in the merge tree and stable sorting of key-payload data; explore the limits of in-register processing in modern 512-bit architectures an eventually GPUs; explore how to interact with the compiler in order to optimize code for complex tasks such as multi-stage multi-threaded merge-sorting of large arrays.

5. References

[1] Harold Abelson, Gerald Jay Sussman, and with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd edition, 1996.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[3] Matt Bauman et al. Julia issue #31442. `https://github.com/JuliaLang/julia/issues/31442`, 2019.

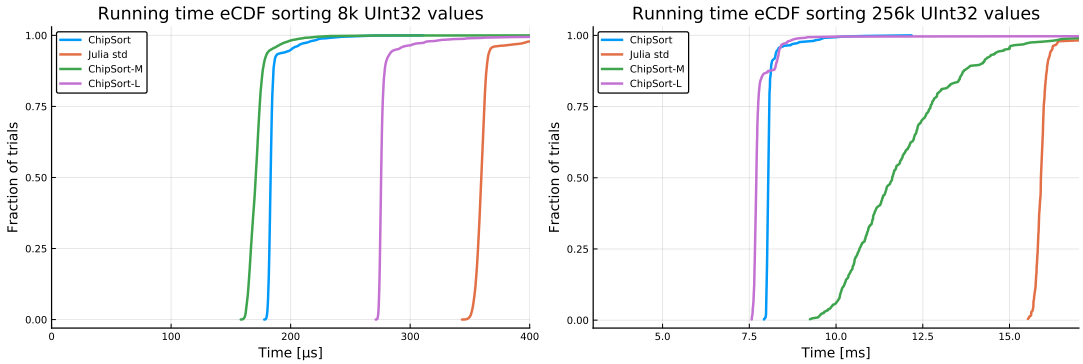

Fig. 6. Empirical CDF from different methods with 8k and 256k `UInt32` values. Results from 10,000 measurements.

[4] Jacco Bikker. Practical SIMD programming. `http://www.cs.uu.nl/docs/vakken/magr/2017-2018/files/SIMD%20Tutorial.pdf`, 2017.

[5] Yen-Kuang Chen, Eric Q. Li, Xiaosong Zhou, and Steven Ge. Implementation of H.264 encoder and decoder on personal computers. *Journal of Visual Communication and Image Representation*, 17(2):509 – 532, 2006.

[6] 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. *PVLDB*, 1(2):1313–1324, 2008.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[8] Wlodzimierz Dobosiewicz. An efficient variation of bubble sort. *Information Processing Letters*, 11(1):5–6, 1980.

[9] Dee Dong and Saatvik Shah. Ultra-sort: Extremely parallel hardware optimized sorting. `https://github.com/PatwinchIR/ultra-sort/tree/09fa26c3c77cb31b98031bdf1f76c54361d75b8e`, 2018.

[10] Ulrich Drepper. What every programmer should know about memory, 2007. Linux Weekly News.

[11] Jeffrey Sarnoff et alii. Sortingnetworks.jl. `https://github.com/JeffreySarnoff/SortingNetworks.jl/blob/b97c8eec6e0f4d3cc3b86a700e3c1dccaa6d159f/src/swapsort.jl`, 2018.

[12] Janet Incerpi and Robert Sedgewick. Practical variations of shellsort. *Information Processing Letters*, 26(1):37 – 43, 1987.

[13] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. AA-Sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT*, pages 189–198, 2007.

[14] Hiroshi Inoue and Kenjiro Taura. SIMD- and cache-friendly algorithm for sorting an array of structures. *PVLDB*, 8(11):1274–1285, 2015.

[15] Donald Ervin Knuth. *The art of computer programming*, volume 1: Fundamental Algorithms. Addison-Wesley, 3rd edition, 1997.

[16] Donald Ervin Knuth. *The art of computer programming*, volume 3: Sorting and Searching. Addison-Wesley, 2nd edition, 1998.

[17] Daniel Kusswurm. *Modern X86 Assembly Language Programming: Covers x86 64-bit, AVX, AVX2, and AVX-512*. 01 2018.

[18] Stephen Lacey and Richard Box. A fast, easy sort. *BYTE*, 16(4):315–ff., April 1991.

[19] E Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 55–55. ACM, 2001.

[20] Wan-Chun Ma and Chia-Lin Yang. Using Intel streaming SIMD extensions for 3D geometry processing. In *Third IEEE Pacific Rim Conference on Multimedia*, pages 1080–1087, 2002.

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

[22] Sanjay Patel. LLVM diff D62272. `https://reviews.llvm.org/D62272`, 2019.

[23] Martín Knoblauch Revuelta. Comb sort, or Dobosiewicz sort. `https://code.google.com/archive/p/combsortcs2p-and-other-sorting-algorithms/wikis/CombSort.wiki`, 2013.

[24] Erik Schnetter, Takafumi Arakaki, Valentin Churavy, Kristoffer Carlsson, Nicolau Werneck, Gunnar Farnebäck, Miguel Raz Guzmán Macedo, Matt Bauman, Kenta Sato, and Elliot Saba. eschnett/simd.jl. `https://doi.org/10.5281/zenodo.2592633`, March 2019.

[25] Nathan T. Slingerland and Alan Jay Smith. Multimedia extensions for general purpose microprocessors: a survey. *Microprocessors and Microsystems*, 29(5):225–246, 2005.

[26] Alfred Strey and Martin Bange. Performance analysis of intel's MMX and SSE: A case study. In *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference*, pages 142–147, 2001.

[27] The Julia Project. Julia 1.1 documentation. `https://docs.julialang.org/en/v1.1/manual/metaprogramming/`, 2019.

[28] Chris J. Thompson, Sahngyun Hahn, and Mark Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 306–317, 2002.

[29] Craig Topper. LLVM diff 3acc4236b871. https://reviews.llvm.org/rG3acc4236b8717bf493b81905194a10e71f526702, 2019.

[30] Nicolau Leal Werneck. nlw0/chipsort.jl: First release. https://doi.org/10.5281/zenodo.3228848, May 2019.

[31] Sophie Wilson. The future of microprocessors. https://www.youtube.com/watch?v=zX4ZNfvw1cw, 2018. JuliaCon keynote address.

[32] P. F. Windley. Transposing Matrices in a Digital Computer. *The Computer Journal*, 2(1):47–48, 01 1959.