A Framework for the Automatic Vectorization of Parallel Sort on x86-based Processors

Kaixi Hou, Hao Wang, *Member, IEEE,* and Wu-chun Feng, *Senior Member, IEEE*

Abstract—The continued growth in the width of vector registers and the evolving library of intrinsics on the modern x86 processors make manual optimizations for data-level parallelism tedious and error-prone. In this paper, we focus on parallel sorting, a building block for many higher-level applications, and propose a framework for the Automatic SIMDization of Parallel Sorting (ASPaS) on x86-based multi- and many-core processors. That is, ASPaS takes any sorting network and a given instruction set architecture (ISA) as inputs and automatically generates vector code for that sorting network. After formalizing the sort function as a sequence of comparators and the transpose and merge functions as sequences of vector-matrix multiplications, ASPaS can map these functions to operations from a selected "pattern pool" that is based on the characteristics of parallel sorting, and then generate the vector code with the real ISA intrinsics. The performance evaluation on the Intel Ivy Bridge and Haswell CPUs, and Knights Corner MIC illustrates that automatically generated sorting codes from ASPaS can outperform the widely used sorting tools, achieving up to 5.2x speedup over the single-threaded implementations from STL and Boost and up to 6.7x speedup over the multi-threaded parallel sort from Intel TBB.

Index Terms—SIMD, IMCI, AVX2, Xeon Phi, sorting networks, merging networks, code-generation.

1 Introduction

Increasing processor frequency to improve performance is no longer a viable approach due to its exponential power consumption and heat generation. Therefore, modern processors integrate multiple cores onto a single die to increase inter-core parallelism. Furthermore, the vector processing unit (VPU) associated with each core can enable more fine-grained intra-core parallelism. Execution on a VPU follows the "single instruction, multiple data" (SIMD) paradigm by performing a "lock-step" operation over packed data. Though many regular codes can be auto-vectorized by the modern compilers, some complex loop patterns prevent performant auto-vectorization, due to the lack of accurate compiler analysis and effective compiler transformations [1]. Thus, the burden falls on programmers to implement the manual vectorization using intrinsics or even assembly code.

Writing efficient vectorized (SIMD) code by hand is a time-consuming and error-prone activity. First, vectorizing existing (complex) codes requires expert knowledge in restructuring algorithms to exploit SIMDization potentials. Second, a comprehensive understanding of the actual vector intrinsics is needed. The intrinsics for data management and movement are equally important as those for computation because programmers often need to rearrange data in the vector units before sending them to the ALU. Unfortunately, the flexibility of the data-reordering intrinsics is restricted, as directly supporting an arbitrary permutation is impractical [2]. As a consequence, programmers must resort to a combination of data-reordering intrinsics to attain a desired computational pattern. Third, the vector instruction set architectures (ISA) continue to evolve and expand, which in turn, lead to potential portability issues. For example, to port codes from the Advanced Vector Extensions (AVX) on the CPU to the codes of the Initial Many Core Instructions (IMCI) on the Many Integrated Core (MIC), we either need to identify the instructions with equivalent functionalities or rewrite and tune the codes using alternative instructions. While library-based optimizations [3] can hide the details of vectorization from the end user, these challenges are still encountered during the design and implementation of the libraries themselves.

One alternate solution to relieve application programmers from writing low-level code is to let them focus only on domain-specific applications at a high level, help them to abstract the computational and communication patterns with potential parallelization opportunities, and leverage modern compiler techniques to automatically generate the vectorization codes that fit in the parallel architectures of given accelerators. For example, McFarlin et al. [4] abstract the data-reordering patterns used in the Fast Fourier Transform (FFT) and generate the corresponding SIMD codes for CPUs. Mint and Physis [5], [6] capture stencil computation on GPUs, i.e., computational and communication patterns across a structured grid. Benson et al. [7] focus on abstracting the different algorithms of matrix multiplication by using mathematic symbols and automatically generating sequential and parallel codes for the CPU.

In this paper, we focus on the sorting primitive and propose a framework – Automatic SIMDization of Parallel Sorting (a.k.a ASPaS) – to automatically generate efficient SIMD codes for parallel sorting on x86-based multicore and manycore processors, including CPUs and MIC, respectively. ASPaS takes any sorting network and a given ISA as inputs and automatically produces vectorized sorting code as the output. The code adopts a bottom-up approach to sort and merge segmented data. Since the vectorized sort function puts partially sorted data across different segments, ASPaS gathers the sorted data into contiguous

• *K. Hou, H. Wang, and W. Feng are with the Department of Computer Science, Virginia Tech, Blacksburg, VA, 24060. E-mail: {kaixihou, hwang121, wfeng}@vt.edu.*

regions through a transpose function before the merge stage. Considering the variety of sorting and merging networks[1] [8] that correspond to different sorting algorithms (such as Odd-Even [9], Bitonic [9], Hibbard [10], and Bose-Nelson [11]) and the continuing evolution of instruction sets (such as SSE, AVX, AVX2, and IMCI), it is imperative to provide such a framework to hide the instruction-level details of sorting and allow programmers to focus on the use of the sorting algorithms instead.

ASPaS consists of four major modules: (1) Sorter, (2) Transposer, (3) Merger, and (4) Code Generator. The *SIMD Sorter* takes a sorting network as input and generates a sequence of comparators for the sort function. The *SIMD Transposer* and *SIMD Merger* formalize the data-reordering operations in the transpose and merge functions as sequences of vector-matrix multiplications. The *SIMD Code Generator* creates an ISA-friendly pattern pool containing the requisite data-comparing and reordering primitives, builds those sequences with primitives, and then translates them to the real ISA intrinsics according to the platforms.

We make the following contributions in this paper. First, for **portability**, we propose the ASPaS framework to automatically generate the cross-platform parallel sorting codes using architecture-specific SIMD instructions, including AVX, AVX2, and IMCI. Second, for **functionality**, using ASPaS, we can generate various parallel sorting codes for the combinations of five sorting networks, two merging networks, and three datatypes (integer, float, double) on Intel Ivy Bridge, Haswell CPUs, and Intel Knights Corner MIC. In addition, ASPaS generates the vectorization codes not only for the sorting of array, but also for the sorting of {key,data} pairs, which is a requisite functionality to sort the real-world workloads. Third, for **performance**, we conduct a series of rigorous evaluations to demonstrate how the ASPaS-generated codes can yield performance benefits by efficiently using the vector units and computing cores on different hardware architectures.

For the one-word type,[2] our SIMD codes on CPUs can deliver speedups of up to 4.1x and 6.5x (10.5x and 7.6x on MIC) over the serial sort and merge kernels, respectively. For the two-word type, the corresponding speedups are 1.2x and 2x on CPUs (6.0x and 3.2x on MIC), respectively. Compared with other single-threaded sort implementations, including `qsort` and `sort` from STL [12] and `sort` from Boost [13], our SIMD codes on CPUs deliver a range of speedups from 2.1x to 5.2x (2.5x to 5.1x on MIC) for the one-word type and 1.7x to 3.8x (1.3x to 3.1x on MIC) for the two-word type. Our ASPaS framework also improves the memory access pattern and thread-level parallelism. That is, we leverage the ASPaS-generated SIMD kernels as building blocks to create a multi-threaded sort (via multiway merging). Compared with the `parallel_sort` from Intel TBB [14], ASPaS delivers speedups of up to 2.5x and 1.7x on CPUs (6.7x and 5.0x on MIC) for the one-word type and the two-word type, respectively.

Our previous research [15] first proposes the ASPaS framework to generate the vectorization codes of parallel sorting on Intel MIC. This work extends our previous research in three directions: (1) for portability, we extend the ASPaS framework to support AVX and AVX2 CPUs, e.g., Intel Ivy Bridge and Haswell, respectively; (2) for functionality, we extend ASPaS to vectorize parallel sorting codes for {key, data} pairs required by many real-world applications; (3) for performance, we further optimize our implementations of the code generation by using a cache-friendly organization of generated kernels and an improved multithreading strategy. Finally, we conduct a series of new evaluations on three hardware architectures and demonstrate the significant performance benefits compared to existing parallel sorting tools.

2 BACKGROUND

This section presents (1) a brief overview of the vector ISAs on modern x86-based systems, including the AVX, AVX2, and IMCI. (2) a domain-specific language (DSL) to formalize the data-reordering patterns in our framework, and (3) a sorting and merging network.

2.1 Intel Vector ISAs

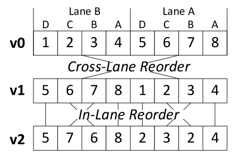
The VPUs equipped in the modern x86-based processors are designed to perform one single operation over multiple data items simultaneously. The width of the vectors and richness of the instruction sets are continuously expanding and evolving, thereby forming the AVX, AVX2, and IMCI.

AVX/AVX2 on CPU: The AVX is first supported by Intel's Sandy Bridge processors and each of their 16 256-bit registers contains two 128-bit lanes (lane `B` and `A` in Fig. 1a), together holding 8 floats or 4 doubles. The three-operands in AVX use a non-destructive form to preserve the two source operands. The AVX2 is available since the Haswell processors and it expands the integer instructions in SSE and AVX and supports variable-length integers. Moreover, AVX2 increases the instructional functionalities by adding, for example, *gather* support to load non-contiguous memory locations and per-element *shift* instructions. In both AVX and AVX2, the data-reordering operations contain permutation within each 128-bit lane and cross the two lanes. The latter is considered more expensive. Fig. 1a shows an example of rearranging data in the same vector. We first exchange the two lanes `B` and `A` and then conduct the in-lane permutation by swapping the middle two elements. Fig. 1b illustrates an another example of using the `unpacklo` instruction to interleave the odd numbers from two input vectors. The specific instructions used in AVX and AVX2 might be different from one another.

IMCI on MIC: The MIC coprocessor consists of up to 61 in-order cores, each of which is outfitted with a new VPU. The VPU state for each thread contains 32 512-bit general registers, eight 16-bit mask registers, and a status register. The IMCI is introduced in accordance with the new VPU. Previous SIMD ISAs, e.g. SSE and AVX, are not supported by the vector architecture of MIC, due to the issues from the wider vector, transcendental instructions, etc. [16].

On MIC, each 512-bit vector is subdivided into four lanes and each lane contains four 32-bit elements. Both of the lanes and elements are ordered as `DCBA`. Fig. 1c illustrates

1. In this paper, we distinguish the sorting network and the merging network.
2. We use the 32-bit Integer datatype as the representative of the one-word type, and the 64-bit Double datatype for the two-word type.



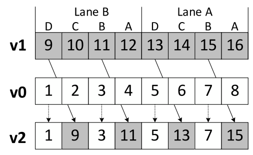
AVX intrinsics

```
v1=_mm256_permute2f128_ps(v0, v0, 0x03);
v2=_mm256_permute_ps(v1, 0xd8);
```

AVX2 intrinsics

```
v1=_mm256_permute2x128_si256(v0, v0, 0x03);
v2=_mm256_shuffle_epi32(v1, 0xd8);
```

(a) CPU: Reorder data within one vector



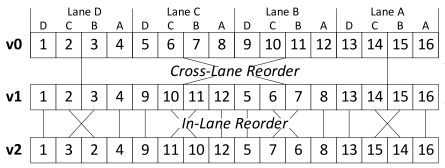
AVX intrinsics

```
v2=_mm256_unpacklo_ps(v0, v1);
```

AVX2 intrinsics

```
v2=_mm256_unpacklo_epi32(v0, v1);
```

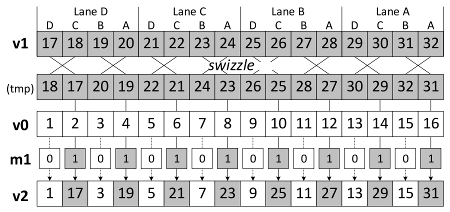
(b) CPU: Reorder data between two vectors


Lane D
Lane C
Lane B
Lane A
D C B A D C B A D C B A D C B A
v0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Cross-Lane Reorder
v1 1 2 3 4 9 10 11 12 5 6 7 8 13 14 15 16
In-Lane Reorder
v2 1 3 2 4 9 11 10 12 5 7 6 8 13 15 14 16

IMCI intrinsics

```
v1=_mm512_permute4f128_epi32
   (v0, _MM_PERM_DBCA);
v2=_mm512_shuffle_epi32
   (v1, _MM_PERM_DBCA);
```

(c) MIC: Reorder data within one vector


Lane D
Lane C
Lane B
Lane A
D C B A D C B A D C B A D C B A
v1 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
swizzle
(tmp) 18 17 20 19 22 21 24 23 26 25 28 27 30 29 32 31
v0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
m1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
v2 1 17 3 19 5 21 7 23 9 25 11 27 13 29 15 31

IMCI intrinsics

```
v2=_mm512_mask_swizzle_epi32
   (v0, m1, v1,
   _MM_SWIZ_REG_BADC);
```

(d) MIC: Reorder data between two vectors

Fig. 1: Rearrange data on Intel CPU and MIC from: (a,c) the same vector register with the permute and shuffle instructions; (b,d) two vector registers with the specific unpack instructions on CPU or masked swizzle instructions on MIC

the data rearrangement in the same vector register with the shuffle and permute intrinsics. The permute intrinsic using `_MM_PERM_DBCA` is for cross-lane rearrangement, which exchanges data in lanes `C` and `B`. The shuffle intrinsic conducts the same operation but on the element-level within each lane. Because the permute and shuffle intrinsics are executed by different components of the hardware, it is possible to overlap the permute and shuffle intrinsics with a pipeline mode [2]. In the design of ASPaS, we use this characteristic to obtain instruction-level overlapping. Fig. 1d shows picking data from two vector registers with the masked swizzle intrinsics. To that end, we use the mask `m1` to select elements from either the swizzled vector of `v1` or the vector `v0`, and then store the result to a blended vector `v2`. The behavior of the mask in Intel MIC is non-destructive, in that no element in source `v0` has been changed if the corresponding mask bit is 0.

2.2 DSL for Data-Reordering Operations

To better describe the data-reordering operations, we adopt the representation of a domain-specific language (DSL) from [17], [18] but with some modification. In the DSL, the first-order operators are adopted to define operations of basic data-reordering patterns, while the high-order operators connect such basic operations into complex ones. Those operators are described as below.

First-order operators (x is an input vector):

S_2 $(x_0, x_1) \mapsto (min(x_0, x_1), max(x_0, x_1))$. The comparing operator resembles the comparator which accepts two arbitrary values and outputs the sorted data. It can also accept two indexes explicitly written in following parentheses.

A_n $x_i \mapsto x_j, 0 \leqslant i, j < n$, iff $A_{ij} = 1$. A_n represents an arbitrary permutation operators denoted as a permutation matrix which has exactly one "1" in each row and column.

I_n $x_i \mapsto x_i, 0 \leqslant i < n$. I_n is the identity operator and outputs the data unchanged as its inputs. Essentially, I_n is a diagonal matrix denoted as $I_n = diag(1, 1, \cdots, 1)$.

L^{km}_m $x_{ik+j} \mapsto x_{jm+i}, 0 \leqslant i < m, 0 \leqslant j < k$. L^{km}_m is a special permutation operator, performing a stride-by-m permutation on the input vector of size km.

High-order operators (A, B are two permutation operators):

($\circ$) The composition operator is used to describe a data flow. $A_n \circ B_n$ means a n-element input vector is first processed by A_n and then the result vector is processed by B_n. The product symbol $\prod$ represents the iterative composition.

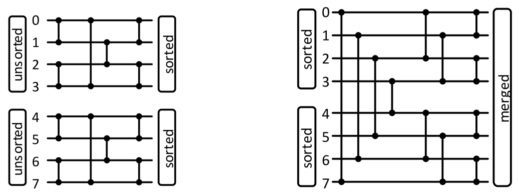
($\oplus$) The direct sum operator is served to merge two operators. $A_n \oplus B_m$ indicates that the first n elements of the input vector is processed by A_n, while the rest m elements follow B_m.

($\otimes$) The tensor product we used in the paper will appear like $I_m \otimes A_n$, which equals to $A_n \oplus \cdots \oplus A_n$. This means the input vector is divided into m segments, each of which is mapped to A_n.

With the DSL, a sequence of data comparing and reordering patterns can be formalized and implemented by a sequence of vector-matrix multiplications. Note that we only use the DSL to describe the data-comparing and data-reordering patterns instead of creating a new DSL.

2.3 Sorting and Merging Network

The sorting network is designed to sort the input data by using a sequence of comparisons, which are planned out in advance regardless of the value of the input data. The sorting network may depend on the merging network to merge pairs of sorted subarrays. Fig. 2a exhibits the Knuth diagram [8] of two identical bitonic sorting networks. Each 4-key sort network accepts 4 input elements. The paired dots represent the comparators that put the two inputs into the ascending order. After threaded through the wires of the network, these 4 elements are sorted. Fig. 2b is a merging network to merge two sorted 4-key vectors to an entirely sorted 8-key vector. Although sorting and merging networks are usually adopted in the circuit designs, it is also suitable for SIMD implementation thanks to the absence of unpredictable branches.



(a) sorting networks (b) merging network

Fig. 2: Bitonic sorting and merging networks (a) Two 4-key sorting networks (b) One 8-key merging network

In this paper, the sorting and merging networks are represented by a list of comparators, each of which is denoted as CMP(x, y) that indicates a comparison operation between x-th and y-th elements of the input data.

3 Framework and Generalized Patterns

The ASPaS parallel sorting uses an iterative bottom-up scheme to sort and merge segmented data. Alg. 1 illustrates the scheme: First, the input data are divided into contiguous segments, each of whose size equals to the built-in SIMD width to the power of 2. Second, these segments are loaded into vector registers for sorting with the functions of `aspas_sort` and `aspas_transpose` (the *sort* stage in loop of ln. 3). Third, the algorithm will merge neighboring sorted segments to generate the output by iteratively calling the function of `aspas_merge` (the *merge* stage in loop of ln. 9). The functions of `load`, `store`, `aspas_sort`, `aspas_transpose`, and `aspas_merge` will be generated by ASPaS using the platform-specific intrinsics. Since the `load` and `store` can be directly translated to the intrinsics once the ISA is given, we focus on other three kernel functions with the prefix `aspas_` in the remaining sections.

Algorithm 1: ASPaS Parallel Sorting Structure

```
   /* w is the SIMD width                                          */
1  Function aspas::sort(Array a)
2      Vector $v_1, ..., v_w$;
3      foreach Segment $seg$ in $a$ do
4          // load $seg$ to $v_1, ..., v_w$
5          aspas_sort($v_1, ..., v_w$);
6          aspas_transpose($v_1, ..., v_w$);
7          // store $v_1, ..., v_w$ to $seg$
8      Array $b$ ← new Array[$a$.size];
9      for $s$ ←w; $s < a$.size; $s$*=2 do
10         for $i$ ←0; $i < a$.size; $i$+=2*$s$ do
11             // merge subarrays $a + i$ and $a + i + s$
12             // to $b + i$ by calling Function aspas::merge()
13         // copy $b$ to $a$
14     return;
15 Function aspas::merge(Array a, Array b, Array out)
16     Vector $v, u$;
17     // $i_0, i_1, i_2$ are offset pointers on $a, b, out$
18     // load w numbers from $a$ to $v$
19     // load w numbers from $b$ to $u$
20     aspas_merge($v, u$);
21     // store $v$ to $out$ and update $i_0, i_1, i_2$
22     while $i_0 \leqslant a$.size and $i_1 \leqslant b$.size do
23         if $a[i_0] \leqslant b[i_1]$ then
24             // load w numbers from $a + i_0$ to $v$
25         else
26             // load w numbers from $b + i_1$ to $v$
27         aspas_merge($v, u$);
28         // store $v$ to $out + i_2$ and update $i_0, i_1, i_2$
29     // process the remaining elements in $a$ or $b$
30     return;
```

Fig. 3 depicts the structure of the ASPaS framework to generate the `sort` function. Three modules —*SIMD Sorter*, *SIMD Transposer*, and *SIMD Merger* — are responsible for building the sequences of comparing and data-reordering operations for the aforementioned kernel functions. Then, these sequences are mapped to the real SIMD intrinsics through the module of *SIMD Code Generator*, and the codes will be further optimized from the perspectives of memory access pattern and thread-level parallelism (in Sec. 4).

3.1 SIMD Sorter

The operations of `aspas_sort` are taken care by the *SIMD Sorter*. As shown in Fig. 4, `aspas_sort` loads n-by-n elements into n n-wide vectors and threads them through the

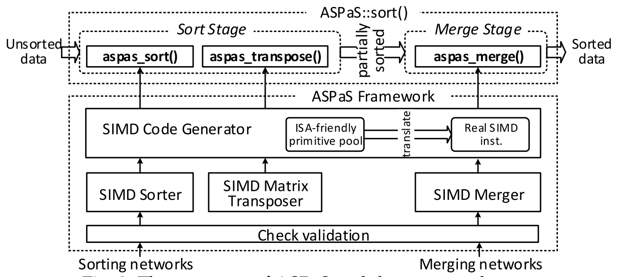


Fig. 3: The structure of ASPaS and the generated `sort`

given sorting network, leading to the data sorted for the aligned positions across vectors. Fig. 5 presents an example of a 4-by-4 data matrix stored in vectors and a 4-key sorting network (including its original input macros). Here, each dot represents one vector and each vertical line indicates a vector comparison. The six comparisons rearrange the original data in ascending order in each column. Fig. 5 also shows the data dependency between these comparators. For example, CMP(0,1) and CMP(2,3) can be issued simultaneously, while CMP(0,3) can occur only after these two. It is natural to form three groups of comparators for this sorting network. We also have an optimized grouping mechanism to minimize the number of groups for other more complicated sorting networks. For more details, please refer to the original paper [15].

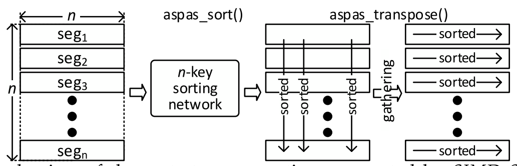


Fig. 4: Mechanism of the sort stage: operations generated by *SIMD Sorter* and *SIMD Transposer*

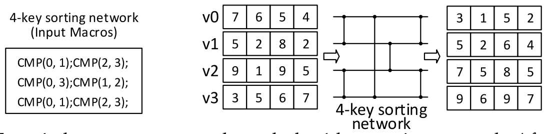


Fig. 5: Four 4-element vectors go through the 4-key sorting network. Afterwards data is sorted in each column of the matrix.

Since we have the groups of comparators, we can generate the vector codes for the `aspas_sort` by keep two sets of vector variables `a` and `b`. All the initial data are stored in the vectors of set `a`. Then, we jump to the first group of the sorting network. For each comparator in the current group, we generate the vector operations to compare relevant vector variables, and store the results to the vectors in set `b`. The unused vectors are directly copied to set `b`. For the next group, we flip the identities of `a` and `b`. Therefore, the set `b` becomes the input, and the results will be stored back to `a`. This process continues until all groups of the sorting network have been handled. All the vector operations in the `aspas_sort` will be mapped to the ISA-specific intrinsics (e.g., `_mm256_max` and `_mm256_min` on CPUs) later by the *SIMD Code Generator*. At this point, the data is partially sorted but stored in column-major order.

3.2 SIMD Transposer

As illustrated in Fig. 4, the `aspas_sort` function has scattered the sorted data across different vectors. The next task is to gather them into the same vectors (i.e., rows) for further operations. There are two alternative ways to achieve the gathering: one directly uses the gather/scatter SIMD intrinsics; and the other uses the in-register matrix transpose over loaded vectors. The first scheme provides a convenient means to handle the non-contiguous data in memory, but with the penalty of high latency of accessing scattered locations. The second one avoids latency penalty at the expense of using complicated data-reordering operations. Considering the high latency of the gather/scatter intrinsics and the incompatibility with architectures that do not support gather/scatter intrinsics, we adopt the second scheme in the *SIMD Transposer*. To decouple the binding between the operations of matrix transpose and the dedicated intrinsics with various SIMD widths, we formalize the data-reordering operations using the sequence of permutation operators. Subsequently, the sequence will be handed over to the *SIMD Code Generator* to generate the platform-specific SIMD codes for the `aspas_transpose` function.

$$\prod_{j=1}^{t-1}(L_2^{2^t} \circ (I_{2^{t-j-1}} \otimes L_{2^j}^{2^{j+1}}) \circ (I_{2^{t-j}} \otimes L_2^{2^j}) \circ L_{2^{t-1}}^{2^t}[v_{id}, v_{id+2^{j-1}}]) \quad (1)$$

Eq. 1 gives the operations performed on the preloaded vectors for the matrix transpose, where w is the SIMD width, $t = log(2w)$, and for each j, $id \in \{i \cdot 2^j + n | 0 \leqslant i < \frac{w}{2^j}, 0 \leqslant n < 2^{j-1}\}$, which will form $\frac{w}{2^j} \cdot 2^{j-1} = \frac{w}{2}$ pairs of operand vectors. The sequence of permutation operators preceded each operand pair will be applied on them. The square brackets wrap these pairs of vectors.

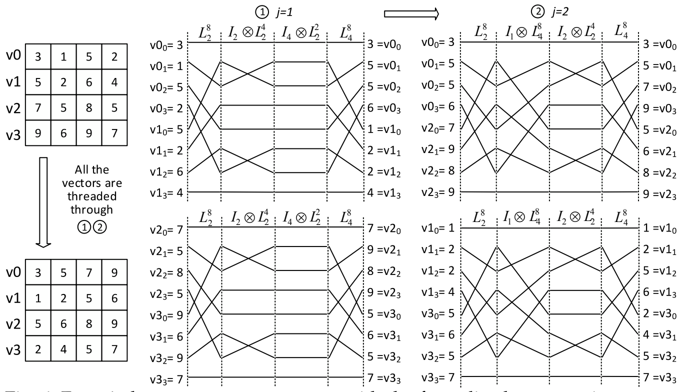


Fig. 6: Four 4-element vectors transpose with the formalized permutation operators of DSL.

Fig. 6 illustrates an example of in-register transpose with $w = 4$. The elements are preloaded into vectors $v0$, $v1$, $v2$, and $v3$ and have been already sorted vertically. $t - 1 = 2$ indicates that there are 2 steps denoted as ① and ② in the figure. For the step $j = 1$, the permutation operators are applied on the pairs $[v0, v1]$ and $[v2, v3]$; and for $j = 2$, the operations are on the pairs $[v0, v2]$ and $[v1, v3]$. After the the vectors go through the two steps accordingly, the matrix is transposed, and the elements are gathered in the same vectors.

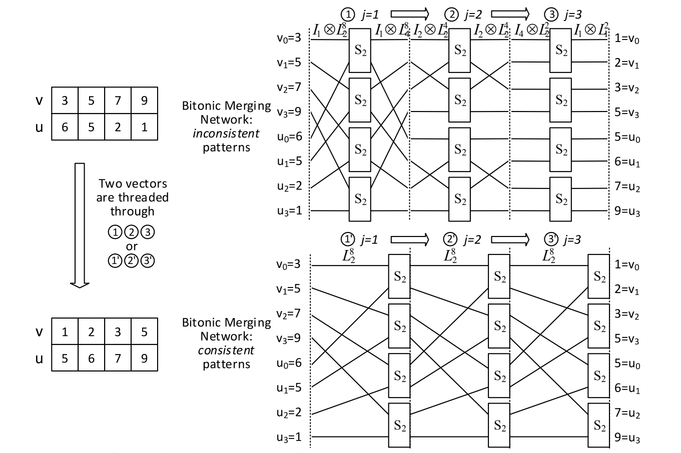
3.3 SIMD Merger

For now, the data have been sorted in each segment thanks to the `aspas_sort` and `aspas_transpose`. Then, we use the `aspas_merge` to combine pairs of sorted data into a larger sequence iteratively. The *SIMD Merger* is responsible for its comparison and data-reordering operations according to given merging networks, e.g., odd-even and bitonic networks. In ASPaS, we select the bitonic merging network for three reasons: (1) the bitonic merging network can be easily scaled to any 2^n-sized keys; (2) there is no idle element in the input vectors for each comparison step; and (3) symmetric operations can facilitate the vector instruction selection (discussed in Sec. 4.1). As a result, it is much easier to vectorize the bitonic merging network than others. In terms of implementation, we have provided two variants of the bitonic merging networks [18] to achieve the same functionality. Their data-reordering operations can be formalized, as shown below.

$$\prod_{j=1}^{t}(I_{2^{j-1}} \otimes L_2^{2^{t-j+1}}) \circ (I_{2^{t-1}} \otimes S_2) \circ (I_{2^{j-1}} \otimes L_{2^{t-j}}^{2^{t-j+1}})[v, u] \quad (2)$$

$$\prod_{j=1}^{t} L_2^{2^t} \circ (I_{2^{t-1}} \otimes S_2)[v, u] \quad (3)$$

Similar with Sec. 3.2, $t = log(2w)$ and w is the SIMD width. The operand vectors v and u represent two sorted sequences (the elements of vector u are inversely stored in advance). In Eq. 2, the data-reordering operations are controlled by the variable j and varies in each step, while in Eq. 3, the permutation operators are independent with j, thereby leading to the uniform permutation patterns in each step. Hence, we label the pattern in Eq. 2 as the inconsistent and that in Eq. 3 as the consistent. These patterns will be transmitted to and processed by the *SIMD Code Generator* to generate the `aspas_merge` function. We will present the performance comparison of these two patterns in Sec. 5.



v 3 5 7 9
u 6 5 2 1
Two vectors are threaded through ①②③ or ①②③
v 1 2 3 5
u 5 6 7 9
Bitonic Merging Network: inconsistent patterns
Bitonic Merging Network: consistent patterns
① j=1
② j=2
③ j=3
S_2

Fig. 7: Two formalized variants of bitonic merging networks: the inconsistent pattern and the consistent pattern. All elements in vector v and u are sorted, but inversed in vector u.

Fig. 7 presents an example of the two variants of bitonic merging networks under the condition of $w = 4$. The data-reordering operations from the inconsistent pattern keep changing for each step, while those from the consistent one stay identical. Though the data-reordering operations of the two variants are quite different, both are able to successfully achieve the same merging functionality within the same number of steps, which is actually determined by the SIMD width w.

4 Code Generation and Optimization

In the section, we will first show the searching mechanism of ASPaS framework to find out the most efficient SIMD instructions. Then, the generated codes will be optimized to take advantage of memory hierarchy and multi/manycore resources of x86-based systems.

4.1 SIMD Code Generator

This module in ASPaS accepts the comparison operations from *SIMD Sorter* and the data-reordering operations from *SIMD Transposer* and *SIMD Merger* in order to generate the real ISA-specific vector codes. We will put emphasis on finding the most efficient intrinsics for the data-reordering operations, since mapping comparison operations to SIMD intrinsics is straightforward. In the module, we first define a SIMD-friendly primitive pool based on the characteristics of the data-ordering operations, then dynamically build the primitive sequences according to the *matching score* between what we have achieved on the way and the target pattern, and finally translate the selected primitives into the real intrinsics for different platforms.

4.1.1 Primitive Pool Building

Some previous research, e.g., the automatic Fast Fourier transform (FFT) vectorization [4], uses the exhaustive and heuristic search on all possible intrinsics combinations, which is time-consuming, especially for the richer instruction sets, such as IMCI. To circumvent the limitation, we first build a primitive pool to prune the search space and the primitives are supposed to be SIMD-friendly. The most notable feature of the data-reordering operations for the transpose and merge is the symmetry: all the operations applied on the first half of input are equivalent with those on the second half in a mirror style. We assume that all the components of the sequences to achieve these operations are also symmetric. We categorize these components as (1) the primitives for the symmetric permute operations on the same vector and (2) the primitives for the blend operations across two vectors.

Permute Primitives: Considering 4 elements per lane (e.g., Integer or Float) or 4 lanes per register (e.g., IMCI register), there are $4^4 = 256$ possibilities for either intra-lane or inter-lane permute operations. However, only those permutations without duplicated values are useful in our case, reducing the possibilities to $4! = 24$. Among them, merely 8 symmetric data-reordering patterns will be selected, i.e. `DCBA` (original order), `DBCA`, `CDAB`, `BDAC`, `BADC`, `CADB`, `ACBD`, and `ABCD`, in which each letter denotes an element or a lane. If we are working on 2 elements per lane (e.g., Double) or 2 lanes per register (e.g., AVX register), there are two symmetric patterns without duplicated values, i.e. `BA` (original order) and `AB`.

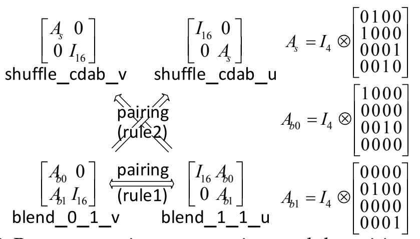
Blend Primitives: While blending two vectors into one, the elements are supposed to be equally and symmetrically distributed from the two input vectors. Hence, we can boil down the numerous mask modifiers to only a few. We define a pair of blend patterns $(0\_2^i,\ 2^i\_2^i)$, where $0 \leqslant i < log(w)$ and w is the vector width. Each blend pattern in the pair represents a 2^{i+1}-bit stream. The first number 0 or 2^i denotes the offset of the first set bit, and the second number 2^i is the number of consecutive set bits. All the other bits are filled with clear bits. The bit streams need to be extended to the vector width by duplicating themselves $\frac{w}{2^{i+1}}$ times. For example, if the w equal to 16, there are 4 possible pairs of patterns: (0_1, 1_1), (0_2, 2_2), (0_4, 4_4), and (0_8, 8_8). Among them, the pair (0_2, 2_2) corresponds to $i = 1$, representing the bit streams $(1100)_4$ and $(0011)_4$ (The subscript 4 means the repetition times).

Now, we further categorize the primitives into 4 types based on permute or blend and intra-lane or inter-lane. Tab. 1 illustrates the categories and associative exemplar operations, where the vector width w is set to 8 (2 lanes) for clarity.

TABLE 1: Primitive Types

Type #	Type	Example (vector_width=8)
0	intra-lane-permute	ABCDEFGH$\rightarrow$BADCFEHG (cdab)
1	inter-lane-permute	ABCDEFGH$\rightarrow$EFGHABCD (−−ab)
2	intra-lane-blend	ABCDEFGH | IJKLMNOP$\rightarrow$ABKLEFOP (2_2)
3	inter-lane-blend	ABCDEFGH | IJKLMNOP$\rightarrow$IJKLEFGH (0_4)

The primitives are materialized into permutation matrices in ASPaS. Since the blend primitives always operate on two vectors (concatenated as one $2w$ vector), the dimensions of the blend permutation matrices are expanded to $2w$ by $2w$ as well. Accordingly, for the permute primitives, we pair an empty vector to the single input vector and specify the primitive works on the first vector v or the second vector u. Therefore, for example, if $w = 16$, there are 32=8(permute primitives)$*$2(intra-lane or inter-lane)$*$2(operating on v or u) and 8(4 pairs of the blend primitives) permutation matrices. Fig. 8 illustrates examples of the permutation matrices. The matrix "shuffle_cdab_v" and "shuffle_cdab_u" correspond to the same permute primitive on the halves of the concatenated vectors. The matrix "blend_0_1_v" and "blend_1_1_u" correspond to one pair of blend primitives (0_1, 1_1). So far, 4 sub-pools of permutation matrices are created according to the 4 primitive types.



$\begin{bmatrix} A_s & 0 \\ 0 & I_{16} \end{bmatrix}$
shuffle_cdab_v
$\begin{bmatrix} I_{16} & 0 \\ 0 & A_s \end{bmatrix}$
shuffle_cdab_u
$A_s = I_4 \otimes \begin{bmatrix} 0100 \\ 1000 \\ 0001 \\ 0010 \end{bmatrix}$
pairing (rule2)
$A_{b0} = I_4 \otimes \begin{bmatrix} 1000 \\ 0000 \\ 0010 \\ 0000 \end{bmatrix}$
$\begin{bmatrix} A_{b0} & 0 \\ A_{b1} & I_{16} \end{bmatrix}$
pairing (rule1)
$\begin{bmatrix} I_{16} & A_{b0} \\ 0 & A_{b1} \end{bmatrix}$
$A_{b1} = I_4 \otimes \begin{bmatrix} 0000 \\ 0100 \\ 0000 \\ 0001 \end{bmatrix}$
blend_0_1_v
blend_1_1_u

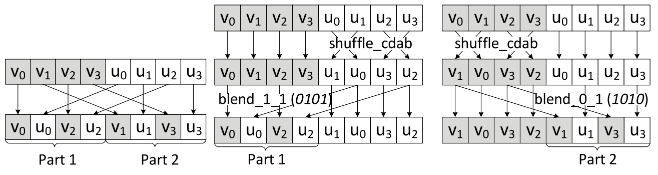
Fig. 8: Permute matrix representations and the pairing rules

4.1.2 Sequence Building

Two rules are used in the module to facilitate the searching process. They are based on two observations from the formalized data-reordering operations illustrated in Eq. 1, Eq. 2, and Eq. 3. Obs.1 The same data-reordering operations are always conducted on two input vectors. Obs.2 The permute operations always accompany the blend operations to keep the symmetric pattern. Fig. 9 exhibits the symmetric patterns, which are essentially the first step in Fig. 6. The default blend is limited to pick elements from aligned positions of two input vectors, while the symmetric blend can achieve an interleaving mode by coupling permute primitives with blend primitives, as the figure shown. Hence, the usage of the two rules in the sequence building algorithm are described as below.

Rule 1: when a primitive is selected for one vector v, pair the corresponding primitive for the other vector u. For a permute primitive, the corresponding permute has the totally same pattern; while for a blend primitive, the corresponding blend has the complementary blend pattern (i.e. the bit stream, which has already been paired).

Rule 2: when a blend primitive is selected, pair it with the corresponding permute primitive: pair the intra-lane-permute of swapping adjacent elements (`CDAB`) for (0_1, 1_1) blend, the intra-lane-permute of swapping adjacent two elements (`BADC`) for (0_2, 2_2), the inter-lane-permute of swapping adjacent lanes (`CDAB`) for (0_4, 4_4), and the inter-lane-permute of swapping adjacent two lanes (`BADC`) for (0_8, 8_8).



(a) `sym-blend` (b) `sym-blend(details)`

Fig. 9: Symmetric blend operation and its pairing details

The sequence building algorithm targets at generating sequences of primitives to achieve given data-reordering patterns for Eq. 1, Eq. 2, and Eq. 3. Two w-sized input vectors of v and u are used and concatenated into the vec_{inp}. Its initial elements are set to the default indices (from 1 to $2w$). The vec_{trg} is the target derived by applying the given data-reordering operators on the vec_{inp}. Then, the building algorithm will select the permutation matrices from the primitive pool, do the vector-matrix multiplications over the vec_{inp}, and check whether the intermediate result vec_{im} approximates the vec_{trg} by using our defined two *matching scores*:

l-score lane-level matching score, accumulate by one when the corresponding lanes have exactly same elements (no matter orders).

e-score element-level matching score, increase by one when the element matches its counterpart in the vec_{trg}.

Suppose we have a vector of w (vector width) and e (number of elements per lane), the maximum l-score equals to $2w/e$ when all the aligned lanes from two vectors match, while the maximum e-score is $2w$ when all the aligned elements match. With the matching scores, the process of sequence building is transformed to finding the maximum scores. For example, if we have the input "AB|CD|EF|GH" and the output "HG|DC|FE|BA" (assuming four lanes and two elements per lane), we first search primitives for the inter-lane reordering, e.g, from "AB|CD|EF|GH" to "GH|CD|EF|AB", and then search primitives for the intra-lane reordering and reach to, e.g., from "GH|CD|EF|AB" to "HG|DC|FE|BA". By checking the primitives hierarchically, we add those primitives increasing l-score or e-score and thus approximate to the desired output pattern.

Alg. 2 shows the pseudocode of the sequence building algorithm. The input contains the aforementioned vec_{inp} and vec_{trg}. The output $seqs_{\mathsf{ret}}$ is a container to hold the built sequences of primitives, which will be translated to the real ISA intrinsics soon. The $seqs_{\mathsf{cand}}$ is to store candidate sequences and initialized to contain a ø sequence. First, the algorithm checks the initial vec_{inp} with the vec_{trg} and get the l-score. If it equals to $2w/e$, meaning aligned lanes have already matched, we only need to select "intra-lane-permute" primitives (ln. 4) to improve the e-score. Otherwise, we will work on the sub-pools of type 1, 2, or 3 in a round-robin manner. In the while loop, for each sequence in $seqs_{\mathsf{cand}}$, we first calculate the l_score_{old}, and then we will calculate the l_score_{new} by tentatively adding primitives one by one from the current sub-pool. If the primitive $prim$ comes from the "inter-lane-permute", we produce the paired permute primitive $prim_{\mathsf{prd}}$ based on the Rule 1 (ln. 14). If $prim$ is from the blend types, we produce the paired blend primitive $prim_{\mathsf{prd}}$ based on the Rule 1 and then find their paired permute primitives $perm_0$ and $perm_1$ based on the Rule 2 (ln. 18-20). The two rules help to form the symmetric operations.

After the selected primitives have been applied, which corresponds to several vector-matrix multiplications, we can get a vec_{upd}, leading to a new l-score l_score_{new} compared to vec_{trg} (ln. 25). If the l-score is increased, we add the sequence of the selected primitives to $seqs_{\mathsf{cand}}$ for further improvement. The threshold (ln. 7) is a configuration parameter to control the upper bound of how many iterations the algorithm can tolerate, e.g., we set it to 9 in the evaluation in order to find the sequences as many as possible. Finally, we use `PickLaneMatched` to select those sequences that can make l-score equal to $2w/e$, and go to the "intra-lane-permute" selection (ln. 32), which can ensure us the complete sequences of primitives.

4.1.3 Primitives Translation

Now, we can map the sequences from the $seqs_{\mathsf{ret}}$ to the real ISA intrinsics. Although the vector ISAs from CPU or MIC platforms are distinct from one another, we can still find desired intrinsics thanks to the SIMD-friendly primitives. If there are multiple selections to achieve same primitive, we always prefer the selection having least intrinsics.

On MIC: if there are multiple shortest solutions exist, we use the interleaved style of inter-lane and intra-lane primitives, which could be executed with a pipeline mode on MIC as discussed in Sec. 2.1. For the primitives from "intra-lane-permute" and "inter-lane-permute", we directly map them into vector intrinsics of `_mm512_shuffle` and `_mm512_permute4f128` with appropriate permute parameters. For the primitives from "intra-lane-blend" and "inter-lane-blend", we map them to the masked variants of permute intrinsics `_mm512_mask_shuffle` and `_mm512_mask_permute4f128`. The masks are derived from their blend patterns. Furthermore, when a primitive is from "intra-lane" and its parameter is supported by the swizzle intrinsics, we will use the light-weighted swizzle intrinsics to optimize the performance.

Algorithm 2: Sequence Building Algorithm

```
Input: $vec_{inp}$, $vec_{trg}$
Output: $seqs_{ret}$
Sequences $seqs_{cand}$ ← new Sequences(∅);    // put an null sequence
Int $l\_score_{init}$ ←LaneCmp($vec_{inp}$, $vec_{trg}$);
if $l\_score_{init}$=2w/e then
    $seqs_{ret}$ ← InstSelector($seqs_{cand}$,Type[0]);
else
    $i$ ←1;
    while not Threshold() do
        $ty$ ←Type[$i$];
        foreach Sequence $seq$ in $seqs_{cand}$ do
            $vec_{im}$ ←Apply($vec_{inp}$, $seq$);
            $l\_score_{old}$ ←LaneCmp($vec_{im}$, $vec_{trg}$);
            foreach Primitive $prim$ in $ty$ do
                if $n$=1 then
                    $prim_{prd}$ ←Pair($prim$, RULE1);
                    $vec_{upd}$ ←Apply($vec_{im}$, $prim + prim_{prd}$);
                    $seq_{ext}$ ←$prim + prim_{prd}$;
                else
                    $prim_{prd}$ ←Pair($prim$, RULE1);
                    $perm_0$ ←Pair($prim$, RULE2);
                    $perm_1$ ←Pair($prim_{prd}$, RULE2);
                    $vec_{upd0}$ ←Apply($vec_{im}$, $perm_0 + prim$);
                    $vec_{upd1}$ ←Apply($vec_{im}$, $perm_1 + prim_{prd}$);
                    $vec_{upd}$ ←Combine($vec_{upd0}$, $vec_{upd1}$);
                    $seq_{ext}$ ←$perm_0 + prim + perm_1 + prim_{prd}$;
                $l\_score_{new}$ ←LaneCmp($vec_{upd}$, $vec_{trg}$);
                if $l\_score_{new} > l\_score_{old}$ then
                    $seqs_{buf}$.add($seq + seq_{ext}$);
        $seqs_{cand}$.append($seqs_{buf}$);
        $seqs_{buf}$.clear();
        $i$ ←((++$i$)-1)%3+1;
    $seqs_{sel}$ ←PickLaneMatchedSeqs($seqs_{cand}$);
    $seqs_{ret}$ ← InstSelector($seqs_{sel}$,Type[0]);
Function InstSelector(Sequences $seqs_{cand}$,Type $ty$)
    foreach Sequence $seq$ in $seqs_{cand}$ do
        $vec_{im}$ ←Apply($vec_{inp}$, $seq$);
        foreach Primitive $prim$ in $ty$ do
            $prim_{prd}$ ←Pair($prim$, RULE1);
            $vec_{upd}$ ←Apply($vec_{im}$, $prim + prim_{prd}$);
            $e\_score$ ←ElemCmp($vec_{upd}$, $vec_{trg}$);
            if $e\_score$=2w then
                $seqs_{ret}$.add($seq + prim + prim_{prd}$);
    return $seqs_{ret}$;
```

On CPU: for the primitives from "intra-lane-permute" and "inter-lane-permute", we map them into vector intrinsics of AVX's `_mm256_permute` (or AVX2's `_mm256_shuffle`[3]) and `_mm256_permute2f128` with appropriate permute parameters. For the primitives of blend primitives, we need to find specific combinations of intrinsics, since there are no similar mask mechanisms in AVX or AVX2 as IMCI. For "intra-lane-blend" primitives, if the blend pattern is picking interleaved numbers from two vectors, e.g., 0101 or 1010, we use the `_mm256_unpacklo` and `_mm256_unpackhi` to unpack and interleave the neighboring elements. In contrast, for the patterns that select neighboring two elements, e.g., 0011 or 1100, we use AVX's `_mm256_shuffle`, which can take two vectors as input and pick every two elements from each input. For the "inter-lane-blend" primitives, we use `_mm256_permute2f128`. Note that, since many intrinsics in AVX only support operations on floating point elements, we have to cast the datatypes if we are working on integers; while on AVX2, we can directly use intrinsics handling integers without casting. As a result, for parallel sorting on integers, the generated codes on AVX2 may use much less intrinsics than those on AVX.

3. AVX's `_mm256_shuffle` is different from AVX2's and it reorders data from two input vectors

4.2 Organization of the ASPaS Kernels

So far, we have generated three building kernels in ASPaS: `aspas_sort()`, `aspas_transpose()`, and `aspas_merge()`. As shown in Fig. 10, we carefully organize these kernels to form the `aspas::sort` as illustrated in Alg. 1. Note that this figure shows the sort, transpose, and merge stages on each thread and the multithreaded implementation will be discussed in the next subsection. First, the `aspas_sort()` and `aspas_transpose()` are performed on every segment of the input to create a partially sorted array. Second, we enter the *merge* stage. Rather than directly merging the sorted segments level by level in our previous research [15], we adopt the multiway merge [19], [20]: merge the sorted segments for multiple levels in each block, the cache-sized trunk, to fully utilize the data in the cache until we move to the next block. This strategy is cache-friendly, since it avoids frequently swapping data in and out the cache. When the merged segments are small enough to fit into the LLC, which is usual in first several levels, we take this multiway merge strategy. For the large segments in the later levels, we fall back to the two-way merge. Similar to existing libraries, e.g., STL, Boost, and TBB, we also provide the merge functionality for programmers as a separate interface. The interface `aspas::merge` is similarly organized as `aspas::sort` shown in the figure but only uses `aspas_merge()`.

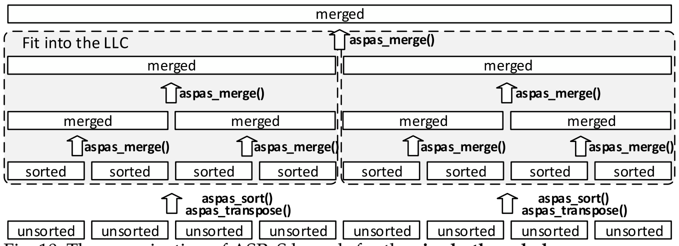


Fig. 10: The organization of ASPaS kernels for the **single-threaded** `aspas::sort`

4.3 Thread-level Parallelism

In order to maximize the utilization of multiple cores of modern x86-based systems, we integrate the `aspas::sort` and `aspas::merge` with the thread-level parallelism using Pthreads. Initially, we split the input data into separate parts, each of which is assigned to one thread. All the threads can sort their own parts using the `aspas::sort` independently. Then, we merge each thread's sorted part together. The simplest way might be assigning half of the threads to merge two neighboring sorted parts into one by iteratively calling the `aspas::merge` until there is only one thread left. However, this method significantly underutilizes the computing resources. For example, in the last level of merging, there is only one thread merging two trunks but all the other threads are idle. Therefore, for the last several levels of merging, we adopt MergePath [21] to let multiple threads merge two segments. Assume for each two sorted segments with the lengths of m and n, we have k threads working on them. First, each thread calculates the i/k-th value in the imagined merged array without actually merging the inputs, where the i is the thread index. This step can be done in O(log(m+n)). Second, we split the workloads into k exclusive and balanced portions according to the k

splitting values. Finally, each thread can merge their assigned portions independently. Note, this strategy is capable of minimizing the data access overhead on remote memory bank of NUMA architecture, since the array is equally split and stored in each memory bank and a thread will first merge data in the local memory region, and then on demand access remote data in a serial mode [19]. In the evaluation, our multithreaded version adopts this optimized design.

4.4 Sorting of {key,data} Pairs

In many real-world applications, sorting is widely used to reorder some specific data structures based on their keys. To that end, we extend ASPaS with this functionality: generate the vectorization codes to sort {*key*, *data*} pairs, where the *key* represents the target for sorting and the *data* is the address to the data structures containing that *key*. The research work [20] proposes two strategies to sort {*key*, *data*} pairs. The first strategy to sort {*key*, *data*} pairs is to pack the relative *key* and *data* into a single entry. Then, sorting the entries is equivalent to sorting the keys, since the keys are placed in the high bits. However, if the sum of lengths of *key* and *data* exceeds the maximum length of the built-in data types, it is non-trivial to carry this strategy out. The second strategy is to put the keys and data into two separate arrays. While sorting the keys, the comparison results are stored as masks that will be used to control the data-reordering of associative data. In this paper, we use the second method. Differed from [20], which focuses on the 32-bit key and data, ASPaS is able to handle different combinations of 32/64-bit keys and 32/64-bit data and their varied data-reordering patterns accordingly.

For implementation, ASPaS uses *compare* intrinsics rather than *max/min* intrinsics to get appropriate masks. The masks may need be stretched or split depending on the differences between the lengths of keys and data. With the masks, we use *blend* intrinsics on both key vectors and data vectors to reorder elements. Tab. 2 shows how the building modules are used to find the desired intrinsics for key and data vectors, respectively.

In the table, w represents the number of keys the built-in vector can hold. The modules are in the format of *modName*[*count*](*vlist*), which means generating the *modName* data-reordering intrinsics for vectors in *vlist* and each vector contains *count* elements. There are three possible combinations for different keys and data: (1) When the key and data has the same length, we use the totally same data-reordering intrinsics on the key and data vectors. (2) When the data length doubles the key length, we correspondingly double the number of vectors to hold the enlarged data values. Then, the building modules are performed on halves of the input data vectors as shown in the table: for transpose, we need to use four times intrinsics on data vectors than key vectors to transpose four blocks of data vectors, and change the layout of data vectors from [00, 01, 10, 11] to [00, 10, 01, 11]; for merge, we need to double the intrinsics on data vectors than key vectors since the input vectors are doubled. (3) When the key length exceeds the data length, we take distinct strategies according to the platforms. On CPU, we simply use the SSE vector ISA, because of the backward compatibility of AVX. On MIC, since the platform doesn't support previous vector ISA, we keep the effective values always in the first halves of each 512-bit vectors.

One may wonder why we need to reorder the *data* along with the *key* in each step rather than do it only in the final step. The reason is that this alternative requires an additional "index" vector to keep track of *key* movement, which occurs during each step of reordering of the *keys*. Thus, it is same to our strategy because the *data* in our method is the address to the real data structure. Moreover, the reordering of *data* in our method has adopted ISA intrinsics for vectorization, which can avoid the irregular memory access. In the perspective of performance, the execution time of sorting {key,data} pairs grows asymptotically compared to sorting the pure key array. Henceforth, we will focus on the performance analysis of sorting pure key array in the evaluation section.

5 Performance Analysis

ASPaS supports major built-in data types, i.e., integers, single and double precision floating point numbers. In our evaluation, we use the Integer for the one-word type (32-bit) and the Double for the two-word type (64-bit). Our codes use different ISA intrinsics according to the different platforms. Tab. 3 shows the configurations of the three platforms with Intel Ivy Bridge (IVB), Haswell (HSW), and Knights Corner (KNC), respectively. The ASPaS programs are implemented in C++11 and compiled using Intel compiler *icpc* 15.3 for HSW and KNC and *icpc* 13.1 for IVB. On CPUs, we use the compiler options of *-xavx* and *-xCORE-AVX2* to enable AVX and AVX2, respectively. On MIC, we run the experiments using the *native* mode and compile the codes with *-mmic*. All codes in our evaluations are optimized in the level of *-O3*. All the input data are generated randomly ranging from 0 to the data size, except in Sec. 5.5. This paper focuses on the efficiency of vectorization; and we show detailed performance analysis on a single thread in most sections, while Sec. 5.4 evaluates the best vectorized codes in a multi-core design.

5.1 Performance of Different Sorting Networks

We first test the performance of the `aspas_sort` and `aspas_merge` kernels, whose implementation depends on the input sorting and merging networks. For brevity, we only show the graphical results of Integer datatype. We repeat the execution of the kernels for 10 million times and report the total seconds in Fig. 11.

In the *sort* stage, ASPaS can accept any type of sorting networks and generate the `aspas_sort` function. We use five sorting networks, including Hibbard (HI) [10], Odd-Even (OE) [9], Green (GR) [22], Bose-Nelson (BN) [11], and Bitonic (BI) [9]. In Fig. 11, since GR cannot take 8 elements as input, the performance for it on CPUs is not available. The labels of x-axis also indicate how many comparators and groups of comparators in each sorting network are. On CPUs, the sorting networks have same number of comparators except the BI sort, thereby yielding negligible time difference with a slight advantage to BN sort on IVB. On MIC, GR sort has the best performance that stems from the less comparators and groups, i.e., (60, 10). Although

TABLE 2: The building modules to handle the data-reordering for {key,data} pairs in ASPaS

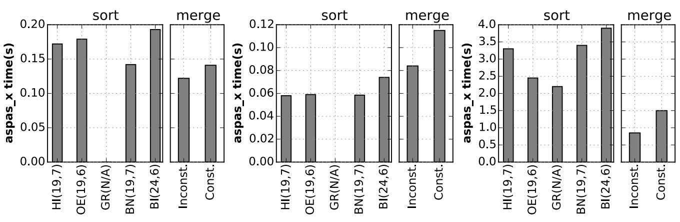
{key,data}	Input (key)	Building Modules (key)	Input (data)	Building Modules (data)
32-bit, 32-bit	$v_0,v_1,...,v_{w-1}$	Transpose[w]$(v_0,v_1,...,v_{w-1})$	$v'_0,v'_1,...,v'_{w-1}$	Transpose[w]$(v'_0,v'_1,...,v'_{w-1})$
64-bit, 64-bit	v,u	Merge_Reorder[w](v,u)	v',u'	Merge_Reorder[w](v',u')
32-bit, 64-bit	$v_0,v_1,...,v_{w-1}$	Transpose[w]$(v_0,v_1,...,v_{w-1})$	$v'_0,v'_1,v'_2,v'_3,$ $...,v'_{2w-2},v'_{2w-1}$	$\begin{bmatrix}\text{Transpose[w/2]}(v'_0,v'_2,...,v'_{w-2}),\ \text{Transpose[w/2]}(v'_{w+0},v'_{w+2},...,v'_{2w-2})\\ \text{Transpose[w/2]}(v'_1,v'_3,...,v'_{w-1}),\ \text{Transpose[w/2]}(v'_{w+1},v'_{w+3},...,v'_{2w-1})\end{bmatrix}$
	v,u	Merge_Reorder[w](v,u)	v'_0,v'_1,u'_0,u'_1	Merge_Reorder[w/2](v'_0,u'_0); Merge_Reorder[w/2](v'_1,u'_1)
64-bit, 32-bit	$v_0,v_1,...,v_{w-1}$	Transpose[w]$(v_0,v_1,...,v_{w-1})$	$v'_0,v'_1,...,v'_{w-1}$†	Transpose[w]$(v'_0,v'_1,...,v'_{w-1})$
	v,u	Merge_Reorder[w](v,u)	v',u'†	Merge_Reorder[w](v',u')

†: On MIC, only the first halves of each vector are effective; On CPU, SSE vectors are adopted.

TABLE 3: Testbeds for ASPaS

Model	Intel Xeon CPU E5-2697 v2	Intel Xeon CPU E5-2680 v3	Intel Xeon Phi 5110P
Codename	Ivy Bridge	Haswell	Knights Corner
Frequency	2.70GHz	2.50GHz	1.05GHz
Cores	24	24	60
Threads/Core	2	1	4
Sockets	2	2	-
L1/L2/L3	32kb/256kb/30mb	32kb/256kb/30mb	32kb/512kb/-
Vector ISA	AVX	AVX2	IMCI
Memory	64GB	128GB	8GB
Mem Type	DDR3	DDR3	GDDR5

BI sort follows a balanced way to compare all elements in each step and is usually considered as a candidate for better performance, it uses more comparators, leading to the relatively weak performance for the *sort* stage. Base on the results, in the remaining experiments, we choose the BN, OE, and GR sorts for the Integer datatype on IVB, HSW, and KNC, respectively. For the Double datatype, we also choose the best one, i.e., OE sort, for the rest of the experiments.



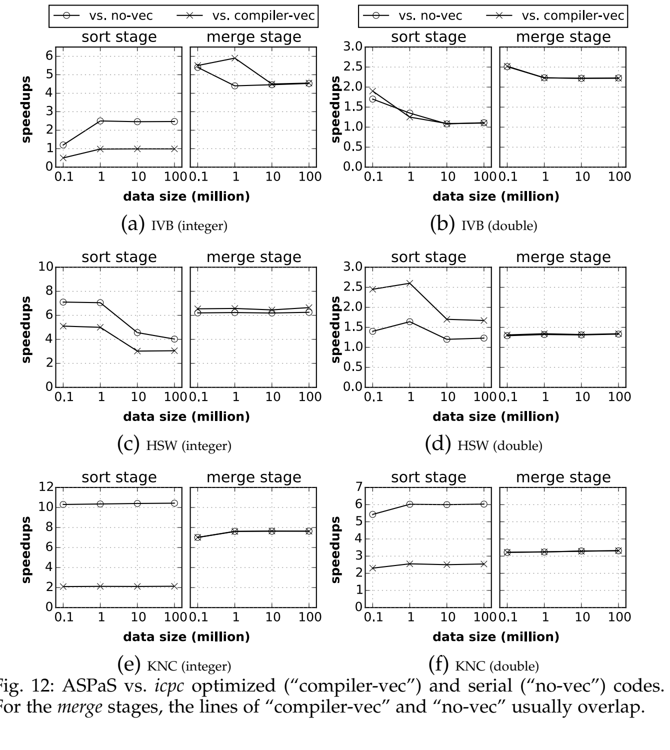
(a) IVB (b) HSW (c) KNC

Fig. 11: Performance comparison of `aspas_sort` and `aspas_merge` with different sorting and merging networks. The kernels are repeated by 10 million times over a built-in vector-length array and total times are reported. The numbers of comparators and groups are given in parenthesis for sorting networks.

In the *merge* stage, we set two variants of bitonic merging networks (Eq. 2 and Eq. 3 in Sec. 3.3) as the input of ASPaS. Fig. 11 also presents the performance comparisons for these two variants. The inconsistent merging can outperform the consistent one by 12.3%, 20.5%, and 43.3% on IVB, HSW, and KNC, respectively. Although the consistent merging has uniform data-reordering operations in each step as shown in Fig. 7, the operations are not ISA-friendly and thus requires a longer sequence of intrinsics. For example, based on Eq. 3, the consistent merging uses 5 times of the L^{32}_2 data reordering operations on MIC, each of which needs 8 permute/shuffle IMCI intrinsics. In contrast, the inconsistent merging only uses L^{32}_2 once and compensate it with much lighter operations (e.g., $I_1 \otimes L^{32}_{16} \circ I_2 \otimes L^{16}_2$ and $I_2 \otimes L^{16}_8 \circ I_4 \otimes L^8_2$, each of which can be implemented by an average of 2 IMCI intrinsics). On CPUs, the L^{16}_2 operation in the consistent variant only needs 4 AVX intrinsics, leading to the smaller disparity. But, in all cases, the inconsistent bitonic merge provides the best performance. The Double datatype exhibits similar behaviors. Thus we will adopt the inconsistent merging in the remaining experiments.

5.2 Speedups from the ASPaS Framework

In this section, we compare the ASPaS *sort* and *merge* stages with their serial counterparts. The counterparts of `aspas_sort` and `aspas_merge` are serial sorting and merging networks (one comparison and exchange at a time) respectively. Note, in the *sort* stage, the `aspas_transpose` is not required in the serial version, since the partially sorted data can be stored directly in a consecutive manner. Ideally, the speedups from the ASPaS should approximate the built-in vector width; though this is impractical because of the extra and required data reordering instructions. By default, the compiler will auto-vectorize the serial codes[4], which is denoted as "compiler-vec". Besides, we explicitly turn off the auto-vectorization, which is shown as "no-vec".



(a) IVB (integer) (b) IVB (double)

(c) HSW (integer) (d) HSW (double)

(e) KNC (integer) (f) KNC (double)

Fig. 12: ASPaS vs. *icpc* optimized ("compiler-vec") and serial ("no-vec") codes. For the *merge* stages, the lines of "compiler-vec" and "no-vec" usually overlap.

For the *sort* stages with Integer datatype on CPUs in Fig. 12 (a,c), the ASPaS codes can deliver more performance improvements on HSW over IVB, since the AVX on IVB does not support native integer operations as in AVX2. Thus, we have to split the AVX vector to two SSE vectors before resorting to the SSE ISA for comparisons. For the *sort*

4. We also use the SIMD pragma *pragma vector always* on the target loops

stages with Double in Fig. 12 (b,d), the ASPaS codes exhibit similar performance gains over "no-vec", achieving slight 1.1~1.2x speedups. The vectorization benefits of Double drop down because less elements in each vector than Integer, leading to relatively higher data reordering overhead. On KNC, ASPaS Integer and Double *sort* codes in Fig. 12 (e,f) outperform the "no-vec" counterparts up to 10.5x and 6.0x. In addition, the ASPaS codes can also achieve better performance than the "compiler-vec" versions in most cases. By analyzing the generated assembly codes in "compiler-vec", we find: on IVB, the compiler uses multiple `insert` instructions to construct vectors slot by slot from non-contiguous memory locations; instead, the `gather` instructions are used on HSW and KNC. However, neither can mitigate the high latency of non-contiguous memory access. The ASPaS codes, in contrast, can outperform the "compiler-vec" by using the `load/store` on the contiguous data and the `shuffle/permute` for the transpose in registers. We also observe that in Fig. 12 (d) the "compiler-vec" of *sort* stage slowdowns the execution compared to the "no-vec". This may stem from the fact that the HSW supports vector `gather` but no equivalent vector `scatter` operations. The asymmetric load-and-store fashion on non-contiguous data with larger memory footprint (Double) causes negative impacts on the performance [1].

The *merge* stages in Fig. 12 on the three platforms show that the "compiler-vec" versions have the similar performance with the "no-vec". This demonstrates that even with the most aggressive vectorization pragma, the compiler fails to vectorize the merge codes due to the complex data dependency within the loops.

5.3 Comparison to Previous SIMD Kernels

In this section, we compare our generated kernels with those manually optimized kernels proposed in previous research. These existing vector codes also focus on using vector instructions and sorting networks to sort small arrays with sizes of multiple of SIMD-vector's length. The reasons for comparing kernels with smaller data sizes rather than any large data size are following: (1) the kernels for sorting small arrays are usually adopted to construct efficient parallel sort algorithms in a divide-and-conquer manner (e.g., quick-sort [23], [24], merge-sort [20], [25]), where input data is split into small chunks each of which fits into registers, the sort kernel is applied on each chunk, and the merge kernel is called iteratively to merge chunks until there is only one chunk left. Under this circumstance, the overall performance significantly depends on the vectorization kernels [23]; (2) Our major motivation of this paper is to efficiently generate combinations of permutation instructions instead of proposing a new divide-and-conquer strategy for any large data size. As a result, we compare vector codes from Chhugani et al. (CH) [20] and Inoue et al. (IN) [19] on CPUs; while on MICs, we compare vector codes from Xiaochen et al. (XI) [26] and Bramas (BR) [24]. The datatype in this experiment is the 32-bit integer[5]. We use one core (vector unit) to process randomly-generated data in the segment of 8x8=64 integers for CPUs and of 16x16=256 integers for MICs, respectively. The experiments are repeated for 1 million times and we report the total execution time.

5. The BR paper [24] only provides AVX-512 codes for Knights Landing (KNL). Therefore, we have to port the codes using corresponding IMCI instructions on KNC, e.g., replacing `permutexvar_pd` with `permutevar_epi32` and correct parameters.

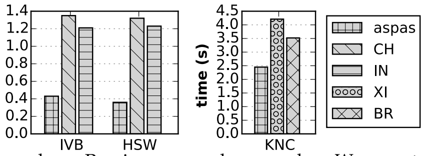


Fig. 13: ASPaS kernels vs. Previous manual approaches. We repeatedly (1 million times) sort 8x8=64 integers for CPUs and 16x16=256 integers for MICs, respectively. The time on data load from memory to registers and store from registers to memory are included with the sort and merge in registers.

Fig. 13 shows the performance comparison. On CPUs, both CH and IN methods use SSE instructions to handle intra-lane data-reordering, leading to extra instructions used to process inter-lane communications. Compared to our generated codes using AVX/AVX2 instructions, these solutions are relatively easier to implement, because they only need to process vector lanes one by one and there are always one unused lane for every operation, thus delivering sub-optimal performance. To use the AVX/AVX2 instructions, one has to redesign their method and consider the different register length and corresponding instructions. In contrast, our solution automatically looks for the architecture-specific instructions to handle both intra- and inter-lane communications and deliver up to 3.4x speedups over these manual approaches. On MICs, the XI method adopts *mask* instructions to disable some elements for each min/max operation. These unused slots inevitably under-utilize the vector resources. The BR method, on the other hand, directly uses the expensive `permutexvar` instructions to conduct the global data-reordering. As a contrast, our code generation framework can satisfy the underlying architectures, e.g., preferring lightweight intra-lane and swizzle instructions when making the code generation. Therefore, on the KNC platform, our codes can provide up to 1.7x performance improvements over the manually optimized methods.

5.4 Comparison to Sorting from Libraries

In the section, we will evaluate the single-threaded `aspas::sort` and multi-threaded `aspas::parallel_sort` by comparing them with their related mergesorts and various sorting tools from existing libraries.

Single-threaded ASPaS: ASPaS is essentially based on the bottom-up mergesort as the partition strategy. We first compare the single-threaded `aspas::sort` with two mergesort variants: top-down and bottom-up. The top-down mergesort recursively splits the input array until the split segments only have one element. Subsequently, the segments are merged together. As a contrast, the bottom-up mergesort, which directly works on the elements in the input array and iteratively merge them into sorted segments. For their implementation, we use the `std::inplace_merge` as the kernel to conduct the actual merging operations. Fig. 14 (a,b,c) illustrate the corresponding performance comparison on IVB, HSW, and KNC. The bottom-up mergesort can outperform the top-down slightly due to the recursion overhead in the top-down method.

The ASPaS of Integer datatype outperforms the bottom-up mergesorts by 4.3x to 5.6x, while the Double datatype provides 3.1x to 3.8x speedups.

ASPaS can efficiently vectorize the *merge* stage, even though the complexity of ASPaS merging is higher than the `std::inplace_merge` used in the bottom-up mergesort. In ASPaS, when merging each pair of two sorted segments, we fetch w elements into a buffer from each segment and then merge these $2w$ elements using the $2w$-way bitonic merging. After that, we store the first half of merged $2w$ elements back to the result, and load w elements from the segment with the smaller first element into the buffer; and then, the next round of bitonic merge will occur (ln. 18-28 in Alg. 1). Since the $2w$-way bitonic merging network contains $2log(2w)2^{log(2w)-2}$ comparators [9], for every w elements, the total number of comparisons is $(N/w) * 2log(2w)2^{log(2w)-2} = log(2w)N$. As a contrast, the `std::inplace_merge` conducts exactly N-1 comparisons if enough additional memory is available. Therefore, the comparisons in the bottom-up mergesort are considerably less than what we use in Alg. 1. However, because our code carries out better memory access pattern: fetching multiple contiguous data from the memory and then conducting the comparisons in registers with a cache-friendly manner, we observe better performance of `aspas::sort` over any of the bottom-up mergesort on all three platforms in Fig. 14 (a,b,c).

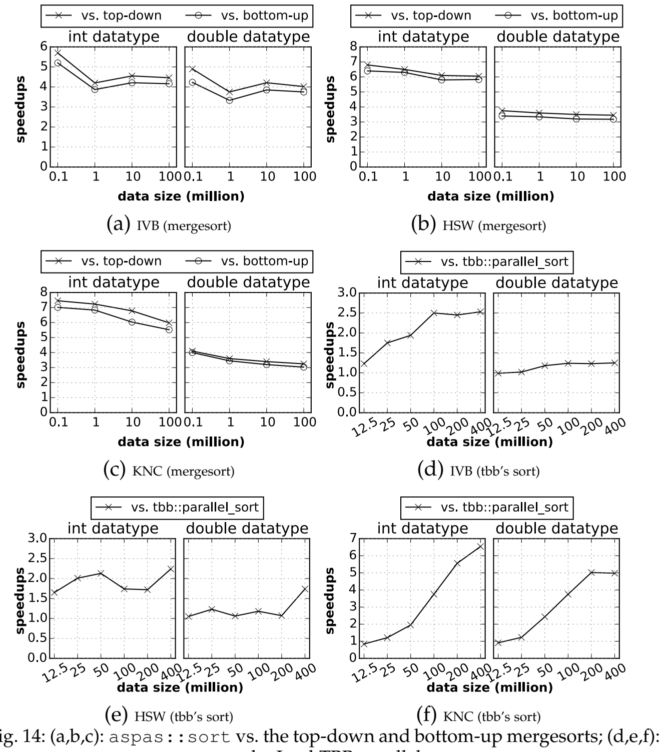


Fig. 14: (a,b,c): `aspas::sort` vs. the top-down and bottom-up mergesorts; (d,e,f): `aspas::parallel_sort` vs. the Intel TBB parallel sort.

Then, we compare the `aspas::sort` with other existing sorting tools from widely-used libraries, including the `qsort` and `sort` from STL (libstdc++.so.6.0.19), `sort` from Boost (v.1.55), and `parallel_sort` from Intel TBB (v.4.1) (using a single thread). Fig. 15 presents that the ASPaS codes can provide the highest performance over the other four sorts. The `aspas::sort` on the Integer array can achieve 4.2x, 5.2x, and 5.1x speedups over the `qsort` on IVB, HSW, and KNC, respectively (`qsort` is also notorious about its function callback for every comparison.). Over the other sorting tools, it can still provide up to 2.1x, 3.0x, and 2.5x speedups. For Double datatype, the performance benefits of `aspas::sort` become 3.8x, 2.9x, and 3.1x speedups over the `qsort`, and 1.8x, 1.7x, 1.3x speedups over others on the three platforms correspondingly.

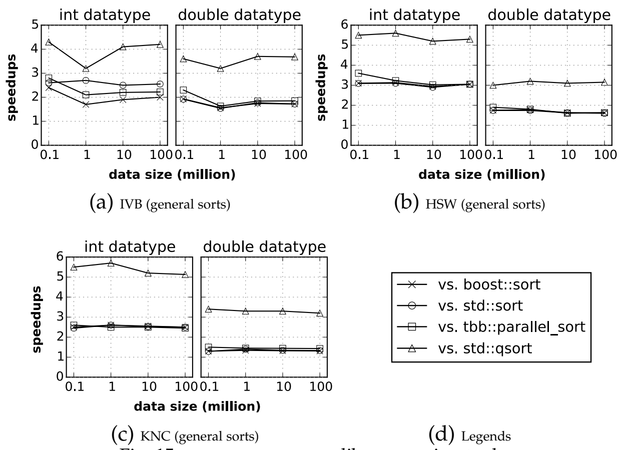


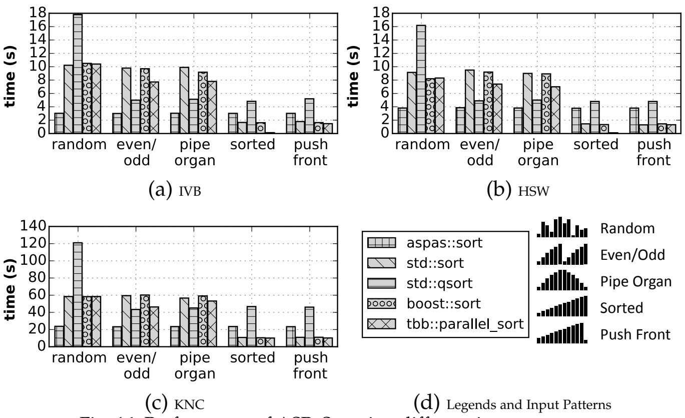
Fig. 15: `aspas::sort` vs. library sorting tools.

Multi-threaded ASPaS: In Fig. 14 (d,e,f), we compare the multi-threaded ASPaS to the Intel TBB's `parallel_sort` for a larger dataset from 12.5 to 400 million Integer and Double elements. We configure the thread numbers to the integral multiples of cores and select the one that can provide the best performance. On the three platforms, our `aspas::parallel_sort` can outperform the `tbb::parallel_sort` by up to 2.5x, 2.3x, and 6.7x speedups for the Integer datatype and 1.2x, 1.7x, and 5.0x speedups for the Double datatype.

5.5 Sorting Different Input Patterns

Finally, we evaluate the `aspas::sort` using different input patterns. As shown in Fig. 16 (d), we use five input patterns defined in the previous research [23], including random, even/odd, pipe organ, sorted, and push front input. With these input patterns, we can further evaluate the performance of our generated vector codes with existing methods from widely used libraries.

In Fig. 16 (d), we can find that the sorting tools from modern libraries can provide better performance than our generated codes for the almost sorted inputs, i.e., "sorted" and "push front". That is because these libraries can be adaptive to different patterns by using multiple sorting algorithms. For example, `std::sort` uses a combination of quick sort and insertion sort. For an almost sorted input array, `std::sort` switches from the partition of the quick sort to the insertion sort, which is good at handling the sorted input within O(n). As a contrast, our work focuses on automatically generating efficient sorting kernels for more general cases, e.g., random, even/odd, and pipe organ. At these cases, our sorting codes can yield superior performance.



(a) IVB (b) HSW (c) KNC (d) Legends and Input Patterns

Fig. 16: Performance of ASPaS sorting different input patterns.

6 Related Work

Sorting is a widely-used algorithm in a plethora of applications. Many research efforts have been made to modify and optimize sorting algorithms on such modern hardware architectures. The early research done by Levin [27] has adapted sorting algorithms on vector computers by fully vectorizing both partition and base cases of quick sort on Cray and Convex. The sss-sort [28] and the sort by Codish et al. [23] focus on eliminating conditional branches in the generalized quick sort and facilitate the instruction-level parallelism. The cpp-sort [29] provides a set of fixed-size sorters for users to synthesize sorting algorithms. The sorters are based on sorting network but not explicitly using vector instructions.

Satish et al. [25], [30] compare and analyze the radix sort and merge sort on modern accelerators, including CPUs and GPUs, and point out that the merge sort is superior, since it can benefit more from the efficient SIMD operations in modern accelerators. Furtak et al. [31] use SIMD optimizations to solve the base cases of recursive sorting algorithms. The AA-sort [32] is a two-phase sorting algorithm with vectorized combsort and odd-even merge on CPUs. Chhugani et al. [20] devise another SIMD-friendly mergesort algorithm by using the odd-even and bitonic sorting networks. Their solution provides an architecture-specific and hard-coded solution of SSE and Larrabee ISAs but not revealing many details on how the parameters are selected and how to deal with data across vector lanes (e.g., on Larrabee). Inoue et al. [19] propose a stable sorting SIMD algorithm to rearrange the actual database records. On MICs, Bramas [24] proposes an efficient partition solution for quicksort by using "store some" instructions in AVX-512. Xiaochen et al. [26] studies the bitonic merge sort using both mask and permute IMCI instructions. These existing studies have to explicitly use SIMD intrinsics to handle the tricky data-reordering operations required by different sorting and merging algorithms; while our work, in contrast, formalizes the patterns of sorting networks and vector ISAs to facilitate the automatic code generation of efficient and "cross-platform" vector codes.

There are also existing SIMD-friendly programming techniques from compilers, e.g. ISPC [33] and Clang 6.0 [34], where the vector operations are usually simplified to the array operations (treating vectors as arrays). However, because not revealing the details of the actual vector instructions being used, the performance of these frameworks is not well-understood. Furthermore, programmers still need to explicitly point out the correct permute parameters when using these tools.

For certain specific applications, whose computation and communication patterns can be formalized, frameworks are proposed to automatically generate parallel codes. With emphasis on using the intra-core resources, Ren et al. [35] present a SIMD optimization engine for irregular data-traversal applications. AAlign [36] presents an automation method to transform sequential alignment algorithms to vector operations. Mint [5], Physis [6], Zhang and Mueller [37] can generate effective GPU codes for stencil applications. McFarlin et al. [4] demonstrate another superoptimizer to conduct a guided search of the shortest sequence of SIMD instructions over a large candidate pool. Ren et al. [38] provide an approach to optimize the SIMD code generation for generic permutations. In the code searching part, they select correct elements to approximate the target vectors, and try to minimize the number of data movements as what we are doing in this work. However, this method is based on an assumption that any data movements can be directly translated by the SSE's `shufps` instruction, which is reasonable since SSE only contains one lane and the translation part could be rather straightforward. In contrast, the modern SIMD units of AVX/AVX2/AVX512 contain more lanes and more reordering instructions, thereby increasing the complexity of efficient code selection. The direct mapping method that was fine for SSE instructions becomes inefficient. In this work, we select a small number of efficient data-reordering patterns, e.g., the symmetric patterns formalized in Sec. 4.1, and use a searching algorithm to explore the best combinations to achieve optimal performance, e.g., preferring lightweight intra-lane and swizzle instructions, for different platforms.

7 Conclusion

In this paper, we propose the ASPaS framework to automatically generate vectorized sorting code for x86-based multicore and manycore processors. ASPaS can formalize the sorting and merging networks to the sequences of comparing and reordering operators of DSL. Based on the characteristics of such operators, ASPaS first creates an ISA-friendly pool to contain the requisite data comparing and reordering primitives, then builds those sequences with primitives, and finally maps them to the real ISA intrinsics. Besides, the ASPaS codes can exhibit a efficient memory access pattern and thread-level parallelism. The ASPaS-generated codes can outperform the compiler-optimized ones and meanwhile yield highest performance over multiple library sorting tools on Ivy Bridge, Haswell, and Knights Corner architectures.

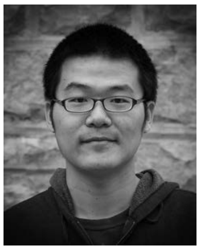
With the emerge of Skylake and Knights Landing architecture, our work can be easily ported to AVX-512, since the ISA subset AVX-512F contains all the permute/shuffle instructions we need for sorting. For GPUs, we will also extend ASPaS to search shuffle instructions to support fast data permutation at register level.

ACKNOWLEDGMENTS

This research was supported in part by NSF-BIGDATA program via IIS-1247693, the NSF-XPS program via CCF-1337131. We acknowledge Advanced Research Computing at Virginia Tech for the computational resources.

REFERENCES

[1] S. Maleki, Y. Gao, M. Garzaran, T. Wong, and D. Padua, "An Evaluation of Vectorizing Compilers," in *ACM Int. Conf. Parallel Arch. Compil. Tech. (PACT)*, 2011.
[2] Intel Xeon Phi Coprocessor System Software Developers Guide. [Online]. Available: https://software.intel.com/sites/default/files/article/334766/intel-xeon-phi-systemsoftwaredevelopersguide.pdf
[3] X. Huo, B. Ren, and G. Agrawal, "A Programming System for Xeon Phis with Runtime SIMD Parallelization," in *ACM Int. Conf. Supercomput. (ICS)*, 2014.
[4] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, "Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets," in *ACM Int. Conf. Supercomput. (ICS)*, 2011.
[5] D. Unat, X. Cai, and S. B. Baden, "Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C," in *ACM Int. Conf. Supercomput. (ICS)*, 2011.
[6] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers," in *Int. Conf. High Perf. Comput., Netw., Storage and Anal. (SC)*, 2011.
[7] A. R. Benson and G. Ballard, "A Framework for Practical Parallel Fast Matrix Multiplication," in *ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, 2015.
[8] S. W. Al-Haj Baddar and K. W. Batcher, *Designing Sorting Networks: A New Paradigm*. Springer Sci. & Bus. Media, 2011.
[9] K. E. Batcher, "Sorting Networks and Their Applications," in *ACM Spring Joint Computer Conf.*, 1968.
[10] T. N. Hibbard, "An Empirical Study of Minimal Storage Sorting," *Commun. ACM*, 1963.
[11] R. C. Bose and R. J. Nelson, "A Sorting Problem," *J. ACM*, 1962.
[12] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov, *C++ Standard Template Lib.*, 1st ed. Prentice Hall PTR, 2000.
[13] B. Schling, *The Boost C++ Libraries*. XML Press, 2011.
[14] J. Reinders, *Intel Threading Building Blocks*, 1st ed. O'Reilly & Associates, Inc., 2007.
[15] K. Hou, H. Wang, and W.-c. Feng, "ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors," in *ACM Int. Conf. Supercomput. (ICS)*, 2015.
[16] R. Rahman, *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*, 1st ed. Apress, 2013.
[17] F. Franchetti, F. Mesmay, D. Mcfarlin, and M. Püschel, "Operator Language: A Program Generation Framework for Fast Kernels," in *IFIP TC 2 Working Conf. on Domain-Specific Lang. (DSL)*, 2009.
[18] M. Zuluaga, P. Milder, and M. Püschel, "Computer Generation of Streaming Sorting Networks," in *ACM Design Autom. Conf. (DAC)*, 2012.
[19] H. Inoue and K. Taura, "SIMD- and Cache-friendly Algorithm for Sorting an Array of Structures," *Proc. VLDB Endow. (PVLDB)*, 2015.
[20] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture," *Proc. VLDB Endow. (PVLDB)*, 2008.
[21] O. Green, R. McColl, and D. A. Bader, "GPU Merge Path: a GPU Merging Algorithm," in *ACM Int. Conf. Supercomput. (ICS)*, 2012.
[22] M. W. Green, "Some Improvements in Non-adaptive Sorting Algorithms," in *Annu. Princeton Conf. Inf. Sci. Syst.*, 1972.
[23] M. Codish, L. Cruz-Filipe, M. Nebel, and P. Schneider-Kamp, "Applying Sorting Networks to Synthesize Optimized Sorting Libraries," in *Int. Symp. Logic-Based Program Synth. Transform. (LOPSTR)*, 2015.
[24] B. Bramas, "Fast Sorting Algorithms using AVX-512 on Intel Knights Landing," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1704.08579
[25] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort," in *ACM SIGMOD Int. Conf. Manage. of Data*, 2010.
[26] T. Xiaochen, K. Rocki, and R. Suda, "Register Level Sort Algorithm on Multi-core SIMD Processors," in *ACM Workshop Irregular App.: Arch. Alg. (IA3)*, 2013.
[27] S. A. Levin, "A Fully Vectorized Quicksort," *Parallel Comput.*, 1990.
[28] P. Sanders and S. Winkel, "Super Scalar Sample Sort," in *Annu. Euro. Symp. Alg. (ESA)*. Springer, 2004.
[29] Morwenn. (2016) cpp-sort: A Generic C++14 Header-only Sorting Library. [Online]. Available: https://github.com/Morwenn/cpp-sort
[30] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," in *IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, 2009.
[31] T. Furtak, J. N. Amaral, and R. Niewiadomski, "Using SIMD Registers and Instructions to Enable Instruction-level Parallelism in Sorting Algorithms," in *ACM Symp. Parallel Alg. Arch. (SPAA)*, 2007.
[32] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors," in *ACM Int. Conf. Parallel Arch. Compil. Tech. (PACT)*, 2007.
[33] M. Pharr and W. Mark, "ispc: A SPMD Compiler for High-Performance CPU Programming," in *Innovative Parallel Comput. (InPar)*, 2012.
[34] Clang 6 Doc.: Clang Language Extensions. [Online]. Available: https://clang.llvm.org/docs/LanguageExtensions.html
[35] B. Ren, T. Mytkowicz, and G. Agrawal, "A Portable Optimization Engine for Accelerating Irregular Data-Traversal Applications on SIMD Architectures," *ACM Trans. Archit. Code Optim. (TACO)*, 2014.
[36] K. Hou, H. Wang, and W.-c. Feng, "AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-based Multi- and Many-core Processors," in *IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, 2016.
[37] Y. Zhang and F. Mueller, "Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters," *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 2013.
[38] G. Ren, P. Wu, and D. Padua, "Optimizing Data Permutations for SIMD Devices," in *ACM SIGPLAN Conf. Program. Lang. Design Impl. (PLDI)*, 2006.



Kaixi Hou is a Ph.D. candidate in the Dept. of Computer Science at Virginia Tech, USA, where he is a member of the Synergy Lab. He received his M.S. of Computer Science at the Beijing University of Chemical Tech. (BUCT), China. His primary research area is in high-performance computing, with a focus on accelerating applications of bioinformatics and graph processing by using data-level parallelism on multi/manycore processors.

Hao Wang is a Senior Research Associate in the Dept. of Computer Science at Virginia Tech. He received his Ph.D. in the Institute of Computing Technology at Chinese Academy of Sciences. His research interests include high performance, parallel, and distributed computing. His current focus is on designing algorithms of graph processing, bioinformatics, and computational fluid dynamics on HPC clusters with InfiniBand interconnects, GPUs and Intel MIC co-processors.

Wu-chun Feng is a professor and Elizabeth & James E. Turner Fellow in the Dept. of Comp. Science, Dept. of Electrical and Computer Engineering, Health Sciences, and Virginia Bioinformatics Institute at Virginia Tech. His interests lie broadly at the synergistic intersection of computer architecture, systems software and applications software. Most recently, his research has dealt with accelerator-based computing for bioinformatics and power-aware computing.