ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors

Kaixi Hou, Hao Wang, and Wu-chun Feng
Department of Computer Science
Virginia Tech
Blacksburg, VA, 24060
{kaixihou,hwang121,wfeng}@vt.edu

ABSTRACT

Due to the difficulty that modern compilers have in vectorizing applications on vector-extension architectures, programmers resort to manually programming vector registers with intrinsics in order to achieve better performance. However, the continued growth in the width of registers and the evolving library of intrinsics make such manual optimizations tedious and error-prone. Hence, we propose a framework for the *A*utomatic *S*IMDization of *Pa*rallel *S*orting (*ASPaS*) on x86-based multicore and manycore processors. That is, ASPaS takes any sorting network and a given instruction set architecture (ISA) as inputs and automatically generates vectorized code for that sorting network.

By 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 vectorized code with the real ISA intrinsics. The performance evaluation of our ASPaS framework on the Intel Xeon Phi coprocessor illustrates that automatically generated sorting codes from ASPaS can outperform the sorting implementations from STL, Boost, and Intel TBB.

Categories and Subject Descriptors

D.3.4 [**Programming Language**]: Processors—*Code generation, Optimization*

General Terms

Performance

Keywords

sort; merge; transpose; vectorization; SIMD; ISA; MIC; AVX; AVX-512;

1. INTRODUCTION

To boost performance, modern processors put multiple cores onto a single die rather than increase processor frequency. In addition to this inter-core parallelism, a vector processing unit (VPU) is associated with each core to enable further intra-core parallelism. Execution on a VPU follows a "single instruction, multiple data" (SIMD) paradigm by carrying out the "lock-step" operations over packed data. Although modern compilers can automatically vectorize most regular codes, they often fail to vectorize complex loop patterns due to the lack of accurate compiler analysis and effective compiler transformations [15]. Therefore, the burden falls on programmers' shoulders to implement the vectorization using intrinsics or even assembly code.

Writing efficient SIMD code by hand is a time-consuming and error-prone activity. First, the vectorization of existing (complex) codes requires expert knowledge of the underlying algorithm. Second, the vectorization requires a comprehensive understanding of the SIMD intrinsics. The intrinsics for data management and movement are as important as those for computation because programmers often need to rearrange data in the vector units before sending them to the ALU. However, the flexibility of the data-reordering functions is restricted, since directly supporting an arbitrary permutation is impractical [14]. Consequently, programmers have to resort to a combination of data-reordering functions to attain a desired pattern. Third, the architecture-specific nature of vector instructions, i.e., different processors might support different vector widths and versions of an instruction set architecture (ISA), can cause portability issues. For example, to port Advanced Vector Extensions (AVX) codes to the AVX-512 architecture of the Intel Many Integrated Core (MIC), we either need to identify the instructions with equivalent functionalities or rewrite and tweak the codes using alternative instructions. While library-based optimizations [12] can hide the details of vectorization from the end user, these challenges are still encountered during the design and implementation of the libraries.

To address the above issues with respect to the sorting primitive, we propose a framework called ASPaS, short for Automatic SIMDization of Parallel Sorting, to automatically generate efficient SIMD codes for parallel sorting on x86-based multicore and manycore processors. ASPaS can take any sorting network and a given ISA as inputs and automatically produce vectorized sorting code as the output. The generated code adopts a bottom-up scheme to sort and merge segmented data. Because the vectorized sort function puts partially sorted data in column-major order, AS-

PaS compensates it with the transpose function before the merge stage. Considering the variety of sorting and merging networks[1] [1] that correspond to different sorting algorithms (such as odd-even [3], bitonic [3], Hibbard [11], and Bose-Nelson [4]) and the continuing evolution of the instruction set [7] (such as SSE, AVX, AVX2, and AVX-512), 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 parts: (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 maps them to the real ISA intrinsics.

The major contributions of our work include the following. First, we propose the ASPaS framework to automatically generate parallel sorting code using architecture-specific SIMD instructions. Second, using ASPaS, we generate various parallel sorting codes for the combinations of five sorting networks, two merging networks, and three datatypes (integer, float, double) on Intel MIC, and then conduct a series of evaluations. For the one-word type,[2] our SIMD codes can deliver up to 7.7-fold and 5.7-fold speedups over the serial sort and merge, respectively. For the two-word type, the corresponding speedups are 6.3-fold and 3.7-fold, respectively. Compared with other single-threaded sort implementations, including `qsort` and `sort` from STL [19], and `sort` from Boost [24], our SIMD codes deliver a range of speedups from 2.4-fold to 4.3-fold for the one-word type and 1.3-fold to 2.6-fold for the two-word type. We also wrap up our SIMD codes into a multi-threaded version. Compared with `parallel_sort` from Intel TBB [21], ASPaS delivers speedups of up to 2.1-fold and 1.4-fold for the one-word type and the two-word type, respectively.

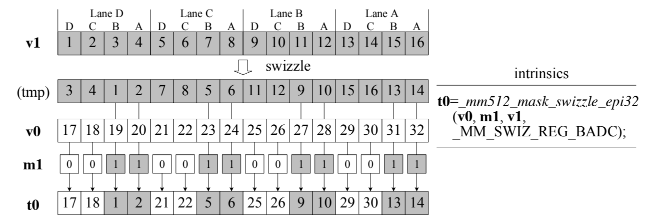
2. BACKGROUND

This section presents (1) a brief overview of the vector architecture of Intel MIC, (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 MIC Vector Architecture

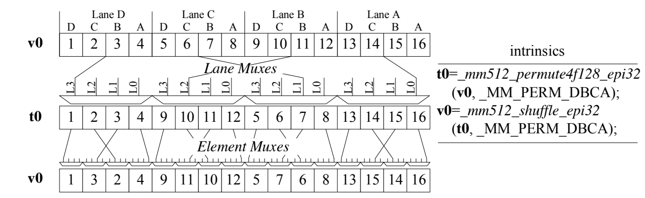
The MIC coprocessor consists of up to 61 in-order cores, each of which is outfitted with a new vector processing unit (VPU). The VPU state for each thread contains 32 512-bit general registers (zmm0-31), eight 16-bit mask registers (k0-7), and a status register. A 512-bit SIMD ISA is introduced in accordance with the new VPU. However, 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. [20].

In 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. 1a shows an example to rearrange data from two vector registers with the masked swizzle instruction. The intrinsics indicate 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 `t0`. 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.



t0=_mm512_mask_swizzle_epi32
(v0, m1, v1,
_MM_SWIZ_REG_BADC);

(a) masked swizzle



t0=_mm512_permute4f128_epi32
(v0, _MM_PERM_DBCA);
v0=_mm512_shuffle_epi32
(t0, _MM_PERM_DBCA);

(b) permute and shuffle

Figure 1: Rearrange data on Intel MIC from: (a) two vector registers with the masked swizzle operation; (b) the same vector register with the permute and shuffle operations

Fig. 1b illustrates an example to rearrange data in the same vector register with the shuffle and permute intrinsics. The permute intrinsic using `_MM_PERM_DBCA` is for inter-lane rearrangement, which exchanges data in lanes `B` and `C`. The shuffle intrinsic with the same parameter is for intra-lane rearrangement to exchange elements at positions `B` and `C` of the same lane. This figure also shows the micro-architecture details [14]. The *Lane Muxes* are used to select desired lanes, and the *Element Muxes* are used to select desired elements in 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. In the design of ASPaS, we use this characteristic to obtain instruction-level overlapping.

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 [8, 27] 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.

[1]In this paper, we distinguish the sorting network and the merging network.

[2]We use the Integer datatype as the representative of the one-word type, and the Double datatype for the two-word type.

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_m^{km} $x_{ik+j} \mapsto x_{jm+i}, 0 \leqslant i < m, 0 \leqslant j < k$. L_m^{km} 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.

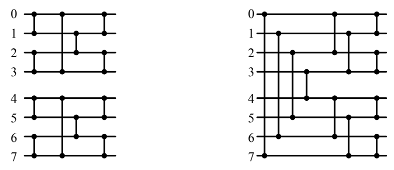
($\oplus$) The direct sum operator is served to merge two operators. $A_n \oplus B_m$ indicates that the first n elements of the input vector is processed by A_n, while the rest m elements follow B_m.

($\otimes$) 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 [1] 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

Figure 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. METHODOLOGY

Our parallel sorting adopts a bottom-up approach to sort and merge segmented data, as illustrated in Alg. 1. This algorithm first divides the input data into contiguous segments, each of which size equals to a multiple times of SIMD width. Second, it loads each segment into registers and do the in-register vectorized sorting by calling the functions of `aspas_sort` and `aspas_transpose` (the sort stage in line 3-7). Then, the algorithm will merge successive sorted segments iteratively to generate the final output by calling `aspas_merge` (the merge stage of line 9-13). The functions of `load`, `store`, `aspas_sort`, `aspas_transpose`, and `aspas_merge` will be generated by ASPaS using the ISA-specific intrinsics. Because the `load` and `store` can be translated to the intrinsics once the ISA is given, we focus on other three 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 illustrates the structure of the ASPaS framework and the generated `sort` function. Three modules of ASPaS —*SIMD Sorter*, *SIMD Transposer*, and *SIMD Merger* — are responsible for generating the sequences of comparing and data-reordering operations for the corresponding functions. These sequences will be mapped to the real SIMD intrinsics by the module of *SIMD Code Generator*.

3.1 SIMD Sorter

The operations of `aspas_sort` are generated by the *SIMD Sorter*. As shown in Fig. 4, `aspas_sort` loads the n-by-n matrix of data into n vectors and processes them based on the given sorting network, leading to the data sorted along each column of the matrix. Fig. 5 presents an example of a 4-by-4 data matrix going through a 4-key sorting network. Here, each dot represents one vector and each vertical line represents the vectorized comparison on the corresponding vectors. After six comparisons, the original data

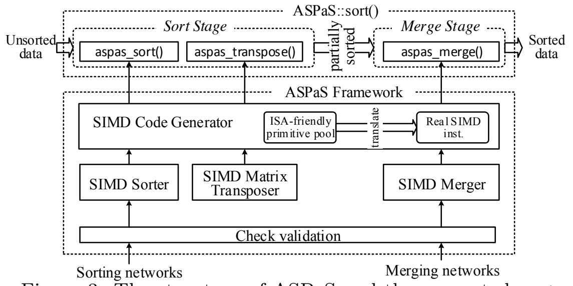


Figure 3: The structure of ASPaS and the generated `sort`

is sorted in ascending order in each column. Fig. 5 also shows the data dependency among these comparators. For example, CMP(0,1) and CMP(2,4) can be issued simultaneously, while CMP(0,3) can occur only after these two. It is straightforward to achieve three groups of comparators for this sorting network. However, for some sorting networks, we need a careful analysis of the data dependency when grouping the comparators. In the *SIMD Sorter*, we design an optimized grouping mechanism to analyze the input sequence of comparators and organize them into multiple groups. Our mechanism can facilitate the code generation by minimizing the number of groups, and in turn, reduce the code size.

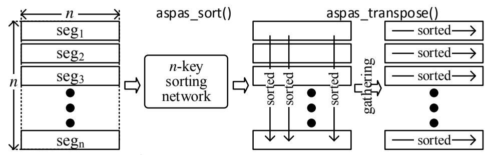


Figure 4: Mechanism of the sort stage: operations generated by *SIMD Sorter* and *SIMD Transposer*

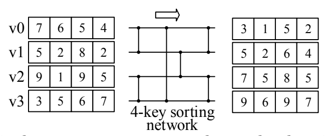


Figure 5: Four 4-element vectors go through the 4-key sorting network. Afterwards data is sorted in each column of the matrix.

Fig. 6 shows the Knuth diagram based on the Bose-Nelson sorting network [4]. The results of the original grouping mechanism and the optimized grouping mechanism are shown in this figure. The original grouping mechanism puts one comparator into the current group if this comparator has no data dependency with all the other operators in the current group; otherwise, a new group is created for this operator. As a result, the comparators are organized as seven groups by this scheme. However, if no data dependency exists among adjacent comparators, changing their order does not matter. Our optimized grouping exploits this feature and optimizes the original grouping. That is, it not only checks the data dependency between the comparators within the current group, but it also checks previous groups in a traceback manner until the data dependency is found or all groups are checked. Then, the comparator is put into the last-found group without any data dependency. By using this scheme, we can reorder S_2(3,7) and S_2(2,6) and place them into the appropriate groups. As a result, the number of groups with the optimized grouping is decreased from seven to six.

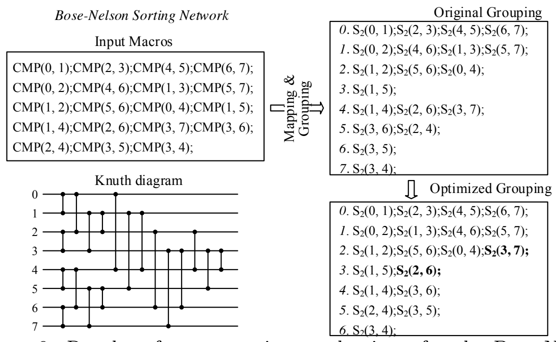

Bose-Nelson Sorting Network
Input Macros
CMP(0, 1);CMP(2, 3);CMP(4, 5);CMP(6, 7);
CMP(0, 2);CMP(4, 6);CMP(1, 3);CMP(5, 7);
CMP(1, 2);CMP(5, 6);CMP(0, 4);CMP(1, 5);
CMP(1, 4);CMP(2, 6);CMP(3, 7);CMP(3, 6);
CMP(2, 4);CMP(3, 5);CMP(3, 4);
Mapping & Grouping
Original Grouping
0. S2(0, 1);S2(2, 3);S2(4, 5);S2(6, 7);
1. S2(0, 2);S2(4, 6);S2(1, 3);S2(5, 7);
2. S2(1, 2);S2(5, 6);S2(0, 4);
3. S2(1, 5);
4. S2(1, 4);S2(2, 6);S2(3, 7);
5. S2(3, 6);S2(2, 4);
6. S2(3, 5);
7. S2(3, 4);
Knuth diagram
0
1
2
3
4
5
6
7
Optimized Grouping
0. S2(0, 1);S2(2, 3);S2(4, 5);S2(6, 7);
1. S2(0, 2);S2(1, 3);S2(4, 6);S2(5, 7);
2. S2(1, 2);S2(5, 6);S2(0, 4);S2(3, 7);
3. S2(1, 5);S2(2, 6);
4. S2(1, 4);S2(3, 6);
5. S2(2, 4);S2(3, 5);
6. S2(3, 4);

Figure 6: Results of two grouping mechanisms for the Bose-Nelson sorting network. The input is the comparator sequence of the sorting network, and the output is the sequence of grouped S_2 comparison operators of DSL.

To generate the SIMD codes for the `aspas_sort`, we keep two sets of vector variables `a` and `b`. Initially, all the loaded vectors are stored in the set `a`. Then, we move through the groups of the sorting network. For each comparator in the current group, we generate the operations to compare the vector variables, and the results are stored to the set `b`. For those vectors not used in the current group, we directly copy them 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 are gone through. Finally, we obtain all the operations, which will be mapped to the ISA-specific intrinsics (`_mm512_max` and `_mm512_min`) 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 elements in column-major order. The next task is to gather them into the same vectors (i.e., rows). The gathering process corresponds to a matrix transpose. There are two alternative methods to achieve matrix transposition on Intel MIC: one uses the gather/scatter SIMD intrinsics introduced in AVX-512; and the other uses the in-register matrix transpose. The first solution provides a convenient means to handle the scattered data in memory, but with the penalty of high latency from the non-contiguous memory access. The second solution can avoid such a penalty but 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 have gather/scatter intrinsics, we choose the second solution for the *SIMD Transposer*. In order to decouple the binding between the operations of matrix transpose and the real intrinsics with various SIMD widths, we formalize the data-reordering operations using the sequence of the permutation operators. After that, we can hand it over to the *SIMD Code Generator* to generate the efficient SIMD code for the `aspas_transpose` function.

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

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

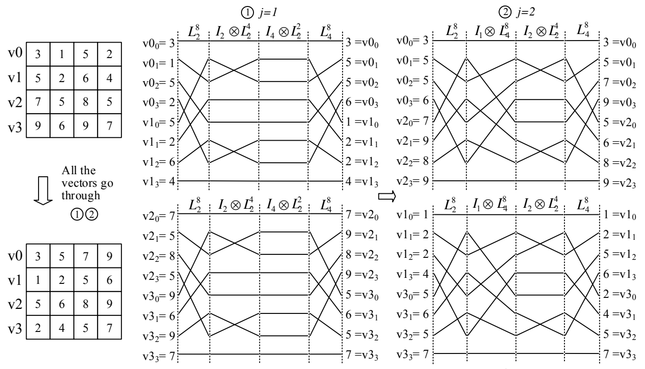


Figure 7: Four 4-element vectors transpose with the formalized permutation operators of DSL.

Fig. 7 illustrates an example with the SIMD width $w = 4$. The elements are preloaded to vectors $v0$, $v1$, $v2$, and $v3$ and have already been sorted vertically. Since $t - 1 = 2$, there are 2 steps denoted as ① and ② in the figure. When $j = 1$, the permutation operators are applied on the pairs $[v0, v1]$ and $[v2, v3]$; and when $j = 2$, the operations are on the pairs $[v0, v2]$ and $[v1, v3]$. The figure shows the values when the vectors go through the two steps accordingly. After that, the matrix is transposed, and the elements are gathered in the same vectors.

3.3 SIMD Merger

After the data is sorted in each segment using the `aspas_sort` and `aspas_transpose`, the `aspas_merge` is used as a kernel function to combine pairs of sorted data into a larger sequence iteratively. This function is generated by *SIMD Merger* based on appropriate merging networks, e.g., odd-even and bitonic networks. Here, we adopt the bitonic merging network for two reasons: (1) the bitonic merging network can be easily extended to any 2^n sized data; and (2) in each comparison step, all elements from the input vectors are processed, leading to symmetric operations on the elements. As a result, it is much easier to vectorize the bitonic merging network than others. From the perspective of the implementation, we have provided two variants of the bitonic merging networks [27], whose 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 Section 3.2, $t = log(2w)$, where w is the SIMD width of the vector units. The 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 changed in each step, while in Eq. 3, the permutation operators are independent with j, leading to the uniform data reordering patterns in each step. Therefore, we call the pattern in Eq. 2 as the inconsistent and that in Eq. 3 as the consistent. These patterns will be transmitted to *SIMD Code Generator* and generate the `aspas_merge` function. We will present the performance comparison of these two patterns in Section 4.

Figure 8 presents an example of these two variants of bitonic merging networks with the SIMD width $w = 4$.

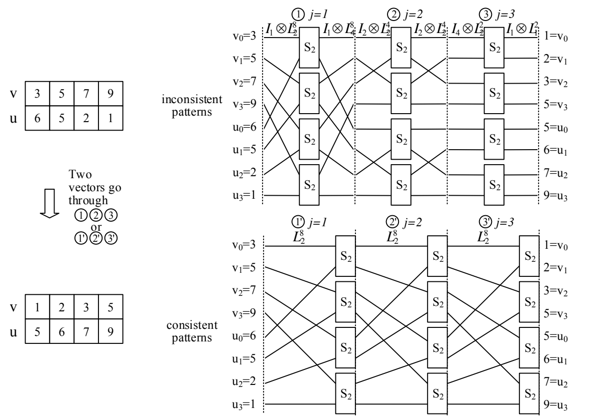


Figure 8: Two formalized variants of bitonic merging networks: the inconsistent pattern and the consistent pattern. All elements in vector v and u are sorted, but inversed in vector u.

As shown, the data-reordering operations derived from the inconsistent pattern are different in each step, while those from the consistent one are identical. Although the data-reordering operations adopted by these two variants are quite different, both of them can successfully achieve the merging functionality within the same number of steps, which is determined by the SIMD width w.

3.4 SIMD Code Generator

This module takes the data comparison operations from *SIMD Sorter* and the data-reordering operations from *SIMD Transposer* and *SIMD Merger* as the input to produce the real vectorized codes. Considering the simplicity of mapping the data comparison operations to real SIMD intrinsics, in this section, we only focus on how to find the most efficient intrinsics sequence to attain the given data-reordering operations. Our method is first to build a primitive pool based on the characteristics of the data-ordering operations, then dynamically build the primitive sequences based on the *matching score* between what we have achieved on the way and the targeting pattern, and finally map the selected primitives to the real intrinsics.

3.4.1 Primitive Pool Building

Different with the previous research, e.g. the automatic Fast Fourier transform (FFT) vectorization [17] using the exhaustive and heuristic search on all possible intrinsics combinations, we first build an ISA-friendly primitive pool to prune the search space. 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 the input are equivalent with those on the second half in a mirror style. So, to produce efficient operation sequences, we assume all the components of the sequences 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: For Intel MIC, there are 4 32-bit words (e.g. Integer or Float) per lane and 4 lanes in total for each vector register. Although there are $4^4 = 256$ possibilities for either intra-lane or inter-lane permute operations, we only consider those permutations without repetition and thus reduce the possibilities to $4! = 24$. Among them, only

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.

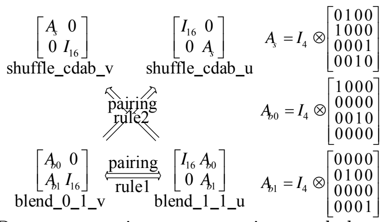
Blend Primitives: To guarantee the elements of the blended vector are from the two input vectors equally and symmetrically, we boil down the numerous mask modifiers to a few pairs of patterns. We define a pair as $(0\_2^i,\ 2^i\_2^i)$, where $0 \leqslant i < log(w)$ and w is the vector length. Each pattern in the pair represents a 2^{i+1}-bit stream. The first number 0 or 2^i represents the offset of the first 1, and the second number 2^i is the number of consecutive 1s. All the other bits in the bit stream are filled with 0s. The bit streams will be extended to the real masks by duplicating themselves $\frac{w}{2^{i+1}}$ times. For Intel MIC with the w equal to 16, there are 4 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), where i is equal to 1, represents the bit streams 1100 and 0011. The corresponding masks are $(1100)_4$ and $(0011)_4$.

Now, we can categorize the primitives into 4 types based on permute or blend and intra-lane or inter-lane. Table 1 illustrates the categories and the corresponding operations, where the vector width w is set to 8 (containing 2 lanes) to save the space.

Table 1: Primitive Types

Type	Example (vector_width=8)
intra-lane-permute	ABCDEFGH→BADCFEHG (cdab)
inter-lane-permute	ABCDEFGH→EFGHABCD (−−ab)
intra-lane-blend	ABCDEFGH|IJKLMNOP→ABKLEFOP (2_2)
inter-lane-blend	ABCDEFGH|IJKLMNOP→IJKLEFGH (0_4)

ASPaS stores these primitives as the permutation matrices. Since the blend primitives always work over two input vectors, the input can be concatenated as one $2w$ vector and the dimensions of the permutation matrices are expanded to $2w$ by $2w$. In order to unify the format, for the permute primitives, we add an empty vector and specify the primitive operates on the first vector v or the second vector u. Therefore, on MIC, 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. 9 illustrates examples of the permutation matrices. The matrix "shuffle_cdab_v" and "shuffle_cdab_u" correspond to the same permute primitive on two halves of the concatenated vector. The matrix "blend_0_1_v" and "blend_1_1_u" correspond to one pair of blend primitives (0_1, 1_1). Finally, 4 sub-pools of permutation matrices are created according to the 4 primitive types.

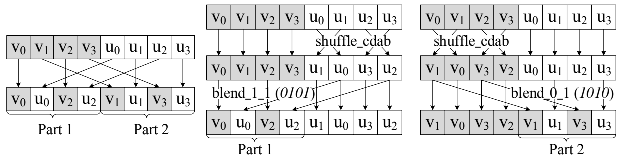


$\begin{bmatrix} A_s & 0 \\ 0 & I_{16} \end{bmatrix}$
shuffle_cdab_v
$\begin{bmatrix} I_{16} & 0 \\ 0 & A_s \end{bmatrix}$
shuffle_cdab_u
$A_s = I_4 \otimes \begin{bmatrix} 0100 \\ 1000 \\ 0001 \\ 0010 \end{bmatrix}$
pairing rule2
$A_{b0} = I_4 \otimes \begin{bmatrix} 1000 \\ 0000 \\ 0010 \\ 0000 \end{bmatrix}$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

Figure 9: Permute matrix representations and the pairing rules

3.4.2 *Sequence Building*

ASPaS takes two rules in the sequence building algorithm to facilitate the building process. The rules are based on two observations from the formalized data-reordering operations illustrated in Eq. 1, Eq. 2, and Eq. 3. Obs.1 The data-reordering operations are always conducted on two input vectors. Obs.2 The permute operations always accompany the blend operations to keep them symmetric. Fig. 10 shows the data-reordering pattern of a symmetric blend, which is essentially the first step in Fig. 7. Because such an interleaving mode of symmetric blend cannot be directly achieved by the blend primitives, which are limited to pick elements from aligned positions of two input vectors, the associative permute primitives are necessary and coupled with the blend primitives, as the figure shown. Hence, two rules used 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 mask, 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 with CDAB pattern for (0_1, 1_1) blend, the intra-lane-permute with BADC for (0_2, 2_2), the inter-lane-permute with CDAB for (0_4, 4_4), and the inter-lane-permute with BADC for (0_8, 8_8).



(a) sym-blend (b) sym-blend(details)
Figure 10: Symmetric blend operation and its pairing details

The sequence building algorithm can generate sequences of primitives to implement a given data-reordering pattern for Eq. 1, Eq. 2, and Eq. 3. Two $2w$-sized vectors are used as the input. The vec_{inp} represents the original concatenated vector of v and u and is set to the default offsets (from 1 to $2w$). The vec_{trg} is the target derived by applying the desired data-reordering operators on the vec_{inp}. Our algorithm will select the permutation matrices from the primitive pool, do the vector-matrix multiplications on the vec_{inp}, and check whether the intermediate result vec_{im} approximates the vec_{trg}. We define two types of *matching score* to exhibit how well the vec_{im} matches the vec_{trg}:

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}.

If only considering the 32-bit words with 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 score computations. For example, if we have the input "ABCDEFGH" and the output "HGDCFEBA" (assuming four lanes and two elements per lane), we first search primitives for the inter-lane reordering, e.g, from "ABCDEFGH" to "GHCDEFAB", and then search primitives for the intra-lane reordering, e.g., from "GHCDEFAB" to "HGDCFEBA". By checking the primitives hierarchically, we add those primitives increasing l-score or e-score and approximating to the desired output pattern.

Algorithm 2: Sequence Building Algorithm

```
   Input: $vec_{\mathsf{inp}}, vec_{\mathsf{trg}}$
   Output: $seqs_{\mathsf{ret}}$
 1 Sequences $seqs_{\mathsf{cand}}$ ← new Sequences(∅);   // put an null sequence
 2 Int $l\_score_{\mathsf{init}}$ ←LaneCmp($vec_{\mathsf{inp}}, vec_{\mathsf{trg}}$);
 3 if $l\_score_{\mathsf{init}}$=2w/e then
 4 |   $seqs_{\mathsf{ret}}$ ← InstSelector($seqs_{\mathsf{cand}}$,Type[0]);
 5 else
 6 |   $i$ ←1;
 7 |   while not Threshold() do
 8 |   |   $ty$ ←Type[$i$];
 9 |   |   foreach Sequence $seq$ in $seqs_{\mathsf{cand}}$ do
10 |   |   |   $vec_{\mathsf{im}}$ ←Apply($vec_{\mathsf{inp}}, seq$);
11 |   |   |   $l\_score_{\mathsf{old}}$ ←LaneCmp($vec_{\mathsf{im}}, vec_{\mathsf{trg}}$);
12 |   |   |   foreach Primitive $prim$ in $ty$ do
13 |   |   |   |   if $n$=1 then
14 |   |   |   |   |   $prim_{\mathsf{prd}}$ ←Pair($prim$, RULE1);
15 |   |   |   |   |   $vec_{\mathsf{upd}}$ ←Apply($vec_{\mathsf{im}}$, $prim + prim_{\mathsf{prd}}$);
16 |   |   |   |   |   $seq_{\mathsf{ext}}$ ←$prim + prim_{\mathsf{prd}}$;
17 |   |   |   |   else
18 |   |   |   |   |   $prim_{\mathsf{prd}}$ ←Pair($prim$, RULE1);
19 |   |   |   |   |   $perm_0$ ←Pair($prim$, RULE2);
20 |   |   |   |   |   $perm_1$ ←Pair($prim_{\mathsf{prd}}$, RULE2);
21 |   |   |   |   |   $vec_{\mathsf{upd0}}$ ←Apply($vec_{\mathsf{im}}$, $perm_0 + prim$);
22 |   |   |   |   |   $vec_{\mathsf{upd1}}$ ←Apply($vec_{\mathsf{im}}$, $perm_1 + prim_{\mathsf{prd}}$);
23 |   |   |   |   |   $vec_{\mathsf{upd}}$ ←Combine($vec_{\mathsf{upd0}}, vec_{\mathsf{upd1}}$);
24 |   |   |   |   |   $seq_{\mathsf{ext}}$ ←$perm_0 + prim + perm_1 + prim_{\mathsf{prd}}$;
25 |   |   |   |   $l\_score_{\mathsf{new}}$ ←LaneCmp($vec_{\mathsf{upd}}, vec_{\mathsf{trg}}$);
26 |   |   |   |   if $l\_score_{\mathsf{new}} > l\_score_{\mathsf{old}}$ then
27 |   |   |   |   |   $seqs_{\mathsf{buf}}$.add($seq + seq_{\mathsf{ext}}$);
28 |   |   $seqs_{\mathsf{cand}}$.append($seqs_{\mathsf{buf}}$);
29 |   |   $seqs_{\mathsf{buf}}$.clear();
30 |   |   $i$ ←((++$i$)-1)%3+1;
31 |   $seqs_{\mathsf{sel}}$ ←PickLaneMatchedSeqs($seqs_{\mathsf{cand}}$);
32 |   $seqs_{\mathsf{ret}}$ ← InstSelector($seqs_{\mathsf{sel}}$,Type[0]);
33 Function InstSelector(Sequences $seqs_{\mathsf{cand}}$,Type $ty$)
34 |   foreach Sequence $seq$ in $seqs_{\mathsf{cand}}$ do
35 |   |   $vec_{\mathsf{im}}$ ←Apply($vec_{\mathsf{inp}}, seq$);
36 |   |   foreach Primitive $prim$ in $ty$ do
37 |   |   |   $prim_{\mathsf{prd}}$ ←Pair($prim$, RULE1);
38 |   |   |   $vec_{\mathsf{upd}}$ ←Apply($vec_{\mathsf{im}}, prim + prim_{\mathsf{prd}}$);
39 |   |   |   $e\_score$ ←ElemCmp($vec_{\mathsf{upd}}, vec_{\mathsf{trg}}$);
40 |   |   |   if $e\_score$=2w then
41 |   |   |   |   $seqs_{\mathsf{ret}}$.add($seq + prim + prim_{\mathsf{prd}}$);
42 |   return $seqs_{\mathsf{ret}}$;
```

Alg.2 shows the pseudocode of the sequence building algorithm. The input includes the aforementioned vec_{inp} and vec_{trg}. The output $seqs_{\mathsf{ret}}$ is an array to hold selected sequences of primitives, which will be used to generate the real ISA intrinsics later. The $seqs_{\mathsf{cand}}$ is used to store candidate sequences and initialized to contain an empty sequence. First, the algorithm checks the initial vec_{inp} with the vec_{trg} and get the l-score. If it equals to $2w/e$, which means corresponding lanes have matched, we only need to select primitives from "intra-lane-permute" (ln.4) and care about 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 get l_score_{new} by checking primitives one by one from the current type. 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 $seqs_{\mathsf{cand}}$. 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 so as 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.

Now, we can map the sequences from the $seqs_{\mathsf{ret}}$ to the real ISA intrinsics. We have two criteria to pick up the efficient sequences. The first criterion is the length of the sequence. We prefer the shortest one. The second criterion is when multiple shortest solutions exist, we prefer the interleaved style of inter-lane and intra-lane primitives, which could be executed with a pipeline mode on Intel MIC as discussed in Sec.2.1. When the most efficient sequences are collected, we convert them into real intrinsics. 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 corresponding permute parameters. For the primitives of "intra-lane-blend" and "inter-lane-blend", we map them to the masked variants of permute intrinsics `_mm512_mask_shuffle` and `_mm512_mask_permute 4f128`. The masks are from their blend patterns. If the primitive belongs to "intra-lane" and the permute parameter is supported by the swizzle intrinsics, we will use the light-weighted swizzle intrinsics instead.

3.5 Thread-level Parallelism

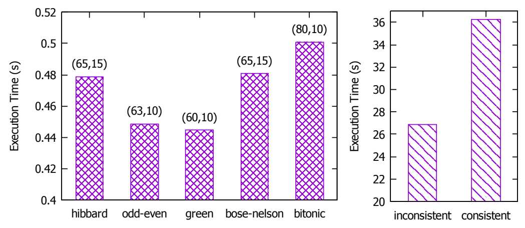
After generating the intrinsics sequences for the vector comparators and the data-reordering operators in the `aspas_sort`, `aspas_transpose`, and `aspas_merge`, we have the single threaded version of the `aspas::sort` and `aspas::merge` as illustrated in Alg.1. In order to maximize the utilization of the computational resources of Intel MIC, we integrate these two functions with the thread-level parallelism using Pthreads. First, we make each thread work on their own parts of the input data by using the `aspas::sort`. Second, we use 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. In the evaluation, our multi-threaded version refers to this design.

4. PERFORMANCE ANALYSIS

In our evaluation, we use the Integer for the one-word type and the Double for the two-word type. Our experiments are conducted on the Intel Xeon Phi 5110P coprocessor with the codename Knight Corner, which has 60 cores running on 1.05 GHz with 8 GB GDDR5 memory and includes 32 KB L1 cache and 512 KB L2 cache. The compiler is Intel *icpc* 13.0.1, and the compiler options include *-mmic* and *-O3*. We run all experiments using the *native* mode. All the data are generated randomly ranging from 0 to the data size.

4.1 Performance of Different Sorting Networks

In the *sort* stage, ASPaS can accept any type of sorting networks and generate corresponding `aspas_sort` function. We use five sorting networks corresponding to five sorting algorithms, including hibbard[11], odd-even [3], green[10], bose-nelson[4], and bitonic[3]. Fig. 11a illustrates the performance of the `aspas_sort` for five sorting networks over an array with 100 million integers. The data labels also show how many comparators and groups of comparators in each sorting network. Green sort has the best performance that stems from the less comparators and groups, i.e., (60, 10). Although bitonic sort follows a balanced way to compare all elements in each step and is usually considered as the candidate for better performance, it uses more comparators, leading to the relatively weak performance for the *sort* stage. In the remaining experiments, we choose green sort as the sort algorithm for the Integer datatype. For the Double datatype, we choose the second best, i.e., odd-even sort, because green sort cannot take 8 elements as the input.



(a) `aspas_sort` (b) `aspas_merge`

Figure 11: Performance comparison of `aspas_sort` and `aspas_merge` for different sorting and merging networks. In (a), the number of comparators and groups are in parenthesis.

In the *merge* stage, ASPaS can solve two variants of bitonic merging networks as shown in Eq.2 and Eq.3 of Sec.3.3. Figure 11b presents the performance comparison for these two variants. The inconsistent merging can outperform the consistent one by 28.6%. Although the consistent merging has uniform data-reordering operations in each step as shown in Fig. 8, the operations are not ISA-friendly and thus requires a longer intrinsic sequence on Intel MIC. Based on Eq.3, the consistent merging uses 5 times of the L_2^{32} data-reordering operations, each of which needs 8 permute/shuffle intrinsics on Intel MIC. In contrast, the inconsistent merging only uses L_2^{32} once and compensate it with much lighter data-reordering operations, including $I_1 \otimes L_{16}^{32} \circ I_2 \otimes L_2^{16}$ and $I_2 \otimes L_8^{16} \circ I_4 \otimes L_2^{8}$, each of which can be implemented by only 2 intrinsics on Intel MIC. Therefore, we will adopt the inconsistent bitonic merge in the remaining experiments for better performance.

4.2 Vectorization Efficiency

To compare with the auto-vectorized code, we implement a serial sorting based on Alg. 1. Because the serial sorting doesn't store the partially sorted data in a scattered manner, the `aspas_transpose` function has no serial counterpart. We use the Intel compiler option *-vec-* to turn off the vectorization for the serial version and denote it as "no-vec" in the figures. On the other hand, we use the aggressive *pragma simd* and option *-O3* to guide the compiler to generate the vectorized code and denote it as "Comp-vec".

Fig. 12a shows the performance comparison for the *sort* stage with the Integer datatype. Compared with the "no-vec" version without the compiler auto-vectorization, the code generated by ASPaS can get 7.7-fold speedup. Using the Green sorting network, 60 comparators are needed to sort every 16 elements. As a result, for every 16*16 elements, the "no-vec" version needs $16 * 60 = 960$ comparators to partially sort the data. In contrast, the code generated by ASPaS only needs 60 vector comparators, and then uses 4 data-reordering operations (Eq.1) for each pair of vectors in the `aspas_transpose` to transpose the data. In total, the ASPaS code needs $60 + (16/2) * 4 = 92$ vector operations. The theoretical speedup should be $960/92 = 10.4$-fold. The experiment shows we can achieve 74% of the theoretical speedup for the *sort* stage. The figure also shows that the ASPaS code can outperform the "auto-vec" version. Actually, the auto-vectorized code utilizes the gather/scatter instructions to get/put non-contiguous data from/to memory on Intel MIC. However, it cannot mitigate the high latency of non-contiguous memory access. ASPaS can still outperform it by 1.6-fold by using the load/store intrinsics on the contiguous data and the shuffle/permute intrinsics for the transpose in registers.

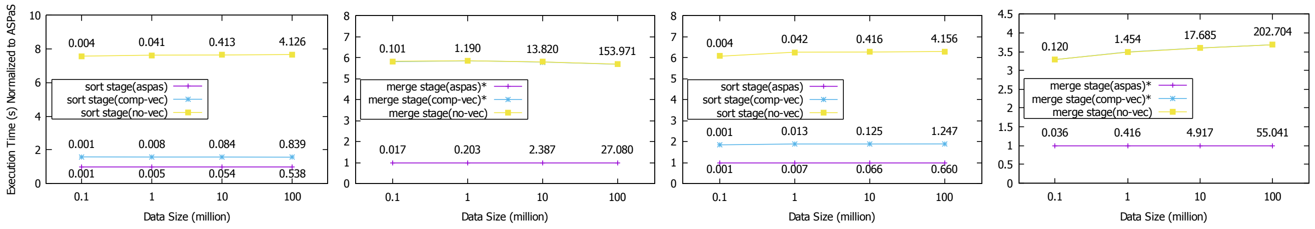
Fig. 12b presents the performance comparison for the *merge* stage with the Integer datatype. Compared with the "no-vec" version, the ASPaS code can achieve 5.7-fold speedup. Since we use the bitonic merging to merge two sorted 16-element vectors, 80 comparators are needed in the "no-vec" version. In contrast, the ASPaS code only needs 5 comparisons with additional 5 data-reordering operations as shown in Eq.2. This provides a theoretical upper bound of $80/10 = 8$-fold speedup. The results show ASPaS can achieve 71% of the theoretical speedup for the *merge* stage. Note that the "auto-vec" version has the same performance with the "no-vec" version, meaning that even with the most aggressive vectorization pragma, the compiler fails to vectorize the merge code due to the complex data dependency in the loop.

Similarly, we also show the results for the Double datatype in Fig. 12c and Fig. 12d. The ASPaS *sort* and *merge* stages can outperform their counterparts by 6.3-fold and 3.7-fold, respectively.

4.3 Comparison to Sorting from Libraries

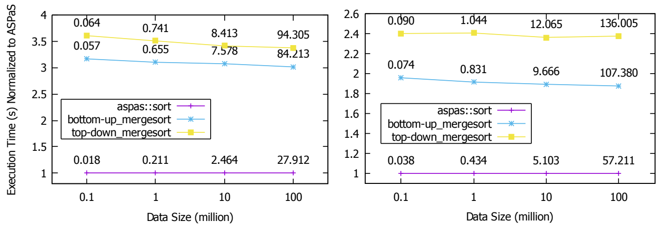
We first focus on the single threaded sorting. Since ASPaS uses the similar scheme with the mergesort, we compare the `aspas::sort` with two mergesort variants: top-down and bottom-up. The top-down mergesort recursively splits the input array until the split segments only have one element. Subsequently, the segments are merged together. In contrast, the bottom-up variant directly works on the elements and iteratively merge them into sorted segments. Fig. 13 illustrates the corresponding performance comparison. The `aspas::sort` for Integer datatype can obtain 3.4-fold and 3-fold speedup over the top-down and bottom-up mergesort, respectively; while for Double datatype, the `aspas::sort` can obtain 2.4-fold and 1.9-fold speedup instead.

Note that the speedups of `aspas::sort` over the top-down and bottom-up mergesorts are smaller than what we have reached in Sec.4.2 when compared to the sorting denoted as "no-vec". The "no-vec" sorting is a serial implementation of Alg. 1 using essentially the scheme of a bottom-up mergesort. However, the scheme induces extra comparisons for the ease of vectorization. When merging each pair of two sorted segments, in the case of one-word elements, we fetch w elements into a buffer from each segment and then merge these $2w$ elements using the 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). In contrast, the top-down and bottom-up mergesort keep two pointers, each of which points to the head of its sorted segment, and continuously fetch the smaller element one by one. Therefore, the



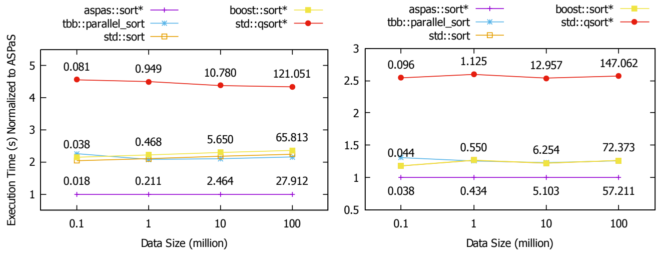
(a) *sort* stage (`int`) (b) *merge* stage (`int`) (c) *sort* stage (`double`) (d) *merge* stage (`double`)
Figure 12: ASPaS codes vs. *icpc* optimized codes (The sorts asterisked are labeled with the execution time, the same below). In (b) and (d), the lines of "comp-vec" and "no-vec" are overlapped.

comparisons in these two variants are considerably less than what we use in Alg. 1. However, because of the more potential for the vectorization in the scheme of Alg. 1, we observe better performance of `aspas::sort` over the top-down and bottom-up mergesorts.



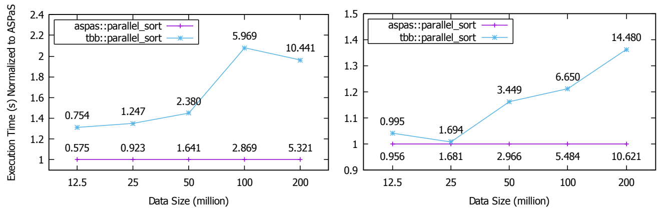
(a) vs. mergesorts (`int`) (b) vs. mergesorts (`double`)
Figure 13: `aspas::sort` vs. top-down and bottom-up mergesort

For the single threaded `aspas::sort`, we also compare it with other sorting tools from those widely-used libraries, including the `qsort` and `sort` from STL, `sort` from Boost library, and the `parallel_sort` from Intel TBB (with a single thread). Fig. 14a exhibits that `aspas::sort` for the Integer datatype can achieve up to 4.3-fold speedup over the `qsort`, and up to 2.4-fold speedup over others. For the Double datatype in Fig. 14b, the `aspas::sort` can attain 2.6-fold speedup over the `qsort` and 1.3-fold speedup over others.



(a) vs. sorting tools (`int`) (b) vs. sorting tools (`double`)
Figure 14: `aspas::sort` vs. other sorting tools

Then, we compare our multi-threaded version described in Sec.3.5 with the `parallel_sort` from Intel TBB. We choose a larger dataset from 12.5 million to 200 million integer or double elements. As shown in Fig. 15, our multi-threaded version can outperform the `parallel_sort` by up to 2.1-fold speedup for the Integer datatype and 1.4-fold speedup for the Double datatype. It is worth mentioning that both of these two parallel sort functions can achieve their best performance while using 60 threads and each on a dedicated core.



(a) vs. tbb::parallel_sort (`int`) (b) vs. tbb::parallel_sort (`double`)
Figure 15: `aspas::parallel_sort` vs. TBB's parallel sort

4.4 Discussion

Merging Networks: Currently, ASPaS supports two variants of bitonic merging networks, because their corresponding permutation operators can symmetrically and completely utilize all elements in each input vector. In contrast, other merging networks, e.g., odd-even network, cannot rearrange all elements and need more masks to hide irrelevant elements in each concurrent step, leading to additional overhead. On the other hand, once those additional data permute and blend primitives for other merging networks are selected out, ASPaS can add them into the primitive pool and generate vectorized merge functions.

Portability: ASPaS can also generate parallel sorting for x86-based multicore processors. We only need to modify one portion in the *SIMD Code Generator*, i.e., how to translate the primitives to the read ISA intrinsics. For the permute primitives, SSE, AVX, and AVX-2 of CPUs also provide the lane-level and element-level permutation intrinsics, such as `_mm256_shuffle` and `_mm256_2f128`. Therefore, these primitives can be also directly mapped to intrinsics. In contrast, for the blend primitives, the behavior of mask on CPUs is quite limited and most intrinsics have no masked variants. Alternatively, we use other intrinsics to emulate similar blend functionalities. For example, on AVX, we use the `_mm256_unpacklo` and `_mm256_unpackhi` intrinsics to implement the intra-lane blend primitives.

5. RELATED WORK

Many sorting algorithms have been modified and optimized to utilize both of the multiple cores and the intra-core parallelism on CPUs, GPU and MIC. Davidson et al.[6] designs a parallel merge sort by increasing the register communication on GPUs. Satish et al.[22, 23] compares and analyzes the radix sort and merge sort on modern accelerators, such as CPUs and GPUs. Chhugani et al.[5] show a SIMD-friendly merge sorting algorithm by using the sorting networks on multicore CPUs. AA-Sort in [13] is a new parallel sorting algorithm for utilizing the SIMD units of multicore CPUs. Their algorithm focuses on taking advantage of SIMD instructions and eliminating the unaligned memory access pattern. HykSort in [25] targets at distributed memory architectures, but on a single node, HykSort also uses SIMD optimized merge sort. For the past work of using the vector resources, developers have to explicitly use the compiler intrinsics to handle the tricky data-reordering operations required by the algorithm. Even with some SIMD-friendly programming compilers (e.g. ISPC [18]), developers still need to refer to the `shuffle()` functions and deal with the corresponding permute parameters.

Some frameworks are proposed to automatically generate application codes to utilize modern parallel architectures. Mint and Physis in [26, 16] can generate effective GPU codes for stencil computations. Benson et al.[2] provides a code generation tool to automatically implement various matrix multiplication algorithms. To facilitate the utilization of the intra-core resources, Huo et al.[12] presents a system with runtime SIMD parallelization with override operators and functions. Operator Language described in [9] is a framework to generate efficient vector load and shuffle instructions to achieve desired data-reordering pattern based on a rewriting system and a search mechanism. McFarlin et al.[17] shows another superoptimizer to conduct a guided search of the shortest sequence of instructions over a large candidate pool. Compared to the past work, the ASPaS uses a pruned ISA-fridendly primitive pools which are derived from the symmetric data-reordering patterns in the parallel sorting.

6. CONCLUSION

In this paper, we propose the ASPaS framework to automatically generate vectorized sorting code for x-86 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. With ASPaS, we generate various parallel sorting codes on Intel MIC. The performance evaluation illustrates our automatically generated codes can outperform multiple sorting functions from STL, Boost, and Intel TBB. In the future, we will generate sorting codes for multicore architectures, and look for a theoretical model to verify the generated codes can obtain the best performance on the given ISA.

7. ACKNOWLEDGEMENT

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

8. REFERENCES

[1] S. W. Al-Haj Baddar and K. W. Batcher. *Designing Sorting Networks: A New Paradigm.* Springer, 2011.

[2] Austin R. Benson and Grey Ballard. A Framework for Practical Parallel Fast Matrix Multiplication. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[3] K. E. Batcher. Sorting Networks and Their Applications. In *Proc. of ACM Spring Joint Computer Conf.*, 1968.

[4] R. C. Bose and R. J. Nelson. A Sorting Problem. *J. ACM*, 9(2), 1962.

[5] 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. of the VLDB Endowment (PVLDB)*, 2008.

[6] A. Davidson, D. Tarjan, M. Garland, and J. Owens. Efficient Parallel Merge Sort for Fixed and Variable Length Keys. In *Innovative Parallel Computing (InPar)*, 2012.

[7] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency. White Paper Intel Co, 2008.

[8] F. Franchetti, F. Mesmay, D. Mcfarlin, and M. Püschel. Operator Language: A Program Generation Framework for Fast Kernels. In *Proc. of the IFIP TC 2 Working Conf. on Domain-Specific Languages (DSL)*, 2009.

[9] F. Franchetti and M. Püschel. Generating SIMD Vectorized Permutations. In *Proc. of the Joint Eur. Conf. on Theory and Practice of Software Int'l Conf. on Compiler Construction (CC/ETAPS)*, 2008.

[10] M. W. Green. Some Improvements in Non-adaptive Sorting Algorithms. In *Proc. of the Annual Princeton Conf. on Information Sciences and Systems*, 1972.

[11] T. N. Hibbard. An empirical study of minimal storage sorting. *Commun. ACM*, 6(5), 1963.

[12] X. Huo, B. Ren, and G. Agrawal. A Programming System for Xeon Phis with Runtime SIMD Parallelization. In *Proc. of the ACM Int'l Conf. on Supercomputing (ICS)*, 2014.

[13] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *Proc. of the ACM Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[14] Intel. Intel Xeon Phi Coprocessor System Software Developers Guide, 2012. `https://software.intel.com/sites/default/files/article/334766/intel-xeon-phi-systemsoftwaredevelopersguide.pdf`.

[15] S. Maleki, Y. Gao, M. Garzaran, T. Wong, and D. Padua. An Evaluation of Vectorizing Compilers. In *Proc. of the ACM Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[16] 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 *Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[17] 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 *Proc. of the ACM Int'l Conf. on Supercomputing (ICS)*, 2011.

[18] M. Pharr and W. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar)*, 2012.

[19] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Lib.* Prentice Hall PTR, 1st edition, 2000.

[20] R. Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers.* Apress, 1st edition, 2013.

[21] J. Reinders. *Intel Threading Building Blocks.* O'Reilly & Associates, Inc., 1st edition, 2007.

[22] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *IEEE Int'l Symp. on Parallel Distributed Processing (IPDPS)*, 2009.

[23] 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 *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 2010.

[24] B. Schling. *The Boost C++ Libraries.* XML Press, 2011.

[25] H. Sundar, D. Malhotra, and G. Biros. HykSort: A New Variant of Hypercube Quicksort on Distributed Memory Architectures. In *Proc. of the ACM Int'l Conf. on Supercomputing (ICS)*, 2013.

[26] D. Unat, X. Cai, and S. B. Baden. Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In *Proc. of the ACM Int'l Conf. on Supercomputing (ICS)*, 2011.

[27] M. Zuluaga, P. Milder, and M. Püschel. Computer Generation of Streaming Sorting Networks. In *Proc. of the ACM Design Automation Conf. (DAC)*, 2012.