A Fast and Simple Approach to Merge Sorting using AVX-512

By Alex Watkins

Undergraduate Research Thesis
Georgia Institute of Technology
Submitted Fall 2017

Advisor: Dr. Oded Green

Secondary Reviewer: Dr. Jeff Young

A Fast and Simple Approach to Merge Sorting using AVX-512

Alex Watkins[1]

[1]Computational Science and Engineering, Georgia Institute of Technology- USA

***Abstract*— Merging and Sorting algorithms are the backbone of many modern computer applications. As such, efficient implementations are desired. New architectural advancements in CPUs allow for ever-present algorithmic improvements. This research presents a new approach to Merge Sorting using SIMD (Single Instruction Multiple Data). Traditional approaches to SIMD sorting typically utilize a bitonic sorting network (Batcher's Algorithm) which ads significant overhead. Our approach eliminates the overhead from this approach. We start with a branchless merge algorithm and then use the Merge Path algorithm to split up merging between the different SIMD paths. Testing demonstrates that the algorithm not only surpasses the SIMD based bitonic counterpart, but that it is over 2.94 times faster than a standard merge, merging over 300M elements per second. A full sort reaches to over 5x faster than a quicksort and 2x faster than Intel's IPP library sort, sorting over 5.3M keys per second. A 256 thread parallel sort reaches over 500M keys per second and a speedup of over 2x from a regular merge sort. These results make it the fastest sort on Intel's KNL processors that we know of.**

I. Introduction

Sorting algorithms are a crucial aspect in computer applications [1]. Database systems use sorting algorithms to organize internal data and to present the data to the users in a sorted format. Graph searching uses sorting to speedup lookups. A variety of high-speed sorting algorithms have been proposed including quicksort [2], merge sort, radix sort, Batcher's algorithm (bitonic) [3], and several others [1], [4], [5]. Each sorting algorithm contains its own advantages and disadvantages. Quick sort works great in the general case but does not scale well in parallel. Radix sort has a favorable efficiency but suffers from memory bandwidth overhead. Because processor architecture changes constantly, sorting algorithms need to be reevaluated in conjunction with new architectures. The naturally independent merges in the lower layers of a merge sort coupled with Merge Path [6] in the upper layers make it ideal for a well balanced parallel sort. The objective of this paper is to present a novel approach to the merging of sorted sub-arrays using vector instructions as well as an efficient and scalable sorting algorithm designed for Intel's AVX-512 instruction set.

A common feature algorithms use is SIMD (Single Instruction Multiple Data) instructions , a popular accelerator component found in modern architectures such as x86. Many previous sorting implementations have taken advantage of SIMD to further improve speed [5], [7], [8], [9], [10], [11], [12], [13], [14]. For example, [7] showed a version of bitonic sort using 128-bit wide SIMD. This approach requires numerous unnecessary comparisons during the pairwise merge of sorted sub-arrays since some comparisons from each stage cannot be reused in later stages and must be compared again. AVX-512 is a huge step in SIMD architecture not only because of the larger width and increased number of supported data elements, but because the gather and scatter instructions allow for efficient non-sequential memory accesses. Our algorithm utilizes these instructions and as such can better utilize each core.

Our new algorithm uses new SIMD instructions to merge sort independent arrays using a very different method than the previous bitonic implementation. Assuming 32-bit data and 512-bit SIMD vector, 16 elements can be merged at a time. Extending the Merge Path concept [6], which requires two sorted arrays and enables merging using 32 sub arrays on a single core using the vector instructions. Our algorithm partitions two sorted arrays into smaller sub-arrays that can be merged independently and anonymously of each other, each in a different SIMD thread. The overhead of Merge Path is relatively low and enables improved performance over past algorithms. Testing demonstrates that the algorithm not only surpasses the SIMD based bitonic counterpart, but that it is over 2.94 times faster than a standard merge merging over 300M elements per second.

Since partitioning is not needed when the number of merges to be made in a single round exceeds the SIMD width, we also present two AVX-512 merge algorithms that do not use Merge Path but still follow the branch avoiding merge pattern. In order to achieve an optimal algorithm all three of these AVX-512 merge algorithms are combined into one unified sorting algorithm. A full sort reaches to over 5x faster than a quicksort and 2x faster than Intel's IPP library sort, sorting over 5.3M keys per second. This sorting algorithm can be run in parallel using Merge Path to split the merging between threads. Using the full KNL system at 256 thread the parallel sort reaches over 500M keys per second and a speedup of over 2x from a regular merge sort.

II. Related Work

CPU architecture changes rapidly adding features like increased memory bandwidth, SIMD width, number of

Variable	Description
$AIndex, BIndex, CIndex$	Current index in A,B, and C, respectively.
$AStop, BStop$	Max index for A and B, respectively.
$AEndMask, BEndMask$	Mask marking whether a sub-array has elapsed in A and B, respectively.
$maskA, maskB$	Comparison mask for A and B, respectively.
cmp	Comparison mask
$AElems, BElems, CElems$	Vector values of A, B, and C, sub-arrays respectively.
$n, \mid A \mid, \mid B \mid, \mid C \mid$	Array size
p	Number of Threads

TABLE I
DESCRIPTION OF IMPORTANT VARIABLES USED IN PSEUDOCODE

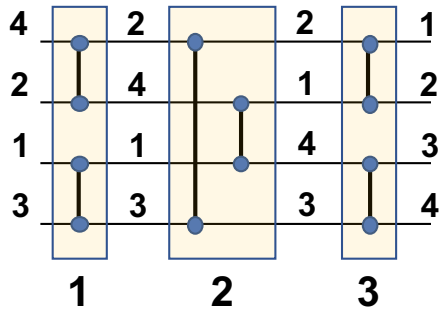

Fig. 1. 3 level bitonic sorting network. Each vertical line between two points represents a comparison between those points. Comparisons that are boxed can be done simultaneously.

Instruction	Description
`Gather_W`(src*,indicies)	instruction used for loading multiple memory locations
`Scatter_W`(dest*,elems, indicies, mask)	stores data at multiple memory locations if mask bit is set
`Load_W`(src*)	loads a sequential segment of memory
`Store_W`(dest*,elems)	store a sequential segment of values to memory
`CompareLT_W`(A,B)	pairwise SIMD vector comparison (less than)
`CompareGT_W`(A,B)	pairwise SIMD vector comparison (greater than)
`Blend_W`(A,B,mask)	selectively pulls elements from vector B if the mask bit is set and from vector A otherwise
`Add_W`(A,value)	adds a value to each element in the given A vector
`MaskAdd_W`(A,mask, value)	Adds a value to each element in the given A vector only if the corresponding mask bit is set for that element. If the mask bit is not set, the original element is placed as is.
`BitAnd_W`(maskA, maskB)	bitwise and for masks
`BitOr_W`(maskA,maskB)	bitwise or for masks
`LeftShift_W`(A,amount)	left shift each element in A by amount

TABLE II
SUBSET OF SIMD INSTRUCTIONS USED BY OUR MERGING AND SORTING ALGORITHMS. W DEMOTES THE WIDTH OF THE VECTOR INSTRUCTION.

threads/cores, and other advances allowing more instruction level parallelism, data level parallelism, thread level parallelism, and memory level parallelism [4], [15], [16], [17], [18], [9], [19], [20], [18], [21]. To have an effective and efficient algorithm all factors of the modern CPU need to be utilized. The following examines sorting or sorting related algorithms with a focus on these hardware improvements.

A. Sorting Networks

One approach to sorting is using a sorting network. Sorting networks involve a fixed set of comparisons that are made each sort. Traditional computers implemented these in hardware for more efficient sorting. Some modern devices such as FPGAs sometimes still do this [23].

One commonly used sorting network is the bitonic sorting network (also known as Batcher's Algorithm) [3]. Bitonic involves a multilevel network with multiple comparisons at each level. The network depth is proportional to the size of the network. Figure 1 shows a 4 input network consisting of 3 levels of comparisons. Values are compared pairwise across the vertical lines. Then, the smaller sorted arrays are merged together to produce the final array.

[7] produced a version of bitonic using Intel's SSE instructions. This approach performed has continueoly been referenced amoung SIMD based sorting for comparison. Their impleentation uses bitonic as a merge to produce something similar to a merge sort using a bitonic merger. [7]'s full sort was about 3.3x faster than standard. The main overhead the bitonic approach is due to extra unnecessary comparisons that are made during the merging step. The bitonic merging algorithm does the same number of comparisons as the bitonic sorting algorithm, ignoring the fact in many cases that the inputs are presorted.

This algorithm also does not scale well for larger SIMD widths because more merge network levels are needed and extra overhead is added. For example, a 16-way network (16x2 input elements) is actually faster than a 32-way (32x2 input elements) network when working with 512-bit SIMD [22]. Table III demonstrates how bitonic scales for wider width on different instruction sets. Notice how on larger networks, the instruction count becomes huge compared to the initial count. This is because there is finite overhead added with every new

	Bitonic		Our Algortithm
Width	SSE	AVX-512 [22]	AVX-512
4	25	17	10
8	50	23	10
16	100	29	10

TABLE III

NUMBER OF VECTOR INSTRUCTIONS NEEDED IN THE MAIN LOOP FOR DIFFERENT MERGE IMPLEMENTATIONS USING INTEL'S INTRINSICS AND 32 BIT DATA. VALUES ARE ESTIMATES GIVEN BASED ON OUR IMPLEMENTATIONS; ACTUAL COUNT MAY VARY BY IMPLEMENTATION.

level. In contrast, our algorithm uses the same number of instructions for different sized levels.

B. Other SIMD Algorithms

Another approach to sorting using SIMD is using the AA-Sort [5] algorithm which is similar to Comb sort and Merge sort [9], [24]. This algorithm, named after the Aligned-Access sort, presents a novel way to perform a Comb sort while eliminating unaligned memory access. The algorithm also exploits SIMD using an odd-even sort network. It performs two passes, one performing a local Comb sort, and the other using a merge. The merging step uses a similar algorithm to the Intel SSE bitonic merge [7]. The out-of-core version has a speedup of about 3.33x which makes it almost identical to the bitonic network SIMD sort [7] when comparing only speedup. It is also worth noting that this algorithm surpassed GPUTeraSort [25] by 3.3x when sorting 32 M random 32-bit integers [5]. [10] presents a modified quick sort variant using 512bit SIMD on the KNL (Intel Knights Landing) processor. [11] also presents an a AVX-512 based sort.

[19], shows a sorted-set SIMD intersection; however, this implementation is restricted to 8 bit keys and the Intel SSE instruction set[6].

C. Other Sorting Algorithms

Radix sort is a non-comparison based sorting algorithm. Radix sort suffers greatly from high memory bandwidth because the elements must be transferred in and out of buckets and have multiple passes for each array element [9]. It also has irregular memory access patterns resulting in poor cache utilization and even more memory bandwidth inefficiency [9]. This means for larger data sets, any performance gain from radix sorts is quickly dissipated despite radix sort's preferable computational complexity. A hybrid radix sorts was proposed in [26] which alleviates this overhead

D. Branch-Avoiding Merge

Sorting algorithms are data dependent algorithms by nature and often have irregular branching patterns at runtime. This creates a challenge for CPU branch predictors to efficiently work as intended when running these heavily data dependent algorithms. Branch misses can cost tens of cycles [27] so therefore, the algorithm presented in this paper will only have a single branch (estimated) per loop iteration and will not suffer largely from branch misses.

Algorithm 1: Basic merge

```
function Basic Merge (A, B)
Input  : Two sorted sub arrays: A and B
Output : Sorted array C
AIndex ← 0; BIndex ← 0; CIndex ← 0;
while AIndex < |A| and BIndex < |B| do
    if A[AIndex] < B[BIndex] then
        C[CIndex++] ← A[AIndex++]
    else
        C[CIndex++] ← B[BIndex++]
// Copy remaining elements into C
```

Algorithm 2: Branch Avoiding Merge - assuming 32-bit data

```
function Branch Avoiding Merge (A, B)
Input  : Two sorted sub arrays: A and B
Output : Sorted array C
AIndex ← 0; BIndex ← 0; CIndex ← 0;
while AIndex < |A| and BIndex < |B| do
    flag ← Right_Shift(A[AIndex] − B[BIndex], 31);
    C[CIndex++] ←
      (flag) * A[AIndex] + (1 − flag) * B[BIndex];
    AIndex ← AIndex + flag;
    BIndex ← BIndex + (1 − flag);
// Copy remaining elements into C
```

A traditional merging algorithm looks like the one seen in Algorithm 1. There is a check for bounds, then a simple if else structure. Once the bounds on one subarray have elapsed, the remaining elements are copied. At a high level and ignoring the while loop there are two branch possibilities in the code, either the code block under the if statement is executed or the code block under the else statement is executed. The branching behavior in this case is purely dependent on the data itself. This therefore poses issues for branch predictors, because the predictions will not be based on the data that is causing the branching behavior. The while loop is not affected because it will almost always be taken by the branch predictor.

[28] presents a merging algorithm which uses a flag to effectively mask elements when merging and incrementing the pointers. Performance evaluations show this algorithm general is slightly slower for less random data or data with few unique keys; however, for more random data with more unique keys, this algorithm performs more favorably than a traditional merge.

Even with the branchless merge, some of the SIMD merges presented earlier outperform the branchless version. SIMD merges including the bitonic algorithm have limited branches; however, the algorithm has some conditional jumps in the last steps of the network [28]. The branchless merge algorithm can be extended to use SIMD

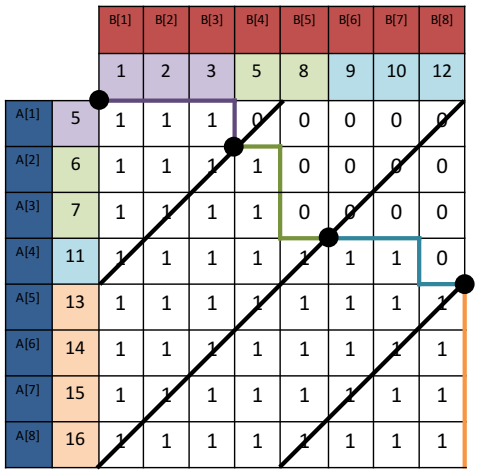

Fig. 2. Merge Path matrix showing intersection lines and points

as seen in the later algorithms of this paper.

III. Merge

In this section we present our research into SIMD based merging algorithms. First we present Merge Path [29], [6], which provides a way of parallelizing the merging of two sorted sub-arrays. We then show how this algorithm can be used to produce a SIMD merge. Lastly, we present two algorithms that extend Algorithm 2 and optimize our SIMD merging when the number of sub-arrays is greater than the number of SIMD threads.

A. Merge Path

Merge sorting starts from small arrays and gradually merges up. When the number of arrays to merge is well greater than the number of threads, it is easy to do these merges in parallel. But at the point where the number of merges is less than the number of threads, the merges must be split into smaller chunks to continue to utilize all threads. The Merge Path algorithm [6] intelligently splits up these chunks by finding merge splitters to split two sorted arrays into smaller sub-arrays for use in merging.

Merge Path is similar to [30], but more intuitive and offers a visual explanation. The following is a high-level overview of Merge Path. The reader is referred to [6] and [29] for additional details. Given two sorted arrays, Place one array horizontal and one vertical so that a grid forms in between the two arrays as seen in Figure 2. We draw lines across the arrays, selecting evenly spaced elements from one array. We then binary search on the opposing array to find the closest split point to the point in the first array. Using these split points, the arrays can be merged separately into one final sorted array.

1) Merge Path SIMD Merge: Here we present a merge algorithm that utilizes Merge Path and therefore is able to fully utilize the hardware at all stages of the merge. Algorithm 3 presents the outline of our approach. Recall that we are merging sub-arrays from A and sub-arrays from B together pairwise.

Algorithm 3: Merge Path Based SIMD Merge

$AIndex, BIndex, AStop, BStop \leftarrow MergePath;$
$CIndex \leftarrow AIndex + BIndex;$
$AEndMask, BEndMask \leftarrow 0xF;$
while $BitOr\_16(AEndMask, BEndMask) \neq 0$ **do**
  ▷ Pull the elements from memory
  $AElems \leftarrow Gather\_16(A, AIndex);$
  $BElems \leftarrow Gather\_16(B, BIndex);$
  ▷ Compare the elements
  $cmp \leftarrow CompareLT\_16(AElems, BElems);$
  $cmp \leftarrow \neg BitOr\_16(BEndMask, (BitAnd\_16(cmp, AEndMask));$
  $CElems \leftarrow Blend\_16(AElems, BElems, cmp);$
  ▷ Store output to memory
  $Scatter\_16(C, CElems, CIndex, BitOr\_16(AEndMask, BEndMask));$
  ▷ Increment Indicies
  $AIndex \leftarrow MaskAdd\_16(AIndex, BitAnd\_16(cmp, AEndMask), 1);$
  $BIndex \leftarrow MaskAdd\_16(BIndex, BitAnd\_16(cmp, BEndMask), 1);$
  $AEndVector \leftarrow CompareGT\_16(AStop, AIndex);$
  $BEndVector \leftarrow CompareGT\_16(BStop, BIndex);$
  $CIndex \leftarrow MaskAdd\_16(CIndex, BitOr\_16(AEndVector, BEndVector), 1);$

First we begin by running the merge path algorithm on the input arrays. The thread count in this case will be the SIMD width. Each sub-array outputs from Merge Path are merged in each SIMD thread. Now that we have the sub-arrays determined we can loop through a simple process. This involves pulling the elements from memory, comparing them, storing them, and then incrementing indices.

Pulling the elements from memory involves the use of the gather instruction. This is a non-sequential memory access. This coupled with the scatter is probably the two big performance hits of this algorithm. We use index variables into both A and B to select which indices to load from A and B into output vectors.

Next we do a less than comparison between the A vector and the B vector. If one of the merge sectors has elapsed for one array, then we want to select the item from the array that has not elapsed. This is done using bit operations and comparing the index vectors with a vector that stores the max index for each sub-array. Once this comparison has been made, we can write the output elements to memory non-sequentially using scatter. We use a masked scatter so that if one of the SIMD threads has finished, we do not continue writing in that thread.

Lastly, we increase the indices for each sub-array who's element was chosen as the smallest. This loop continues as long as at least one of the sub-arrays still had elements left.

While we believe this algorithm to be very efficient in later merge stages, if the number of merges to be done exceeds the number of threads available, there is no need for Merge Path. This occurs when the array size is equal to the total array size divided by the total number of threads, both SIMD threads and processor threads. In

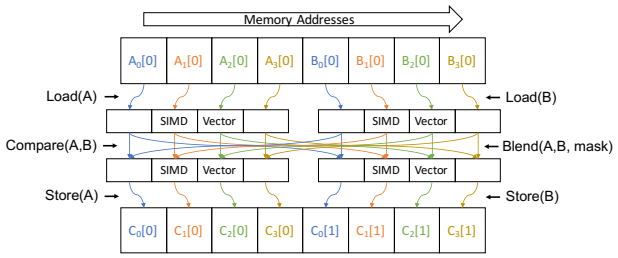

Fig. 3. SIMD merge using sequential loads and stores. Shown here for merging unsorted data up too sub-arrays of size 2 using 4 wide SIMD. Note C is not sorted.

other words, $\frac{n}{(w*p)}$. This is the impetus behind the next two merging algorithms.

2) SIMD Merge Using Sequential Loads and Stores: This merge algorithm we use looks rather similar to a regular merge sort. The distinction of this algorithm is sequential loads and stores coupled with non-sequentially stored sub-arrays. Take a look at figure 3. For the first round we start with an unsorted array. Using a sequential load, two SIMD vectors are loaded into memory. Each vector's elements are compared pairwise with the other vector's elements. The smaller elements are stored sequentially first, followed by the larger elements. This produces sub-arrays of size 2 stored in a non-sequential fashion where each sub-array is stored with its elements offset from each other by the SIMD width.

This same idea is used to merge all the way. Except that at each iteration, more vectors are loaded in at a time. These elements are compared and then written in the same sequential fashion as before, except that only 1 SIMD width of elements are written at a time, and then the elements are pulled down from the next indici in the sub-array and then compared and written again. Code for this can be seen in Algorithm 4.

The advantage of this merge algorithm is the sequential loads and stores, these are much more efficient than the scatters and gathers used in the other algorithms. The downsides of this algorithm are the high overhead of having to load many of the vectors every round even when their elements are not needed yet. Secondly, this algorithm suffers from the disadvantage that at sub-arrays of size $\frac{n}{(w*p)}$ and above it can no longer perform at full system utilization. This is why we still need the first Merge Path based SIMD algorithm.

3) SIMD Merge Using Non-Sequential Loads and Stores: This merge algorithm is similar to the previous except it uses non-sequential loads and stores coupled with sequentially stored sub-arrays. Figure 4 shows this approach for merging sub-arrays of size 1 up to sub arrays of size 2. Pseudocode for this algorithm is seen in figure 5. Since we use gather and scatter, there is no need to store more than two value vectors at a time; However, this approach utilizes index vectors to track where in A,B, and C the algorithm currently is. The indexes are used to select the correct elements in the gather instruction and too store in the correct location using the scatter instruction.

Algorithm 4: Single core SIMD merge using sequential loads and stores. This is essentially a vectorized version of algorithm 2

```
function Sort (array)
Input  : Partially sorted array: array
Output : Partially sorted array: C
for subArraySize ← 1; subArraySize <
 n/16; subArraySize∗ = 2 do
    for
     offset ← 0; offset < n; offset+ = subArraySize ∗ 32
     do
        for index ← 0; index < subArraySize; index + + do
            AElems[index] ←
              Load_16(array + index ∗ 16 + offset);
            BElems[index] ← Load_16(array + index ∗ 16 +
              subArraySize + offset);
        ACount, BCount ← 0;
        for index ← 0; index < subArraySize ∗ 2; index + +
         do
            cmp ← CompareLT(AElems[0], BElems[0]);
            cmp ←
              BitAnd_16(cmp, compareLT(ACount, subArraySize));
            cmp ←
              BitOr_16(cmp, compareLT(BCount, subArraySize));
            CElems ←
              Blend_16(BElems[0], AElems[0], cmp);
            Store_16(C + index ∗ 16 + offset, CElems);
            ACount ← MaskAdd_16(ACount, cmp, 1);
            BCount ← MaskAdd_16(BCount, ¬cmp, 1);
            for
              index ← 0; index < subArraySize−1; index++
              do
                AElems[index] ←
                  Blend_16(AElems[index], AElems[index +
                  1], cmp);
                BElems[index] ←
                  Blend_16(BElems[index +
                  1], BElems[index], cmp);
    // Swap pointers for C and array
// Finish merging with another algorithm
```

The advantage of this merge algorithm is the low storage overhead since only the elements that need to be compared are loaded from memory each round. The downside of this is the overhead of using the non-sequential gather and scatter SIMD instructions. Secondly, this algorithm suffers from the same disadvantage as the previous where at sub-arrays of size $\frac{n}{(w*p)}$ and above it can no longer perform at full system utilization.

IV. Sort

Thus far, we have presented 3 different approaches to merging using SIMD. Now we will present two approaches to utilizing the previous merge algorithms to produce a full sorting algorithm.

1) Iterative Merge Sort: The heart of our sorting algorithm is a loop that repeatedly merges one merge at a time. Based off our own initial testing we determined it was more efficient to qsort chunks first before beginning the merge phase. We begin by partitioning the given array into chunks of size 64 and then quick sorting

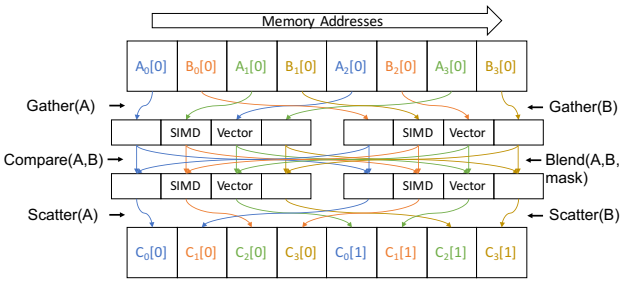

Fig. 4. SIMD merge using gather and scatter. Shown here for merging sub-arrays of size 1 into sub-arrays of size 2 using 4 wide SIMD. Note C is not sorted.

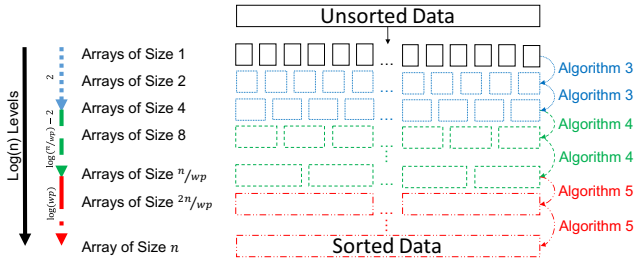

Fig. 5. Optimized Merge Sort using three different merging algorithms

each chunk. We then loop through merging two chunks at a time until a final sorted array is reached. Our implementation is designed to easily switch out different merge algorithms for easy comparison. This approach is seen in Figure 6.

This approach can easily be extended to a parallel merge as well. Instead of using chunks of size 64, the chunks are evenly split based on the array size and the number of threads. Each thread then does its own single core sort on this chunk. Next we merge the chunks in parallel. Since there are less merges to be done then there are threads, Merge Path is used to split each merge into the number of threads that are assigned to that merge. This process is continued until one sorted array remains.

2) Optimized Merge Sort: Our initial testing showed that the initial sorting phase for single core sorting was over 50% of the total sort time. Since this was just a quicksort there was no performance gain during this step. Meaning, we were missing out on a lot of performance gain. It is for this reason that we decided to change our approach with the more optimized version. We start from unsorted data and using the two-non merge path SIMD algorithms from earlier to merge up until we can no longer continue. At this point, the Merge Path SIMD merge is used. This pattern is seen in Figure 5.

V. Experimental Setup

Our system runs an Intel Xeon Phi 7210 with 128GB of memory. This processor has 64 cores and 256 threads. There is 16GB of MCDRAM which we configured as a separate numa node and used as our primary memory for the application.

Algorithm 5: SIMD merge using gather and scatter. This is a cross between Algorithms 4 and 3

```
function Sort (array)
Input  : Partially sorted array: array
Output : Partially sorted array: C
x ← 0;
for index ← 0; index < 16; index++ do
    AIndexStore ← Set(x++, index);
    AStopStore ← Set(x, index);
    BIndexStore ← Set(x++, index);
    BStopStore ← Set(x, index);
for subArraySize ← 1; subArraySize <
 n/16; subArraySize* = 2 do
    for
     offset ← 0; offset < n; offset+ = subArraySize * 32
     do
        AIndex ← AIndexStore;
        AStop ← AStopStore;
        BIndex ← BIndexStore;
        BStop ← BStopStore;
        CIndex ← 0;
        for Repeat subArraySize*2 times do
            maskA = CompareLT(AIndex, AStop);
            maskB = CompareLT(BIndex, BStop);
            AElems ← Gather_16(array + offset, AIndex);
            BElems ← Gather_16(array + offset, BIndex);
            cmp ← CompareLT_16(AElems, BElems);
            cmp ← BitAnd_16(cmp, maskA);
            cmp ← BitOr_16(cmp, maskB);
            Celems ← Blend_16(Aelems, Belems, cmp);
            Scatter_16(C + offset, CElems, CIndex);
            Aindex ← MaskAdd_16(Aindex, cmp, 1);
            BIndex ← MaskAdd_16(BIndex, ¬cmp, 1);
            CIndex ← Add_16(CIndex, 1);
    AIndexStore ← LeftShift_16(AIndexStore, 1);
    AStopStore ← LeftShift_16(AStopStore, 1);
    BIndexStore ← LeftShift_16(BIndexStore, 1);
    BStopStore ← LeftShift_16(BStopStore, 1);
    // Swap pointers for C and array
// Finish merging with another algorithm
```

VI. Results

Figure 6 shows merging results of merging two evenly sized sub arrays into one. Merging is done using one thread with the array size held at 1M. The Basic Merge merged at 100M elements per second at a maximal number of keys of 2^4. When the maximal number of keys was increased up too 2^{28}, the Basic Merge merged 97M elements per second. The AVX-512 Merge Path Based Merge merged 286M and 288M at the same maximal number of keys respectively.

Figure 7 shows merging results of merging two evenly sized sub arrays into one. Merging is done using one thread with the maximal number of keys held at a constant 2^{28}. The Basic Merge merged at 26M elements per second at an array size of 1,000. When the array size increased to 100M, the Basic Merge merged 97M elements per second. The AVX-512 Merge Path Based Merge merged 60M and 286M at the same array sizes respectively.

Figure 8 shows sorting results. Sorting is done using one thread with the array size held at 2^{24}. Merge Sort Standard sorted 2.2M elements per second at a maximal

Algorithm 6: Single core iterative merge sorting algorithm

```
function Sort (array)
Input  : Unsorted: array, Merge Algorithm: merge
Output : Sorted array: C
for offset ← 0; offset < n; offset+ = 64 do
    qsort(array + offset, 64);
for
 subArraySize ← 64; subArraySize < n; subArraySize* = 2
 do
    for
     offset ←= 0; offset < n; offset+ = subArraySize * 2
     do
        Merge(array + offset, subArraySize, array +
          offset + subArraySize, subArraySize, C +
          offset, subArraySize * 2);
    // Swap pointers for C and array
```

Algorithm	Description
Standard	Implementation of Algorithm 1.
Bitonic	Implementation of [7] using SSE instructions.
AVX-512 MP	Implementation of Algorithm 3

TABLE IV
A DESCRIPTION OF THE MERGE ALGORITHM IMPLEMENTATIONS USED IN THESE RESULTS

number of keys of 2^4. When the maximal number of keys was increased up too 2^{28}, Merge Sort Standard sorted 2.1M elements per second. The AVX-512 Merge Path Based Sort sorted 2.5M and 2.4M at the same maximal number of keys respectively. The AVX-512 Hybrid Merge Sort sorted 6M and 5.3M at the same maximal number of keys respectively.

Figure 9 shows sorting results. Sorting is done using one thread with the maximal number of keys of 2^{28}. Merge Sort Standard sorted 2.5M elements per second at an array size of 2^{18}. When the array size was increased up too 2^{24}, Merge Sort Standard sorted 2.1M elements per second. The AVX-512 Merge Path Based Sort sorted 2.9M and 2.4M at the same maximal number of keys respectively. The AVX-512 Hybrid Merge Sort sorted 9.2M and 5.3M at the same maximal number of keys respectively.

Figure 10 shows parallel sorting results. Sorting is done using 256 threads with the array size held at 2^{24}. Merge Sort Standard sorted 218M elements per second at a maximal number of keys of 2^4. When the maximal number of keys was increased up too 2^{28}, Merge Sort Standard sorted 204M elements per second. The AVX-512 Hybrid Merge Sort sorted 547M and 511M at the same maximal number of keys respectively.

Figure 11 shows parallel sorting results. Sorting is done using 256 threads with the maximal number of keys held at 2^{28}. Merge Sort Standard sorted 58M elements per second at an array size of 2^{18}. When the array size was increased up too 2^{24}, Merge Sort Standard sorted

Algorithm	Description
Standard	Iterative merge sort using Standard merge
Bitonic	Iterative merge sort using Bitonic merge
AVX-512 MP	Iterative merge sort using AVX-512 MP merge
AVX-512 Optimized	Uses SIMD Merge with sequential loads and stores (Algorithm 4) to merge from unsorted data up too sub-arrays of size 4. Then SIMD Merge with sequential loads (Algorithm 5) is used to merge up too sub-arrays of size $\frac{n}{16}$. Lastly the remaining sub-arrays are merged using AVX-512 MP until they are fully sorted.
IPP	SortAscend from the Intel IPP library
Quick Sort	This is qsort from the compiler.

TABLE V
A DESCRIPTION OF THE SORTING ALGORITHM IMPLEMENTATIONS USED IN THESE RESULTS

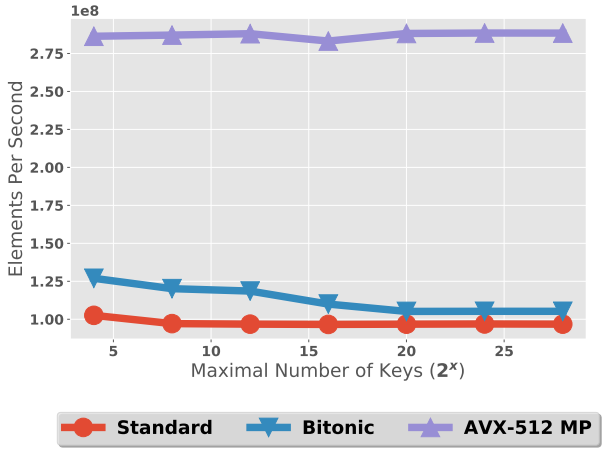

Fig. 6. Sequential Merging results measured in elements merged per second vs the maximal number of keys. Array size was held at a constant 1M

204M elements per second. The AVX-512 Hybrid Merge Sort sorted 57M and 511M at the same array sizes respectively.

VII. Discussion

When comparing the merge algorithms, both the Basic Merge and the SSE Bitonic Merge reduce in elements per second as the maximal number of keys increases. For the AVX-512 Merge Path Based Merge, this is not the case, it remains about the same. When comparing the results based on array size, all the algorithms start about the same. However, after reaching an array size of 100,000, the AVX-512 Merge Path Based Merge surpasses all other merges and remains at a fairly consistent 2.94x faster than the others.

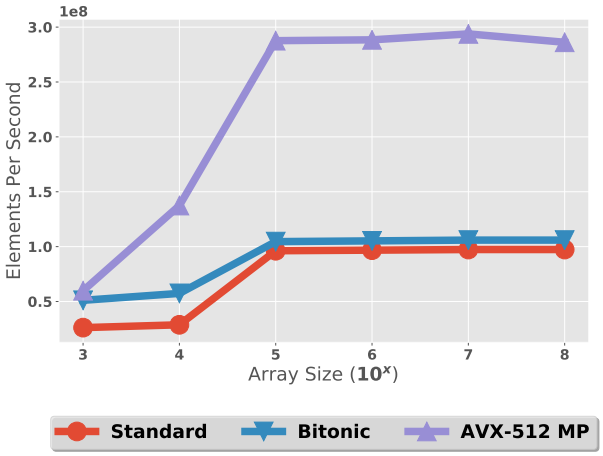

Fig. 7. Sequential Merging results measured in elements merged per second vs the array size. The maximal number of keys held at a constant 2^{28}

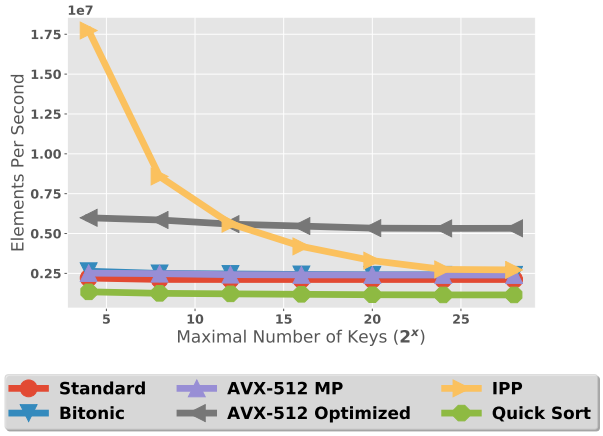

Fig. 8. Sorting results measured in elements sorted per second vs the maximal number of keys. Array size was held at a constant 2^{24}

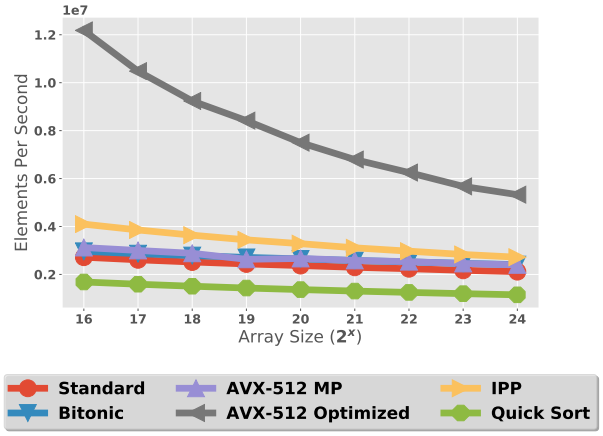

Fig. 9. Sorting results measured in elements sorted per second vs the array size. Maximal number of keys was held at a constant 2^{28}

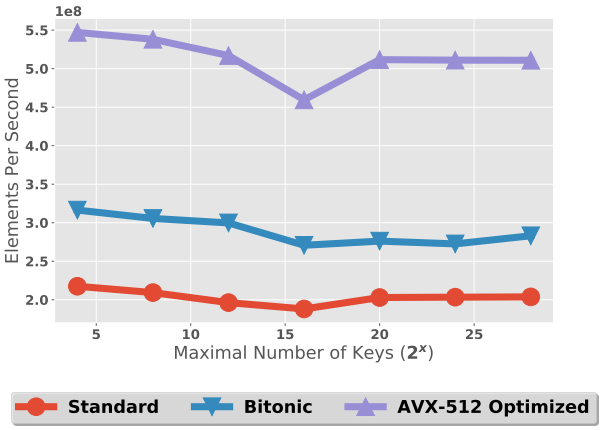

Fig. 10. Parallel Sorting results for 256 threads measured in elements merged per second vs the maximal number of keys. Array size is 2^{24}

For the sorting algorithms, the maximal number of keys effects all the algorithms by reducing their elements per second as it increases. Most of the algorithms stay rather constant despite array size, however, the AVX-512 Hybrid Merge Sort does not. This is likely because as the array size increases, less percentage of the total time is spent in the non Merge Path SIMD merge phase. This same algorithm starts with a 3.64 times speedup over the Merge Sort Standard and dwindles to 2.51x for the highest array size tested.

Parallel results are rather stagnant for maximal number of keys. The elements per second increased for all algorithms as the array size increases. Finally, all algorithms increased in elements per second as the number of threads increase; however, AVX-512 Hybrid Merge Sort increases drastically at 64 threads, and tops out at a 2.5x speedup of the Merge Sort Standard for 256 threads.

VIII. Conclusions

This paper has presented a new way of approaching SIMD based merge sorting. Utilization of Merge Path and branchless merging provides for a clean efficient merge algorithm. Our AVX-512 based algorithm outperforms the other merges presented here, notably the SSE based bitonic merge. This is thanks to the low overhead and efficiency of the algorithm. The results demonstrate that the algorithm is not highly affected by the the number of keys in the data and it also is consistently the fastest even for the smaller array sizes tested on. This algorithm scales well for larger widths as SIMD width increases.

IX. Future Work

One thing that could be extended is the AVX-512 Hybrid Merge Sort being used. Ideally this algorithm would use the same approach in the first few steps, but use our merge path based merge in the higher steps. In addition, some of the lower steps of this algorithm could be improved by using a non-sequential gather but then using a sequential store. This would remove the overhead of storing all the values when using sequential instructions but would still eliminate the need for non-sequential stores. Lastly, there are many optimizations

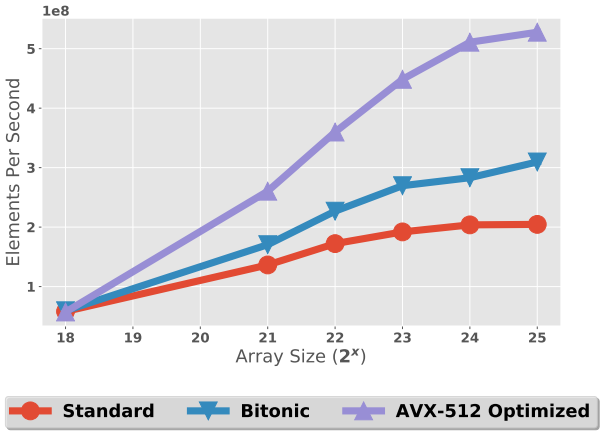

Fig. 11. Parallel sorting results for 256 threads measured in elements merged per second vs the array size. Maximal number of keys was held at a constant 2^{28}

available in the Intel intrinsics not taken advantage of in our implementations.

References

[1] W. A. Martin, "Sorting," *ACM Comput. Surv.*, vol. 3, no. 4, pp. 147–174, 1971.

[2] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, p. 321, 1961.

[3] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, Conference Paper, pp. 307–314.

[4] N. Amato, R. Iyer, S. Sundaresan, and Y. Wu, "A comparison of parallel sorting algorithms on different architectures," Texas A & M University, Report, 1998.

[5] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "Aa-sort: A new parallel sorting algorithm for multi-core simd processors," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, 2007, Conference Proceedings, pp. 189–198.

[6] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, "Merge path - parallel merging made simple," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, Conference Proceedings, pp. 1611–1618.

[7] 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.*, vol. 1, no. 2, pp. 1313–1324, 2008.

[8] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, "A high-performance sorting algorithm for multicore single-instruction multiple-data processors," *Software: Practice and Experience*, vol. 42, no. 6, pp. 753–777, 2012.

[9] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, Conference Paper, pp. 351–362.

[10] B. Bramas, "Fast sorting algorithms using AVX-512 on intel knights landing," *CoRR*, vol. abs/1704.08579, 2017.

[11] H. Inoue and K. Taura, "Simd- and cache-friendly algorithm for sorting an array of structures," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1274–1285, 2015.

[12] T. Furtak, J. N. Amaral, and R. Niewiadomski, "Using simd registers and instructions to enable instruction-level parallelism in sorting algorithms," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, 2007, Conference Paper, pp. 348–357.

[13] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking simd vectorization for in-memory databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, Conference Paper, pp. 1493–1508.

[14] T. Xiaochen, K. Rocki, and R. Suda, "Register level sort algorithm on multi-core simd processors," in *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, 2013, Conference Paper, pp. 1–8.

[15] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. D. Blas, and P. Dubey, "Sort vs. hash revisited: fast join implementation on modern multi-core cpus," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1378–1389, 2009.

[16] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: sort vs. hash revisited," *Proc. VLDB Endow.*, vol. 7, no. 1, pp. 85–96, 2013.

[17] B. Chandramouli and J. Goldstein, "Patience is a virtue: revisiting merge and sort on modern processors," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, Conference Paper, pp. 731–742.

[18] H. Inoue, M. Ohara, and K. Taura, "Faster set intersection with simd instructions by reducing branch mispredictions," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 293–304, 2014.

[19] B. Schlegel, T. Willhalm, and W. Lehner, "Fast sorted-set intersection using simd instructions," in *ADMS@ VLDB*, 2011, Conference Proceedings, pp. 1–8.

[20] S. G. Akl and N. Santoro, "Optimal parallel merging and sorting without memory conflicts," *IEEE Transactions on Computers*, vol. C-36, no. 11, pp. 1367–1369, 1987.

[21] S. Matvienko, N. Alemasov, and E. Fomin, "Interaction sorting method for molecular dynamics on multi-core simd cpu architecture," *Journal Of Bioinformatics And Computational Biology*, vol. 13, no. 1, pp. 1 540 004–1 540 004, 2015.

[22] Y. Liu, T. Pan, O. Green, and S. Aluru, "Parallelized kendall's tau coefficient computation via simd vectorized sorting on many-integrated-core processors," *arXiv:1704.03767*, 2017.

[23] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on fpgas," *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, feb 2012.

[24] S. Lacey and R. Box, "A fast easy sort," *Byte Magazine*, pp. 315–320, April 1991.

[25] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," pp. 325–336, 2006.

[26] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on gpus," *CoRR*, vol. abs/1611.01137, 2016.

[27] I. Pavlov, "7-zip lzma benchmark," 2017. [Online]. Available: http://7-cpu.com/

[28] O. Green, "When merging and branch predictors collide," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. IEEE Press, 2014, Conference Proceedings, pp. 33–40.

[29] O. Green, R. McColl, and D. A. Bader, "Gpu merge path: a gpu merging algorithm," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, Conference Proceedings, pp. 331–340.

[30] N. Deo and D. Sarkar, "Parallel algorithms for merging and sorting," *Information Sciences*, vol. 56, no. 1, pp. 151 – 161, 1991.