Register Level Sort Algorithm on Multi-Core SIMD Processors

Tian Xiaochen, Kamil Rocki and Reiji Suda
Graduate School of Information Science and Technology
The University of Tokyo & CREST, JST
{xchen, kamil.rocki, reiji}@is.s.u-tokyo.ac.jp

ABSTRACT

State-of-the-art hardware increasingly utilizes SIMD parallelism, where multiple processing elements execute the same instruction on multiple data points simultaneously. However, irregular and data intensive algorithms are not well suited for such architectures. Due to their importance, it is crucial to obtain efficient implementations. One example of such a task is sort, a fundamental problem in computer science. In this paper we analyze distinct memory accessing models and propose two methods to employ highly efficient bitonic merge sort using SIMD instructions as register level sort. We achieve nearly 270x speedup (525M integers/s) on a 4M integer set using Xeon Phi coprocessor, where SIMD level parallelism accelerates the algorithm over 3 times. Our method can be applied to any device supporting similar SIMD instructions.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: PROCESSOR ARCHITECTURES—*Single-instruction-stream, multiple-data-stream processors (SIMD)*

General Terms

Algorithms, Performance, Design

Keywords

SIMD, Parallel, Sort, Xeon Phi, Register, Irregular

1. INTRODUCTION

Multi-core architecture has been widely used in state-of-art processors. For example, NVIDIA's Tesla K20 GPU has 14 Stream Multiprocessors(SMX) and Intel's Xeon Phi has 60 cores. Xeon Phi supports 512 bits SIMD instruction i.e. AVX-512 which can process 16 integers (4 bytes) simultaneously. GPU employs a similar architecture: single-instruction-multiple-threads(SIMT). On K20, each SMX has 192 CUDA cores which act as individual threads. Even a desktop type multi-core x86 platform such as i7 Haswell CPU family supports AVX2 instruction set (256 bits). Developing algorithms that support multi-core SIMD architecture is the precondition to unleash the performance of these processors. Without exploiting this kind of parallelism, only a small fraction of computational power can be utilized.

Parallel sort as well as sort in general is fundamental and well studied problem in computer science. Algorithms such as *Bitonic-Merge Sort* or *Odd Even Merge Sort* are widely used in practice. However, implementing sort algorithm on SIMD processors efficiently, especially on Xeon Phi or x86 CPU remains a challenging job. Until now, many algorithms have been proposed for GPU or CPU. GPU version of the algorithm cannot be directly transplanted a processor supporting 512-bit wide vector instructions such as Xeon Phi. Nvidia GPUs have fast, explicitly manageable on-chip shared memory and their cores are more functional and execute individual threads concurrently with SIMT synchronization on hardware level. Programming Xeon Phi poses a different challenge - the instruction restrictions of memory accesses (discussed in Section 3.1) needed during SIMD sort or merge. Previous algorithms designed for SSE or CELL processor require too many registers or not strong scalable enough for long SIMD instructions. On Xeon Phi, processor's resources can easily run out with previous methods. A general, strong scaling algorithm for Xeon Phi or future SIMD instruction capable processors is necessary.

In this paper, we focus on developing parallel sort and merge algorithms only in register, under the constraints of memory accessing model within registers. We propose two in-register sort/merge algorithms which only take a constant number of registers (2 or 3) no matter how long SIMD instruction is. Then, we present theoretical and empirical analysis of the execution time. We chose AVX-512 and Xeon Phi as our main experiment environment as a extreme case of SIMD parallelism combined with restricted access patterns and limited bandwidth. We also implemented the algorithm on Intel E5-2670 with AVX instructions. The same algorithm can be derived for other SIMD processors.

2. RELATED WORK

First algorithm that needs to be mentioned is parallel *Radix sort*[5] as it is often used. The main advantage of the method is that if the size of digits is fixed, it only needs $O(n)$ time complexity, where n is the length of data. The idea is

SC13 November 17-21, 2013, Denver CO, USA

ACM 978-1-4503-2503-5/13/11
http://dx.doi.org/10.1145/2535753.2535762.

to sort data digit by digit. Intel Laboratories' report[2] is the first published result presenting performance of radix sort on Xeon Phi. Later in the paper, we are presenting our results compared to those shown in this paper. Its main disadvantage on the other hand is that it is limited to the integer type of data and exploits its decimal character. Contrary to more general, comparative sort for example, float numbers need extra transformation[13] using radix sort. Furthermore, if the data type size is longer, the algorithm becomes slower. Nadathur Satish et al.[6] studied how radix sort and merge sort performs when size of data type varies. Though both of them get slower when data type changes from 32bit to 128bit, radix sort get higher deceleration.

Comparison based sort is a more general sort algorithm. $\Omega(nlog(n))$ is the lower bound of its time complexity. Many theoretical results have been published in the field of parallel sorting. *Bitonic-Merge Sort*[3] or *Odd-Even Merge Sort*[8] are just two algorithms usually used in practice. They are also sorting networks traversed in $O(log^2(n))$ steps. Though $log(n)$-step algorithm has been invented[9], due to the large overhead it is hardly used in practice. Our algorithm keeps the generality of a comparative sort while achieving speedup only slightly worse than with radix sort.

There is a large number of research papers which describe implementing parallel sort. We classify the algorithms into two types: *Register Level Sort* and *Multi-Core Level Sort*. This classification is made by the way of synchronization in processors. In register level, SIMD instructions play the role of cores. Synchronization is not necessary(GPU may need synchronization explicitly) since SIMD computation is naturally synchronized after each instruction. In the situation of Multi-Core level, communication happens in slow cache or RAM. The cost of synchronization becomes more expensive.

2.1 Register Level Sort

Register level sort means sorting a part of a small amount of data only using SIMD instructions. Usually the data should be loaded into a register from the main memory and then sorted in registers in parallel. Finally the data should be copied back to the main memory. Some implementations may use *Odd-Even Merge Sort* at this point, for example an algorithm using SSE instructions[1] by Jatin Chhugani et al. In their approach, the algorithm requires sorting 4 groups at the same time , each group comprises 4 elements (since SSE registers can hold 4 integer values - 128 bits) at once, using 4 registers at least. Then it transposes these 16 elements so that elements belonging to one group can be laid in the same register. An example is shown in Figure 1. This algorithm works in the following way - let 4 cores do 4 jobs. It is not strong scalable enough and requires a large number of registers in practice. If the length of SIMD instruction is K, then K registers are needed. In the situation of sorting 4 bytes integers on Xeon Phi, it requires 16 registers.

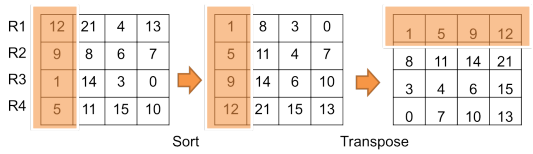

Figure 1: Example of Jatin's register level sort algorithm

Another highly scalable algorithm was proposed by Timothy Furtak[7], it generates the code with search method. The main idea is to search a set of SSE instructions which follow exactly the same logic as *Odd-Even Merge Sort* at each step. The advantage of this method is that the generated code can be locally optimal. However they ignore the usage of the number of registers and the speed of generation. Finally, AA sort[10] is a different approach which does not employ odd-even or any other sorting network. *Comb Sort*[11] is their choice. However, *Comb Sort* needs $O(n^2)$ in worst case, also this algorithm requires the same number of registers as the aforementioned Jatin's algorithm.

We propose two strong scaling register-level-sort algorithms which only need constant number of registers. The two algorithms are both derived from the code generation approach which is based on constructing methods. It can be finished in polynomial time (the same as bitonic merge sort).

2.2 Multi-Core Level Sort

In principle, *Multi-Core Level Sort* focuses on utilizing multiple cores. MIMD (Multiple Instruction, Multiple Data) and PRAM are the appropriate hardware models. On GPU, *Odd-Even Merge Sort* is still a widely chosen algorithm [12, 14]. It has high parallelism, but suffers from high work complexity. Theoretically *Bitonic-Merge Sort* can be performed in $O(nlogn)$ time by using a subtree(i.e. an address pointer) exchanging algorithm[16] utilizing a tree data structure. Contrary to merge, divide-and-conquer is another idea of sorting elements. For example, Bucket Sort, the quick sort's parallel version can divide the data into a numbers of bucket by selecting several pivots. The data in each bucket can be sorted by other sorting algorithms, since the buckets are ordered by pivots. An GPU implementation[15] uses hybrid algorithm of bucket sort and merge sort. However the algorithm may not be well balanced since buckets' sizes depend on the chosen of pivots. Some buckets may be too large or too small. A typical x86-family processor, such as Intel i7 does not have as many core as common GPUs. Multi-way merge is an optional algorithm in such a CPU implementation[1]. Although, multi-way merge can be as efficient as merge sort from time complexity aspect. If there are too many cores, the communication between cores will be the major bottleneck of the algorithm. Since Xeon Phi may have more than 60 cores and *Odd-Even merge* takes higher work complexity, we choose parallel merge sort based on partition as our algorithm.

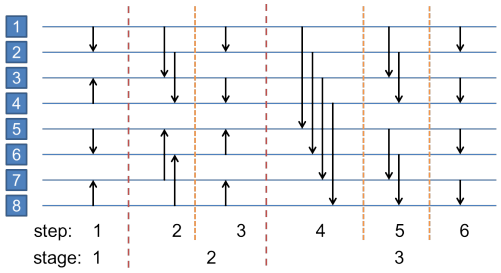

Figure 2: Bitonic-Merge sorting network of 8 element.

3. OUR APPROACH

First, let's define K as the number of elements that one vector instruction can handle and two available registers as $R1$ and $R2$. *Bitonic-Merge Sort* is a often used as a parallel

sort algorithm. Figure 2 shows the sorting network of 8 elements.

The number in the box represents the indices of element. An arrow represents a comparison of a pair of elements, storing the larger element where the arrow is pointing and the smaller element on the opposite side. Figure 3 shows an example of *Bitonic-Merge Sort*.

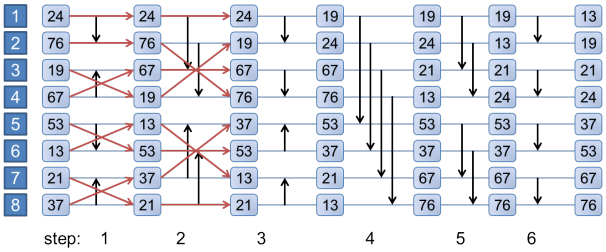

Figure 3: An example of Bitonic-Merge sort.

3.1 AVX-512 Capability of Comparison

To implement Bitonic-Merge Sorting network using recent Intel's vector extensions, it is necessary to ravel out what AVX-512 can do in a sort problem. On GPU, CUDA core seems to more flexible, since one core corresponds to one real thread from the programming point of view (CUDA, OpenCL and etc.). On the other hand, programming using AVX-512 is more like assembling instructions together in a procedural way. From the hardware perspective, GPU has on-chip shared memory which makes CUDA random accesses memory possible. Therefore memory accessing pattern of CUDA core is PRAM. When we consider AVX-512, vector register has to load data from main memory or cache before calculation in a vector-manner, meaning chunks of memory at once. Many operations have to be executed between two registers. So memory access pattern is not PRAM.

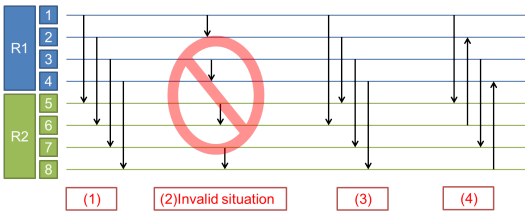

Figure 4: Capability of AVX-512 instruction set

Let's assume that some data is loaded into registers $R1$ and $R2$. Figure 4 shows the capability of AVX-512. Notice that the second operation is invalid, because elements in the same register can not be compared with each other. With permutation operation (e.g. _mm512_shuffle_epi32) and mask operation (e.g. _mm512_mask_min_epi32), it is possible to do comparison like (3) and (4). Corresponding pseudocode is shown in Table 1. We can conclude from the discussion that: A comparison is valid on AVX-512 if each pair of elements be is placed in two separate registers and compare option is applied between them. Our goal is to generate the sorting network like in Figure 2 and at the same time satisfy the memory access constraints. Then, the program can be hard-coded from generated network with AVX-512 instructions. Intuitively the fewer instructions the code has, the faster the program becomes. We propose two methods to generate new sorting network from Bitonic-Merge sorting network.

Table 1: Capability of AVX-512

Situation	Pseudocode
(1)	R1' = SIMD_min(R1,R2); R2' = SIMD_max(R1,R2);
(2)	(invalid)
(3)	R2' = permute(R2,<2,1,3,4>); R1' = SIMD_min(R1,R2); R2' = SIMD_max(R1,R2);
(4)	R1' = SIMD_mask_min(R1,R2,0b1010); R2' = SIMD_mask_min(R1,R2,0b0101); R1' = SIMD_mask_max(R1,R2,0b0101); R2' = SIMD_mask_max(R1,R2,0b1010);

3.2 1-Register Method

The *1-Register Method* does not mean that we are only using one register, but it means sorting such number of elements that can be stored in one register at the same time, i.e. K elements. Let's assume that the data is loaded into $R1$ initially. The target is to get elements which are stored in $R1$ to be compared with each other. The idea is to make a copy of $R1$, for instance copy it to $R2$. In such a way, elements can be compared with each other. The right-hand part of Figure 5 shows the example of 4 elements sorting networking using 1-register method. The left-hand part of the figure represents the 4 elements sorting network of bitonic merge. Obviously, the left-hand side and the right-hand side parts follow the same logic. The difference is *1-Register Method* compares each pair of elements twice. The order of permutation is generated from bitonic merge sorting network.

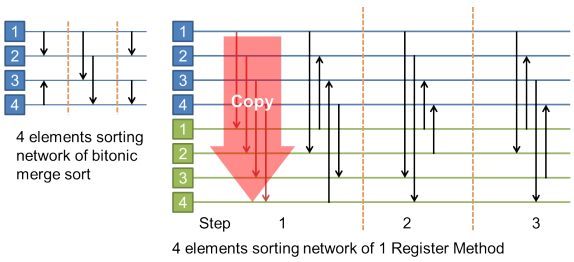

Figure 5: 1-Register Method Sorting Network

Let's define x as the position of element and $f_i(x)$ as the corresponding position should be compared to at ith step. $f_i(x)$ can be calculated from normal bitonic merge sort. Then we have code as in Algorithm 1 using AVX-512 instructions. There are 10 steps in total, and each step contains 3 AVX-512 instructions.

Algorithm 1 1-Register Method Sorting Algorithm

1: $\_m512i\ a, b;$
2: $b = \_mm512\_shuffle\_epi32(a, \_MM\_PERM\_CDAB);$
3: $a = \_mm512\_mask\_min\_epi32(a, 0x6996, a, b);$
4: $a = \_mm512\_mask\_max\_epi32(a, 0x9669, a, b);$
5: ... ▷ 3 instructions in one step. The rest of the steps are similar.

3.3 2-Register Method

One disadvantage of the previous approach is that the elements need to be compared twice. Therefore, we developed the *2-Register Method* to avoid it. The 2-Register Method sorts $2K$ elements at once each time. At first, the data is loaded into registers $R1$ and $R2$. However, this time, bitonic merge sorting network scheme is applied directly. Unfortunately, bitonic merge cannot be simply applied even in the first step (leftmost arrows in Figure 6).

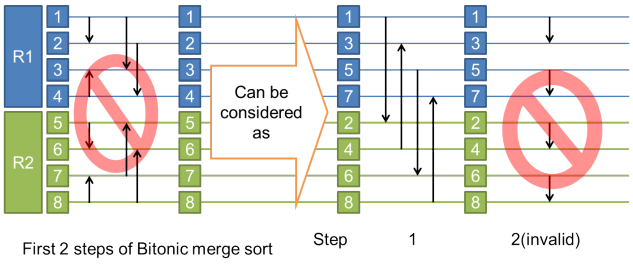

Figure 6: Invalid situation of first 2 steps

Since initially the dataset is unsorted at all, it can be considered as being in any order. In the second step, invalid situation would occur again if the sorting network is applied directly. To solve this problem, the position of the elements should be prepared for the next step so that no invalid situation occurs. The idea is to **look one step ahead**. Before introducing the *one step ahead* approach, it is better to describe how to perform exchanging the elements at the same time with comparing them. Doing comparison and then exchange of a pair of elements is equivalent to doing the comparison in the opposite direction. Figure 7 shows this procedure.

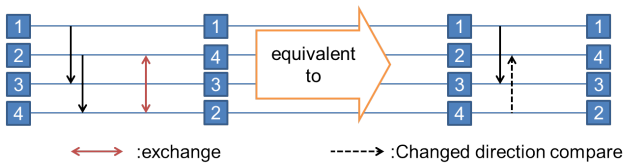

Figure 7: Exchange a pair of two elements

If the following step requires a pair of elements to be compared, and they are in the same register in the current step, they should be exchanged with one another, so that they are in different registers in the next step.

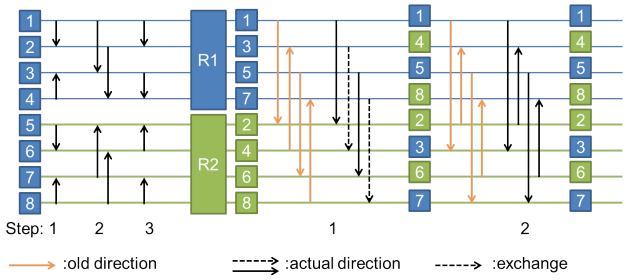

Figure 8: Solution of invalid situation

Figure 8 shows an example how the invalid situation be avoided in the first two steps. The whole sorting network is given in Figure 9. In this figure, a dashed black arrow corresponds to exchanging positions of the elements.

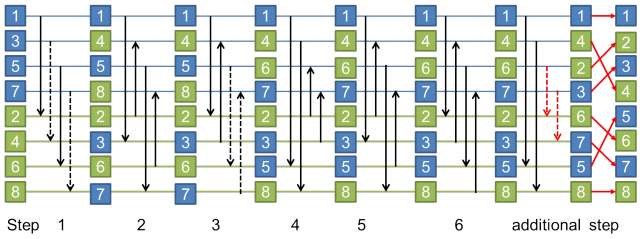

Figure 9: Capability of AVX-512 instruction set

In order to generate the network, not only do we need to know $f_i(x)$ of each step, but also we should decide whether to put x in $R1$, $f_i(x)$ in $R2$ or $f_i(x)$ in $R1$, x in $R2$. Specifically, in ith step, being in position j we should look ahead $i+1$ steps to check whether $f_{i+1}(j)$ is in the same register or not. If it is, it is necessary to exchange j and $f_i(j)$ in the ith step. After applying bitonic merge sort, the elements are ordered as <1 4 6 7 2 3 5 8> finally. Based on the last step of Bitonic-merge, elements 2 and 3, 6 and 7 are in *wrong* registers so to speak, because the final order should be $1 \ldots 8$. It is required to restore the order of the elements, so that they can be written back to memory. To bring elements 2 and 3 back to $R1$, 2 , another comparison is needed as an *additional step* shown in Figure 9 (dashed red arrow). Finally, we permute registers $R1$ and $R2$ (red line in figure) and $2K$ elements are sorted. Algorithm2 shows the pseudocode of the 2-register method and how the sorting network can be generated.

Algorithm 2 Algorithm for Generating 2-Register Method Sorting Network

Initial State: R1 holds elements $1, 3 \ldots 2K-1$. R2 holds $2, 4 \ldots 2K$.
Output: Element order X in R1 at each step, (the corresponding element order in R2 will be $f_i(X)$), and flag array D to record whether changing comparison directions.
for $i = 1$ to *number of steps* **do**
    $X_i \leftarrow R1$
    ▷ Look ahead next step:
    $Y \leftarrow f_{i+1}(X_i)$
    ▷ Find intersection(invalid situation):
    $Z \leftarrow X_i \cap Y$
    ▷ Exchange conflict elements:
    for all x in Z **do**
        ▷ Swap the smaller element(but either is fine):
        $x \leftarrow min(x, f_{i+1}(x))$
        mark $D_i[x]$ *as changing comparison direction*
        ▷ exchange with corresponding element in R2:
        $R1[x] \leftarrow f_i(x)$
    end for
end for

The essential sorting program code is similar to the 1-Register Method, and it is presented as Algorithm 1.

Algorithm 3 2-Register Method

```
1: __m512i a, b, a2;
2: a2 = _mm512_mask_min_epi32(a, 0x6996, a, b);
3: a2 = _mm512_mask_max_epi32(a, 0x9669, a, b);
4: b = _mm512_mask_max_epi32(a, 0x6996, a, b);
5: b = _mm512_mask_min_epi32(a, 0x9669, a, b);
6: b = _mm512_shuffle_epi32(b, _MM_PERM_CDAB);
7: ...    ▷ five instructions consist of one step. The rest of
   the steps are similar.
8: ...                                   ▷ additional step:
9: a = _mm512_min_epi32(a2, b)
10: b = _mm512_max_epi32(a2, b)
11: permute a and b to correct order.
```

Because the procedure is exactly the same as Bitonic-merge sort, the theoretical time complexity has no difference in these two methods. Since 2-Register Method only uses 1 permute instruction per step, it saves 1 instruction in each given step. However, the 2-Register Method does not keep the order of elements in registers, so it needs one more step to rearrange the elements. Table 2 shows a comparison of the two methods regarding instruction count.

Table 2: Comparison of 1-Register and 2-Register methods

Algorithm	1-Register Method	2-Register Method
Set Length	K	$2K$
Number of steps	$\frac{log(K)(log(K)+1)}{2}$	$\frac{log(2K)(log(2K)+1)+2}{2}$
Intructions/step	3	5
Total instructions[a]	86	77

[a] When sorting 32 elements and $K = 16$ (Xeon Phi implementation)

3.4 Register Merge

Merging two sorted sequence is an essential part of the merge sort algorithm. Sequential merge algorithm can be done in $O(n)$ time complexity, where n is defined as the length of data. However, in the parallel case, it is very hard to achieve linear speedups in practice, e.g. merging K elements with K cores in $O(1)$ time. Needless to say efficient merging using SIMD Instructions is even harder. With bitonic merge, $O(Klog(K))$ work complexity, it is possible to merge K elements in $O(log(K))$ steps. The procedure of merging in registers is exactly the same as bitonic merge, which is also the last stage of bitonic merge sort. There are two methods in this case too: 1-Register and 2-Register. Since it is only the last stage of bitonic merge sort, it shares some common code with register-level sort. Input is given a two sorted sequences. One needs to be sorted in descending and the other in ascending order. Applying SIMD parallelism to merge longer sequence (e.g.longer than K) needs a little bit more work. Assume that the two input sequences are $L1$, $L2$ and we would like to merge them into an non-descending sequence $L3$.

Let's recall a naive merge procedure:

1. Pick two elements a, b from $L1$ and $L2$
2. Compare a to b, put the smaller element into $L3$
3. If a is put to $L3$, assign a another element fetched from $L1$, go to step 2 until one sequence is empty. Perform analogous steps for b.

In case of SIMD, $2K$ elements are merged in one step and we require that $L1$ and $L2$ are sorted in different direction. We use the Kth position element to decide from which sequence to fetch another K elements. Algorithm 4 shows this procedure.

Algorithm 4 Merge with SIMD

```
1: Input: Sequence L1 sorted in assending order, L2 in
   decending order.
2: Output: Sequence L3 sorted in assending order
3:          ▷ R_a and R_b are two K length vector registers.
4: R_a ← fetch K elements from the beginning of L1
5: R_b ← fetch K elements from the end of L2
6: repeat
7:     ▷ R_a[K] is the largest in X, R_b[1] is the largest in Y
8:     if R_a[K] ≤ R_b[1] then
9:         SIMD merge R_a and R_b, keep R_b decending
10:        Write R_a back to memory
11:        R_a ← fetch next K elements from L1
12:    else
13:        SIMD merge R_a and R_b, keep R_b assending
14:        Write R_a back to memory
15:        R_a ← fetch next K elements from L2
16:    end if
17: until reach the end of L1 or L2
18: concatenate the rest elements of L1 or L2 to L3
```

Thus all SIMD components of the algorithm are prepared. The rest is to assemble them together.

3.5 Implementation

Figure 10 shows the overview of the whole algorithm. The subroutines are divided according to the size of data. Generally the outer algorithm is merge sort algorithm. Firstly, each core sorts a part of data recursively (the orange color merge sort procedure in Figure 10). When the size of data is small enough, register-level sort is employed. Alternately sorted subsequence should be merged recursively with the algorithm introduced in Section 3.4. When each core finishes sorting its own data, merge procedure becomes multi-core-level merge (cyan color merge sort procedure in Figure 10). This subroutine is in charge of merging two sorted subsequences in parallel using multiple cores. Synchronization between multi-cores is required. Here our algorithm uses Merge Path algorithm[4] to help dividing sequences, so that each core can do merging independently. Merge Path algorithm is a parallel merge algorithm which uses binary search in order to divide two sequences into pairs of subsequences. Then each core can merge such two short subsequence. It is highly balanced algorithm because the partitioning guarantees that the total length of any pair of subsequences are the same. Hence each core process the same mount of data. Merge Path algorithm is used only for partitioning, but it

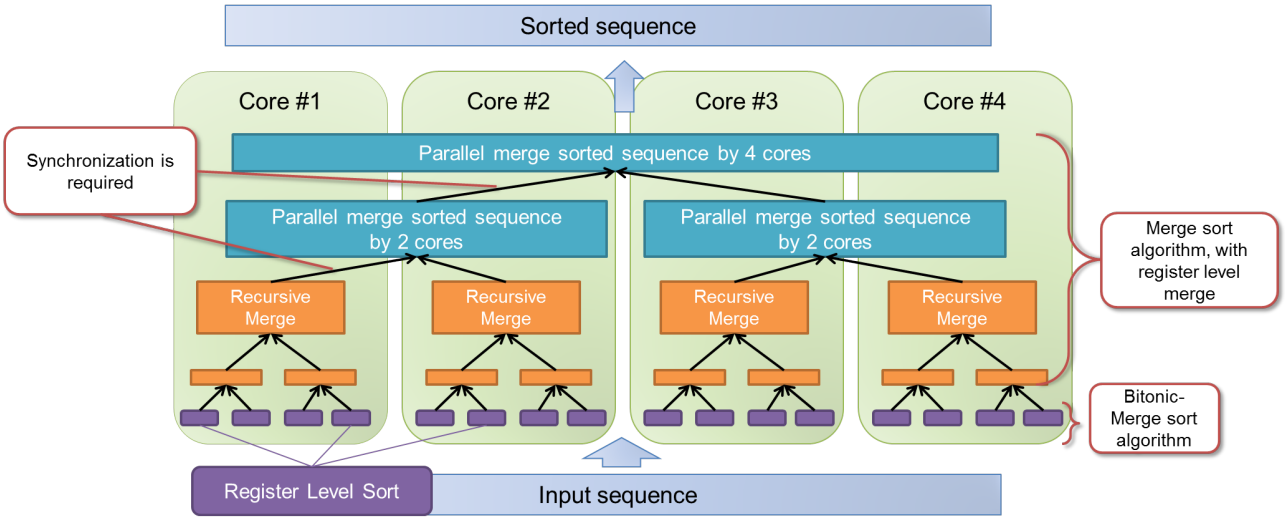

Figure 10: Algorithm Overview

contributes to extra computation which becomes the main overhead of our algorithm. Recursive merging is being done by multiple cores and whole data is finally sorted.

4. PERFORMANCE AND ANALYSIS

We analyse the algorithm by studying the time and work complexity of the subroutine in a single core or multi-core separately. Except for register level sort which uses Bitonic merge sort, other parts are all Merge sort. The work complexity is generally in $O(nlog(n))$. The result are shown in Table 3.

Table 3: Theoretical Analysis

Subroutine/Complexity
Merge sort work complexity of one core $O(\frac{N}{p}log^2(K) + \frac{N}{p}log(K)log(\frac{N}{p*K}))$
Merge sort time complexity of one core $O(\frac{N}{p*K}log^2(K) + \frac{N}{p*K}log(K)log(\frac{N}{p*K}))$
Parallel merge work complexity $O(log(p)(p*log(N) + N*log(K)))$
Parallel merge time complexity $O(log(p)(log(N) + \frac{N}{p*K}log(K)))$
Total work complexity $O(p*log(p)log(N) + N*log(N)log(K) + N*log^2(K))$
Total time complexity $O(log(p)log(N) + \frac{N}{p*K}(log(N)log(K) + log^2(K)))$
Synchronization times $O(log(p))$

In Figure 10, Merge Sort (orange) and Register Sort (purple) are performed within a single core and shown as the first two subroutines in Table 3. Parallel Merge requires synchronization between multiple cores, it is shown as the 3rd and 4th line in table. Work complexity shows how much computation a parallel algorithm does. Theoretical lower bound of a general sorting algorithm based on comparison is $O(Nlog(N))$. Ignoring negligible overhead (the term that does not contain N), our algorithm has $O(Nlog(N)log(K))$ work complexity. $log(K)$ is the consequence of the register level merge using SIMD instructions. However, it gives K times speed up, as shown in time complexity analysis. Implementation and empirical experiments are done on Xeon Phi 5100 series. Data for experiment is generated randomly, distributed uniformly in range from 0 to $2^{31}-1$. Table 4 shows speedup provided by SIMD and multi-core respectively, when sorting 4 million integer elements.

Table 4: Performance of our sort algorithm using Xeon Phi

Configuration	Speed Up	Time(sec)
Sequential merge sort	1.0	2.3
Sequential merge sort w/ SIMD	3.7	0.61
240-thread merge sort w/o SIMD	79	0.029
240-thread merge sort w/ SIMD	291	0.0079

The performance is measured by how much data (how many elements) can be sorted per second conventionally. Since comparison sort takes $O(nlog(n))$ time complexity, the performance will drop with the growth of data length n. Our algorithm achieves the peak performance when the data size is 4 million, which is about 525M integer elements per second. In our opinion, the architecture of Xeon Phi, particularly it's memory bandwidth and cache hierarchy are responsible for this phenomenon. Further experiments and optimization of the algorithm may be needed in order to fully understand the performance peak occurring for a particular number of elements. Figure 11 compares our result with other platforms and algorithms. Compared to the fastest, but limited radix-sort, our algorithm performs almost as well. It is only 13% slower for that particular number of elements with performance slightly dropping as we increased the dataset size. To show the generality of the algorithm,

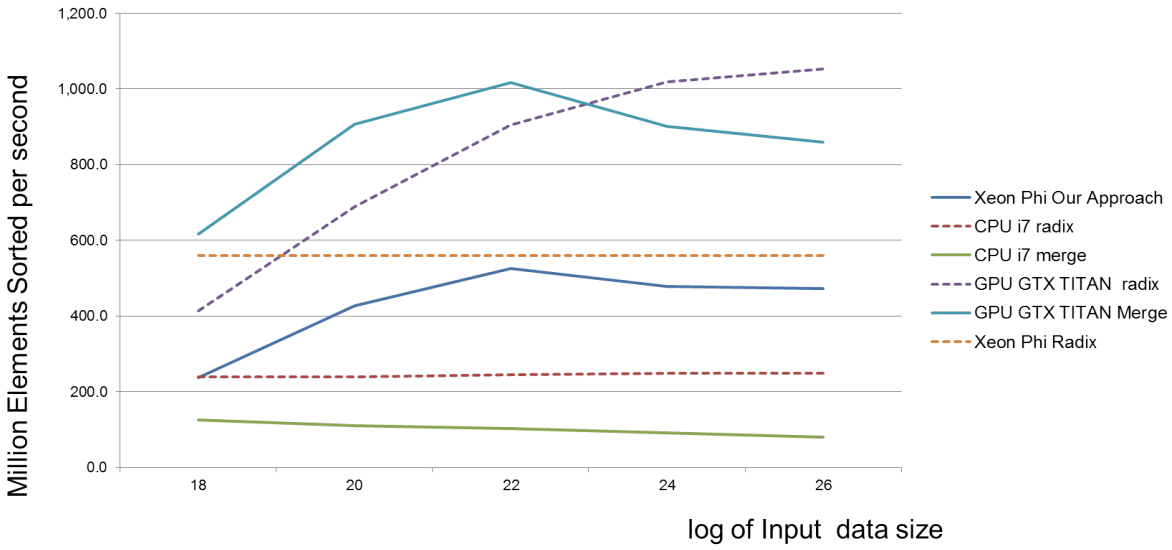

Figure 11: Performance compare to other algorithms and platforms

we implement float data sorting program on Xeon Phi and Intel E5-2670 with AVX, shown in Figure 12. Sorting float takes about 2 times longer than integer on Xeon Phi, due to difference in hardware. It proves the main advantage of our algorithm, meaning its generality regarding the data type (since it is based on comparison sort, as opposed to radix-sort).

5. CONCLUSIONS

This paper explored the issue of parallel sort, a fundamental computer science problem, on a class of SIMD processors. Main contribution is the proposed register level sort combined with merge algorithm. The key idea of our approach is changing the bitonic merge sorting network in order to satisfy the constraint of memory accesses within registers. Our algorithm uses constant number of registers to sort SIMD instruction length data, which exposes strong scalability. Generation of the sorting network has the same time complexity as bitonic merge sort, which can handle long vector instruction situation. Furthermore this algorithm can be generally employed in SSE, AVX or any similar instruction set.

We also implemented the algorithm on Xeon Phi combined with merge sort algorithm. Empirical performance results showed promising speed which is not far from the fastest radix sort implementation, while maintaining the generality of comparison sort. We achieve nearly 270x speedup (525M integers/s) on a 4M integer set using Xeon Phi coprocessor, where SIMD level parallelism accelerates the algorithm over 3 times. Our method can be applied to any device supporting similar SIMD instructions.

6. FUTURE WORK

Our algorithm is able to apply Bitonic-Merge sorting network on SIMD instruction processor like AVX-512. Other types of sorting network (e.g. Odd-Even or even less step algorithm) should also be possible to be adapted to AVX-512. To verify this, more work is required. Usually, not only numbers need to be sorted. Sorting keys with values (e.g. indices or address) is widely used. Efficient implementations may need more work in order to apply SIMD instructions like AVX-512. We would also like to perform experiments using other hardware, such as AVX2 supporting CPUs and run comparative benchmark on many machines using different data types.

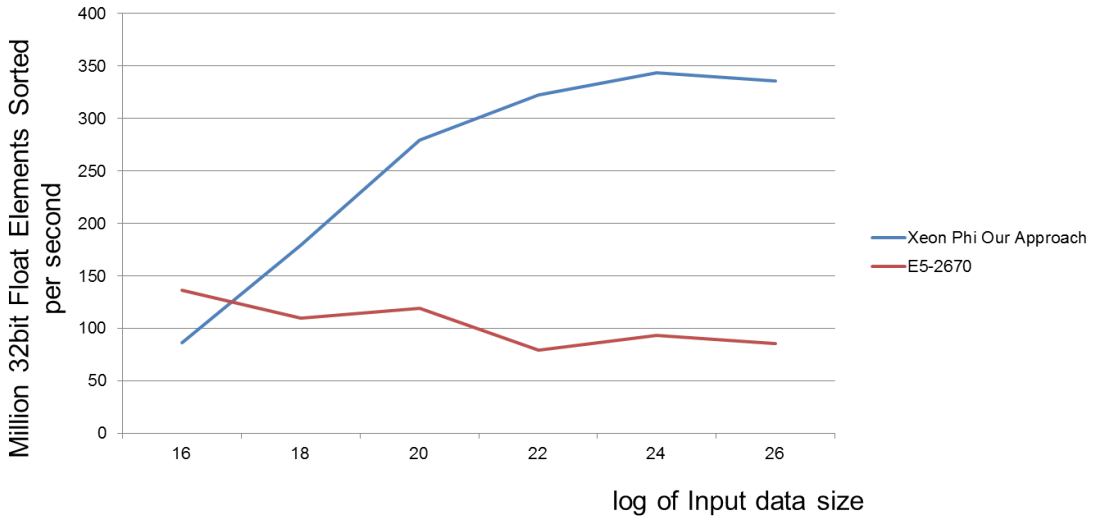

Figure 12: Float elements sorting performance

7. REFERENCES

[1] Jatin Chhugan et al. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture Proceedings of the VLDB Endowment, Volume 1 Issue 2, August 2008, pp. 1313-1324

[2] Nadathur Satish et al. Fast Sort on CPUs, GPUs and Intel MIC Architectures Technical Report, 2010

[3] K. E. Batcher Sorting Network and Their Applications AFIPS '68 (Spring) Proceedings of the April 30–May 2, 1968, spring joint computer pp. 307-314

[4] Saher Odeh et al. Merge Path - Parallel Merging Made Simple Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. pp. 1611 - 1618

[5] Nadathur Satish et al. Designing Efficient Sorting Algorithms for Manycore GPUs Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on Parallel and Distributed Processing

[6] Nadathur Satish et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort SIGMOD '10 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data pp. 351-362

[7] Timothy Furtak, Jose Nelson Amaral and Robert Niewiadomski Using SIMD Registers and Instructions to Enable Instruction-Level Parallelism in Sorting Algorithms SPAA '07 Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures pp. 348-357

[8] D.E. Knuth The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 5.3.4: Networks for Sorting, pp. 219-247

[9] Miklos Ajtai et al. An nlog(n) sorting network STOC '83 Proceedings of the fifteenth annual ACM symposium on Theory of computing pp. 1-9

[10] Hiroshi Inoue et al. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors PACT '07 Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques pp. 189-198

[11] Bronislava Brejov Analyzing variants of Shellsort Information Processing Letters Volume 79, Issue 5, 15 September 2001, pp. 223-227

[12] Naga K. Govindaraju et al. GPU:TeraSort: High Performance Graphics Co-processor Sorting for Large Database Management SIGMOD 2006 Chicago, Illinois, USA

[13] Michael Herf, Dec 2001 Radix Tricks: Retrieved Oct 13rd, 2013 from: http://stereopsis.com/radix.html

[14] Andrew Davidson et al. Efficient Parallel Merge Sort for Fixed and Variable Length Keys Innovative Parallel Computing(InPar) ,2012 San Jose, USA

[15] Erik Sintorn and Ulf Assarsson Fast parallel GPU-sorting using a hybrid algorithm Journal of Parallel and Distributed Computing Volume. 68, Issue 10, October 2008, pp. 1381–1388

[16] Gianfranco Bilardi and Alexandru Nicolau Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines SIAM, Journal on Computing, 1989, Volume 15 Issue 2, pp. 216-228