TELKOMNIKA, Vol.14, No.1, March 2016, pp. 309~318
ISSN: 1693-6930, accredited **A** by DIKTI, Decree No: 58/DIKTI/Kep/2013
DOI: 10.12928/TELKOMNIKA.v14i1.2741

Optimized Merge Sort on Modern Commodity Multi-core CPUs

Ming Xu*[1], Xianbin Xu[1], MengJia Yin[1], Fang Zheng[2]
[1]Computer School of Wuhan University, Wuhan 430072, Hubei, China
[2]College of Informatics, Huazhong Agricultural University, Wuhan 430070, Hubei, China
*Corresponding author, email: xuming@whu.edu.cn

Abstract

Sorting is a kind of widely used basic algorithms. As the high performance computing devices are increasingly common, more and more modern commodity machines have the capability of parallel concurrent computing. A new implementation of sorting algorithms is proposed to harness the power of newer SIMD operations and multi-core computing provided by modern CPUs. The algorithm is hybrid by optimized bitonic sorting network and multi-way merge. New SIMD instructions provided by modern CPUs are used in the bitonic network implementation, which adopted a different method to arrange data so that the number of SIMD operations is reduced. Balanced binary trees are used in multi-way merge, which is also different with former implementations. Efforts are also paid on minimizing data moving in memory since merge sort is a kind of memory-bound application. The performance evaluation shows that the proposed algorithm is twice as fast as the sort function in C++ standard library when only single thread is used. It also outperforms radix sort implemented in Boost library.

Keywords*: merge sort, bitonic network, SIMD, AVX*

Copyright © 2016 Universitas Ahmad Dahlan. All rights reserved.

1. Introduction

Sorting is a kind of basic algorithm that is deeply researched and widely used. Sorting algorithms are utilized by many computer applications, which typcially include but will not limited to database system [1] and image processing applications [2]. Also all platforms have their respective implementations of various sorting algorithms. As hardware evolving, new sort implementations are continually emerging. Among them is parallel sorting, thanks to increasingly popular multi-core technology.

In the past few years, one of the major transitions in computer chip industry is from single cores to multiple cores. Processors based on multi-core micro-architecture become more and more popular. The two major types are today's CPUs and GPUs respectively. Among them are generations of Intel® Core™ Process Family and NVIDIA® GeForce series. For example, an Intel's multi-core CPU consists of multiple cores, each of which features sophisticated technologies such as out-of-order execution, hyper-threading, cache hierarchy, single instruction, multiple data (SIMD) instructions and etc. While a GPU processor can typically contains more cores called stream multi-processors, each of which contains more scalar processors that have the capable of execute a same instruction in concurrent. This makes GPU a good tool to accomplish missions that need process vast data in a same way. SIMD instructions is somewhat similar with those on GPU. Generally, a SIMD instruction can operate on a vector of data, which are stored in vector registers. Without exploiting this kind of parallelism, only a small fraction of computational power of a modern multi-core CPU can be utilized.

In this paper, a new optimized parallel sorting algorithm on CPU is proposed. It utilizes the multi-core CPU of 4th Generation Intel® Core™ Family [3-5] which provides new SIMD instructions, that is Intel® Advanced Vector Extensions (Intel® AVX) and Intel® Advanced Vector Extensions 2 (Intel® AVX2) [7]. Performance comparisons with other state-of-the-art sorting algorithms are also provided.

Received September 20, 2015; Revised December 26, 2015; Accepted January 17, 2016

2. Related Work

There are a lot of literatures on sorting and parallel sorting. Various kinds of sorting algorithms are deeply researched. They can be roughly classified to two major types- *comparison* and *non-comparison* sort [6]. The two representative comparison sort are merge sort and quick sort. A typical non-comparison sort is radix sort. There are also derivations of these basic algorithms such as sample sort [16] and bucket sort [22]. Parallel sorting algorithms can also be classified as either *merge-based* or *splitter-based* [20]. Usually different algorithms are combined to sort large data lists when they are sorted concurrently. For example, bucket sort or sample sort can divide items in a list to a set of continuous chunks, and then each chunk can be sorted independently by other algorithms [14, 15]. On the other hand, multi-way merge [12] can evenly divide the task of merging multiple data chunks which were sorted by other algorithms in advance and assign each part of the task to different processors.

Duane Merrill [8] presented a family of very efficient parallel algorithms for radix sorting on GPU. Allocation-oriented algorithmic design strategies that matches the strengths of GPU processor architecture to the kind of dynamic parallelism are described. The sorting passes are constructed from a very efficient parallel prefix scan runtime. Up to now, it is the state-of-the-art implementation of radix sort on GPU. there are also other algorithm implementations on GPU as described in [17-19], [21].

Nadathur Satish [11] presented a competitive analysis of comparison and non-comparison based sorting algorithms on the latest CPU and GPU architectures. Novel CPU radix sort and GPU merge sort implementations are proposed. To alleviate irregular memory accesses of CPU radix sort implementations, they proposed a buffer based scheme that collect elements belonging to the same radix into buffers in local storage, and write out the buffers from local storage to global memory only when enough elements have accumulated. By comparative analysis, merge sort, especially bitonic sorting network [13], is more SIMD friendly. The analysis points to merge sort winning over radix sort on future architectures due to its efficient utilization of SIMD and low bandwidth utilization. The combination of SIMD implementation of merging network and multi-way merge are proposed.

The main challenge of implementing a bitonic sorting network by SIMD instructions is to design efficient horizontal comparison processes. That is, comparisons between items within one vector register. Given that each vector register can hold v items, usually a bitonic sorting network implementation loads v^2 items into v SIMD registers. Items in these registers will be sorted and eventually form a small sorted data block end by end, which is then copied to memory by vector store instructions. To illustrate the sorting process, we can imagine that all items reside in a one-dimensional array, and the distance of two items is the difference of their indices of the array. During sorting process, bitonic sorting network will constantly pick two items to compare, and swap them if they are out of order. It is easily found that the distances of two items picked range from $v^2 - 1$, namely the first and the last item are picked, to 1, namely two adjacent items that reside in same vector register are picked. However, all forms of SIMD min/max instructions that are often used in a bitonic sorting network implementation take two SIMD registers as their input parameters. Two items to be compared must be chosen from different registers, whether they are in the same slot of the two registers or not. Due to lack of direct instruction to compare items in same register, it is essential to use data interchange instructions first to move one of each pair of items to be compared to another register before vertical comparison instructions can be used.

Chhugani [9] implemented a high performance bitonic sorting network in 128-bit wide SIMD instructions provided by Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2) and Streaming SIMD Extensions 3 (SSE3). Given items to be sorted are 32-bit integers, each XMM register can hold 4 items. A 4X4 sorting network is used to sort these loaded integers. A lane refer to all the same slots in SIMD registers that holds data. Odd-even sorting network is used to sort each lane, before all the items reside in a lane can be transposed to a SIMD register. After then bitonic sorting network is used to sort 4 registers, SIMD instructions including min/max and shuffle are used.

Xiaochen Tian [10] implemented a bitonic merge sort algorithm on Intel® Xeon Phi CPUs, which provide Intel® AVX-512 instruction set. To reduce data shuffle operations, masked comparison operations (e.g. _mm512_mask_min_epi32) were used. However, the algorithm also rearrange data items in register after each iteration.

Our previous work [23] have implemented bitonic sorting network by utilizing SSE intrinsic instructions. In this paper, a new implementation that utilizing new SIMD instructions and wider vector registers is described. CPUs that belong to the 4th and after generation Intel® Core™ Processor Family provide Intel® AVX and Intel® AVX2 that contain instructions can operate on vector data registers of 256-bit width, namely YMM registers. Under x86-64 mode, each core within a CPU can use up to 16 YMM registers. We implemented the bitonic sorting network against a quad-core Intel Core i7 CPU which based on the Haswell microarchitecture. Bitonic sorting network is used to sort a block of unsorted data items and merge two sorted lists step by step. The main contribution of this paper is: 1) propose a bitonic sorting network implementation utilize wider SIMD instructions and try to reduce data interchange between SIMD registers; 2) propose an optimized multi-way merge algorithm; 3) propose an merge process that can reduce data move times in memory. In the following section all the details are described, followed the fourth section that records the performance evaluation. The last section is conclusion.

3. Optimized Parallel Merge Sort

The basic process of our proposed implementation is very straightforward. First the unsorted input data list is divided into a certain number of data chunks, each of which is sorted by a single CPU core independently and concurrently. Then the multi-way merge [12] is used to partition these data chunks according to segments in the final sorted list, and gather data items for each such segment. The last process is to sort data items for all segments which can then form the final output. The first and last process are done both by recursively utilizing bitonic sorting network. All processes can be executed concurrently. In the remainder of this paper, notations below are used:

n Number of items of the input list.
c Number of data chunks that will be merged by multi-way merge
q Number of data segments that be produced by multi-way merge
p Number of CPU cores.
k The width of a vector register.
b The length of each item.
l Number of items can be sorted by running bitonic sorting network once.

3.1. Bitonic Sorting Network

The number of input items of the bitonic sorting network is decided both by the capacity of vector registers and the length of each item. An YMM register has capacity $k = 32$ bytes, and in our performance evaluation items of input lists are 32-bit integers, namely $b = 4$ bytes. So an YMM register can contain 8 items, and the number of input items of the bitonic sorting network implemented by us is $l = (k/b)^2 = 64$ since there are totally 16 YMM registers.

3.1.1. Initially Bitonic Sorting Network

If we divide the whole bitonic merge process to several steps by the unit length of input blocks to be merged in each step, then there are totally 6 steps, according to the unit lengths 1, 2, 4, 8, 16, and 32 respectively. The first three steps can be combined to one, because only min and max operations are used. Then there are 4 steps in a full sorting network. Same as [9], each lane is sorted in ascending order at first. This is what step 1 does. But then, items are interleaved instead of being fully transposed for the next comparison. During merging, items are always stored in lane order. After the second step, each even indexed lane and its next lane stores 16 sorted items end by end. After the last step, all sorted items stored in lanes as showed in Figure 1 (a). We try to reduce data interchange operations in each step as few as possible.

Now there is a branch in remainder of the process: if a full sorted output block is essential, sorted items must be transposed before they are stored to memory; if two partially sorted blocks (each contains $l/2 = 32$ items) is enough, then YMM registers that contains the result will be permuted so that all the higher half part of them will be moved together to form one output block, so does all the lower half part to form the other. In most cases items are stored in partially sorted style, except that multi-way merge is to be used next or this is the last step of the

sorting process. Store 32 items into memory in full sorted order needs 23 instructions, while store the same items in partially sorted order only need 8 instructions. Algorithm 1 illustrates the process stated above.

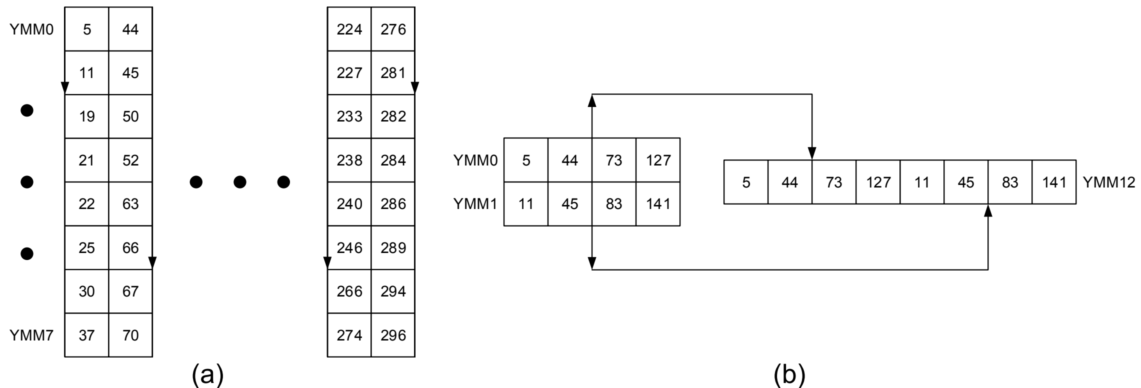

Figure 1. Illustrations of data layouts in YMM registers. (a) sorted items are placed in lane order. (b) lower half items are rearranged to partially sorted style.

Algorithm 1 Initially Bitonic Sort
Input: integer list ***S***, its length n . { n is multiple of 64}
1 **repeat**
2 load data from ***S*** to vector registers {vmovdqa used}
3 sort all lanes {vpminsd and vpmaxsd used}
4 horizontally and/or vertically interleave items {vpshufd, vpblendd or vperm2i128 used}
5 compare items
6 ... go on the bitonic sorting network, instructions same with line 3 and 4
7 **if** $n = 64$ **then**
8 transpose data items
9 **else**
10 rearrange data items to partially sorted blocks
11 **end if**
12 store items to ***S***
13 **until** all $n/64$ blocks in ***S*** are sorted

3.1.2. Intermediate Results

As stated above, items can be rearranged in two different ways before stored to memory. In practice, AVX/AVX2 instructions can take three or more parameters, for example, a permutation instruction takes four parameters, namely two source registers, one destination register and one mask integer. Then one half of YMM registers can be used to hold sorted data, the other hold the rearranged results. So in any case the registers used as source of vector store instructions can be the same, so does the items in them, only difference is the layouts of items. As shown in Figure 1 (b), items in lower 4 lanes of YMM0 ~ YMM7 are permuted into YMM12 ~ YMM15, which is in partially sorted layout.

When all items are initially sorted by a full bitonic sorting network, small blocks can now be merged to longer lists recursively. When merging is necessary, the way to merge is same as in [9], items are loaded from each list respectively. Here only the last step of a bitonic sorting network is executed, because the unit length of blocks is 32. Recall that the data being loaded has two possible layouts, If they are partially sorted, 12 instructions are needed to load and place them in lane order; and if they are fully sorted, 18 instructions are needed, including gather instructions that supporting load non-continuous items based on certain rules (base memory address, scale and mask), which are a new kind of operations in AVX2.

3.1.3. Only Copy

There is one more point need to be pointed out-two data blocks should be merged only when they contains items that are out of order. Otherwise the merging is redundant. Even the last step of bitonic sorting network needs 102 instructions. In performance evaluation, data sets used are all in uniform distribution. It is found that whatever the length of a list, there are chance of 12% ~ 15% that two data blocks to be merged are in order. Combine the two, it is worthwhile to know whether two data blocks are in order or not. The instructions used are two extract instructions, which extract higher half part of an YMM register to a XMM register and then extract the highest item from it afterwards. During merging process, higher half of YMM registers are always filled first. Each time when one block is loaded, the maximum item in it is extracted, if it is not greater than the minimum of the remainder items, then these items are stored to memory directly. Otherwise, the other block that contains the minimum item is loaded into lower half of YMM registers. After two blocks are merged, lower half of the result is stored to memory. Algorithm 2 illustrates the process stated above.

Algorithm 2 Bitonic Merge

Input: sorted source lists ***S1***, ***S2***, destination list ***S3***, input state ***T1***, output state ***T2*** {state indicates partially or full sorted}

1 load items from a source list, rearrange them base on ***T1***
2 **while** there are unmerged items **do**
3 ***S*** ← list contains minimal unmerged item or the only one contains unmerged items
4 ***RMax*** ← the maximum item in registers
5 ***LMin*** ← the minimum item in ***S***
6 **if** ***RMax*** ≤ ***LMin*** then
7 rearrange items in registers base on ***T2***, store them to ***S3***
8 load items from ***S***, rearrange them base on ***T1***
9 **else**
10 load items from ***S***, rearrange them base on ***T1***
11 merge two loaded blocks
12 rearrange the lower half of result base on ***T2***, store them to ***S3***
13 **end if**
14 **end while**

3.2. Multi-Way Merge

When the number of sorted data chunks is small, say $c \leq 2q$, the multi-way merge algorithm described in [12] is used to evenly assign remainder merging tasks to CPU cores. Like the respectively sorted chunks, the final sorted output can also be divided evenly to q data segments, what multi-way merge can do is to find each final segment's items from sorted chunks and aggregate them together. Then each segment can be merged independently and concurrently. To achieve the goal, all the boundaries that partition items which belong to different segments are found in all sorted data chunks.

Boundaries of a partitioning are formed by items in different data chunks. Suppose that the maximum value within the partitioning is $L_{\max}$, the minimum value within the upper boundary is $R_{\min}$, a correct partitioning must satisfy two requirements: The *ordering requirement* is that all items within a partitioning must have values less than or equal to all values of elements lying on its upper boundary, namely $L_{\max} \leq R_{\min}$; and the *distance requirement* is that the total number of included elements must be n/q. The lowest boundary is already known which is formed by all the first element of data chunks, the upper boundary of a partitioning is guessed first, which ensure that the partitioning satisfy the distance requirement. Followed several times of inspections and adjustments will ensure the partitioning satisfy the both. Partitioning can be found by this recursive looking up process because the upper boundary of current partitioning is also the lower boundary of the next partitioning.

The algorithm in [12] is a bit inefficient when it traverse items to find $L_{\max}$ and $R_{\min}$. The conditions used to check ordering requirement are also a bit complex. Given that the iteration

times of the loop is m, then the computation complexity of the loop is obviously $O(mc)$. We propose a new concise bounds look up process which has $O((c+m)\log(c))$ complexity and simpler judging condition. For each traversed item, a key-value pair is created, of which the key and value is the value and index of the item, respectively. Two binary tree is built based on these key-value pairs first, which takes $O(c\log(c))$. The tree used to find $R_{\min}$ is built with a less comparator, such that the root of the tree always holds the information of $R_{\min}$. Similarly, the tree used to find $L_{\max}$ is built with a greater comparator, such that the root of the tree always holds the information of $L_{\max}$. Just a simple comparison on the keys of the two root nodes is enough to know whether the ordering requirement is satisfied. The complexity of get root nodes is only $O(1)$. The process of adjustment is also easy. Two root nodes and at most 2 more nodes which have the same index value with any of the two roots will be deleted. New nodes added after adjustment are also come from the same data chunks which the old root nodes lying in. All these operations need $O(\log(c))$ complexity. To implement tree structure, multi-map container in C++ STL library is used, which is based on a red-black tree. Sum up all above, the computing complexity of bounds looking up process can be readily known. Algorithm 3 illustrates the process stated above.

Algorithm 3 Multi-way Merge

Input: list ***S***, temporary list ***T***, offsets of sorted chunks ***O***, length of a final sorted segment ***M***

1 ***L*** ← ***O*** {current lower boundary}
2 **repeat**
3 ***left*** ← a red-black tree has greater comparator {e.g. std::greater<int>}
4 ***right*** ← a red-black tree has less comparator
5 ***U*** ← upper boundary that satisfy distance requirement
6 traverse items on ***U*** and its left neighbors, build key-value pairs and insert them to ***right*** and ***left*** respectively
7 **while** key of ***left***'s root $\leq$ key of ***right***'s root **do**
8 ***LIndex*** ← index of ***left***'s root
9 ***RIndex*** ← index of ***right***'s root
10 remove nodes that have same index with ***LIndex*** or ***RIndex*** in both trees
11 select two items have greater/smaller value from chunk ***LIndex***/***RIndex*** respectively
12 insert nodes built based on selected items to trees
13 update ***U***
14 **end while**
15 copy blocks between ***L*** and ***U*** to ***T*** in descending order of their length
16 ***L*** ← ***U***
17 **until** upper boundary of each segment are found

3.3. Data Moving

After multi-way merge complete, blocks in same segment are merged. Bitonic sorting network are used again. It is known that merge sort is a kind of memory-bound operation. To speed up the merging process, we use low latency instructions as most as possible. We also try to optimize the merging process by reduce the data copying times.

3.3.1. Aligned Vector Memory Operations

Only aligned SIMD vector load and store instructions are used in our bitonic sorting network implementation. So the start memory address of input data array and temporary buffer must be both 32-byte aligned. And without loss of generality, all the data lists used for performance evaluation have length that is multiple of 64. However, it is almost certain that multi-way merge will produce data blocks that has start and/or end addresses are not 32-byte aligned. These data blocks need to be adjusted before aligned load instructions can be used.

The length of a final sorted segment which was stated above, namely n/q, can be set to multiple of 32. At the end of multi-way merge data blocks belongs to a same final segment will be copied together in a contiguous place in temporary buffer. Before copying, two more

tasks are done: one is to make all the start and end address of a block in new array is 32-bye aligned, and the other is to know the length of each data block, since they are to be copied according to the descending order of their length which is prepared for the next optimized merging.

To attain these goals, memory addresses of lower and upper boundary of each data block are checked. If the address of a lower boundary is not 32-byte aligned, then the nearest successive item which has 32-byte aligned address becomes the new lower boundary; and if the address of an upper bound is not 32-byte aligned, then the nearest previous item which has 32-byte aligned address is chosen as the new upper boundary. Items between old and new boundary are copied to one of two temporary vectors and then sorted. Vectors are allocated use 32 byte aligned allocator, too. Because the length of each segment is multiple of 32, so is the length of each data block, it is not difficult to found that, the number of items that copied to vectors is also multiple of 32. They can then be copied to temporary buffer as extra blocks and merged with others. So except boundary looking up process and copying items to vectors, all the other memory operations use aligned instructions.

Algorithm 4 Optimized Recursive Merge

Input: destination segment ***S,*** temporary buffer ***T***

1 ***Len*** ← function that returns the number of blocks of an array
2 **while** the total number of blocks >2 **do**
3   **if** ***Len(S)*** ≤ 1 **then**
4   **If** ***Len(S)*** $=1$ **and** ***Len(T)*** is odd **then**
5   merge the block in ***S*** and the first block in ***T***, store the result to the tail of ***S***
6   **end if**
7   merge blocks in ***T*** from head to tail, store results to ***S*** from tail to head
8   **if** ***Len(T)*** is even, **then** the result of the two last blocks store to the head of ***T***
9   **else**
10   **if** the head of ***T*** is not empty **then**
11   merge blocks in ***S***, store results to ***T*** from tail to head
12   **if** ***Len(S)*** is odd, **then** merge the first block of ***T*** and the last block of ***S*** into tail of ***S***
13   **else**
14   **if** ***Len(S)*** is odd, **then** merge the block in ***T*** and the first block in ***S***.
15   merge other blocks in ***S***, store all results of 14 and 15 to ***T***.
16   **end if**
17   **end if**
18 **end while**
19 merge the last two blocks, store result to ***S***. {all above merging is done by Bitonic Merge}

3.3.2. Ways to Reduce Data Copying

Besides utilizing aligned vector load and store instructions, the other way to optimizing memory operation is to reduce memory operations as much as possible. We have carefully designed the merging process in the last step so that data moving times between input data segments and their temporary buffers are dropped.

Before describing the merging process, one thing must be clarified. If two data blocks lies in a contiguous chunk of memory end to end, then the same chunk cannot be used as the destination of output, since the merged output will likely overwrite the unmerged inputs. Copying data to a temporary buffer is necessary to avoid overwrite. The thing is, only copying the data block that lying in the lower memory space is enough. Regardless of the length of two data blocks, if the data block in front of the memory space is moved to buffer, then the chunk of memory can safely store merged results. Based on it we reduced the data moving in merging process.

We started at temporary buffer. All blocks are copied here after boundary looking up complete, and recall that block are stored in descending order of their length. Blocks are merged from the head of the buffer to the tail, while merged results are placed into input segment from tail to head. There is a block stay in the tail of the temporary buffer, if the number of blocks in the buffer are odd; otherwise the merged result of the last two blocks will placed to the head of the temporary buffer. The space is enough to hold the result of the two, because the

blocks before them are all not shorter than them. Now the blocks in the input segment are stored in ascending order of their length.

Then the number of blocks lying in the input segment may be more than one. If the block in the temporary buffer are stored in the head, then merged output yielded by blocks lying in the input segment are stored from the tail of the buffer to the head. If the number of blocks in the input segment are odd, then the first block in the buffer will be merged into the input segment with the last block in the input segment. On the other hand, if the block in the temporary buffer is stored at the tail, and the number of blocks in input segment is odd, the two first block are merged, stored at the tail of the buffer. Other blocks are merged successively.

On the whole, our goal is to ensure the temporary buffer is never empty so that memory copying in merging process can be decreased. When the total number of unmerged blocks is more than two, the process of merging is as stated above and illustrated in Algorithm 4; so that when the number of blocks is two, they can be merged into the input directly.

4. Performance Evaluation

The machine used for performance evaluation has an Intel Core i7 4700MQ CPU which works at 3100MHz. The CPU has four cores that can deliver 8 threads via Intel Hyper-Threading Technology. The chip on mainboard has two dual-channel DDR3 memory controllers which provides 1600MHz memory clock frequency. The total capacity of main memory is 16GB. The operating system and compiler used is Windows 10 and Visual Studio 2013, respectively. All the results are average values attained by running test application 30 times. In figures below, all the axes are logarithmic which base is 2.

First we compared our bitonic sorting network with other two algorithms in single thread. One is the sort function in C++ standard library which uses quicksort; the other is spread sort introduced in Boost library 1.58 which uses radix sort for large data sets. In Figure 2 we can see that our bitonic sorting network is strikingly faster than the sort function in C++ standard library. The speed is never lower than twice of the latter. Our bitonic network also outperform the spread sort, although the gap is narrow when the length of data sets is huge.

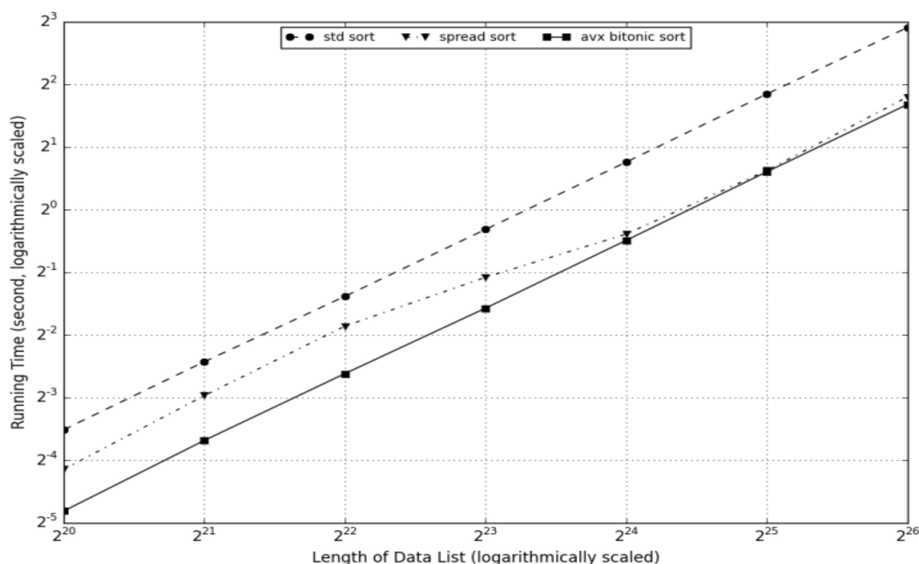

Figure 2. Performance comparison of different algorithms running in single thread

Next we tested multi-way merge with multi-threads. From Figure 3 we can see that when we use 8 threads, the performance get a noticeable improvement. This shows that our implementation can also benefit from hyper-threading. So we use 8 threads for the following tests. In Figure 4 we show the performance of our multi-threaded sorting implementation, which can sort one billion 32-bit integers in 17 seconds.

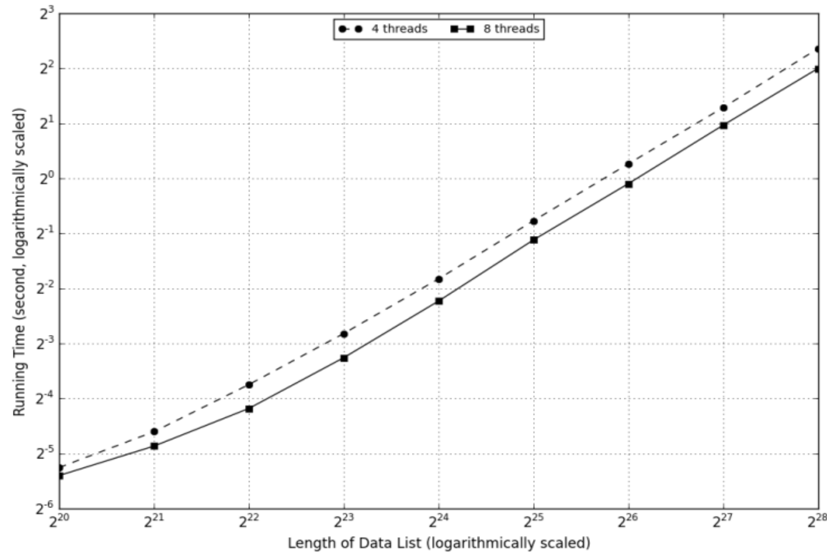

Figure 3. Performance comparison bitonic sorting in 4 and 8 threads respectively.

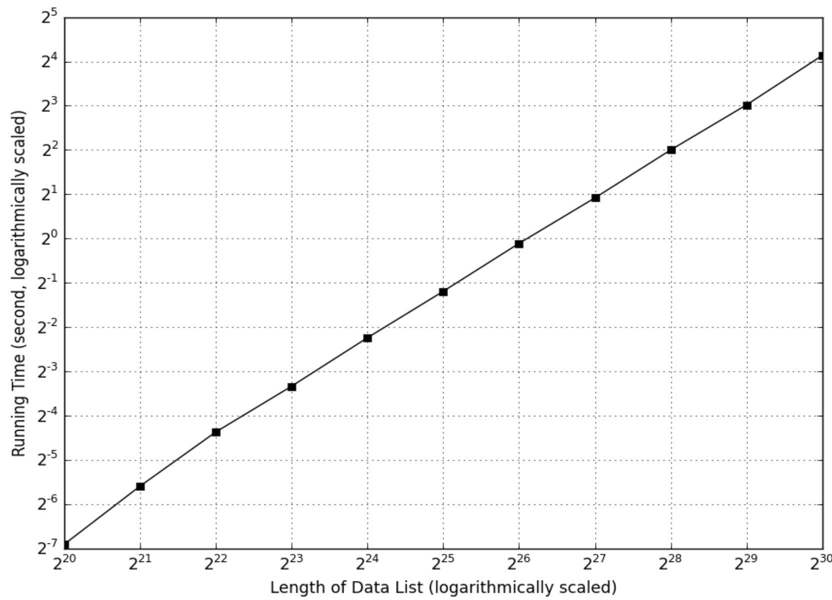

Figure 4. Performance evaluation of optimal merge sort

5. Conclusion

We proposed a new parallel sorting implementation which utilizes new AVX instructions. Data are stored by lanes when they are being sorted in vector registers, which reduces the number of SIMD instructions needed. Our single thread bitonic network strikingly outperform the sort function in C++ standard library, so does the radix sorting implementation in Boost library. Our multi-threaded algorithm can sort one billion integers in less than 17 seconds.

Acknowledgements

The work described in this paper is supported by the Natural Science Foundation of Hubei Province of China (No.:4006-36115030), and the Fundamental Research Funds for the Central Universities (No.:52902-0900206297).

References

[1] Subroto IMI, Sutikno T, Stiawan D. The Architecture of Indonsian Publication Index: A Major Indonsian Academic Database. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*. 2014;12(1): 1-5.

[2] Djajadi A, Laoda F, Rusyadi R, et al. A Model Vision of Sorting System Application Using Robotic Manipulator. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*. 2010; 8(2): 137-148.

[3] Kurd N, Chowdhury M, Burton E, et al. Haswell: A Family of IA 22 nm Processors. *IEEE Journal of Solid-state Circuits*. 2015; 50(1): 49-58.

[4] Hammarlund P, Kumar R, Osborne RB, et al. Haswell: The Fourth-generation Intel Core Processor. *IEEE Mirco*. 2014; 2: 6-20.

[5] Jain T, Agrawal T. The Haswell Microarchitecture-4th Generation Processor. *International Journal of Computer Science and Information Technologies*. 2013; 4(3): 477-480.

[6] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3rd Edition. MIT press. 2009.

[7] Intel® 64 and IA-32 Architectures Software Developers Manual. Intel Corporation. 2014.

[8] Merrill D, Grimshaw A. High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Pomputing. *Parallel Processing Letters*. 2011; 21(02): 245-272.

[9] Chhugani J, Nguyen AD, Lee VW, et al. *Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture*. Proceedings of The VLDB Endowment. 2008; 1(2): 1313-1324.

[10] Xiaochen T, Rocki K, Suda R. *Register Level Sort Algorithm on Multi-core SIMD Processors*. Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms. ACM. 2013: 9-16.

[11] Satish N, Kim C, Chhugani J, et al. *Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort*. Proceedings of The 2010 ACM SIGMOD International Conference on Management of Data. ACM. 2010: 351-362.

[12] Francis R, Mathieson I, Pannan L. *A Fast, Simple Algorithm to Balance A Parallel Multiway Merge*. PARLE'93 Parallel Architectures and Languages Europe. Springer. 1993: 570-581.

[13] Batcher, Kenneth E. *Sorting Networks and Their Applications*. Proceedings of The April 30-May 2, 1968, Spring Joint Computer Conference. ACM. 1968: 307-314.

[14] Chen S, Qin J, Xie Y, et al. *A Fast and Flexible Sorting Algorithm with CUDA*. Algorithms and Architectures for Parallel Processing. Springer Berlin Heidelberg. 2009: 281-290.

[15] Zhang K, Wu B. *A Novel Parallel Approach of Radix Sort with Bucket Partition Preprocess*. 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS). IEEE. 2012: 989-994.

[16] Sanders P, Winkel S. *Super Scalar Sample Sort*. ESA. Springer. 2004; 3221: 784-796.

[17] Leischner N, Osipov V, Sanders P. *GPU Sample Sort*. 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE. 2010: 1-10.

[18] Satish N, Harris M, Garland M. *Designing Efficient Sorting Algorithm for Manycore GPUs*. IEEE International Symposium on Parallel & Distributed Processing, IPDPS. IEEE. 2009: 1-10.

[19] Ye X, Fan D, Lin W, et al. *High Performance Comparison-based Sorting Algorithm on Manycore GPUs*. 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE. 2010: 1-10.

[20] Solomonik E, Kale LV. *Highly Scalable Parallel Sorting*. 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE. 2010: 1-12.

[21] Peters H, Schulz-Hildebrandt O, Luttenberger N. *A Novel Sorting Algorithm for Many-core Architectures Based on Adaptive Bitonic Sort*. 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS). IEEE. 2012: 227-237.

[22] Zhao Z, Min C. *An Innovative Bucket Sorting Algorithm Based on Probability Distribution*. IEEE WRI World Congress on Computer Science and Information Engineering. 2009;7: 846-850.

[23] Xu M, Xu XB, Zheng F, et al. A Hybrid Sorting Algorithm on Heterogeneous Architectures. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*. 2015; 13(4): 1399-1407.