TELKOMNIKA, Vol. 13, No. 4, December 2015, pp. 1399~1407
ISSN: 1693-6930, accredited A by DIKTI, Decree No: 58/DIKTI/Kep/2013
DOI: 10.12928/TELKOMNIKA.v13i4.1896

A Hybrid Sorting Algorithm on Heterogeneous Architectures

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

Abstract

Nowadays high performance computing devices are more common than ever before. The capacity of main memories becomes very huge; CPUs get more cores and computing units that have greater performance. There are more and more machines get accelerators such as GPUs, too. Take full advantages of modern machines that use heterogeneous architectures to get higher performance solutions is a real challenge. There are so much literatures on only use CPUs or GPUs; however, research on algorithms that utilize heterogeneous architectures is comparatively few. In this paper, we propose a novel hybrid sorting algorithm that let CPU cooperate with GPU. To fully utilize computing capability of both CPU and GPU, we used SIMD intrinsic instructions to implement sorting kernels that run on CPU, and adopted radix sort kernels that implemented by CUDA (Compute Unified Device Architecture) that run on GPU. Performance evaluation is promising that our algorithm can sort one billion 32-bit float data in no more than 5 seconds.

Keywords*: SIMD, CUDA, Heterogeneous Architecture, Sorting Algorithm*

1. Introduction

Sorting is used by so many computer applications [1]. Database applications have sortingas an internal operation which is used by SQL operations, and hence all applications using a database can take advantage of an efficient sorting algorithm [2]. Sorting facilitates statistics related applications including finding closest pair, determining an element's uniqueness, finding *kth* largest element, and identifying outliers. Other applications that use sorting include computer graphics, computational geometry, computational biology, supply chain management, image processing [3, 4] and data compression.

Recently, main memory capacity in modern machines increase rapidly, which makes in-memory sorting feasible. Today's CPUs are typically multi-core processors with improved cache throughput and slowed down power consumption [5, 6, 7]. Cores on a multi-core processor may be homogeneous or not, share a certain level of cache memory, and implement architectures such as SIMD (single instruction, multiple data), multithreading [8]. Algorithms that utilize these advantages may get higher performance.Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth [9]. GPU is specialized for compute-intensive, highly parallel computation- exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. NVIDIA introduced CUDA™, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than that on a CPU. When correctly leverage the compute capability of GPUs, algorithms that migrated to GPU may get 10X and much more speed up than they run on CPU.

Many researches have done on high performance computing that only use CPUs or GPUs, but few on how to coordinate the two kind of processors. We are facing the challenge that take full advantages of a heterogeneous architecture machine can have. As a kind of fundamental algorithm, sorting is chosen by us to be re-implemented on a heterogeneous architecture machine. In the next section we summarize various parallel sorting algorithms as

Received August 26, 2015; Revised October 29, 2015; Accepted November 12, 2015

well as research on heterogeneous architectures, in section 3 we present details of our novel hybrid sorting algorithm, performance evaluations are presented in section 4, and section 5 is conclusion.

2. Related Research

There were many researches on parallel sort algorithms, include CPU and GPU implementations.

Peter Sanders and Sebastian Winkel [10] implemented Sample Sort on single processor machines, which is formerly known as the best practical comparison based sorting algorithm for distributed memory parallel computers. They used random sample to find *k-1* splitters that partition the input list into buckets of about equal size, which were then sorted recursively. Each element were placed into correct bucket that determined by binary search. To avoid conditional branches the splitters are formed a search tree analogous to the tree structure used for binary heaps. Elements traverses the search tree for placement independently and introduction of oracles from radix sort make the compiler optimization more easily. They also argued that sample sort is more cache efficient than quick sort because it moves the elements only $\log_k n$ times rather than $\log n$ times. The oversampling factor is used to make a flexible tradeoff between the overhead for handling the sample and the accuracy of splitting. They argued that their implementations obtained of up to 2X acceleration over std::sort from the GCC STL library. Shifu Chen and Jing Qin [11] proposed a sorting algorithm implemented in CUDA. They used an algorithm similar with that used in [10], but implemented it on GPU using CUDA. To improve performance, they used different methods to load elements from unsorted lists and nearly sorted lists, respectively. They argued that their experiments show that the algorithm achieves a 10X to 20X acceleration over STL quicksort.

Hiroshi Inoue and Takao Moriyama [12] proposed an aligned access sorting algorithm which takes advantage of SIMD instructions and thread-level parallelism available on today's multi-core processors. They used an improved combsort as their in-core algorithm, which is similar with bitonic sorting networks. After all divided blocks which length less than the capacity of L2 cache of a core are sorted, they are merged use out-core algorithm, which integrate odd-even merge algorithm implemented with SIMD instructions.

Nadathur Satish and Mark Harris [14] described the design of high-performance parallel radix sort and merge sort routines for many-core GPUs, taking advantage of the full programmability offered by CUDA. They designed a more efficient radix sort by making efficient use of memory bandwidth, which is gained by (1) minimizing the number of scatters to global memory and (2) maximizing the coherence of scatters. They also designed a merge sort on GPU, which is similar with other merge sort implementations that divide input into equal-sized tiles and sort them simultaneously, then merge these sorted tiles. To avoid inefficient coarse-grained merge process, they merge larger arrays by dividing them up into small tiles that can be merged independently using the block-wise process. They argued that their performance experiments show that their radix sort is the fastest GPU sort and their merge sort is the fastest comparison-based sort in the literature.

Jatin Chhugani and William Macy [2] proposed an efficient merge sort on multi-core SIMD CPU architecture. They first examined how instruction-level parallelism, data-level parallelism, thread-level parallelism, and memory-level parallelism impact the performance of sorting algorithm, then described their algorithm that utilize these parallelisms. They first evenly divide the input data into blocks and sort each of them individually. Blocks are assigned to a thread and sorted by a SIMD implementation of merge sort. Then these blocks are merged by iterations. In each iteration, pairs of lists are merged to obtain sorted sequences of twice the length than the previous iteration. When blocks becomes too long to be merged individually, threads corporate to merge pairs of blocks. In this case the median of the merged list are computed, which cut pairs of blocks into mutually exclusive parts that can be merged by single thread individually. They also use multi-way merge for large input list to reduce memory access during sorting.

Nadathur Satish and Changkyu Kim [15] presented a competitive analysis of comparison and non-comparison based sorting algorithms on 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

base 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. They adopted the GPU radix sort implementation described in [11]. CPU merge sort implementation described in [2] and GPU merge sort implementation described in were also adopted, besides on GPU, wider merge network were used.

All algorithms mentioned above only utilize single homogeneous processors. However, multicore machines equipped with accelerators are becoming increasingly popular. In the field of HPC, the current hardware trend is to design multiprocessor architectures featuring heterogeneous technologies such as specialized coprocessors or data-parallel accelerators, too. We face the challenge that building applications that can fully utilize the capability of entire machine, that is, parallel tasks can be scheduled over the full set of available processing units to maximize performance. Andre R. Brodtkorb and Christopher Dyken [16] provided an overview of node-level heterogeneous computing, including hardware, software tools and state-of-the-art algorithms and applications.

Edgar Solomonik and Laxmikant V. Kal´ e [17] have implemented several parallelsorting including histogram sort, sample sort and radix sort using Charm++. They focused on optimizing histogram sort and argue that histogram sort is mostreliable to achieve a defined level of load balance. They illustrated their point ofview by performance evaluations run on machines with modern supercomputerarchitectures.

Cédric Augonnet and Samuel Thibault [18] designed StarPU, which is a software tool aiming to allow programmers to exploit the computing power of the available CPUs and GPUs, while relieving them from the need to specially adapt their programs to the target machine and processing units. They implement several scheduling algorithms on StarPU and compared them with similar tasks.

In this article, we use a simpler way to implement hybrid sorting on heterogeneous architectures. Using Open MP, we use one single CPU thread to communicate with GPU when we need assign tasks to GPU while the other CPU threads execute tasks assigned to them. In whole sorting process, CPU and GPU are tightly bound that they cooperate with each other to complete each step, how tasks are assigned must be decided beforehand to guarantee CPU and GPU take almost same time to complete their tasks. To harness the capability of heterogeneous architectures, we implemented sorting kernels for CPU, and adopted a GPU implementation, both of them can take full advantage of CPU and GPU resources, respectively.

3. Proposed Hybrid Sorting Algorithm

We use SSE intrinsic instructions to utilize SIMD capabilities of CPU cores; CUDA is used to utilize GPU computing capabilities; and OpenMP runtime is used to coordinate tasks among CPU cores as well as between CPU and GPU.

We use 32-bit float as the data type of the elements to be sorted. Hence one 128-bit SIMD vector register contains 4 keys. Note that our algorithm is not limited to this data type and degree of data parallelism as long as the SIMD instructions support them. We suppose that the number of elements of input data list is N, which is power of 2 without loss of generality. We always use aligned move when SIMD registers load or store data, so the first elements of lists are aligned to 16-byte boundary, and we will handle unaligned cases during sorting process which will be explanation in section 3.3.

There are two stages in an entire hybrid sorting. In the first stage, CPU and GPU may have their own tasks that to sort unsorted parts of input list. In second stage, all sorted parts is merged. We implemented multi-way merge [19] to reduce the global synchronization in second stage.

We implemented merge sort described in [2] on CPU, which uses two steps to sort a whole data sequence. In first step, a data list is divided to several data chunks of same length, which is sorted one by one. In second step, sorted chunks are merged. We use SSE instructions implement a bitonic merging network [13, 20] which is used in both steps. SIMD implementation can sort 32 elements once a time, confined to the capacity of SIMD registers.

We adopted GPU radix sort kernels in CUB library [21]. CUB provides state-of-the-art, reusable software components for every layer of the CUDA programming model, including device-wide, block-wide and warp-wide primitive functions as well as many useful utilities. We used device-wide radix sort kernels in CUB library to ease our GPU programming tasks.

As the performance evaluation results below show, parallel merge sort kernels run on CPU have similar performance with radix sort kernels run on GPU. When the input data list is small, CPU sort implementations will even outperform GPU sort implementations. Since the main memory is often huge, CPU kernels can sort very large data lists. On the contrary, due to the capacity limit of global memory, only small or medium sized data lists can be sorted by GPU directly, large data lists must be divided into several parts and sorted by running GPU kernels several times, which is very inefficient. The advantages of GPU sort implementation is that when the length of input data list increase, the time needed to sort increases more slowly than CPU kernel does. The bigger the data list is, the more outperforms that GPU kernels than CPU kernels.

3.1. GPU Kernels

We use GPU to unload CPU sorting tasks. We adopted device-wide radix sort functions in CUB library [21] as our GPU kernels. In each stage, GPU kernels may sort one or several data chunks. Because data transfer between main memory and GPU global memory is relatively slow and GPU kernel will surpass CPU kernel more and more when the length of input data list increases as long as the data list can fit in the GPU global memory, we cannot assign same proportion of tasks when we sort different data lists. In this paper, we use a segmented way that designate several length intervals based on our performed evaluations. When the length of an input list belong to a length interval, it is divided into parts which will be sorted by CPU and GPU respectively and simultaneously by the corresponding proportion of the interval. If input data list is small enough, it is sorted only by CPU; when the length of it increases, proportion of GPU tasks will also increase. Different stages of our sorting process may use different proportions, too. This way of task scheduling fit well in our implementation, though it is not very flexible.

3.2. CPU Kernels

We use two steps to sort data lists on CPU. In first step, we partition the whole list into several data chunks, the length of each chunk is selected so that L3 cache can safely hold two chunks. Each chunk is further divided into blocks so that each L1 cache of a CPU core can safely hold two blocks. CPU cores sort these blocks concurrently and then coordinate with each other to merge them to a sorted chunk. Next in second step, chunks are merged to a sorted list. If the number of chunks is small, the kernel merges two chunks once a time, each two chunks are segmented by medians [22], and then each two segments of same index in two chunks respectively are merged simultaneously. If the number of chunks is large, we use multi-way merge [19] to segment them and several same index parts are merged. To full utilize the capability of caches of CPU cores, all the data are sorted and merged in cache. The size of data chunks and data blocks are calculated based on the size of L1 and L3 cache as well as the size of input data list. Each time a data block is moved into L1 cache, it is sorted by a sorting network which is implemented using SIMD intrinsic instructions.

3.2.1. Sorting Network

The core of our CPU sort kernels is sorting network implemented using SIMD intrinsic instructions, include odd-even sorting network and bitonic sorting network [2, 13, 20]. For unsorted blocks, we first load keys into SIMD registers, then use odd-even sorting network to make all keys within a single SIMD register are sorted. For partially sorted blocks, we use bitonic sorting network merge keys. In merge processes, 4 by 4 bitontic sorting network, 8 by 8 bitonic sorting network and 16 by 16 bitonic sorting network are used, decided by the length of key slices that load into SIMD registers. Since entire sorting takes most time to merge partial sorted data, 16 by 16 bitonic sorting network is mostly used.

There is also a trick on how to load keys to SIMD registers. Each SIMD register can contain a 4-element vector. The CPU kernel we first implemented would check if two arrays that to be merged are empty whenever it want to load keys from anyone of them to SIMD registers, which is a bit tedious. To be concise, we changed the keys chosen logic so that the first keys of two last data vectors that belongs to the two sorted array respectively will be compared beforehand. If a data vector's first key is smaller, then the array it belongs to will be empty first. Furthermore, all key vectors in the other array that should be loaded after this last vector can be

found, that is, the times of load operations, before and after an array is empty, is clear and definite.

3.2.2. Merging Two Chunks

If the number of sorted chunks is not greater than 4, every two chunks will be merged to a larger data chunk. Medians [22] are first calculated which will partition each chunk into segments, so that two segments of same index in different chunks respectively will belong to a same finally sorted data block. The sum of the length of two same indexed segment is also equal to the length of a block. Data chunks are sorted, so any two segment in same chunk that belong to two adjacent sorted blocks respectively are also adjacent, which divides by a median. Then every two same indexed segments are merged simultaneously by 16 by 16 bitonic sorting network. Because segments separated by medians are not necessarily aligned, that is, the memory address of the first key of a segment is not divisible by 16, or the length of the segment is not divisible by 16. We need to handle these unaligned case before we can use bitonic sorting network merge them.

3.2.3. Multi-Way Merge

Suppose that there are L sorted data chunks after stage 1 finished and L is also power of 2, if we use kernels that described in section 3.2.2 to merge them, we will run these kernels $L-1$ times to get an entire sorted output. So if $L>4$, this way will no longer be efficient, let alone transfer data between main memory and GPU for so many times. The machine we used for performance evaluation has a 4-core CPU with Hyper-Threading technology, which can at most sort 8 blocks simultaneously. Above kernel is inefficient to merge so many blocks, too.

We implemented multi-way merge algorithm proposed in [19] to merge more than two sorted chunks at a time. Similar with two chunks merge, we need first find all the elements belong to a same block in the merged list within the data chunks. We copy all data slices that belong to a same sorted block into a single CPU core L1 cache, merge them use 16 by 16 bitonic sorting network, once two data slices.

By using multi-way merge algorithm we can merge several data chunks in a single merge kernels run, however, the process of find the quantiles in chunks, namely upper bounds and lower bounds of the sorted blocks is fairly complex and will take longer time if the number of chunks merged every time is too large. So there must be a tradeoff. We choose the number of sorted chunks that using multi-way merge every time no more than 16. If the number of chunks is greater than 16, we recursively use multi-way merge or two chunks merge on merged larger chunks.

There's another factor that effect the quantile search process. As described in [19], only the upper bound of a sorted block are searched. That requires the lower bound and the length of the block must be known. The first elements of all chunks can be used as the lower bound for all upper bound calculation, but the length of blocks will then multiplied by the index of the searching upper bound, which end up to many redundancy and lengthy search processes. To avoid such inefficient case, first bounds for each sorted chunk within the finally sorted list are successively searched, each chunk can take the former outcome – the upper bound of last adjacent chunk – also is the lower bound of it. Then CPU cores simultaneously search bounds of blocks within sorted chunks, one core takes an entire chunk – also each block has its lower bound found, which is very efficient.

3.3. Unaligned Case Handle

The most important kernel in our CPU sorting is bitonic merging network whic3h occupies most CPU running time. To get high performance, we must ensure that all data transfer to and from SIMD registers is aligned. So unaligned cases is detected and handled before blocks of keys could be merged using merging network. Unaligned cases often occurred in stage 2, when we calculate medians for two chunks or bounds for several chunks. Because we always select the length of chunks or blocks is power of 2, the sum of length of segments divided by medians or quantiles is always power of 2, too. What we did is to cut down some elements from the head or tail of each unaligned segment to make the residual part of it and a new array that consist of all cut elements are all aligned.

3.3.1. Handle Unaligned Case When Merging Two Chunks

If the address of the head or tail element of one segment is not aligned by 16-byte, then so does the head or tail element of the other segment. If that is the case, the nearest element to the head or tail that its address are 16-byte aligned are computed. If the head of segments is not aligned, all elements before the nearest aligned elements from both segments are cut and copied into an aligned buffer and sorted. The length of buffer must be 16. The keys in buffer are loaded as usual as if we “insert” them into the segment that the last element of the buffer belongs to.

Similarly, unaligned tail elements are cut and combined to a buffer of 16 elements, the buffer is sorted and merged as if it is appended to the segment that the first element of the buffer belongs to.

3.3.2. Handle Unaligned Case When Multi-Way Merging

When multi-way merge are used, a few more segments that belongs to a block will be merged. Recall that the sum of length of all segments belong to a sorted block is always power of 2. Before they are merged, some tail elements will be cut down from segments whose length cannot divisible by 16, the number of cut elements is the remainder. All the cut elements are copied to a buffer, whose length must be divisible by 16. The buffer is then sorted and appended to the segment that its first element belongs to. Now all segments are divisible by 16, when they are copied to a L1 cache successively, all segment is aligned.

3.4. Hybrid Sorting Implementation

The hybrid sort implementation has two stages. Because CPU and GPU simultaneously sort different parts of the input list, global synchronize is necessarily between two stages. In each stage, Open MP library is responsible to manage task assignments as well as synchronization between CPU cores and GPU. Sections directive is used to assign tasks in each stage, described in the pseudo code below:

```
1  void hybrid_sort(data)
2  {
3  #pragma omp parallel
4      {
5          omp_set_nested(2);
6  #pragma omp sections
7          {
8  #pragma omp section
9              {
10                 gpu_kernel(data_chunk1);
11             }
12 #pragma omp section
13             {
14                 cpu_kernel(data_chunk2);
15             }
16         }
17     }
18     cudaDeviceSynchronize();
19 }
```

Code blocks that in different section directive will execute simultaneously. Since OpenMP library also coordinate different CPU cores, nested directive is necessarily. At the end of each stage, CUDA synchronization function guarantees that GPU has finished its tasks.

In stage 1, input data list is broken into two parts, one will be sorted by CPU, and the other will be sorted on GPU. The part to be sorted on CPU will be further broken into smaller chunks, each chunk can be safely hold by L3 cache, they will be sorted one by one. When all these chunks are sorted, they’ll be merge to a sorted part. Similar as CPU, the part to be sorted on GPU will be broken into chunks if the length of it exceeds GPU kernel can sort once. In stage 2, multi-way merge algorithm first searches the finally sorted chunks, whose bounds divide sorted parts in stage 1 into segments. Different segments are again assigned to CPU and GPU

respectively. To get best performance, the length of parts in each stage must be deliberately given so that CPU kernel and GPU kernel runs similar time. Based on performance evaluation for each sort kernel, different ratios are specified for different sort stage and different length of input list. When the length of list is small (less than L3 cache can hold and be sorted once with CPU kernel of stage 1), CPU kernel is enough; as the length of data list grows larger, part that assigned to GPU kernel also increase to get performance boost in stage 1 due to GPU sort kernel has better performance than CPU kernel when list is unsorted. The more data list longer, the more GPU will do than CPU will. In both stage, GPU kernel uses radix sorting, performance discrepancy is very low; however, CPU kernel in stage 2 will take much less time than in stage 1 as well as GPU kernel, due to multi-way merge can take full advantage of the former partial results. So, in stage 2 we assign CPU more task than GPU.

4. Performance Evaluation

We have evaluated the performance of our hybrid sort on a machine that has an Intel Core i7 4700MQ CPU which works at 2.40GHz, and a NVIDIA GeForce GTX 765M GPU which works at 800MHz. the CPU has four cores that can deliver 8 threads via Intel Hyper-Threading Technology. The GPU has 768 CUDA cores. The capacity of main and global memory for CPU and GPU is 16GB and 2GB, respectively. We run our implementations on OpenSuse Linux 13.1.

Table 1. Single GPU performance evaluation

Number of Elements	One-way transfer (ms)	Sorting (ms)	Sum (ms)
2^{17}	0.139693	0.645861	0.925247
2^{18}	0.267403	1.00632	1.541126
2^{19}	0.432259	1.64044	2.504958
2^{20}	0.76247	2.92104	4.44598
2^{21}	1.42201	5.43771	8.28173
2^{22}	2.73591	10.529	16.00082
2^{23}	5.36332	20.7026	31.42924
2^{24}	10.6202	41.1297	62.3701
2^{25}	21.1348	81.6049	123.8745
2^{26}	42.16	163.207	247.527
2^{27}	84.213	327.713	496.139

We first present the evaluation results of GPU sorting. We use radix sort in the CUB library to sort random distribution 32-bit floats. Table 1 presents the average results of 50 times run for each input list. Time taken by one-way data transfer, sorting and the whole process are listed respectively. Because of the capacity constraints of global memory of the GPU, the length of lists is only up to 2^{27}.

Table 2. Sing CPU performance evaluation

Number of Elements	Step 1(s)	Step 2(s)	Sum(s)
2^{16}	0.00026897	-	0.000269016
2^{17}	0.000564282	-	0.00056433
2^{18}	0.00118396	-	0.001184011
2^{19}	0.00251854	-	0.002518589
2^{20}	0.00571485	-	0.005714911
2^{21}	0.0137335	-	0.013733559
2^{22}	0.0222362	0.00565383	0.02789003
2^{23}	0.0449629	0.0136799	0.0586428
2^{24}	0.0892215	0.0419758	0.1311973
2^{25}	0.178331	0.104341	0.282672
2^{26}	0.389743	0.193178	0.582921
2^{27}	0.798254	0.45844	1.256694
2^{28}	1.60057	1.14128	2.74185
2^{29}	3.25119	2.63423	5.88542
2^{30}	6.55064	5.1366	11.68724

Next we present our evaluation of CPU sorting. The data lists and times run on each list are identical with the lists used by evaluation of GPU sorting. Both of two steps of CPU sorting are recorded, as well as the total time taken. The capacity of main memory of the CPU is so large that we can even sort a list of length up to 2^{30}, though it will take a longer time. Table 2 presents all the results. It is worth noting that we chose the length of a data chunk that be merged in cache be no more than 2^19, when there will be no more than 4 data chunks, namely when the number of elements of the list is less or equal than 2^21, step 1 is enough to sort the entire list.

Being carefully scheduled, CPU and GPU can cooperate with each other to get higher performance. Figure 1 illustrated performance evaluation results of our hybrid sort algorithm. The time axis of it was logged, the graphic is nearly linearly except some points. We consider it is because our task assignment algorithm is not very flexible. We illustrated sorting rate of each three algorithms in Figure 2. It is obviously that GPU sorting always has highest rate for 32bit float keys, but the memory of GPU is too small to sort large datasets. Hybrid sort have much higher rate than CPU sort when the size of data is medium or large. We think hybrid sort rate lower than GPU sort rate due largely to the partial results of Stage 1 is not used by GPU in Stage 2.

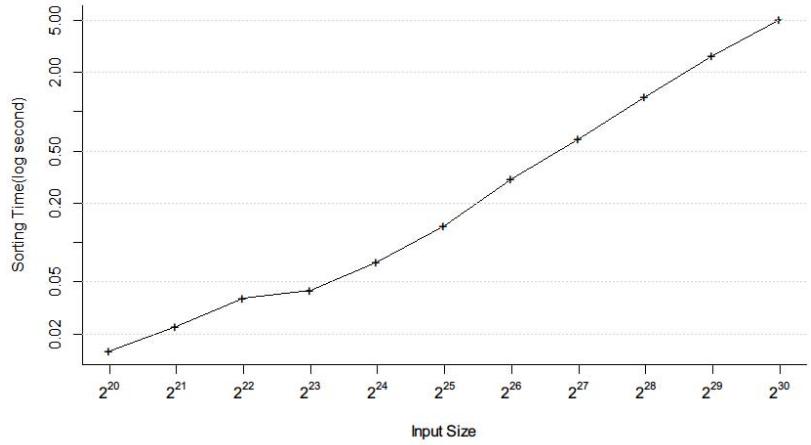

Figure 1. Hybrid sort performance evaluation

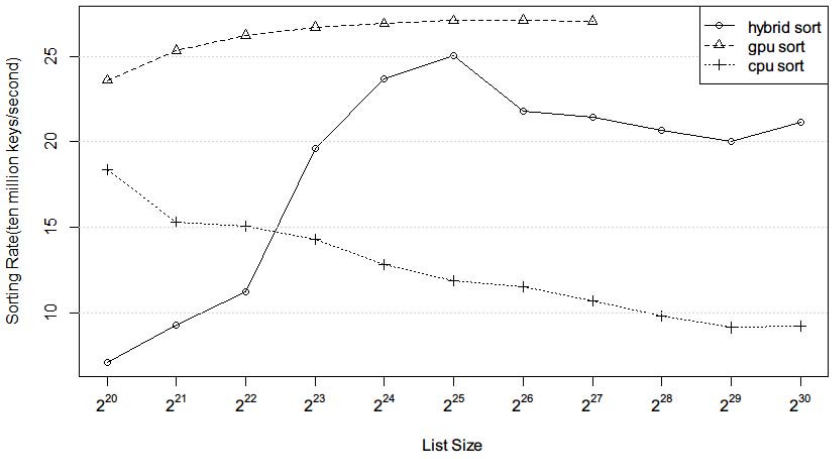

Figure 2. Sorting rate comparison

5. Conclusion

Heterogeneous architectures appears to prevail nowadays. Software developers must face the challenges that to design applications that can make full utilize of all these processors to get higher performance. We used a simple way to implement a hybrid sorting algorithm, which can sort one billion 32-bit float data in no more than 5 seconds.

Future work has two aspects. First, we shall update our applications to make more use of the processors. We can adopt a wider SIMD length by use AVX and AVX2 instruction sets, and there is a GPU integrated into Intel Haswell CPU, which we can use it to do some tasks. Second, performance evaluation show that radix sort kernel does not fit both stage well, we must use more efficient GPU kernels in stage 2 which fully utilize the partial sorted chunks. Finally, we must make our algorithm more flexible, we need to improve our scheduling algorithm to smoothly fit in various cases and different machines.

References

[1] Martin, William A. Sorting. *ACM Computing Surveys (CSUR).* 1971; 3(4); 147-174.
[2] Chhugani, Jatin, et.al. *Efficient implementation of sorting on multi-core SIMD CPU architecture*. Proceedings of the VLDB Endowment. 2008; 1(2): 1313-1324.
[3] Djajadi, Arko, et.al. A model vision of sorting system application using robotic manipulator. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*. 2010; 8(2): 137-148.
[4] Hartati, Sri, Agus Harjoko, and Tri Wahyu Supardi. The digital microscope and its image processing utility. *TELKOMNIKA (Telecommunication Computing Electronics and Control).* 2011; 9(3); 565-574.
[5] West, Richard, et.al. Online cache modeling for commodity multicore processors. *ACM SIGOPS Operating Systems Review*. 2010; 44(4): 19-29.
[6] Arora, Krishan, Paramveer Singh Gill, and Parul Mehra. Design of high performance and low power simultaneous multi-threaded processor. *International Journal of Electrical and Computer Engineering (IJECE).* 2013; 3(3); 423-428.
[7] Chowdhury, Rezaul Alam, et.al. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*. 2013; 73(7): 911-925.
[8] Guide, Part. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* 2010.
[9] Nvidia CUDA. Nvidia cuda c programming guide. *Nvidia Corporation*. 2011.
[10] Sanders, Peter, and Sebastian Winkel. Super scalar sample sort.*Algorithms—ESA 2004.* 2004; 784-796.
[11] Chen, Shifu, et.al. A fast and flexible sorting algorithm with cuda. *Algorithms and Architectures for Parallel Processing.* 2009; 281-290.
[12] Inoue, Hiroshi, et.al. *AA-sort: A new parallel sorting algorithm for multi-core SIMD processors*. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. 2007; 189-198.
[13] Parberry, Ian. The pairwise sorting network. *Parallel Processing Letters*. 1992; 2(2, 3): 205-211.
[14] Satish, Nadathur, Mark Harris, and Michael Garland. *Designing efficient sorting algorithms for manycore GPUs.* Proceedings of Parallel & Distributed Processing (IPDPS 2009). 2009; 1-10.
[15] Satish, Nadathur, 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. 2010; 351-362.
[16] Brodtkorb, Andre R et.al. State-of-the-art in heterogeneous computing. *Scientific Programming*. 2010; 18(1): 1-33.
[17] Solomonik, Edgar, and Laxmikant V. Kale. *Highly scalable parallel sorting*. Proceedings of 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2010). 2010; 1-12.
[18] Augonnet, Cédric, et.al. Star PU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*. 2011; 23(2): 187-198.
[19] Francis, Rhys, Ian Mathieson, and Linda Pannan. *A fast, simple algorithm to balance a parallel multiway merge.* Proceedings of Parallel Architectures and Languages Europe (PARLE'93). Berlin, Heidelberg. 1993; 570-581.
[20] Batcher, Kenneth E. *Sorting networks and their applications.* Proceedings of the spring joint computer conference. 1968; 307-314.
[21] Merrill, Duane, Michael Garland, and Andrew Grimshaw. *Policy-based tuning for performance portability and library co-optimization.* Innovative Parallel Computing (InPar), 2012. 2012; 1-10.
[22] Francis, Rhys S and Ian D. Mathieson. A benchmark parallel sort for shared memory multiprocessors. *IEEE Transactions on Computers*. 1988; 37(12): 1619-1626.