AA-sort: A new parallel sorting algorithm for multi-core SIMD processors

Conference Paper *in* Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT · October 2007

DOI: 10.1109/PACT.2007.4336211 · Source: IEEE Xplore

4 authors, including:

Takao Moriyama
IBM
23 PUBLICATIONS **167** CITATIONS
SEE PROFILE

Hideaki Komatsu
IBM
95 PUBLICATIONS **1,316** CITATIONS
SEE PROFILE

AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors

Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu and Toshio Nakatani
IBM Tokyo Research Laboratory
{inouehrs, moriyama, komatsu, nakatani}@jp.ibm.com

Abstract

Many sorting algorithms have been studied in the past, but there are only a few algorithms that can effectively exploit both SIMD instructions and thread-level parallelism. In this paper, we propose a new parallel sorting algorithm, called Aligned-Access sort (AA-sort), for shared-memory multi processors. The AA-sort algorithm takes advantage of SIMD instructions. The key to high performance is eliminating unaligned memory accesses that would reduce the effectiveness of SIMD instructions. We implemented and evaluated the AA-sort on PowerPC® 970MP and Cell Broadband Engine™. In summary, a sequential version of the AA-sort using SIMD instructions outperformed IBM's optimized sequential sorting library by 1.8 times and GPUTeraSort using SIMD instructions by 3.3 times on PowerPC 970MP when sorting 32 M of random 32-bit integers. Furthermore, a parallel version of AA-sort demonstrated better scalability with increasing numbers of cores than a parallel version of GPUTeraSort on both platforms.

1. Introduction

Many modern high-performance processors provide multiple hardware threads within one physical processor with multiple cores and simultaneous multithreading. Many processors also provide a set of Single Instruction Multiple Data (SIMD) instructions, such as the SSE instruction set [1] or the VMX instruction set [2]. They can operate on multiple data values in parallel to accelerate computationally intensive programs for a broad range of applications.

An obvious advantage of the SIMD instructions is the degree of data parallelism available in one instruction. In addition, they allow programmers to reduce the number of conditional branches in their programs. Branches can potentially incur pipeline stalls and thus limit the performance of superscalar processors with long pipeline stages. For example, a program can select the smaller values from two vectors using vector compare and vector select instructions without conditional branches. The benefit of reduction in the number of conditional branches is significant for many workloads. For example, Zhou *et al*. [3] reported that SIMD instructions can accelerate many database operations by removing branch overhead.

Sorting is one of the most important operations for many commercial applications, especially database management systems. Hence many sequential and parallel sorting algorithms have been studied in the past [4, 5]. However popular sorting algorithms, such as quicksort, are not suitable for exploiting SIMD instructions. For example, a VMX instruction or a SSE instruction can load or store 128 bits of data between a vector register and memory with one instruction, but it is effective only when the data is aligned on a 128-bit boundary. Many sorting algorithms require unaligned or element-wise memory accesses, which incur additional overhead and attenuate the benefits of SIMD instructions. There is no known technique to remove unaligned memory accesses from quicksort.

In this paper, we propose a new parallel sorting algorithm suitable for exploiting both the SIMD instructions and thread-level parallelism available on today's multi-core processors. We call the new algorithm *Aligned-Access sort* (AA-sort). The AA-sort consists of two algorithms: an in-core sorting algorithm and an out-of-core sorting algorithm. Both algorithms can take advantage of the SIMD instructions and can also run in parallel with multiple threads.

The in-core sorting algorithm of the AA-sort extends combsort [6] and makes it possible to eliminate all unaligned memory accesses and fully exploit the SIMD instructions. The key idea to improve combsort is to first sort the input data into the transposed order, and then reorder it into the desired order. Its computational time is proportional to N•log(N) on average. Its disadvantages include poor memory access locality. Thus we used another sorting algorithm, an out-of-core sorting algorithm, to make it possible for the entire AA-sort to use the cache more efficiently.

The out-of-core algorithm is based on mergesort and employs our new vectorized merge algorithm. It has better memory access locality than our in-core algorithm. Its computational complexity is $O(N \cdot \log(N))$ even in the worst case.

The complete AA-sort algorithm first divides all of the data into blocks that fit in the L2 cache of each processor core. Next it sorts each block with the in-core sorting algorithm. Finally it merges the sorted blocks with our vectorized merge algorithm to complete the sorting. Both the sorting phase and the merging phase can be executed by multiple threads in parallel.

We implemented and evaluated the AA-sort on a system with 4 cores of the PowerPC® 970MP processors and a system with 16 cores of the Cell Broadband Engine™ (Cell BE) processors [7]. In summary, a sequential version of the AA-sort using SIMD instructions outperformed that of IBM's optimized sequential sorting library by 1.8 times and GPUTeraSort [8], an existing state-of-the-art sorting algorithm for SIMD processors, by 3.3 times on PowerPC 970MP when sorting 32 M of random 32-bit integers. Furthermore, a parallel version of the AA-sort demonstrated better scalability with increasing numbers of cores than a parallel version of the GPUTeraSort. It achieved a speed up of 12.2x by 16 cores on the Cell BE, while the GPUTeraSort achieved 7.1x. As a result the AA-sort was 4.2 times faster on 4 cores of PowerPC 970MP and 4.9 times faster on 16 cores of Cell BE processors compared to GPUTeraSort.

The main contribution of this paper is a new parallel sorting algorithm that can effectively exploit the SIMD instructions. It consists of two algorithms: an in-core sorting algorithm and an out-of-core sorting algorithm. In the in-core algorithm, it is possible to eliminate all unaligned memory accesses from combsort. For the out-of-core algorithm, we proposed a novel linear-time merge algorithm that can take advantage of the SIMD instructions. As far as the authors know, the AA-sort is the first sorting algorithm with computational complexity of $O(N \cdot \log(N))$ that can fully exploit the SIMD instructions of today's processors. We show that our AA-sort achieves higher performance and scalability with increasing numbers of processor cores than the best known algorithms.

The rest of the paper is organized as follows. Section 2 gives an overview of the SIMD instructions we use for sorting. Section 3 discusses related work. Section 4 describes the AA-sort algorithm. Section 5 discusses our experimental environment and gives a summary of our results. Finally, Section 6 draws conclusions.

2. SIMD instruction set

In this paper we use the Vector Multimedia eXtension [2] (VMX, also known as AltiVec) instructions of the PowerPC instruction set to present our new sorting algorithm. It provides a set of 128-bit vector registers, each of which can be used as sixteen 8-bit values, eight 16-bit values, or four 32-bit values. The following VMX instructions are useful for sorting: vector compare, vector select, and vector permutation.

The vector compare instruction reads from two input registers and writes to one output register. It compares each value in the first input register to the corresponding value in the second input register and returns the result of comparisons as a mask in the output register.

The vector select instruction takes three registers as the inputs and one for the output. It selects a value for each bit from the first or second input registers by using the contents of the third input register as a mask for the selection.

The vector permutation instruction also takes three registers as the inputs and one for the output. The instruction can reorder the single-byte values of the input arbitrarily. The first two registers are treated as an array of 32 single-byte values, and the third register is used as an array of indexes to pick 16 arbitrary bytes from the input register.

These instructions are not unique to the VMX instruction set. The SPE instruction set of Cell BE provides these instructions. The SSE instruction set of the IA32 also provides similar instructions in the current implementation or will provide them in a future instruction set [9].

3. Related work

Many sorting algorithms [4, 5] have been proposed in the past. The quicksort is one of the fastest algorithms used in practice, and hence there are many optimized implementations of quicksort available. However there is no known technique to implement quicksort algorithm using existing SIMD instructions.

Sanders and Winkel [10] pointed out that the performance of sorting on today's processors is often dominated by pipeline stalls caused by branch mispredictions. They proposed a new sorting algorithm named super-scalar sample sort (sss-sort), to avoid such pipeline stalls by eliminating conditional branches. They implemented the sss-sort by using the predicated instructions of the processor and showed that the sss-sort achieves up to 2 times higher performance over the STL sorting function delivered with gcc. Our algorithm can also avoid pipeline stalls caused by branch miss predictions. Moreover, our algorithm makes it possible to take advantage of data parallelism of SIMD instructions.

There are some sorting algorithms suitable for exploiting SIMD instructions [8, 11, 12]. They were originally proposed in the context of sorting on graphics processing units (GPUs), which are powerful programmable processors with SIMD instruction sets. Recent programmable GPUs are becoming increasingly close to multi-core general-purpose processors. Such programmable GPUs and multi-core processors with SIMD instructions show the same characteristics, such as the high thread-level parallelism and data parallelism of the SIMD engines, which we focus on in this paper.

Govindaraju *et al.* [8] presented a sorting architecture called GPUTeraSort. It included a new sorting algorithm for SIMD instructions that improved on Batcher's bitonic merge sort [13]. We refer to this algorithm, not the entire architecture, as the GPUTeraSort. The bitonic merge sort has computational complexity of $O(N \bullet \log(N)^2)$ and it can be executed by up to N processors in parallel. The GPUTeraSort improves this algorithm by altering the order of comparisons to improve the effectiveness of the SIMD comparisons and also increase the memory access locality to reduce cache misses. Comparing the AA-sort to the GPUTeraSort, both algorithms can be effectively implemented with SIMD instructions and both can exploit thread-level parallelism. An advantage of our AA-sort is smaller computational complexity of $O(N \bullet \log(N))$ compared to complexity of $O(N \bullet \log(N)^2)$ for the GPUTeraSort.

Furtak *et al.* [14] showed the benefit of exploiting SIMD instructions for sorting of very small arrays. They demonstrated that replacing only the last few steps of quicksort by sorting networks implemented with SIMD instructions improved the performance of the entire sort up to 22%. They evaluated the performance benefits for the SSE instructions and the VMX instructions. The AA-sort can take advantage of SIMD instructions not only in the last part of the sorting but also the entire stages.

4. AA-sort algorithm

In this section, we present our new sorting algorithm that we call AA-sort. We use 32-bit integers as the data type of the elements to be sorted. Hence one 128-bit vector register contains four values. 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 assume the first element of the array to be sorted is aligned on a 128-bit boundary and the

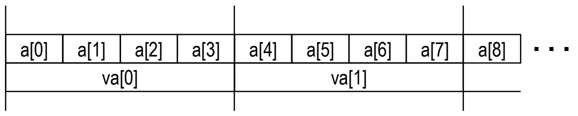


Figure 1. Data structure of the array.

```
gap = N / SHRINK_FACTOR;
while (gap > 1) {
  for (i = 0; i < N - gap; i++)
    if (a[i] > a[i+gap]) swap(a[i], a[i+gap]);
  gap /= SHRINK_FACTOR;
}
do {
  for (i = 0; i < N - 1; i++)
    if (a[i] > a[i+1]) swap(a[i], a[i+1]);
} while( not totally sorted );
```

Figure 2. Pseudocode of combsort.

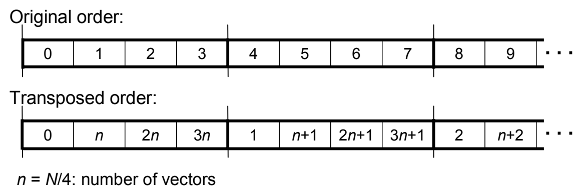
number of elements in the array, N, is a multiple of four for ease of explanation. Figure 1 illustrates the layout of the array, $a[N]$. The array of integer values $a[N]$ is equivalent to an array of vector integers $va[N/4]$. A vector integer element $va[i]$ consists of the four integer values of $a[i*4]$ to $a[i*4+3]$.

AA-sort consists of two algorithms, the in-core sorting algorithm and the out-of-core sorting algorithm. The overall AA-sort executes the following phases using the two algorithms: (1) Divide all of the data into blocks that fit into the cache of the processor. (2) Sort each block with the in-core sorting algorithm. (3) Merge the sorted blocks with the out-of-core algorithm. First we present these two sorting algorithms and then illustrate the overall sorting scheme.

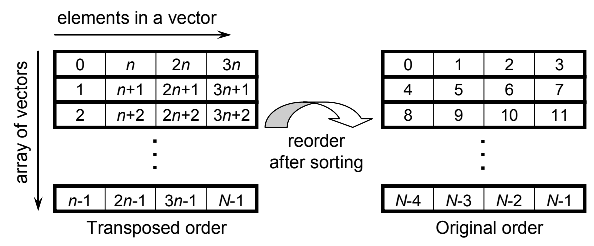
4.1. In-core algorithm

Our in-core algorithm of AA-sort improves on combsort [6], an extension to bubble sort. Bubble sort compares each element to the next element and swaps them if they are out of sorted order. Combsort compares and swaps two non-adjacent elements. Comparing two values with large separations improves the performance drastically, because each value moves toward its final position more quickly. Figure 2 shows the pseudocode of combsort. The separation (labeled *gap* in Figure 2) is divided by a number so called *shrink factor* in each iteration until it becomes one. The authors used 1.3 for the shrink factor. Then the final loop is repeated until all of the data is sorted. The computational complexity of combsort approximates $N \bullet \log(N)$ on average [6].

The fundamental operation of many sorting algorithms including combsort, bitonic merge sort, and GPUTeraSort, is to compare two values and swap them


Original order:
0 1 2 3 4 5 6 7 8 9 · · ·
Transposed order:
0 n 2n 3n 1 n+1 2n+1 3n+1 2 n+2 · · ·
n = N/4: number of vectors

(a) Original and transposed order in memory



(b) Schematic of reordering after the sorting

Figure 3. Transposed order.

if they are out of order. Each conditional branch in this operation will be taken in arbitrary order with roughly 50% probability for random input data, and therefore it is very hard for branch prediction hardware to predict the branches. This operation can be implemented using vector compare and vector select instructions without conditional branches.

Combsort has two problems that reduce the effectiveness of SIMD instructions: (1) unaligned memory accesses and (2) loop-carried dependencies. Regarding the unaligned memory accesses, combsort requires unaligned memory accesses when the value of the gap is not a multiple of the degree of data parallelism of the SIMD instructions. A loop-carried dependency prevents exploiting the data parallelism of the SIMD instructions when the value of the gap is smaller than the degree of data parallelism.

In our proposed in-core sorting algorithms, we resolved these issues of combsort. The key idea of our improvement is to once sort the values into the transposed order shown in Figure 3(a) and reorder the sorted values into the original order after the sorting as shown in Figure 3(b). Our in-core sorting algorithm executes the following 3 steps:

(1) sort values within each vector in ascending order,
(2) execute combsort to sort the values into the transposed order shown in figure 3(a), and then
(3) reorder the values from the transposed order into the original order as shown in Figure 3(b).

Step 1 sorts four values in each vector integer *va*[*i*]. This step corresponds to the loops with the gaps of *N*/4, *N*/4*2, and *N*/4*3 in combsort, because the gap between consecutive elements in one vector register is *N*/4 in the transposed order. This step can be implemented by using vector instructions.

```
gap = (N/4) / SHRINK_FACTOR;
while (gap > 1) {
  /* straight comparisons     */
  for (i = 0; i < N/4 - gap; i++)
      vector_cmpswap(va[i], va[i+gap]);
  /* skewed comparisons       */
  /* when i+gap exceeds N/4   */
  for (i = N/4 - gap; i < N/4; i++)
      vector_cmpswap_skew(va[i],
                          va[i+gap – N/4]);
  /* dividing gap by the shrink factor */
  gap /= SHRINK_FACTOR;
}
do {
  for (i = 0; i < N/4 - 1; i++)
      vector_cmpswap(va[i], va[i+1]);
  vector_cmpswap_skew(va[N/4-1], va[0]);
} while( not totally sorted );
```

Figure 4. Pseudocode of the Step 2.

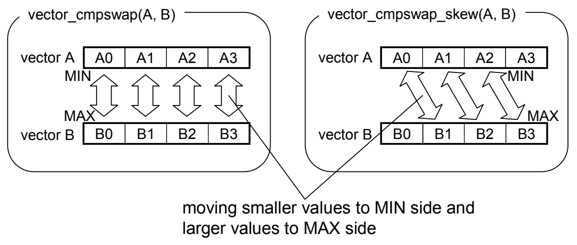


Figure 5. The vector_cmpswap and vector_cmpswap_skew operations.

Step 2 executes combsort on the vector integer array *va*[*N*/4] into the transposed order. Figure 4 shows pseudocode for the Step 2 of our in-core algorithm. In this code, *vector_cmpswap* is an operation that compares and swaps values in each element of the vector register A with the corresponding element of the vector register B as shown in Figure 5. Similarly *vector_cmpswap_skew* is an operation that compares and swaps the first to third elements of the vector register A with the second to fourth elements of the vector register B. It does not change the last element of the vector register A and the first element of the vector register B. Both operations can be implemented using SIMD instructions. Comparing the code of Figure 4 to the code of the original combsort in Figure 2, the inner-most loop is divided into two loops with these two operations. By these two loops, all of the values are compared and swapped with the values with the distance of the *gap* in the transposed order. The do-while loop in Figure 4 assures correct order of the output.

Step 3 reorders the sorted values into the correct order. This step does not require data-dependent conditional branches because it only moves each element in predefined orders, and hence reordering

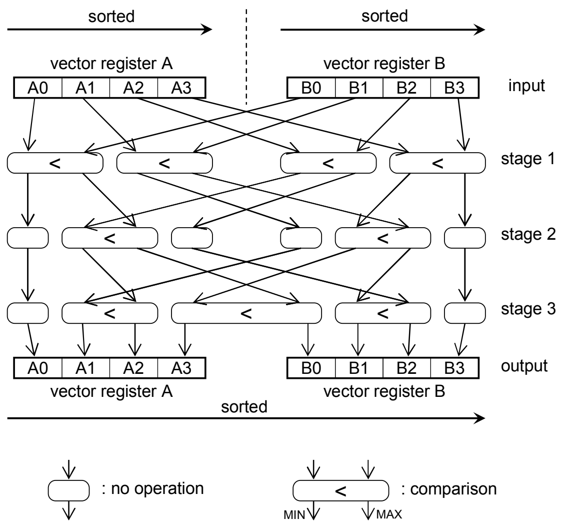


Figure 6. Data flow of vector merge operation.

does not incur any troublesome overhead. Vector permutation instructions can efficiently execute this step.

In summary, our in-core sorting algorithm consists of three steps. All three of the steps can be executed by SIMD instructions without unaligned memory accesses. Also all of them can be implemented with a negligible number of data-dependent conditional branches. The computational complexity of the entire in-core algorithm is dominated by Step 2 and is the same as that of combsort, approximating $O(N \cdot \log(N))$ on average and $O(N^2)$ in the worst case.

Our in-core sorting algorithm suffers from poor memory access locality. Thus its performance may degrade if the data cannot fit into the cache of the processor. We propose another sorting algorithm, the out-of-core sorting algorithm, which takes that problem into account.

4.2. Out-of-core algorithm

For the out-of-core sorting, we propose an innovative method to integrate odd-even merge algorithm [13] implemented with SIMD instructions into the usual merge algorithm. Our method make it possible for the merge operations to take advantage of SIMD instructions while still keeping the computational complexity of $O(N)$. This complexity is smaller than the complexity of $O(N \cdot \log(N))$ for the odd-even merge or the bitonic merge.

Figure 6 shows the data flow of the odd-even merge operation for eight values stored in the two vector registers, which contain four sorted values each. In the figure the boxes with inequality symbols signify comparison operations. Each of them reads two values

```
aPos = bPos = 0;
vMin = va[aPos++];
vMax = vb[bPos++];
while (aPos < aEnd && bPos < bEnd) {
  /* merge vMin and vMax */
  vector_merge(vMin, vMax);
  /* store the smaller vector as output*/
  vMergedArray[i] = vMin;
  /* load next vector and advance pointer  */
  /* a[aPos*4] is first element of va[aPos]*/
  /* and b[bPos*4] is that of vb[bPos]     */
  if (a[aPos*4] < b[bPos*4])
      vMin = va[aPos++];
  else
      vMin = vb[bPos++];
}
```

Figure 7. Pseudocode of the merge operation in memory.

from the two inputs (one each) and sends the smaller value to the left output and the larger one to the right. The odd-even merge operation requires $\log(P)+1$ stages to merge two vector registers, each of which contain P elements. Here $P=4$ and $\log(P)+1=3$. Each stage executes only one vector compare, two vector select and one or two vector permutation instructions. If an SIMD instruction set does not support a vector permutation operation, repeating a *vector_cmpswap* operation and a rotation of one vector register can substitute for the odd-even merge. However this requires P stages instead of $\log(P)+1$ stages.

The merge operation for two large arrays stored in memory can be implemented using this merge operation for the vector registers. Figure 7 shows the pseudocode for merging of two vector integer arrays *va* and *vb*. In this code, the *vector_merge* operation is the merge operation for the vector registers shown in Figure 6. In each iteration, this code

(1) executes a merge operation of two vector registers, *vMin* and *vMax*,
(2) stores the contents of *vMin*, the smallest four values, as output,
(3) compares the next element of each input array, and
(4) loads four values into *vMin* from the array whose next element is smaller and advances the pointer for the array.

Loading new elements from only one input array is sufficient, because the four data values in *vMax* are not larger than at least one of the next elements of the each input array and hence the larger of the two next elements must not be contained in the next four output values. There is only one conditional branch for the output of every P elements, while the naive merge operation requires one conditional branch for each output element.

Our out-of-core sorting algorithm recursively repeats the merge operation described earlier. It does not require any unaligned memory accesses. However, it achieves lower performance than our in-core algorithm for small amounts of data that can fit in the cache. On the other hand, the out-of-core sorting algorithm achieves higher performance than the in-core algorithm when data cannot fit in the cache. This is because the out-of-core algorithm has much better memory access locality compared to our in-core sorting algorithm.

4.3. Overall parallel sorting scheme of AA-sort

The overall AA-sort executes the following phases using the two algorithms:

(1) Divide all of the data to be sorted into blocks that fit in the cache or the local memory of the processor.

(2) Sort each block with the in-core sorting algorithm in parallel by multiple threads, where each thread processes an independent block.

(3) Merge the sorted blocks with the out-of-core sorting algorithm by multiple threads.

The block size for the in-core sorting is an important parameter. The selection of the block size depends on bandwidth and latency for each level of memory hierarchy. On the PowerPC 970MP processors, for example, half of the size of L2 cache was best for the block size because its L2 cache was fast enough to keep the processor core busy even there were many L1 cache misses.

If the number of elements of data is N and that the number of elements in one block is B, then the number of blocks for the in-core algorithm is (N/B). The computational time for the in-core sorting of each block is proportional to $B \bullet \log(B)$ on average and hence the total computational complexity of the in-core sorting phase is $O(N)$, since B is a constant. The sorting of each block is independent of the other blocks, so they can run in parallel on multiple threads up to the number of blocks. In the out-of-core sorting phase, merging the sorted blocks involves $\log(N/B)$ stages and the computational complexity of each stage is $O(N)$, and thus the total computational complexity of this phase is $O(N \bullet \log(N))$, even in the worst case. For parallelizing the last few stages of the out-of-core sorting, the number of blocks becomes smaller than the number of threads, and hence multiple threads must cooperate on one merge operation to fully exploit the thread-level parallelism [15].

The entire AA-sort has the computational complexity of $O(N \bullet \log(N))$ even in the worst case. Also it can be executed in parallel by multiple threads with complexity of $O(N \bullet \log(N)/k)$ assuming the number of threads, k, is smaller than the number of blocks, (N/B).

Table 1. Comparisons of algorithms

algorithm	SIMD	thread parallel	complexity	
			average	worst
AA-sort	Yes	Yes	$N \bullet \log(N)$	←
GPUTeraSort	Yes	Yes	$N \bullet \log(N)^2$	←
ESSL	No	No	$N \bullet \log(N)$	N^2
STL (introsort)	No	No	$N \bullet \log(N)$	←

4.4. Sorting of {key, data} pairs

In real-world workloads, sorting is mostly used to reorder data structures according to their keys. We can extend the AA-sort for such purposes. To that end, we consider sorting for pairs consisting of a key of 32-bit integer value and a 32-bit piece of associated data, such as a pointer to the data structure that contains the key. Assuming the keys and the attached data are stored in distinct arrays, the comparing and swapping operations can be implemented by using the result of the comparison for the key to move both the keys and the data. When the keys and attached data are stored in an array one after another, comparing and swapping of {key, data} pairs can be implemented by adding one vector permutation instruction after the vector compare instruction to replace the result of the comparison of the data with the result of the comparison of the keys. Hence the data always move with the associated keys in both cases.

5. Experimental results

We implemented the AA-sort and the GPUTeraSort for the PowerPC 970MP and the Cell BE with and without using the SIMD instructions. We used the GPUTeraSort for comparison because it is the best existing sorting algorithm for SIMD instructions. General purpose processors with SIMD instructions, however, are not the best platforms to execute the algorithm because it is originally designed for GPUs with much higher thread-level parallelism and memory bandwidth than the general purpose processors. We also evaluated two library functions, IBM's Engineering and Scientific Subroutine Library (ESSL) version 4.2 [16] and the STL library delivered with gcc implementing introsort [17], on the PowerPC 970MP. Table 1 summarizes characteristics of each algorithm.

The PowerPC 970MP system used for our evaluation was equipped with two 2.5 GHz dual-core PowerPC 970MP processors and 8 GB of system memory. In total, the system had 4 cores, each of which had 1 MB of L2 cache memory. The Linux kernel 2.6.20 was running on the system. We also evaluated the performance of the sorting programs on a system equipped with two 2.4 GHz Cell BE processors with 1 GB of system memory. The Cell BE is an asymmetric multi-core processor that combines a PowerPC core with eight accelerator cores called SPEs. We used only the SPE cores for sorting. Thus, 16 SPE cores with 256 KB local memory each were available on the system. The Linux kernel 2.6.15 was running on the system.

5.1. Implementation details

The programs for the PowerPC 970 were written in C using the AltiVec intrinsics [18]. We compiled all of the programs with the IBM XL C/C++ compiler for Linux v8. The programs for the Cell BE were also written in C using the intrinsics for SPE [19]. We compiled our programs with the IBM XL C Compiler for SPE. All of the programs used the memory with a 16 MB page size to reduce the overhead of TLB handling on both platforms.

In the implementations of AA-sort, we selected a half of the size of L2 cache or local memory as the block size for the in-core sorting phase, 512 KB (128 K of 32-bit values) on the PowerPC 970MP and 128 KB (32 K of 32-bit values) on the SPE. The shrink factor for our in-core sorting algorithm was 1.28. We chose these parameters based on our measurements.

We used some techniques to reduce the required bandwidth for system memory. The experimental implementations of the AA-sort used a multi-way merge technique [5, 20] with our out-of-core sorting. We employed a 4-way merge; input data was read from 4 streams and output into a merged output stream. It reduced the number of merging stages from $\log_2(N/B)$ to $\log_4(N/B)$. Our implementation of the GPUTeraSort for the Cell BE reduced the amount of data read from system memory by directly copying data from the local memory of another SPE core instead of from system memory whenever possible. It can benefit from huge bandwidth of the on-chip bus of the Cell BE.

In our parallel implementation of the AA-sort, all of the threads first execute in-core sorting and then move onto a merging phase after all of the blocks of input data are sorted. When executing our out-of-core sorting with multiple threads, each thread executes independent merge operations as long as there are enough blocks to merge. In the last few stages, the number of blocks becomes smaller than the number of threads, and hence multiple threads must cooperate on one merge operation. Our implementation first divides one input stream into chunks of equal size for each thread, and then finds a corresponding starting point and finishing point for another input stream by executing binary search. Additionally it executes rebalancing of the data among threads if the data size for each thread is not balanced.

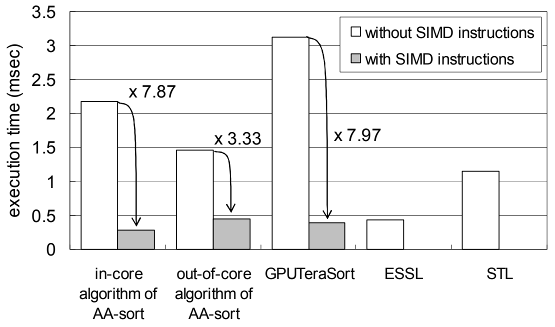


Figure 8. Acceleration by SIMD instructions for sorting 16 K random integers on one core of PowerPC 970MP

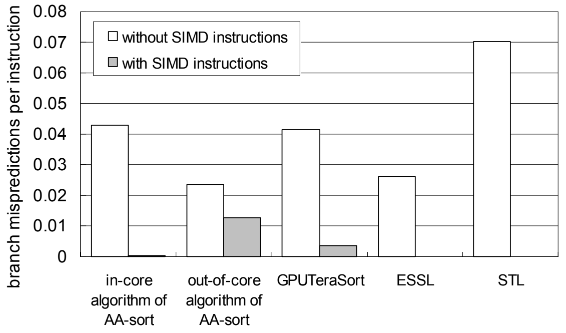


Figure 9. Branch misprediction rate.

5.2. Effects of using SIMD instructions

This section focuses on the performance of sequential implementations of each algorithm with great emphasis on the effect of SIMD instructions. We use both the in-core sorting algorithm and the out-of-core sorting algorithm of the AA-sort separately to illustrate the effect of SIMD instructions for each algorithm. Note that the out-of-core sorting algorithm is not used with such small amount of data when executing the entire AA-sort.

Figure 8 compares the performance of the sorting algorithms for 16 K of random 32-bit integers using only one PowerPC 970MP core. All of the data to be sorted can fit into the L2 cache of the processor. The performance of our in-core algorithm, out-of-core algorithm and the GPUTeraSort was drastically improved by using the SIMD instructions and the in-core sorting algorithm with SIMD instructions achieved the highest performance among all of the algorithms tested. We also implemented and evaluated the original combsort using SIMD instructions. The degree of acceleration by SIMD instructions for the original combsort was only 2.2. It was not significant compared to our in-core algorithm because the benefit of the SIMD instructions was reduced by unaligned memory access and loop-carried dependencies.

The degrees of acceleration with the SIMD instructions for our in-core algorithm and the GPUTeraSort were larger than the degree of parallelism available by the SIMD instructions (4x) due to reduced number of branch mispredictions. Figure 9 shows the branch misprediction rate measured by using a performance monitor counter of the processor. The branch misprediction rates were reduced by more than a factor of 10 for our in-core algorithm and the GPUTeraSort. The change of misprediction rate was smaller for our out-of-core algorithm because data-dependent conditional branches were reduced but not totally eliminated.

Table 2 shows a breakdown of the performance gain with SIMD instructions shown in Figure 8 into two reasons: reduction in numbers of instructions and improvements in cycles per instruction (CPI). The reduction in numbers of instructions was mainly owing to data parallelism of the SIMD instructions and the CPI improvements were due to reduced branch overhead. For our in-core algorithm and GPUTeraSort, the numbers of instructions were reduced almost in proportional to the degree of data parallelism available by the SIMD instructions, while the reduction was not significant for our out-of-core algorithm. This is because the vectorized merge for vector registers shown in Figure 6 is more complicated and requires more instructions than the naive merge operation for scalar values.

5.3. Performance for 32-bit integers

In this section, we discuss the performance of sorting for large 32-bit integer arrays. Figure 10 compares the performance of sequential versions of four algorithms on the PowerPC 970MP. The AA-sort and GPUTeraSort were implemented with SIMD instructions. The x-axis shows the number of elements up to 128 M elements (512 MB) and the y-axis shows the execution time. The AA-sort achieved the best result among all algorithms for every data size. It was faster by 1.8 times than the ESSL and by 3.0 times than the STL when sorting 32 M integers. It also surpassed the performance of the GPUTeraSort by 3.3 times. The performance advantage of the AA-sort over the GPUTeraSort became larger with a larger data size because of the larger computational complexity of the GPUTeraSort.

Figure 11 illustrates how the performance of each algorithm depends on four input datasets as shown in Table 3. The AA-sort and the GPUTeraSort showed obviously much smaller dependence on the input

Table 2. Breakdown of performance gain

algorithm	speed up by SIMD	=	reduction in instructions†	x	improvement in CPI‡
in-core algorithm	7.87		4.06		1.94
out-of-core algorithm	3.33		2.92		1.14
GPUTeraSort	7.97		4.69		1.70

† *reduction in instructions* $= instruction\_count_{scalar} / instruction\_count_{SIMD}$

‡ *improvement in CPI* $= CPI_{scalar} / CPI_{SIMD}$

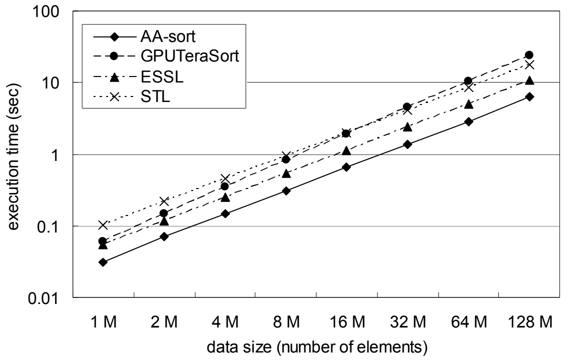


Figure 10. Performance of sequential version of each algorithm on a PowerPC 970MP core for sorting random 32-bit integers with various data sizes.

dataset than the other two algorithms. This is because the two implementations with SIMD instructions did not suffer from branch mispredictions even for the random input. The performance of the ESSL and the STL degraded severely for some cases. Our in-core algorithm may also show poor performance for some datasets, since it uses a heuristic approach. Our out-of-core algorithm, however, does not show catastrophic performance even for the worst case. As a result, overall the AA-sort also does not depend too much on the input data, because the input size for the in-core algorithm is limited to the block size.

Figure 12 shows the execution time of parallel versions of the AA-sort and the GPUTeraSort on 1, 2, and 4 PowerPC 970MP cores for 32 M random integers. It also shows the performance of the ESSL and the STL on only one core. The AA-sort achieved larger speed up by using multiple cores than GPUTeraSort. As a result, the performance of the AA-sort was 4.2 times as high as the performance of the GPUTeraSort with 4 cores of PowerPC 970MP. To see the scalability with larger numbers of cores, Figure 13 shows the scalability of both algorithms on the Cell BE up to 16 cores when sorting 32 M of 32-bit integers. Both algorithms showed almost linear speed up for up to 4 cores. With more than 4 cores, our AA-sort

Table 3. Description of datasets.

dataset	description	pseudocode of initialization
A	random	`for (i=0; i<N; i++) { data[i] = random32(); }`
B	almost presorted	`for (i=1; i<N; i++) { data[i] = i; } data[0] = N;`
C	presorted (forward)	`for (i=0; i<N; i++) { data[i] = i; }`
D	presorted (reversed)	`for (i=0; i<N; i++) { data[i] = N-i; }`

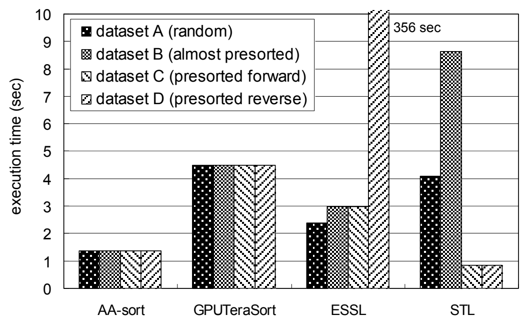


Figure 11. Performance comparison on one PowerPC 970MP core for various input datasets with 32 million integers.

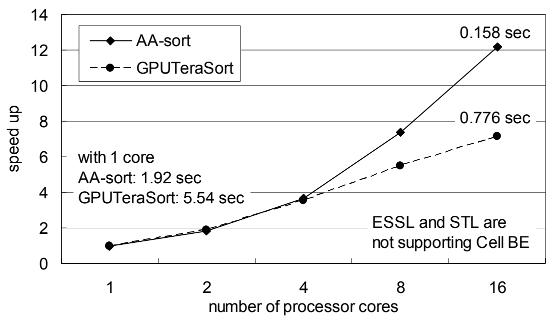


Figure 13. Scalability with increasing number of cores on Cell BE for 32 million integers.

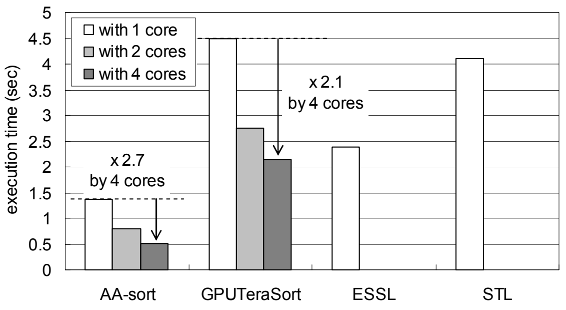


Figure 12. The execution time of parallel versions of AA-sort and GPUTeraSort on up to 4 cores of PowerPC 970MP.

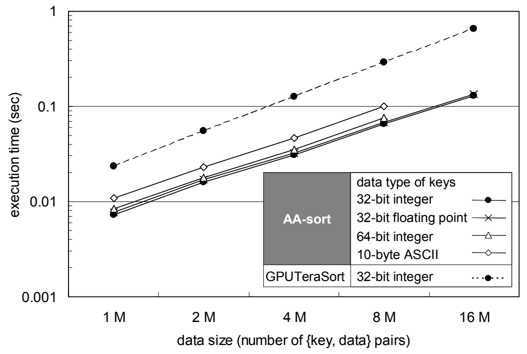


Figure 14. Performance comparisons of the data type of the key for sorting {key, data} pairs on the Cell BE using 16 cores.

demonstrated better scalability than the GPUTeraSort. For example the AA-sort achieved a speed up of 12.2x for 16 cores while the GPUTeraSort achieved 7.1x. This was due to the fact that the GPUTeraSort has a higher communication/computation ratio than the AA-sort and the memory bandwidth was a bottleneck that limited the scalability. The GPUTeraSort requires higher memory bandwidth because it assumes huge memory bandwidth of GPUs. As a result, the performance of the AA-sort was better than the GPUTeraSort by 4.9 times with 16 cores of the Cell BE.

5.4. Performance for {key, data} pairs

This section focuses on sorting for pairs of key and associated data such as a pointer to the structure having that key value. Figure 14 shows the sorting times of the AA-sort and the GPUTeraSort for sorting pairs with various data types for keys while using 16 cores on the Cell BE. The x-axis shows the number of elements up to 16 M pairs (128 MB). In the measurements the keys and the attached data are stored in distinct arrays. The tested data types of the keys included single-precision floating-point values, 64-bit integers, and 10-byte ASCII strings. The floating point keys and the integer keys were initialized using random numbers. For the ASCII string keys, we used the input data generator of the Sort Benchmark [21] to initialize the keys, and sorted them into the order of the strnicmp() function.

Our implementations for wider keys, 64-bit integers and 10-byte ASCII strings, employed the hybrid approach of our algorithm and radixsort. Govindaraju *et al.* [8] also used a similar hybrid approach for the improved bitonic merge sort and radixsort in the GPUTeraSort. First it extracts the first few bytes from the keys and encodes them into 32-bit integer values, then sorts the pairs according to the encoded keys. After sorting by the first few bytes, when and only when multiple pairs have the same encoded keys, the pairs having the same encoded key are sorted using the next few bytes. The results shown in Figure 14 include the time for key extraction and encoding. The input for our sorting function was pairs of {key, pointer}, and the output was a sorted array of pointers. The performance of the AA-sort for sorting 16 M pairs with random integer keys was about 1.6 times slower than that for sorting 16 M of simple 32-bit integer values. However the AA-sort achieved up to 5.0 times faster results than the GPUTeraSort for the {key, pointer} pairs with 32-bit integer keys. For the wider keys, the performance was slightly degraded due to the overhead of the key encoding and repeated sorting. Even the slowest case with the AA-sort, for the keys of 10-byte ASCII strings, was much faster than the GPUTeraSort for the pairs with 32-bit integer keys.

6. Conclusions

This paper describes a new parallel sorting algorithm that we call Aligned-Access sort (AA-sort). The AA-sort is suitable for exploiting both the SIMD instructions and thread-level parallelism available on today's multi-core processors. The AA-sort does not involve any unaligned memory accesses that attenuate the benefit of SIMD instructions, and hence it can effectively exploit the SIMD instructions. We implemented and evaluated the AA-sort on PowerPC 970MP and Cell Broadband Engine processors. In summary, a sequential version of the AA-sort using SIMD instructions outperformed that of IBM's ESSL by 1.8 times and the GPUTeraSort using SIMD instructions by 3.3 times on the PowerPC 970MP when sorting 32 M of random 32-bit integers. Furthermore, a parallel version of the AA-sort demonstrated better scalability with increasing numbers of cores than a parallel version of the GPUTeraSort. The AA-sort achieved speed up of 12.2x by 16 cores on the Cell BE, while the GPUTeraSort achieved 7.1x. As a result the AA-sort was 4.2 times faster on 4 cores of the PowerPC 970MP and 4.9 times faster on 16 cores of the Cell BE processors compared to the GPUTeraSort.

7. References

[1] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual*.

[2] IBM Corp. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*.

[3] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pp. 145–156, 2002.

[4] W. A. Martin. Sorting. *ACM Computing Surveys*, 3(4), pp. 147–174, 1971.

[5] D. E. Knuth. *The Art of Computer Programming. Vol. 3: Sorting and Searching*, 1973.

[6] S. Lacey and R. Box. *A Fast, Easy Sort. Byte Magazine*, April, pp. 315–320, 1991.

[7] D. Pham *et al*. The Design and Implementation of a First-Generation CELL Processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pp. 184–185, 2005.

[8] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 325–336, 2006.

[9] R. M. Ramanathan. Extending the World's Most Popular Processor Architecture. *Technology@intel Magazine*, 2006.

[10] P. Sanders and S. Winkel. Super Scalar Sample Sort. In *Proceedings of the European Symposium on Algorithms*, volume 3221 of LNCS, pp. 784–796, 2004.

[11] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of ACM SIGGRAPH/ EUROGRAPHICS Workshop On Graphics Hardware*, pp. 41–50, 2003.

[12] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 611–622, 2005.

[13] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, 32, pp. 307–314, 1968.

[14] T. Furtak, J. N. Amaral, and R. Niewiadomski. Using SIMD Registers and Instructions to Enable Instruction-Level Parallelism in Sorting Algorithms. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 348–357, 2007.

[15] R. Francis, and I. Mathieson. A Benchmark Parallel Sort for Shared memory Multiprocessors, *IEEE Transactions on Computers*, 37(12), pp. 1619–1626, 1988.

[16] IBM Corp. *Engineering Scientific Subroutine Library (ESSL) and Parallel ESSL*, http://www-03.ibm.com/systems/p/software/essl.html.

[17] D. R. Musser. Introspective Sorting and Selection Algorithms, *Software Practice and Experience*, 27(8), pp. 983-993, 1997.

[18] Freescale Semiconductor Inc. *AltiVec Technology Programming Interface Manual*, 1999.

[19] IBM Corp., Sony Computer Entertainment Inc., and Toshiba Corp. *SPU C/C++ Language Extensions*, 2005.

[20] T. Nakatani, S. T. Huang, B. W. Arden, and S. K. Tripathi. K-Way Bitonic Sort, *IEEE Transactions on Computers*, 38(2), pp. 283–288, 1989.

[21] Sort Benchmark, http://research.microsoft.com/barc/SortBenchmark.