SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **42**:753–777
Published online 19 July 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1102

A high-performance sorting algorithm for multicore single-instruction multiple-data processors

Hiroshi Inoue*,†, Takao Moriyama, Hideaki Komatsu and Toshio Nakatani

IBM Research, Tokyo, Japan

SUMMARY

Many sorting algorithms have been studied in the past, but there are only a few algorithms that can effectively exploit both single-instruction multiple-data (SIMD) instructions and thread-level parallelism. In this paper, we propose a new high-performance sorting algorithm, called aligned-access sort (AA-sort), that exploits both the SIMD instructions and thread-level parallelism available on today's multicore processors. Our algorithm consists of two phases, an in-core sorting phase and an out-of-core merging phase. The in-core sorting phase uses our new sorting algorithm that extends combsort to exploit SIMD instructions. The out-of-core algorithm is based on mergesort with our novel vectorized merging algorithm. Both phases can take advantage of SIMD instructions. The key to high performance is eliminating unaligned memory accesses that would reduce the effectiveness of SIMD instructions in both phases. We implemented and evaluated the AA-sort on PowerPC 970MP and Cell Broadband Engine platforms. In summary, a sequential version of the AA-sort using SIMD instructions outperformed IBM's optimized sequential sorting library by 1.8 times and bitonic mergesort using SIMD instructions by 3.3 times on PowerPC 970MP when sorting 32 million random 32-bit integers. Also, a parallel version of AA-sort demonstrated better scalability with increasing numbers of cores than a parallel version of bitonic mergesort on both platforms. Copyright © 2011 John Wiley & Sons, Ltd.

Received 20 June 2010; Revised 25 April 2011; Accepted 17 May 2011

KEY WORDS: sorting; merging; SIMD; VMX; parallel algorithms

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 single-instruction multiple-data (SIMD) instructions, such as the Streaming SIMD Extensions (SSE) instruction set of the x86 or the vector multimedia extension (VMX) instruction set of the PowerPC. 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. For example, a program can select the smaller or larger value from each element's pair of two vectors without conditional branches. Branches can potentially incur pipeline stalls and thus limit the performance of superscalar processors with long pipeline stages. Therefore, the benefit of reduction in the number of conditional branches is significant for many workloads. For example, Zhou and Ross [1] reported that SIMD instructions can accelerate many database operations, such as scan operations and join operations, by removing branch overhead.

*Correspondence to: Hiroshi Inoue, IBM Research, Tokyo, Japan.
†E-mail: inouehrs@jp.ibm.com

Sorting is one of the most important building blocks for operating systems and many commercial and scientific applications, such as database management systems [2]. Hence, many sequential and parallel sorting algorithms have been studied in the past [3, 4]. However, SIMD instructions in today's processors have limitations, and popular sorting algorithms, such as quicksort, are not suitable to exploit SIMD instructions efficiently. For example, a VMX instruction or an SSE instruction can load or store 128 bits of data between a vector register and memory with one instruction, but this 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.

In this paper,[‡] we propose a new high-performance sorting algorithm suitable for exploiting both the SIMD instructions and thread-level parallelism available on today's multicore processors. We call the new algorithm aligned-access sort (AA-sort). The AA-sort consists of two phases: an in-core sorting phase and an out-of-core merging phase. Both phases can take advantage of the SIMD instructions and can also run in parallel with multiple threads.

The in-core sorting phase uses our new algorithm that extends combsort [5] to exploit SIMD instructions. This 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 a transposed order using vector comparisons, and then reorder it into the desired order. The computational complexity for both the combsort and our vectorized combsort is $O(N \cdot \log(N))$[§] on average and $O(N^2)$ in the worst case when sorting N elements. Disadvantages of the vectorized combsort include poor memory access locality, so we combine it with another sorting algorithm in the out-of-core merging phase to make it possible for the entire AA-sort to use the cache more efficiently.

The out-of-core merging phase 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 in the in-core sorting phase. Finally, it merges the sorted blocks with our vectorized merge algorithm to complete the sorting in the out-of-core merging phase. Both phases can be executed by multiple threads in parallel. The entire AA-sort has the computational complexity of $O(N \cdot \log(N))$. In addition, it can be executed in parallel by multiple threads assuming that the number of threads is smaller than the number of blocks for the in-core phase.

We implemented and evaluated the AA-sort on a system with four cores of the PowerPC 970MP processor and a system with 16 cores of the Cell Broadband Engine (Cell BE) processor [7]. In summary, a sequential version of the AA-sort using SIMD instructions outperformed IBM's optimized sequential sorting library by 1.8 times and the bitonic mergesort that uses SIMD instructions, the best existing sorting algorithm for SIMD processors, by 3.3 times on the PowerPC 970MP when sorting 32 million random 32-bit integers. The performance of the AA-sort did not depend on key distributions of the input data by eliminating the data-dependent conditional branches. Moreover, a parallel version of the AA-sort demonstrated better scalability with increasing numbers of cores than a parallel version of the bitonic mergesort. It achieved a speedup of 12.2 for 16 cores on the Cell BE, whereas the bitonic mergesort achieved a speedup of 7.1. As a result, the AA-sort was 4.2 times faster on four cores of the PowerPC 970MP and 4.9 times faster on 16 cores of the Cell BE processor than the bitonic mergesort when sorting 32 million random 32-bit integers.

The main contribution of this paper is a new high-performance sorting algorithm that can effectively exploit SIMD instructions. It consists of two algorithms: a vectorized combsort and a vectorized mergesort. In our vectorized combsort, it is possible to eliminate all unaligned memory accesses from combsort. For the vectorized mergesort, we proposed a novel linear-time merge algorithm that can take advantage of the SIMD instructions. We show that our AA-sort achieves

[‡]A preliminary version of this paper was published in the proceedings of the 16th IEEE Parallel Architecture and Compilation Techniques (PACT 2007) [6]. This paper adds more descriptions of our new algorithm. It also includes more detailed analysis of the results of our measurements, including the effects of important parameters on performance.

[§]log refers to logarithm with base 2 unless a different value is specified.

DOI: 10.1002/spe

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 that 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. SINGLE-INSTRUCTION MULTIPLE-DATA INSTRUCTION SET

In this paper, we use the VMX [8] (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 16 8-bit values, 8 16-bit values, or 4 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, and thus, our algorithm can be implemented using other SIMD instruction sets, such as the synergistic processing element (SPE) instruction set of Cell BE and the SSE instruction set of the x86. We present an implementation of our algorithm on Cell BE in this paper, and Chhugani *et al.* [9] described an implementation of a part of our algorithm using the SSE instruction set [10].

3. RELATED WORK

Many sorting algorithms have been proposed in the past. Quicksort is one of the fastest practical algorithms, and hence, there are many optimized implementations of quicksort available. However, its scattered memory accesses make it difficult for quicksort to efficiently exploit the SIMD instructions of today's processors. There are other sorting algorithms that exploit SIMD instructions and thread-level parallelism. They were originally proposed for sorting on graphics processing units (GPUs), which are powerful programmable processors with SIMD instruction sets.

Pioneering projects that sorted on GPUs [11–13] used sorting networks. In particular, the bitonic mergesort proposed by Batcher [14] was most widely used as the base algorithm. Bitonic mergesorts, or any other sorting networks, always compare values in a predefined order regardless of the input values. This characteristic of the sorting network helps in implementing the algorithms using SIMD instructions. The bitonic mergesort has computational complexity of $O(N\cdot(\log(N))^2)$, and it can be executed by up to N processors in parallel. For example, Govindaraju *et al.* [13] presented a sorting algorithm called GPUTeraSort, which improved the bitonic mergesort by altering the order of comparisons to improve the effectiveness of the SIMD comparisons and also by increasing the memory access locality. 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 the computational complexity of $O(N\cdot\log(N))$, which is the optimal complexity for any comparison-based sorting algorithm, while the complexity for the GPUTeraSort is $O(N\cdot(\log(N))^2)$. Gedik *et al*. [15] presented a sorting algorithm for Cell BE called the CellSort. They also used the bitonic mergesort as their computing kernel to exploit the SIMD instruction set and thread-level parallelism of the processor. The resulting computational complexity of their algorithm was again larger than ours.

DOI: 10.1002/spe

Furtak *et al*. [16] showed the benefits of exploiting SIMD instructions for sorting very small arrays. They demonstrated that replacing only the last few steps of quicksort by a sorting network implemented with SIMD instructions improved the performance of the entire sort by 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 for entire stages. Later, Furtak [17] proposed a technique to implement quicksort using the SSE instructions. The proposed technique can reduce the number of comparison instructions and also avoid some load and store instructions. However, the performance of this technique depends on the distributions of the input data because of the complicated data movements.

After our initial paper on the AA-sort [6] was published, many research projects focused on efficient sorting on multicore SIMD processors or GPUs. Most of them did not use bitonic mergesort to improve the computational complexity.

Cederman and Tsigas [18] demonstrated that their quicksort implementation on recent NVIDIA GPUs achieved much better performance than quicksort on general-purpose CPUs or the GPUTeraSort running on the same GPUs. Their quicksort for GPUs exploits the flexible memory access mechanisms of the recent GPUs. With these GPUs, each slot of a vector load or store instruction can access an arbitrary memory address, whereas the corresponding VMX or SSE instructions can only access properly-aligned contiguous 128-bit blocks of data. Using this flexible memory access, each slot of the GPU's vector instructions can act as a separate thread. NVIDIA calls this processor architecture SIMT (single-instruction, multiple-thread) in contrast to the traditional SIMD [19]. Our AA-sort targets the SIMD processors, which have more limitations than the SIMT processors.

Sintorn and Assarsson [20] proposed a hybrid approach of bucketsort and mergesort for GPUs. They used bucketsort to provide enough thread-level parallelism for mergesort. Although developed independently, their technique for exploiting the GPU's SIMD instructions for mergesort was similar to ours. They use bitonic merge implemented with SIMD instructions as the kernel and integrated it with a traditional merge operation. Their algorithm outperformed radixsort implemented on the same GPU because of its smaller memory bandwidth requirement. Chhugani *et al.* [9] implemented a hybrid merge algorithm consisting of a vectorized bitonic merge network and a traditional merge operation, which is also very similar to ours, on a quad-core Intel Xeon processor using SSE instructions. They increase the instruction-level parallelism by executing multiple merge operations on one processor core and by using bitonic merge networks larger than the vector register size.

Satish *et al.* [21] proposed a technique to improve radixsort on GPUs by reducing scatter operations to a global memory that reorders all of the data to be sorted. They reported that their radixsort implemented on NVIDIA GPUs outperformed the bitonic mergesort or mergesort implemented on the same GPU or high-performance sorting algorithms on CPUs. Later, Satish *et al.* [22] conducted a detailed analysis of mergesort and radixsort on the latest CPUs and GPUs, where radixsort outperformed mergesort on both. The down side was that radixsort required more memory bandwidth and, hence, mergesort may surpass radixsort on future processors with more cores or wider SIMD capabilities. Our AA-sort is aligned with their conclusion because we use mergesort for out-of-core sorting and a multi-way merge technique to reduce the memory bandwidth. In addition, we enhance the overall sorting performance by using a vectorized combsort for in-core sorting. As shown later, using vectorized combsort as the in-core algorithm improved the performance of the vectorized mergesort by more than 20% on a PowerPC 970MP core.

Because the memory bandwidth tends to become a bottleneck in systems with Cell BE processors, Keller and Kessler [23] focused on reducing the main memory bandwidth requirements in the out-of-core phase of our algorithm. They reduced the memory bandwidth requirements by a factor of 2.5 by directly copying data from the local memory of other SPE cores to exploit the huge on-chip bandwidth of the Cell BE. The current implementation of the AA-sort uses a multi-way merge technique to reduce memory bandwidth and achieves almost linear speedup with up to eight cores, though the speedup ratio declines with 16 cores. With their proposed technique, our AA-sort may achieve better scalability with larger numbers of cores in exchange for more complex data movement and process scheduling.

We achieved a large performance improvement by reducing the branch mispredictions using SIMD instructions. Sanders and Winkel [24] also pointed out that the performance of sorting was

DOI: 10.1002/spe

often dominated by pipeline stalls caused by branch mispredictions. They proposed a new sorting algorithm, super-scalar sample sort (sss-sort), to avoid pipeline stalls by eliminating conditional branches. They implemented sss-sort by using the predicated instructions of the processor and showed that sss-sort achieves up to 2 times higher performance over the Standard Template Library sorting function delivered with GNU C Compiler (gcc). Our algorithm also avoids pipeline stalls caused by branch mispredictions while making it possible to take advantage of the data parallelism of SIMD instructions.

4. ALIGNED-ACCESS SORT ALGORITHM

In this section, we present our new sorting algorithm called 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. 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 $a[i*4+3]$.

For ease of explanation, we assume that the first element of the array to be sorted is aligned on a 128-bit boundary and the number of elements in the array, N, is a multiple of the degree of data parallelism of the SIMD instructions. We can handle sorting of an array whose head or tail is not aligned on a 128-bit boundary without adding significant overhead. For example, when the first element of the array to be sorted is stored in the second slot of a vector, one of the easiest solutions is to replace the value in the first slot with the minimum possible value, for example, 0 for an unsigned integer, before sorting and writing back the original value after all of the values in the array are sorted.

Aligned-access sort consists of two algorithms, a vectorized combsort sort and a vectorized mergesort. 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 and sort each block with the vectorized combsort (in-core sorting phase).
2. Merge the sorted blocks with the vectorized mergesort (out-of-core merging phase).

First, we present these two vectorized sorting algorithms and then illustrate the overall sorting scheme.

4.1. Vectorized combsort

Our vectorized combsort improves on combsort [5], an extension to bubble sort. Figure 2 shows the pseudocode of combsort. Bubble sort compares each element to the next element and swaps them if they are not in sorted order. Combsort compares and swaps non-adjacent elements. The separation (labeled *gap* in Figure 2) between the two values to compare is constant in each iteration and is divided by a number, the *shrink factor*, in each iteration until it becomes one. Comparing two values with large separations improves the performance drastically because each value moves toward its final position more quickly. The authors used 1.3 for the shrink factor. A good value for the shrink factor is quite important for the overall sorting performance in the original combsort and also in our vectorized combsort. We use a variation of combsort called combsort11 [5], which sets the separation to 11 when it becomes 9 or 10 because the sequence of separation values starting with 11 is much more effective in sorting all of the data than the sequences starting from 9 or 10. Then,

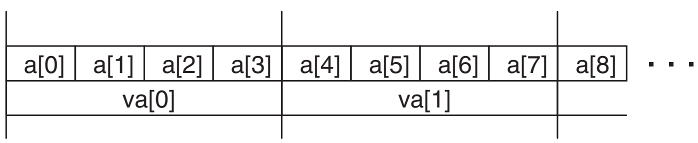

Figure 1. Data structure of the array to be sorted.

Softw. Pract. Exper. 2012; **42**:753–777
DOI: 10.1002/spe

```
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;
  if (gap == 9 || gap == 10) gap = 11;
}
do {
  for (i = 0; i < N - 1; i++)
    if (a[i] > a[i+1]) swap(a[i], a[i+1]);
} while( not fully sorted );
```

Figure 2. Pseudocode of combsort.

the final loop is repeated until all of the data is sorted. The computational complexity of combsort approximates $N \cdot \log(N)$ on average [5].

The fundamental operation of many sorting algorithms including combsort and bitonic mergesort is to compare two values and swap them 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: (i) unaligned memory accesses and (ii) 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 exploitation of the data parallelism of the SIMD instructions when the value of the gap is smaller than the degree of data parallelism.

In our vectorized combsort, we resolved these problems with combsort. The key idea of our improvement is to first sort the values into the *transposed* order and reorder the sorted values into the original order after the sorting. Here are the three steps of our vectorization technique for combsort:

1. Sort the values within each vector.
2. Use combsort to sort the values into the transposed order.
3. Then, reorder the values from the transposed order into the original order.

Figure 3 visualizes an example of our combsort for an array with 16 integers (or four vector integers).

Step 1 sorts the four values in each vector integer $va[i]$ ($0 \leqslant i < N/4$, where $N = 16$ in Figure 3). 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. The sorting for values in a vector register can be implemented as a sorting network using vector comparison and vector permutation instructions [16].

Step 2 executes combsort on the vector integer array $va[N/4]$ in the transposed order. Figure 4 shows pseudocode for our vectorized combsort 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. This operation can be implemented using a pair of vector minimum and maximum instructions or one vector-compare instruction and two vector-select instructions. 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 nor the first element of the vector register B. We use vector_cmpswap_skew operation to compare values when the comparison wraps around the $N/4$ boundary in the transposed order. For example, of the right-bottom figure in Figure 3, comparing values shown as 0 and 2 are compared by using a vector_cmpswap operation, and 2 and 4 are compared by using a vector_cmpswap_skew operation. Both operations can be

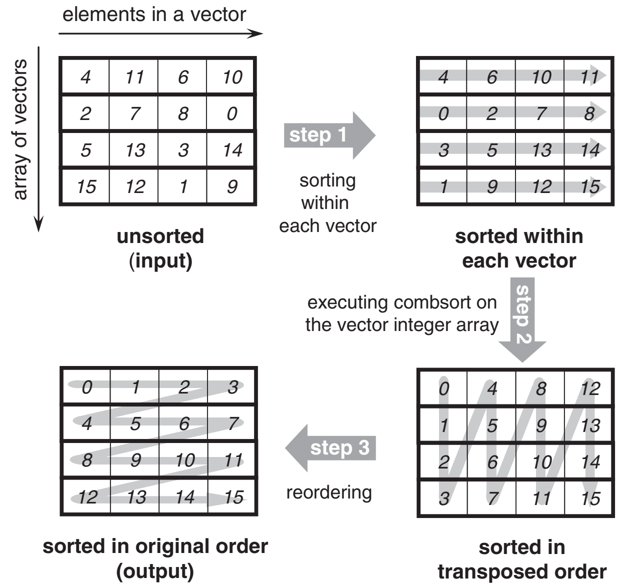

Figure 3. Steps of our vectorized combsort algorithm for sorting 16 values (4 vectors).

```
/* Step 1 */
for (i = 0; i < N/4; i++)
   sort_within_a_vector(va[i]);

/* Step 2 */
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;
   if (gap == 9 || gap == 10) gap = 11;
}

loop_count = 0;
do { /* executing bubble sort */
   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 && loop_count++ < THRESHOLD);

/* abort and switch another algorithm when loop_count reaches threshold */
if (not totally sorted) return false;

/* Step 3 */
for (i = 0; i < N/16; i++)
   transpose_4x4_block(va[i*4], va[i*4+1], va[i*4+2], va[i*4+3]);

for (i = 0; i < N/4; i++)
   move_vector_to_desired_location(va[i]);

return true; /* completed successfully */
```

Figure 4. Pseudocode of our vectorized combsort.

implemented using SIMD instructions. Comparing the code of Figure 4 to the code of the original combsort in Figure 2, the innermost loop is divided into two loops with these two operations. Within these two loops, all of the values are compared and swapped with the values for the distance of the

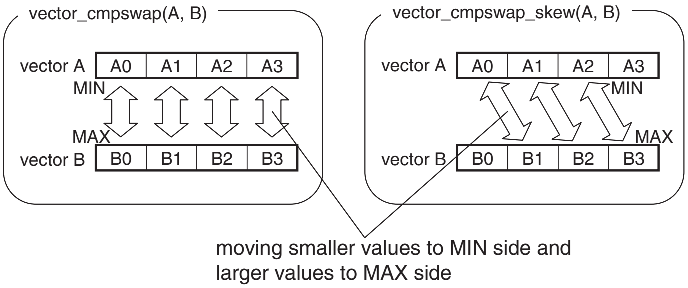

Figure 5. The vector_cmpswap and vector_cmpswap_skew operations.

gap in the transposed order. The original loop was divided into two because the pairs to be compared may reside in the same positions of the vector registers or in different positions.

The last do–while loop of Step 2 executes bubble sort to assure the correct order of the output. To guarantee against the worst-case performance of $O(N^2)$ caused by the bubble sort, we cancel the last loop in Step 2 after executing a constant number of iterations. In our AA-sort that uses the vectorized combsort in the in-core phase, we use 10 for the threshold and switch to the vectorized mergesort, whose complexity is $O(N \cdot \log(N))$ even for the worst case, when the execution of the combsort is canceled. In practice, however, we have never observed any cancelations of the vectorized combsort in our evaluations.

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 a predefined order, and hence, the reordering cannot incur troublesome overhead. Vector permutation instructions can efficiently execute this step when the number of vectors is a multiple of the number of elements in each vector, four in this example. For example, the four vectors in Figure 3 can be reordered by using only eight vector permutation instructions. The first four permutations swap the upper-right 2×2 block (consists of 8, 9, 12, and 13) and the lower-left one (2, 3, 6, and 7). The next four permutations swap the upper-right value and lower-left value in each 2×2 block (such as 1 and 4). After using this vectorized transposition technique, all of the vectors contain four sequential values, and thus, the program can reorder the values into the original order by simply moving vectors with vector load and vector store instructions. For the final data moves, our implementation uses a temporary memory space of the same size as the data. This step also does not cause any unaligned memory accesses.

In summary, our vectorized combsort consists of three steps. All three steps can be executed by SIMD instructions without unaligned memory accesses. In addition, all of them can be implemented with a negligible number of data-dependent conditional branches.

Let N be the total number of elements to be sorted. The computational complexity of Step 1 and Step 3 is $O(N)$ and that of Step 2 is the same as that of combsort, $O(N \cdot \log(N))$ on average. Thus, the computational complexity of the entire algorithm is dominated by Step 2. In Step 2, we cancel the execution of the vectorized combsort if the number of iterations exceeds a constant threshold and switch to the vectorized mergesort to guarantee the $O(N \cdot \log(N))$ complexity even for the worst case.

Our vectorized combsort 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 vectorized mergesort, which takes that problem into account.

4.2. Vectorized mergesort

For the vectorized mergesort, we propose an innovative method to integrate the odd–even merge algorithm [14] implemented with SIMD instructions into a traditional merge algorithm. Our method makes it possible for the merge operations to take advantage of SIMD instructions while still retaining the computational complexity of $O(N)$.

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

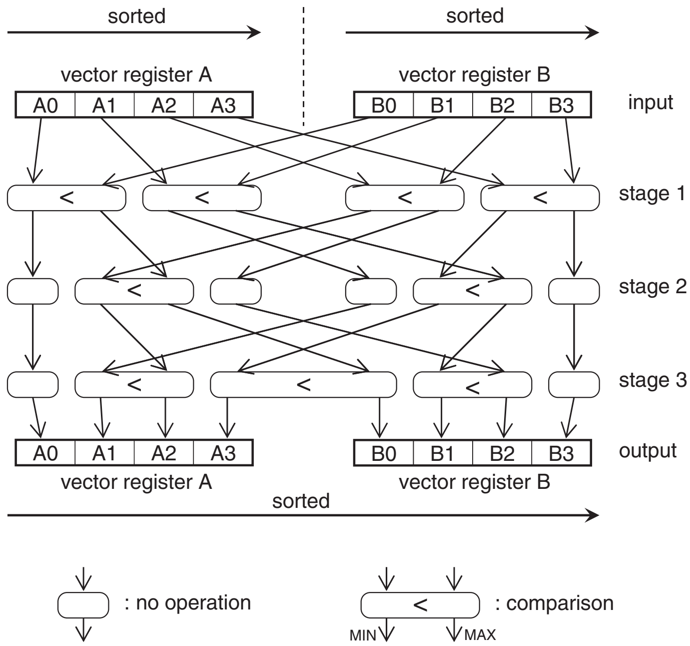

Figure 6. Data flow of odd–even merge operation for two vector registers.

symbols signify comparison operations. Each of them reads two values 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 selects, and one or two vector permutation instructions. If an SIMD instruction set does not support a vector permutation operation, then 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 the 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 four smallest 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 smallest and advances the pointer for the array.

The computational complexity of this vector_merge operation is $O(N)$, which is same as the traditional merge operation, though the odd–even merge that we used in the merge operation for two vector registers has the computational complexity of $O(N \log(N))$.

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

The vectorized mergesort recursively repeats the merge operation described earlier. It does not require any unaligned memory accesses. However, it has lower performance than our vectorized combsort for the small amounts of data that can fit in the cache because the vectorized mergesort reduces the number of data-dependent conditional branches but still uses them while the vectorized

DOI: 10.1002/spe

```
aPos = bPos = outPos = 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[outPos++] = 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++];
}

/* process remaining values */
while (aPos < aEnd) {
   vector_merge(vMin, vMax);
   vMergedArray[outPos++] = vMin;
   vMin = va[aPos++];
}
while (bPos < bEnd) {
   vector_merge(vMin, vMax);
   vMergedArray[outPos++] = vMin;
   vMin = vb[bPos++];
}
vMergedArray[outPos++] = vMax;
```

Figure 7. Pseudocode of the merge operation in memory.

combsort totally eliminates them. In contrast, the vectorized mergesort achieves higher performance than the vectorized combsort when the data cannot fit in the cache. This is because the vectorized-mergesort has much better memory access locality compared with the vectorized combsort.

In order to reduce the required memory bandwidth, which limits the performance of sorting when using many cores, we use a multi-way merge technique [4] with the out-of-core phase in our experimental implementation of the AA-sort. We employ a four-way merge, so input data is read from four streams and output into one merged output stream. This does not change the required number of comparisons but reduces the number of merging stages from $\log_2(N/B)$ to $\log_4(N/B)$. The vectorized mergesort scans all of the elements in merging stage and thus reducing the number of stages also reduces the required memory bandwidth by improving memory access locality. To execute the four-way merge using SIMD instructions, we generated two temporary arrays each having 1000 elements. At the beginning of the merging operation, we fill the first temporary array by merging the first two input streams and the second temporary array by merging other two input streams with the vectorized merge operations. Then, the output stream is generated by merging those two temporary arrays. When a temporary array becomes empty while generating the final output stream, we refill the temporary array by going back to the merging of two input arrays for the temporary array. We repeat these operations until we hit the ends of all of the input streams.

4.3. Overall parallel sorting scheme of AA-sort

The overall AA-sort executes in two 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, and sort each block with the vectorized combsort in parallel using multiple threads, where each thread processes an independent block (in-core sorting phase).

DOI: 10.1002/spe

2. Merge the sorted blocks with the vectorized mergesort using multiple threads (out-of-core merging phase).

Figure 8 shows the pseudocode for the entire AA-sort scheme, and Figure 9 depicts an example of the entire AA-sort execution with four parallel threads.

The block size for the in-core sorting phase is an important parameter. The selection of the block size depends on bandwidth and latency for each level of memory hierarchy of the target system. 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 though there were many L1 cache misses. We discuss how we chose the block size in Section 5.2.

If the total number of elements of data to sort is N and the number of elements in one block is B, then the number of blocks for the in-core algorithm is (N/B). The computational complexity of the in-core sorting of each block is $O(B \cdot \log(B))$. We avoid the worst case computational time

```
/* in-core sorting phase */
numBlocks = N / B;
blockSize = B;
blocksPerThread = numBlocks / numThreads;
for (i = blocksPerThread * myThreadID; i < blocksPerThread * (myThreadID+1); i++) {
  /* parameters are a pointer to the input data, a number of elements to sort,
     and a threshold to cancel executing combsort */
  sorted = vectorized_combsort(data[blockSize*i], blockSize, 10);

  /* switch to vectorized mergesort if combsort did not completed within the predefined number of iterations */
  if (!sorted) vectorized_mergesort(data[blockSize*i], blockSize);
}

/* out-of-core merging phase */
while (numBlocks > 1) {
  blocksPerThread = numBlocks / numThreads;
  /* if there are enough blocks to execute 4-way merge by each thread */
  if (numBlocks >= numThreads * 4) {
    for (i = blocksPerThread * myThreadID; i < blocksPerThread * (myThreadID+1); i+=4)
      /* parameters are four pointers to the input data buffers, a pointer for the output buffer, */
      /* and a number of elements to merge in each input data */
      vectorized_4way_merge(data[blockSize*i    ], data[blockSize*i+1],
                            data[blockSize*i+2], data[blockSize*i+3],
                            tmp [blockSize*i    ], blockSize);
    numBlocks /= 4; blockSize *= 4;
  }
  /* if there are enough blocks to execute 2-way merge by each thread */
  else if (numBlocks >= numThreads * 2) {
    for (i = blocksPerThread * myThreadID; i < blocksPerThread * (myThreadID+1); i+=2)
      vectorized_2way_merge(data[blockSize*i], data[blockSize*i+1],
                            tmp [blockSize*i], blockSize);
    numBlocks /= 2; blockSize *= 2;
  }
  /* if there are not enough blocks to work all thread independently */
  else {
    barrier(); /* a barrier synchronization among threads required before cooperative merge operations */
    numThreadsToCooperate = numThreads / (numBlocks / 2);
    assignedBlock = myThreadID / numThreadsToCooperate;
    vectorized_merge_with_multiple_threads(data[ 2*assignedBlock     * blockSize],
                                           data[(2*assignedBlock+1) * blockSize],
                                           tmp [ 2*assignedBlock     * blockSize], blockSize,
                                           blockSize, numThreadsToCooperate, myThreadID);
    numBlocks /= 2; blockSize *= 2;
  }
  swap(data, tmp); numMergeStages++; /* swap pointers for the input and output buffers */
}
// copy the sorted data into the final location if needed
if (numMergeStages & 1) { memcpy(tmp, data, N * sizeof(element type)); }
```

Figure 8. Pseudocode of the entire AA-sort algorithm.

DOI: 10.1002/spe

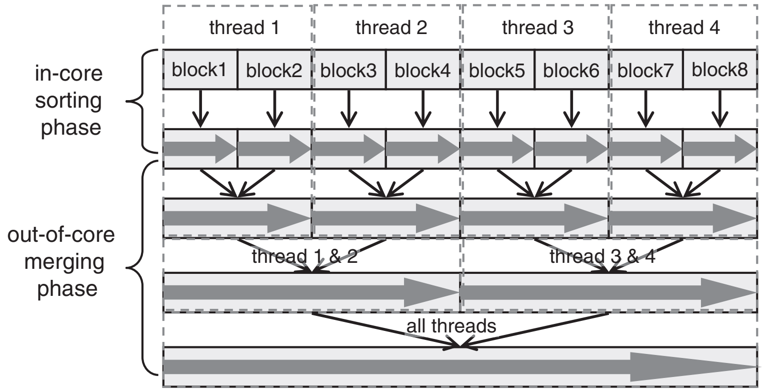

Figure 9. The entire AA-sort process, where the number of blocks $(N/B) = 8$ and the number of threads $(k) = 4$.

of $O(B^2)$ and guarantee the complexity for the in-core sorting by switching from the vectorized combsort to vectorized mergesort. Hence, the total computational complexity of the in-core sorting phase is $(N/B) \cdot O(B \cdot \log(B)) = O(N \cdot \log(B))$. 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. Thus, the total parallel complexity of the in-core phase with multiple threads is $O(N \cdot \log(B)/k)$ assuming the number of threads, k, is smaller than the number of blocks, (N/B).

In the out-of-core merging 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 $O(N \cdot \log(N/B))$, even in the worst case. Note that this complexity is not changed with the four-way merge technique. It only reduced the required system memory bandwidth. The total parallel complexity of the out-of-core merging phase with k threads is $O(N \cdot \log(N/B)/k)$. For parallelizing the last $\log(k)$ stages of the out-of-core phase, 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 [25].

The entire AA-sort has the computational complexity of $O(N \cdot \log(N))$. In addition, it can be executed in parallel by multiple threads with parallel complexity of $O(N \cdot \log(N)/k)$.

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 a 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 results of the comparisons 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 bitonic mergesort for the PowerPC 970MP [26] and the Cell BE [7] with and without using the SIMD instructions. We implemented the bitonic mergesort by following the GPUTeraSort [13] for comparison because it is one of the best existing sorting algorithms for SIMD instructions. The CellSort by Gedik [15] uses almost the same algorithm for its sorting kernel. Our measured sorting times for the bitonic mergesort on the Cell BE were comparable with

DOI: 10.1002/spe

the results of the CellSort on the Cell BE shown in their paper. For example, Gedik's paper reported that sorting 32 million random integers using 16 SPE cores took 0.746 s for the CellSort (on 3.2-GHz Cell BE), while this took 0.776 s with our implementation of the bitonic mergesort (on 2.4-GHz Cell BE).

On the PowerPC 970MP, we also evaluated two library functions, IBM's Engineering and Scientific Subroutine Library (ESSL) version 4.2 and the STL library delivered with gcc that uses the quicksort variant called introsort [27]. Table I summarizes the 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 four cores, each of which had 1 MB of L2 cache memory and the Linux kernel 2.6.20. 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 multicore processor that combines a PowerPC core with eight accelerator cores called SPEs. We used only the SPE cores for sorting. Thus, 16 SPE cores, each with 256-KB local memory, were available on the system. This system ran Linux kernel 2.6.15.

5.1. Implementation details

The programs for the PowerPC 970 were written in C using the AltiVec intrinsics [28]. 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. 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 used half of the size of the L2 cache or local memory as the block size for the in-core sorting phase, 512 KB (128,000 32-bit values) on the PowerPC 970MP and 128 KB (32,000 32-bit values) on the SPE. The shrink factor for our vectorized combsort was 1.28. We discuss how to choose the parameters in the next section.

In our parallel implementation of the AA-sort, all of the threads first execute in-core sorting and then move to the out-of-core merging phase after all of the blocks of input data are sorted. When executing the out-of-core merging phase 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 a binary search. In addition, it rebalances the data among the threads if the amount of data for each thread is not balanced [25].

To achieve the best performance on multicore processors, we used implementation techniques to reduce the required bandwidth for system memory. The experimental implementations of the AA-sort used a four-way merge. Our implementation of the bitonic mergesort 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. This technique benefits

Table I. Comparisons of algorithms.

Algorithm	SIMD	Thread parallel	Complexity	
			Average	Worst
AA-sort	Yes	Yes	$N \cdot \log(N)$	$\leftarrow$
bitonic mergesort	Yes	Yes	$N \cdot (\log(N))^2$	$\leftarrow$
ESSL	No	No	$N \cdot \log(N)$	N^2
STL (introsort)	No	No	$N \cdot \log(N)$	$\leftarrow$

SIMD, single-instruction multiple-data; AA-sort, aligned-access sort; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.
N: number of data items to sort.

DOI: 10.1002/spe

from the huge bandwidth of the on-chip bus of the Cell BE. These techniques do not change the computational complexity, but they reduce the system memory bandwidth by changing the order of comparisons to improve memory locality.

5.2. *Performance of vectorized combsort and vectorized mergesort*

This section focuses on the performance of sequential implementations of each algorithm with primary emphasis on the SIMD instructions. Then, we discuss how to select the parameters including the block size and the shrink factor for in-core sorting. In this section, we separately evaluate the two algorithms used in the AA-sort, our vectorized combsort and our vectorized mergesort, to illustrate the effects of the SIMD instructions on each algorithm. Note that the vectorized mergesort is not used with such small amounts of data when executing the entire AA-sort.

Figure 10 compares the performance of each of the sorting algorithms for 16,000 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 vectorized combsort, out-of-core algorithm, and the bitonic mergesort with SIMD instructions were drastically improved compared with the implementations without using the SIMD instructions, and the vectorized combsort achieved the highest performance among all of the algorithms tested.

The degrees of acceleration with the SIMD instructions for the vectorized combsort and the bitonic mergesort were larger than the degree of parallelism available with the SIMD instructions (4×) because of reduced number of branch mispredictions. Figure 11 shows the branch misprediction rate measured by using a performance monitor counter in the processor. The branch misprediction rates were reduced by more than a factor of 10 for the combsort and the bitonic mergesort. The change in the misprediction rate was smaller for mergesort because the data-dependent conditional branches were reduced but not totally eliminated.

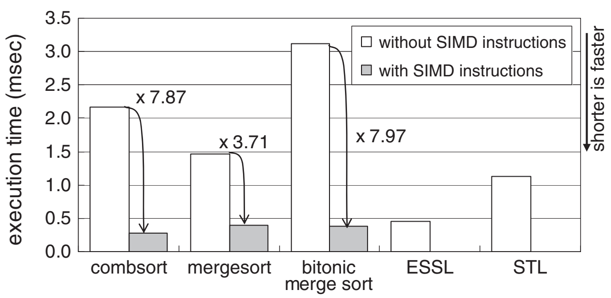

Figure 10. Acceleration by SIMD instructions with our vectorization techniques for Combsort and Mergesort when sorting 16,000 random integers on one PowerPC 970MP core. SIMD, single-instruction multiple-data; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

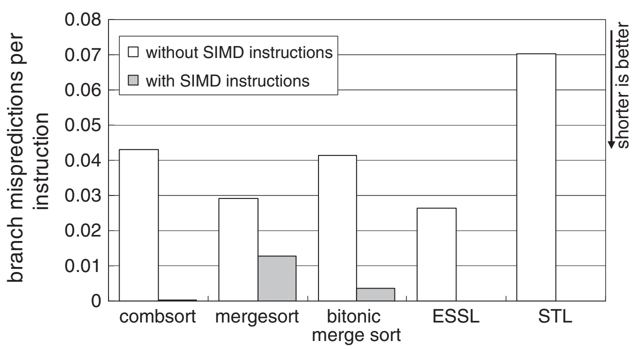

Figure 11. Improvements in branch misprediction rates by using single-instruction multiple-data instructions. ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

DOI: 10.1002/spe

Table II shows a breakdown of the performance gain with SIMD instructions shown in Figure 10 related to two factors: reductions in the numbers of instructions and improvements in cycles per instruction (CPI). The reductions in numbers of instructions were mainly due to the data parallelism of the SIMD instructions, and the CPI improvements were due to the reduced branch overhead. For our vectorized combsort and bitonic mergesort, the numbers of instructions were reduced almost in proportion to the degree of data parallelism available from the SIMD instructions, whereas the reduction was not significant for the vectorized mergesort. This is because the vectorized merge operation for the vector registers shown in Figure 6 is more complicated and requires more instructions than the naive merge operations for scalar values.

To determine the best value for the block size for PowerPC 970 processors, we evaluated the performance of the vectorized combsort and the vectorized mergesort with different amounts of data. Figure 12 shows the relationship between the performances and the amounts of data. The x-axis shows the number of elements to be sorted and the y-axis shows the sorting time. Both axes are displayed as logarithmic scales. The figure shows that the vectorized combsort was the fastest for all amounts of data smaller than the size of the L2 cache. However, its performance degraded drastically when the amount of data exceeded the L2 cache size, and it was the slowest for larger amounts of data. This was due to the high cache miss ratio caused by a poor access locality in the combsort. For all amounts of data larger than the L2 cache size, the vectorized mergesort outperformed bitonic mergesort implemented with SIMD instructions, ESSL, and STL regardless of the amount of data. Based on this result we selected 128,000 of 32-bit integers, or 512 KB, as the block size for the in-core sorting phase. This size corresponds to half of the size of the L2 cache of the processor.

Figure 13 shows the average sorting time of a block of 128,000 random 32-bit integers with the vectorized combsort with different values of the shrink factor. A good value for the shrink factor is a key to achieve high performance in the in-core sorting phase. The fastest times are when the shrink factor is 1.28 or 1.29. This result is consistent with the results for the original combsort by Lacey,

Table II. Breakdown of performance gain.

Algorithm	Speedup by SIMD	=	reduction in instructions†	×	improvement in CPI‡
Combsort	7.87		4.06		1.94
Mergesort	3.33		2.92		1.14
Bitonic mergesort	7.97		4.69		1.70

SIMD, single-instruction multiple-data; CPI, cycles per instruction.
† *reduction in instructions* $=$ *instruction_count*$_{\text{scalar}}$/*instruction_count*$_{\text{SIMD}}$.
‡ *improvement in CPI* $= CPI_{\text{scalar}}/CPI_{\text{SIMD}}$.

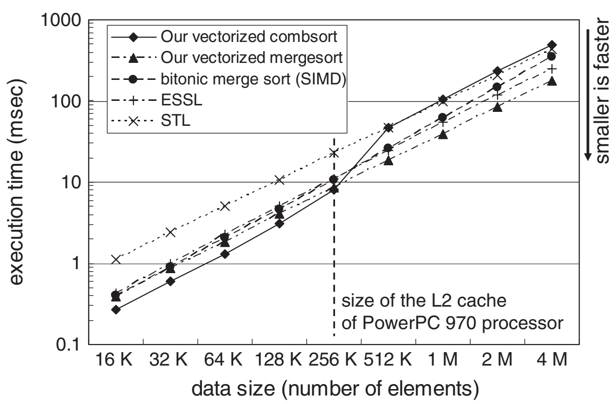

Figure 12. Performance comparisons of our vectorized combsort and vectorized mergesort to other algorithms on one PowerPC 970MP core for sorting random 32-bit integers with various amounts of data. SIMD, single-instruction multiple-data; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

DOI: 10.1002/spe

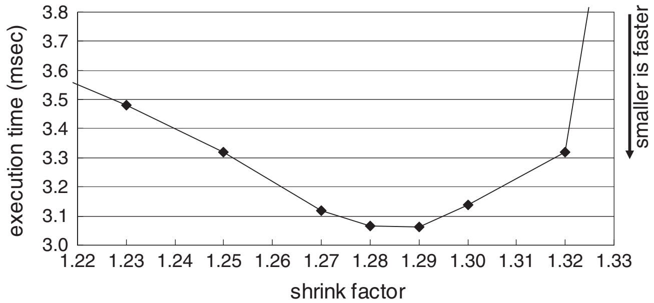

Figure 13. Average execution time for sorting 128,000 random integers with our vectorized combsort with different shrink factors.

the originator of the algorithm. The paper by Lacey and Box [5] empirically showed that a shrink factor around 1.3 gave the best results.

To quantitatively evaluate the benefit of using vectorized combsort for the in-core sorting phase, Figure 14 compares the performance of each algorithm for sorting a block of 128,000 integers of the five input datasets shown in the Table III. Our vectorized combsort clearly outperformed the other algorithms for all of the datasets. The advantage over the second-best algorithm, our vectorized mergesort, was about 40%. The three algorithms using SIMD instructions, vectorized combsort, vectorized mergesort, and the bitonic mergesort, showed much smaller dependencies on the input dataset than the other two algorithms. This is because they did not suffer from branch mispredictions even for random inputs. The performance of the ESSL and the STL were severely degraded in some cases. Our vectorized combsort may also show poor performance for some datasets because it uses a heuristic approach, but we did not observe significant degradations in any of the datasets we tested. Our vectorized mergesort did not show pathological performance even in the worst case.

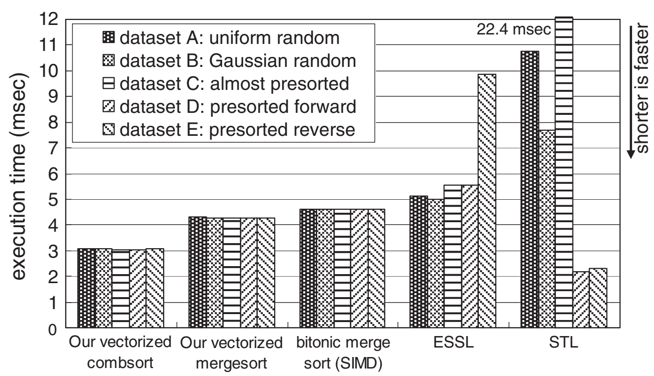

Figure 14. Average execution times for sorting various input datasets with 128,000 integers on one PowerPC 970MP core. SIMD, single-instruction multiple-data; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

Table III. Description of datasets.

Dataset	Description	Pseudocode for initialization
A	Uniform random	`for (i=0; i<N; i++) { data[i] = uniform_random(); }` // uniform random values in the range from 0 to 0xFFFFFFFF
B	Gaussian random	`for (i=0; i<N; i++) { data[i] = gaussian_random(); }` // Gaussian random values with standard deviation of 2^{24}
C	Almost presorted	`for (i=1; i<N; i++) { data[i] = i; } data[0] = N;`
D	Presorted (forward)	`for (i=0; i<N; i++) { data[i] = i; }`
E	Presorted (reversed)	`for (i=0; i<N; i++) { data[i] = N-i; }`

DOI: 10.1002/spe

Based on these results, our AA-sort combines the vectorized combsort for the in-core sorting and the vectorized mergesort for the out-of-core merging phase with a block size for in-core sorting of 128,000 32-bit integers (using 512 KB) and a shrink factor of 1.28.

5.3. Performance for 32-bit integers and floating-point values

In this section, we discuss the sorting speeds for large arrays of 32-bit integers or floating-point numbers. Figure 15 compares the performance of sequential versions of the four algorithms sorting random 32-bit integers on the PowerPC 970MP. The AA-sort and the bitonic mergesort were implemented with SIMD instructions. The x-axis shows the number of elements up to 128 million elements (512 MB), and the y-axis shows the execution time. The AA-sort achieved the best result among all of the algorithms for all amounts of data. It was 1.8 times faster than the ESSL and 3.0 times faster than the STL when sorting 32 million integers. It also surpassed the performance of the bitonic mergesort by 3.3 times. The performance advantage of the AA-sort over the bitonic mergesort became larger with larger amounts of data because of the higher computational complexity of the bitonic mergesort.

Figure 16 shows the execution time of parallel versions of the AA-sort and the bitonic mergesort on one, two, and four PowerPC 970MP cores for various numbers of uniform random integers. The results showed that both algorithms benefited from multiple cores and our AA-sort outperformed the bitonic mergesort regardless of the amount of data when using the same number of cores.

Figure 17 compares the speedups when using multiple cores. It also shows the performance of the ESSL and the STL on only one core. The AA-sort achieved larger speedups compared with

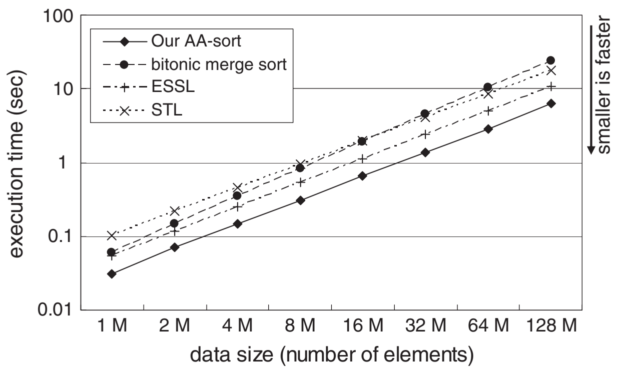

Figure 15. Performance of sequential version of each algorithm on a PowerPC 970MP core for sorting uniform random 32-bit integers with various data sizes. AA-sort, aligned-access sort; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

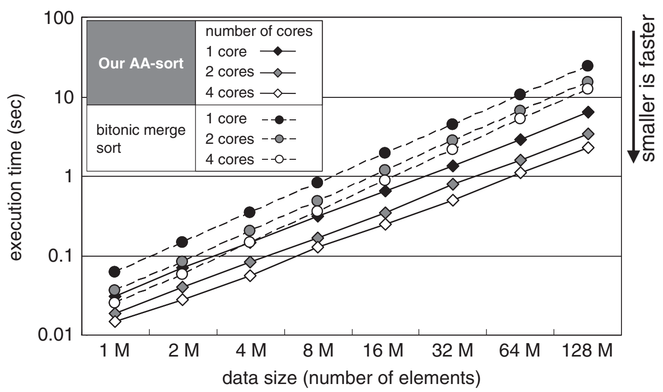

Figure 16. Performance of parallel versions of our aligned-access sort (AA-sort) and GPUTeraSort using up to four cores of PowerPC 970MP for sorting uniform random 32-bit integers with various data sizes.

DOI: 10.1002/spe

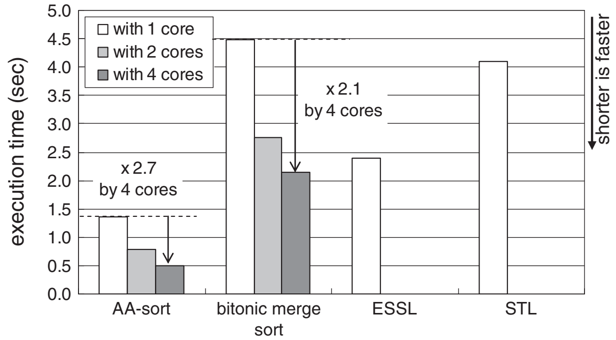

Figure 17. The execution times of parallel versions of aligned-access sort (AA-sort) and GPUTeraSort for sorting 32 million uniform random integers on up to four cores of PowerPC 970MP. ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

bitonic mergesort. As a result, the performance of the AA-sort was 4.2 times that of the bitonic mergesort when using four PowerPC 970MP cores. The AA-sort has better scalability because the bitonic mergesort has a higher communication/computation ratio than the AA-sort and the memory bandwidth was a bottleneck that limited the scalability.

Figure 18 illustrates how the performance of each algorithm depends on the five input datasets described in Table III when sorting a block of 32 million integers using up to four cores of PowerPC 970MP. The results were consistent with the results for a smaller block that fitted into the L2 cache shown in Figure 14. The AA-sort and the bitonic mergesort implemented with SIMD instructions showed much smaller dependencies on the input dataset than the other two algorithms, even when using multiple cores. The performance of ESSL was quite poor when the input is presorted in the reverse order. We found that the execution time of ESSL for this dataset was almost quadratic to the input size.

Figure 19 shows the sorting time for 32 million of Gaussian random integers with different standard deviations. The AA-sort and the bitonic mergesort showed almost constant sorting times regardless of the standard deviations of the input dataset. ESSL and STL achieved the fastest results with the smallest standard deviations. Even in the best case for those algorithms, the AA-sort outperformed ESSL by 36% and STL by 65% using only one core. Considering these results, for the rest of this paper we will use the uniform random values for the input datasets.

Figure 20 compares the sorting time for 32 million uniform random 32-bit integers and 32-bit (single-precision) floating-point values. The AA-sort took 4.8% longer to sort the floating-point values compared with sorting the same number of integers using one core. The performance difference

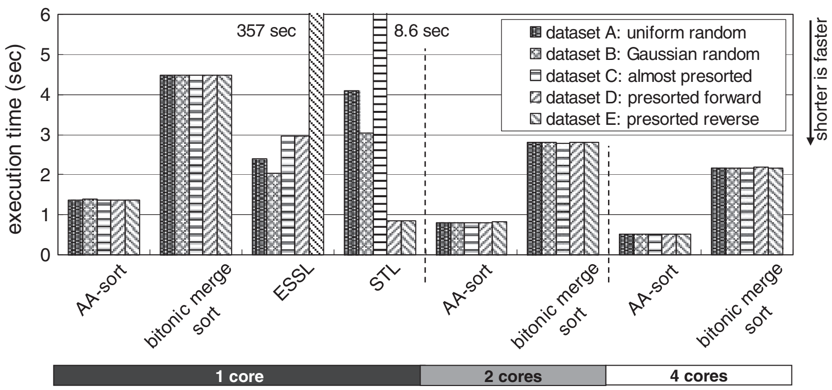

Figure 18. Performance comparison for various input datasets with 32 million integers on up to four PowerPC 970MP cores. AA-sort, aligned-access sort; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

DOI: 10.1002/spe

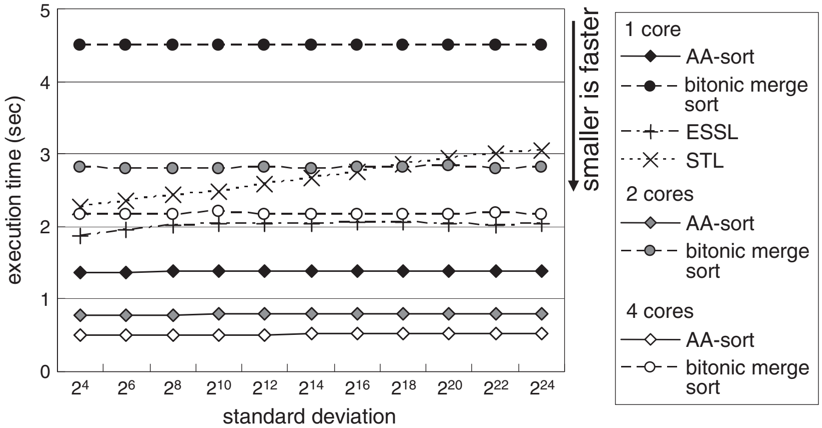

Figure 19. The effect of the standard deviation on the performance of each algorithm when sorting 32 million Gaussian random integers. AA-sort, aligned-access sort; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

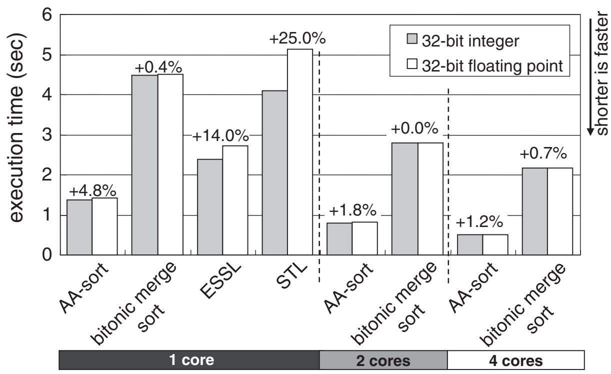

Figure 20. Performance comparisons for 32-bit integer arrays and 32-bit floating-point arrays with 32 million values using up to four cores of PowerPC 970MP. The values above the bars are the differences in execution times for 32-bit floating-point arrays compared to 32-bit integer arrays. AA-sort, aligned-access sort; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

between integers and floating-point values was much larger for the two scalar sorting algorithms, whereas it was only 0.4% for the bitonic mergesort. This is because floating-point values have longer compare-to-branch latency than integers for scalar comparisons on the PowerPC 970MP, whereas both data types have the same latency for vector comparisons. The bitonic mergesort uses only vector comparisons, and hence, its performance did not depend on the data type being sorted. The AA-sort uses scalar comparisons in addition to vector comparisons in the out-of-core merging phase, and thus, its performance differs slightly for integers and floating-point values. The difference is reduced with larger numbers of cores because the system memory bandwidth limited the performance when using multiple cores, and hence, the effects of the longer instruction latency became less significant.

Figure 21 shows how the block size for the in-core sorting phase affected the performance of the AA-sort when sorting 32 million uniform random integers using one PowerPC 970MP core. As shown in Figure 14, the vectorized combsort was faster than the vectorized mergesort for in-core sorting, and thus, increasing the block size increased the performance as long as the size of a block was smaller than the size of the L2 cache of the processor, 256,000 elements (or 1 MB). However, the block sizes larger than the size of the L2 cache drastically increased the execution time of the in-core sorting phase due to frequent L2 cache misses. Because we used 128,000 elements as the block size, the AA-sort outperformed the vectorized mergesort by 24.2%.

DOI: 10.1002/spe

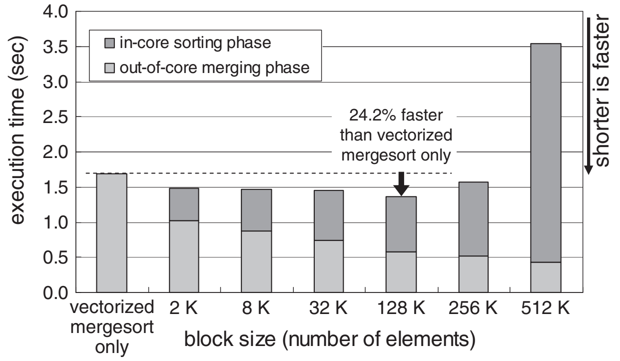

Figure 21. Average execution times for sorting 32 million uniform random integers for aligned-access sort with different block sizes.

Figure 22 illustrates the execution time breakdowns of the AA-sort into the in-core sorting phase and the out-of-core merging phase with various numbers of uniform random integers to sort. To show the importance of the in-core sorting phase, the figure also illustrates the relative performance of the vectorized mergesort. As discussed in Section 4, the computational complexity of the in-core sorting phase is $O(N \cdot \log(B))$, whereas that of the out-of-core merging phase is $O(N \cdot \log(N/B))$. Here, the block size, B, is a constant during the experiments. This means that the out-of-core merging phase consumed larger parts of the total execution time with larger amounts of data. However, even with the largest amount of data that we tested, 128 million 32-bit integers or 512 MB, more than half of the total execution time was consumed by the in-core sorting phase, and thus, the performance of the in-core sorting phase mattered. The AA-sort improved the performance by up to 25.6% compared with the vectorized mergesort. Although the performance advantage of the AA-sort became smaller for larger data sizes, the benefit was still significant even when sorting 128 million integers.

By considering the computational complexity of the two phases, we can estimate the computation time ratios in the AA-sort for much larger amounts of data. For example, we estimate that the in-core sorting phase will still use more than 40% of the total computation time when sorting 8 billion 32-bit integers, or 32 GB.

Figure 23 shows the execution time breakdown of the AA-sort for sorting 32 million uniform random integers with up to four PowerPC 970MP cores. The fraction of execution time for the out-of-core merging phase increased with increasing numbers of processor cores. This was due to the poorer scalability of the out-of-core merging phase compared with the in-core sorting phase, which

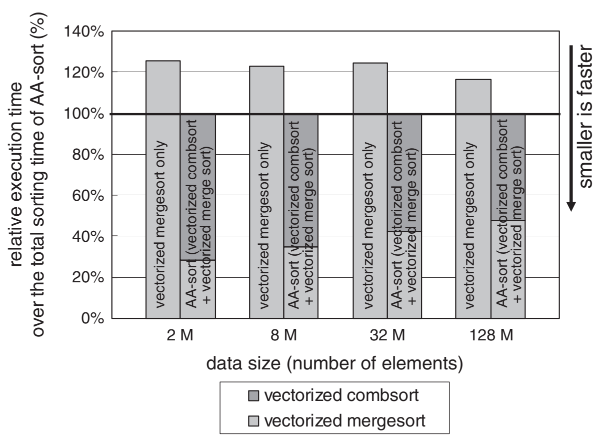

Figure 22. Performance improvements by using the vectorized combsort for the in-core sorting phase over the sorting using only the vectorized mergesort on one PowerPC 970MP core. AA-sort, aligned-access sort.

DOI: 10.1002/spe

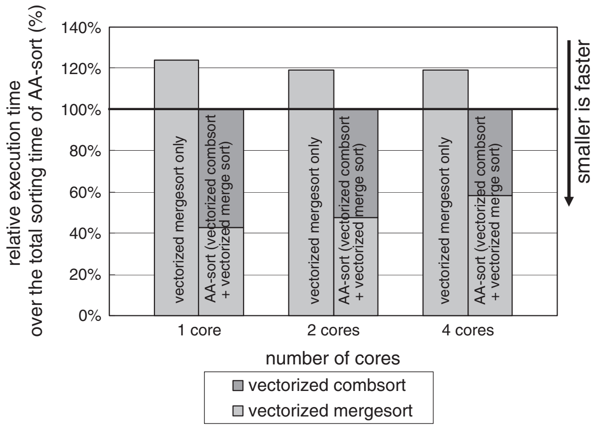

Figure 23. Execution time breakdown of the parallel version of aligned-access sort (AA-sort) with 32 million uniform random integers on up to four cores of PowerPC 970MP.

scaled almost linearly with number of cores. In the in-core sorting phase, most memory accesses were served by the L2 cache because this phase sorted blocks that fit into the L2 cache. The out-of-core merging phase generated more L2 cache misses, and hence, the bandwidth to the system memory became a significant scalability bottleneck. Figure 23 also shows the performance advantage of the AA-sort over the vectorized mergesort. The advantages were not significantly affected by the number of cores used.

To see the performance scalability with larger numbers of cores, Figure 24 shows the scalability of the AA-sort and the bitonic mergesort on the Cell BE up to 16 cores when sorting 32 million 32-bit integers. Both algorithms showed almost linear speedup for up to four cores because Cell BE provides more memory bandwidth than PowerPC 970MP. With more than four cores, our AA-sort demonstrated better scalability than the bitonic mergesort. For example, the AA-sort achieved a speedup of 12.2 for 16 cores, whereas the bitonic mergesort achieved 7.1. This was because the bitonic mergesort requires higher system memory bandwidth than the AA-sort, and the memory bandwidth was a bottleneck that limited the scalability. As a result, the performance of the AA-sort was better than the bitonic mergesort by 4.9 times with 16 Cell BE cores.

Our implementation of the AA-sort and bitonic mergesort used techniques to reduce the required system memory bandwidth as described in Section 5.2: the four-way merge for the AA-sort and the inter-SPE data transfer for the bitonic mergesort. Figure 25 depicts how those techniques affected the overall sorting performance. The techniques improved the performance when using 16 SPE cores by 16.6% for the AA-sort and by 36.9% for the bitonic mergesort, whereas they degraded the performance using only one SPE core. This was because those techniques incurred additional

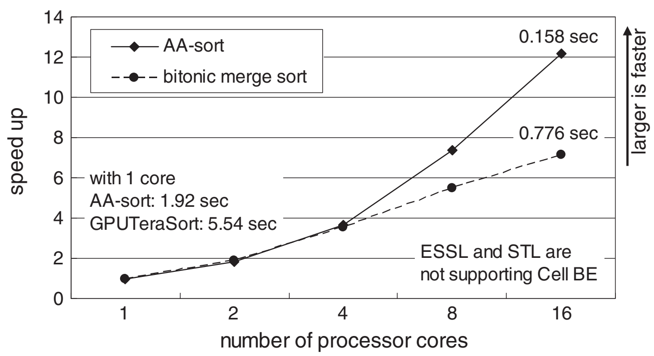

Figure 24. Scalability with increasing number of cores on Cell BE for 32 million random integers. AA-sort, aligned-access sort; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

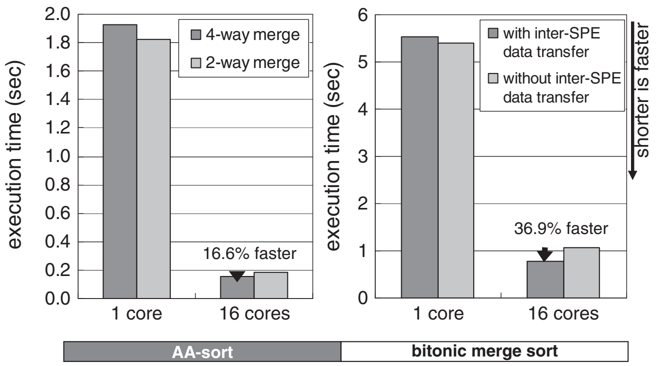

Figure 25. Average execution time of sorting for 32 million random integers with and without techniques to reduce system memory bandwidth on Cell BE. AA-sort, aligned-access sort; SPE, synergistic processing element.

computation overhead as a trade-off for reduced memory bandwidth, and thus, they did not improve the performance with a small number of cores, where the memory access latency was totally hidden behind the computation by a double buffering technique, and thus, the system memory bandwidth was not a limiting factor of the performance.

5.4. Performance for {key, data} Pairs

This section focuses on sorting for pairs of a key and associated data such as a pointer to the structure having that key value. Figure 26 compares the performance of sorting pairs of a 32-bit random integer key and 32-bit associated data using one PowerPC 970MP core using four algorithms. For the AA-sort and the bitonic mergesort, we show the results with two different data formats. One format uses two separate arrays to store the keys and the attached data. Another format uses only one array, and the keys and attached data are stored one after another. The x-axis shows the number of elements up to 64 million pairs (512 MB). The results show that the performance of both the AA-sort and the bitonic mergesort implemented with SIMD instructions did not depend on the data format of input data significantly. Compared with the scalar sorting libraries, ESSL, and STL, the performance advantages of the AA-sort became much smaller for sorting {key, data} pairs than for sorting only 32-bit values. The AA-sort was about 50% faster than the STL and was almost comparable with the ESSL when sorting 64 million pairs. This was because the main memory bandwidth became the main bottleneck when sorting {key, data} pairs on the PowerPC 970MP system used in experiments.

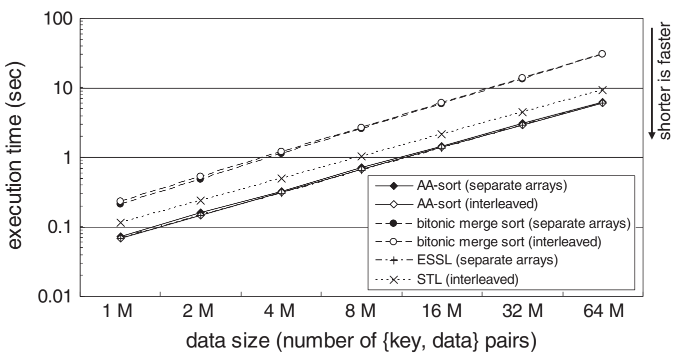

Figure 26. Performance comparisons of the data type of the key for sorting key, data pairs on a PowerPC 970MP core using two different data formats. 'Separate arrays' means that the keys and the attached data are stored in separate arrays, and 'interleaved' means that the keys and attached data are stored in an array one after another. AA-sort, aligned-access sort; ESSL, Engineering and Scientific Subroutine Library; STL, Standard Template Library.

To see the performance of parallel versions of the AA-sort and the bitonic mergesort for {key, data} pairs, we measured the performance on the Cell BE system, which provides much larger system bandwidth. Figure 27 shows the sorting times of the AA-sort and the bitonic mergesort for sorting pairs with various data types for keys while using 16 Cell BE cores. The x-axis shows the number of elements up to 16 million 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 uniform random numbers. For the ASCII string keys, we used the input data generator of the Sort Benchmark (http://sortbenchmark.org/) to initialize the keys and sorted them into order with the strnicmp(·) function.

Our implementations for wider keys, 64-bit integers, and 10-byte ASCII strings use the hybrid approach of our algorithm and radixsort. Govindaraju *et al.* [13] also used a similar hybrid approach for the improved bitonic mergesort and radixsort in the bitonic mergesort. 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 26 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 million pairs with random integer keys was about 1.6 times slower than that for sorting 16 million simple 32-bit integer values. However, the AA-sort achieved up to 5.0 times faster results than the bitonic mergesort for the {key, pointer} pairs with 32-bit integer keys. For the wider keys, the performance was slightly degraded because of 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 bitonic mergesort for the pairs with 32-bit integer keys.

6. CONCLUSION

This paper describes a new high-performance sorting algorithm that we call AA-sort. The AA-sort is suitable for exploiting both the SIMD instructions and thread-level parallelism available on today's multicore 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 BE 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 bitonic mergesort using SIMD instructions by 3.3 times on the PowerPC 970MP when sorting 32 million random 32-bit integers. Also, a parallel version of the AA-sort demonstrated better scalability with increasing numbers of cores than a parallel version of the bitonic mergesort.

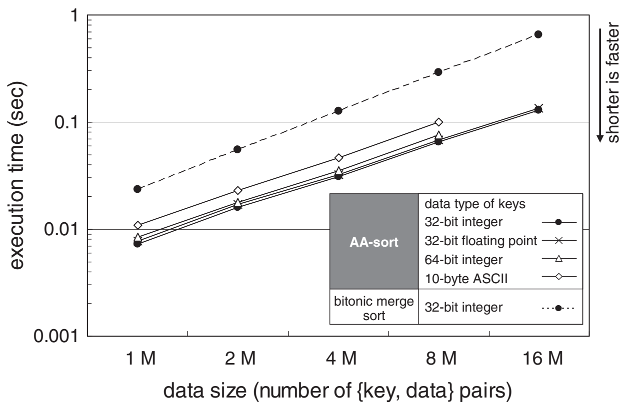

Figure 27. Performance comparisons of the data type of the key for sorting key, data pairs stored in separate arrays on the Cell BE using 16 cores.

DOI: 10.1002/spe

The AA-sort achieved a speedup of 12.2 for 16 cores on the Cell BE, whereas the bitonic mergesort achieved 7.1. As a result, the AA-sort was 4.2 times faster on four PowerPC 970MP cores and 4.9 times faster on 16 Cell BE cores compared with the bitonic mergesort.

We achieved this high performance in our algorithm by (i) improving efficiency in sorting kernel and (ii) reducing the main memory bandwidth requirements compared with the bitonic mergesort. The key for efficient sorting kernel is reduced branch mispredictions and data parallel processing by SIMD instructions. Improving efficiency in sorting kernel is quite important for the overall sorting performance because the clock speed of each processor core are no longer increasing as rapidly as in the past. To exploit the huge number of processor cores in the future multicore processors, reducing the memory bandwidth is critically important. We believe that these key considerations are important for not only sorting but also for any other high-performance algorithms and applications seeking to exploit the current and future multicore processors.

REFERENCES

1. Zhou J, Ross KA. Implementing database operations using SIMD instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, 2002; 145–156, DOI: 10.1145/564691.564709.
2. Graefe G. Implementing sorting in database systems. *ACM Computing Surveys* 2006; **38**(3).
3. Martin WA. Sorting. *ACM Computing Surveys* 1971; **3**(4):147–174.
4. Knuth DE. *The Art of Computer Programming. Vol. 3: Sorting and Searching*, 1973.
5. Lacey S, Box R. *Byte Magazine (April)*, 1991.
6. Inoue H, Moriyama T, Komatsu H, Nakatani T. AA-Sort: A new parallel sorting algorithm for multi-core SIMD processors. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2007; 189–198, DOI: 10.1109/PACT.2007.12.
7. Pham D, Asano S, Bolliger M, Day MN, Hofstee HP, Johns C, Kahle J, Kameyama A, Keaty J, Masubuchi Y, Riley M, Shippy D, Stasiak D, Suzuoki M, Wang M, Warnock J, Weitzel S, Wendel D, Yamazaki T, Yazawa K. The design and implementation of a first-generation CELL processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*. IEEE Computer Society: Los Alamitos, 2005; 184–185.
8. IBM Corp. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*, 1998.
9. Chhugani J, Nguyen AD, Lee VW, Macy W, Hagog M, Chen Y, Baransi A, Kumar S, Dubey P. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *Proceedings of the International Conference on Very Large Data Bases*. VLDB Endowment Inc., 2008; 1313–1324, DOI: 10.1145/1454159.1454171.
10. Intel Corp. *IA-32 Intel Architecture Software Developer's Manual*.
11. Purcell T, Donner C, Cammarano M, Jensen H, Hanrahan P. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. ACM Press: New York, 2003; 41–50.
12. Govindaraju NK, Raghuvanshi N, Manocha D. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, 2005; 611–622, DOI: 10.1145/1066157.1066227.
13. Govindaraju NK, Gray J, Kumar R, Manocha D. GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, 2006; 325–336, DOI: 10.1145/1142473.1142511.
14. Batcher KE. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32*. AFIPS, 1968; 307–314.
15. Gedik B, Bordawekar RR, Yu PS. CellSort: High performance sorting on the cell processor. In *Proceedings of the ACM International Conference on Very Large Data Bases*. VLDB Endowment Inc., 2007; 1286–1297.
16. Furtak T, Amaral JN, Niewiadomski R. 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*. ACM Press: New York, 2007; 348–357, DOI: 10.1145/1248377.1248436.
17. Furtak T. Sorting Using SIMD Registers. Masters Thesis, University of Alberta, 2007.
18. Cederman D, Tsigas P. A practical quicksort algorithm for graphics processors. In *Proceedings of the European Symposium on Algorithms*. Springer-Verlag (LNCS vol. 5193): Berlin, 2008; 246–258, DOI: 10.1007/978-3-540-87744-8_21.
19. Nickolls J, Buck I, Garland M. Scalable parallel programming with CUDA. *ACM Queue* 2008; **6**(2):40–53.
20. Sintorn E, Assarsson U. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing* 2008; **68**(10):1381–1388.
21. Satish N, Mark Harris M, Garland M. Designing efficient sorting algorithms for manycore GPUs. In *Proceeding of the 2009 IEEE International Symposium on Parallel and Distributed Processing*. IEEE Computer Society: Los Alamitos, 2009; 1–10, DOI: 10.1109/IPDPS.2009.5161005.

22. Satish N, Kim C, Chhugani J, Nguyen AD, Lee VW, Kim D, Dubey P. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 International Conference on Management of Data*. ACM Press: New York, 2010; 351–362, DOI: 10.1145/1807167.1807207.
23. Keller J, Kessler CW. Optimized pipelined parallel merge sort on the Cell BE. In *Euro-Par 2008 Workshops—Parallel Processing*. Springer-Verlag: Berlin, 2009, DOI: 978-3-642-00955-6_18.
24. Sanders P, Winkel S. Super scalar sample sort. In *Proceedings of the European Symposium on Algorithms*. Springer-Verlag (LNCS vol. 3221): Berlin, 2004; 784–796.
25. Francis R, Mathieson I. A benchmark parallel sort for shared memory multiprocessors. *IEEE Transactions on Computers* 1988; **37**(12):1619–1626.
26. IBM Corp. *IBM PowerPC 970MP RISC Microprocessor User's Manual*, 2005.
27. Musser DR. Introspective sorting and selection algorithms. *Software Practice and Experience* 1997; **27**(8):983–993.
28. Freescale Semiconductor Inc. *AltiVec Technology Programming Interface Manual*, 1999.

DOI: 10.1002/spe