Sorting Large Records On A Cell Broadband Engine*

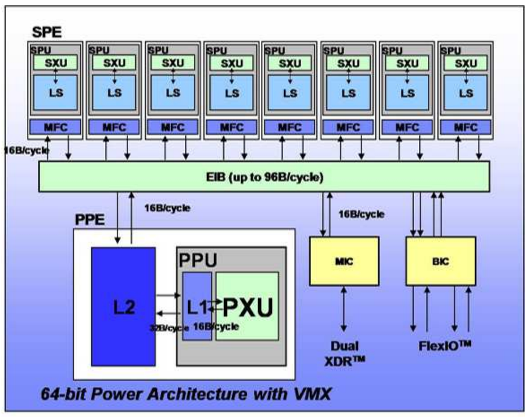
Shibdas Bandyopadhyay and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
shibdas@ufl.edu, sahni@cise.ufl.edu

***Abstract*— We consider the sorting of a large number of multifield records on the Cell Broadband engine. We show that our method, which generates runs using a 2-way merge and then merges these runs using a 4-way merge, outperforms previously proposed sort methods that use either comb sort or bitonic sort for run generation followed by a 2-way odd-even merging of runs. Interestingly, best performance is achieved by using scalar memory copy instructions rather than vector instructions.**

***Index Terms*— Cell broadband engine, sorting multifield records, merge sort.**

I. Introduction

The Cell Broadband Engine (CBE) is a heterogeneous multicore architecture developed by IBM, Sony, and Toshiba. A CBE (Figure 1) consists of a Power PC (PPU) core, eight Synergistic Processing Elements or Units (SPEs or SPUs), and associated memory transfer mechanisms [7]. The SPUs are connected in a ring topology and each SPU has its own local store. However, SPUs have no local cache and no branch prediction logic. Data may be moved between an SPUs local store and central memory via a DMA transfer, which is handled by a Memory Flow Control (MFC). Since the MFC runs independent of the SPUs, data transfer can be done concurrently with computation.



1: Architecture of the Cell Broadband Engine [6]

* This research was supported, in part, by the National Science Foundation under grant 0829916 The authors acknowledge Georgia Institute of Technology and its Sony-Toshiba-IBM Center of Competence for the use of their Cell Broadband Engine Resource.

Sorting on a CBE has been considered earlier in [1], [5], [9], [13]. The algorithm of Sharma et al. [13] uses the master-slave model that was also used by Won and Sahni [14]. In this model, the SPUs act as slaves to the PPU. Each SPU sorts a memory load of data and the PPU merges the sorted memory loads received from the SPUs. The remaining papers use a hierarchical model in which the merging is done also by the SPUs. In this paper, we focus on the hierarchical model. The CBE sort algorithms of [1], [5], [9], [13] are designed to sort numbers and make extensive use of the CBE's SIMD vector instructions. Inoue et al. [9] describe how their number sorting algorithm may be adpated to sort multifield records. The focus of this paper is sorting a large number of multifield records.

The CBE number sorting algorithms of [1], [5], [9] follow the general two phase paradigm for an external sort [8]. In the first phase sorted sequences called runs are generated using the SPUs as independent processors. In the second phase, called run merging, the runs generated in the first phase are merged together into a single run. AA-sort, which is the CBE sort algorithm proposed in [9], uses an adaptation of comb sort for the run generation phase. The complexity of comb sort, which was originally proposed by Knuth [10] and rediscovered by Dobosiewicz [3] and Box and Lacey [2], is $O(n^2)$ [3]. The second phase of AA-sort is a SIMD adaptation of odd-even merge [10] and a standard 2-way merge of runs. CellSort, which is the CBE sort algorithm proposed in [5], generates runs using an adaptation of bitonic sort (e.g., [10]) whose complexity ins $O(n \log^2 n)$. Run merging is done using a bitonic merge. Bandyopadhyay and Sahni [1] develop SPU adaptations of brick, shaker, and merge sorts. Extensive experiments conducted by them reveal that their SPU merge sort adaptation out performs their SPU brick and shaker sort adaptations as well as the SPU adaptations of comb sort [9] and bitonic sort [5]. So, merge sort is the best algorithm for the run generating phase, at least when we are sorting numbers rather than multifield records.

Inoue et al. [9] extend their number sorting algorithm to sort records; each record has a key and one or more other fields. They consider two cases. In the first, the keys of all records are stored in one array, K, and the remaining fields are stored in another array O. In this case, the code for AA sort is augmented with instructions to move corresponding elements of O whenever elements of K are moved, ensuring that $K[i]$ and $O[i]$ always define a record. In the second scheme, each record is followed by its other fields. This layout is considered only for the case when each of the other fields of a record occupy 32 bits. Now, each 128-bit vector is comprised of two records (2 32-bit keys and their corresponding other fields). When two records are compared using an SIMD compare instruction, 4 pairs of 32-bit compares are done. Two of these involve keys and two involve data. The results of the data compare are replaced by the results of the corresponding key compares so as to ensure that the ensuing key move also results in a move of the associated other fields. In this second layout, sort efficiency declines as each vector compare really compares only 2 pairs of keys (rather than 4 pairs) and there is the added overhead of replacing the result of data compares with that of key compares.

As indicated earlier, our focus in this paper is sorting multifield records. We consider both the layouts considered by Inoue et al. [9] as well as the case when the other fields of a record occupy more than 32 bits. Our main contribution is the development of a 4-way run merging algorithm for the CBE. Using this 4-way run merging algorithm we are able to sort multifield records faster than using a 2-way run merging algorithm as proposed in [9]. We begin in Section II by describing SPU vector and memory operations used in the remainder of this paper. Then in section III we describe more carefully the two different record layouts used in [9] and describe also how algorithms to sort numbers may be adapted to sort multifield records in these two layouts. In section **??** we describe the SIMD version of our 4-way merge algorithm and in Section **??** we describe the scalar version of this 4-way merge algorithm. Experimental results comparing various algorithms to sort multifield records are presented in Section V.

II. SPU Vector and Memory Operations

We describe the key SIMD functions that operate on a vector of 4 numbers in this section. These functions are used in the development of our sorting algorithms. In the following, $v1$, $v2$, min, max, and $temp$ are vectors, each comprised of 4 numbers and p, $p1$, and $p2$ are bit patterns. Also, $dest$ (destination) and src (source) are addresses in the local store of an SPU and $bufferA$ is a buffer in local store while $streamA$ is a data stream in main memory. Function names that begin with spu are standard C/C++ Cell SPU intrinsics. Our description of these functions is tailored to the sorting application of this paper.

1) $spu\_shuffle(v1, v2, p) \cdots$ This function returns a vector comprised of a subset of the 8 numbers in $v1$ and $v2$. The returned subset is determined by the bit pattern p. Let W, X, Y, and Z denote the 4 numbers (left to right) of $v1$ and let A, B, C, and D denote those of $v2$. The bit pattern $p = XCCW$, for example, returns a vector comprised of the second number in $v1$ followed by two copies of the third number of $v2$ followed by the first number in $v1$. In the following, we assume that constant patterns such as XYZD have been pre-defined.
2) $spu\_slqwbyte(v1, n) \cdots$ Returns a vector obtained by shifting the bytes of $v1$ m bytes to the left, where m is the number represented by the 5 least significant bits of n. The left shift is done with zero fill. So, the rightmost m bytes of the returned vector are 0.
3) $spu\_cmpgt(v1, v2) \cdots$ A 128-bit vector representing the pairwise comparison of the 4 numbers of $v1$ with those of $v2$ is returned. If an element of $v1$ is greater than the corresponding element of $v2$, the corresponding 32 bits of the returned vector are 1; otherwise, these bits are 0.
4) $spu\_select(v1, v2, p) \cdots$ Returns a vector whose ith bit comes from $v1$ ($v2$) when the ith bit of p is 0 (1).
5) $memcpy(dest, src, size)$ copies the $size$ number of bytes from the local store location beginning at src to $dest$.
6) $dmaIn(bufferA, streamA)$ This function triggers a DMA transfer of the next buffer load of data from $streamA$ in main memory into $bufferA$ in the local store. This is done asynchronously and concurrently with SPU execution.
7) $dmaOut(bufferA, streamA)$ This function is similar to $dmaIn$ except that a buffer load of data is transferred asynchronously from $bufferA$ in the local store to $streamA$ in main memory.

III. Record Layout And Sorting

A record R is comprised of a key k and m other fields $f_1, f_2, \cdots, f_m$. For simplicity, we assume that the key and each other field occupies 32 bits. Hence, a 128-bit CBE vector may hold up to 4 of these 32-bit values. Although the development in this paper relies heavily on storing 4 keys in a vector (hence on each key occupying 32 bits), the size of the other fields isn't very significant. Let k_i be the key of record R_i and let f_{ij}, $1 \leq j \leq m$ be this record's other fields. With our simplifying assumption of uniform size fields, we may view the n records to be sorted as a two-dimensional array $fieldsArray[][]$ with $fieldsArray[i][0] = k_i$ and $fieldsArray[i][j] = f_{ij}$, $1 \leq j \leq m$, $1 \leq i \leq n$. When this array is mapped to memory in column-major order, we get the first layout considered in [9]. We call this layout the *ByField* layout as, in this layout, the n keys come first. Next we have the n values for the first field of the records followed by the n second fields, and so on. When the fields array is mapped to memory in row-major order, we get the second layout considered in [9]. This layout, which is a more common layout for records, is called the *ByRecord* layout as, in this layout, all the fields of R_1 come first, then we have all fields of R_2 and so on. When the sort begins with data in the *ByField* (*ByRecord*) layout, the result of the sort must also be in the *ByField* (*ByRecord*) layout.

There are two high-level strategies to sort multifield records. In the first, we strip the keys from the records and create n tuples of the form (k_i, i). We then sort the tuples by their first component. The second component of the tuples in the sorted sequence defines a permutation of the record indexes that corresponds to the sorted order for the initial records. The records are rearranged into this permutation by either copying from $fieldsArray$ to new space or inplace using a cycle chasing algorithm as described for a table sort in [8]. This strategy has the advantage of requiring only a linear number of record moves. So, if the size of each record is s and if the time to sort the tuples is $O(n \log n)$, the entire sort of the n records can be completed in $O(n \log n + ns)$ time. The second high-level strategy is to move all the fields of a record each time its key is moved by the sort algorithm. In this case, if the time to sort the keys alone is $O(n \log n)$, the time to sort the records is $O(ns \log n)$. For even relatively small s, the first strategy outperforms the second when the records are stored in uniform access memory. However, since reordering records according to a prescribed permutation with a linear number of moves makes random accesses to memory, the second scheme outperforms the first (unless s is very large) when the records to be rearranged are in relatively slow memory such as disk or the main memory of the CBE. For this reason, we focus, in this paper, on using the second strategy. That is, our sort algorithm moves all the fields of a record whenever its key is moved.

Two SIMD vector operations used frequently in number sorting algorithms are $findmin$ and $shuffle$. The $findmin$ operation compares corresponding elements in two vectors and returns a vector min that contains, for each compared pair, the smaller. For example, when the two vectors being compared are (4, 6, 2, 9) and (1, 8, 5, 3), the min is (1, 6, 2, 3). Suppose that v_i and v_j are vectors that, respectively, contain the keys for records $R_{i:i+3}$ and $R_{j:j+3}$. Figure 2 shows how we may move the records with the smaller keys to a block of memory beginning at $minRecords$.

```
pattern = spu_cmpgt(vi, vj);
minRecords = fields_select(vi, vj, pattern);
```

2: The $findmin$ operation for records

When the $ByField$ layout is used, $fields\_select$ takes the form given in Figure 3.

```
for(p = 1; p <= m; p++) {
  minRecords[p] = spu_select(fieldsArray[i][p],
fieldsArray[j][p], pattern);
}
```

3: $fields\_select$ operation in $ByField$ layout

Notice that in the $byField$ layout, the elements $fieldsArray[i : i+3][p]$ are contiguous and define a 4-element vector. However, in the $ByRecord$ layout, these elements are not contiguous and a different strategy (Figure 4) must be employed. The function $memcpy(dest, src, size)$ moves $size$ number of bytes from the local store location beginning at src to the local store location beginning at $dest$; $recSize$ is the length of a record including its key (i.e., it is the number of bytes taken by a row of $fieldsArray$).

The $shuffle$ operation defined by $spu\_shuffle$ for the case of sorting numbers may be extended to the case of multifield records using the code of Figure 5 for the $byField$ layout and that of Figure 6 for the $ByRecord$ layout. Both codes are for the case when the shuffle pattern is $WYAC$. Other shuffle patterns are done in a similar way.

```
for(p = 0; p < 4; p++) {
  memcpy(minRecords + p * recSize, (pattern[p] == 1)
? fieldsArray[i+p] : fieldsArray[j+p], recSize);
}
```

4: $fields\_select$ operation in $ByRecord$ layout

```
for(p = 0; p <= m; p++) {
  resultRecords[p] = spu_shuffle(fieldsArray[i][p],
fieldsArray[j][p], WYAC);
}
```

5: Shuffling two records in $ByField$ layout

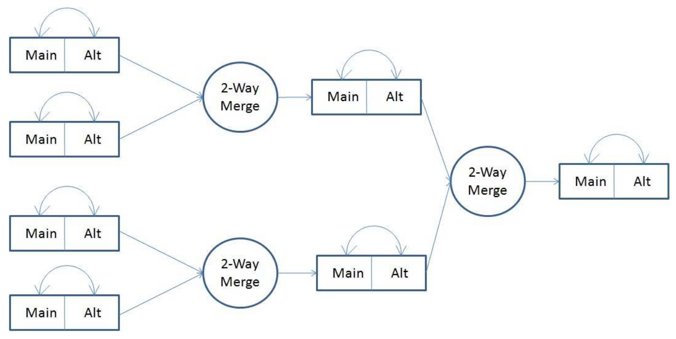
```
memcpy(resultRecords, arrayFields[i], recSize);
memcpy(resultrecords + recSize, arrayFields[i + 2],
recSize);
memcpy(resultFields + 2 * recSize, arrayFields[j],
recSize);
memcpy(resultFields + 3 * recSize, arrayFields[j +
2], recSize);
```

6: Shuffling two records in $ByRecord$ layout

We observe that when the $ByField$ layout is used the $findMin$ and $shuffle$ operations perform $O(m)$ vector operations and that when the $ByRecord$ layout is used, a constant number (4) of $memcpy$ operations are done. However, the time for each $memcpy$ operation increases with m.

IV. 4-Way Merge

In Section III, we saw the adaptations needed to the run generation algorithms of [1], [5], [9] so that these may be used to generate runs for multifield records rather than for numbers. In the run merging phase, each of the SPUs independently merges a different set of runs. So, one need develop only a merge algorithm for a single SPU. Inoue et al. [9] propose a single SPU merging algorithm that merges runs in pairs (i.e., a 2-way merge) using an adaptation of odd-even merge. It takes 3 spu_cmpgt instructions, 6 $spu\_shuffle$ and 6 spu_select instructions to merge two SPU vectors using this scheme. The run merging strategy of [9] may be adapted to the case of multifield records using the methods of Section III. When we are sorting numbers, the SPU processing time (computation time) exceeds the time spent moving data between the main memory and the local SPU memories (IO time) and it is possible to hide virtually all of the IO time using double buffering and asynchronous DMA transfers. Since the use of a higher-order merge in the run merging phase reduces IO time and has little impact on computation time [8], there is no advantage to considering a higher-order merge when sorting numbers. However, when sorting multifield records, the IO time increases with the size of a record and for suitably large records, this IO time will exceed the computation time and so cannot be effectively hidden using a 2-way merge and double buffering. So, for multifield records, there is merit to developing a higher-order merge. We propose two 4-way merge algorithms. One is a scalar algorithm and the other a vectorized SIMD algorithm. Both algorithms are based on the high-level strategy shown in Figure 7.



7: 4-way merge

Our 4-way merge strategy involves performing 3 2-way merges in a single SPU using two buffers (main and alt) for each of the 4 input streams A, B, C, and D as well as 2 buffers for the output stream O. An additional buffer is used for the output (E and F, respectively) of each of the two left 2-way merge nodes of Figure 7. So, we employ a total 12 buffers. Runs A and B are pairwise merged using the top left 2-way merge node while runs C and D are pairwise merged using the bottom left 2-way merge node. The former 2-way merge generates the intermediate run E while the latter generates the intermediate run F. The intermediate runs E and F are merged by the right 2-way merge node to produce the output run O, which is written back to main memory. Run generation is done one block or buffer load at a time. Double buffering is employed for the input of A, B, C, and D from main memory and the output of O to main memory. By using double buffering and asynchronous DMA transfers to and from main memory, we are able to overlap much of the IO time with computation time.

A. Scalar 4-way Merge

Figure 8 gives the pseudocode for our scalar 4-way merge algorithm.

For simplicity, algorithm $4wayMerge$ assumes that we have an integral number of blocks of data in each run. So, if each of the runs A, B, C, and D is (say) 10 blocks long, the output run O will be n = 40 blocks long. $4wayMerge$ generates these output n blocks one block at a time. Even blocks are accumulated in one of the output buffers and odd blocks in the other. When an output buffer becomes full, we write the block to memory using an asynchronous DMA transfer ($dmaOut$) and continue output run generation using the other outbut buffer. So, other than when the first output

```
Algorithm 4wayMerge(A, B, C, D, O, n)
{// Merge runs/streams A, B, C, and D to produce O
with
  n blocks of size bSize
   // bufferA is a buffer for A
   initiate a $dmaIn$ for $bufferA$, $bufferB$, $bufferC$,
and $bufferD$;
   for (i = 0; i < n; i++) {
     for (j = 0; j < bSize; j++) {
do block i         if(bufferE is empty)
           mergeEF(A, B, E);
         if(bufferF is empty)
           mergeEF(C, D, F);
         move smaller record from front of bufferE
           and bufferF to bufferO}
     dmaOut(bufferO, O);
     switch the roles of the output buffers;
   }
}
```

8: 4-way merge

block is being generated and the last being written to main memory, one of the output blocks is being written to main memory while the other one is being filled with records for the next block. At the end of each iteration of the outer **for** loop, we switch the roles of the two output buffers–the one that was being written to main memory becomes the buffer to place records for the next block and the one that was being filled is written out. Of course, this switch may entail some delay as we must wait for the ongoing (if any) $dmaOut$ to complete before we use this buffer for the records of the next block. When generating a block of the output run, we merge from the buffers $bufferE$ and $bufferF$ to the output buffer $bufferO$ that is currently designated for this purpose. The number of records in a full buffer (i.e., the block size) is $bSize$. In case either $bufferE$ or $bufferF$ is empty, the generation of the output block is suspended and we proceed to fill the empty buffer using the method $mergeEF$, which merges from either input streams A and B to $bufferE$ or from streams C and D to $bufferF$. The algorithm $mergeEF$ merges for either the input streams A and B to $bufferE$ or from E and F to $bufferF$. It uses double buffering on the streams A, B, C, and D and ensures that there is always an active $dmaIn$ for these four input streams. Since the pseudocode is similar to that for $4wayMerge$, we do not provide this pseudocode here. Records are moved between buffers using the $memcpy$ instruction when the $byRecord$ layout is used and moved one field at a time when the layout is $ByField$..

B. SIMD 4-way Merge

The SIMD version differs from the scalar version only in the way each of the three 2-way merges comprising a 4-way merge works. These 2-way merges move 4 records at a time from input buffers to the output buffer. This is done using an adaptation of the odd-even merge scheme of Inoue et al. [9]

for merging numbers to the case of merging multifield records. Figure 9 gives the pseudocode for this adaptation.

```
Algorithm oddEvenMerge(v1, v2)
{// Merge records whose fields are in v1 and v2
  vector temp1, temp2, temp3, temp4;
  fields temp1Fields[], temp2Fields[];
  fields temp3Fields[], temp4Fields[];
  pattern = spu_cmpgt(v1, v2);
  temp1 = spu_select(v1, v2, pattern);
  temp2 = spu_select(v2, v1, pattern);
  temp1Fields = fields_select(v1Fields, v2Fields,
pattern);
  temp2Fields = fields_select(v2Fields, v1Fields,
pattern);
  // Stage 2
  temp3 = spu_slqwbyte(temp1, 8);
  pattern = spu_cmpgt(temp3, temp2);
  pattern = spu_shuffle(pattern, vZero, WXAC);
  temp1 = spu_select(temp3, temp2, pattern);
  temp4 = spu_select(temp2, temp3, pattern);
  temp2 = spu_shuffle(temp1, temp4, WACY);
  temp3 = spu_shuffle(temp1, temp4, ZXBD);
  temp3Fields = fields_rotate(temp1Fields, 8);
  temp1Fields = fields_select(temp3Fields,
temp2Fields, pattern);
  temp4Fields = fields_select(temp2Fields,
temp3Fields, pattern);
  temp2Fields = fields_shuffle(temp1Fields,
temp4Fields, WACY);
  temp3Fields = fields_shuffle(temp1Fields,
temp4Fields, ZXBD);
  // Stage 3
  pattern = spu_cmpgt(temp2, temp3);
  pattern = spu_shuffle(pattern, vZero, WXYA);
  temp1 = spu_select(temp2, temp3, pattern);
  temp4 = spu_select(temp3, temp2, pattern);
  temp1Fields = fields_select(temp2Fields,
temp3Fields, pattern);
  temp4Fields = fields_select(temp3Fields,
temp2Fields, pattern);
  // Stage 3
  v1 = spu_shuffle(temp1, temp4, ZWAX);
  v2 = spu_shuffle(temp1, temp4, BYCD);
  v1Fields = fields_shuffle(temp1Fields,
temp4Fields, ZWAX);
  v2Fields = fields_shuffle(temp1Fields,
temp4Fields, BYCD);
}
```

9: SIMD 2-way merge of 2 vectors $v1$ and $v2$

In Algorithm $oddEvenMerge$, $v1$ and $v2$ are two vectors each containing the keys of the next 4 records in the input buffers for the two streams being merged. It is easy to see that the next four records in the merged output are a subset of these 8 records and in fact are the 4 records (of these 8) with the smallest keys. Algorithm $oddEvenMerge$ determines these 4 smallest records and moves these to the output buffer.

XXXXShibdas, pl adapt algorithm to new terminology and add fields_rotate to section 2 XXXXXXXX

V. Experimental Results

We programmed several multifield record sorting algorithms using Cell BE SDK 3.1. Specifically, the following algorithms were coded and evaluated:

1) 2-way AA Sort ... this is the multifield record sorting algorithm of Inoue et al. [9]. This uses a comb sort variant for run generation and 2-way odd-even merge for run merging.
2) 4-way AA Sort ... this uses a comb sort variant for run generation as in [9] and our 4-way odd-even merge for run merging (Section IV-B).
3) 2-way Bitonic Sort ... this is an adaptation of the Cell-Sort algorithm of Gedik et al. [5] to multifield records (Section III). It uses bitonic sort for run generation and bitonic merge for run merging.
4) 4-way Bitonic Sort ... this uses bitonic sort for tun generation as in [5] and our 4-way odd-even merge for run merging (Section IV-B).
5) 2-way Merge Sort ... this uses an adaptation of the SPU merge sort algorithm of Shibdas and Sahni [1] to multifield records (Section III) for run generation and the 2-way odd-even merge of [9] for run merging.
6) 4-way Merge Sort ... this uses an adaptation of the SPU merge sort algorithm of Shibdas and Sahni [1] to multifield records (Section III) for run generation and our 4-way odd-even merge for run merging (Section IV-B).
7) 2-way Scalar Merge Sort ... this uses an adaptation of the SPU merge sort algorithm of Shibdas and Sahni [1] to multifield records (Section III) for run generation. Run merging is done using a 2-way scalar merging algorithm derived from the 4-way scalar merging algorithm of Section IV-A by eliminating the bottom left and the right 2-way merge nodes. for run merging (Section IV-A).
8) 4-way Scalar Merge Sort ... this uses an adaptation of the SPU merge sort algorithm of Shibdas and Sahni [1] to multifield records (Section III) for run generation and our 4-way scalar merge for run merging (Section IV-A).

We experimented with the above 8 multifield sorting algorithms using randomly generated input sequences. In our experiments, the number of 32-bit fields per record varied from 5 to 15 (in addition to the key field) and the number of records varied from 4K to 1M. Also, we tried both layouts– $ByField$ and $ByRecord$. For each combination of number of fields, number of records, and layout type, the time to sort 10 random sequences was obtained. The standard deviation in the observed run times was small and we report only the average times.

A. Run Times For $ByField$ Layout

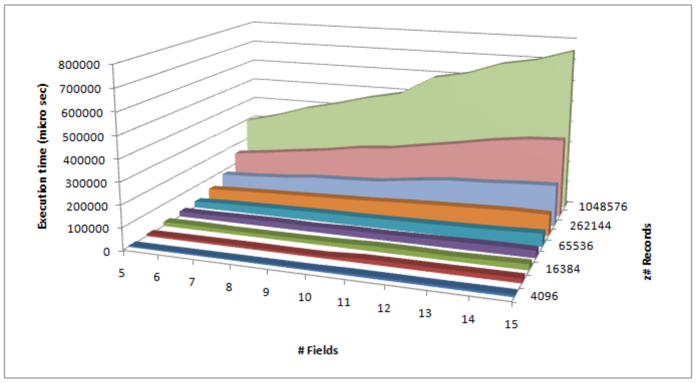
Figures 14 through 13 give the average run times for our 8 sorting algorithms using the $ByField$ layout and Figures 18 through 21 compare the average run times for the 2-way and 4-way versions of each of our sort algorithms for the case when the number of records to be sorted is 1M. For all our data, the 4-way version outperformed the 2-way version. For 1M records with 5 32-bit fields (in addition to a 32-bit key), the 4-way versions of AA Sort, Bitonic Sort, Merge Sort, and Scalar Merge Sort, respectively, took a%, b%, c%, and d% less time than taken by their 2-way counterparts and these percentages for 15 fields were e%, f%, g%, and h%.

10: 2-way AA-Sort ($ByField$)

11: 2-way Bitonic Sort ($ByField$)

12: 2-way Merge Sort ($ByField$)

13: 2-way Scalar Merge Sort ($ByField$)

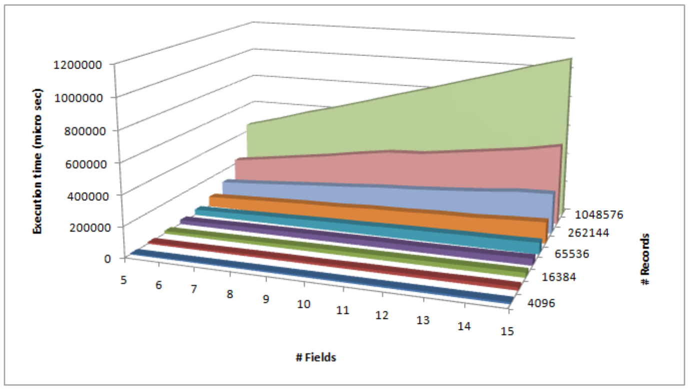


14: 4-way AA-Sort ($ByField$)

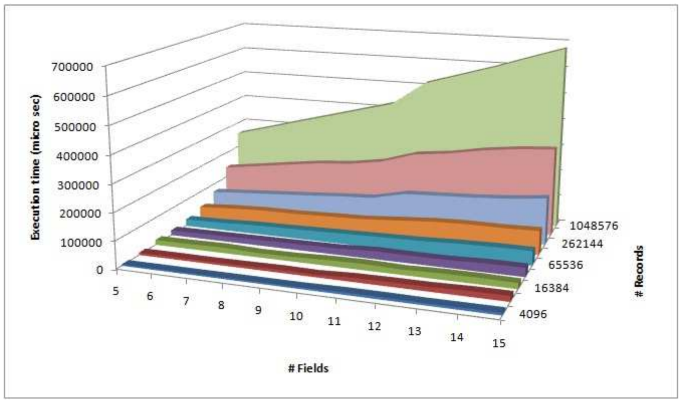
Figure 22 shows the run times for the 4 4-way sort algorithms for 1M records. As can be seen, 4-way Bitonic Sort is the slowest, followed by 4-way AA Sort, followed by 4-way Merge Sort; 4-way Scalar Merge Sort was the fastest. In fact, across all our data sets, 4-way Bitonic Sort took between 30% and 35% more time than taken by 4-way AA Sort, which in turn took between 10% and 15% more time than taken by 4-way Merge Sort. The fastest 4-way sort algorithm, 4-way Scalar Merge Sort took, respectively, between a% and b%, c% and d%, e% and f% less time than taken by 4-way AA Sort, 4-way Bitonic Sort, and 4-way Merge Sort.

B. Run Times For $ByRecord$ Layout

XXXXAdd similar material hereXXXXXXX

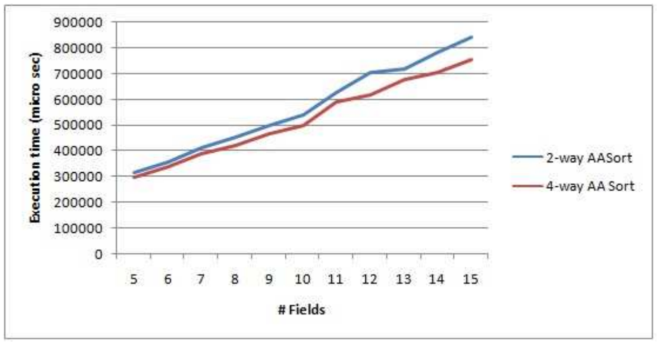


15: 4-way Bitonic Sort ($ByField$)



16: 4-way Merge Sort ($ByField$)

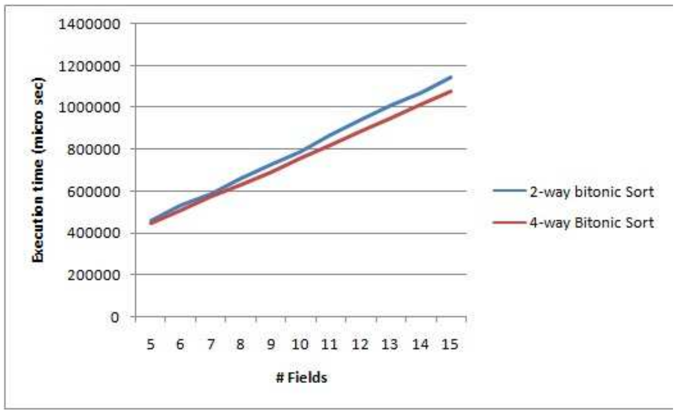
17: 4-way Scalar Merge Sort ($ByField$)



18: 2-way and 4-way Merge Sort ($ByField$), 1M records

C. Cross Layout Comparison

Although in a real application one may not be able to choose the layout format for the data to be sorted, it is worthwhile to compare the relative performance of the 8 sort methods using the better layout for each. This means that we use the $ByField$ layout for AA Sort and Bitonic Sort and the $ByRecord$ layout for Merge Sort and Scalar Merge Sort. Figure 24 gives the run times for the 4-way versions using

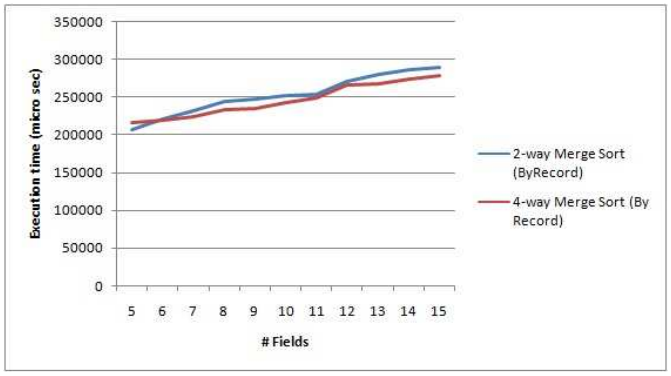


19: 2-way and 4-way Bitonic sort ($ByField$), 1M records

20: 2-way and 4-way Merge Sort ($ByField$), 1M records

21: 2-way and 4-way Scalar Merge Sort ($ByField$), 1M records

22: 4-way sorts ($ByField$), 1M records



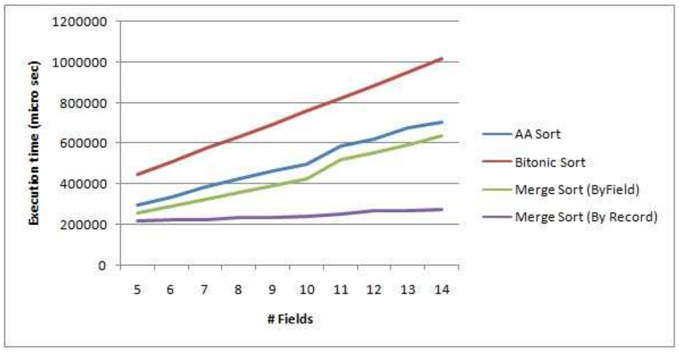
23: 2-way and 4-way Merge Sort ($ByRecord$)

these formats for the case of 1M records. Although Figure 24 is only for the case of 1M records, 4-way Scalar Merge Sort was the fastest for all of our data sets. For 5 32-bit fields (in addition to the key field) 4-way Scalar Merge Sort $ByRecord$) ran a% faster than 4-way Bitonic Sort ($ByField$), b% faster than 4-way AA Sort ($ByField$), and c% faster than 4-way Merge Sort ($ByRecord$). When the number of fields was 5, these percentages were ....

XXXX why is the 15 field data missing?XXXX

VI. Conclusion

We have shown how to adapt number sorts to sort multifield records on the Cell Broadband Engine. We also have developed two 4-way merge algorithms for the run merging phase. One of these is a scalar version and the other is an SIMD version. Our experiments indicate that the 4-way Scalar Merge



24: All Sorting algorithms in different layouts

Sort developed in this paper is the fastest method (from among those tested) to sort multifield records on the CBE.

References

[1] Bandyopadhyay, S. and Sahni, S., Sorting on a Cell Broadband Engine SPU, *IEEE International Symposium on Computers and Communications* (ISCC), 2009.
[2] Box, R. and Lacey, S., A fast, easy sort. *Byte*, 4, 1991, 315-318.
[3] Dobosiewicz, W., An efficient variation of bubble sort, *Information Processing Letters*, 11, 1980, 5-6.
[4] Drozdek, A., Worst case for Comb Sort, *Informatyka Teoretyczna i Stosowana*, 5, 9, 2005, 23-27.
[5] Gedik, B., Bordawekar, R., and Yu,P., CellSort: High performance sorting on the Cell processor, *VLDB*, 2007, 1286-1297.
[6] A.C. Chow, G.C. Fossum, and D.A. Brokenshire, A Programming Example: Large FFT on the Cell Broadband Engine.
[7] H. Hofstee, Power efficient processor architecture and the Cell Processor, *Proc. 11the International Symposium on High Performance Computer Architecture*, 2005.
[8] Horowitz, E., Sahni, S., and Mehta, D., Fundamentals of data structures in C++, Second Edition, Silicon Press, 2007.
[9] Inoue, H., Moriyama, T., Komatsu, H., and Nakatani, T., AA-sort: A new parallel sorting algorithm for multi-core SIMD processors, *16th International Conference on Parallel Architecture and Compilation Techniques* (PACT), 2007.
[10] Knuth, D., *The Art of Computer Programming: Sorting and Searching*, Volume 3, Second Edition, Addison Wesley, 1998.
[11] Lemke, P., The performance of randomized Shellsort-like network sorting algorithms, SCAMP working paper P18/94, Institute for Defense Analysis, Princeton, NJ, 1994.
[12] Sedgewick, R., Analysis of Shellsort and related algorithms, *4th European Symposium on Algorithms*, 1996.
[13] Sharma, D., Thapar, V., Ammar, R., Rajasekaran, S., and Ahmed, M., Efficient sorting algorithms for the Cell Broadband Engine, *IEEE International Symposium on Computers and Communications* (ISCC), 2008.
[14] Won, Y. and Sahni, S., Hypercube-to-host sorting, *Jr. of Supercomputing*, 3, 41-61, 1989.