Sorting On A Cell Broadband Engine SPU [*][†]

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

March 17, 2009

Abstract

We adapt merge sort for a single SPU of the Cell Broadband Engine. This adaptation takes advantage of the vector instructions supported by the SPU. Experimental results indicate that our merge sort adaptation is faster than other sort algorithms (e.g., AA sort, Cell sort, quick sort) proposed for the SPU as well as faster than our SPU adaptations of shaker sort and brick sort. An added advantage is that our merge sort adaptation is a stable sort whereas none of the other sort adaptations is stable.

Keywords
Cell broadband engine, SPU sorting, merge sort.

1 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 [6]. 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. The absence of branch prediction logic in an SPU and the availability of SIMD instructions that can operate on vectors that are comprised of 4 numbers poses a challenge when developing high performance CBE algorithms.

Recently, two sorting algorithms–AA-sort [8] and CellSort [4]–were proposed for the CBE. AA-sort is an adaptation of comb sort, which was originally proposed by Knuth [9] and rediscovered by Dobosiewicz [2] and Box and Lacey [1]. CellSort is an adaptation of bitonic sort (e.g., [9]). Both AA-sort and CellSort are based on sorting algorithms that are inefficient on a single processor. Hence, parallelizing these algorithms begins with a handicap relative to the fastest serial sorting algorithms–merge sort for worst-case behavior and quick sort for average behavior. Comb sort is known to have a worst-case complexity that is $O(n^2)$ [3]. Although the best upper bound known for its average complexity is also $O(n^2)$, experimental results indicate an average complexity of $O(n \log n)$ [1, 3]. On the other hand, the average complexity of quick sort is known to be $O(n \log n)$. Since experiments indicate that comb sort runs in about twice as much time on a single processor as does quick sort [3], attempts such as [8] to develop a fast average-case sort, for a single SPU of the CBE that begin with comb sort, are handicapped by a factor of two compared to attempts that begin with quick sort. This handicap needs to be overcome by the CBE adaptation.

For integers and floats, the CBE supports 4-way parallelism within a single SPU as 4 integers (floats) may be stored in each of the SPU's 128-bit vector registers. Hence, we expect at best two-fold speed up over a conventional implementation of quick sort. However, due to possible anomalous behavior resulting from such factors as the

[*]This research was supported, in part, by the National Science Foundation under grant ITR-0326155

[†]The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine Resource that have contributed to this research

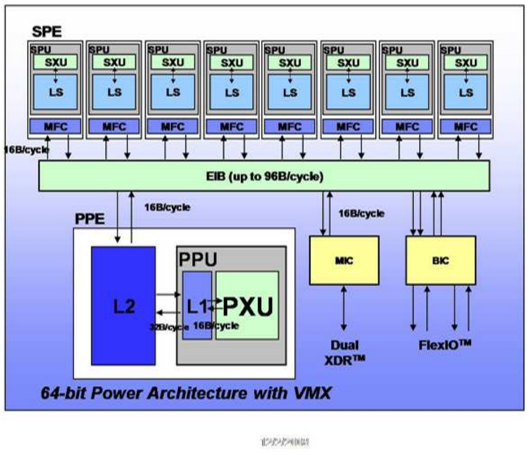

Figure 1: Architecture of the Cell Broadband Engine [5]

absence of branch prediction, we may actually observe a greater speed up [10]. Similarly, attempts such as [4] to develop a fast worst-case sort for a single SPU starting with bitonic sort are handicapped relative to starting with merge sort because the worst-case complexity of bitonic sort is $O(n \log^2 n)$ while that of merge sort is $O(n \log n)$.

As noted in [4], a logical way to develop a sorting algorithm for a heterogeneous multicore computer such as the CBE is to (1) begin with a sorting algorithm for a single SPU, then (2) using this as a core, develop a sort algorithm for the case when data fits in all available cores, then (3) use this multi-SPU algorithm to develop a sort algorithm for the case when the data to be sorted does not fit in the local stores of all available SPEs but fits in main memory. Our long-term strategy is to extend this hierarchical plan to the case where data to be sorted is so large that it is distributed over the main memories of a cluster of CBEs.

An alternative strategy to sort is to use the master-slave model in which the PPU serves as the master processor and the SPUs are the slave processors. The PPU partitions the data to be sorted and sends each partition to a different SPU; the SPUs sort their partition using a single SPU sort; the PPU merges the sorted data from the PPUs so as to complete the sort of the entire data set. This strategy is used in [14] to sort on the nCube hypercube and in [13] to sort on the CBE.

Regardless of whether we sort large data sets using the hierarchical strategy of [4] or the master-slave strategy of [13, 14], it is important to have a fast algorithm to sort within a single SPU. The absence of any branch prediction capability and the availability of vector instructions that support SIMD parallelism on an SPU make the development of a competitive merge sort a challenge, a challenge that we address in this paper. Our development of a competitive merge sort is specifically for the case of sorting integers and floats and makes use of the fact that 4 integers or floats may be stored in a single register of an SPU. In Section 2, we describe SIMD functions that are used in the specification of our SPU algorithms. Comb sort and its SPU adaptation AA-sort are reviewed in Section 3. In this section, we develop SPU adaptations for brick sort and shaker sort as well. Our SPU adaptation of merge sort is developed in Section 4 and experimental results comparing various SPU sorting algorithms are presented in Section 5. In the remainder of this paper, we use the term *number* to refer to an integer or a float. Note that 4 numbers may be stored in a 128-bit vector.

2 SPU Vector Operations

We shall use several SIMD functions that operate on a vector of 4 numbers to describe the SPU adaptation of sorting algorithms. We describe these in this section. In the following, $v1$, $v2$, min, max, and $temp$ are vectors, each comprised of 4 numbers and p, $p1$, and $p2$ are bit patterns. Function names that begin with spu are standard C/C++ Cell SPU intrinsics while those that begin with $mySpu$ are defined by us. 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_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.

3. *spu_add*$(v1, v2) \cdots$ Returns the vector obtained by pairwise adding the numbers of $v1$ with the corresponding numbers of $v2$.

4. *spu_sub*$(v1, v2) \cdots$ Returns the vector obtained by pairwise subtracting the numbers of $v2$ from the corresponding numbers of $v1$.

5. *spu_and*$(p1, p2) \cdots$ Returns the vector obtained by pairwise anding the bits of $p1$ and $p2$.

6. *mySpu_not*$(p) \cdots$ Returns the vector obtained by complementing each of the bits of p. Although the CBE does not have a *not* instruction, we can perform this operation using the *nor* function that is supported by the CBE and which computes the complement of the bitwise *or* of two vectors. It is easy to see that *spu_nor*$(p, v0)$ where $v0$ is an all zero vector, correctly computes the complement of the bits of p.

7. *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).

8. *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.

9. *spu_splat*$(s) \cdots$ Returns a vector comprised of 4 copies of the number s.

10. *mySpu_cmpswap*$(v1, v2) \cdots$ Pairwise compares the numbers of $v1$ and $v2$ and swaps so that $v1$ has the smaller number of each compare and $v2$ has the larger number Specifically, the following instructions are executed:
$p = spu\_cmpgt(v1, v2);$
$min = spu\_select(v1, v2, p);$
$v2 = spu\_select(v2, v1, p);$
$v1 = min;$

11. *mySpu_cmpswap_skew*$(v1, v2) \cdots$ Performs the comparisons and swaps shown in Figure 2. Specifically, the following instructions are executed:
$temp = spu\_slqwbyte(v2, 4);$
$p = spu\_cmpgt(v1, temp);$
$min = spu\_select(v1, temp, p);$
$v1 = spu\_shuffle(min, v1, WXYD);$
$max = spu\_select(temp, v1, p);$
$v2 = spu\_shuffle(max, v2, AWXY);$

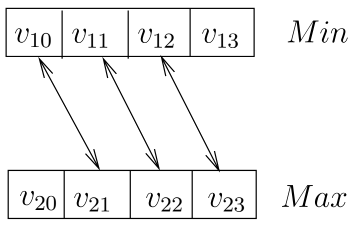

Figure 2: Comparisons for *mySpu_cmpswap_skew*

12. $mySpu\_gather(vArray, v1)$ $\cdots$ Here $vArray$ is an array of vectors. Let W, X, Y and Z be the numbers of $v1$. The function returns a vector whose first number is the first number of $vArray[W]$, its second number is the second number of $vArray[X]$, its third number is the third number of $vArray[Y]$, and its fourth number is the fourth number of $vArray[Z]$. One implementation of this function first extracts W, X, Y, and Z from $v1$ using the function *spu_extract* and then executes the code:
    $temp = spu\_shuffle(vArray[W], vArray[X], WBWW)$;
    $temp = spu\_shuffle(temp, vArray[Y], WXCC)$;
    return $spu\_shuffle(temp, vArray[Z], WXYD)$;

13. $mySpu\_gather12(vArray, v1)$ $\cdots$ This function, which is similar to $mySpu\_gather$, returns a vector whose first number is the first number of $vArray[W]$ and whose second number is the second number of $vArray[X]$. The third and fourth numbers of the returned vector are set arbitrarily. Its code is:
    return $spu\_shuffle(vArray[W], vArray[X], WBWW)$;

14. $mySpu\_gather34(vArray, v1)$ $\cdots$ This function, which is similar to $mySpu\_gather12$, returns a vector whose first number is the third number of $vArray[W]$ and whose second number is the fourth number of $vArray[X]$. The third and fourth numbers of the returned vector are set arbitrarily. Its code is:
    return $spu\_shuffle(vArray[W], vArray[X], YDYY)$;

15. $mySpu\_gatherA(vArray, v1)$ $\cdots$ This function is similar to $mySpu\_gather$ and returns a vector whose first number is the first number of $vArray[W]$, its second number is the third number of $vArray[X]$, its third number is the first number of $vArray[Y]$, and its fourth number is the third number of $vArray[Z]$. The code:
    $temp = spu\_shuffle(vArray[W], vArray[X], WCWW)$;
    $temp = spu\_shuffle(temp, vArray[Y], WXAA)$;
    return $spu\_shuffle(temp, vArray[Z], WXYC)$;

16. $mySpu\_gatherB(vArray, v1)$ $\cdots$ This too is similar to $mySpu\_gather$. The function returns a vector whose first number is the second number of $vArray[W]$, its second number is the fourth number of $vArray[X]$, its third number is the second number of $vArray[Y]$, and its fourth number is the fourth number of $vArray[Z]$. The code is:
    $temp = spu\_shuffle(vArray[W], vArray[X], XDXX)$;
    $temp = spu\_shuffle(temp, vArray[Y], WXBB)$;
    return $spu\_shuffle(temp, vArray[Z], WXYD)$;

3 Shellsort Variants

Shellsort [9] sorts a sequence of n numbers in m passes employing a decreasing increment sequence $i_1 > i_2 > \cdots > i_m = 1$. In the jth pass, increment $h = i_j$ is used; the sequence is viewed as comprised of h subsequences with the kth subsequence comprised of the numbers in positions k, $k+h$, $k+2h$, $\cdots$, of the overall sequence, $0 \leq k < h$; and each subsequence is sorted. The sorting of the subsequences done in each pass is called an h-sort. While an h-sort is typically accomplished using insertion sort, other simple sorting algorithms such as bubble sort may also be used. With the proper choice of increments, the complexity of Shellsort is $O(n \log^2 n)$ [9]. Shellsort variants replace the h-sort used in each pass of Shellsort with an h-pass that only partially sorts the subsequences. For example, in an h-bubble pass we make only the first pass of bubble sort on each subsequence. Since replacing h-sort by h-pass in Shellsort no longer guarantees a complete sort, we follow with some simple sorting algorithm such as bubble sort to complete the sort. So, the h-passes may be viewed as preprocessing passes done so as to improve the performance of the ensuing sort algorithm In Shellsort, $i_m = 1$ is used to assure that the sequence is sorted following the final h-sort. However, in a Shellsort variant, this assurance comes from the sort algorithm run following the preprocessing h-passes. So, the h-pass with $h = 1$ is typically skipped. The general structure of a Shellsort variant is:

Step 1 [Preprocess] Perform h-passes for $h = i_j$, $1 \le j < m$.

Step 2 [Sort] Sort the preprocessed sequence

3.1 Comb and AA Sort

Knuth [9] proposed a Shellsort variant in which each h-pass is a bubble pass (Figure 3). This variant was rediscovered later by Dobosiewicz [2] and Box and Lacey [1]. Box and Lacey [1] named this variant *comb sort*. The increment sequence used by comb sort is geometric with factor s. Dobosiewicz [2] has shown that the preprocessing step sorts $a[0 : n-1]$ with very high probability whenever $s < 1.33$. As a result, $s = 1.3$ is recommended in practice (note that a larger s requires a smaller number of h-passes). With this choice, the outer **for** loop of the second step (bubble sort) is entered only once with high probability and the complexity of comb sort is $O(n \log n)$ with high probability. Experiments indicate that the algorithm's average run time is close to that of quick sort [3]. However, the worst-case complexity of comb sort is $O(n^2)$ [3].

```
Algorithm combsort(a,n)
{// sort a[0:n-1]
    // Step 1:  Preprocessing
    for (h = n/s; h > 1; h /= s) {
       // h-bubble pass
       for (i = 0; i < n-h; i++)
         if (a[i] > a[i+h]) swap(a[i],a[i+h]);
    }
    sorted = false;
    // Step 2:  Bubble sort
    for (pass = 1; pass < n && !sorted; pass++) {
       sorted = true;
       for (i = 0; i < n-pass; i++)
         if (a[i] > a[i+1]) {swap(a[i],a[i+1]); sorted = false;}
    }
}
```

Figure 3: Comb sort

Inoue et al. [8] have adapted comb sort to the CBE to obtain the sort method AA-sort, which efficiently sorts numbers using all 8 SPUs of a CBE. We describe only the single SPU version of AA-sort here as this paper's focus is sorting using a single SPU. The single SPU version begins with a vector array $d[0 : r-1]$ of numbers; each vector d[i] has 4 numbers. Hence, d is an $r \times 4$ matrix of numbers. This matrix is first sorted into column-major order and then the numbers permuted so as to be sorted in row-major order. Figure 4 gives the algorithm for the column-major sort and Figure 6 gives the column-major to row-major reordering algorithm.

The column-major to row-major reordering is done in two steps. In the first step, the numbers in each 4×4 submatrix of the $r \times 4$ matrix of numbers are transposed so that each vector now has the 4 numbers in some row of the result. For simplicity, we assume that r is a multiple of 4. In the second step, the vectors are permuted into the correct order. For the first step, we collect the first and second numbers in rows 0 and 2 of the 4×4 matrix being transposed into the vector $row02A$. The third and fourth numbers of these two rows are collected into $row02b$. The same is done for rows 1 and 3 using vectors $row13A$ and $row13B$. Figure 5 shows this rearrangement. Then, the transpose is constructed from the just computed 4 vectors.

3.2 Brick Sort

In brick sort, we replace the h-bubble pass of comb sort by an h-brick pass [11, 12] in which we first compare-exchange positions i, $i+2h$, $i+4h$, $\cdots$ with positions $i+h$, $i+3h$, $i+5h$, $\cdots$, $0 \le i < h$ and then we compare-exchange positions $i+h$, $i+3h$, $i+5h$, $\cdots$ with positions $i+2h$, $i+4h$, $i+6h$, $\cdots$, $0 \le i < h$. Figure 7 gives our CBE adaptation of the preprocessing step (Step 1) for brick sort. Step 2 is a bubble sort as was the case for AA-sort. The bubble sort needs to be followed by a column-major to row-major reordering step (Figure 6). It is known that

```
Algorithm AA(d,r)
{// sort d[0:r-1] into column-major order
   // Step 1:  Preprocessing
   for (i = 0; i < r; i++) sort(d[i]);
   for (h = r; h > 1; h /= s) {
      // h-bubble pass
      for (i = 0; i < r-h; i++)
         mySpu_cmpswap(d[i],d[i+h]);
      for (i = r-h; i < r; i++)
         mySpu_cmpswap_skew(d[i],d[i+h-r]);
   }
   sorted = false;
   // Step 2:  Bubble sort
   do {
      for (i = 0; i < r-1; i++)
        mySpu_cmpswap(d[i],d[i+1]);
     mySpu_cmpswap_skew(d[r-1],d[0]);
   } while (not sorted)
}
```

Figure 4: Single SPU column-major AA-sort [8]

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

(a) Initial

a	i	b	j
c	k	d	l
e	m	f	n
g	o	h	p

(b) Rows 02A, 02B, 03A, 03B, top to bottom

a	e	i	m
b	f	j	n
c	g	k	o
d	h	l	p

(c) Transpose

Figure 5: Collecting numbers from a 4×4 matrix

the preprocessing step of brick sort nearly always does a complete sort when the increment sequence is geometric with shrink factor (i.e., s) less than 1.22 [11, 12]. Hence, when we use $s < 1.22$, the **do-while** loop of Step 2 (bubble sort) is entered only once (to verify the data are sorted) with high probability.

3.3 Shaker Sort

Shaker sort differs from comb sort in that h-bubble passes are replaced by h-shake passes. An h-shake pass is a left-to-right bubble pass as in comb sort followed by a right-to-left bubble pass. Figure 8 gives our CBE adaptation of shaker sort. The preprocessing step of shaker sort almost always sorts the data when the shrink factor s is less than 1.7.

4 Merge Sort

Unlike the Shellsort variants comb, brick, and shaker sort of Section 3 whose complexity is $O(n \log n)$ with high probability, the worst-case complexity of merge sort is $O(n \log n)$. Further, merge sort is a stable sort (i.e., the relative order of elements that have the same key is preserved). While this property of merge sort isn't relevant when we are simply sorting numbers (as you can't tell two equal numbers apart), this property is useful in some applications where each element has several fields, only one of which is the sort key. The Shellsort variants of Section 3 are not stable sorts. On the down side, efficient implementations of merge sort require added space. When sorting numbers in the vector array $d[0 : r-1]$ we need an additional vector array $t[0 : r-1]$ to support the merge. We present our CBE merge sort adaptation in the context of a stable sort and later point out the

```
Algorithm transpose(d,r)
{// Column major to row major reordering
   // Step 1:  Transpose 4 x 4 submatrices
   for (i = 0; i < r; i += 4) {
      // Compute row02A, row02B, row13A, and row13B
      row02A = spu_shuffle(d[i], d[i+2], WAXB);
      row02B = spu_shuffle(d[i], d[i+2], YCZD);
      row13A = spu_shuffle(d[i+1], d[i+3], WAXB);
      row13B = spu_shuffle(d[i+1], d[i+3], YCZD);
      // Complete the transpose
      d[i] = spu_shuffle(row02A, row13A, WAXB);
      d[i+1] = spu_shuffle(row02A, row13A, YCZD);
      d[i+2] = spu_shuffle(row02B, row13B, WAXB);
      d[i+3] = spu_shuffle(row02B, row13B, YCZD);
   }
   // Step 2:  Reorder vectors
   for (i = 0; i < r; i++)
      if (!inPlace[i]) {
         current = i;
         next = i/(r/4) + (i mod (r/4))*4;
         temp = d[i];
         while (next != i) {// follow cycle
            d[current] = d[next];
            inPlace[current] = true;
            current = next;
            next = current/(r/4) + (current mod (r/4))*4;
         }
         d[current] = temp;
         inPlace[current] = true;
      }
}
```

Figure 6: Column major to row major

simplifications that are possible when we wish to sort numbers rather than elements that have multiple fields. We assume that the element keys are in the vector array $d[0 : r-1]$ and that the remaining fields are stored elsewhere. Our discussion focuses only on the needed data movements for the keys; each time a key is moved, the remaining fields associated with this key need also to be moved (alternatively, we may use the table sort strategy discussed in [7] to first determine the sorted permutation and later rearrange the remaining fields of the elements being sorted).

There are 4 phases to our stable merge sort adaptation:

Phase 1: Transpose the elements of $d[0 : r-1]$, which represents a $r \times 4$ matrix, from row-major to column-major order.

Phase 2: Sort the 4 columns of the $r \times 4$ matrix independently and in parallel.

Phase 3: In parallel, merge the first 2 columns together and the last 2 columns together to get two sorted sequences of length $2r$ each.

Phase 4: Merge the two sorted sequences of length $2r$ each into a row-major sorted sequence of length $4r$.

We note that Phase 1 is needed only when we desire a stable sort. Figure 9 shows an initial 8×4 matrix of numbers and the result following each of the 4 phases of our merge sort adaptation.

The Phase 1 transformation is the inverse of the column-major to row-major transformation done in Figure 6 and we do not provide its details. Details for the remaining 3 phases are provided in the following subsections.

```
Algorithm Brick(d,r)
{// sort d[0:r-1] into column-major order
   // Step 1:  Preprocessing
   for (i = 0; i < r; i++) sort(d[i]);
   for (h = r; h > 1; h /= s) {
      // h-brick pass
      // compare-exchange even:odd bricks
      for (i = 0; i < r-2*h; i += 2*h)
         for (j = i; j < i + h; j++)
            mySpu_cmpswap(d[j],d[j+h]);
      // handle end conditions
      if (j < n - h) {// More than 1 brick remains
      end = j + h;
      for (; j < n - h; j++)
         mySpu_cmpswap(d[j],d[j+h]);
      }
      else end = r;
      while (j < end) {
         mySpu_cmpswap_skew(d[j],d[j+h-n]);
         j++;
      }
      // compare-exchange odd:even bricks beginning with i = h
      // similar to even:odd bricks
   // Step 2:  Bubble sort
   // same as for AA-sort
}
```

Figure 7: Column-major brick sort

```
Algorithm Shaker(d,r)
{// sort d[0:r-1] into column-major order
   // Step 1:  Preprocessing
   for (i = 0; i < r; i++) sort(d[i]);
   for (h = r; h > 1; h /= s) {
      // h-shake pass
      // left-to-right bubble pass
      for (i = 0; i < r-h; i++)
         mySpu_cmpswap(d[i],d[i+h]);
      for (i = r-h; i < r; i++)
         mySpu_cmpswap_skew(d[i],d[i+h-r]);
      // right-to-left bubble pass
      for (i = r-h-1; i > 0; i--)
         mySpu_cmpswap(d[i],d[i+h]);
      for (i = r-1; i >= r - h; i--)
         mySpu_cmpswap_skew(d[i],d[i+h-r]);
   }
   // Step 2:  Bubble sort
   // Same as for AA-sort
}
```

Figure 8: Column-major shaker sort

22	30	5	17
14	26	32	9
25	6	20	10
2	28	16	11
19	13	29	1
21	4	24	7
15	23	8	31
12	18	27	3

(a) Initial

22	25	19	15
30	6	13	23
5	20	29	8
17	10	1	31
14	2	21	12
26	28	4	18
32	16	24	27
9	11	7	3

(b) Phase 1

5	2	1	3
9	6	4	8
14	10	7	12
17	11	13	15
22	16	19	18
26	20	21	23
30	25	24	27
32	28	29	31

(c) Phase 2

2	17	1	18
5	20	3	19
6	22	4	21
9	25	7	23
10	26	8	24
11	28	12	27
14	30	13	29
16	32	15	31

(d) Phase 3

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
17	18	19	20
21	22	23	24
25	26	27	28
29	30	31	32

(e) Phase 4

Figure 9: Merge sort example

4.1 Merge Sort Phase 2–Sort Columns

Phase 2 operates in $\log r$ subphases characterized by the size of the sorted segments being merged. For instance, in the first subphase, we merge together pairs of sorted segments of size 1 each, in the next subphase the segment size is 2, in the third it is 4, and so forth. At any time, the two segments being merged have the same physical locations in all 4 columns. So, for our 8×4 example, when merging together segments of size 2, we shall first merge, in parallel, 4 pairs of segments, one pair from each column. The first segment of a pair is in rows 0 and 1 of the $r \times 4$ matrix and the second in rows 2 and 3. Then, we shall merge together the segments of size 2 that are in rows 4 through 7. Following this, the segment size becomes 4.

To merge 4 pairs of segments in parallel, we employ 8 counters to keep track of where we are in the 8 segments being merged. The counters are called $a_0, \cdots, a_3, b_0, \cdots, b_3$. (a_i, b_i) are the counters for the segments of column i, $0 \leq i \leq 3$ that are being merged. When the segment size is s and the segments being merged occupy rows i through $i + 2s - 1$, the a counters are initialized to i and the b counters to $i + s$. Although all a counters have the same value initially as do all b counters, as merging progresses, these counters have different values. Figure 10 gives the Phase 2 algorithm. For simplicity, we assume that r is a power of 2.

```
Algorithm Phase2(d,r)
{// sort the 4 columns of d[0:r-1], use additional array t[0:r-1]
   for (s = 1; s < r; s *= 2)
      for (i = 0; i < r; i += 2*s) {
         A = spu_splats(i); // initialize a counters
         B = spu_splats(i+s); // initialize b counters
         for (k = i; k < i + 2*s; k++) {// merge the segments
            // one round of compares
            aData = mySpu_gather(d,A);
            bData = mySpu_gather(d,B);
            p = spu_cmpgt(aData,bData);
            t[k] = spu_select(aData,bData,p);
            // update counters
            notP = mySpu_not(p);
            A = spu_sub(A,notP);
            B = spu_sub(B,p);
         }
         swap(d,t); // swap roles
      }
}
```

Figure 10: Phase 2 of merge sort

4.2 Merge Sort Phase 3–Merge Pairs of Columns

In Phase 3 we merge the first two and last two columns of the $r \times 4$ matrix together to obtain 2 sorted sequences, each of size $2r$. The first sequence is in columns 0 and 1 and the second in columns 2 and 3 of an output matrix. We do this merging by using 8 counters. Counters a_0, b_0, a_1, b_1 start at the top of the 4 columns of our matrix and move downwards while counters a_2, b_2, a_3, b_3 start at the bottom and move up (see Figure 11(a)). Let $e(c)$ be the matrix element that counter c is at. The comparisons $e(a_i) : e(b_i)$, $0 \le i \le 3$ are done in parallel and depending on the outcome of these comparisons, 4 of the 8 elements compared are moved to the output matrix. When $e(a_0) \le e(b_0)$ ($e(a_1) \le e(b_1)$), $e(a_0)$ ($e(a_1)$) is moved to the output and a_0 (a_1) incremented by 1; otherwise, $e(b_0)$ ($e(b_1)$) is moved to the output and b_0 (b_1) incremented by 1. Similarly, when $e(a_2) \le e(b_2)$ ($e(a_3) \le e(b_3)$), $e(b_2)$ ($e(b_3)$) is moved to the output and b_2 (b_3) decremented by 1; otherwise, $e(a_2)$ ($e(a_3)$) is moved to the output and a_2 (a_3) decremented by 1. The merge is complete when we have done r rounds of comparisons. Figure 12 gives the algorithm for Phase 3.

Theorem 1 *Algorithm $Phase3$ correctly merges 4 sorted columns into 2.*

Proof To prove the correctness of Algorithm $Phase3$ we need to show that each element of the first (last) two columns of the input $r \times 4$ matrix is copied into the first (last) two columns of the output matrix exactly once and that the elements of the first (third) output column followed by those of the second (fourth) are in sorted order. It is sufficient to show this for the first two columns of the input and output matrices. First, observe that when $a_0 \le a_2$ ($b_0 \le b_2$), these counters are at input elements that have yet to be copied to the output. Further, when $a_0 > a_2$ ($b_0 > b_2$) all elements of the respective column have been copied from the input to the output (note that a counter is updated only when its element has been copied to the output matrix). We consider 4 cases: $a_0 < a_2$, $a_0 = a_2$, $a_0 = a_2 + 1$, and $a_0 > a_2 + 1$.

Case $a_0 < a_2$ When $b_0 < b_2$ (Figure 11(a)), exactly one of $e(a_0)$ and $e(b_0)$ and one of $e(a_2)$ and $e(b_2)$ are copied to the output and the corresponding counters are advanced. No element is copied to the output twice.

Next, consider the case $b_0 = b_2$ (Figure 11(b)). If $e(a_0) \le e(b_0)$, $e(a_0)$ and one of $e(a_2)$ and $e(b_2)$ are copied to the output and the corresponding counters advanced. Again no element is copied to the output twice. If $e(a_0) > e(b_0) = e(b_2)$, $e(b_2) < e(a_0) \le e(a_2)$ and $e(b_0)$ and $e(a_2)$ are copied to the output and their counters advanced. Again, no element is copied twice.

The next case we consider has $b_0 = b_2 + 1$. Let the values of b_0 and b_2 be b_0' and b_2' just before the update(s) that resulted in $b_0 = b_2 + 1$ and let a_0' and a_2' be the values of the a counters at this time. One of the following must be true: (a) $b_2' = b_0' + 1$ (both b_0 and b_2 were advanced, Figure 11(c)), (b) $b_0' = b_2' = b_0$ (only b_2 was advanced, Figure 11(d)), or (c) $b_0' = b_2' = b_2$ (only b_0 was advanced, Figure 11(e)). In (a), it must be that $b_2 = b_0'$ and $b_0 = b_2'$. So, $e(a_0) > e(b_0')$ and $e(a_2) \le e(b_2')$. Hence, $e(a_0) \le e(a_2) \le e(b_2') = e(b_0)$ and $e(a_2) \ge e(a_0) > e(b_0') = e(b_2)$. Therefore, $e(a_0)$ and $e(a_2)$ are copied to the output and a_0 and a_2 advanced. Again, only previously uncopied elements are copied to the output and each is copied once. For subcase (b), when b_2' was decremented to b_2, a_0' was incremented to a_0, $e(b_2') \ge e(a_2)$ and $e(a_0') \le a(b_0')$. Since $b_0 > b_2$, all elements of the second column have been copied to the output. We see that $e(a_0) \le e(a_2) \le e(b_2') = e(b_0)$. So, $e(a_0)$ is copied and a_0 is advanced. Further, as a result of some previous comparison, b_0 was advanced to its current position from the present position of b_2. So, there is an $a_0'' \le a_0$ such that $e(b_2) < e(a_0'') \le e(a_0) \le e(a_2)$. Therefore, $e(a_2)$ is copied and a_2 advanced. Again, no previously copied element is copied to the output and no element is copied twice. Subcase (c) is symmetric to subcase (b).

The final case has $b_0 > b_2 + 1$ (Figure 11(f)). From the proof of subcases $b_0 = b_2$ and $b_0 = b_2 + 1$, it follows that this case cannot arise.

Case $a_0 = a_2$ There are 4 subcases to consider–(a) $b_0 < b_2$, (b) $b_0 = b_2$, (c) $b_0 = b_2 + 1$, and (d) $b_0 > b_2 + 1$ (Figures 11(g–j)). Subcase (a) is symmetric to the case $a_0 < a_2$ and $b_0 = b_2$ considered earlier. In subcase (b), independent of the outcome of the comparison $e(a_0) : e(b_0)$, which is the same as the comparison $e(a_2) : e(b_2)$, $e(a_0)$ (equivalently $e(a_2)$) and $e(b_0)$ (equivalently $e(b_2)$) are copied to the output. For subcase (c), we notice that when $a_0 = a_2$, these two counters have had a cumulative advance of $r - 1$ from their initial values and when $b_0 = b_2 + 1$ these two counters have together advanced by r. So, the 4 counters together have advanced by $2r - 1$ from their initial values. This isn't possible as the 4 counters advance by a total of 2 in each

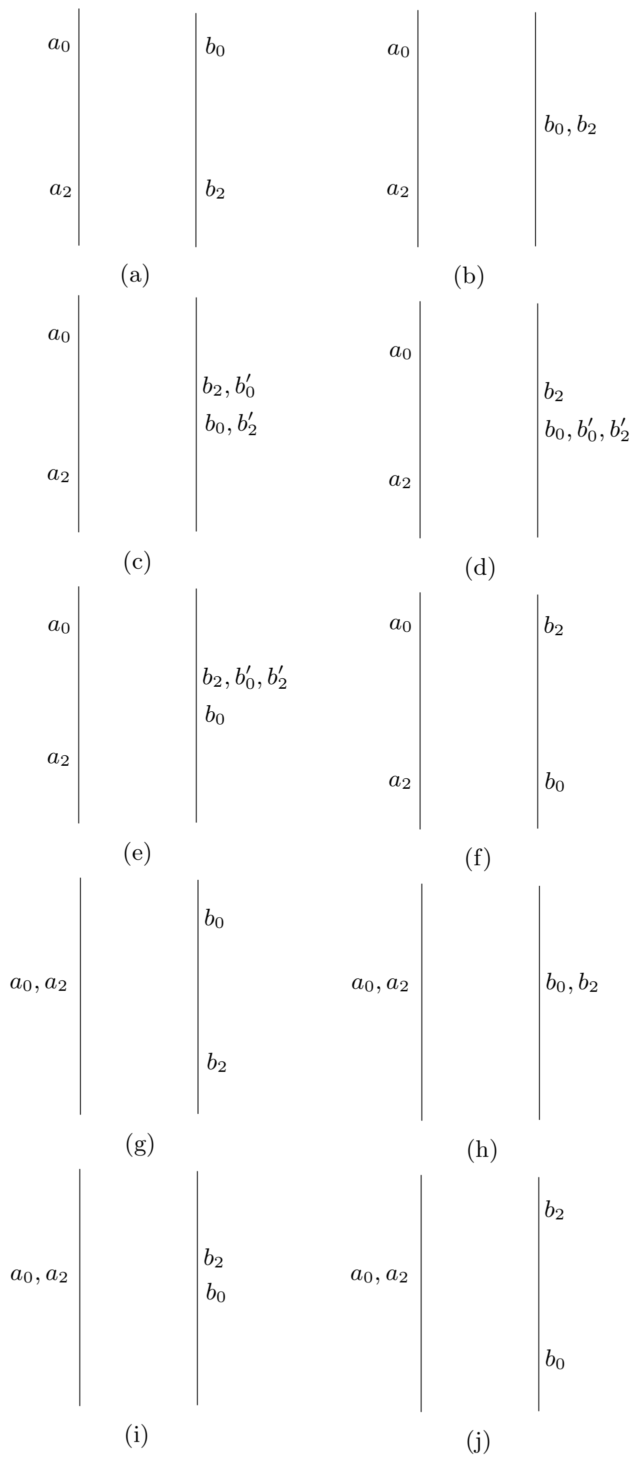

Figure 11: Phase 3 counters

iteration of the **for** loop. So, subcase (c) cannot arise. Next, consider subcase (d). From the proof for the case $a_0 < a_2$, we know that we cannot have $b_0 > b_2 + 1$ while $a_0 < a_2$. So, we must have got into this state from a state in which $a_0 = a_2$ and $b_0 \leq b_2$. It isn't possible to get into this state from subcase (a) as subcase

```
Algorithm Phase3(d,r)
{// merge the 4 sorted columns of d[0:r-1] into 2 sorted sequences
// use additional array t[0:r-1]
   A = {0, 0, r-1, r-1}; // initialize a counters
   B = {0, 0, r-1, r-1}; // initialize b counters
   for (k = 0; k < r; i++) {
      aData = mySpu_gatherA(d,A);
      bData = mySpu_gatherB(d,B);
      p = spu_cmpgt(aData,bData);
      e = spu_equal(aData,bData);
      e = spu_and(e, vector(0,0,-1,-1));
      p = spu_or(p, e);
      min = spu_select(aData, bData, p);
      max = spu_select(bData, aData, p);
      t[k] = spu_shuffle(min,t[k],WBXD);
      t[r-k-1] = spu_shuffle(max,t[r-k-1],AYCZ);
      // update counters
      notP = mySpu_not(p);
      f1 = spu_and(p,vector(-1,-1,0,0));
      s1 = spu_and(p,vector(0,0,-1,-1));
      f2 = spu_and(notP,vector(-1,-1,0,0));
      s2 = spu_and(notP,vector(0,0,-1,-1));
      A = spu_sub(A,f2);
      A = spu_add(A,s2);
      B = spu_sub(B,f1);
      B = spu_add(B,s1);
   }
}
```

Figure 12: Phase 3 of merge sort

(a), at worst increases b_0 by 1 and decreases b_2 by 1 each time we are in this subcase. So, it is possible to get into this subcase only from subcase (b). However, subcase (b) only arises at the last iteration of the for loop. Even otherwise, subcase (b) either increments b_0 by 1 or decrements b_2 by 1 and so cannot result in $b_0 > b_2 + 1$.

Case $a_0 > a_2 + 1$ From the proofs of the remaining cases, it follows that this case cannot arise.

■

From the proof of Theorem 1, it follows that when we are sorting numbers rather than multi-field elements with numeric keys, algorithm *Phase*3 works correctly even with the statements

```
e = spu_equal(aData,bData);
e = spu_and(e, vector(0,0,-1,-1));
p = spu_or(p, e);
```

omitted.

4.3 Merge Sort Phase 4–Final Merge

For the Phase 4 merge, we employ 4 counters. Counters a_0 and a_1, respectively begin at the first and last element of the first sorted sequence (i.e., at the top of the first column and bottom of the second column, respectively) while b_0 and b_1 begin at the first and last elements of the second sequence (Figure 13). In each round, the comparisons $a_0 : b_0$ and $a_1 : b_1$ are done in parallel. $e(a_0)$ $(e(b_1))$ is moved to the output if $e(a_0) \leq e(b_0)$ $(e(b_1) \geq e(a_1))$. Otherwise, $e(b_0)$ $(e(a_1))$ is moved to the output. The sorted output is assembled in row-major order in the vector array t. We use the variables k and pos to keep track of the row and column in t in which to place the output element from the comparison $e(a_0) : e(b_0)$. The output element from $e(a_1) : e(b_1)$ goes into row $(r - k - 1)$ and column $(3 - pos)$ of t. Figure 14 gives the algorithm for the case when the counters remain within the bounds of their respective columns. $mask[pos]$, $0 \leq pos \leq 3$ is defined so as to change only the number in position pos of a $t[]$ vector.

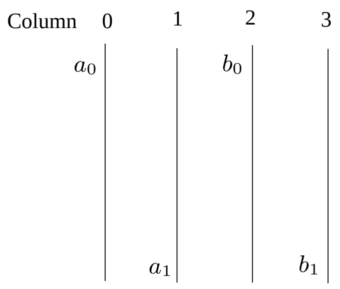

Figure 13: Phase 4 counters

```
Algorithm Phase4(d,r)
{// partial algorithm to merge 2 sorted sequences of d[0:r-1]
// into 1 sorted sequence
// use additional array t[0:r-1]
    A = {0, r-1, 0, 0}; // initialize a counters
    B = {0, r-1, 0, 0}; // initialize b counters
    k = 0; pos = 0;
    while (no column is exhausted) {
       aData = mySpu_gather12(d,A);
       bData = mySpu_gather34(d,B);
       p = spu_cmpgt(aData,bData);
       e = spu_equal(aData,bData);
       e = spu_and(e, vector(0,-1,0,0));
       p = spu_or(p, e);
       min = spu_select(aData, bData, p);
       max = spu_select(bData, aData, p);
       max = spu_slqwbyte(max, 4);
       t[k] = spu_shuffle(min,t[k],mask[pos]);
       t[r-k-1] = spu_shuffle(max,t[r-k-1],mask[3-pos]);
       // update counters
       notP = mySpu_not(p);
       f1 = spu_and(p,vector(-1,0,0,0));
       s1 = spu_and(p,vector(0,-1,0,0));
       f2 = spu_and(notP,vector(-1,0,0,0));
       s2 = spu_and(notP,vector(0,-1,0,0));
       A = spu_sub(A,f2);
       A = spu_add(A,s1);
       B = spu_sub(B,f1);
       B = spu_add(B,s2);
       k += (pos+1)/4;
       pos = (pos+1)%4;
    }
}
```

Figure 14: Phase 4 of merge sort with counters in different columns

As was the case in Phase 3, the statements

```
e = spu_equal(aData,bData);
e = spu_and(e, vector(0,0,-1,-1));
p = spu_or(p, e);
```

may be omitted when we are sorting numbers rather than multi-field elements with numeric keys.

5 Experimental Results

We programmed our merge sort, brick sort, and shaker sort adaptations using the CBE SDK Version 3.0. For comparison purposes, we used an AA sort code developed by us, the Cell sort code of [4], a non-vectorized merge sort code developed by us, and the quick sort routine available in the CBE SDK. The codes were first debugged and optimized using the CBE simulator that is available with the SDK. The optimized codes were run on the Georgia Tech-STI Cellbuzz cluster to obtain actual run times. Figure 15 gives the average time required to sort n 4-byte integers for various values of n. The average for each n is taken over 5 randomly generated sequences. The variance in the sort time from one sequence to the next is rather small and so the reported average is not much affected by taking the average of a larger number of random input sequences. Figure 16 is a plot of the average times reported in Figure 15. The shown run times include the time required to fetch the data to be sorted from main memory and to store the sorted results back to main memory.

Number of Integers	AASort	Shaker sort	Brick Sort	Bitonic Sort	Merge Sort	Merge Sort (Sequential)	Quick Sort
128	52	53.6	53	47.8	50.8	146.6	145.6
256	62.4	65.4	63.4	65.6	57.6	178.6	206.8
512	81.8	86.4	81.4	72.6	70.4	272.2	332
1024	123.8	142.2	116.8	125.4	97	315.4	605.6
2048	222.8	262	190.2	165.8	142	543	1164
4096	438.6	494.8	332.6	297.8	268.4	989.8	2416.6
8192	912.4	1033.6	663.8	609.6	508	2011.2	4686.6
16384	1906.4	2228	1361	1331.2	1017	4103	9485.2

Figure 15: Average time (in microseconds) to sort 4-byte integers

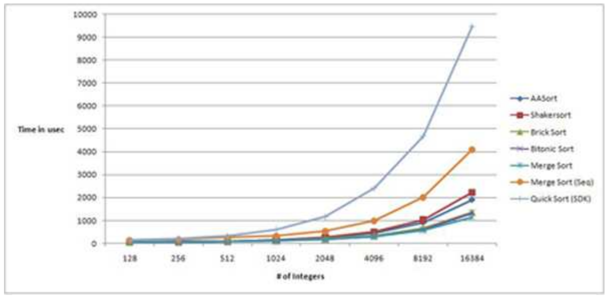

Figure 16: Plot of average time to sort 4-byte integers

Our experiments reveal that a standard non-vectorized textbook implementation of merge sort takes about 4 times the time taken by the vectorized merge sort adaptation developed in this paper. Further, the quick sort method that is part of the CBE SDK takes about 9 times the time taken by our merge sort adaptation. Brick sort is the fastest of the shell sort like algorithms–AA sort, shaker sort and brick sort–considered in this paper, taking about 71% the time taken by AA sort to sort 16384 integers. Although cell (bitonic) sort is slightly faster than brick sort, it takes about 31% more time to sort 16384 integers than taken by merge sort.

6 Conclusion

We have developed SPU sorting algorithms based on merge sort, shaker sort, and brick sort. Our merge sort adaptation is a stable sort whereas no other SPU sorting algorithm developed either by here in this paper or by others is a stable sort. Experiments show that our merge sort adaptation takes about 53% the time taken by AA sort [8], 76% the time taken by Cell sort [4], and 10% the time taken by the quick sort method that is standard in the SDK to sort 16384 4-byte integers. Further, merge sort is a stable sort while the remaining sorts are not. On the down side, merge sort requires $O(n)$ additional space to sort n numbers while the remaining methods require only $O(1)$ added space.

References

[1] Box, R. and Lacey, S., A fast, easy sort. *Byte*, 4, 1991, 315-318.

[2] Dobosiewicz, W., An efficient variation of bubble sort, *Information Processing Letters*, 11, 1980, 5-6.

[3] Drozdek, A., Worst case for Comb Sort, *Informatyka Teoretyczna i Stosowana*, 5, 9, 2005, 23-27.

[4] Gedik, B., Bordawekar, R., and Yu,P., CellSort: High performance sorting on the Cell processor, *VLDB*, 2007, 1286-1297.

[5] A.C. Chow, G.C. Fossum, and D.A. Brokenshire, A Programming Example: Large FFT on the Cell Broadband Engine.

[6] H. Hofstee, Power efficient processor architecture and the Cell Processor, *Proc. 11the International Symposium on High Performance Computer Architecture*, 2005.

[7] Horowitz, E., Sahni, S., and Mehta, D., Fundamentals of data structures in C++, Second Edition, Silicon Press, 2007.

[8] 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.

[9] Knuth, D., *The Art of Computer Programming: Sorting and Searching*, Volume 3, Second Edition, Addison Wesley, 1998.

[10] Lai, S. and Sahni, S., Anomalies in parallel branch-and-bound algorithms, *Comm. of the ACM*, 27, 6, 1984, 594-602.

[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.