Sorting Using the Xilinx Virtex-4 Field Programmable Gate Arrays on the Cray XD1

Stephen Bique, *Center for Computational Science (CCS), Naval Research Laboratory, Washington, D.C., USA,* **Wendell Anderson**, *CCS*, **Marco Lanzagorta**, *ITT Corporation, Alexandria, Virginia, USA,* **Robert Rosenberg**, *CCS*

ABSTRACT: *Sorting of lists is required by many scientific applications. To do so efficiently raises many questions. Is it faster to sort many lists concurrently using a slow sequential algorithm, or a small number of lists sequentially using a fast parallel algorithm? How large a list can be sorted using various data structures and what is the relative performance? The focus of this study is parallel algorithms, including parallel bubblesort, mergesort, counting sort, and novel combinations of these algorithms. Several implementations are described in detail using the Mitrion-C programming language. Appreciable speedup is observed and results are reported from running programs on a core of a Cray XD1 node with a Xilinx Virtex-4 LX FPGA as a coprocessor sorting data generated by a symmetric multiprocessor (SMP).*

KEYWORDS: XD1, FPGA, Mitrion-C, VHDL, HPC, SMP

1. Introduction

A Cray XD1 system is one of the high performance computing (HPC) resources at the Naval Research Laboratory [1]. The field programmable gate arrays (FPGAs) in a Cray XD1 system are intended as application acceleration processors (AAPs), which improve performance by performing operations in parallel and pipelining. Sorting is among the target applications for acceleration [3].

A list of n elements is sorted in optimal sequential time O($n \log n$). There exist O($\log n$) parallel algorithms for sorting (for example, see [6]). If an FPGA coprocessor must initially access every element one at a time to sort a list, then at least $\Omega(n)$ time is needed to sort a list of n elements using a single external memory bank; whence, the maximum speedup is limited to O($\log n$).

The speedup is diminished due to the overhead of transferring data (lists to be sorted) and loading a binary file with the configuration bit stream to the target FPGA device. This overhead may significantly impact performance unless it is possible to execute an application logic many times without reconfiguration. Sorting is carried out repeatedly for some HPC applications (see, for example [2], p. 2791). For these applications, the overhead of loading the application logic is reduced. To reduce overhead further, at least one of the two-way symmetric multiprocessor (SMPs) executes concurrently instead of waiting while an FPGA coprocessor is running.

Under investigation are various implementations in which an SMP "calls" upon a Xilinx Virtex-4 LX FPGA coprocessor to carry out a stable sort on a list of keys and to return the locations for the keys after sorting (instead of returning only a sorted list). For example, given the list <7,1,6,3,4,5,2,8>, the task is to find the corresponding list <1,6,3,4,5,2,0,7> that specifies the locations of the sorted keys (counting from zero). The keys are arbitrary 16-bit nonnegative integers k, $0 \le k \le 65534$. The number of sequences and the length of all sequences are fixed in advance. After execution on a coprocessor, an SMP uses the returned lists to perform further processing on the fields (of records). In a stable sort, two records maintain their relative order when the keys are the same.

To compare different implementations, the measure chosen is to compute the speedup using only the time spent sorting on an FPGA versus the time spent sorting on an SMP (AMD Opteron processor) by calling the C

Standard Library function *qsort*(), which is widely implemented. Define

$$speedup = \frac{time\ running\ qsort()\ on\ SMP}{time\ running\ FPGA}.$$

Although there is some additional overhead that is not taken into account, such a measure is nonetheless valid to compare the relative performance of different implementations. This measure underestimates the speedup because *qsort*() does not carry out a stable sort.

The measure is useful to compare various implementations that sort lists independent of the type of algorithms employed. The speedup is not independent of the input because the time to run *qsort*() depends upon the input, although the time spent running on an FPGA does not depend on the input for the same problem size.

To obtain more reliable results, the input for testing consists of as many pseudorandom permutations as fit into the FPGA's external memory (subject to program constraints). The following linear congruential generator compatible with the Cray MATLIB routine RANF is used to generate pseudorandom permutations on an SMP [5]:

$$x_{n+1} = 4485709377909 \cdot x_n \bmod 2^{48}.$$

Using this method, consistent results were observed.

A goal of this work is to explore different algorithmic solutions that are suitable for implementation on an FPGA. The plan is to exploit the architecture of an FPGA device in every implementation. By experimentation with different programming constructs and features, it is possible to discover which choices yield the best performance. Ultimately, the aim is to write implementations for which the speedup is appreciable.

Source	Destination	Read	Write
FPGA	FPGA QDR	Fast (3.2 GB/s)	Fast (3.2 GB/s)
FPGA	Host RAM	Unavailable	Slow (0.7 GB/s)
Host	FPGA QDR	Slow (10 KB/s)	Fast (3.2 GB/s)
Host	Host RAM	Fast (GB/s)	Fast (GB/s)

Table 1 Approximate Bandwidth of I/O Operations

To exploit the architecture, an FPGA device reads from and writes to its external memory in every implementation. The speed of I/O operations is vastly asymmetrical. Table 1 summarizes the I/O channels and their speeds [7]. In the case of reads and writes by an FPGA to its four external Quad Data Rate (QDR) II static RAM (SRAM) cores, actual bandwidth depends directly on the number of I/O operations performed in parallel.

To enhance software development, high-level languages (HLLs) are used to study various ways of sorting on an FPGA coprocessor. Although there is some degradation in performance and chip utilization compared to programming in a hardware description language (HDLs) such as Verilog and VHDL, or even another HLL, it is reasonable to expect that the relative performance of different implementations would depend more on the design techniques and algorithms employed and less on the programming environment. El-Araby et. al. offer a comparative analysis of DSPLogic, Impulse-C and Mitrion-C in a Cray XD1 environment [4]. The scope of their work (wavelet transforms and encryption algorithms) does not include sorting algorithms.

Mitrion-C is a novel HLL for the so-called Mitrion Virtual Processor (MVP) [8]. This soft core may be seen as a virtual machine for which special programming constructs are mapped to custom VHDL code for a particular FPGA chip. The MVP is a massively parallel computer that permits the type of fine-grain parallelism that characterizes FPGAs. This virtual machine implements speculative evaluation of **if** statements.

The MVP is part of the overhead that incorporates the Mitrion-C code and actually runs on an FPGA. A disadvantage of using Mitrion-C is that the MVP uses resources so that only a fraction of the FPGAs resources are available to the programmer. Another weakness is the fact that the MVP runs at a slower clock frequency than current FPGAs permit. As technology changes rapidly, it is not feasible to fully, or to continue to, support every chip unless there is strong market demand. Currently, the MVP does not fully implement all of the functionality supported by the Cray API for Virtex-4 FPGAs.

In subsequent sections, several different implementations that carry out a stable sort are described using Mitrion-C version 1.3, which is the latest version available for the Cray XD1. For each implementation, the observed speedup for various runs on a Cray XD1 node is reported. The next section briefly introduces the data types available in Mitrion-C [8][9][10].

2. Data Types

A compiler is able to automatically infer types and bit-widths. If the programmer has additional information that permits the bit-width to be reduced, then it is advantageous to specify the bit-width to reduce program complexity. In some cases it is necessary to specify the bit width, e.g., when shifting left.

Mitrion-C replaces arrays in C by three different so-called “collections:” lists, streams and vectors. Lists and streams are suitable for LISP programming (LISt Processing) whereas vectors suit parallel programming. The base type can be integer, floating point or Boolean.

Unlike arrays in C, collections are immutable objects that cannot be modified once defined. Loops in Mitrion-C provide the chief tool to form new collections from existing ones of the same type. In particular, only lists and vectors (not streams) can be redefined using this tool. However, redefining is possible only if the size is unchanged since the length of lists and vectors is fixed.

Although not considered a collection, a tuple is essentially a small sequence. Unlike collections, tuples may contain different types (and instance tokens). The base type in memory cannot be a collection or a tuple. Collections may contain tuples, provided each entry has the same sequence of types.

The **for** and **foreach** expressions are loops over a collection. To run blocks of code in parallel, a programmer uses vectors in a **foreach** loop to automatically unroll the code, i.e., to make copies of the loop body. To implement pipelining in the context of a loop, a programmer writes a **for** loop over vectors. In practice, only small vectors may be used due to the limited availability of resources and the duplicate configuration for every iteration.

A common technique is to reshape a list into a multidimensional list of vectors. It is then possible to iterate over the smaller vectors using a **foreach** inner loop. In this way, a programmer reduces the resource requirements and still achieves a degree of parallelism. This approach is relevant if the problem can be decomposed in a suitable way. Matrix multiplication is a good example that permits such decomposition.

A list or a vector defined outside a loop, can be referenced in the body of the loop, which is not possible with streams. Lists and vectors (not streams) can also be built-up or taken apart via the concatenation (><), take (</) and drop (/<) operators. Although not documented, the operands should be one-dimensional, which is not checked by the compiler.

3. Performing Sorts Concurrently

The speedup in sorting a single list of n elements is limited to O($log\ n$). A way to improve performance is to perform many sorts at the same time. The number of sorts that can be performed concurrently depends both on the amount of resources required to perform a single sort and the amount of resources available on the FPGA.

To take advantage of the architecture and minimize the time spent on I/O, a simple strategy is proposed. For each memory bank in parallel, iteratively read lists, sort them and write the resulting lists back at corresponding locations. For the Cray XD1 node with a Virtex-4 FPGA, this strategy leads to a fourfold improvement in performance (since there are four QRD IIs), provided sufficient resources are available. The external memories (QDR IIs) are dual ported, allowing simultaneous reads and writes.

Another way to improve performance is to overlap I/O activity with useful work. I/O takes O(n) time to read a list of n elements. Although this time cannot be reduced, ideally the coprocessor is busy performing other useful work so that the cost of I/O is small. Pipelining is one way to overlap computations and I/O operations. Another strategy is to perform many non-pipelined sorts concurrently for each external memory bank. Both cases may be implemented using function calls.

Consider many concurrent sorts using a slow algorithm. As sorting is a relatively slow process, many lists may be read and written while sorting takes place. The challenge is to write efficient Mitrion-C code that runs on the Cray XD1 and that reads or writes many lists using separate function calls without any memory access conflicts.

A solution is to perform I/O operations in a round-robin fashion and commence a sort as soon as a list is read. Hence, many sorts are scheduled in a “circular pipeline” using function calls. A **foreach** “block expression” is suitable to implement such a circular queue. The programming challenge is to make function calls back to back.

4. Parallel Bubblesort

A parallel version of bubblesort is commonly known as “odd-even sort.” The name parallel bubblesort is preferred because it emphasizes the implicit parallelism. For convenience, define for any finite list, say

$$L = \langle \boldsymbol{a_0}, \boldsymbol{a_1}, \ldots, \boldsymbol{a_p}, \boldsymbol{a_q}, \ldots, \boldsymbol{a_{n-1}} \rangle,$$

with $\|L\| = n$ elements, any pair of consecutive elements, say ($\boldsymbol{a_p}$, $\boldsymbol{a_q}$), is said to be even or odd if the position $\boldsymbol{p}$ of the first element of the pair is even or odd, respectively, e.g., ($\boldsymbol{a_0}$, $\boldsymbol{a_1}$) is an even pair while ($\boldsymbol{a_1}$, $\boldsymbol{a_2}$) is an odd pair. An algorithm is stated next.

Algorithm Parallel Bubblesort (**L**)
Input: **L** a list
Output: **J** indices for sorted list
Requirement: The length of **L** is even.

1. Sort all even pairs in parallel.
2. Sort all odd pairs in parallel.
3. Repeat steps 1-2 sequentially exactly $\frac{\|L\|}{2}$ times.

Example. Consider how parallel bubblesort works on the list 1,3,8,5,6,7,2,5. Label duplicates using subscripts so that $\boldsymbol{a_k}$ indicates the element $\boldsymbol{a}$ appears for the $\boldsymbol{k}^{\text{th}}$ time in the original sequence. Comparisons of odd pairs are indicated using arrows after every "even" pass (step 1) in the next illustration.

$$\begin{array}{cccccccc}
1 & 3 & 8 & 5_1 & 6 & 7 & 2 & 5_2 \\
1 & \vec{3} & \overleftarrow{5_1} & \vec{8} & \overleftarrow{6} & \vec{7} & \overleftarrow{2} & 5_2 \\
1 & 3 & 5_1 & 6 & 8 & 2 & 7 & 5_2 \\
1 & \vec{3} & \overleftarrow{5_1} & \vec{6} & \overleftarrow{2} & \vec{8} & \overleftarrow{5_2} & 7 \\
1 & 3 & 5_1 & 2 & 6 & 5_2 & 8 & 7 \\
1 & \vec{3} & \overleftarrow{2} & \vec{5_1} & \overleftarrow{5_2} & \vec{6} & \overleftarrow{7} & 8 \\
1 & 2 & 3 & 5_1 & 5_2 & 6 & 7 & 8
\end{array}$$

The algorithm requires $\frac{\|L\|}{2}$ passes in case a maximum or minimum of a finite set of distinct elements appears furthest from its location after sorting. Such an extreme value necessarily must be moved after every comparison except for the last odd pair, which yields

$$2\left(\frac{\|L\|}{2} - 1\right) + 1 = \|L\| - 1$$

moves. This sorting algorithm is stable provided equal elements are never swapped.

Which of the three types of collections is a suitable choice for implementation? Actually the only choice is a vector. Neither lists nor streams permit iteration over more than one element at a time. Using multiple internal memory banks, it is possible to perform multiple memory accesses in parallel. Since it is not possible to create a collection whose base type is an instance token (which are memory references), it follows that considerable resources would be needed to manage many internal memory banks.

How is the basic operation (sort pairs in parallel) efficiently implemented using vectors? Using indices is too costly. Lists may be partitioned into two vectors containing the even and odd elements. It is then possible to iterate over such pairs of vectors in a **foreach** loop to access all elements in parallel.

The stages are different as there is one less comparison to carry out during the "odd" stage because the number of odd pairs is one less than the number of even pairs. The elements that are not used need to be dropped and later added back. The concatenation, take and drop operators are useful to manage these operations. Figure 1 shows a suitable Mitrion-C code fragment.

```
ELEMENT_TYPE[4] EvenVector = reformat( e, [4] );
ELEMENT_TYPE[4] OddVector = reformat( o, [4] );


(x, w) =  for( EvenOddPasses in  <1..4> ) {

   /** Pass: even part **/
   (EvensEP, OddsEP) =  foreach( a,b in EvenVector, OddVector ) {
       (p,q) = if( a > b ) (b,a) else (a,b);
    } (p,q);

   /** Pass: odd part **/
    (OddsOP,EvensOP) = foreach( a,b in (EvensEP /< 1),(OddsEP </ 3)) {
        (p,q) = if ( b > a ) (a,b) else (b,a);
    } (p,q);


    /* Add first and last elements back **/
    (EvenVector, OddVector) = foreach( a,b in
              ( EvensEP </ 1 ) >< EvensOP, OddsOP >< (OddsEP /< 3 ) ) {
    } (a,b);

} (EvenVector, OddVector);
```

Figure 1 Parallel Bubblesort in Mitrion-C

This partitioning into evens and odds is easily done when a list is read from external memory. In particular, iterate over the list of indices in a **foreach** loop and then **reformat** the returned lists into vectors. Reading via iteration over a vector in a **for** loop surprisingly yields worst performance in the vast majority of test cases.

If the number of elements is odd, then use an **if** statement to assign a large value for the last odd element; otherwise, read elements from external memory. Such large values are not written at the end. Use the take operator (</) in a **foreach** loop to write all of the elements except the last even and odd elements. Then write the last even element.

Consider the number of operations performed on lists with eight elements using four memory banks. A total of 28 comparisons are performed each iteration. Sixteen comparisons are performed in parallel during the "even" pass because there are four lists (one per external memory bank) and four pairs for which the position of the first element is even. Twelve comparisons are performed in parallel during the "odd" pass because there are four lists and three pairs for which the position of the first element is odd.

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
3	699048	12.0
4	524288	13.2
5	419428	16.4
6	349524	17.1
7	299592	18.9
8	262144	19.8
9	233016	21.5
10	209712	21.2
11	190648	22.6
12	174760	22.8
13	161316	23.9
14	149796	24.2
15	139808	24.9
16	131072	24.6
17	123360	25.4
18	116508	26.2
20	104856	25.9
24	87380	27.3
26	80656	28.4
32	65536	29.1
34	61680	29.8
36	58252	30.2
38	55188	30.3
39	53772	30.3
40	52428	30.0

Table 2 Performance of Parallel Bubblesort

The observed speedup for various runs on the Cray XD1 is shown in Table 2. If there are fewer than four elements, then there is at most one even or odd pair, which is a degenerate case. There are not sufficient resources to sort lists with more than forty elements using the same implementation.

5. Mergesort

Mergesort is a classical sorting algorithm that is typically regarded as a recursive algorithm. Such "top-down" algorithms are suitable for sequential computers. Yet this algorithm may be viewed as a "bottom-up" algorithm with the number of merging phases being logarithmic in the length of the list.

In sequential programming, the merging procedure terminates by copying the shorter of the two lists that remain (one of which is the partially merged list) as soon as one of the lists is emptied. In this way, the sequential time complexity of merging is minimized. Such processing is not relevant for FPGA programming for which more regular program flow is needed.

A merge operation is linear in the length of the merged list, which is a consequence of the fact that an element is removed from one of the two lists after each comparison (there are only so many elements to remove). Hence, a **for** loop is suitable to implement each merge operation.

There are different programming techniques to handle the case when one of the lists becomes empty. A sentinel can be used to handle the special cases. Storing the sentinel at the end of the list in memory using a brute force approach is costly. There are strategies to reduce such costs. Notwithstanding, tests using sentinels did not yield better performance.

Another method involves counting the elements remaining in each sublist. When one of the lists becomes empty, copy the elements from the other list to the end of the merged list. Writing such code is slightly more complicated. Good performance has been observed using this approach.

A common trick is to scan pairs of lists in opposite directions and iterate until the indices cross, instead of scanning in the same direction. Using this method, the special case when one of the lists becomes empty is avoided. To do so, reverse the right list and merge with the left list scanning from the "outside."

Example. Suppose the left list is 1,2 and the right list is 3,4. Reverse the right list and concatenate to obtain the new list 1,2,4,3. Set the left index **i** = 0 and the right index **j** = 3.

i	j	Merged List
0	3	1
1	3	12
2	3	123
2	2	1234

The alternating method, however, requires special handling to implement a stable algorithm. The reason is that when a list on the left is emptied and the right list contains duplicates, they are not in the original order. Additional steps are required to force the algorithm to be stable. The following invariant is proposed:

> The left list is sorted in nondecreasing order with duplicates in original order and the right list is sorted in nonincreasing order with duplicates in reverse of their original order.

A stable merging algorithm based on this invariant is stated next.

Algorithm Alternating Stable Mergesort (**L**)
Input: **L** a list
Output: **J** indices for sorted list
Requirement: $\|L\| = 2^k \cdot m$, for some *k* and *m*.

1. Merge lists of length *m* into lists of length 2*m* in alternating non-decreasing and nonincreasing orders. When comparing elements from a left and right list, choose the smaller element and in the case of equality, choose the element from the left list. If the right index has crossed into the left list, choose the leftmost element, which is the only choice by the invariant. If the left index has crossed into the right list, then choose the rightmost element by the invariant.
2. Double *m* and repeat step 2 as long as $2m \le \|L\|$.

Example. Below is an illustration how mergesort operates on the list 1,3,5,8,6,7,2,5. Subscript $\boldsymbol{k}$ in $\boldsymbol{a_k}$ indicates the element $\boldsymbol{a}$ appears for the $\boldsymbol{k}^{\text{th}}$ time in the original sequence.

$1 \quad 3 \qquad 5_1 \quad 8 \qquad\qquad 6 \quad 7 \qquad 2 \quad 5_2$

$1,3 \qquad 8,5_1 \qquad\qquad 6,7 \qquad 5_2,2$

$1,3,5_1,8 \qquad\qquad 7,6,5_2,2$

$1,2,3,5_1,5_2,6,7,8$

What data structure should be used to implement the mergesort algorithm? Collections are useful to iterate over elements. The mergesort algorithm does not prescribe such iteration as an element may or may not be removed during any iteration. A solution is to use the equivalent of an array in local memory so that elements can be accessed repeatedly.

Viewing mergesort as a bottom-up tree, each merge operation at the same level can be performed sequentially. In this way, successive merges at different levels take the same amount of time. A pipelined solution is then easy to implement.

In order to pipeline the stages of the mergesort algorithm, a separate internal memory bank is created for each stage. If these banks were only large enough to hold a single list, then there can be unacceptable delays in the pipeline. In order to allow a stage to operate continuously, larger banks are created so that different stages can use the same bank concurrently. To do so, the internal memory banks are treated as circular queues.

A pipelined implementation of mergesort using function calls in a **foreach** loop yields good performance for intermediate size lists. Table 3 shows results from runs on the Cray XD1 using counters to carry out the merge operations. The observed speedup decreases as the lengths of the lists become smaller.

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
64	32768	16.2
128	16384	18.5
256	8192	20.0

Table 3 Mergesort Performance

A bottom-up approach offers a high degree of implicit parallelism. Each successive merge (row) can be executed in a pipeline. All pairs (in a row) can also be merged concurrently. However, the former type of parallelism is more efficient. Pipelining without concurrently merging pairs implies each merge operation requires approximately the same amount of work and time, which allows most of the configured hardware to remain operational most of the time.

If each successive stage merged pairs concurrently, then later stages would take longer time; whence, a bottleneck would be created. Besides merging all pairs concurrently would quickly exhaust all resources. In particular, extra internal memory banks would be needed to perform all of the required concurrent reads and writes. Regardless, read and write operations must be carried out sequentially for both the original list and the results. Hence there is little to be gained by concurrent merging since the pipeline is only as fast as the slowest link in the chain.

Is it practical to merge four lists instead of just two? Consider merging four sorted lists using an efficient **for** loop so that an element is removed during every iteration. To do so requires finding the minimum of four elements.

Finding the minimum can be done using only three comparisons as demonstrated in Figure 2.

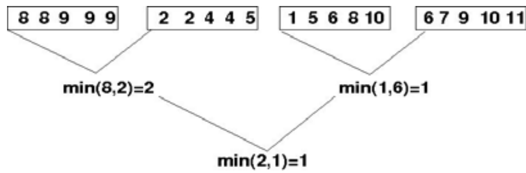

Figure 2 Minimum of Four Sorted Lists

Actually, it is unnecessary to perform all three comparisons every iteration. The reason is after discovering a minimum among the four lists, only one item is removed from one of the lists; whence the minimum of one pair of lists does not need to be recomputed. Therefore only two comparisons are needed per iteration: one to find the minimum from a pair of lists, and one to find the minimum of two pre-computed minimums.

How does the complexity of 4-way merge compare to 2-way merge? Suppose there exist four sorted lists, each having, say m, elements. Using 2-way merge, at most 2m-1 comparisons are needed to merge each pair of lists since after each comparison one element is removed (from 2m elements). Up to 4m-1 comparisons are needed to combine the merged pairs, which yields a total of

$$2(2m-1)+4m-1=8m-3$$

comparisons. Using 4-way merge, two initial comparisons plus two additional comparisons per iteration for every removal, which yields

$$2+2(4m)=8m+2$$

comparisons.

Hence, 2-way and 4-way merge carry out nearly the same number of basic operations (comparisons).

Figure **3** shows C code that implements 4-way merging. This code demonstrates that it is possible to avoid many array accesses, i.e., fewer reads are required, another advantage of 4-way merging.

While reducing the number of I/O operations is in general advantageous, some I/O operations can be performed in parallel, which negates the advantage. More importantly, 4-way merge is disadvantageous in a pipeline since merges take twice as long. Notwithstanding, an optimized implementation that sorts lists of exactly four elements using 4-way merge yields speedup of 13.1 on a Virtex-4 FPGA compared to calling the C standard library function *qsort*() on an AMD processor.

6. Counting Sort

Counting sort finds the position of an element by counting the number of smaller elements. If all elements are distinct, then the position is exactly the number of elements smaller than e. For example, the minimum and maximum appear in the first and last positions. In the case of duplicates, count also those identical elements that appear earlier in the original list so that the algorithm is stable.

The sequential time complexity of counting sort is comparable to selection sort. Selection sort works like bubblesort except the former does a single swap after each pass in which another element "bubbles" its way to its final position. Counting sort requires additional space to calculate positions. For this reason, counting sort is not used in practice. However, a Virtex-4 FPGA coprocessor offers greater resources and potential for speedup.

```
void Merge_4(int * a, int * b, int * c, int * d, int * result, int M) {
  int i_L = 0, j_L = 0, i_R = 0, j_R = 0, k,
  min_L, min_R, a_ = a[i_L], b_ = b[j_L], c_ = c[i_R], d_ = d[j_R];

  // Do the first two of three comparisons
  if( a_ > b_ ) {
    min_L = b_; j_L++; b_ = b[j_L];
  } else {
    min_L = a_; i_L++; a_ = a[i_L];
  }

  if( c_ > d_ ) {
    min_R = d_; j_R++; d_ = d[j_R];
  } else {
    min_R = c_; i_R++; c_ = c[i_R];
  }
  // Do the remaining two comparisons:
 for( k=0; k<M; k++  ) {

    if(min_L > min_R)
      {
        result[k] = min_R;
        if( c_ > d_ ) {
          min_R = d_; j_R++; d_ = d[j_R];
        } else {
          min_R = c_; i_R++; c_ = c[i_R];
        }
      }
    else
      {
        result[k] = min_L;
        if( a_ > b_ ) {
            min_L = b_; j_L++; b_ = b[j_L];
        } else {
            min_L = a_; i_L++; a_ = a[i_L];
        }
      }
  }
}
```

Figure 3 C Code for 4-Way Merge

Algorithm Counting Sort (**L**)
Input: **L** a list
Output: **J** indices for sorted list

1. In parallel for each position in the list **L**, sequentially count the total number of elements less than the element appearing at this position plus the number of identical elements appearing before this position in **L**.
2. Return the sums.

Example. Consider the list 1,3,8,5,6,7,2,5. Calculate the position of any element by scanning the list and counting all smaller elements as well as any duplicates that appear earlier in the list. The computed positions of corresponding elements are 0,2,7,3,5,6,1, and 4.

```
CountingSort( Mem1, Offset )
{
  list = foreach( Index in <0.. L_1 > ) {
     SHORT_TYPE data=_memread(Mem1,Offset+Index);
  } data;

  v = reformat( list, [ L ] );

  Last_m = foreach( e in v by INDEX_TYPE Pos_e ) {
    INDEX_TYPE sum = 0;
    Sum = for( Pos_a, a in <0.. L_1>, list ) {
      sum = if( a > e ) {
            } sum else {
               le=if((a==e) && (Pos_a >= Pos_e)) {
                    } sum else {
                         s =sum + 1;
                    } s;
            } le;
      } sum;
    m  = _memwrite( Mem1, Offset + Sum, Pos_e );
  } m;
} Last_m;
```

Figure 4 Counting Sort Function

The parallel part of this algorithm is implemented by iteration in a **foreach** loop over a vector. The sequential part is implemented by iteration over a list in a **for** loop. There is little advantage in using a vector to unroll this **for** loop, which wastes resources. Figure 4 displays a counting sort function in Mitrion-C, which is succinct. An implementation iteratively sorts lists with up to 50 elements using a function call in a **foreach** loop. The main() function makes four calls, one per memory bank, to the function that carries out the iteration.

Table 4 shows results of runs on the Cray XD1. The relatively low speedup in a couple cases (length = 13,14) indicates a weakness of the MVP in the current version 1.3.

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
3	699048	12.0
4	524288	13.2
5	419428	16.3
6	349524	17.0
7	299592	18.8
8	262144	19.9
9	233016	21.5
10	209712	21.3
11	190648	22.6
12	174760	22.9
13	161316	12.4
14	149796	12.6
16	131072	24.8
20	104856	26.0
24	87380	27.2
32	65536	29.1
36	58252	30.3
40	52428	30.0
46	45588	30.7
47	44620	31.4
48	43688	31.3
49	42796	31.7
50	41940	32.0

Table 4 Counting Sort

Instead of computing the sums using a single **for** loop, consider breaking the list into two nearly equal halves and writing two **for** statements, which are executed in parallel, to compute the sum. There are sufficient resources to sort lists with up to 40 elements. This modification yields an appreciable improvement in performance in only those couple instances for which the previously observed speedup was surprisingly low as shown in Table 5. Performing additions faster is not effective. The time required to read *n* elements is about the same as the time needed to perform *n* additions sequentially.

Because the lists are partitioned into two parts, it is natural to write a loop in which two reads are executed per iteration. Hence, the number of iterations is balanced for performing reads and computing sums. Yet, Table 5 shows that this strategy does not yield performance improvement in many cases. If the length is odd, then there are not sufficient resources to perform two read operations per iteration plus an extra read for lists with more than twenty elements (using the same techniques). In such cases, it is efficient to perform a single read per iteration and then partition using the take and drop operators.

For small lists, there are enough resources to use a vector instead of a list and compute the terms in a **foreach** expression (instead of computing the sum in a **for** loop). It is then possible to compute the sum of the terms in parallel (inside the outer **foreach** expression) using a

binary tree as a model. An example of Mitrion-C code is given in Figure 5 for lists with eleven elements. There are sufficient resources to sort lists with up to 14 elements.

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
3	699048	12.1
4	524288	13.1
5	419428	16.4
6	349524	17.0
7	299592	18.9
8	262144	19.8
10	209712	21.2
11	190648	22.6
12	174760	22.8
13	161316	23.9
14	149796	24.3
16	131072	24.7
19	110376	26.0
21	99864	26.8
24	87380	27.2
32	65536	29.1
36	58252	30.2
39	53772	30.3
40	52428	30.1

Table 5 Counting Sort - Two Summations

```
CountingSort( Mem1, Offset )
{
  v = foreach( Index in [0.. L_1 ] ) {
    SHORT_TYPE data = _memread( Mem1, Offset + Index );
  } data;

  Last_m = foreach( e in v by INDEX_TYPE Pos_e ) {
    Sum = foreach( a in v by Pos_a ) {
      sum = if( a > e ) {z=0;} z  else {le =
        if ( (a==e) && (Pos_a >= Pos_e)
           ) {z=0;} z  else {k=1;} k;} le;
    } sum;

   Sum2 = foreach ( a,b in (Sum </ 5), (Sum /< 6 ) ) {
      ab = a + b;
   } ab;
   Sum4 = foreach ( c,d in (Sum2 </ 2), (Sum2 /< 3 ) ) {
       cd = c + d;
   } cd;
   INDEX_TYPE u = Sum[5]+Sum2[2]+Sum4[0]+Sum4[1];

   m = _memwrite( Mem1, Offset+u, Pos_e );
 } m;
} Last_m;
```

Figure 5 Counting Sort – Sums in Parallel

Testing yielded appreciable speedup only in the exceptional cases noted earlier. Table 6 shows results from various runs on the Cray XD1. There is little to be gained in speedup as the lists become small because there are few operators that can be applied in parallel and there are I/O limitations as well.

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
3	699048	12.1
4	524288	13.2
5	419428	16.4
6	349524	16.9
7	299592	19.0
8	262144	19.8
9	233016	21.4
10	209712	21.2
11	190648	22.6
12	174760	22.9
13	161316	23.8
14	149796	24.2

Table 6 Counting Sort – Sums in Parallel

In this case, a binary tree is a model that bears little fruit. Instead of trying to perform additions in parallel, consider pipelining the summation by unrolling the **for** loop. This optimization can only be carried out for small lists. Table 7 displays results from runs on the Cray XD1. Again the speedup is not appreciable except in a couple instances.

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
3	699048	12.0
4	524288	13.2
5	419428	16.4
6	349524	17.0
7	299592	18.8
8	262144	19.8
9	233016	21.5
10	209712	21.3
11	190648	22.6
12	174760	23.0
13	161316	23.9
14	149796	24.2

Table 7 Counting Sort – Unrolled Loop

7. Sorting Long Lists

Is it possible to achieve significant speedup for long lists? A common technique to manage larger problems is to decompose them into smaller ones. The programming challenge is to find the right match between the different stages to avoid bottlenecks.

Counting sort can be applied to large lists. Instead of computing positions for every element in parallel, a programmer arranges to compute them in parallel only for

all elements in a "segment." Thus the list is divided into contiguous segments.

```
(list,positions) =
  foreach(INDEX_TYPE i in <0.. L_1>){
    SHORT_TYPE x =_memread( Mem1, Offset + i );
} (x,i);

list_k = reshape( list, < K >< M > );
positions_k = reshape( positions, < K >< M > );

Last_m =
  foreach(l_k,p_k in list_k,positions_k) {
    v = reformat( l_k , [ M ] );
    p = reformat( p_k, [ M ] );

    lm = foreach( e,Pos_e  in v, p ) {
      INDEX_TYPE sum = 0;
      Sum = for( Pos_a, a in <0.. L_1>, list ) {
        sum = if( a > e ) {} sum
          else {
           le=if((a==e)&&(Pos_a >= Pos_e)){}sum
              else {s = sum + 1;} s;} le;
      } sum;
      m  = _memwrite( Mem1, Offset + Sum, Pos_e )
   } m;
 } lm;

} Last_m;
```

Figure 6 Counting Sort for Long Lists

Figure 6 shows a fragment of Mitrion-C code that performs a sort after partitioning a list into K segments of length M using **reshape**(). Vectors are then iteratively created from the segments using **reformat**(). An implementation partitions lists into segments with M=32 elements. Table 8 shows results of runs on the Cray XD1. There are sufficient resources for intermediate size lists to divide the lists into segments and calculate the sums for each segment. Table 8 shows results using a single summation, except for lists with 64 elements for which four sums are computed in parallel. Since only M sums (instead of M·K sums) are computed in parallel, performance is relatively low in all cases.

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
64	32768	8.3
128	16384	9.5
512	4096	2.8
768	2728	2.0
1024	2048	1.5

Table 8 Counting Sort for Long Lists

Parallel bubblesort and mergesort do not belong to the same sequential time complexity class. Parallel bubblesort takes $\Theta(\|L\|)$ time to sort a list L since there are $\frac{1}{2}\|L\|$ alternating stages that take a constant amount of time. Although mergesort performs a total of $\Theta(\|L\|\log\|L\|)$ operations, a single stage of the algorithm takes only $\Theta(\|L\|)$ sequential time. Thus, parallel bubblesort and any *stage* of a pipelined mergesort algorithm do belong to the same complexity class.

These algorithms can be combined via pipelining. A programmer partitions each list into segments of equal length, and creates a pipeline via function calls. The first stage iteratively applies the parallel bubblesort algorithm to sort each segment. Each subsequent stage is an application of the mergesort algorithm to merge all of the segments.

By experimentation, a programmer finds a suitable length for each segment to achieve good performance. An implementation uses four and eight elements in the segments for lists with 64 and 128 elements, respectively, and uses the alternating mergesort algorithm. Table 9 shows results of runs on the Cray XD1.

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
64	32768	9.9
128	16384	12.5

Table 9 Parallel Bubblesort and Mergesort

The counting sort and mergesort algorithms yield the best performance on small and long lists, respectively. An implementation combines the counting sort algorithm and mergesort algorithms. The first stage sorts sublists with 16 and 32 elements to sort lists with 64 and 128 elements, respectively. Table 10 shows results of runs on the Cray XD1.

Length of List	Number of Lists	Approximate Speedup vs. *qsort*()
64	32768	11.0
128	16384	12.6

Table 10 Counting Sort and Mergesort

8. Conclusion

Substantial speedup has been observed compared to calling the Standard C Library function *qsort*() on an AMD processor. Table 11 displays a summary of observed speedups for different sizes of lists. These speedups were achieved by exploiting the architecture of a Xilinx Virtex-4 FPGA, and by taking full advantage of

Mitrion-C. The task performed on an FPGA coprocessor is to perform many stable sorts and to return the locations of sorted keys for further processing by an SMP. The case studies address difficult issues such as I/O.

Several implementations have been described in detail. Various programming techniques and algorithms suitable for implementation on an FPGA, including novel combinations of algorithms, have been studied to investigate which programming choices lead to the best performance. This work shows that counting sort, parallel bubblesort and mergesort are especially well suited to FPGA implementation. 2-way selection sort performs well for lists with four elements.

Length of List	**Algorithm**	**Approximate Speedup vs. *qsort*()**
3	Counting Sort	12.1
4	2-Way Selection Sort	13.8
5	Counting Sort	16.4
6	Parallel Bubble Sort	17.1
7	Counting Sort	18.9
8	Counting Sort	19.9
9	Counting Sort	21.5
10	Counting Sort	21.3
11	Counting Sort	22.6
12	Counting Sort	22.9
13	Pipelined Counting Sort	24.0
14	Pipelined Counting Sort	24.4
16	Counting Sort	24.8
20	Counting Sort	26.0
21	Counting Sort	26.8
24	Pipelined Counting Sort	27.3
32	Counting Sort	29.1
36	Counting Sort	30.3
39	Counting Sort	30.3
40	Counting Sort	30.1
46	Counting Sort	30.7
47	Counting Sort	31.4
48	Counting Sort	31.3
49	Counting Sort	31.7
50	Counting Sort	32.0
64	Mergesort	16.2
128	Mergesort	18.5
256	Mergesort	20.0

Table 11 Summary of Speedups Observed

Future work will focus on incorporating described implementations in real-world applications and investigating other types of problems that can be solved on FPGAs to speedup applications. In particular, more significant performance gains might be achieved by having an FPGA perform computations on the data before sorting. As FPGA technology advances and especially as new versions of the tools are released, it will become possible to accelerate applications in an increasing number of ways.

Acknowledgments

Dr. Jeanie Osburn, the head of the Operational Computer Section of CCS, provided valuable input and user support. Ray Yee, the onsite XD1 Cray engineer, helped to resolve programming issues. Jace Mogill of Mitrionics provided customer support. This work was performed entirely on the Cray XD1 system at NRL-DC under the auspices of the U. S. Department of Defense (DoD) High Performance Computer Modernization Program (HPCMP).

About the Authors

Stephen Bique is Computer Scientist in the Center for Computational Science (CCS) at the Naval Research Laboratory, Marco Lanzagorta is Senior Principal Scientist for ITT Corporation, Robert Rosenberg is Information Technology Specialist in CCS, and Wendell Anderson is Mathematician and the head of the Research Computers Section of CCS.

References

[1] **W. Anderson, M. Lanzagorta, R. Rosenberg, and J. Osburn,** *Early Experiences with XD1 at the Naval Research Laboratory*, CUG 2006 Proceedings, 2006.

[2] **C. M. Bachmann, T. L. Ainsworth, and R. A. Fusina**, *Improved Manifold Coordinate Representations of Large-Scale Hyperspectral Scenes*, IEEE transactions on geoscience and remote sensing, vol. 44, no. 10, pp. 2786-2803, October 2006.

[3] **Cray, Inc,** Cray XD1TM FPGA Development, S-6400-14, May 2006.

[4] **E. El-Araby, M. Taher, M. Abouellail, T. El-Ghazawi, and G.B. Newby**, *Comparative Analysis of High Level Programming for Reconfigurable Computers: Methodology and Empirical Study*, in 3rd Southern Conference on Programmable Logic, Mar del Plata, Argentina, ISBN 1-4244-0606-4, pp. 99-106, February 2007.

[5] **GSL Team**, *Other random number generators* in GNU Scientific Library: Reference Manual [http://www.gnu.org/software/gsl/manual/html_node/index.html], 2007.

[6] **J. JáJá,** An introduction to parallel algorithms, Addison-Wesley Publishing Co., Reading, Mass., ISBN 0-201-54856-9, 566 p., 1992.

[7] **M. Lanzagorta**, **and S.Bique**, *An Introduction to Reconfigurable Supercomputing*, to appear in a tutorial presentation at the Department of Defense High Performance Computing Modernization Program Users Group Conference, Seattle, Washington, July 2008.

[8] **Mitrionics**, Running the Mitrion Virtual Processor on XD1, 1.2-003, 2007.

[9] **Mitrionics**, The Mitrion Software Development Kit, 1.3.0-002, 2007.
[10] **S. Möhl**, The Mitrion-C Programming Language, Mitrionics, 1.3.0-001, 2007.