Sorting on Architecturally Diverse Computer Systems

Roger D. Chamberlain
Narayan Ganesan

Roger D. Chamberlain and Narayan Ganesan, “Sorting on Architecturally Diverse Computer Systems,” in *Proc. of Third Int’l Workshop on High-Performance Reconfigurable Computing Technology and Applications,* November 2009 (associated with Supercomputing’09).

Dept. of Computer Science and Engineering
Washington University in St. Louis

Sorting on Architecturally Diverse Computer Systems

Roger D. Chamberlain
Dept. of Computer Science and Engineering
Washington University in St. Louis
roger@wustl.edu

Narayan Ganesan
Dept. of Computer Science and Engineering
Washington University in St. Louis
nganesan@wustl.edu

ABSTRACT

Sorting is an important problem that forms an essential component of many high-performance applications. Here, we explore the design space of sorting algorithms in reconfigurable hardware, looking to maximize the benefit associated with high-bandwidth, multiple-port access to memory. Rather than focus on an individual implementation, we investigate a family of approaches that exploit characteristics fairly unique to reconfigurable hardware.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; E.5 [**Files**]: Sorting/searching

General Terms

Algorithms, Design, Performance

Keywords

sorting, architecturally diverse systems, reconfigurable hardware, field-programmable gate arrays

1. INTRODUCTION

Reconfigurable hardware, primarily in the form of field-programmable gate arrays (FPGAs), has proven to be a beneficial technology for a wide range of uses. Historically relegated to the task of providing simple "glue logic" in custom hardware designs, FPGAs have recently enjoyed tremendous growth in capacity to the point that entire applications can effectively deployed on them. Manufacturers are also tailoring FPGAs to the needs of different application areas, including optional embedded processor cores, multiply-accumulate units, specialized I/O functionality, etc. Carefully architecting an FPGA design can lead to a large speedup over a general-purpose processor in many non-trivial applications [2, 3, 9, 18, 26, 27]. El-Ghazawi et al. [10] recently published a comprehensive discussion of the applicability of FPGAs to high-performance computing.

Our group has developed the Auto-Pipe tool set [8, 13], an application development environment that supports the design, implementation, performance analysis, and execution of streaming data applications on architecturally diverse systems (those containing both general-purpose processors and FPGAs). One of the goals of the Auto-Pipe tool set is to ease the task of deploying applications onto diverse systems. To that end, Auto-Pipe instantiates all of the data communications between compute blocks in a streaming data application, removing that responsibility from the application developer.

Here, we focus on sorting, an important component of many applications, and explore the use of reconfigurable hardware to accelerate the task of sorting a stream of records. Rather than focus on an individual instantiation of a sorting problem, we will focus on how to best exploit one feature of reconfigurable hardware platforms that is fairly unique to these platforms, the availability of multiple, independent memory ports, both on-chip and off-chip.

Three fairly distinct approaches to implementing the sorting function are investigated, ranging from a memory array sort, to a systolic array sort, to a merge sort. Each is quantitatively assessed in terms of resource usage as well as performance.

2. RELATED WORK

Various optimal sorting algorithms on systems with a traditional memory hierarchy have been studied in detail. An optimal sort algorithm on n elements performs $O(n \log n)$ comparisons [19]. Various measures of presortedness [11, 22] could be defined on a sequence of records. Common measures such as *Inv*, the number of inversions in the input, i.e., the total number of pairs of elements that are switched in order; *Runs*, the number of boundaries between ascending subsequences; *Max* the largest difference between the ranks of an element between the input and the sorted sequence; *Dis*, the largest distance determined by an inversion, etc, give a good indication as to the number of comparisons required to fully sort the sequence. A sorting algorithm is called *adaptive* if its time complexity depends both on the number of elements as well as on the measure of presortedness. In [17], the performance of an adaptive sorting algorithm with respect to the measure *Inv* is presented. Such models do not consider the effects of cache and memory hierarchy, which affect the performance of the algorithms.

Since many systems contain multiple levels of memory, it is imperative that the sorting algorithm also be optimized for data movement as well as comparison count. A successful model to describe the behavior of sorting algorithm on a two-level memory heirarchy with a finite-size, fast memory and an infinite-size, slow memory is described in [1]. The time complexity was shown to be dependent on the memory characteristics as well as on an optimal access pattern. Since the performance of the optimal algorithm [1] depends on the fast memory size, it is denoted *cache-aware*. Many modern microprocessor systems operate on multiple levels of caches (L1, L2, L3) with different access times. The cache-aware model becomes increasingly complex with each additional level of memory. A model which works without the knowledge about the cache size or its properties was proposed in [14]. The idea is that this *cache-oblivious* algorithm, if successful with two-level memory systems, would also be successful with multi-level memory systems. Two optimal algorithms, the *Funnelsort* and a variant of *Distributionsort* are presented [14] for the cache-oblivious model. In [5], Brodal et al. proposed a reduction from adaptive-sorting to non-adaptive, cache-oblivious sorting which also yields a comparision as well as I/O optimal cache-aware and cache-oblivious sorting algorithms with respect to the measure *Inv*. Furthermore, cache-aware and cache-oblivious versions of *Genericsort* [12], are also proposed in [5].

Work on sorting in FPGAs has included a number of studies. Marcelino et al. [23] report on FPGA implementations of several traditional sorting algorithms, and Martinez et al. [24] describe an FPGA-based suffix sorter for use with BWT compression. Bednara et al. [4] describe a hybrid hardware/sofware sorter that uses an insertion sort on the FPGA and a merge sort on the general-purpose processor. The techniques we investigate come closest to this design, although the transition from the initial sorting technique to the merge sort that follows isn't constrained to happen at the hardware/software boundary.

3. SORTING

A common approach to sorting is to first sort groups of records that are subsequently merged in a later step [28]. This approach exploits record locality in both the group sort stage(s) and the merge stage(s), and data locality frequently provides significant performance benefits, both on traditional general-purpose processors and non-traditional specialized processors such as FPGAs and GPUs [16]. Here, we describe this approach in terms of a streaming data sorting application, starting first with the general application topology and following with descriptions of the internals of the constituent blocks.

3.1 Application topology

Viewing the sorting application as a streaming application (e.g., see [7]), we start by dividing the records to be sorted into small groups that are sorted within the group. In cache-aware sorting algorithms for general-purpose processors [5], the group size is typically determined by available cache size. The point is to exploit locality and use the fastest available memory. This is illustrated in Figure 1 with the `split` block and the `sort groups` blocks. The `split` block is responsible for receiving the stream of records from the left, dividing the stream into appropriate sized groups of records, and delivering those groups to the various `sort groups` blocks. More than one group of records is delivered to an individual `sort groups` block, the groups themselves also represent a stream.

The output of the `sort groups` blocks is routed to a pipeline of `merge groups` blocks. Figure 1 shows two `merge groups` blocks in each pipeline, but the number is arbitrary. The `merge groups` blocks take a stream of sorted groups, and perform a merge sort on M input groups, sending out a stream of groups that are M times as large as the input groups. For example, if the `merge groups` blocks are configured to merge two input groups, the output groups will contain twice as many records as the input groups. Clearly, each `merge groups` block must have sufficient buffer capacity to hold all M groups of input records, since the last record into the block might need to be the first record out. We will return to this memory requirement shortly.

At the end of the individual pipelines, another merge sort is performed in the `merge` block. This block performs the same merge sort operation as before; however, it receives as input N sorted groups of records on N distinct input ports (one per pipeline), resulting in a single output stream.

In the illustration, these merged groups of records are delivered to the general-purpose processor, which has its own implementation of a `merge groups` block, performing a merge sort on the groups of records it receives.

With the success of cache-aware sorting techniques applied to general-purpose processors, we will explore a similar approach to provisioning and tuning of the overall sorting application deployed on reconfigurable hardware. Rather than focus on the computational aspects of the application, we will make the assumption that the comparison operations are relatively free (or at least not the dominate factor for making design decisions) and will focus our attention on memory requirements.

We envision the basic application topology as illustrated in Figure 1, with the number of pipelines (N), the number of records in each group into the `sort groups` blocks, the number of `merge groups` blocks in each pipeline, and the number of groups that each `merge groups` block merges (M) all determined by the size, bandwidth, and number of ports available to the memory subsystem. We assume that the `sort groups` blocks are restricted to using on-chip memory (either distributed memory or block RAMs) and the `merge groups` blocks might use on-chip memory or off-chip SRAM or DRAM. In short, the properties of the available memory will determine the configuration of the sorting application. How this works will be described in Section 4, but first we must consider the details of the individual blocks.

There are many approaches to designing the blocks that perform the `sort groups` function. In the two sections below, we describe a pair of candidates. To keep the discussion straightforward, we will constrain the individual sort block descriptions to sorting 64-bit records that each contain a 32-bit key and a 32-bit tag. Clearly, if the records to be sorted in a particular application deviate from this assumption, the block implementations would need to be adapted to accommodate the alternate record format.

3.2 Memory array sort

The first approach to the `sort groups` block is based upon the ability in an FPGA to both build a multi-port memory and also instantiate a number of comparators or processing elements. The sorting algorithm itself is based upon the comb sort [20], which starts by comparing (and conditionally swapping) records that are initially distant from one another in memory, and progressively comparing closer and closer records until the gap between compared records

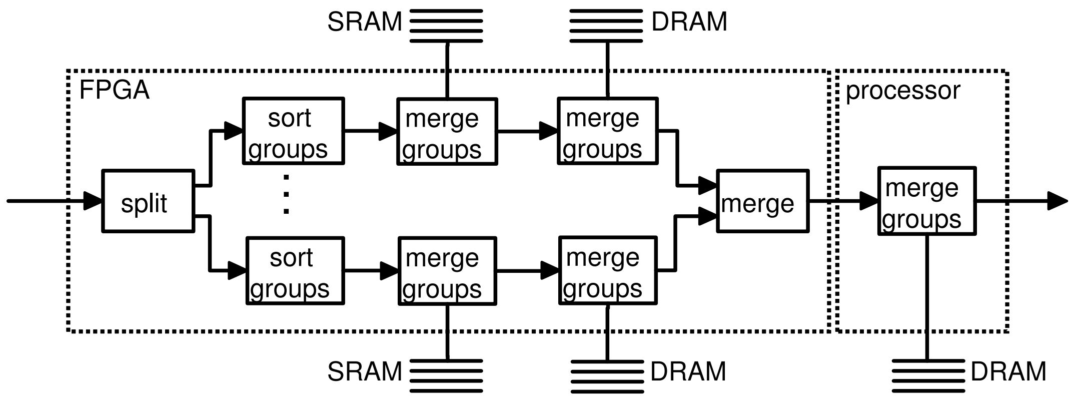

Figure 1: Sorting application topology. N parallel pipelines sort progressively larger groups of records. Groups of records are merged, M at a time, in the individual pipelines prior to being merged for delivery from the FPGA to the processor.

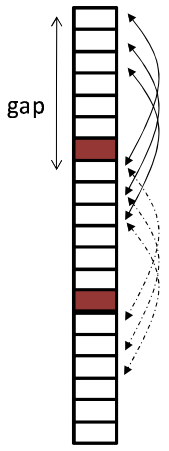

Figure 2: Utilization of 3 parallel comparators to compare and swap records gap $= 7$ apart.

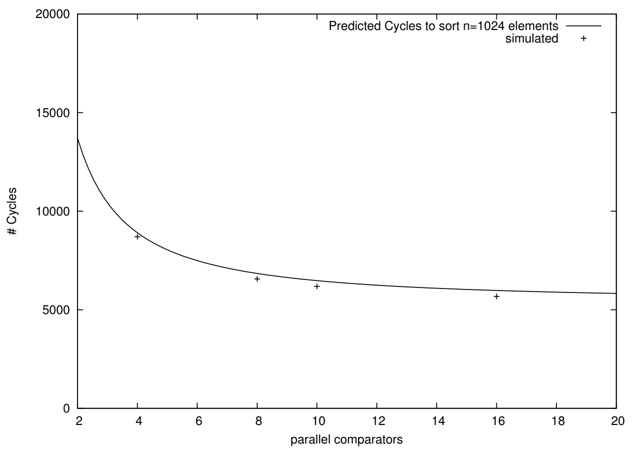

Figure 3: Modeled and simulated time to sort n=1024 records vs. the total number of parallel comparators, p.

is just 1. With a multi-port memory and p comparators, p comparisons can concurrently execute, and this capability will be exploited to accelerate the execution time of the sort.

In the classic description of the comb sort algorithm, two records that are gap records apart are compared and conditionally swapped based upon their key values. After a full sweep of the array, the value of gap, is set to gap/1.3. With sufficient distinct pairs of records that are gap records apart within a single sweep, it is possible to perform the compare and swap operations in parallel on each pair, as allowed by the number of comparators. It can be seen that with gap $\geq p$, all the comparators can be used in parallel to operate on distinct pairs of records. Thus it is possible to achieve a high utilization of the available comparators and a speedup of individual sweeps dictated by the total number of active comparators. This is illustrated in Figure 2.

With gap $< p$, only gap active comparators can be utilized for the current sweep. Hence with smaller gap, the average utilization of each comparator decreases with only one active comparator in the last iteration where successive records are compared and swapped with gap $= 1$. At this point, the computation is fully sequential. The total time to completion with p processors can be expressed as follows,

$$T = T_c \left[\frac{n}{p} \left[\log_r \left(\frac{n}{p} \right) + \frac{r(p-1) - 1 + p/n}{(r-1)} \right] - \log_r(p) + n \right]$$

as shown in Figure 3. Here, n is total number of records, T_c is the cycle time, r, the gap ratio $= 1.3$, and p is the total number of parallel comparators. It can be seen that with $p = 1$, the time complexity becomes $O(n \log n)$.

Even though the above strategy produces speedup with increasing number of comparators, the bottleneck in computation is the sequential comparison, when gap $= 1$. Hence, in order to accelerate the sequential computation and utilize all the comparators, we propose a strategy to extract parallelism out of the sequential comparisons. The idea is to utilize all the comparators to perform the sequential operations on different "partitions" of the array in parallel. Any dependencies between the partitions are then resolved with a less expensive "dependency resolution" phase described in [15]. This is illustrated in Figures 4 and 5, and helps

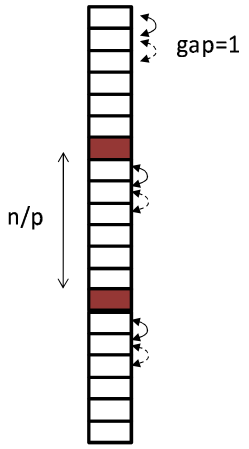

Figure 4: By dividing the array into p non-overlapping partitions, all the available comparators can be utilized to compare and swap successive records.

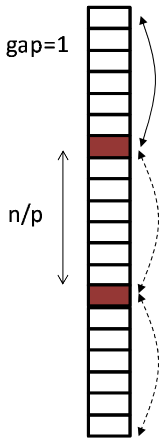

Figure 5: The dependencies between the partitions can then be resolved with a less expensive computation.

achieve another $O(p)$ speed-up for the sequential comparisons in addition to $O(p)$ speedup for the parallel comparisons. Though the technique could be utilized for any sufficiently small value of gap, our implementation only takes advantage of this technique when gap $= 1$, thus speeding up the last iteration. The performance is then

$$T = T_c \left[\frac{n}{p} \left[\log_r \left(\frac{n}{p} \right) + \frac{r(p+1) - 3 + p/n}{(r-1)} \right] - \log_r(p) + p \right]$$

as shown in Figure 6.

The above sort algorithm is used in a streaming manner by adding an additional two ports to the memory array and using them to enable the concurrent loading of a first group of records, sorting of a second group of records, and unloading of a third group of records. The capacity of the memory array must be 3 times the size of the group of records.

The memory array sorter was coded at the behavioral level, and the simulation performance results are included

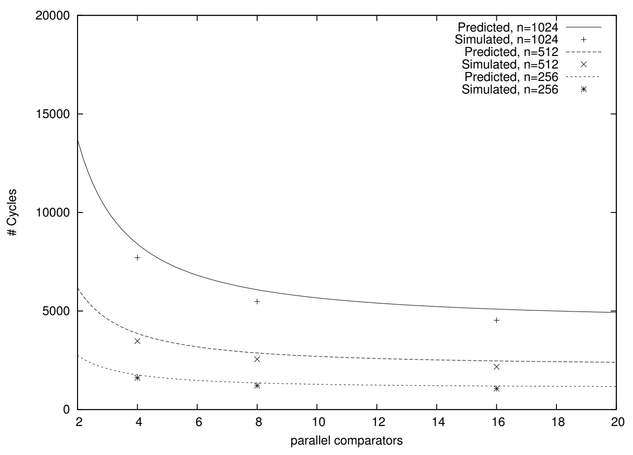

Figure 6: Modeled and simulated time to sort n=1024, n=512, and n=256 records vs. the total number of parallel comparators, p.

in Figures 3 and 6. Note the close match between the performance predictions based on the analyic models and the performance predictions based on the simulations. Unfortunately, when this model was presented to the synthesis engine, it was unable to effectively create the shared-port memory structures, and a small (30 record) array consumed almost 40% of the resources on a Xilinx Virtex-5 LX300T-1 FPGA. This is clearly infeasible as a deployment option in its current form.

3.3 Systolic array sort

An alternate approach to the `sort groups` block is the use of a systolic array sort. One classic approach to systolic array sorting was described by Leiserson [21]. In this design (illustrated in Figure 7), systolic cells are arranged in a linear array, each cell containing storage for a pair of records. New records are pushed in from the top, with the array acting as a push-down stack for the insertion operation. After each insertion, the key fields for each pair of records are compared and the record with the smaller key is placed in the higher register (closer to the insertion end of the array). Once the group of records is present, they are removed (in sorted order) by popping out of the top (with a compare and swap at each cell after each pop operation).

If the shift (either up or down) and compare and swap are all implemented in a single clock cycle, the array can be loaded one record per clock and unloaded one record per clock. We have coded this design and the following results are reported from the tool set. Targeting a Xilinx Virtex-5 LX330T-1, and specifying a 128 cell array (which holds 256 records), the synthesis and place-and-route tools report an operating frequency capable of 140 MHz and a LUT utilization of 27% (flip-flop utilization is lower, at 8%). As the registers are all allocated to flip-flops, 100% of the block RAMs are available for the merge operations and other buffering.

3.4 Merge sort

Given the above two options for the `sort groups` blocks, what remains are the single-input and dual-input merge blocks. Figure 8 illustrates how simple (conceptually) it is

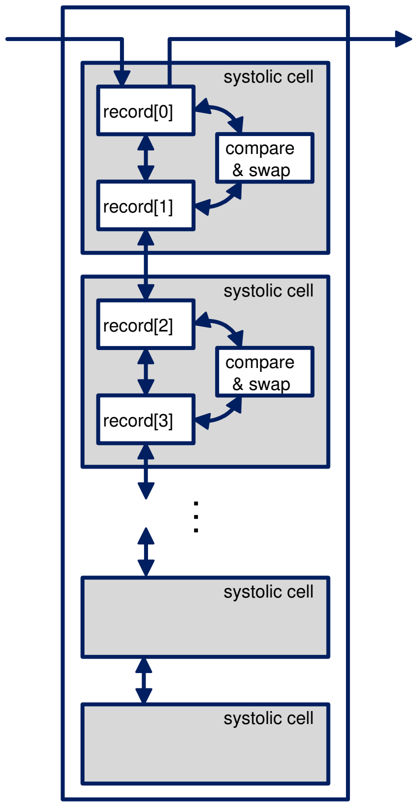

Figure 7: Systolic array based `sort groups` block. The group size that can be supported is twice the number of systolic cells, since each systolic cell holds up to two records.

to implement the single-input `merge groups` block. We will constrain our analysis to sorting $M = 2$ groups of records.

For each pair of input groups of records, the first group is routed to the top FIFO and the second group is routed to the bottom FIFO. The capacity of each FIFO is double the incoming group size. Once a pair of groups have been input, the key fields of the records are compared and they are output, smallest key to largest. Concurrent with this comparison, another pair of input groups are received.

Depending on the bandwidth of external memory (e.g., using QDR SRAM), it might be possible to share a single external memory port across more than one `merge groups` block. In the performance analysis of the next section, we will assume that the capacity of any external memories (both in terms of size and effective bandwidth) are provided and that a single record input per clock and a single record output per clock is the I/O capability of the block.

The dual-input `merge` block is simpler, only requiring the comparator itself, assuming it can exploit the input buffers available from the Auto-Pipe tool set on each incoming edge (the buffers will be considered in the performance analysis below). As above, a single record is input on each port per clock and a single record can be output per clock. Again targeting a Virtex-5 LX330T-1, the synthesis and place-and-route tools report an achievable operating frequency of 180 MHz and a utilization of under 1% (for both LUTs and flip-flops).

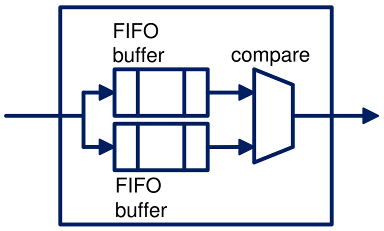

Figure 8: Structure of `merge groups` block. The input group size that can be supported is determined by the capacity of the FIFO buffers, which can be implemented with on-chip block RAMs or off-chip SRAM or DRAM. The output group size is twice the input group size.

4. PERFORMANCE

In this section, we will relate the throughput and capacity of the sorting application to the properties of the individual `sort groups` blocks and `merge groups` blocks, the properties of the memory subsystems available on the FPGA execution platform, and the I/O capability of the FPGA execution platform. The model will first focus on the FPGA subsystem, and then relate that to the combined FPGA/processor-based overall sorting application.

4.1 FPGA subsystem

The performance model initially assumes that the FPGA subsystem is doing a complete sort on a substantial subset of the data, and that all of the memory bandwidth available is dedicated to sorting. As such, it will not be able to concurrently input records and output records, since the last input must be available before the first output can be sent out. This assumption will later be relaxed, and its implications examined. One of the implications of this assumption is that we can assess the throughput in terms of a pair of properties, the ingest rate (the rate at which records can be input to the FPGA sorting application) and the egress rate (the rate at which records can be output from the FPGA sorting application). Any delays due to processing of the records internal to the FPGA will be accounted for as impacting ether the ingest or egress rates. The overall latency of an individual sort execution is then determined by the sum of the ingest time, egress time, and any internal latency between the input of the last record and the output of the first record.

Table 1 lists the variables used as part of the model. Where their definitions are not immediately apparent, they are described at their introduction in the text below.

Starting with the `split` blocks, we refer to this as pipeline stage 1, with group size G_1. The `split` block is responsible for dividing the input stream of individual records into groups of size G_1 and routing them to the appropriate `sort groups` block. With reasonable values of R, this should be achievable in one clock per record, so the inbound throughput will be the I/O bandwidth and the outbound throughput (for each pipeline) will be the minimum of f (in records/s) and the pipeline's fraction of the I/O bandwidth.

$$Tput_{in,1} = \frac{BW_{IO}}{R}$$

Table 1: Summary of variable definitions for the performance model.

Variable	Units	Meaning
N	–	number of parallel pipelines
P	–	number of pipeline stages
f	MHz	clock frequency
R	Bytes	size of a record
BW_{IO}	MB/s	I/O bandwidth to/from FPGA
G_i	records	group size at pipeline stage i
$BW_{mem,i}$	MB/s	memory bandwidth at stage i
$Tput_{in,i}$	records/s	ingest rate for pipeline stage i
$Tput_{in}$	records/s	overall ingest rate
$Tput_{out,i}$	records/s	egress rate for pipeline stage i
$Tput_{out}$	records/s	overall egress rate

$$Tput_{out,1} = \min\left(f, \frac{BW_{IO}}{NR}\right)$$

The group size G_1 is determined by the capacity of the `sort groups` blocks, so $G_1 = G_2$.

Stage 2 is comprised of the `sort groups` blocks. With the infeasibility of the memory array sort algorithm, we will concentrate on the use of the systolic array sorter. The ingest of the systolic array sorter is clearly one record per clock, giving a throughput of f when it is accepting input. Since it takes the same amount of time to deliver the sorted group downstream, the overall effective rate for a single `sort groups` block is $f/2$. If we add an additional `split` block with a fanout of 2, deploy two parallel `sort groups` blocks, and follow them with a `merge` block (as illustrated in Figure 9), the total effective rate can be raised to f by simply alternating which of the 2 `sort groups` blocks are accepting input and delivering sorted records downstream.

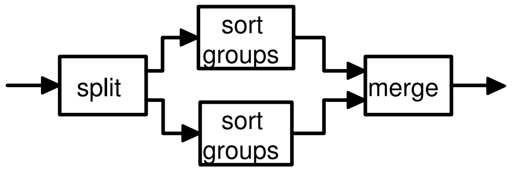

Figure 9: Stage 1 with a pair of systolic array `sort groups` blocks.

If there are P total pipeline stages, stage 1 is comprised of the initial `split` block, and stage 2 is comprised of one or two `sort group` blocks as described above, stages 3 to $P-1$ are then comprised of `merge groups` blocks. Defining the group size G_i as the size of the group output from a merging block, $G_i = MG_{i-1}$ for each of these pipeline stages ($3 \leq i \leq P-1$). With $M = 2$, this reduces to $G_i = 2G_{i-1}$ and $G_{P-1} = 2^{P-3}G_2$.

For the initial stages of merge sorting, the buffers can be internally constructed using block RAMs. For the later stages, the buffers will be external and off-chip SRAM or DRAM will be needed. We will model the access to memory (either internal or external) at pipeline stage i by its effective bandwidth, $BW_{mem,i}$, which might reflect sharing with other stages and/or parallel pipelines.

Since on each clock a write is issued to only one of the two FIFOs and a read is only required from one of the FIFOs, the bandwidth required from the memory is 2 records per clock cycle. This gives a ingest and egress throughput of

$$Tput_{in,i} = Tput_{out,i} = \min\left(f, \frac{BW_{mem,i}}{2R}\right).$$

The `merge` block (stage P) is capable of providing a single output record per clock, and consuming (on average) a single input record every Nth clock on each input port. This gives an ingest rate for each input port of

$$Tput_{in,P} = \min\left(\frac{f}{N}, \frac{BW_{IO}}{NR}\right)$$

and an egress rate of

$$Tput_{out,P} = \min\left(f, \frac{BW_{IO}}{R}\right).$$

The group size output from the `merge` block is $G_P = NG_{P-1}$.

With expressions for each of the stage throughputs, we can express the overall pipeline throughput as the minimum of the constituent throughputs:

$$Tput_{in} = \min\left(Tput_{in,1}, N \cdot \min_{2\leq i\leq P} Tput_{in,i}\right)$$

and

$$Tput_{out} = \min\left(N \cdot \min_{1\leq i\leq P-1} Tput_{out,i}, Tput_{out,P}\right).$$

These relationships between block performance, I/O performance, memory performance, and throughput can be used in a number of ways. We will seek to determine the largest size group of records, G_P, that can be sorted at maximum throughput.

Consider a first candidate design that is to be deployed on a board such as the Nallatech FSB-Expansion [25]. In addition to a Virtex-5 LX330T-1, the board contains 4 independent banks of QDR-II SRAM with a total aggregate bandwidth of 16 GB/s (4 GB/s per bank). On-chip, the FPGA has more than 1.3 MB of block RAM available with over 600 independent memory ports.

If we use the systolic array sort engine, the achievable clock rate is $f = 140$ MHz and $G_1 = 256$ records. Using the topology of Figure 9, over 50% of the LUTs are consumed by the pair of `sort groups` blocks, so only one pipeline will fit in the part (i.e., $N = 1$). The first several rows of Table 2 show the memory usage of the `merge groups` blocks that use internal block RAM. G_2 is measured at the output of the `merge` block of Figure 9 and represents the memory provided by the Auto-Pipe system in the edges of the application topology graph.

Using each of the 4 SRAM banks on the board, this enables 4 stages of `merge groups` blocks that use external SRAM. These banks cannot be effectively shared across more than one block, as their operating frequency isn't double the rate required by the block. Stages 9 through 12 in Table 2 quantify the external memory usage. This yields a design in which the FPGA sorts groups of size one half million records (i.e., $G_P = 512$ Krecords). The throughput achievable is 140 million records per second, since the I/O bandwidth and memory bandwidths are not performance limiters at any stage. The latency for a single group sort is 7 ms, 3.5 ms for ingest and 3.5 ms for egress. Note that this design doesn't use the DRAM at all. One could add an additional

Table 2: **Memory usage in merge operations.**

Pipeline Stage (i)	Memory Type	Memory Size	Group Size (G_i)
2	block RAM	8 KB	512 records
3	block RAM	16 KB	1 Krecords
4	block RAM	32 KB	2 Krecords
5	block RAM	64 KB	4 Krecords
6	block RAM	128 KB	8 Krecords
7	block RAM	256 KB	16 Krecords
8	block RAM	512 KB	32 Krecords
$\sum$ 2 to 8	block RAM	1 MB	← total block RAM
9	SRAM	512 KB	64 Krecords
10	SRAM	1 MB	128 Krecords
11	SRAM	2 MB	256 Krecords
12	SRAM	4 MB	512 Krecords
$\sum$ 9 to 12	SRAM	8 MB	← total SRAM

merging stage exploiting the system DRAM, but that would have to contend with processor access to the memory, so we do not consider that option here.

Given that there is sufficient I/O bandwidth, the above FPGA subsystem is capable of ingesting and sorting the second half million records concurrent with the egress of the initial half million records. As such, the 140 million record per second rate can be sustained across an indefinite quantity of streaming data. While we do not have access to the Nallatech board that formed the basis for the above example design, our past experience has shown that up to about 75% efficiency on an I/O bus (a category typically less capable than the front-side bus) is readily achievable [6].

As a second candidate design, a 64-cell systolic array consumes only 13% of the LUTs on the FPGA, which implies that 4 copies of the `sort groups` block could fit on one part. If the topology of Figure 1 is revisited with $N = 2$ pipelines, each pipeline can be clocked at 140 MHz as before, but the `merge` block at stage P can be clocked at 180 MHz. (Actually, one would want the `merge` as the last block using block RAMs for buffering with the SRAM-based `merge groups` blocks following it. They can all be clocked at 180 MHz.) While the number of merging stages will increase, the total buffering available (both on-chip and off-chip) is the same, giving $G_P = 512$ Krecords as before. This design, however, has a throughput of 180 million records per second, an improvement of about 30% over the earlier design.

The simplicity and speed of the merge blocks guides us to consider a design that eliminates the `sort groups` blocks entirely. Figure 10 illustrates a design comprised entirely of a single pipeline of `merge groups` blocks.

Here, the initial `merge groups` block is actually only merging a pair of individual records. In general, $G_i = 2^i$ for this pipeline. With 15 pipeline stages using block RAM, the 1 MB of available on-chip memory is allocated, and 4 additional stages using SRAM returns us to the half million record capacity of the previous two designs. At less than 1% resource (i.e., LUT and/or flip-flop) utilization for each stage to implement the `merge groups` block, the 19 stages easily fit on the part. Here, the entire pipeline can be clocked at 180 MHz, again achieving a throughput of 180 million records per second.

This last design has the elegant property that there are $O(\log n)$ comparators used to sort in $O(n)$ time on a problem that requires $\Theta(n \log n)$ comparisons. It is asymptotically optimal in terms of time and comparator usage.

4.2 Full sorting application

If the sorting application as a whole is sorting only half a million records, the only task for the processor to do is provide the records to be sorted and receive the sorted results. On the other hand, in the more likely scenario that more records are to be sorted, the processor (or collection of processors) only need to perform additional merge sorts on the groups of sorted records returned from the FPGA. These merge sorts will use the system DRAM available to the processors, which typically has much greater capacity than the on-board memory of an FPGA board.

5. CONCLUSIONS

We have presented a family of approaches to sorting on architecturally diverse systems. Quantitative assessments have been made concerning a memory array sort (based in part on comb sort), a systolic array sort, and a pipeline of merge sorts. The merge sort pipeline comes out ahead in virtually every comparison, achieving asymptotically optimal comparator usage and execution time.

As future work, we believe that a carefully placed and routed `merge groups` block (with manual help) could potentially clock at a higher rate, further improving the overall throughput. In addition, the simple FIFO queueing employed in these designs is somewhat wasteful, as queue occupancy averages approximately half full, only filling to capacity in extreme circumstances. A more space-efficient queueing discipline could help to increase the size of the record groups that are sorted on the FPGA.

With respect to the memory array sort algorithm, two possible paths present themselves. First, a low-level, hand-coded, multi-port memory structure is very likely feasible in an FPGA. We are currently working on implementing just such a design. Second, graphics engines have precisely this structure in their on-chip shared memories. It is possible that the memory array sort algorithm is well suited to execution on a graphics engine.

6. ACKNOWLEDGMENTS

This research has been supported by NSF grants 0427794, 0751212, and 0905368, NIH grant R42HG003225 (through BECS Technology, Inc.), and Exegy, Inc. R.D. Chamberlain is a principal in BECS Technology, Inc., and Exegy, Inc.

7. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Atieno, J. Allen, D. Goeckel, and R. Tessier. An adaptive Reed-Solomon errors-and-erasures decoder. In *Proc. of Int'l Symposium on Field Programmable Gate Arrays*, pages 150–158, Feb. 2006.

[3] Z. K. Baker and V. K. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. In *Proc. of 13th IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 3–12, 2005.

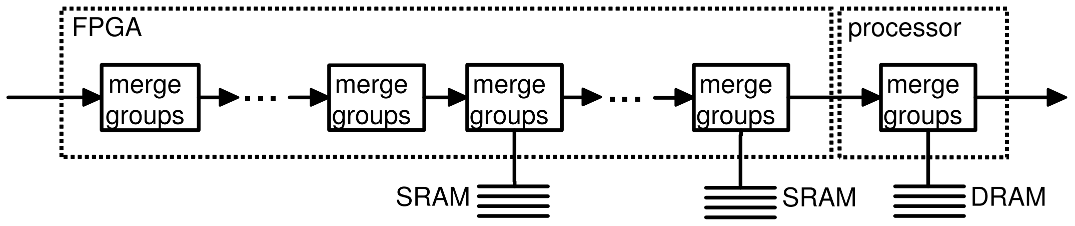

Figure 10: Sorting application topology comprised entirely of `merge groups` blocks.

[4] M. Bednara, O. Beyer, J. Teich, and R. Wanka. Tradeoff analysis and architecture design of a hybrid hardware/software sorter. In *12th IEEE Int'l Conf. on Application-Specific Systems, Architectures and Processors*, pages 299–308, July 2000.

[5] G. S. Brodal, R. Fagerberg, and G. Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proc. of Int'l Conf. on Automata, Languages and Programming*, pages 576–588, Aug. 2005.

[6] R. Chamberlain, B. Shands, and J. White. Achieving real data throughput for an FPGA co-processor on commodity server platforms. In *Proc. of 1st Workshop on Building Block Engine Architectures for Computers and Networks*, Oct. 2004.

[7] R. D. Chamberlain, J. M. Lancaster, and R. K. Cytron. Visions for application development on hybrid computing systems. *Parallel Computing*, 34(4–5):201–216, May 2008.

[8] R. D. Chamberlain, E. J. Tyson, S. Gayen, M. A. Franklin, J. Buhler, P. Crowley, and J. Buckley. Application development on hybrid systems. In *Proc. of ACM/IEEE Supercomputing Conf.*, Nov. 2007.

[9] W. Chen, P. Kosmas, M. Leeser, and C. Rappaport. An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm. In *Proc. of Int'l Symposium on Field Programmable Gate Arrays*, pages 213–222, Feb. 2004.

[10] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell. The promise of high-performance reconfigurable computing. *IEEE Computer*, 41(2):69–76, Feb. 2008.

[11] V. Estivill-Castro and D. Wood. A new measure of presortedness. *Information and Computation*, 83(1):111–119, 1989.

[12] V. Estivill-Castro and D. Wood. Practical adaptive sorting. In *Advances in Computing and Information – Proc. of Int'l Conf. on Computing and Information*, pages 47–54. Springer-Verlag, 1991.

[13] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer. Auto-pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, Apr. 2006.

[14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. of 40th IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[15] N. Ganesan, R. D. Chamberlain, and J. Buhler. Acceleration of binomial options pricing via parallelizing along time-axis on GPU. In *Proc. of Symp. on Application Accelerators in High Performance Computing*, July 2009.

[16] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. of SIGMOD Int'l Conf. on Management of Data*, pages 325–336, 2006.

[17] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation of linear lists. In *Proc. of 9th ACM Symposium on Theory of Computing*, pages 49–60, 1977.

[18] M. C. Herbordt, J. Model, Y. Gu, B. Sukhwani, and T. VanCourt. Single pass, BLAST-like, approximate string matching on FPGAs. In *Proc. of 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 217–226, 2006.

[19] D. E. Knuth. *The Art of Computer Programming. Vol 3. Sorting and Searching.* Addison-Wesley, 1973.

[20] S. Lacey and R. Box. A fast, easy sort. *Byte*, 16(4):315–ff., Apr. 1991.

[21] C. Leiserson. Systolic priority queue. In *Proc. of Caltech Conference on VLSI*, pages 200–214, Jan. 1979.

[22] H. Manilla. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, 34:318–315, 1985.

[23] R. Marcelino, H. Neto, and J. Cardoso. Sorting units for FPGA-based embedded systems. In *Distributed Embedded Systems: Design, Middleware and Resources – Proc. of IFIP 20th World Computer Congress*, pages 11–22, 2008.

[24] J. Martinez, R. Cumplido, and C. Feregrino. An FPGA-based parallel sorting architecture for the Burrows Wheeler transform. In *Proc. of Int'l Conf. on Reconfigurable Computing and FPGAs*, 2005.

[25] Nallatech. `http://www.nallatech.com`.

[26] R. Scrofano, M. Gokhale, F. Trouw, and V. K. Prasanna. Hardware/software approach to molecular dynamics on reconfigurable computers. In *Proc. of 14th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 23–34, 2006.

[27] J. L. Tripp, H. S. Mortveit, A. A. Hansson, and M. Gokhale. Metropolitan road traffic simulation on FPGAs. In *Proc. of 13th IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 117–126, 2005.

[28] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.