FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting

Dirk Koch and Jim Torresen
Department of Informatics, University of Oslo, Norway
Email: {koch, jimtoer@ifi.uio.no

ABSTRACT

This paper analyses different hardware sorting architectures in order to implement a highly scaleable sorter for solving huge problems at high performance up to the GB range in linear time complexity. It will be proven that a combination of a FIFO-based merge sorter and a tree-based merge sorter results in the best performance at low cost. Moreover, we will demonstrate how partial run-time reconfiguration can be used for saving almost half the FPGA resources or alternatively for improving the speed. Experiments show a sustainable sorting throughput of 2GB/s for problems fitting into the on-chip FPGA memory and 1 GB/s when using external memory. These values surpass the best published results on large problem sorting implementations on FPGAs, GPUs, and the Cell processor.

Categories and Subject Descriptors

B.0 [**Hardware**]: GENERAL

General Terms

Design, Experimentation, Performance

1. INTRODUCTION

Large problem sorting is a process that is heavily used in database systems [8]. In this paper we introduce a highly optimized sorter implementation for FPGAs that has the potential to scale to sort problem sizes of several GB of data. The data items to be sorted are commonly referred as *keys*. Keys can be integer, floating-point numbers or any other sortable data type. Throughout this paper we focus on 64-bit integers.

Sorting a huge amount of data means that the whole sorting problem cannot fit into on-FPGA memory. Consequently external memory is required to store intermediate values. This also means that the sort keys have to be transferred to the FPGA multiple times, according to the actual problem size and the used sorter architecture.

For the following observations, let us assume the system illustrated in Figure 1. The System provides an FPGA that includes the basic infrastructure, consisting of a host interface (e.g., PCIe or a front-side bus), memory controllers, and logic to perform run-time reconfiguration. At first, let us consider only one local memory channel. Then, the sort keys arrive via the host interface to the FPGA from where they are stored in external memory attached to the FPGA. We assume that the local on-board memory provides sufficient capacity to host the whole search problem and that it provides the same throughput as the host interface.

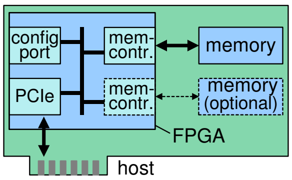
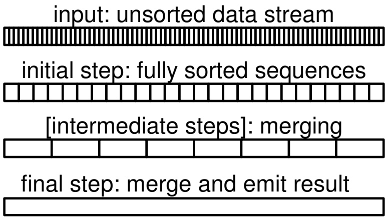


Figure 1: Assumed system with host connection and local memory.


input: unsorted data stream
initial step: fully sorted sequences
[intermediate steps]: merging
final step: merge and emit result

$$
\begin{aligned}
\underbrace{N}_{\#keys} &= \underbrace{b}_{\#blocks} \times \underbrace{s}_{blocksize\,[keys]} \\
&= b_1 \times s_1 \quad ; b_1 = N,\ s_1 = 1(init) \\
&= b_2 \times s_2 \quad ; b_2 < N,\ s_2 > 1 \\
&= \quad \ldots \\
&= b_M \cdot s_M \quad ; b_M = 1,\ s_M = N(final)
\end{aligned}
$$

Figure 2: Sorting N keys in M steps.

While the sort keys arrive from the host, a first sorting step can be performed such that the local memory will store blocks of fully sorted sequences. As illustrated in Figure 2, these sequences will then be merged to fewer but larger fully sorted sequences in one or more successive sorting steps. Note that throughout the whole paper, we assume that the input sort keys are fully random.

Similar to performing some of the sorting during arrival of the data, a final sort step might be performed when writing the result back to the host. Independent of the problem size, this can only start after the arrival of all sort keys. This is obvious as the last input key from the host could be the first key of the result.

The following sections first discuss the compare-swap el-

FPGA'11, February 27–March 1, 2011, Monterey, California, USA.

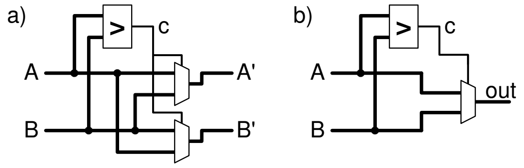


Figure 3: Basic sorting elements: a) compare-swap element, b) select-value element.

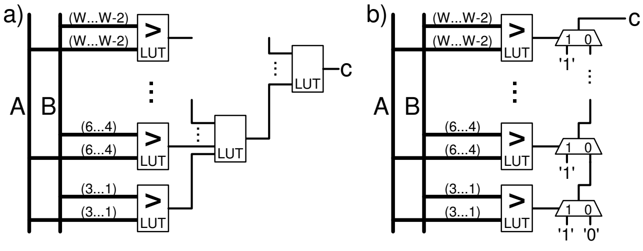


Figure 4: Magnitude comparator implemented with 6-input LUTs. a) tree-based, b) carry-chain logic.

ement, which is the basic building block of a sorter. After this, various sorter architectures are analyzed with respect of their applicability of sorting the different steps.

1.1 Basic Sorting Elements

The basic building block for most sorting architectures is a compare-swap element that compares two input values and swaps the values at the output, if required. A compare-swap element is depicted in Figure 3 a). Another widely used basic building block is a select-value element shown in Figure 3 b). The function of this element is to provide the smaller (larger) value if the task is a ascending (descending) sorting. Typically, the selected value will be consumed at the input while leaving the other (not selected) value for a comparison in the next cycle.

1.1.1 Comparators

While a multiplexer contributes only one logic level to the latency, the comparator is typically the critical part of the compare-swap element with respect to latency. Depending on the FPGA synthesis tool, comparators are commonly mapped to tree-based structures or implementations using the carry chain logic that is commonly provided in FPGA architectures. Figure 4 shows examples of how comparators for natural numbers ($\mathbb{N}_0$) can be mapped to 6-input LUT-based FPGAs.

Assuming 6-input LUTs, which are common for all recent high-density FPGAs, the amount of logic levels in a tree-based comparator is for W bit wide operands $1 + \lceil log_6 W \rceil$ levels. For 32-bit or 64-bit operands this results in three logic levels only for the comparator. The corresponding carry chain would be $\lceil \frac{W}{3} \rceil$ LUTs or $\lceil \frac{W}{(3 \cdot 4)} \rceil$ Xilinx Virtex-5 slices long. On recent Xilinx devices (such as Virtex-5/6 or Spartan-6 FPGAs), one slice comprises a cluster of four 6-input look-up tables. For 32-bit (64-bit) operands, this results in a chain being 3 (6) slices long. Consequently, carry-chains result in lower latency for wide operands. However, tree-based comparator architectures are still an option for Spartan-6 devices where only half the amount of LUTs provide attached carry chain logic.

We found that the recent vendor logic synthesis and mapping tools (XST 12.1) were only able to compare two operand bits per 6-input LUT, in spite of the FPGA fabric allowing the implementation of a comparator with three operand bits exactly as shown in Figure 4 b). For investigating a denser fitting, we explored the physical implementation results achieved for a Xilinx Virtex-5 device. The results listed in Table 1 denote only the latency of a combinatory comparator chain and the values do not include any further elements such as the routing latency for connecting the comparator. Exploiting all six look-up table inputs would reduce the length of the carry chain to $\frac{2}{3}$ times the size of what is required when using the Xilinx vendor tools. This would allow implementing the 64-bit comparator with the latency that is listed for the 48-bit case in Table 1. However, the saving is then only 120 ps or roughly 5% of the slack available in a data path running at 250 MHz. To take advantage of the more compact comparators when using the Xilinx vendor tools, it is possible to instantiate hard macros that have to be manually created.

Table 1: Latency of a carry-chain comparator implemented on a Virtex-5 XC5VLX50-3 device.

Operand width (W)	8	16	32	48	64
latency [ns]	0.57	0.65	1.15	1.3	1.42
LUT count	4	8	16	24	32

1.1.2 Multiplexers

In the basic compare-swap element, the result of the comparator is used to control a multiplexer pair for adjusting the input operands in order to deliver a sorted result. Therefore, the basic element demands two look-up tables per signal bit of the operand, which is $2 \times W$ LUTs in total. Consequently, the multiplexers contribute to the majority of the implementation cost in terms of LUTs. When using the Xilinx vendor tools, this results in 80% of the logic cost just for the multiplexers when implementing a basic compare swap element (see also Table 1 for the comparator logic cost). However, this value can be improved by utilizing the second look-up table output in order to implement a single bit multiplexer pair inside the same LUT. Then the total logic cost for implementing the compare-swap element can be reduced to 60% as compared to the baseline implementation when using the standard VHDL operators and the Vendor tools. A second LUT output is available on 6-input LUT devices from Xilinx and Altera. In case of Xilinx devices, this technique is typically only suited for Spartan-6 or Virtex-6 series FPGAs, because these devices provide a flip-flop for both LUT outputs whereas the older Virtex-5 device only allowed one result to be registered. Despite the benefit and low cost of the look-up table sharing, the Xilinx vendor synthesis and mapping tools (XST 12.1) do not make use of the second LUT output when implementing multiplexers. Again, this can be exploited by instantiating corresponding LUT primitives or manually implemented hard macros.

Depending on the sorter architecture, it can be beneficial to sort more than two values within a basic compare-swap element at the same time. For example, by sorting 4 input vectors simultaneously, the work of 6 (5) two-value compare-swap elements can be done in one step when considering bitonic (even-odd) merge sorting networks (see Section 2.1 for more details on sorting networks). This approach does not help to reduce the number of comparators, but it helps to save logic for the multiplexers that contribute to the majority of the logic cost within a compare-swap element. Combining multiple sorter stages is particularly useful when considering the select-value element with only one

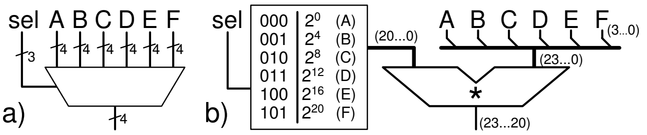


Figure 5: Multiplexer implementation using a dedicated multiplier primitive.

output multiplexer for picking the entire smallest or largest sort key as the result.

While single 6-input LUTs permit directly implementing 4:1 multiplexers, wider multiplexers comprise more logic levels and thus result in higher latency. As one solution to this problem, dedicated hard-IP multipliers (found in all high-density FPGAs), can be used to efficiently implement wide input multiplexers [1]. This approach costs nothing as these primitives would otherwise be unused in our sorting application. An example of a 4-bit wide 6:1 multiplexer implemented using a single multiplier is depicted in Figure 5. The same multiplier would allow a 2-bit (1-bit) wide 12:1 (24:1) multiplexer implementation with the same latency (when not further considering the select encoding).

1.1.3 Wide and Fast Operand Sorting

Large problem sorting is often performed in databases. Here, the search keys typically represent not only simple integer numbers, but structs concatenating multiple sub-keys of arbitrary type, including integer, floating point numbers, or even strings. The size of a single key can exceed a hundred bytes or even more.

For biased floating-point number representations (such as IEEE 754 floating point numbers), the comparator design is very similar to the integer case when firstly comparing the exponent lexicographical and secondly taking the comparison of the mantissa into account. The comparison of strings can be very difficult and we will bound our investigation to integer numbers. Strings are more difficult to handle as there exist different text codes (e.g., ASCII, UTF-x) and different language and even context dependent rules for sorting [17].

Considering fully random search keys allow in more than 99% ($\frac{1}{2^8}$) of all comparisons stopping the comparison after processing the most significant byte. However, in databases, search keys might posses a common prefix that has to be verified (e.g., when sorting a date field). For this reason, we consider the whole key at once.

It can be seen from the basic compare-swap element in Figure 3a) that the sorting of wide operands can easily be done in multiple consecutive time steps processing only a subword per step. Then we must add a simple state machine to each compare-swap element that keeps track of the present compare state. When arranging the keys from most significant subword to least significant subword, the compare-swap element can start emitting the result with only one cycle latency. This is possible because the output will either be the same (in the case of a common prefix) or can be determined within the same clock cycle.

At an extreme, the size of the subwords could be reduced to a single bit, allowing the implementation of the basic compare-swap element in a single Virtex-5 slice. This would result in a low throughput of a single element but would allow parallel processing with many tens of thousands compare-swap elements in parallel on the largest available devices.

To enhance the throughput of a single compare-swap element, the data path of a single subword must be wider. For instance, when targeting a sort key throughput of 2GB/s, the data paths of the sorting elements must be 64-bits (128-bits) wide, assuming a key fillrate of 250 MHz (125 MHz).

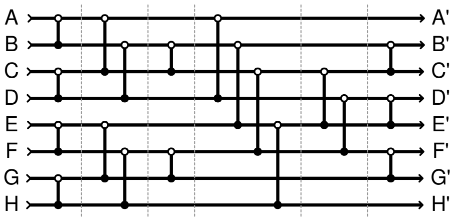


Figure 6: Even-Odd sorting network. The operator ⫯ represents a compare-swap cell from Figure 3b).

As both operands within a compare-swap element traverse the data path at the same speed, a compare-swap element (see Figure 3a)) can be easily pipelined. As opposed to this, only one operand moves in a select-value element (see Figure 3b)), while backpressing the entire other operand input stream. Such a structure is more difficult to implement for wide data paths because of the logic latency and the routing latency between the different sorting elements (see also Table 1 for the latency of a comparator).

2. SORTER ARCHITECTURES

Hardware implementations using different kinds of sorting architectures have been presented multiple times before. This section gives an overview of common approaches.

2.1 Sorting Networks

Sorting networks are mathematical models of networks with compare-swap elements designed to sort a set of input values very fast by exploiting a high degree of parallelism. Common architectures include bitonic merge sorter networks and the more efficient even-odd networks [2]. Figure 6 illustrates an even-odd network for eight input operands. The network could operate fully combinatory, but it is common to add a pipeline register after each compare-swap element. Then a set of N search keys can be sorted in $\frac{log_2N}{2}(log_2N+1)$ clock cycles ($\frac{3}{2}\cdot 4 = 6$ in the shown example). The hardware cost of such a sorting network is $\frac{N}{4}(log_2N)^2$ compare-swap elements.

Sorting Networks have been used for sorting large problems using vector operations, for example, using the Cell Broadband Engine Architecture [6] or GPUs [7]. On FPGAs, sorting networks are typically applied to smaller problem classes, for example, to compute a median value [15]. For large problems, sorting networks require greater I/O throughput as they consume more sort keys and produce a huge amount of result data at the same time. This leads to alternating operation between I/O bursts and processing bursts and consequently in poor utilization of the I/O interface and the logic fabric. However, in [11] symmetries within the even-odd network have been used to solve larger problems using multiple runs on smaller networks.

2.2 Insertion Sorting

Hardware implementations of insertion sorting provide a shift register for storing the search keys. As illustrated in Figure 7, the shifting of each individual register is controlled by attached comparator logic. By comparing the input value with the current register content, a free position will be generated at the right position within the register by shifting

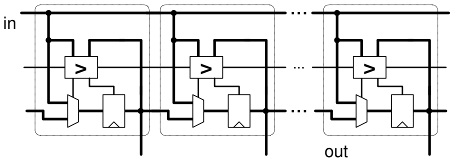


Figure 7: Insertion sorter.

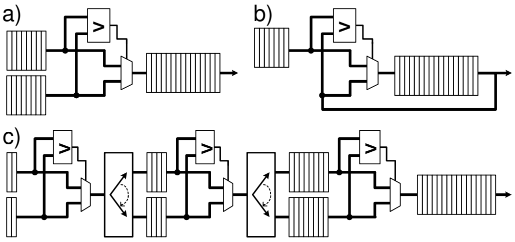


Figure 8: FIFO-based merge sorter. a) baseline approach [12], b) circulating merge sorter [13], c) cascading FIFO-based merge sorters.

all larger sort keys one position further. The resource requirements are N comparators and N registers for sorting N keys in N operation cycles.

The basic architecture of a insertion sorter would not allow fast and large implementations with many comparators because the input signal has to be broadcasted to all comparator instances. However, in [3], a systolic merge sorter implementation is presented that achieved with 128 processing elements a 66 MHz operation speed on a Xilinx Virtex XCV-1000-4 FPGA. Because of the relatively high implementation cost and because the search keys have to be stored in registers instead of using dedicated on-chip memory blocks, we consider insertion sorting only for smaller problem sizes.

2.3 FIFO-Based Merge Sorter

FIFO-based merge sorters consist of a select value element (see Figure 3b)) that is surrounded by FIFOs at the input and output ports. As illustrated in Figure 8a), this allows the generation of an output stream possessing the combined size of the two input FIFOs.

In [13], only one shared FIFO was used by feeding back the output FIFO to one input, as depicted in Figure 8b). We do not consider this approach, as it results in a data-dependent execution time with quadratic time complexity for random input data.

For generating longer fully sorted sequences, multiple FIFO-based merge sorters can be cascaded, as shown in Figure 8c). At the cost of doubling the amount of local on-chip memory in each sorter stage, the size of the sorted sequence is doubled as well. A drawback of the shown data path is that it is not suitable for sorting on a continuous stream. For example, consider only the last output stage: in an initial step, the sorter fills one FIFO. After this, the sorter can immediately start to merge the keys with the arrival of the second input stream. When the second stream is fully written to the input FIFO, it cannot be guaranteed that a new sort sequence will fit in one of the input FIFOs because the processing of the actual sort job is data dependent and consequently also the free space in the input FIFOs.

2.4 Merge Sorter Trees

Select-value elements can alternatively be arranged in a tree based structure for merging multiple sorted subsequences to one combined sequence that will be fully sorted. As illustrated in Figure 9, this can be used to merge long sequences from external memory. Then, small FIFOs attached to the inputs of the first sorter stage can be used to hide the latency of the external memory.

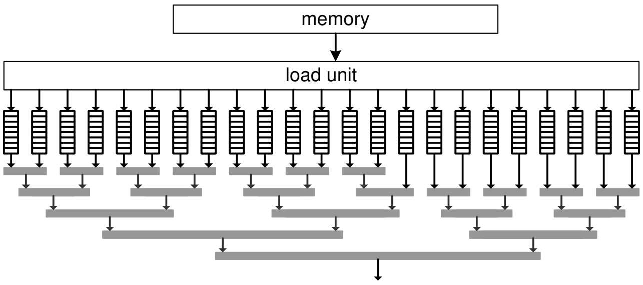


Figure 9: Merge sorter tree with key load unit. Each horizontal bar constitutes a select-value element.

With M inputs at the first sorter stage, the tree structure can merge M streams together in one run with linear time complexity, thus reducing the total amount of streams to $\frac{1}{M}$ for possible further runs. In each run, the whole data is streamed once through the sorting tree.

An interesting property of a tree sorting structure is its little switching activity. Even through the implementation cost scales linearly with M, only $log_2 M$ select-value elements are active within one operation cycle. Besides the reduced power consumption, this results in a homogenous distribution of memory load requests over time. This simplifies to achieve sustained high throughput with the external memory.

A merge sorter with a tree-like network-on-chip architecture has been demonstrated in [18] and points out that NoCs are too slow for high throughput merge sorter implementations.

2.5 Bucket Sort

Bucket sort follows a divide and conquer strategy by splitting the whole search problem into buckets containing fewer search keys than the original problem. This can be recursively repeated until the remaining problem fits into local on-chip memory for fast sorting of the different buckets. The most famous application for bucket search is the Postman's sort, where all letters are consecutively sorted into buckets for countries, states, cities, and streets. For hardware-accelerated sorting, different pivot elements are selected to define the buckets. In the case of uniform distributed search keys, buckets can be defined by the most significant bits (MSBs) of the search keys. This has been demonstrated on GPUs for an implementation of quicksort [16].

In a special case of bucket sort known as radix sort, both the size of the keys and the number of buckets are a power of two, thus requiring inspection of only a prefix of the binary encoding of the key in order to find the corresponding bucket. This was used in the fastest published papers on sorting using GPUs [9, 10].

Bucket sort is an interesting candidate for implementations on FPGAs, where an input stream of search keys might be stored in different regions of attached memory. However, when taking two buckets in each step and recursively sorting each buckets again into two smaller buckets, bucket sort is basically identical to quicksort. As a consequence, bucket sort might perform poorly (with in the worst case quadratic time complexity), if the problem is not equally distributed

Table 2: Benchmarking different sort architectures. N denotes the problem size in terms of keys and M the amount of parallel processed memory blocks. The marked (*) architectures involve bursting I/O traffic.

architecture	time complexity	space complexity logic	memory
bitonic networks*	$\frac{log_2 N}{2}(log_2 N + 1)$	$\frac{N}{4}(log_2^2 N + log_2 N$	$> N$
even-odd networks*	$\frac{log_2 N}{2}(log_2 N + 1)$	$\frac{N}{4}(log_2 N)^2$	$> N$
insertion sorter*	N	N	N
FIFO merge sorter	N	$log_2 N$	N
merge sorter trees	N (one run)	$M - 1$	M+buffer
bucket sort	N (one run)	M	M+buffer

or if the pivot elements haven't been properly chosen. This problem can especially occur when sorting database keys as they often share a common prefix, posses bounded ranges, or represent text. As we want in particular to address also database applications, we do not further consider bucket sort.

2.6 Architecture Benchmarking

In order to allow a direct comparison of the different sorting architectures, we list all candidates together with the entire complexity in both space (implementation cost) and time (throughput) in Table 2. As all data has to be streamed sequentially via an interface (e.g., memory or PCIe) to and from the FPGA, a better than linear time complexity will result at some point in time in idling the sorter hardware. Because of this and the bursting I/O traffic, sorting networks are only appropriate for relatively small problems, when implemented on FPGAs.

The most effective architecture is the FIFO-based merge sorter that achieves a linear throughput at only logarithmic logic cost. Moreover, it can use dedicated on-chip memory (block RAM) to solve larger problems fully on-chip. If the problem size exceeds the local memory, external memory is required to temporarily host blocks containing fully sorted sub-problems. Depending on the number of blocks, one or more additional runs through a tree-based merge sorter may be required.

As the FIFO-based merge sorter requires already-sorted streams at the input and because it requires only little logic cost, other sorting architectures can be used to generate fully sorted initial streams. Here insertion sorters will be most effective because of their linear implementation cost.

3. DESIGN FACTORS

With the knowledge of the right sorting architecture, we now discuss design factors that impact implementation and performance when considering relatively large problem classes. The great advantage of using an FPGA to implement a sorter is that it can implement any combination of sorting algorithms that can be perfectly scaled to the resources provided by the entire target platform. As opposed to GPUs or the Cell processor, FPGAs don't rely on a particular memory hierarchy, a special instruction set, or a communication architecture; they can directly implement the most suitable operands and link them arbitrarily together using a perfectly tailored interconnection network.

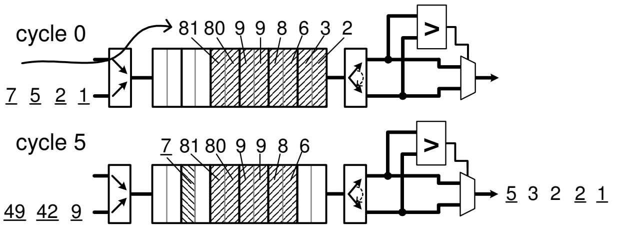


Figure 10: FIFO-based merge sorter using a shared memory block for both input streams.

The following sections address again the most promising sorting architectures, this time focusing on a highly optimized FPGA implementation. After this, we discuss further issues that impact the overall sorting system, including architecture partitioning to FPGA resources, I/O throughput performance, and the benefits of applying partial run-time reconfiguration for sorting.

3.1 Efficient FIFO Merge Sorters

The merge sorter is the heart of the sorting accelerator and has to accomplish the following two main objectives: 1) high throughput and 1) efficient use of the on-chip memory. The design of a pipelined select-value element is presented in the next section, while we concentrate on the memory in the following paragraphs.

By studying Figure 8 again, we can identify that only one input FIFO will be read in each operation cycle of the sorter. Furthermore, as already stated in Section 2.3, the sort operation can start immedatly upon fully receiving the first stream and with the arrival of the first value of the second stream. Then, assuming a streaming operation at maximum speed, one value will arrive at the input and one value will be emitted to the output in each clock cycle. As a consequence, the combined fill level of both input FIFOs will remain constant and one proprietary FIFO using one memory block for both streams is sufficient as sketched in Figure 10.

To allow zero-overhead iterations with the next set of two input streams, we have to allow that data can be written during the time the FIFO is flushed (i.e., the phase after the arrival of the last key in the second stream). As it is not possible to predict when a certain part of the FIFO will be flushed, due to the data-dependent merge sort process, we tile the memory block into multiple smaller blocks. To manage these blocks, we developed an address computation unit that organizes the different blocks as linked lists. This allows fragmentation among the blocks and permits a continues stream of sort keys.

Figure 11 illustrates the fill-phase of a FIFO merge sorter. Always two streams share one linked-list FIFO and during the time a FIFO is flushed, it stores simultaneously the next input stream. In the last select-value stage of a FIFO-based merge sorter, we can directly store the results in external on-board memory, hence not requiring a larger output FIFO. Then the largest fully sorted sequence that can be generated by this sorter is twice the size of the shared combined input FIFO. As the total amount of FIFO storage in all other preceding sorter stages is approximately the size of the last FIFO, the sorted sequence size is equal to the total on-chip memory that can be spent on FIFO storage. This holds under the assumption that both streams are of the same size. Our address computation unit is capable of handling any given input stream size, thus simplifying the partitioning of FIFOs to on-FPGA memory block primitives.

As we have to leave at least two blocks in our FIFO implementation unused and because we are not fully utilizing the

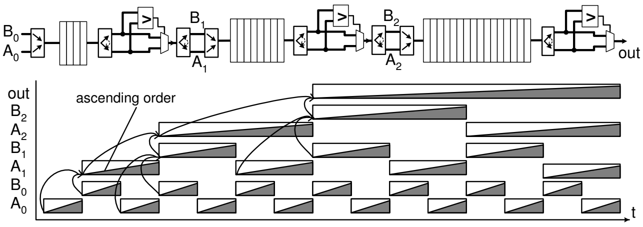


Figure 11: Fill phase of a FIFO-based merge sorter.

memory blocks in the first merge sorter stages, we achieved only an 80% efficiency out of the memory, which is still much better than the achieved 33% efficiency by the baseline approach using multiple parallel FIFOs. This value results from the fact that the baseline approach in Figure 8c) requires a third additional FIFO for storing the next initial sequence while the preceding iteration is flushed.

3.2 Throughput Optimized Tree Merge Sorter

Whenever the problem size exceeds the FPGA on-chip memory capacity, we have to perform one or more further merging steps that are best performed with a tree-based merge sorter, like the one depicted in Figure 9. The main difficulty in the design of a merge sorter tree is the backpressure flow control in the network. If we run through the operation of the tree sorter example, we see that the final root select-value element consumes one of the two input values while backpressing the entire other part. The same situation will recursively repeat on the entire selected branch of the tree in the next level until we reach the input layer to the sort tree. As a consequence, a baseline tree sorter would demand single-cycle flow control from the root sorter to every other sorter. This would consequently hinder fast operation or scaling to larger trees.

To solve this issue, we propose to decouple the operation between the different layers by adding small FIFOs between the different layers of the sorting tree. Then the propagation of the flow control against the data flow direction can take multiple clock cycles without risking the loss of any sort key value. These decoupling FIFOs can be efficiently implemented using distributed memory (i.e., using look-up tables as small memory blocks or shift register primitives); consequently leaving the much larger dedicated block RAMs for implementing input FIFOs for hiding the latency from external memory.

Due to the relatively low operation clock frequency of an FPGA as compared to CPU or GPU chips that have been manufactured in the same process technology, wide data paths must be used to sustain high sorting throughput. However, wide data paths require deep pipelining to avoid a break-down in the clock speed. The afore mentioned backpressure problem also applies within the pipeline. For example, let us assume that we pipeline the compare block of a select-value element into two parts, one for comparing the upper part of the two input operands (AH, BH) and one for the lower part (AL, BL). Then, there is the problem that the temporal results will be invalid because in each clock cycle one operand has to be shifted while leaving the other one behind.

Nevertheless, as illustrated in Figure 12, it is still possible to pipeline the compare operation. For this reason, the circuit provides three independent comparators for comparing the most significant subwords (AH, BH) in a first clock cycle. The state stored in register b is for determining the initial compare value when filling the pipeline. The registers a and c denote the speculative high word comparison in the case that either value B or value A is shifted to the output. Then, depending on the last sort decision (stored in register d), the corresponding state is considered together with the low word (AL, BL) to compute the final compare state.

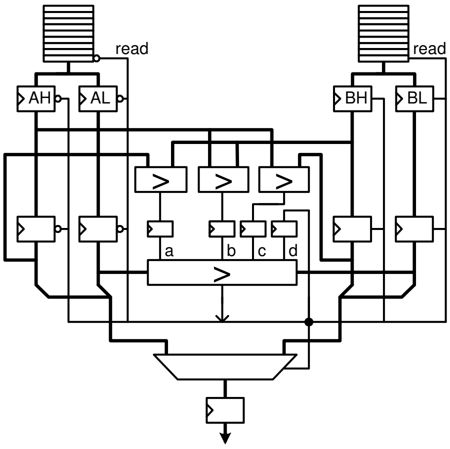


Figure 12: Pipelined select-value compare element. For the sake of readability, the example hides details on the control flow, filling the pipeline, and flushing out correctly all values from the compare element.

This proposed speculative comparison is suitable for even more deeply pipelined (and consequently wider) data paths. This approach can also be used if the path is wider than the search key in order to reorder elements inside the stream, for instance, when sorting 64-bit keys in a 128-bit data path. With these optimizations, high throughput merge sorter trees are possible on FPGAs.

3.3 Optimal Resource Partitioning

With the knowledge of the right sorter modules and their efficient implementation, we have to allocate the FPGA resources for the different modules. In the case of both the FIFO-based merge sorter and the tree merge sorter, the dedicated block RAM memories will be the limiting resource on the FPGA. In the following, we compute an optimal partition of the BRAM resources that maximize the possible problem size N that can be sorted with one initial run of the FIFO-based merge sorter and a following final merging in a sort tree. Ignoring the latency required to flush the FIFO-based merge sorter and the time to initially fill the sort tree, the result can be read back from the FPGA board immediately after the last sort key was sent to the FPGA. Consequently, a two stage sorter provides lowest possible latency and should be used as far as it it possible concerning N.

Given an estimate η denoting the memory efficiency, the cost c_F to implement a FIFO-based merge sorter in terms of memory blocks (where each block can store k keys), is for a problem size N_F:

$$c_F = \frac{N_F}{\eta \cdot k} \qquad (1)$$

Where $\eta \cdot k$ expresses the number of keys that effectively fit in one memory block.

When considering one memory block per stream input, the implementation cost c_t of a tree merge sorter being capable of simultaneously merging M streams is:

$$c_t = M \tag{2}$$

The implementation cost of both sorter modules must fit onto the total number B of available on-FPGA memory blocks:

$$B \leq c_F + c_t = \frac{N_F}{\eta \cdot k} + M \tag{3}$$

The largest problem size N that can be solved with the two modules is therefore:

$$N = N_F \cdot M = (B \cdot k \cdot \eta - M \cdot k \cdot \eta) \cdot M \tag{4}$$

The optimal size for M is found by setting the derivative of N with respect to M equal to zero:

$$\frac{\partial N}{\partial M} = B \cdot k \cdot \eta - 2M \cdot k \cdot \eta = 0 \tag{5}$$

Finally, the optimal memory block partition is:

$$M = \frac{1}{2} B \tag{6}$$

Equation 6 points out that regardless of the size of a memory block or the efficiency of the FIFO-based merge sorter implementation, the available memory resources should be equally balanced among the two sorter modules. Note that this result is valid for an implementation on any target FPGA.

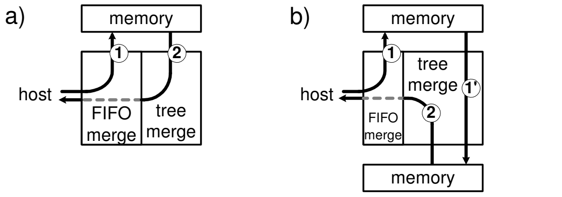
3.4 The Impact of I/O Capacity

With respect to the system depicted in Figure 1, we assumed that for the first run through the FIFO-based merge sorter, the host interface provides sufficient throughput to deliver the operands and that the memory interface provides sufficient throughput to store the intermediate sorted blocks. When running the second tree-based merge sorting step, the data flow will be in the opposite direction (towards the host PC), as sketched in Figure 13a).

If we now consider a second memory channel that is fast enough to simultaneously read and write the stream of sort keys, a merging step can be performed in parallel to the first run, as depicted in Figure 13b). Then, instead of solving a problem size of $N_1^S = \frac{1}{4}B^2k\eta$ (see also Equation 4), it is possible to sort up to $N_2^S = \frac{4}{27}B^3k\eta$ keys in linear time complexity, when allocating one third of the memory resources for the FIFO-based merge sorter and the remaining on-FPGA memory for the merge sorter tree. Again, the result can be read almost immediately after the arrival of the last key (with a small latency for flushing and filling the pipelines).

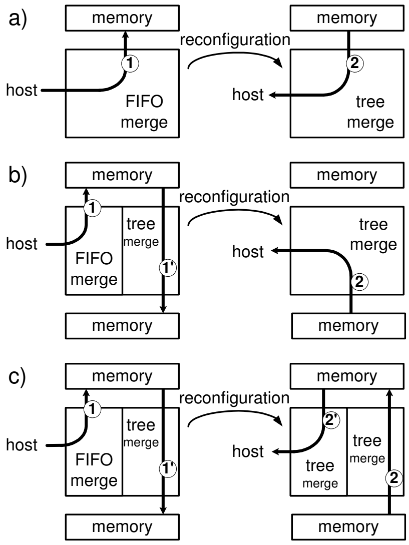
Figure 13b) shows the dataflow in a simplified manner. It must be mentioned that each of the memory blocks is not required to provide sufficient space to store the whole problem. By swapping between the two memory blocks during the run of each step, it is sufficient when both memories together are capable to host the problem.

The comparison between the two approaches does not take the additional logic overhead for a second memory channel and the slightly more advanced flow control into account, but proves that especially for implementations on larger FPGAs a second memory channel should be utilized for sorting. For very large problem sizes with several GB of sort key data, adding further memory channels is recommended if not considering a reconfiguration of the present sorter architecture.



$\frac{1}{2}Bk\eta \cdot \frac{1}{2}B = \frac{1}{4}B^2k\eta$ $\qquad$ $\frac{1}{3}Bk\eta \cdot \frac{2}{3}B \cdot \frac{2}{3}B = \frac{4}{27}B^3k\eta$

Figure 13: Data flow in a static sorting system. a) one memory channel, b) two memory channels.



System with one memory channel: $Bk\eta \cdot B = B^2k\eta$

System with two memory channels (three runs):

$$\frac{1}{2}Bk\eta \cdot \frac{1}{2}B \cdot B = \frac{1}{4}B^3k\eta$$

System with two memory channels (four runs):

$$\frac{1}{2}Bk\eta \cdot (\frac{1}{2}B)^3 = \frac{1}{16}B^4k\eta$$

Figure 14: Data flow in a sorting system using partial run-time reconfiguration.

3.5 Using Partial Run-time Reconfiguration

Throughout the last sections, we have proven that the problem that can be solved by a sorting module within one run of a particular sort module scales linear with the amount of allocated FPGA area, which in particular comprises the On-FPGA memory resources. When assuming the case that a FIFO-based merge sorter and a tree-based merger sorter have to share the same FPGA resources, each module can only solve half the problem size per run. (see also Section 3.3). Let us assume a large sort problem that should be accomplished as fast as possible by starting a single run on the FIFO-based merger sorter followed by the tree-based sorter. Then we can identify, that during the first run of the FIFO-based merge sorter the tree sorter is idle while in the second phase the FIFO-based merge sorter will be idle. For enhancing the FPGA device utilization, it is then an option to allocate the same FPGA resources for both sorter modules and using partial runtime reconfiguration to swap from the FIFO-based merge sorter to the tree-based sorter module, as depicted in Figure 14a).

3.5.1 Reconfiguration Overhead

However, for a fair comparison, we also have to consider the required reconfiguration time. The configuration time of a sorter module is directly proportional to its size. Our case study has shown that a partially reconfigurable module with 100 memory blocks (B=100) on a Xilinx Virtex-5 FPGA results in roughly 3 MB of configuration data for the surrounding logic resources and the routing, which is 30KB

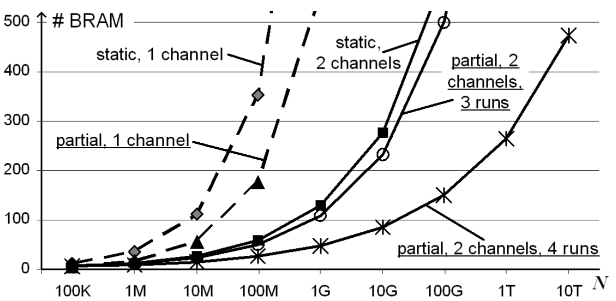


Figure 15: Memory block requirements of the different sorter architectures. The values are for the case that reading the result can be started directly after transferring all search keys to the FPGA.

Table 3: Configuration selection for different problem sizes (one memory channel).

#Keys (N)	remark	ex. time	figure
$0...Bk\eta$	only on-chip memory	$\approx N$	14a) left
$Bk\eta...\frac{1}{4}Bk\eta$	static & off-chip mem.	$\approx 2N$	13a)
$> \frac{1}{4}Bk\eta$	partial reonf.	$> 2N$	14a)

per memory block. As we do not demand special initial values for the configuration of the memory blocks, their content is excluded from the configuration. Furthermore, the maximum specified configuration speed is with 400 MB/s five time slower than the expected throughput of the sorter hardware. See Section 4.3 for the exact throughput measures for the configuration speed and the sorter modules.

With this information, we can state the reconfiguration time overhead by means of sort processing time. Given the 100 memory block example, a single configuration would take as long as sorting $5 \cdot 3$ MB= 15 MB within a single run, because of the five times lower configuration throughput. By multiplying this value with the number of configuration swaps and dividing the result by the pure processing time, the relative overhead can be computed. Considering the example on the Xilinx Virtex-5 FPGA with $B = 100$ memory blocks and one memory channel, the problem size that can be solved within one run of both the FIFO-bsed merge sorter and the tree-based merge sorter is: $B^2k\eta$ (see Figure 14a)). Given that a single memory block provides $k = 4$ KB capacity and that the FIFO implementation efficiency is $\eta = 80\%$, this sorter can process up to 32 MB in two runs. This data will be transferred two times through the FPGA, once when receiving the sort keys (and when simultaneously running the FIFO-based merge sorter) and once more for writing the result back directly from the tree-based merge sorter. Consequently, for this example, the configuration overhead is roughly an additional $\frac{15}{2 \cdot 32}$=23%. Solving the identical problem on the same FPGA in a fully static implementation (see Figure 13a)) would require an additional run through the tree-based merge sorter and results in a respective time overhead of an additional 50%.

This does not mean that the partial version will be always outperform the static only implementation. Partial reconfiguration basically permits to adjust the resource allocation for adapting the sorter hardware to varying problem sizes as listed in Table 3.

3.5.2 Reconfiguration with Additional I/O Capacity

In Section 3.4, we have presented how an additional memory channel permits to run a tree merge step in parallel to any other sorting step. This raises the problem size to higher power. By utilizing partial reconfiguration for swapping from the FIFO-based merge sorter to the tree-based merge sorter, as shown in Figure 14c), two times two parallel runs can be performed to solve problems up to $N_2^R = \frac{1}{16}B^4k\eta$. Given the example with $B = 100$ available memory blocks on a Xilinx Virtex-5 FPGA results in sortable problem sizes up to 20 GB.

This requires streaming the key data four times through the FPGA, while two streams can be processed in parallel (with a small delay to generate sufficient fully sorted blocks for starting the next tree-based sorter). Given a host interface with 2 GB/s and assuming a sufficient fast sorter and memory, the whole sorting process takes theoretically only slightly more than 20 seconds. Meaning that the sort throughput is roughly 1 GB/s up to the problem size of 20 GB. This requires a corresponding amount of on-board memory but comprises only moderate demands on the logic resources and the I/O throughput. Note that the reconfiguration involves only half the amount of resources that have been allocated for the sorting hardware, because only one sorter module is being swapped while keeping one tree-based merge sorter statically in the system. For large sorting problems, the configuration overhead can be ignored.

The introduction of partial run-time reconfiguration permits to sort much larger sequences at a throughput of 1 GB/s than it would be possible in a static only system. Given the two sorting systems shown in Figure 13b) and Figure 14c) providing two memory channels each, the extension λ is:

$$\lambda = \frac{N_2^R}{N_2^S} = \frac{\frac{1}{16}B^4k\eta}{\frac{4}{27}B^3k\eta} = \frac{27}{64}B \qquad (7)$$

For the example, this results in a theoretical $\lambda_{(B=100)} = 42$ times longer sequence that can be sorted by the reconfigurable sorter at 1 GB/s, while the static implementation would only achieve 666 MB/s because of the additional run for the final merge step. Figure 15 denotes the number of BRAMs required to solve a particular problem size at 1 GB/s using any of the proposed sorter architectures in this paper.

3.5.3 Discussion on Using Reconfiguration

As the amount of BRAM is the main limiting FPGA resource, someone can think to multiplex memory blocks between sorter architectures, instead of using partial run-time reconfiguration. This is virtually impossible as the FIFO-based merge sorter requires memory block interfaces with small word sizes to implement the relatively large interfaces, while the memory blocks in the tree-based merge sorter demand bigest word sizes. Furthermore, the memory block multiplexing would add additional logic into the sorter designs and will also affect the routing that has to meet the requirements of the combined data path. Moreover, the design would be very difficult as memory block multiplexing Requires the instantiation of memory blocks at the top level of the design and not encapsulated in the design hierarchy.

An interesting further option is a hierarchical reconfiguration of the sorter with partially exchangeable comparators to adapt the system to different data types with the help of run-time reconfiguration.

4. CASE STUDY

4.1 Implementation and Design Flow

We implemented a test system for hosting all proposed single memory channel sorter architectures using the Xilinx XUPV5 Board. This board provides a XC5VLX110T-1

FPGA, a PCIe host interface, and 250 MB of DDR2-memory. We implemented the three configurations listed in Table 3 each as partially reconfigurable modules.

We haven't considered the partial design flow provided by Xilinx that interfaces reconfigurable modules via look-up table resources (which are called 'proxy logic' by Xilinx [19]). This approach comprises a latency penalty for passing the LUTs and a resource overhead of roughly 300 LUTs in our system. Moreover, the Xilinx vendor flow restricts the routing of reconfigurable modules not to cross regions outside from any module bounding box (i.e., signals of a partial module crossing the static system). This was defined as a requirement to achieve high performance also for the reconfigurable sorter modules.

To overcome these restrictions, we implemented the static system only with placement constraints such that all static resources fit into *configuration rows* A reconfiguration row contains four BRAM memory blocks or 20 configurable logic blocks (CLBs) that contain a cluster of eight 6-input LUTs each. Such a configuration row is the smallest piece of configuration logic that can be partially updated. By separating the static logic and memory resources into strict separated rows, it is ensured that the partial reconfiguration process is not corrupting any logic state. Note that both the static design and the reconfigurable sorter modules make heavily use out of distributed memory, meaning that look-up tables are used as shift registers that would be corrupted if logic is updated within the same configuration row of the distributed memory primitive. We haven't defined restrictions on the routing resources as the routing can be updated independent from the logic, because the configuration rows are written in multiple data items (called *frames*) and there exist separate frames for routing and logic.

If a routed net exists in all partial configuration bitstreams, it will not glitch during the configuration process. In order to guarantee this for the static routing, we converted the static design into a *hard macro* that is instantiated during the implementation of a sorter module. Hard macros are modules that can contain any kind of logic or routing. The routing of a macro can be preserved, hence ensuring for our static design that always identical routing resources will be used. For activating all clock drivers in the reconfigurable region, we instantiated dummy primitives (BRAMs and LUTs) and connected them to the global clock net. After this, we generated the configuration bitstream for the static design. Next we removed the dummy logic and converted the design into a hard macro. In this hard macro, we labeled specific primitive pins as I/O pins. These pins correspond to the top-level entities in the HDL-code of the sorting modules.

4.2 Demo System

The demo system provides a Microblaze soft-core processor connected to a multi-port DDR-2 memory controller module (MPMC) that is provided by Xilinx. The processor controls the operation of the system and permits to verify or initialize the memory. A PCIe core was integrated as the host interface. For ensuring fast FPGA reconfiguration, one port of the MPMC has been connected to the internal configuration access port ICAP. At system start, the processor is used to cache all partial bitfiles in the DDR memory. The partial reconfiguration process itself is then a DMA transfer of the corresponding configuration bitstream to the ICAP port. By running the configuration port at 125 MHz, we configured the device faster than specified, what has been studied in [5].

The sorter modules communicate via command FIFOs with the static system. This was in particular used to decouple the clock of the CPU/memory sub-system from the sorting accelerators.

A weak point of the system is the poor I/O performance of the XUPV5 system based on the MPMC memory core and the single lane PCIe interface. For testing our sorters at full speed, we added I/O emulation modules into the static system. PCIe read access was emulated by supplying random data (from a LFSR), while memory read data was generated by a counter for the upper half of the key data and a random data generator for the lower half. This emulates the reading of pre-sorted blocks as they would have been generated by a FIFO-based merge sorter. For all write operations, the emulators verified an ascending order of the keys. All data paths and the keys have been set to be 64-bit wide.

4.3 Experimental Results

When setting the target clock frequency to 250 MHz for the sorter modules, we achieved the synthesis results listed in Table 4. The table lists also the chosen design parameters and the sort throughput for the different sorter modules. The values do not include the configuration overhead, that is 27 ms for 3.12 MB configuration data. Given a 4 M key sorting problem (which is 32 MB data), the configuration overhead is considerable $(1 + \frac{27ms}{32\,MB \cdot 2\,s/2\,GB})^{-1} = 46\%$ of the total time. However, given a 400 M key problem (which is 3.2 GB data), the configuration overhead can be neglected. Note that partial reconfiguration is the only option to sort such large problems in only three runs on the XUPV5 board (assuming a sufficient large memory module). If implementing the sorter with less resources, partial run-time reconfiguration becomes much more benefitial also for the small sorting example. Assuming an implementation with one half the logic, the reconfiguration overhead is only 22%.

For rating our approach, we compared it with state-of-the-art alternatives. A Cell processor implementation of SIMD bitonic sorting reports a maximum problem size of 128 KB (32-bit keys) that is being processed on one PPE SIMD unit in 2.5 ms [6]. This results in a sorting throughput of 51.2 MB/s or $\frac{1}{39}$ times the throughput of the FIFO-based merge sorter. Consequently, even utilizing all 16 available PPEs of the two Cell processors within the used Cell-Blade system cannot compete with the FPGA solution.

The best published result on sorting using a GPU reports a speed of 178 M keys/s, which is 720 MB/s [10]. But it must be mentioned that this performance is available for up to 16 M keys or (64 MB key storage) where we require three times of streaming the problem through the FPGA, what is then slightly slower. However, by adding a second memory port to the system and using partial run-time reconfiguration, the FPGA will outperform the GPU in speed. This would allow 1GB/s on problems beeing two orders of magnitude larger than the 35 MB result. Note that the required memory throughput of that FPGA solution is only about 6.2% of the memory throughput that is available on the GeForce GTX260 that was used in [10].

[6] lists results for a Quicksort reference implementation running on a 2-core 3.2 GHz Xeon system. For 1M keys (32 bit), the throughput is only 40.4 MB/s (31.7 MB/s for 128M keys) which is almost two orders of magnitude slower than our FPGA solution.

To the best of our knowledge, there is no published FPGA implementation that would allow a direct comparison. Most related FPGA implementations have been done on relatively small sorting problems, e.g. for median filtering in video processing systems and are also not designed for a throughput in the GB/s domain. In a recent publication, the authors

Table 4: Throughput and physical implementation results. $\# N$ states the number of 64-bit keys.

module	clock	# slices	# BRAMs	B	$\# N_F$ / [(KB)]	M	$\# N$ / [(KB)]	throughput	figure
FIFO-merge	252 MHz	10646 (74%)	103 (98%)	99	43 K (344 KB)		43 K (344 KB)	2 GB/s ; $N < 43\,K$	14a) left
tree-merge	273 MHz	12983 (90%)	105 (100%)	102		102	4.39 M 35.1 MB	1 GB/s ; $N < 4.4\,M$	14a) right
Fifo & tree	258 MHz	12254 (85%)	105 (100%)	100	21.5 K (172 KB)	50	1.08 M (8.6 MB)	1 GB/s ; $N < 2.1\,M$	13a)
one run of the FIFO merge sorter and two runs of the tree merge sorter							448 M 3.58 GB	667 MB/s ; $N < 3.5\,G$	

reported a throughput of less than 100 MB/s when fully utilizing a XC2VP30 FPGA on a problem size of 256 MB [14]. However, the chosen XC2VP30 device provides sufficient on-chip memory for solving a 250 MB problem in three runs, when using our sorting architecture. We estimate that our sorter hardware will work with 200 MHz on that device. We can summarize, that our FPGA solution surpasses any GPU, Cell processor or existing FPGA solution in both problem size and throughput, when considering an architecture with two memory channels.

5. CONCLUSIONS

In this paper, we carefully analyzed existing sorting architectures and tuned them into a highly optimized implementation that outperforms the best published results on the Cell processor and GPUs. The system is highly scalable and has the potential to dramatically further improve by introducing second memory channel. Moreover, this paper demonstrated considerable rises in performance as well as in resource efficiency by introducing partial run-time reconfiguration. It must be mentioned that the results have been achieved for the sort kernels but not for a complete system, as the available XUPV5 board provides only a single lane PCIe host interface. However, there exist commercial as well as academic systems that would fulfillour I/O requirements (e.g., the BEE3 system [4]).

Future work will demonstrate the performance with a real host interface and two memory channels to prove our theoretical observations by experiments. A further topic that will be investigated is a hierarchical reconfiguration for only swapping comparator functions.

Acknowledgment

This work is supported in part by the Norwegian Research Council under grant 191156V30

6. REFERENCES

[1] P. Alfke. Take Advantage of Leftover Multipliers and Block RAMs. *Xcell Journal*, 2:48–49, 2001.

[2] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference (AFIPS 68)*, pages 307–314. ACM, 1968.

[3] M. Bednara, O. Beyer, J. Teich, and R. Wanka. Tradeoff Analysis and Architecture Design of a Hybrid Hardware/Software Sorter. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 299–308. IEEE Computer Society, 2000.

[4] Berkeley Wireless Research Center. BEEcube Homepage, 2010. `http://www.beecube.com/platform.html`.

[5] C. Claus, R. Ahmed, F. Altenried, and W. Stechele. Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems. In *Reconfigurable Computing: Architectures, Tools and Applications (ARCS)*, volume 5992 of *LNCS*, pages 55–67. Springer, 2010.

[6] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: High Performance Sorting on the Cell Processor. In *Proceedings of the 33rd international conference on Very large data bases (VLDB)*, pages 1286–1297. VLDB Endowment, 2007.

[7] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In *Proceedings of the ACM international conference on management of data (SIGMOD)*, pages 325–336. ACM, 2006.

[8] G. Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3):10, 2006.

[9] L. K. Ha, J. Krüger, and C. T. Silva. Fast Four-Way Parallel Radix Sorting on GPUs. *Comput. Graph. Forum*, 28(8):2368–2378, 2009.

[10] J. K. L. Ha and C. Silva. Implicit radix sorting on GPUs, 2010. GPU GEMS volume 2, to appear, `www.sci.utah.edu/ csilva/papers/ImplSorting.pdf`.

[11] C. Layer and H.-J. Pfleiderer. A Reconfigurable Recurrent Bitonic Sorting Network for Concurrently Accessible Data. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 648–657, 2004.

[12] R. Marcelino, H. Neto, and J. Cardoso. Sorting Units for FPGA-Based Embedded Systems. In *Distributed Embedded Systems: Design, Middleware and Resources*, volume 271 of *IFIP International Federation for Information Processing*, pages 11–22. Springer Boston, 2008.

[13] R. Marcelino, H. Neto, and J. Cardoso. Unbalanced FIFO Sorting for FPGA-Based Systems. In *16th IEEE International Conference on Electronics, Circuits, and Systems, (ICECS)*, pages 431–434, dec 2009.

[14] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, 2009.

[15] Y. Seddiq, S. Alshebeili, S. Alhumaidi, and A. Obied. FPGA-Based Implementation of a CFAR Processor Using Batcher's Sort and LUT Arithmetic. In *Design and Test Workshop (IDT), 2009 4th International*, pages 1 –6, nov. 2009.

[16] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007.

[17] The Unicode Consortium. About the Unicode Standard, 2010. `http://www.unicode.org`.

[18] S. Wong, S. Vassiliadis, and J. Hur. Parallel Merge Sort on a Binary Tree On-Chip Network. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, pages 365–368, November 2005.

[19] Xilinx Inc. Partial Reconfiguration User Guide, May 2010. Rel 12.1.