The VLDB Journal manuscript No.
(will be inserted by the editor)

Sorting Networks on FPGAs

Rene Mueller · Jens Teubner · Gustavo Alonso

Received: date / Accepted: date

Abstract Computer architectures are quickly changing toward heterogeneous many-core systems. Such a trend opens up interesting opportunities but also raises immense challenges since the efficient use of heterogeneous many-core systems is not a trivial problem. Software-configurable microprocessors and FPGAs add further diversity but also increase complexity. In this paper, we explore the use of sorting networks on field-programmable gate arrays (FPGAs). FPGAs are very versatile in terms of how they can be used and can also be added as additional processing units in standard CPU sockets. Our results indicate that efficient usage of FPGAs involves non-trivial aspects such as having the right computation model (a sorting network in this case); a careful implementation that balances all the design constraints in an FPGA; and the proper integration strategy to link the FPGA to the rest of the system. Once these issues are properly addressed, our experiments show that FPGAs exhibit performance figures competitive with those of modern general-purpose CPUs while offering significant advantages in terms of power consumption and parallel stream evaluation.

Keywords Sorting Networks · FPGA · Hardware Accelerators

1 Introduction

Taking advantage of specialized hardware has a long tradition in data processing. Some of the earliest efforts involved building entire machines tailored to database engines [9]. More recently, graphic processors (GPUs) have been used to efficiently implement certain types of operators [13; 14].

Parallel to these developments, computer architectures are quickly evolving toward heterogeneous many-core systems. These systems will soon have a (large) number of processors [18] and the processors will not be identical. Some will have full instruction sets, others will have reduced or specialized instruction sets; they may use different clock frequencies or exhibit different power consumption; floating point arithmetic-logic units will not be present in all processors; and there will be highly specialized cores such as *field-programmable gate arrays* (FPGAs) [15; 24]. An example of such a heterogeneous system is the Cell Broadband Engine, which contains, in addition to a general-purpose core, multiple special execution cores (synergistic processing elements, or SPEs).

In this paper, we focus our attention on FPGAs as one of the more *different* elements that can be found in many-core systems. FPGAs are (re-)programmable hardware that can be tailored to almost any application. However, it is as yet unclear how the potential of FPGAs can be efficiently exploited. Our contribution with this work is to first study the design trade-offs encountered when using FPGAs for data processing tasks. In particular, we look at sorting networks that are well suited for an implementation in hardware. Second, we provide a set of guidelines for how to make design choices such as:

(1) FPGAs have relatively low clock frequencies. Naïve designs will exhibit a large latency and low throughput. We show how this can be avoided by a careful design using *synchronous* and *asynchronous* cir-

R. Mueller
Systems Group, Department of Computer Science
ETH Zurich, Switzerland
E-mail: rene.mueller@inf.ethz.ch
http://www.systems.ethz.ch/

cuits. While the former increase throughput the latter reduce latency.

(2) Asynchronous circuits are notoriously more difficult to design than synchronous ones. This has led to a preference for synchronous circuits in studies of FPGA usage [15]. Using the example of *sorting networks*, we illustrate systematic design guidelines to create asynchronous circuits that solve database problems.

(3) FPGAs provide inherent *parallelism* whose only limitation is the amount of *chip space* to accommodate parallel functionality. We show how this can be managed and present how the chip space consumption of different implementation can be estimated.

(4) FPGAs can be very useful as database co-processors attached to an engine running on conventional CPUs. This *integration* is not trivial and opens up several questions on how an FPGA can fit into the complete architecture. In two use-cases, we demonstrate an embedded heterogeneous multi-core setup. In the first use-case we connect the custom logic over a bus to an embedded CPU. The second uses a tighter integration to the CPU, implemented though a direct connection to the CPU's execution pipeline. For both approaches study the trade-offs in FPGA integration design.

(5) FPGAs are attractive co-processors because of the potential for tailored design and parallelism. We show that FPGAs are also very interesting in regard to *power consumption* as they consume significantly less power, yet provide at a performance comparable to the one of conventional CPUs. This makes FPGAs good candidates for multi-core systems as cores where certain data processing tasks can be offloaded.

To illustrate the trade-offs for system integration we present two applications scenarios that are based on sorting networks. The first is the implementation of a *median operator*. In the second use-case we evaluate a hardware/software co-design on a FPGA. A 8-element sorting co-processor is implemented in the FPGA logic and combined with a merge sort algorithm running on the embedded CPU. Through an extension of the CPU's instruction set we show how the FPGA accelerator can be used in heterogeneous setup together with existing CPU code. Our experiments show that FPGAs can clearly be a useful component of a modern data processing system, especially in the context of multi-core architectures.

Outline. We start our work by setting the context with related work (Section 2). After introducing the necessary technical background on FPGAs in Section 3 and sorting networks in Section 4, we show how to implement sorting networks on an FPGA (Section 5). We evaluate several implementations of different sorting networks in Section 6. While this allows an in-depth analysis of FPGA-specific implementation aspects it does not provide any insight of how the FPGA behaves in a complete system. We make up for that in Sections 7 and 8 where we illustrate two complete use-cases. In Section 7 we illustrate the implementation of a median operator using FPGA hardware (Section 7). The second use-case (Section 8) consists of a sorting co-processor that is directly connected to the execution pipeline of the embedded PowerPC CPU. We finally wrap up in Section 9.

2 Related Work

A number of research efforts have explored how databases can use the potential of modern hardware architectures. Examples include optimizations for cache efficiency (e.g., [23]) or the use of vector primitives ("SIMD instructions") in database algorithms [37]. The QPipe [17] engine exploits multi-core functionality by building an operator pipeline over multiple CPU cores. Likewise, stream processors such as Aurora [1] or Borealis [2] are implemented as networks of stream operators. An FPGA with database functionality could directly be plugged into such systems to act as a node of the operator network.

The shift toward an increasing heterogeneity is already visible in terms of tailor-made graphics or network CPUs, which have found their way into commodity systems. Govindaraju *et al.* demonstrated how the parallelism built into graphics processing units can be used to accelerate common database tasks, such as the evaluation of predicates and aggregates [13].

GPUTeraSort [14] parallelizes a sorting problem over multiple hardware shading units on the GPU. Within each unit, it achieves parallelization by using SIMD operations on the GPU processors. The AA-Sort [20], CellSort [11], and MergeSort [7] algorithms are very similar in nature, but target the SIMD instruction sets of the PowerPC 970MP, Cell, and Intel Core 2 Quad processors, respectively.

The use of network processors for database processing was studied by Gold *et al.* [12]. The particular benefit of such processors for database processing is their enhanced support for multi-threading. We share our view on the role of FPGAs in upcoming system architectures with projects such as Kiwi [15] or Liquid Metal [19]. Both projects aim at off-loading traditional CPU tasks to programmable hardware. Mitra *et al.* [24] recently outlined how FPGAs can be used as co-processors in

an SGI Altix supercomputer to accelerate XML filtering.

The advantage of using customized hardware as a database co-processor is well known since many years. For instance, DeWitt's DIRECT system comprises of a number of query processors whose instruction sets embrace common database tasks such as join or aggregate operators [9]. Similar ideas have been commercialized recently in terms of database appliances sold by, e.g., Netezza [28], Kickfire [21], or XtremeData [36]. All of them appear to be based on specialized, hard-wired acceleration chips, which primarily provide a high degree of data parallelism. Our approach can be used to exploit the *reconfigurability* of FPGAs at runtime. By reprogramming the chip for individual workloads or queries, we can achieve higher resource utilization and implement data *and* task parallelism. By studying the foundations of FPGA-assisted database processing in detail, this work is an important step toward our goal of building such a system.

FPGAs are being successfully applied in signal processing, and we draw on some of that work in Sections 7. The particular operator that we use in Section 7 is a median over a sliding window. The implementation of a median with FPGAs has already been studied [33], but only on smaller values than the 32 bit integers considered in this paper. Our median implementation is similar to the sorting network proposed by Oflazer [29].

An early version of this work was published in [26]. This article contains significant contributions beyond [26]. Most importantly, a general discussion of even-odd merging and bitonic sorting networks was added. Furthermore, a detailed model describing the chip area utilization of the different designs is presented and evaluated. In this article, all implementations and measurements are based on a new Virtex-5 FPGA chip instead of an older Virtex-II Pro chip. Next to the median operator which served as an example use-case in [26] another use-case is added that illustrates the how sorting on a CPU can be accelerated. This further illustrates how an FPGA-based sorting co-processor can be directly connected to the execution pipeline of a CPU.

3 Overview of FPGAs

Field-programmable gate arrays are re-programmable hardware chips for digital logic. FPGAs are an array of logic gates that can be configured to construct arbitrary digital circuits. These circuits are specified using either circuit schematics or hardware description languages such as Verilog or VHDL. A logic design on an FPGA is also referred to as a *soft IP-core* (intellectual property core). Existing commercial libraries provide a wide range of pre-designed cores, including those of complete CPUs. More than one soft IP-core can be placed onto an FPGA chip.

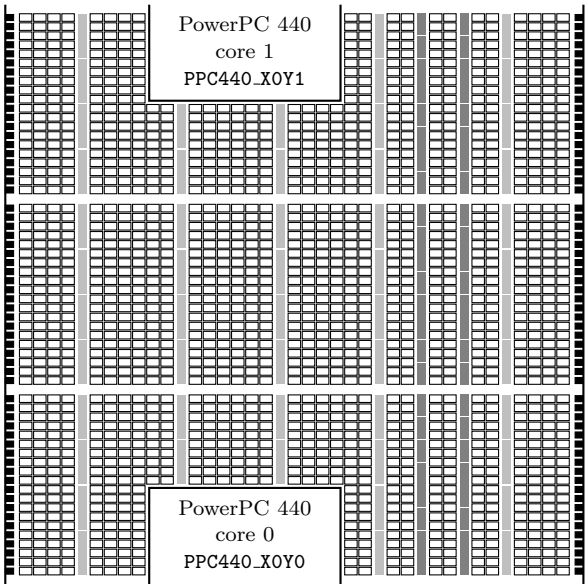


Fig. 1 Simplified FPGA architecture: 2D array of CLBs ▭ each consisting of 2 slices. IOBs ▪ connect the FPGA fabric to the pins of the chip. Additionally available in silicon are: two PowerPC cores, BRAM blocks ▮ and multipliers ▮.

PowerPC cores	2
Slices	20,480
36 kbit BRAM blocks	298 (=10,728 kbit)
25×18-bit multipliers	320
I/O pins	840

Table 1 Characteristics of Xilinx XC5VFX130T FPGA.

3.1 FPGA Architecture

Figure 1 sketches the architecture of the Xilinx Virtex-5 FX130T FPGA used in this paper [34; 35]. The FPGA is a 2D array of *configurable logic blocks* (CLBs). Each logic block consists of 2 *slices* that contain logic gates (in terms of lookup tables, see below) and a switch box that connects slices to an FPGA *interconnect fabric*.

In addition to the CLBs, FPGA manufacturers provide frequently-used functionality as discrete silicon components (*hard IP-cores*). Such hard IP-cores include *block RAM* (BRAM) elements, each containing 36 kbit fast storage, as well as 25×18-bit multiplier units.

A number of *Input/Output Blocks* (IOBs) link to pins, e.g., used to connect the chip to external RAM or networking devices. Two on-chip PowerPC 440 cores

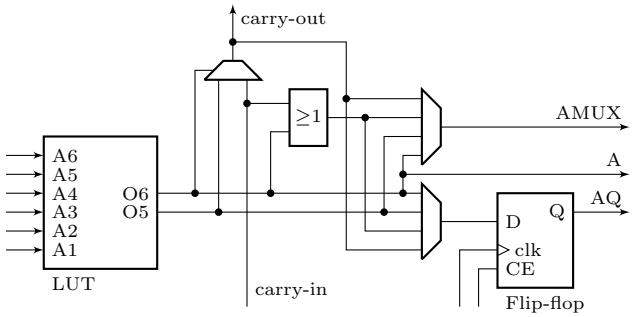


Fig. 2 Simplified LUT–Flip-flop combination of a Virtex-5 slice. A slice contains four of these structures.

are directly wired to the FPGA fabric and to the BRAM components. Table 1 shows a summary of the characteristics of the FPGA used in this paper. Each PowerPC core has dedicated 32 kB data and instruction caches. The caches have similar latency as the BRAM memory and are intended to speed-up accesses to external memory with longer latency. The superscalar cores implement the 32-bit fixed-point subset of the PowerPC architecture. The embedded PowerPC 440 cores are also used in the IBM Blue Gene/L supercomputer where they perform all non-floating point calculations.

Configurable Logic Blocks (CLBs) are further split into *slices*. On the Virtex-5 each CLB is made up of two slices. Each slice contains four *lookup tables* (LUTs) and four *Flip-flops*. Figure 2 depicts one of the four LUT–Flip-flop configurations a Virtex-5 slice. LUTs can implement arbitrary Boolean-valued functions that can have up to six independent Boolean arguments. Traditionally, a LUT has one output. On the Virtex-5 a LUT has two outputs (identified as O5 and O6 in Figure 2). A Virtex-5 LUT either implements a single function on output O6 that uses up to six inputs or, alternatively, two functions on O5 and O6 that in total use five inputs. The outputs O5 and O6 are fed to two multiplexers that configure which signals are fed to the output of the slice and to the flip-flop. The flip-flop acts as a register that can store one single bit. The design of a slice provides dedicated elements for carry logic that allow for efficient implementation of, e.g., adders and comparators. The carry logic connects the LUTs inside a slice and different slices in an FPGA column.

Certain LUTs on the Virtex-5 can also be used as a 16- or 32-element shift register or as 32×1-bit or 64×1-bit RAM cells. Memory instantiated through LUTs configured as RAM cells is referred to as *distributed memory*. In contrast to the aforementioned block RAM (BRAM) distributed memory can be instantiated on finer scale, however, at a significantly lower density.

3.2 Hardware Setup

FPGAs are typically available pre-mounted on a circuit board that includes additional peripherals. Such circuit boards provide an ideal basis for the assessment we perform here. Quantitative statements in this report are based on a Xilinx ML510 development board with a Virtex-5 FX130T FPGA chip. Relevant for the discussion in this paper are the DDR2 DIMM socket which we populated with two 512 MB RAM modules. For terminal I/O of the software running on the PowerPC, a RS-232 UART interface is available. The board also includes a gigabit Ethernet port.

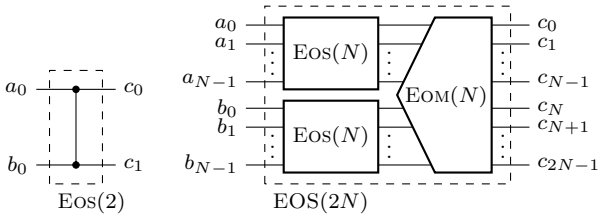
The board is clocked at 100 MHz. From this external clock additional clocks are generated for the various clock regions on the chip and for the external external I/O connectors, such as the DDR RAM. The PowerPC cores are clocked at 400 MHz.

4 Sorting Networks

Some of the most efficient traditional approaches to sorting are also the best options in the context of FPGAs. *Sorting networks* are attractive in both scenarios, because they *(i)* do not require *control flow* instructions or branches and *(ii)* are straightforward to *parallelize* (because of their simple data flow pattern). Sorting networks are suitable for relatively short sequences whose length is known a priori. Sorting networks have been extensively studied in literature. For a detailed treatment see [5; 8; 22]. On modern CPUs, sorting networks suggest the use of vector primitives, which has been demonstrated in [11; 14; 20].

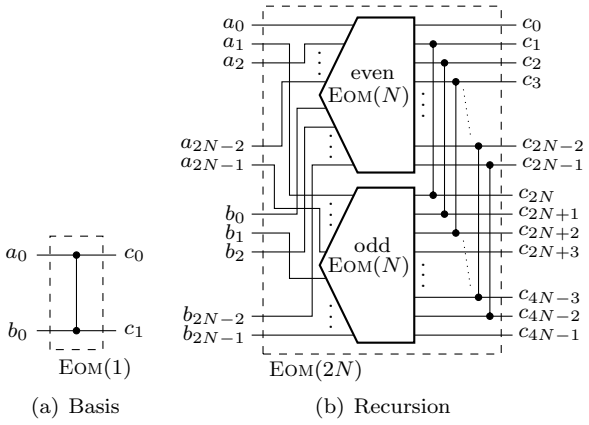
The circuits of the sorting networks are composed of horizontal wire segments and vertical *comparators*. We represent the comparator elements using the widely known a Knuth notation ⁞. The unsorted elements are applied at the left, one element per wire (Figure 8). The sorted output then appears on the right side. Wire segments connect the different compare-and-swap stages. Each wire segment can transfer an m-bit number. The comparator elements perform a two-element sort, such that the smaller of the two input values leaves the element on the right side through the upper output wire and the larger value through the lower wire.

In the following, we describe two systematic methods to build sorting networks. The first method is based on *Even-odd Merging networks*, the second on *Bitonic Merging network*. Both were proposed by K. E. Batcher in [5].



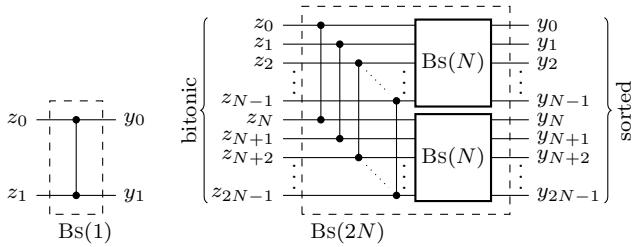
(a) Basis (b) Recursion

Fig. 3 Recursive definition of even-odd sorter EOS($2N$).



(a) Basis (b) Recursion

Fig. 4 Recursive definition of Even-odd merger EOM(N) for $2 \times N$ elements.



(a) Basis (b) Recursion

Fig. 5 Bitonic Sorters produce a sorted output from a bitonic input sequence.

4.1 Even-odd Merging Networks

Even-odd merging networks are built following a recursive definition that is assumed to be efficient when number of elements $N = 2^p$ is a power of two [22]. In this paper, we use the exponent p to describe the size of a network. At the heart of the networks are even-odd merging elements that combine two sorted subsequences $a_0 \leq \cdots \leq a_{N-1}$ and $b_0 \leq \cdots \leq b_{N-1}$ into a single sorted sequence $c_0 \leq \cdots \leq c_{N-1} \leq c_N \leq \cdots \leq c_{2N-1}$. Using these merging elements a sorting network can be built recursively as shown in Figure 3. The input sequence of size $2N$ is split into two sequences of size N. Each of these sequences is sorted by an even-odd sorter EOS(N). The sorted outputs are then merged using an even-odd merger EOM(N) of size N. The recursive definition of EOS(N) is depicted in Figure 3(b). Figure 3(a) shows the basis of the recursion where a single comparator is used to sort the two elements.

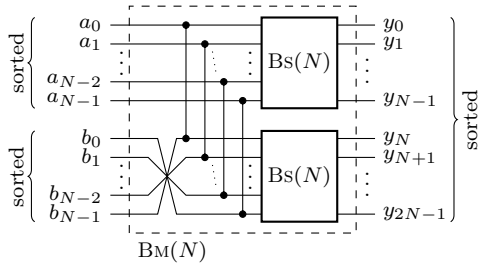
The even-odd merging elements EOM(N) that combine the two sorted sequences of length N are defined in a similar recursive form. The basis of the recursion EOM(1) is a single comparator as shown in Figure 4(a). The recursion step illustrated in Figure 4(b) works as follows: Given are two sorted input sequences $a_0, a_1, \ldots, a_{2N-1}$ and $b_0, b_1, \ldots, b_{2N-1}$ for an even-odd merger EOM($2N$). The N even-indexed elements $a_0, a_2, \ldots, a_{2k}, \ldots, a_{2N-2}$ are mapped to the a-inputs of the "even" merger. The N odd-indexed elements $a_1, a_3, \ldots, a_{2k+1}, \ldots, a_{2N-1}$ are mapped to the a-inputs of the "odd" merger. The b inputs are routed similarly. As it can be easily shown the inputs of the even and odd mergers are sorted, hence, each produces a sorted sequence at the output. The two sequences are then combined by an array of $2N - 1$ comparators as shown in Figure 4(b). By unrolling the recursion of EOS(N) and EOM(N) a sorting network consisting of comparators is created. An example of an even-odd merging network that is able to sort eight inputs is shown in Figure 8(a).

4.2 Bitonic Merging Networks

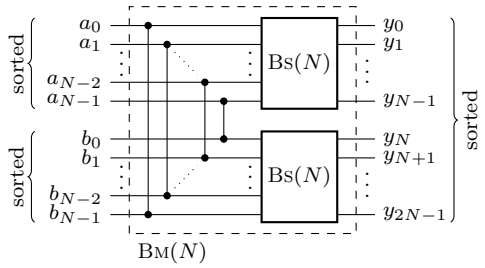
In CellSort [11] and GPUTeraSort [14] sorting is based on *Bitonic Merging networks*. A bitonic sequence can be regarded as a partially sorted list which consists of two sorted monotonic subsequences, one ascending the other descending. For example, $1, 4, 6, 5, 2$ is a bitonic sequence whereas $1, 4, 6, 5, 7$ is not.

A bitonic sequence can be transformed into a sorted monotonic sequence using a *Bitonic Sorter* [5]. Figure 5 shows the recursive definition of a bitonic sorter Bs($2N$) that transforms the $2N$ bitonic input sequence $z_0, z_1, \ldots, z_{2N-1}$ into a sorted output sequence.

The reason for introducing bitonic sequences in the first place is that they can be easily generated from two two sorted sequences $a_0, \ldots, a_{N-1}$ and $b_0, \ldots, b_{N-1}$. It can be shown that concatenating $a_0, \ldots, a_{N-1}$ and the sequence $b_{N-1}, \ldots, b_0$, i.e., the a sequence with the reversed b sequence, yields a bitonic sequence. This bitonic sequence can then be sorted by a bitonic sorter Bs($2N$). This process generates a network that merges two sorted input sequences of length N. The resulting *Bitonic Merging* network is shown in Figure 6(a). Using the fact that reversing a bitonic circuit is also bitonic the circuit can be redrawn without wire-crossings in Figure 6(b).

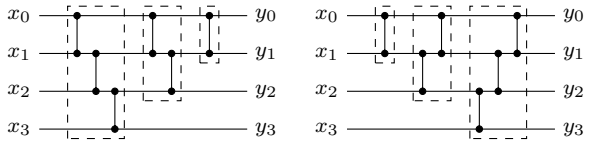


(a) Merge network using two Bitonic Sorters and flipping the b sequence.



(b) Equivalent circuit without wire-crossings.

Fig. 6 Sorting network as recursive definition of a merging network based on Bitonic Sorters.



(a) Bubble Sort (b) Insertion Sort

Fig. 7 Sorting networks based on Insertion Sort and Bubble Sort are equivalent.

Following the divide-and-conquer approach bitonic merger in Figure 6 can be recursively applied producing a complete sorting network. Such a network is shown in Figure 8(b) for size $N = 8$.

4.3 Bubble and Insertion Sort Networks

Sorting networks can also be generated from traditional sorting algorithms. Figure 7 shows two networks that are generated from Bubble and Insertion Sort. When comparing the two circuits diagrams 7(a) and 7(b) it can be seen that the resulting networks are structurally equivalent. Like their algorithmic counterparts these sorting networks are inefficient. Networks generated by this approach require many comparator elements.

4.4 Sorting Networks Comparison

In general, the efficiency of a sorting network can be measured in the number of comparators required and the number of stages, i.e., steps, through the sorting network. Table 2 shows the resulting number of comparators $C(N)$ of a sorting network of size N and $S(N)$ the number of stages.

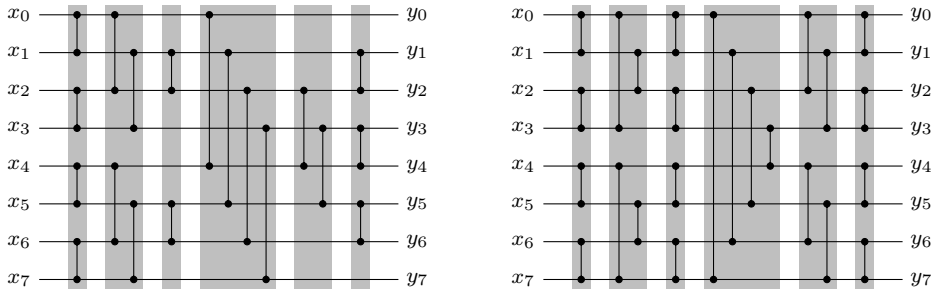
Bitonic Merge and Even-odd Merge sorters have the same depth and the same asymptotic complexity. However, asymptotic behavior is of little interest here, as we are dealing with relatively small array sizes. An interesting result was published in [3] that proposes a sorting network with a better asymptotic complexity $C(N) = O\big(N \ln(N)\big)$ and depth $S(N) = O\big(\ln(N)\big)$. However, the constant dropped in the O-notation is too big and thus renders it unsuitable for practical sorting networks [22].

Despite requiring more comparators bitonic merge sorters are frequently used because they have two important properties: (1) all signal paths have the same length and (2) the number of concurrent compares for each stage is constant. For example, in the Bitonic Merging network in Figure 8(b) every wire undergoes six compares. In contrast, consider the the uppermost wire of the even-odd merger sorter in Figure 8(a). The path $x_0 \rightsquigarrow y_0$ passes only through three comparator stages, whereas $x_2 \rightsquigarrow y_2$ passes through all 6 stages. This has the disadvantage that different signal lengths must be considered, for example, if the data is clocked through the network, i.e., one stage every clock, additional registers may be necessary of buffering intermediate values.

In a bitonic merging network, $N/2$ compares are present in each stage. For even-odd mergers the number is not constant. In Figure 8(b) for example, stage 1 has 4 concurrent compares, whereas stage 3 has only 2. A constant number of comparisons is useful to efficiently implement the sorting network in a sequential form using a fixed number of comparators M given by the architecture, e.g., the SIMD vector length. A single stage can be executed in $N/(2M)$ steps. If a stage is not using all operators, some comparators remain idle. This fact is exploited in [11; 14].

5 Implementing Sorting Networks

The sorting networks shown in Section 4 can be implemented in several ways using different technologies. In this context we study two implementations in FPGA hardware and a CPU-based implementation. For the hardware variant we differentiate between three types of circuits: *asynchronous* and a *synchronous* and *pipelined* implementations. Before diving into the implementation details we first discuss the key properties of these circuit types.

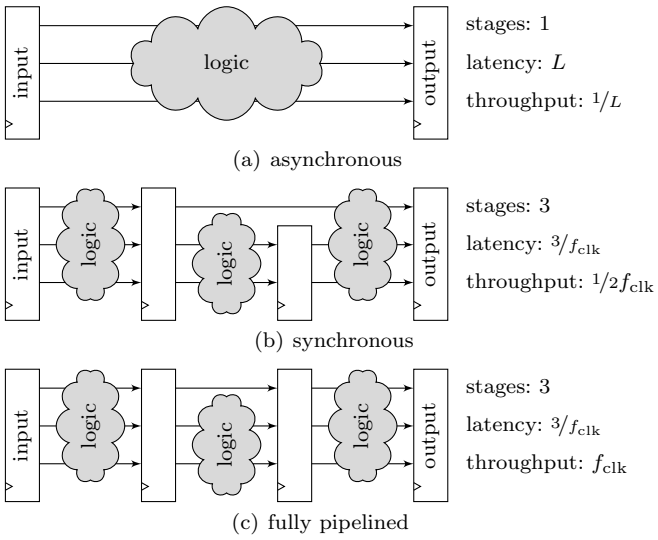


(a) Even-odd Merging Network (b) Bitonic Merging Network

Fig. 8 Sorting networks for 8 elements.

	bubble/insertion	even-odd merge	bitonic merge
exact	$C(N) = \frac{N(N-1)}{2}$ $S(N) = 2N - 3$	$C(2^p) = (p^2 - p + 4)2^{p-2} - 1$ $S(2^p) = \frac{p(p+1)}{2}$	$C(2^p) = (p^2 + p)2^{p-2}$ $S(2^p) = \frac{p(p+1)}{2}$
asymptotic	$C(N) = O(N^2)$ $S(N) = O(N)$	$C(N) = O\left(N \log^2(N)\right)$ $S(N) = O\left(\log^2(N)\right)$	$C(N) = O\left(N \log^2(N)\right)$ $S(N) = O\left(\log^2(N)\right)$
$N = 8$	$C(8) = 28$ $S(8) = 13$	$C(8) = 19$ $S(8) = 6$	$C(8) = 24$ $S(8) = 6$

Table 2 Comparator count $C(N)$ and depth $S(N)$ of different sorting networks of size N. For sizes $N = 2^p$ the exponent p is given.



(a) asynchronous
(b) synchronous
(c) fully pipelined

Fig. 9 Implementation approaches for digital circuits.

5.1 Asynchronous vs. Synchronous Pipelined Circuits

Asynchronous circuits are operated without a clock signal. They consist of *combinatorial logic* only. Figure 9(a) illustrates the combinatorial logic as a cloud between a pair of registers. Input signals are applied from a register at the left. The signals travel through the combinatorial logic that comprises the sorting network. The comparators are implemented primarily using FPGA lookup tables (combinatorial logic). The signals travel through the sorting stages without following a synchronization signal such as a clock. At some predetermined time instances the signals at the output are read and stored in another register. The key characteristics of asynchronous circuits is the absence of registers.

In asynchronous implementations of sorting networks only one single N-set can reside in the network at any given time. The next N-set can only be applied after the output signals for the previous set are stored away in a register following the sorting network. Hence, both the latency and the issue interval are determined by the length of the combinatorial signal path between the input and the output of the sorting network. More precisely, they are given by the longest delay path L from the any input to any output. The throughput of the asynchronous design is $1/L$ N-sets per second. The path delay is difficult to estimate in practice, as it involves not only the delay caused by the logic gates themselves but also the signal delay in the routing fabric. The maximum delay path L directly depends on the number stages $S(N)$ but also on the number of comparators $C(N)$ as they contribute to the routing pressure and further increase latency. In Section 6 we measure the propagation delay for different network sizes. For even-odd merging network with $N = 8$ and $m = 32$ we measure $L = 18$ ns for the FX130T–2 chip. The throughput of the asynchronous design is $1/L$ N-sets per second. For the example circuit this translates into 56 M 8-sets/sec which corresponds to a processing rate of 1.66 GB/s.

By introducing registers in the combinatorial circuit the length of the signal path can be broken up.

The computation of the circuit is thereby divided into stages that are separated by registers (see Figure 9(b)). The resulting circuit is called *synchronous* since a common clock is used to move the data through the network from stage to stage at specific instants. Clearly, both types perform the same computation, hence, the combinatorial logic is conceptually identical. The crucial difference are the registers. They require additional space but have an interesting advantage.

A natural way is to place the register after each stage in the sorting network. Since the path delays between registers are smaller than the delay of the entire asynchronous network it can be clocked faster. The highest possible clock frequency is now determined by a shorter maximal path. The overall latency of the synchronous circuit is $S(N)f_{\text{clk}}$ where f_{clk} is the frequency of the clock that drivers the registers. The registers can be inserted arbitrarily in the combinatorial signal paths, not necessarily at the end of each sorting stage, allowing to trade-off the latency $S(N)f_{\text{clk}}$ with operational clock speed f_{clk}. In VLSI design this technique is known as *register balancing*. In this work, we assume that registers are added after each comparator.

Note, that by introducing stage registers alone the circuit does not necessarily become fully pipelined. In a fully pipelined circuit a new input set can be applied every clock cycle resulting in a throughput of f_{clk} N-sets per second. However, just adding a register at the output of a comparator does not necessarily make it fully pipelined. Additional register are required to buffer the value on wires that are not processed between stages. This is illustrated in Figure 9. The first wire in 9(b) is not buffered by the second register stage. It seems unnecessary as this signal is not involved in the in the combinatorial logic of the second stage. While this saves a flip-flop, now special care needs to be taken for timing the signals. In order to have the signals line up correctly at the output registers, the inputs have to be applied during two consecutive cycles. When buffering every signal as shown in 9(c) the circuit is fully pipelined, i.e., the all signal path reaching the output register have the same length and a new input can be applied at every clock cycle. This is particularly relevant for even-odd sorting networks as we will see later. The network shown in Figure 11 can be clocked at $f_{\text{clk}} = 267\,\text{MHz}$ on our chip. Being fully-pipelined, this directly translates into a data processing rate of 7.9 GB/s.

5.2 Implementing Comparators on FPGAs

The sorting network circuits shown in Section 4 can be directly translated into digital circuits. The essential component is the implementation of the comparator in FPGA logic. The sorting network can then be built by instantiating the required comparators and wiring them up accordingly.

In the following we look at how the comparator can be defined in a high-level hardware description language VHDL. Then we study how the FPGA tool chain translate this description and maps it to the FPGA primitives shown in Figures 1 and 2. This allows us to analyze the resource utilization on the chip.

Asynchronous Comparators. The complexity of the comparator is given by the width of its inputs. For this analysis we consider fixed-length m-bit integer values. The results we provide in this article are based on $m = 32$ bit integers. In general, however, any m and any comparison function can be used, e.g., double precision floats etc. In hardware, the choice of the data type only affects the implementation of the comparator not the sorting network itself. This is different from sorting network realizations on GPUs and CPUs where different types are provided in different configurations, e.g., a compare-and-swap instruction is only provided for integers but not for floating-point values.

We specify the behavior of the comparator element in the VHDL hardware description language as follows (where `<=` indicates a signal assignment):

```
entity comparator is
  port (
    a   : in  std_logic_vector(31 downto 0);
    b   : in  std_logic_vector(31 downto 0);
    min : out std_logic_vector(31 downto 0);
    max : out std_logic_vector(31 downto 0));
end comparator;
architecture behavioral of comparator is
  min <= a when a < b else b;
  max <= b when a < b else a;
end behavioral;
```

The two conditional signal assignments are *concurrent assignments*, i.e., they describe the functional relationship between the inputs and the outputs and can be thought as being executed "in parallel". The component `comparator` is instantiated once for each comparator element in the sorting network. The vendor-specific FPGA synthesis tools will then compile the VHDL code, map it to device-primitives, place the primitives on the 2D grid of the FPGA and finally compute an efficient routing of the signal between the sites on the chip.

Figure 10 shows the circuit for our Virtex-5 FPGA generated by the Xilinx ISE 11.3 tool chain. The 32 bits of the two inputs a and b are compared first (upper half of the circuit), yielding a Boolean output signal c for the outcome of the predicate $a < b$. Signal c drives

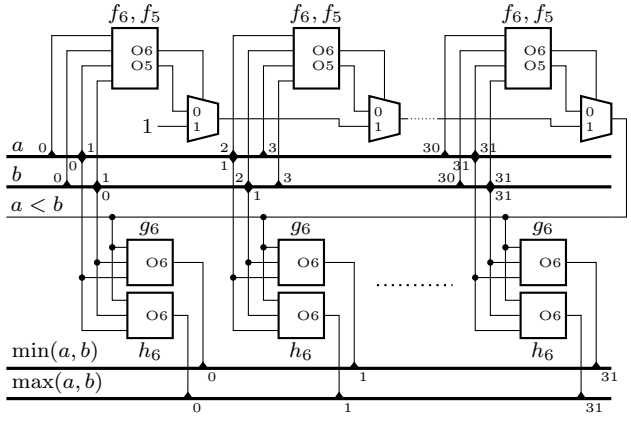


Fig. 10 FPGA implementation of an *asynchronous* 32-bit comparator requiring 80 LUTs (16 for evaluating $a < b$ and 2×32 to select the minimum/maximum values).

2×32 LUTs configured as multiplexers that connect the proper input lines to the output lines for $\min(a, b)$ and $\max(a, b)$ (lower half of the circuit).

For the comparisons $a < b$, the LUTs are configured to compare two bits of a and b each. As shown earlier in Figure 2 the LUTs on the Virtex-5 chip can have up to two outputs and can be connected to up to 5 common inputs. The two outputs are then connected through the fast carry-multiplexers . This results in a carry chain where the multiplexer selects the lower input if O6 is high. Otherwise, the carry-multiplexer selects the output O5 of the LUT. Therefore, the two Boolean functions f_5 and f_6 implemented by the LUT are

$$f_6(a_i, a_{i+1}, b_i, b_{i+1}) = \big(\bar{a}_i\bar{b}_i \vee a_i b_i\big)\big(\bar{a}_{i+1}\bar{b}_{i+1} \vee a_{i+1}b_{i+1}\big)$$
$$f_5(a_i, a_{i+1}, b_i, b_{i+1}) = \bar{a}_i b_{i+1} b_i \vee \bar{a}_{i+1}\bar{a}_i b_i \vee \bar{a}_{i+1} b_{i+1} \ .$$

Here, a_i and b_i refers to the i-th of the two integers a and b bit in little-endian order. f_6 compares the two bit positions (a_{i+1}, a_i) and (b_{i+1}, b_i). If they are not equal f_5 evaluates the predicate $(a_{i+1}2^{i+1} + a_i 2^i) < (b_{i+1}2^{i+1} + b_i 2^i)$.

The lower array of LUT pairs implement a multiplexers that select the right bits for the min- and the max-output of the comparator element using the predicate $a < b$. Let c the Boolean value of the comparison. Then, the LUT g_6 for the minimum-output is

$$g_6(a_i, b_i, c) = a_i c \vee b_i \bar{c}$$

and for the maximum-output

$$h_6(a_i, b_i, c) = b_i c \vee a_i \bar{c} \ .$$

Resource Usage. From Figure 10 it can be seen that a comparator which performs a compare-and-swap operation of two m-bit numbers can be implemented using $\lceil 5m/2 \rceil$ LUTs and $\lceil m/2 \rceil$ carry-multiplexers. Usually, chip utilization is measured in the number of occupied slices. The number of slices used for a design consisting of a given number of LUTs depends on the packaging strategy followed by the placer of the FPGA tool chain. In a optimal packaging with maximum density where all four LUTs in a slice are used (see Figure 2) in total $\lceil 5m/8 \rceil$ FPGA slices are used for each comparator. Thus, for $m = 32$ at least 20 slices are required for each comparator. This results in an upper bound of 1,024 comparators that be placed on our Virtex-5 FX130T chip. Note that in practice, however, the placer does not use this maximum packing strategy. In general, not every slice is fully occupied, i.e., all its LUTs are in use. Sometimes is is more efficient to co-locate a LUT with the input output block (IOBs) to the chip pins in order to reduce routing distances and hence latency. The slice usage can be even higher as the tool may be forced to use LUTs as plain "route-through" elements when it runs short on direct connection wires in the interconnect fabric.

Latency of a Single Comparator. The FPGA implementation in Figure 10 is particularly time efficient. All lookup tables are wired in a way such that all table lookups happen in parallel. Outputs are combined using the fast carry logic implemented in silicon for this purpose. Ignoring routing delays for the moment the latency of the circuit, i.e., the time until until output signals "min" and "max" of the comparator are valid after applying the inputs is given by sum of two LUTs (one of the comparison chain and one multiplexer LUT) and the propagation delay of the chain of $\lceil 5m/2 \rceil$ carry-multiplexer. From the Virtex-5 data sheet [34] the logic delay (excluding routing in the network) is 0.89 ns for $m = 32$.

Comparators for Floating Point Numbers. When sorting floating point numbers the compare-and-swap elements of the sorting network have to be replaced. The logic used to evaluate the predicate $a < b$ for two floating point numbers is significantly different from two integer values. Xilinx provides an IP core that implements the $a > b$ comparison for floating-point numbers. It supports the basic IEEE-754 single- and double-precision format, however, without denormalized numbers (treated as zeros). Table 3 shows the number of lookup tables used for a single compare-and-swap element for different data types. The numbers are subdivided into the logic used to evaluated to the predicate $a > b$ and the multiplexer logic to select the min/max value. Since single-precision is also 32 bits wide the multiplexer logic has the same complexity as the integers used in this paper. The single-precision comparison $a > b$ requires 108 Virtex-5 LUTs in total compared to

	$a > b$	min/max	total
32-bit integer	16 LUTs	64 LUTs	80 LUTs
single precision float	44 LUTs	64 LUTs	108 LUTs
double precision float	82 LUTs	128 LUTs	210 LUTs

Table 3 Number of LUTs required for different comparator types. The numbers are subdivided into the logic evaluating the predicate and the logic that selects the min/max values based on the predicate value.

the $\lceil m/2 \rceil = 16$ LUTs for integers. For double-precision 210 LUTs are required.

5.3 Asynchronous Sorting Networks

The sorting network is implemented by instantiating comparators and wiring them accordingly. As pointed out earlier, there are no explicit stages and register that buffer intermediate results. Instead, the comparator circuits (LUTs and carry-multiplexer) form a large network of *combinatorial logic*.

We can provide a lower-bound for the chip area required for the entire sorting network. As pointed out in the previous section, comparator elements require $5m/2$ LUTs each, where m is the data width in bits. The total number of lookup tables thus is

$$\#\text{LUTs} = \frac{5}{2} C(N) m \ .$$

Using the $C(N)$ from Table 2 we compute the number of lookup tables for even-odd mering and bitonic merging networks:

$$\#\text{LUTs}_{\text{ even-odd}} = 5m(p^2 - p + 4)2^{p-3} - \frac{5m}{2}$$
$$\#\text{LUTs}_{\text{ bitonic}} = 5m(p^2 + p)2^{p-3} \ .$$

The total area consumption measured in number of occupied slices depends on the packaging. We can provide a lower bound based on the following simplifying assumption that that multiplexer LUTs (see Figure 10) are not placed in the same slice slice as the logic used to evaluate the $a < b$ predicates (no combination of "unrelated" logic). The area utilization can be estimated as

$$\#\text{slices}_{\text{ even-odd}} = 5m\left[(p^2 - p + 4)2^{p-5} - 1/8\right]$$
$$\#\text{slices}_{\text{ bitonic}} = 5m(p^2 + p)2^{p-5} \ .$$

Ignoring additional LUTs used as "route-throughs" the chip area of the 8-element even-odd merging network (for $m = 32$) shown in Figure 8(a) containing 19 comparators requires 380 slices, i.e., 1.86 % of the Virtex-5 FX130T chip. The corresponding network based on bitonic mergers (Figure 8(b)) requires 24 comparators resulting in 480 FPGA slices, or equivalently, 2.34 % of the chip.

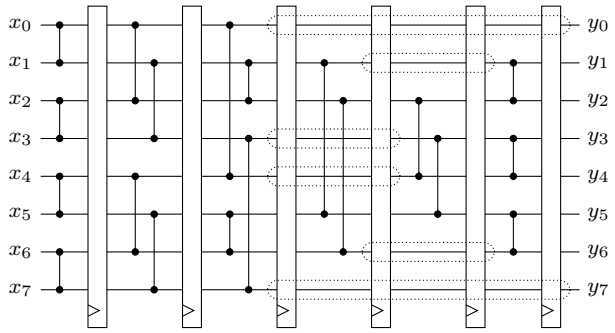


Fig. 11 Pipelined synchronous even-odd merge sorting networks using six $8 \times 32 = 256$ bit pipeline registers. Dotted rectangles indicate register stages that can be combined into a *shift register*.

5.4 Synchronous Implementation on FPGA

In a synchronous design an external clock signal moves the data from stage to stage through the sorting network. To this extent, the comparator outputs are connected to banks of flip-flips, called *stage registers*. The register store the input during the rising edge of the clock signal. The output of a stage register is then fed to the next comparator stage as shown in Figure 11.

Latency. The latency is determined by the clock frequency f and the depth $S(N)$ of the sorting network. In fact, the time between applying the data at the input and reading the sorted data at the output is given by $S(N)/f_{\text{clk}}$. For example, the even-odd merging network shown in Figure 11 on our Virtex FX130T–2 FPGA can be operated at $f_{\text{clk}} = 267\,\text{MHz}$. Hence, the overall latency for the 6-stage network is $6/267\ \text{MHz} = 22.5$ ns.

Pipelining. Synchronous sorting networks can further be implemented in a fully-pipelined way. This allows to keep an N-item set "in-flight" at every stage of the sorting network. Because the outputs of the comparator are buffered in a register after every stage, a complete new N-set can be inserted at the input every cycle.

As it can seen in Figure 11, in even-odd merging networks not all wires are processed by a comparator in every cycle. For example, in the third stage, the wires x_0, x_3, x_4, and x_7 are not processed by a comparator. In order to obtain a pipeline these wires still need to buffered by a register as shown in the Figure 11. This increases the number of occupied slices.

Resource Usage. The synchronous implementation differs from the asynchronous network by the stage registers. In a non-fully pipelined implementation the registers can be easily accommodated in the comparator logic. The outputs of the lookup tables g and h of the asynchronous comparator implementation (see Figure 10) can simply be connected to the corresponding flip-flops (see Figure 2 for the LUT–flip-flop configu-

ration in an FPGA slice). Hence, no additional slices are used for comparators and the total resource consumption of a sorting network identical to the asynchronous implementation, i.e., for a sorting network consisting of $C(N)$ comparators on N m-bit inputs $C(N)\lceil 5m/8 \rceil$ FPGA slices. Again, this is a lower-bound that is only reached if all slices are fully-occupied, i.e., all 4 LUTs/flip-flops of a slice are used, and no additional lookup tables are used as "route-throughs".

For the fully-pipelined implementation we can provide a simpliar lower-bound for the chip area required for fully-pipelined implementation. Now, a complete Nm-bit register is needed for each stage. Hence, the total number of LUTs and flip-flops (FFs) required is

$$\#\text{LUTs} = \frac{5}{2}C(N)m$$
$$\#\text{FFs} = S(N)Nm \ .$$

It can be easily verified that the resource usage in even-odd mering and bitonic merging networks is given by

$$\#\text{LUTs}_{\text{ even-odd}} = 5m(p^2 - p + 4)2^{p-3} - \frac{5m}{2}$$
$$\#\text{LUTs}_{\text{ bitonic}} = 5m(p^2 + p)2^{p-3} \ .$$

The number of stages $S(N)$ is the same for both network types, therefore, also the number of registers:

$$\#\text{FFs} = 4m(p^2 + p)2^{p-3} \ .$$

For the lower bound on the slice count we are using the assumption that the register following a comparator is always placed in the same slice, i.e., the output of the multiplexer-LUT is directly routed to the flip-flop register that is co-located with that LUT. Furthermore, we assume that flip-flops of stage registers without a comparator (e.g., shown inside dotted rectangles in Figure 11) are not placed in the same slice as the logic used to evaluate the $a < b$ predicates (no combination of "unrelated" logic). The area utilization then is:

$$\#\text{slices}_{\text{ even-odd}} = m\left[(5p^2 + 3p + 4)2^{p-5} - 1/8\right]$$
$$\#\text{slices}_{\text{ bitonic}} = 5m(p^2 + p)2^{p-5} \ .$$

Note that under these placement assumptions the slice usage for bitonic merging networks is identical to the asynchronous implementation. This is due to the fact that in bitonic networks there is a comparator for each wire in every stage, hence, all flip-flops registers can be co-located with a lookup table belonging to a comparator such that no additional slices are required.

5.5 Sorting Networks on CPUs

Sorting networks for CPUs have been extensively studied in literature, in particular for exploiting data parallelism on modern SIMD processors [7; 10; 11; 20]. In this section we show how sorting networks can be directly implemented on general-purpose CPUs. We show the implementations for two different hardware architectures: Intel x86-64 and PowerPC. We use these implementations later to compare the FPGA design against.

Neither of the two architectures provides built-in comparator functionality in its instruction set. We therefore emulate the functionality using conditional moves (x86-64) or the carry flag (PowerPC). The following two sequences of assembly code implement the comparator operation for PowerPC and x86-64 processors:

$$[r8, r9] \leftarrow [\min(r8, r9), \max(r8, r9)] \ .$$

PowerPC Assembly	x86-64 Assembly
`subfc r10,r8,r9`	`movl %r8d,%r10d`
`subfe r9,r9,r9`	`cmpl %r9d,%r8d`
`andc r11,r10,r9`	`cmova %r9d,%r8d`
`and r10,r10,r9`	`cmova %r10d,%r9d`
`add r9,r8,r11`	
`add r8,r8,r5`	

Neither piece of code makes use of branching instructions. The same property has important consequences also in code for traditional CPUs. Branch instructions incur a significant cost due to flushing of instruction pipelines (note that sorting algorithms based on branching have an inherently high branch mis-prediction rate). This is why the use of a sorting network is a good choice also for CPU-based implementations.

Related Implementations using SIMD. Chhugani *et al.* [7] describes an SIMD implementation for sorting single precision floating point numbers using Intel SSE instructions. Similar work is done for PowerPC AltiVec instruction set for both integer and single precision floating point data in [20]. In both cases, data parallelism in SIMD is used to sort multiple elements in a SIMD vector register in one step. For example, in [7] four 32-bit single precision floating point numbers are compared and swapped. Below, we briefly outline how in [7] two vectors $\mathtt{A} = (a_3, a_2, a_1, a_0)^T$ and $\mathtt{B} = (b_3, b_2, b_1, b_0)^T$ are compared.

Assume that we want to compare a_0 to a_1, a_2 to a_3, b_0 to b_1 and b_2 to b_3. This pattern occurs, for example, in the first stage of 8-element bitonic merging network shown in Figure 8(b). The Intel SSE architecture provides two instructions that determine the *element-wise* minimum and maximum of two vectors. In order to perform the desired comparisons the elements in the two vectors have to be shuffled into the correct position which is done by additional shuffle instructions. The required shuffle operations are illustrated in Figure 12. The operations can be directly implemented in C using SSE-intrinsics as follows:

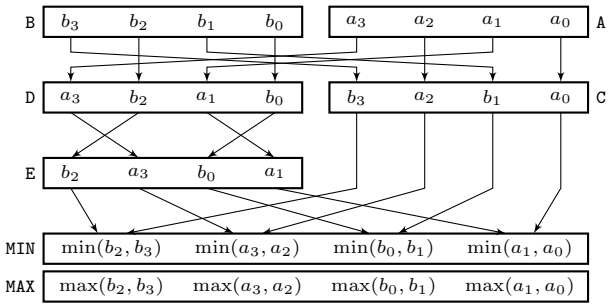


Fig. 12 Vector compare implemented using SSE instructions.

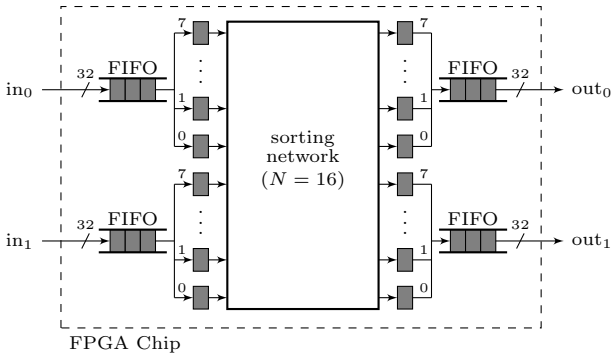


Fig. 13 Sort chip architecture used to evaluate implementations of sorting networks. Example shown for $N = 16$.

```
__m128 A, B, C, D, E, MIN, MAX;

C = _mm_blend_ps(A,B,0xA);
D = _mm_blend_ps(B,A,0xA);
E = (__m128)_mm_shuffle_epi32((__m128)D,0xB1);
MIN = _mm_min_ps(C,E);
MAX = _mm_min_ps(C,E);
```

In [7] sorting is split into different stages to account for the fixed-length SIMD registers. First, an in-register sort phase sorts 16 elements in 4 SIMD registers. Then a 2×4 bitonic merging network $\textsc{Bm}(4)$ is used to merge two resulting sorted lists. The fixed vector length of SSE makes Bitonic networks a good choice as they have a constant number of comparators $N/2$ in each stage. Even-odd sorters would require additional buffering which adds to the cost for shuffling elements in vectors. This shuffling overhead increases for larger networks which along with the fixed number of SIMD registers (16 on x86-64) available limit the scalability of this approach. In FPGAs implementations this additional shuffling translates into an increased signal routing complexity which also limits scalability.

6 Evaluation: Sorting Circuits on FPGAs

In this section we provide a detailed evaluation of the sorting network implementations on our Virtex-5 FPGA (FX130T). Before turning to the application use-cases in Sections 7 and 8 we analyze both the asynchronous and synchronous implementation of the *even-odd merging* and *bitonic merging* networks without the side-effects caused by the attachment of FPGA (e.g., bus and memory performance).

To this extent, we implement the sorting network as a dedicated sort chip. Data to be sorted is applied at I/O pins of the FPGA. Similarly, the sorted output can be read from an other set of I/O pins. The sort chip approach can be regarded as being artificial because an integrated circuit in custom silicon only implementing sorting in practice is of limited use. Nevertheless, it provides an environment to evaluate the sorting networks. When designing the chip we consider two important aspects. First, the implementation must be fully functional, that is, no simplifications are allowed that might lead the FPGA tool chain to shortcut parts of the design. For example, all input and outputs of the sorting networks must be connected to an I/O pin of the chip, otherwise, sub-circuits driving unconnected signals might be pruned in the optimization stage of the synthesis.

Second, for evaluating the raw performance of the sorting circuit, routing pressure when connecting to I/O blocks must not dominate the overall speed. Although it is in principle possible to connect all inputs and outputs for an $N = 8$ element sorting network to the FPGA pins it leads to longer routing distances because a large chip area needs to be covered since the I/O are uniformly distributed over the chip. For larger networks $N > 8$ more than the 840 I/O pins available on the Virtex-5 FX130T FPGA are required to interface the sorting network.

Hence, in order to minimize the impact of routing I/O signals we significantly reduced the width of the chip interface and use an architecture as shown in Figure 13. The key are FIFO buffers (BRAM blocks) placed at the input and output of the sorting network. The FIFO buffers that have different widths at the read and write interfaces. Xilinx provides FIFO IP cores that can have a width ratio between inputs and outputs of up to 1:8 or 8:1. For example, for $N = 8$ we use an input-side FIFO with an write with of 32 bit. This allows us to write one 32 bit word per clock cycle. Using a width ratio 1:8 we can read 8 consecutive elements from this FIFO into the sorting network. Similarly for the output-side FIFO in a 8:1 configuration, we can write all 8 32-bit outputs of the sorting network into

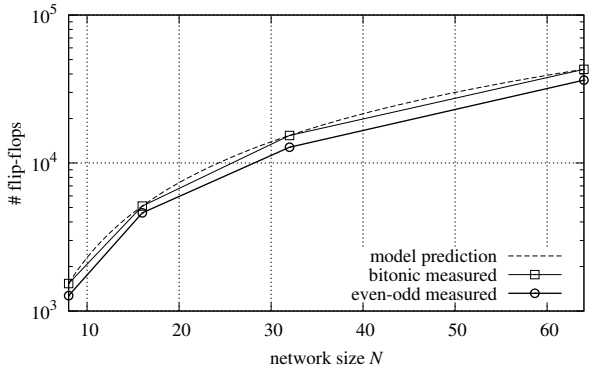


Fig. 14 Flip-flop usage of synchronous fully-pipelined implementations. While the model accurately predicts the resource requirements for bitonic merging networks it overestimates the flip-flop usage for even-odd merging networks.

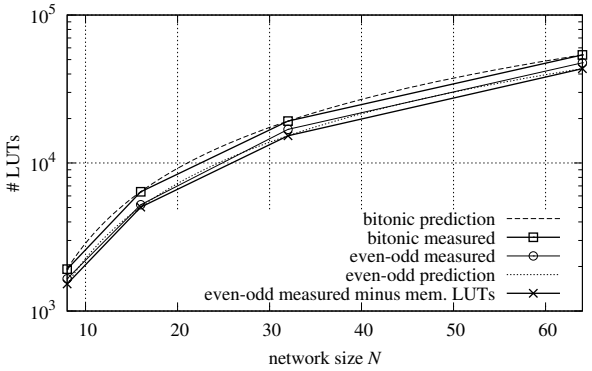


Fig. 15 LUT usage of synchronous fully-pipelined implementations. The model accurately predicts the resource consumption of bitonic merging networks. When LUTs used as shift register lookup table in even-odd merging networks are subtracted the model predictions are also correct for even-odd networks.

the FIFO. The output FIFO is the connected to the output pins of the chip through a 32-bit wide interface, such that we can read the sorted output one element per clock cycle. For network sizes $N > 8$ we use multiple FIFO lanes with ratio 1:8 and 8:1. Figure 13 shows two FIFO lanes for $N = 16$.

An additional advantage of using FIFOs is that they can be clocked at a different rate than sorting network. This isolates the timing analysis of the sorting network from the IOBs. Although it is impossible to clock the FIFOs eight times higher than the sorting network, we nevertheless can try to maximize the clock of the sorting network in order to determine the raw speed of the sorting network alone.

We evaluate resource consumption the network types and the synchronous and asynchronous implementations as follows. The resources used by the implementations (number of lookup tables, flip-flop registers and slices) are shown for the sorting network alone excluding logic for handling clock and the FIFOs. We estimate the resource consumption of the sorting network by using the number of the complete circuit. To this extent, we replace the sorting network by a "pass-through" and the full implementation including the sorting network. Since the input/output logic in both case is the same, the difference is due to the actual sorting network. In the following we only report the difference numbers.

6.1 Synchronous Implementations

Figure 14 shows the number of flip-flops (registers) used in the synchronous, fully-pipelined implementation of the even-odd and bitonic sorting network. The dotted line shows the prediction of the cost model introduced in Section 5.4. The model predicts the same value for both network types. It can be seen in Figure 14 that the model accurately predicts the flip-flop number for the bitonic sorting network. However, for even-odd sorting networks the model overestimates the register usage. This can be explained by the specific structure of even-odd networks that is not considered by the simple model. In even-odd networks not every wire has a comparator in every stage. This has an important consequence in a fully-pipelined implementation shown in Figure 11. Several stages without comparators represent shift registers (shown as dotted rectangles in Figure 11). Instead of using flip-flop registers the shift registers can be implemented more efficiently on Xilinx FPGAs using LUTs configured as such called *shift register lookup tables* (SRL). Hence, the actual number of flip-flop registers is reduced for even-odd sorting networks.

Replacing flip-flops by LUTs increases number of LUTs, such that the LUT resource model will underestimate the LUT usage for even-odd networks. This can be seen in Figure 15 that shows the LUT utilization. The figure also shows the predicted values for both network types (from Section 5.4). Whereas the model prediction is correct for bitonic networks it underestimates the LUT usage for even-odd networks. However, when subtracting the number of LUTs configured as SRL from the total number the model for even-odd networks is accurate too.

When comparing the synchronous implementation the two network architectures even-odd networks require less chip space, both, in number of flip-flop registers and lookup tables.

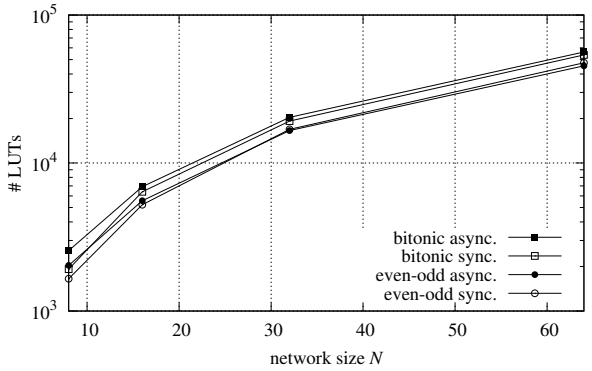


Fig. 16 LUT usage of asynchronous and synchronous fully pipelined implementations. Asynchronous circuits result in a higher LUT usage.

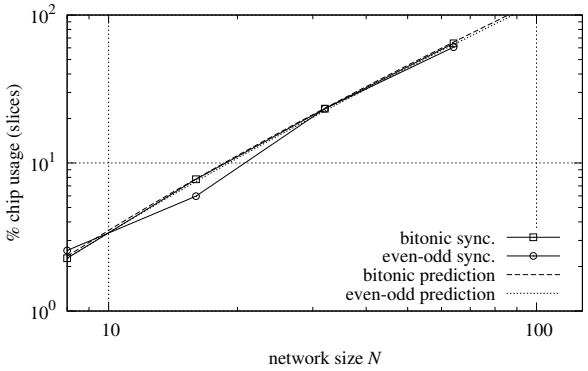


Fig. 17 Chip usage (measured in slices) of fully-pipelined synchronous implementations.

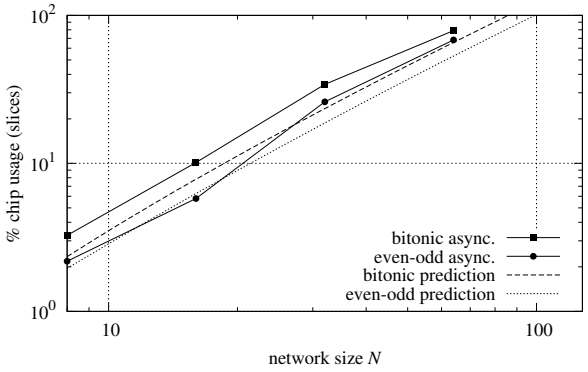


Fig. 18 Chip usage (measured in slices) of asynchronous implementations.

6.2 Asynchronous Implementation

Asynchronous implementations do not contain any flip-flops in the sorting network. By analysing the routed FPGA design we could verify that FPGA design tools furthermore did not introduce any flip-flops, for example, registers for pipelining or latency balancing. The lookup table utilization is shown in Figure 16. For comparison the figure also shows the effective number of LUTs used in the synchronous designs. It can be seen that for bitonic merging networks the asynchronous implementation always requires more LUTs than for the synchronous design. It turns out that this also holds even-odd networks once the additional LUTs used in synchronous implementations for shift registers are subtracted.

It is not quite clear why the asynchronous version require more lookup tables. We believe it is an artefact introduced by the Xilinx design tools. An analysis of the routed design showed that LUTs were not fully used, e.g., not all four inputs in the comparators for evaluating the predicates $a > b$. The difference to synchronous circuits is that in the asynchronous case each output bit of the sorting network can be expressed as a huge Boolean function of all inputs bits, e.g., for $N = 8$ sized network, there are 256 Boolean functions with 256 Boolean inputs each. During synthesis, the tools try to minimize these functions and later on map the resulting expressions back to FPGA LUTs. We believe that this process is based on heuristics and has limitations in performance.

6.3 Chip Usage

Figure 17 and 18 show the overall chip utilization in % of FPGA slices for the synchronous and asynchronous implementations respectively. Both plots are down in double-logarithmic scale. The resulting straight line corresponds to a power function. We are able to place designs up to $N = 64$ elements onto the Virtex-5 FX130T FPGA chip. When synthesizing networks for $N = 128$ the tools will abort due to overmapping over both registers and lookup tables.

In general, the slice utilization highly depends on the timing constraints, i.e., the clock frequency for synchronous networks and the maximum signal delay through asynchronous network. The chip utilization values we report here are obtained at the highest performance constraints that can be met by the FPGA tools. For the fully-pipelined implementations in Figure 17 we can observe that the slice usage roughly corresponds to the model prediction. The outlier for even-odd networks at $N = 16$ seems to related again to heuristics in the tools as it only occurs at tight timing constraints.

The chip utilization for asynchronous circuits (Figure 18) significantly deviates from the model predictions. In general, the model underestimates the utilization, in particular for larger network sizes. An analysis of the synthesized design showed that many slices are not fully occupied, i.e., not all LUTs or flip-flops are used. We observe this behavior when the place&route

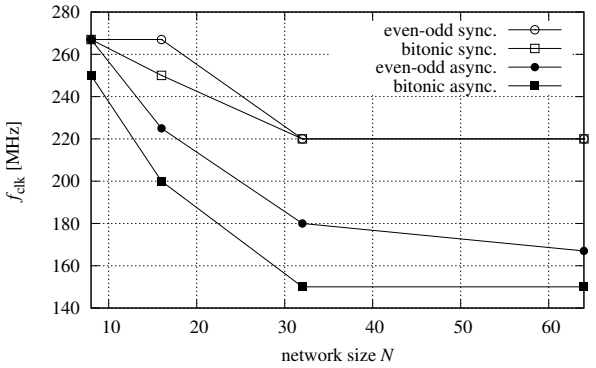


Fig. 19 Maximum clock frequencies the sorting network implementations can be operated.

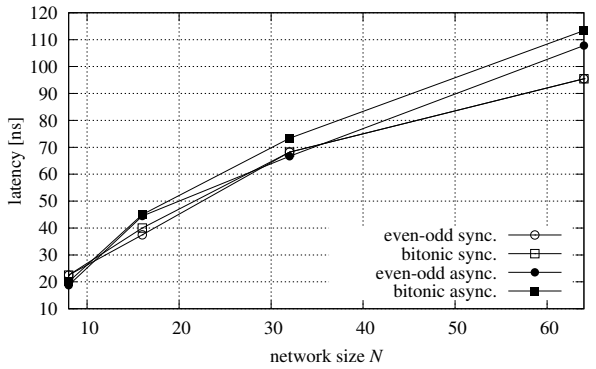


Fig. 20 Data latency in the different sorting network implementations.

stage optimizes for speed instead of space. This the case here as we chose tight timing constraints while there is still enough resources (area) available on the chips such that the tools are not forced to combine unrelated logic into the same slices.

In conclusion, we can observe that the resource models introduced earlier work best for flip-flop and LUT usage for synchronous circuits. They are less accurate for asynchronous circuits, in particular for slice usage.

6.4 Circuit Performance

For FPGA as for any hardware design, performance is given by the timing behavior. The timing behavior is specified during synthesis through one or more time constraints. The tools then try to find a design during the place&route phase that meets these constraints. If this phase completes successfully the system is able to operate that at this timing. The phase fails if one or more constraints are not met. In this case, the system does not operate correctly at this timing. The designer is then left to operate the circuit as close to the desired parameters the tool was able to synthesize a design, otherwise, the timing constraints have to be relaxed and the entire process repeated until a successful design is found.

For synchronous designs we only set the desired clock frequency of the sorting network. We gradually decrease the clock frequency until the place&route phase completes successfully. We plot the clock frequency in Figure 19. Asynchronous networks have two timing constraints. First, the core clock which es needed to drive the scheduler that reads the input data from the FIFOs, applies the data to sorting network, waits until the output of the sorting work is valid, and then stores the output in the out-FIFO. The second, parameter is the latency in the asynchronous sorting network which corresponds to the longest delay path in the sorting network. We round this path delay down to the next closest number of clock cycles. Clearly, the two timing parameters are correlated. We perform a search in this 2-dimensional space as follows. First, we maximize the clock frequency f_{clk} and set delay constraint for the sorting network to $\lfloor {}^{S(N)}/_{f_{\text{clk}}} \rfloor$. $S(N)$ is the number of swap stages in sorting network. Once we found the maximum f_{clk} we gradually reduce the path delay constraint until no valid design can be found.

As it can be seen in Figure 19 the clock frequency decreases as the network size increases. This behavior corresponds to algorithms in traditional computing where execution times increase (or throughput decreases) with the problem size. If the clock frequency would not increase the throughput would increase as N grows. It can also be seen from the figure that synchronous circuits can be clock significantly higher. There is no significant difference between the two network types. For asynchronous circuits the higher complexity of bitonic networks result in a lower clock speed.

Figure 20 shows the latency of the sorting network. It is measured as the time between applying the inputs at the sorting works and reading the sorted data the output of the sorting network. For the fully-pipelined implementations the latency is equal to ${}^{S(N)}/_{f_{\text{clk}}}$. For asynchronous implementations we directly determine the latency L. It can be seen that for networks $N > 8$ synchronous circuits have lower latency, even though the additional register stages in the sorting network inherently uses the signal propagation through the network. The reason why asynchronous networks do have a higher latency is due to the lower overall clock speed that feeds and extracts data to and from the network. The large combinatorial circuits of asynchronous implementations have a significant negative impact on the clock frequency as shown in Figure 19 such that latency gains by omitting the stage register cannot compensate the loss in the overall clock frequency.

Throughput is related to latency L. Fully-pipelined implementations can process an N-set every clock cycles while asynchronous implementations can process a tuple every $\lceil Lf_{\text{clk}} \rceil$ cycle. Here we can observe the significant gains of fully-pipelined designs. For example, both synchronous networks can process 64 elements at 220 MHz. This corresponds to a throughput of 14.08×10^9 elements/sec. Since the elements are 32-bit in size, the resulting throughput is 52.45 GiB/sec. In contrast, the fastest corresponding asynchronous network (even-odd as shown Figure 20) has a latency of 113.3 ns 150 MHz which results in a throughput of 8.8×10^6 elements/sec or equivalently 2.1 GiB/sec. The high throughput numbers are very promising for the FPGA technology. However, so far we only analyzed the isolated performance the sorting network. The overall system performance depends on the integration, that it the attachment of the FPGA to the rest of the system. We analyze this performance through to different use cases in the next two sections.

7 Use Case: A Streaming Median Operator

As a first use-case for the sorting network circuits we choose a *median* operator over a count-based *sliding window* implemented on the aforementioned Xilinx board. This is an operator commonly used to, for instance, eliminate noise in sensor readings [31] and in data analysis tasks [32]. For illustration purposes and to simplify the figures and the discussion, we assume a window size of 8 tuples. For an input stream S, the operator can then be described in CQL [4] as

```
Select median(v)
  From S [ Rows 8 ] .
```
(Q_1)

The semantics of this query are illustrated in Figure 21. Attribute values v_i in input stream S are used to construct a new output tuple T'_i for every arriving input tuple T_i. A conventional (CPU-based) implementation would probably use a ring buffer to keep the last eight input values (we assume unsigned integer numbers), then, for each input tuple T_i,

(1) *sort* the window elements $v_{i-7}, \ldots, v_i$ to obtain an ordered list of values $w_1 \leq \cdots \leq w_8$ and
(2) determine the *mean value* from the ordered list. For the even-sized window we return w_4, corresponding to the *lower median*. Alternatively, w_5 corresponds to the *upper median*.

The ideas presented here in the context of the median operator are immediately applicable to a wide range of other common operators. Operators such as selection, projection, and simple arithmetic operations (max, min, sum, etc.) can be implemented as a combination of logical gates and simple circuits similar to the ones presented here. We described one strategy to obtain such circuits in [27].

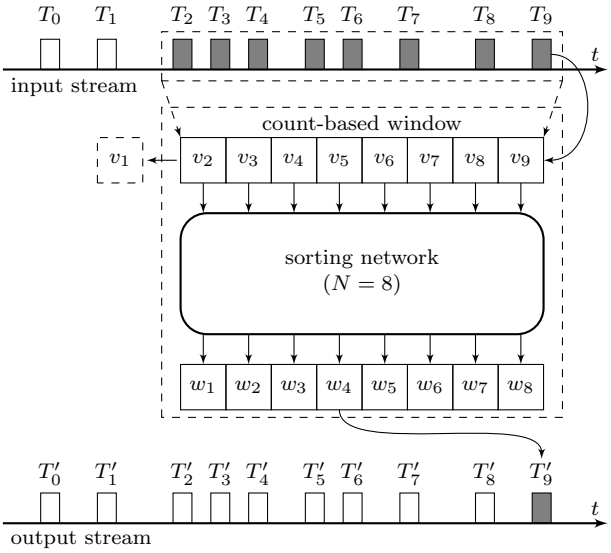


Fig. 21 Median aggregate over a count-based sliding window (window size 8).

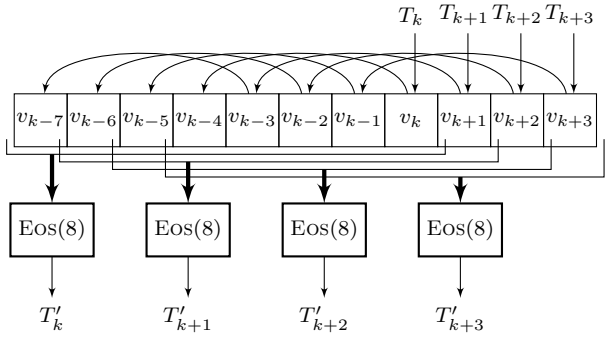


Fig. 22 Implementation of median operator able to process four items per clock tick.

7.1 An FPGA Median Operator

We take advantage of the inherent hardware parallelism when implementing the operator. The goal is to maximize throughput by choosing a design that is able to process several tuples per clock cycle. The design of the median operator is illustrated in Figure 22. The operator accepts four consecutive tuples $T_k, T_{k+1}, T_{k+2}, T_{k+3}$ in every clock cycle. The tuple's values $v_k, \ldots, v_{k+3}$ is then inserted into the sliding window which is implemented as a shift register. The shift register stores 11 32-bit elements. Since four new elements are inserted

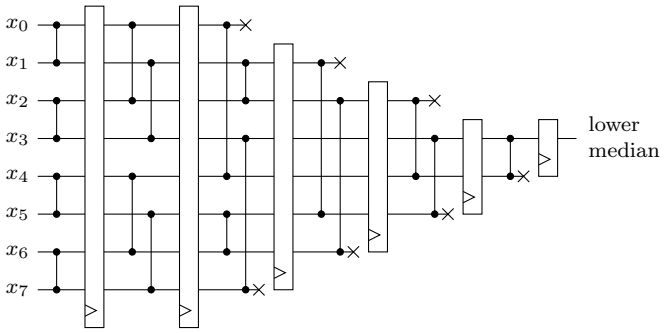


Fig. 23 When computing the median the complexity of the fully-pipelined synchronous even-odd merge sorting network can be reduced from 19 comparators and 48 32-bit registers to 10 comparators, 7 half-comparators and 29 32-bit registers.

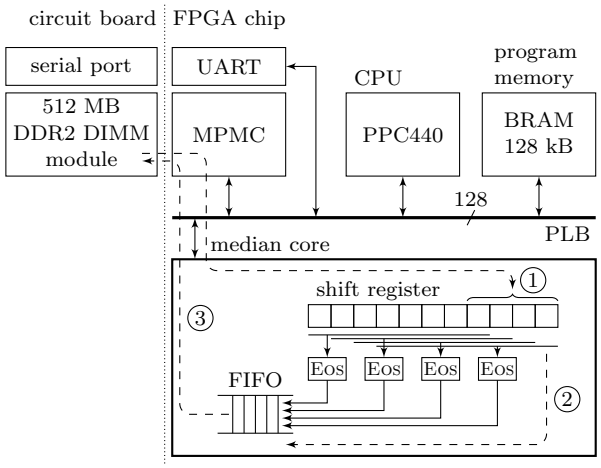


Fig. 24 Architecture of the on-chip system: PowerPC core, 3 aggregation cores, BRAM for program, and interface to external DDR2 RAM.

every cycle the elements have to move by four positions to the left. The 11-element shift register contains four overlapping sliding windows of length eight that are separated by one element. The elements of the four windows are then connected to four instances of synchronous, fully-pipelined *even-odd merging sorting networks* EOS(8) (see Figure 11). The lower median for each window finally appears at the fourth output of the corresponding sorting network. In summary, the result tuples $T'_k, \ldots, T'_{k+3}$ are computed as follows from the windows:

$$
\begin{aligned}
T'_k &\leftarrow \text{EOS}\big([v_k, v_{k-1}, \ldots, v_{v-7}]\big)_4 \\
T'_{k+1} &\leftarrow \text{EOS}\big([v_{k+1}, v_k, \ldots, v_{v-6}]\big)_4 \\
T'_{k+2} &\leftarrow \text{EOS}\big([v_{k+2}, v_{k+1}, \ldots, v_{v-5}]\big)_4 \\
T'_{k+3} &\leftarrow \text{EOS}\big([v_{k+3}, v_{k+2}, \ldots, v_{v-4}]\big)_4 \ .
\end{aligned}
$$

Since we are only interested in the computation of a median, a fully sorted data sequence is more than required. Consider the even-odd sorting network shown in Figure 11. The lower median value appears at output y_3. Therefore the upper and lower comparators of the last stage that sort y_1, y_2 and y_5, y_6 as well as the preceding register stages are not needed and can be omitted. The FPGA synthesis tool is able to detect unconnected signals and automatically prunes the corresponding part of the circuit. The pruned network is shown in Figure 23. Besides the reduction by 2 comparators and 19 32-bit registers the circuit complexity can further be reduced. Note that in Figure 23 7 comparators have one output unconnected. This means that the one the two 32-bit multiplexer that select the min/max value can be saved, which reduces the complexity by 32 LUTs.

7.2 System Design

So far we have looked at our FPGA-based database operator as an isolated component. However, FPGAs are likely to be used to complement regular CPUs in variety of configurations. For instance, to offload certain processing stages of a query plan or filter an incoming stream before feeding it into the CPU for further processing.

In conventional databases, the linking of operators among themselves and to other parts of the system is a well understood problem. In FPGAs, these connections can have a critical impact on the effectiveness of FPGA co-processing. In addition, there are many more options to be considered in terms of the resources available at the FPGA such as using the built-in PowerPC CPUs and soft IP-cores implementing communication buses or controller components for various purposes. In this section we illustrate the trade-offs in this part of the design and show how hardware connectivity of the elements differs from connectivity in software.

7.3 System Overview

Using the Virtex-5-based development board described in Section 3.2, we have implemented the embedded system shown in Figure 24. The system primarily consists of FPGA on-chip components. We use additional external (off-chip) memory to store larger data sets. The memory is provided by a 512 MB DDR2 DIMM module that is directly connected to the FPGA pins. The DIMM module operates at a bus clock of 200 MHz which corresponds to DDR2-400 with a peak transfer rate of 3200 MB/sec.

On the FPGA we use one of the built-in PowerPC 440 cores which we clock at the highest specified frequency of 400 MHz. The on-chip components are connected over a 128-bit wide *processor local bus* (PLB). We use 128 kB on-chip memory (block RAM) to store the code executed by the PowerPC (including code for our measurements). External memory used for the data sets is connected to the PLB through a *multi-port memory controller* (MPMC). It implements the DDR2 protocol. To control our experiments we interact with the system through a serial terminal. To this extent, we instantiate a soft IP-core for the serial UART connection link (RS-232).

Our streaming median operator participates in the system inside a dedicated processing core (Figure 24). As described in Section 7.1 the core contains the 11-element shift register and four sorting network instances. Additional logic is required to connect the core to the PLB. A parameterizable *IP interface* (IPIF, provided by Xilinx as a soft IP-core) provides the glue logic to connect the user component to the bus. In particular, it implements the bus protocol and handles bus arbitration and DMA transfers.

In order maximize performance while minimizing the CPU load we use DMA transfers initiated by the median core to access the memory. The DMA controller is implemented in the IPIF logic of our core. For our experiments we generate random data corresponding to the input stream in the external memory from a program running on the PowerPC core. Next, the CPU sets up the DMA transfers to read the input data from memory and to write back the median results. The two DMA transfers are setup by specifying the start addresses of the input data and result data as well ad the number of items to process. Note that the output data size is equal to the input data size. The CPU communicates these parameters to median core by writing them into three memory-mapped registers of the median core. The logic for the registers is also implemented in the IPIF. We implemented the core such that the processing is started implicitly after writing the size register. The processing consists of three phases shown in Figure 24.

(1) A read transfer moves the data from the external memory into the median transfer. For maximum efficiency the full 128-bit width of the bus is used. In other words 16 bytes are sent over the bus per clock cycle. The PLB operates at 100 MHz resulting in a peak bus bandwidth of 1,600 MB/sec. During for each clock cycle 4 32-bit elements are received by the aggregation. This is the reason why we designed the aggregation core in Figure 22 such that it can process four items at once.

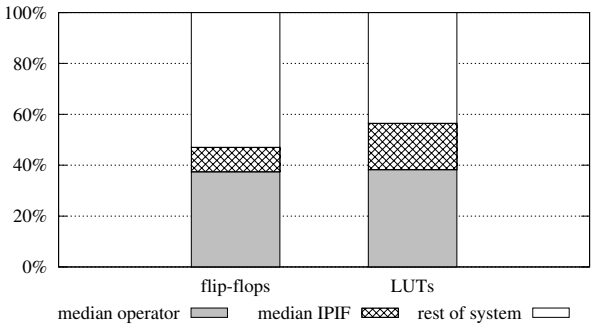


Fig. 25 Distribution of FPGA resources in a system with median core.

(2) The sorting network is fully pipelined hence we can process the input data immediately as it arrives. The median computation is performed in pipelined manner for four elements in parallel. The PLB and IPIF only permit one active DMA transfer at any given time, hence, we need store the result data for the later write-back DMA transfer to external memory. We implement this on-chip buffer as a FIFO memory. The size of the memory is equal to maximum transfer size of 4,080 bytes supported by the controller. Our logic splits larger data sizes in multiple read/write DMA transfers without CPU involvement.
(3) After completing a data chunk the result data in the FIFO buffer is written back to memory by write DMA transfer. After a 4,080 byte chunk is complete the next data chunk is read into the core (phase 1). After the last chunk is processed the median core signals completion to the CPU by rising an interrupt.

7.4 Evaluation

Resource Usage. The entire system occupies 28 % of the FPGA slices available on Virtex-5 FX130T chip. This includes not only the median core but also all additional soft IP-cores that are implemented using FPGA logic, for example, the memory controller, processor bus, and UART core. This figure does not include used components that are available in discrete silicon (hard IP-cores), such as the PowerPC core and block RAM. The design tools report 225 kB of block RAM memory used which corresponds to 17 % of the available on-chip memory.

We further analyzed the resources used by the median core itself. In particular, how much is spent for implementing the interface to the PLB and the DMA logic. As for the evaluation in Section 6 we replace the

operator implementation by a simple "route-through" logic, synthesize the design and compute the difference in flip-flops and LUTs to the complete design in order to estimate the chip resources used by the median operator alone. Figure 25 shows the distribution of flip-flop registers and lookup tables. The median core consisting of four 8-element even-odd sorting networks the sliding window, and control logic that schedules inserting data into the sorting network and extracting the results occupies about 40 % of all flip-flops and LUTs used by design. Also shown in Figure 25 is the space required for IP-core interface (IPIF) implementing the PLB interface and DMA transfer. 10 % of flip-flops and 18 % of the lookup tables are spent for the IPIF. Approximately half of the overall chip resources were used for logic unrelated to the median operator (memory controller, PLB, UART, etc.).

The implementation of median core is dominated by LUT usage. The median core uses 8.2 % of the LUTs available on the Virtex-5 FX130T FPGA while rest of the system occupies 6.3 % of chip's LUTs. Hence, from space perspective, it can be estimated that 11 instances of the median core can be placed on the chip. However, the number of components that can be connected to the PLB for DMA operations is limited to eight. Since the DMA bus functionality is also used by the PowerPC core this leads to at most 7 median cores that can be instantiated under the current system design.

Performance. The overall performance of the system is determined by the clock frequency as well as the latency and bandwidth limitations of the bus and memory interface. We operate the median core at the core system clock of 100 MHz. This clock is determined by other system components. In particular, the DDR2 memory controller is very sensitive to timing errors. Although, the sorting network operates at significantly lower clock speed compared to the evaluation in Section 6 (100 MHz vs 267 MHz) the design of the median operator still allows us to process a full bus width of data every cycle. Performance limitations is not due to the aggregation core but to the other system components.

For processing the 256 MB data set we the hardware implementation of the median operators requires 1.302 seconds. While processing this data set 2×256 MB are transferred, once from external memory into the sorting network and once from the on-chip memory holding the result data back to the external memory. This leads to an effective end-to-end throughput of 393 MB/sec. Putting this figure in contrast to the peak bandwidth of the DDR2 memory (3,200 MB/sec) and the PLB (1,600 MB/sec) there is an significant loss. The memory is accessed sequentially and we paid special care to avoid contention on the processor bus. For example, we made sure that the PLB is not occupied by other components (e.g., the CPU) during while the median core is processing data. We believe that the observed reduction in bandwidth is due to the arbitration overhead of the bus.

FPGA Performance in Perspective. FPGAs can be used as co-processor of data processing engines running on conventional CPUs. This, of course, presumes that using the FPGA to run queries or parts of queries does not result in a net performance loss. In other words, the FPGA must not be significantly slower than the CPU. Achieving this is not trivial because of the much slower clock rates on the FPGA.

Here we study the performance of the FPGA compared to that of CPUs. To ensure that the choice of a software sorting algorithm is not a factor in the comparison, we have implemented eight different sorting algorithms in software and optimized them for performance. Seven are traditional textbook algorithms: quick sort, merge sort, heap sort, gnome sort, insertion sort, selection sort, and bubble sort. Building accelerators with FPGAs is a complex and non-trivial processing. In order to perform a fair comparison we deliberately spent a considerable effort to also optimize the CPU-based implementations. The eighth implementation is based on the even-odd merge sorting network shown in Figure 23 using CPU registers. We implemented the sorting network using the assembly code variant shown in Section 5.5. Just as for the hardware implementation we applied the same optimization of the sorting that are possible for computing the lower median, i.e., removing comparator stages and optimizing the assembly code for "half-comparators". This process has lead to a hand-written, fully optimized and branch-free implementation of median computation in assembly language. The PowerPC implementation consists of 88 instructions. For Intel x86-64 we end up with 61 instructions.

We ran the different algorithms on several hardware platforms. We used an off-the-shelf Intel x86-64 CPU (2.66 GHz Intel Core2 quad-core Q6700) and the following PowerPC CPUs: a 1 GHz G4 (MCP7457) and a 2.5 GHz G5 Quad (970MP), the PowerPC element (PPE not SPEs) of the Cell, and the embedded 405 core of our FPGA. All implementations are single-threaded. For illustration purposes, we limit our discussion to the most relevant subset of algorithms.

Figure 26 shows the wall-clock time observed when processing 256 MB (as 32-bit tuples) through the median sliding window operator shown in Figure 21. The horizontal line indicates the execution time of the FPGA

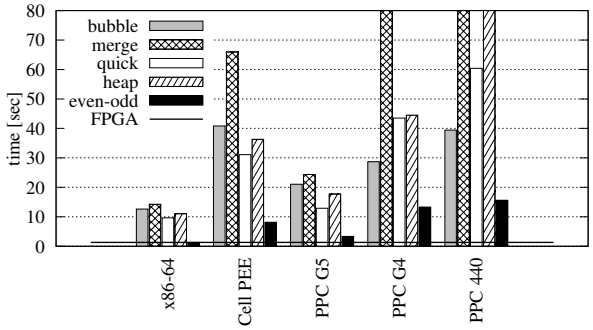


Fig. 26 Execution time for computing the stream median of a 256 MB data set on different CPUs using different sorting algorithms and on the FPGA.

implementation. Timings for the merge and heap sort algorithms on the embedded PowerPC core did not fit into scale (162 s and 92 s, respectively). All our software implementations were clearly CPU-bound. It is also worth noting that given the small window, the constant factors and implementation overheads of each algorithm predominate and, thus, the results do not match the known asymptotic complexity of each algorithm. The best CPU result is obtained for the hand-written even-odd merging implementation on the Intel Core2 Q6700. Processing 256 MB requires 1.314 sec.

The performance observed indicates that the implementation of the operator on the FPGA is able to slightly outperform a modern conventional CPU. Processing 256 MB requires 1.314 sec on the Intel Core2, compared to 1.302 s on the FPGA. This is a bit discouraging result given the large effort spent for the FPGA implementation. As we already pointed out Section 6, the sorting network itself is very fast. Here, the comparatively low full-system performance is due to the currently available system components and bus design. Building FPGA-based accelerators is no unlike to GPU difficult as the culprit is the same; getting data to and from the device. In [25; 27] we show that if directly combined with I/O FPGA can lead to significant performance improvements over traditional information systems. In [27] we build a 1 gigabit network interface (UDP/IP) for the FPGA and combined it with a data processing engine on the same chip. This allowed us to process data wire speed. In [25] we apply the same technique to object deserialization. Nevertheless, being not worse than CPUs the FPGA is a viable option for offloading data processing out of the CPU which then can be devoted to other purposes. When power consumption and parallel processing are factored in, FPGAs look even more interesting as co-processors for data management.

Intel Core 2 Q6700:	
Thermal Design Power (CPU only)	95 W
Extended HALT Power (CPU only)	24 W
Measured total power (230 V)	102 W
Xilinx ML510 development board:	
Calculated power estimate (FPGA only)	10.0 W
Measured total power (230 V)	40.8 W

Table 4 Power consumption of an Intel Q6700-based desktop system and the Xilinx ML510 FPGA board used in this paper. Measured values are under load when running the median computation.

Power Consumption. While the slow clock rate of our FPGA (100 MHz) reduces performance, there is another side to this coin. The *power consumption* of a logic circuit depends linearly on the frequency at which it operates (U and f denote voltage and frequency, respectively):

$$P \propto U^2 \times f \ .$$

Therefore, we can expect our 100 MHz circuit to consume significantly less energy than a 3.2 GHz x86-64 CPU. It is difficult to reliably measure the power consumption of an isolated chip. Instead, we chose to list some approximate figures in Table 4. Intel specifies the power consumption of our Intel Q6700 to be between 24 and 95 W (the former figure corresponds to the "Extended HALT Powerdown State") [30]. For the FPGA, a power analyzer provided by Xilinx reports an estimated consumption of 10.0 W. A large fraction of this power (5.3 W) is used to drive the outputs of the 234 pins that connect the FPGA chip. CPU having large pin counts have the same problem. Additional power is also spent for the PowerPC (0.8 W) and the different clock signals (0.5 W).

More meaningful from a practical point of view is the overall power requirement of a complete system under load. Therefore, we took both our systems, unplugged all peripherals not required to run the median operator and measured the power consumption of both systems at the 230 V wall socket. As shown in Table 4, the FPGA has a 2.5-fold advantage (40.8 W over 102 W) compared to the CPU-based solution here. The power values are signficantly higher for the Virtex-5 board that the 8.3 W wall power what we reported in [26] where we used a smaller board with a Virtex-II Pro FPGA. The higher power consumption is only partially due to increased power requirements of Virtex-5 FGPA. The new ML510 board also contains additional and faster components which even when inactive their quiescent currents lead to a higher overall power consumption. Additionally, the 250 W ATX power supply we use for the ML510 board is a switching power

supply which are known to have a low efficiency when operated singificantly below the nominal power (16 % in our case). A power-aware redesign of the board and the used of matching power supply can reduce the power consumption much below 40 W.

As energy costs and environmental concerns continue to grow, the consumption of electrical power (the "carbon footprint" of a system) is becoming an increasingly decisive factor in system design. Though the accuracy of each individual number in Table 4 is not high, our numbers clearly show that adding a few FPGAs can be more power-efficient than simply adding CPUs in the context of many-core architectures.

Modern CPUs have sophisticated power management such as dynamic frequency and voltage scaling that allow to reduce idle power. FPGAs offer power management even beyond that, and many techniques from traditional chip design can directly be used in an FPGA context. For example, using *clock gating* parts of the circuit can be completely disabled, including clock lines. This significantly reduces the idle power consumption of the FPGA chip.

8 Use Case: A Sorting Co-Processor

In the second use-case we directly integrate the sorting network into a system. We build an 8-element even-odd merging network and connect it to the PowerPC core. Instead of connecting it over the *processor local bus* (PLB) and mapping the core into the main memory seen by the CPU we implement the sorting core is implemented as an *Auxiliary Processor Unit* (APU). The APU is directly connected to the execution pipeline of the PowerPC 440 CPU. It was designed by IBM as a PowerPC extension to connect a floating-point unit (FPU) to the embedded core. For example, the FPUs of the IBM Blue Gene/L supercomputer are connected to the PowerPC 440 core through the APU interface. The APU does not have access to memory bus but the design benefits from a short communication path to the embedded CPU. This use-case illustrates another approach to integrate an FPGA accelerator into a heterogeneous system.

8.1 Heterogeneous Merge Sort

The hardware solutions described in this paper have the disadvantage that they can only operate on a fixed-length data set. As a workaround that allows variable sized input make used of the CPU to merge chunks of data that is sorted in hardware. The sorting algorithm

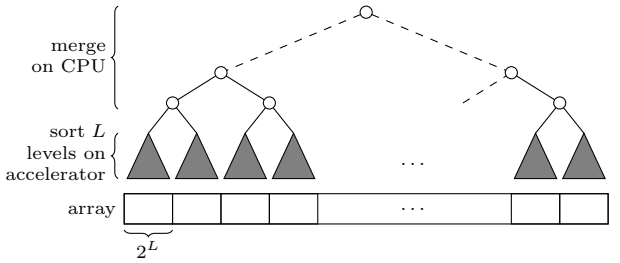


Fig. 27 Heterogeneous merge sort where the lowest L levels are performed on the FPGA accelerator.

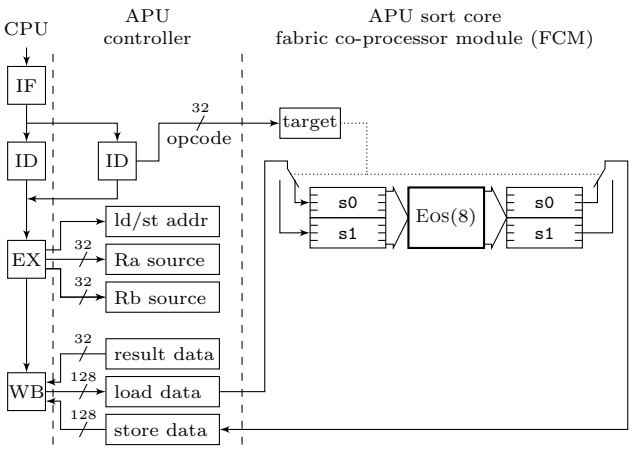


Fig. 28 Sort core as *Auxiliary Processor Unit*.

of choice here is *merge sort*. In this use-case we implement an 8-element even-odd merging network as a sort core in the APU. The FPGA accelerator will sort consecutive blocks of 8 elements in-place. The sorted blocks are then merged on the CPU as shown in Figure 27. This corresponds to a merge sort where the lowest $L = 3$ (leaf) levels are performed by the APU. For sorting N elements in total $\lceil \log_2(N) \rceil$ are required. Hence, for $N = 2^p$ elements, $p - L$ merge levels in software are needed.

8.2 Attachment to CPU Execution Pipeline

The APU processor is accessed through a set of additional machine instructions. These additional instructions include load/store, floating-point and as well as completely user-defined instructions. The APU processor can contain its own register file. The load/store instructions can then be used to transfer data between memory and APU register file. Since the APU is directly connected to the memory unit of the CPU the APU can also benefit from the data cache which makes sharing data between CPU and APU very efficient. The user-defined instruction can be used to pass data from the CPU register to the APU. Figure 28 shows the architecture. The APU consists of a controller imple-

mented in hard silicon and co-located with the CPU core and the custom-defined *fabric co-processor module* (FCM) implemented using FPGA components. The APU connects to the CPU at different points of this simplified 5-stage RISC pipeline. The APU controller first decodes APU-specific instructions (ID stage). Possible instruction include user-defined instructions such as `udi0fcm Rt,Ra,Rb` or FCM load/store instructions. The necessary operands are provided to the APU during the execution phase (EX stage), e.g., values for the two register operands Ra and Rb or the computed memory address for loads and stores. For user-defined instructions, the FCM computes the result and returns it to the CPU where it is into the target register Rt in the CPU's register file. For load instruction the CPU provides the data from memory up to 16 bytes in parallel. For stores the data returned by the FCM is written back to the memory. The connection between the APU controller and the FCM is implemented through a well-defined interface which contain the necessary signals that allow the CPU-side *APU controller* control the FCM and exchange data.

We access our sorting core through load/store instructions. The FCM contains two 16-byte registers `s0` and `s1` that are able to store 4 elements each. The CPU code for sorting 8-elements is shown below:

APU Assembly	
	`r8` ← address of input array `r9` ← address of output array `r10` ← 16 stride in bytes
`ldfcmux s0,r8,r10`	`s0` ← mem[`r8` + `r10`], `r8` ← `r8` + `r10`
`ldfcmx s1,r8,r10`	`s1` ← mem[`r8` + `r10`]
...	*6 additional writes to* `s1`
`stfcmux s0,r9,r10`	mem[`r9` + `r10`] ← `s0`, `r9` ← `r9` + `r10`
`stfcmx s1,r9,r10`	mem[`r9` + `r10`] ← `s1`

For sorting 8 values on the APU, we first load the input data from memory into `s0` and `s1`. The data loaded by using to instruction. `s0` corresponds to the first 4 elements, while the last 4 elements are written to `s1`. The `ldfcmux` instruction also updates the first source register operand whereas `ldfcmx` does not. We designed the FCM such that after writing `s1` the content [`s0`,`s1`] is fed into the sorting network (Eos(8)). The sorting network is implemented following a fully-pipelined synchronous design. In order to simplify instruction scheduling we clock the sorting network based on writing `s1`, i.e., after writing 6× to register `s1` the sorted output appears at the output of the sorting network. The sorted output is written back to memory using a FCM store instruction. Note that in fact `s0` and `s1` each refers to two different registers when loading or storing (see Figure 28). We can hide this 6-instruction latency by using *software pipelining* in the assembly program.

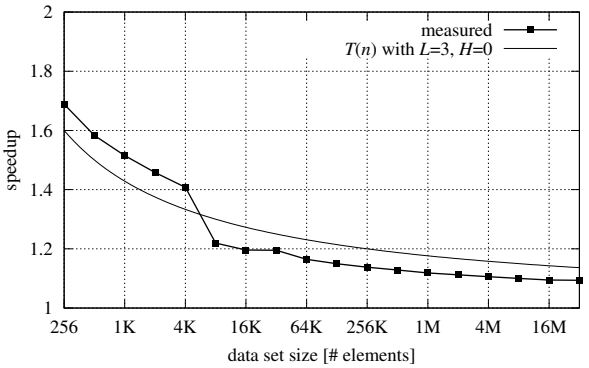


Fig. 29 Speedup of APU sorting core over traditional on chip sort

.

8.3 Evaluation

We evaluate the APU implementation and compare it to a CPU-only version of the merge sort algorithm running on the PowerPC 440 core. Figure 29 shows the speedup of the hardware acceleration for sorting arrays containing 256–16M elements.

The speedup decreases asymptotically as the size of the data set increases. The reason is that ratio between the work done by the CPU to work done in the accelerator decreases as the data set increases as the following simple analysis shows. Let $T(n)$ be the time to sort n elements. The recursive merge sort leads to recurrence equation $T(n) = 2T(n/2) + n$. By considering only $n = 2^p$, i.e., power of twos we obtain the following recursive definition:

$$T(2^p) = 2T(2^{p-1}) + 2^p \quad \text{with} \quad T(2^1) = 2 \ .$$

The solution to this recurrence equation is $T(2^p) = p2^p$. This corresponds to the execution time for the CPU-only implementation. Now, with the hardware acceleration the lowest L levels are performed in the APU, say spending a processing time H for each 2^L-element set. This changes the initial condition of the recurrence equation to $T(2^L) = H$. As it can easily be verified, the solution for the accelerated implementation is $T'(2^p) = H2^{p-L} + (p-L)2^p$. Hence, the speedup is

$$\text{speedup} = \frac{T(2^p)}{T'(2^p)} = \frac{p}{H2^{-L} + p - L} \stackrel{H=0}{\longrightarrow} \frac{p}{p-L}$$

Figure 29 also shows the speedup for the when $H = 0$, i.e., the operation performed by the accelerator requires no time. Clearly, it follows that for large datasets the speedup decreases to 1. The sharp decrease can see between 4K and 8K is the effect of the 32 kB data cache on the core. Using the aggregation the cache pressure can be reduced as not temporary data needs to be kept, hence, as long as the data fits into the cache the aggregation core can provide higher speedups. Although, the absolute values of the speedup obtained is not particularly high, this use-case illustrates how a tightly coupled accelerator can be implemented in a heterogeneous system. Also, it is another attempt to release the power of sorting network in hardware we observed in Section 6.

9 Summary

In this paper we have assessed the potential of FPGAs to accelerate sorting. We presented different approaches to implement sorting networks on FPGAs and discussed the on-chip resource utilization. Despite the complexity involved with designs at the hardware level the flip-flop and LUT utilization of a circuit can be estimated beforehand, in particular, for synchronous fully-pipelined implementations. We also showed how FPGAs can be used as a co-processor for data intensive operations in the context of multi-core systems. We have illustrated the type of data processing operations where FPGAs have performance advantages (through parallelism, pipelining and low latency) and discussed different ways to embed the FPGA into a larger system so that the performance advantages are maximized. Our evaluation shows that implementations of sorting networks on FPGA do lead a high performance (throughput and latency) on their own. However, the two use-cases put these high performance numbers into perspective. It challenging to maintain this performance once the hardware implementation of the algorithm is integrated into a full system. Next to raw performance, our experiments also show that FPGAs bring additional advantages in terms of power consumption. These properties make FPGAs very interesting candidates for acting as additional cores in the heterogeneous many-core architectures that are likely to become pervasive. The work reported in this paper is a first but important step to incorporate the capabilities of FPGAs into data processing engines in an efficient manner. The higher design costs of FPGA-based implementations may still amortize, for example, if a higher throughput (using multiple parallel processing elements as shown in Section 7) can be obtained in a FPGA-based stream processing system for a large fraction of queries.

Acknowledgements

This work was supported by an *Ambizione* grant of the Swiss National Science Foundation under the grant number 126405 and by the Enterprise Computing Center (ECC) of ETH Zurich (`http://www.ecc.ethz.ch/`).

References

1. Abadi DJ, Carney D, Çetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S (2003) Aurora: A new model and architecture for data stream management. The VLDB Journal 12(2):120–139
2. Abadi DJ, Ahmad Y, Balazinska M, Çetintemel U, Cherniack M, Hwang JH, Lindner W, Maskey AS, Rasin A, Ryvkina E, Tatbul N, Xing Y, Zdonik S (2005) The design of the Borealis stream processing engine. In: Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA
3. Ajtai M, Komlós J, Szemerédi E (1983) An O(n log n) sorting network. In: ACM Symposium on Theory of Computing (STOC), pp 1–9
4. Arasu A, Babu S, Widom J (2006) The cql continuous query language: semantic foundations and query execution. The VLDB Journal 15(2):121–142
5. Batcher KE (1968) Sorting networks and their applications. In: AFIPS Spring Joint Computer Conference, pp 307–314
6. Boyd-Wickizer S, Chen H, Chen R, Mao Y, Kaashoek F, Morris R, Pesterev A, Stein L, Wu M, Dai Y, Zhang Y, Zhang Z (2008) Corey: An operating system for many cores. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA
7. Chhugani J, Nguyen AD, Lee VW, Macy W, Hagog M, Chen YK, Baransi A, Kumar S, Dubey P (2008) Efficient implementation of sorting on multi-core SIMD CPU architecture. Proc VLDB Endowment 1(2):1313–1324
8. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to Algorithms, 2nd edn. MIT Press
9. DeWitt DJ (1979) DIRECT—a multiprocessor organization for supporting relational database management systems. IEEE Trans on Computers c-28(6)
10. Furtak T, Amaral JN, Niewiadomski R (2007) Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In: ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp 348–357

11. Gedik B, Bordawekar RR, Yu PS (2007) CellSort: High performance sorting on the Cell processor. In: Proc. of the 33rd Int'l Conference on Very Large Data Bases (VLDB), Vienna, Austria, pp 1286–1297
12. Gold BT, Ailamaki A, Huston L, Falsafi B (2005) Accelerating database operators using a network processor. In: Int'l Workshop on Data Management on New Hardware (DaMoN), Baltimore, MD, USA
13. Govindaraju NK, Lloyd B, Wang W, Lin M, Manocha D (2004) Fast computation of database operations using graphics processors. In: Proc. of the 2004 ACM SIGMOD Int'l Conference on Management of data, Paris, France, pp 215–226
14. Govindaraju NK, Gray J, Kumar R, Manocha D (2006) GPUTeraSort: High performance graphics co-processor sorting for large database management. In: Proc. of the 2006 ACM SIGMOD Int'l Conference on Management of Data, Chicago, IL, USA, pp 325–336
15. Greaves DJ, Singh S (2008) Kiwi: Synthesis of FPGA circuits from parallel programs. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)
16. Gschwind M, Hofstee HP, Flachs B, Hopkins M, Watanabe Y, Yamazaki T (2006) Synergistic processing in Cell's multicore architecture. IEEE Micro 26(2):10–24
17. Harizopoulos S, Shkapenyuk V, Ailamaki A (2005) QPipe: A simultaneously pipelined relational query engine. In: Proc. of the 2005 ACM SIGMOD Int'l Conference on Management of Data, Baltimore, MD, USA
18. Howard J, Dighe S, Hoskote Y, Vangal S, Finan D, Ruhl G, Jenkins D, Wilson H, Borkar N, Schrom G, Pailet F, Jain S, Jacob T, Yada S, Marella S, Salihundam P, Erraguntla V, Konow M, Riepen M, Droege G, Lindemann J, Gries M, Apel T, Henrissi K, Lund-Larsen T, Steibl S, Borkar S, De V, Wijngaart RVD, Mattson T (2010) A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In: ISCC '10: Solid-State Circuits Conference, pp 108 –109
19. Huang SS, Hormati A, Bacon DF, Rabbah R (2008) Liquid Metal: Object-oriented programming across the hardware/software boundary. In: European Conference on Object-Oriented Programming, Paphos, Cyprus
20. Inoue H, Moriyama T, Komatsu H, Nakatani T (2007) AA-Sort: A new parallel sorting algorithm for multi-core SIMD processors. In: Int'l Conference on Parallel Architecture and Compilation Techniques (PACT), Brasov, Romania, pp 189–198
21. Kickfire (2009) `http://www.kickfire.com/`
22. Knuth DE (1998) The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd edn. Addison-Wesley
23. Manegold S, Boncz PA, Kersten ML (2000) Optimizing database architecture for the new bottleneck: Memory access. The VLDB Journal 9(3):231–246
24. Mitra A, Vieira MR, Bakalov P, Tsotras VJ, Najjar W (2009) Boosting XML filtering through a scalable FPGA-based architecture. In: Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA
25. Mueller R, Eguro K (2009) FPGA-accelerated deserialization of object structures. Tech. Rep. MSR-TR-2009-126, Microsoft Research Redmond
26. Mueller R, Teubner J, Alonso G (2009) Data processing on fpgas. Proc VLDB Endowment 2(1)
27. Mueller R, Teubner J, Alonso G (2009) Streams on wires – a query compiler for FPGAs. Proc VLDB Endowment 2(1)
28. Netezza (2009) `http://www.netezza.com/`
29. Oflazer K (1983) Design and implementation of a single-chip 1-d median filter. IEEE Trans on Acoustics, Speech and Signal Processing 31:1164–1168
30. Q6700 datasheet (2007) Intel Core 2 Extreme Quad-Core Processor XQ6000 Sequence and Intel Core 2 Quad Processor Q600 Sequence Datasheet. Intel
31. Rabiner LR, Sambur MR, Schmidt CE (1975) Applications of a nonlinear smoothing algorithm to speech processing. IEEE Trans on Acoustics, Speech and Signal Processing 23(6):552–557
32. Tukey JW (1977) Exploratory Data Analysis. Addison-Wesley
33. Wendt PD, Coyle EJ, Gallagher, Jr NC (1986) Stack filters. IEEE Trans on Acoustics, Speech and Signal Processing 34(4)
34. Xilinx (2009) Virtex-5 FGPA Data Sheet: DC and Switching Characteristics. Xilinx Inc., v5.0 edn
35. Xilinx (2009) Virtex-5 FPGA User Guide. Xilinx Inc., v4.5 edn
36. XtremeData (2009) `http://www.xtremedatainc.com/`
37. Zhou J, Ross KA (2002) Implementing database operations using SIMD instructions. In: Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data, Madison, WI, USA

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Pervasive Computing

René Müller

ETH Zurich
CAB E77.2
CH-8092 Zurich

Tel. +41-44-632 8037
Fax +41-44-632 1425
E-Mail rene.mueller@inf.ethz.ch
Web http://people.inf.ethz.ch/muellren/

Editors of the VLDB Journal
Reviewers of the VLDB Journal

Zurich, 4. April 2010

Submission to VLDB Journal: "Sorting Networks on FPGAs"

Dear Editors,

this journal submission is a generalization of our VLDB 2009 publication "Data Processing on FPGAs". We believe the attached submission makes interesting and relevant contributions over the VLDB submission. The journal article focuses on the generalization of sorting networks on FPGAs while the VLDB publication described a concrete operator implemented using FPGAs. The new paper focuses on sorting networks in general and then explains how to use them to implement data processing operators of which the median operator of previous publication is now just one more use-case.

Although, this article builds on top of the VLDB 2009 publication it has undergone a significant revision. The changes are in the paper are:

- The discussion of sorting networks on FPGAs was significantly extended. In particular, differences between even-odd merging and bitonic sorting networks and their implementations are discussed (Sections 4 and 5).
- A detailed model that describes the chip utilization for the different implementations is added in Sections 5.3 and 5.4. The model prediction are evaluated in Section 6 together with the performance of the implementations.
- We added a section (5.5) that covers how sorting networks can be implemented on traditional CPUs using general-purpose registers. We use this implementation in a direct comparison with the FPGA-based solution (Section 7.4). We also provide an overview of related SIMD work and contrast it with the hardware approach. The section can easily be removed if the reviewers feel this distracts too much from the main story.
- In the evaluation of the implementation we now differentiate between the sorting network itself and the integration of the circuit with the rest of the system.
- All measurements are now based on a new Virtex-5 FPGA chip. We provide a detailed analysis of the chip resource utilization for the different sorting network implementations.
- The median operator that was used as a running example in the VLDB 2009 paper was reduced to a use-case (Section 7). This use-case describes the bus-based system integration of our custom logic. We significantly simplified the discussion of the implementation details.
- As promised in the future work in the VLDB paper we attached the sorting network core as a co-processor to the execution pipeline of the embedded PowerPC CPU. This is presented as a second use-case where we also sketch how a software/hardware co-design can look like (Section 8).

With kind regards,

René Müller, Jens Teubner, Gustavo Alonso