Data Processing on FPGAs

Rene Mueller
rene.mueller@inf.ethz.ch

Jens Teubner
jens.teubner@inf.ethz.ch

Gustavo Alonso
alonso@inf.ethz.ch

Systems Group, Department of Computer Science, ETH Zurich, Switzerland

ABSTRACT

Computer architectures are quickly changing towards heterogeneous many-core systems, where processing units of different characteristics are combined into a single platform. 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. In this paper, we explore how to program data processing operators on top of field-programmable gate arrays (FPGAs). FPGAs are very versatile in terms of how they can be used and now they can also be added as additional processing units in standard CPU sockets. As customizable hardware, however, FPGAs induce several trade-offs.

In the paper, we study how data processing can be accelerated using an FPGA. Our results indicate that efficient usage of FPGAs involves non-trivial aspects such as having the right computation model (an asynchronous 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 or parallel stream evaluation.

1. INTRODUCTION

Taking advantage of specialized hardware has a long tradition in data processing applications. Some of the earliest efforts involved building entire machines tailored to database engines [8]. More recently, graphic processing units (GPUs) are being studied as a way to efficiently implement certain types of operators [11, 12].

Parallel to these developments, computer architectures are quickly evolving towards heterogeneous many-core systems. These systems will soon have a (large) number of processors 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* (FPGA) [13]. 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 (synergetic processing elements, or SPEs).

Given that existing applications and operating systems already have significant problems when dealing with multicore systems [5], such diversity adds yet another dimension to the complex task of adapting data processing software to these new hardware platforms. Unlike in the past, it is no longer just a question of taking advantage of specialized hardware, but a question of adapting to new, inescapable architectures.

In this paper, we focus our attention on FPGAs as one of the more "different" elements that are likely to 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 exploited most efficiently. Our contribution with this work is the first deep study of the design tradeoffs encountered when using FPGAs for data processing, as well as 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 using *asynchronous* circuits. We also show that asynchronous circuits (such as sorting networks) are very well suited for common data processing operations like comparisons and sorting.

(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 [13]. 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 resource can be managed and demonstrate an efficient circuit for parallel stream processing.

(4) FPGAs will be most 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 our work, we demonstrate an embedded heterogeneous multi-core setup and identify 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 while operating 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, we describe the implementation of a *median operator* that depends on *sorting* as well as on *arithmetics*. We use it in a streaming fashion to illustrate *sliding window* functionality. The implementation we discuss in the paper is designed to illustrate the design space of FPGA-based co-processing. 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 in Section 3, we illustrate the implementation of a median operator using FPGA hardware (Section 4). Its integration into a complete multi-core system is our topic for Section 5, before we evaluate our work in Section 6. We wrap up in Section 7.

2. RELATED WORK

A number of research efforts have explored how databases can be re-architected to use the potential of modern hardware architectures. Examples include optimizations for cache efficiency (e.g., [21]) or the use of vector primitives ("SIMD instructions") in database algorithms [27]. The QPipe [14] engine exploits multi-core functionality by building an operator pipeline over multiple CPU cores. Likewise, stream processors such as Aurora [2] or Borealis [1] are implemented as networks of stream operators. An FPGA with database functionality could directly be hooked 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 [12]. The GPUTeraSort algorithm [11] 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 [17], CellSort [9], and MergeSort [6] 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.* [10]. 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 [13] or Liquid Metal [15]. Both projects aim at off-loading traditional CPU tasks to programmable hardware.

The advantage of using customized hardware as a database co-processor has already been recognized in the context of database machines. 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 [8]. Similar ideas have been commercialized recently in terms of database appliances sold by, e.g., Netezza [7], Kickfire [19], or XtremeData [16]. All of them appear to be based on specialized, hard-wired acceleration chips, which primarily provide a high degree of data parallelism. As far as we are aware, we are the first to exploit the *reconfigurability* of FPGAs at runtime and study the design trade-offs in data processing applications. 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 4 and 5. The particular operator that we use to demonstrate FPGA-based co-processing is a median over a sliding window. The implementation of a median with FPGAs has already been studied [25], but the proposed stack filters are only suited to process input where the underlying value domain is small (rather than the 32-bit integer values we assume). Our median implementation is similar to the sorting network proposed by Oflazer [22]. As we demonstrate in Section 6.1, we gain significant performance advantages by designing the network to run in an *asynchronous* mode.

3. OVERVIEW OF FPGAS

Field-programmable gate arrays are reprogrammable hardware chips for digital logic. As the name implies, 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.

3.1 FPGA Architecture

Figure 1 sketches the architecture of the Xilinx Virtex II Pro XC2VP30 FPGA used in this paper [26]. The FPGA is a 2D array of *configurable logic blocks* (CLBs). Each logic block consists of 4 *slices* that contain logic gates (in terms of lookup tables) 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 18 kbit fast storage) as well as 18×18-bit multiplier units. A number of *Input/Output Blocks* (IOBs) link to external RAM or networking devices. Two on-chip PowerPC 405 cores are directly wired to the FPGA fabric and to the BRAM compo-

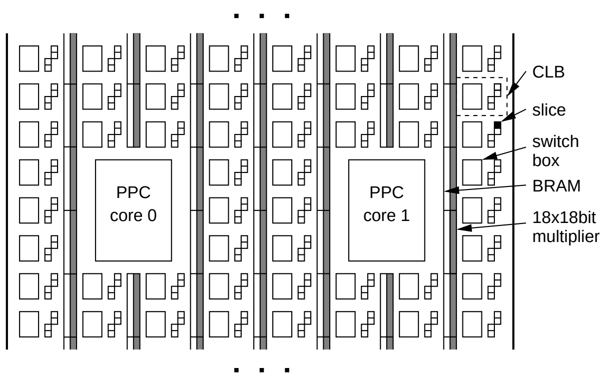


Figure 1: Simplified FPGA architecture: 2D array of CLBs, each consisting of 4 slices and a switch box. Available in silicon: 2 PowerPC cores, BRAM blocks and multipliers.

PowerPC cores	2
Slices	13,696
18 kbit BRAM blocks	136 (=2,448 kbit, usable as 272 kB)
18×18-bit multipliers	136
I/O pads	644

Table 1: Characteristics of Xilinx XC2VP30 FPGA.

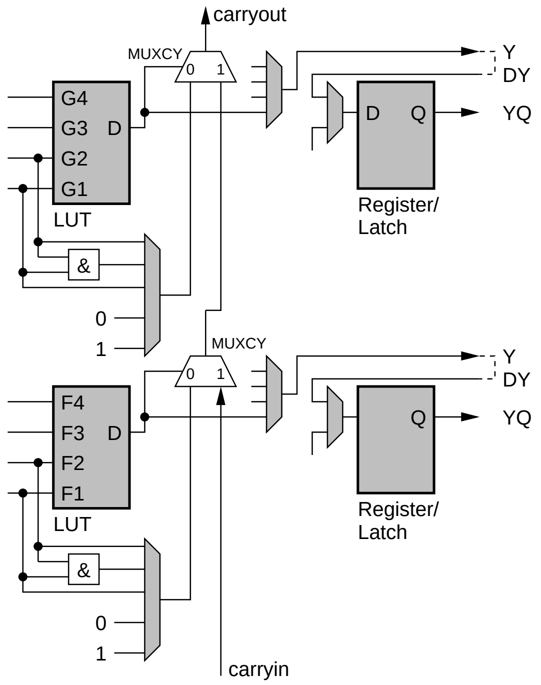


Figure 2: Simplified Virtex-II Pro slice consisting of 2 LUTs and 2 register/latch components. The gray components are configured during programming.

nents. Each PowerPC core has dedicated 16 kB data and instruction caches. The cores provide the 32-bit subset of the PowerPC architecture, but do not have any dynamic branch prediction. The execution pipeline has 5 stages and has single in-order instruction issue. Table 1 shows a summary of the characteristics of the FPGA used in this paper.

A simplified circuit diagram of a programmable slice is shown in Figure 2. Each slice contains two *lookup tables (LUTs)* with four inputs and one output each. A LUT can implement any binary-valued function with four binary-inputs or, equivalently, 16-bit memory. The output of the LUTs can be fed to a buffer block which can be configured as a register (flip-flop) or a latch. The output is also fed to a multiplexer (MUXCY in Figure 2), which allows the implementation of fast carry logic.

3.2 Hardware Setup

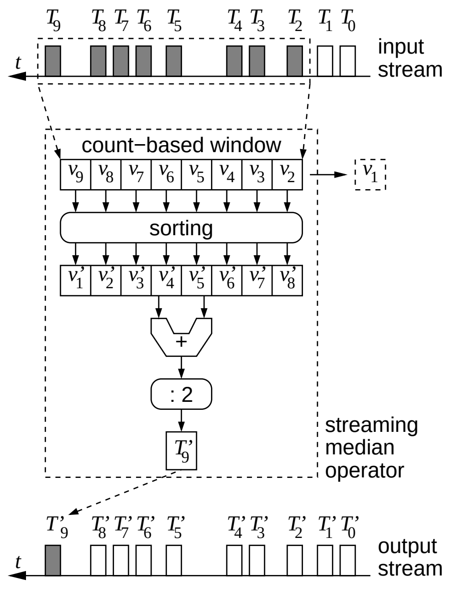
FPGAs are typically available pre-mounted on a circuit board that includes additional peripherals. Such circuit boards provide an ideal basis for the assessment we perform here. Quantitative statements in this report are based on a Xilinx XUPV2P development board with a Virtex-II Pro XC2VP30 FPGA chip. Relevant for the discussion in this paper are the DDR DIMM socket which we populated with a 512 MB RAM module. For terminal I/O of the software running on the PowerPC, a RS232 UART interface is available. A 100 Mbit Ethernet port is also present on the board.

The board is clocked at 100 MHz. This clock drives both, the FPGA-internal buses as well as the external I/O connectors, such as the DDR RAM. With a 64-bit interface, this suggests a theoretical peak bandwidth of 1600 MB/s to the DDR DIMM. The effective bandwidth that can be achieved from the PowerPC core during sequential access is significantly lower. With caches enabled, and hence burst accesses, we measured up to 107.8 MB/s. We assume that the significant difference is due to timing incompatibilities of the existing DDR-RAM controller provided by Xilinx as a soft IP-core. The PowerPC cores are clocked at 300 MHz.

4. A STREAMING MEDIAN OPERATOR

To demonstrate some of the considerations in an FPGA-based database co-processor, we have implemented an operator that covers many of the typical aspects of data intensive operations such as comparisons of data elements, sorting, and I/O issues. The design illustrates many of the design constraints in FPGAs, which are very different from the design constraints encountered in conventional database engines. For instance, parallelism in a normal database is limited by the CPU and memory available. In an FPGA, it is limited by the chip space available. In a CPU, parallel threads may interfere with each other. In an FPGA, parallel circuits do not interfere at all, thereby achieving 100 % parallelism. Similarly, algorithms in a CPU look very different from the same algorithms implemented as circuits and, in fact, they have very different behavior and complexity patterns.

We illustrate many of these design aspects using 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 [23] and in data analysis tasks [24]. For illustration purposes and to simplify the figures and the discussion, we assume a window size of 8 tuples. For an input stream S,



T_9 T_8 T_7 T_6 T_5 T_4 T_3 T_2 T_1 T_0
input stream
t
count−based window
v_9 v_8 v_7 v_6 v_5 v_4 v_3 v_2
v_1
sorting
v'_1 v'_2 v'_3 v'_4 v'_5 v'_6 v'_7 v'_8
+
: 2
T'_9
streaming median operator
T'_9 T'_8 T'_7 T'_6 T'_5 T'_4 T'_3 T'_2 T'_1 T'_0
output stream
t

Figure 3: Median aggregate over a count-based sliding window (window size 8).

the operator can then be described in CQL [3] as

$$\texttt{Select median}(v) \quad \texttt{From } S \texttt{ [Rows 8] .} \qquad (Q_1)$$

The semantics of this query are illustrated in Figure 3. 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 $v'_1 \leq \cdots \leq v'_8$ and

(2) compute the *mean value* between v'_4 and v'_5, $\frac{v'_4+v'_5}{2}$, to construct the output tuple T'_i (for an odd-sized window, the median would instead be the middle element of the sorted sequence).

We will shortly see how the data flow in Figure 3 directly leads to an implementation in FPGA hardware. Before that, we discuss the algorithmic part of the problem for Step (1).

4.1 Sorting

Sorting is the critical piece in the median operator and known to be particularly expensive on conventional CPUs. It is also a common data processing operation that can be very efficiently implemented in FPGAs using asynchronous circuits. Highly tuned and vectorized software implementations require in the order of fifty cycles to sort eight numbers on modern CPUs [6]. We are going to show how to do this in far less cycles within an FPGA.

Sorting Networks. Interestingly, all of the efficient CPU-based solutions use sorting algorithms that are also the preferred choice for an implementation in hardware. *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). On modern CPUs, sorting networks suggest the use of vector primitives, which has been demonstrated in [9, 11, 17].

Figure 4 illustrates two different networks that sort eight input values. Input data enters a network at the left end. As the data travels to the right, *comparators* $\updownarrow$ each exchange two values, if necessary, to ensure that the larger value always leaves a comparator at the bottom. The *bitonic merge* network (Figure 4(a)) is based on a special property of bitonic sequences (i.e., those that can be obtained by concatenating two monotonic sequences). A component-wise merging of two such sequences always yields another bitonic sequence, which is efficiently brought into monotonic (i.e., sorted) order afterward.

In an *even-odd merge* sorting network (Figure 4(b)), an input of 2^p values is split into two sub-sequences of length 2^{p-1}. After the two 2^{p-1}-sized sequences have been sorted (recursively using even-odd merge sorting), an *even-odd merger* combines them into a sorted result sequence. Other sorting algorithms can be represented as sorting networks, too. For details we refer to the work of Batcher [4] or a textbook [20].

Sorting Network Properties. As can be seen in the two example networks in Figure 4, the number of comparisons required for a full network implementation depends on the particular choice of the network. The bitonic merge sorter for $N = 8$ inputs in Figure 4(a) uses 24 comparators in total, whereas the even-odd merge network (Figure 4(b)) can do with only 19. For other choices of N, we listed the required number of comparators in Table 2.

The graphical representation in Figure 4 indicates another important metric of sorting networks. Comparators with independent data paths can be grouped into processing stages and evaluated in parallel. The number of necessary stages is referred to as the *depth* $S(N)$ of the sorting network. For eight input values, bitonic merge networks and even-odd merge networks both have a depth of six.

Compared to even-odd merge networks, bitonic merge networks observe two additional interesting characteristics:

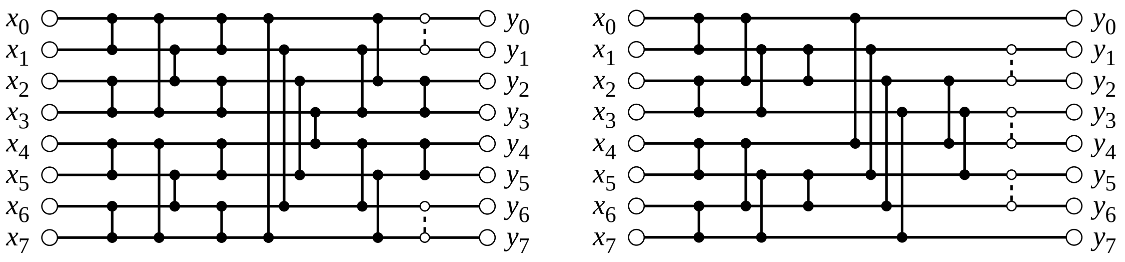
(i) all signal paths have the same length (by contrast, the data path from x_0 to y_0 in Figure 4(b) passes through three comparators, whereas from x_5 to y_5 involves six) and

(ii) the number of comparators in each stage is constant (4 comparators per stage for the bitonic merge network, compared with 2–5 for the even-odd merge network).

CPU-Based Implementations. These two properties are the main reason why many successful implementations of sorting have opted for a bitonic merge network, despite its higher comparator count (e.g., [9, 11]). Differences in path lengths may require explicit *buffering* for those values that do not actively participate in comparisons at specific processing stages. At the same time, additional comparators might cause no additional cost in architectures that can evaluate a number of comparisons in parallel using for instance the SIMD instruction sets of modern CPUs.

4.2 An FPGA Median Operator

Composition of elements in an FPGA is straightfoward. Once the element for sorting is implemented using a sorting network, the complete operator can be implemented in



(a) Bitonic merge sorting network

(b) Even-odd merge sorting network

Figure 4: Sorting networks for 8 elements. Dashed comparators are not used for the median.

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

Table 2: Comparator count $C(N)$ and depth $S(N)$ of different sorting networks.

an FPGA using the sketch in Figure 3. Each of the solid arrows corresponds to 32 wires in the FPGA interconnect fabric, carrying the binary representation of a 32-bit integer number. Sorting and mean computation can both be packaged into logic components, whose internals we now present.

Comparator Implementation on an FPGA. The data flow in the horizontal direction of Figure 4 also translates into wires on the FPGA chip. The entire network is obtained by wiring a set of comparators, each implemented in FPGA logic. The semantics of a comparator is easily expressible in the hardware description language VHDL (where `<=` indicates an 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 resulting logic circuit is shown in Figure 5. 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 $a \geq b$. Signal c drives 2×32 multiplexers that connect the proper input lines to the output lines for $\min(a, b)$ and $\max(a, b)$ (lower half of the circuit). Equality comparisons $\boxed{=}$ and multiplexers $\boxed{}$ each occupy one lookup table on the FPGA, resulting in a total space consumption of 48 FPGA slices for each comparator.

The FPGA implementation in Figure 5 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.

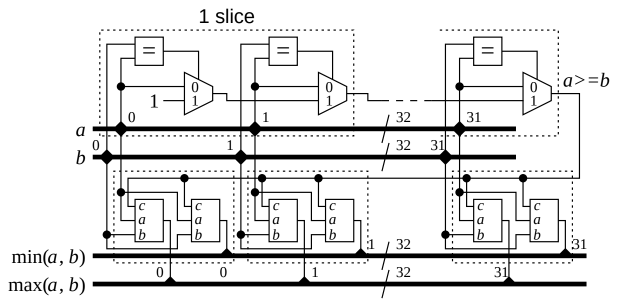


Figure 5: FPGA implementation of a 32-bit comparator. Total space consumption is 48 slices (16 to compare and 32 to select minimum/maximum values).

The Right Sorting Network for FPGAs. To implement a full *bitonic merge* sorting network, 24 comparators need to be plugged together as shown in Figure 4(a), resulting in a total space requirement of 1152 slices (or 8.4 % of the space of our Virtex-II Pro chip). An *even-odd merge* network (Figure 4(b)), by contrast, can do the same work with only 19 comparators, which amount to only 912 slices ($\approx$ 6.7 % of the chip). Available slices are the scarcest resource in FPGA programming. The 20 % savings in space, therefore, makes even-odd merge networks preferable over bitonic merge sorters on FPGAs. The *runtime performance* of an FPGA-based sorting network depends exclusively on the depth of the network (which is the same for both networks).

Optimizing for the Median Operation. Since we are only interested in the computation of a median, a fully

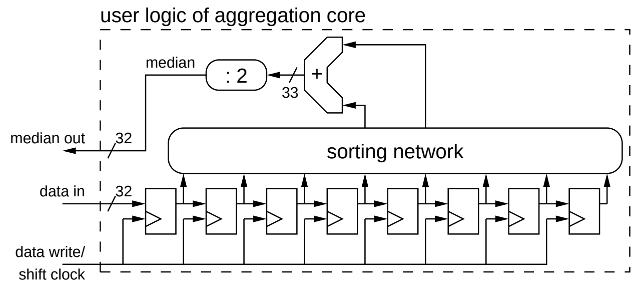


Figure 6: Sliding window implementations as 8 × 32 linear shift register.

sorted data sequence is more than required. Even with the dashed comparators in Figure 4 omitted, the average over y_3 and y_4 will still yield a correct median result.

This optimization saves 2 comparators for the bitonic, and 3 for the even-odd sorting network. Moreover, the even-odd-based network is now shortened by a full stage, reducing its execution time. The optimized network in Figure 4(b) now consumes only 16 comparators, i.e., 768 slices or 5.6 % of the chip.

Averaging Two Values in Logic. To obtain the final median value, we are left with the task of averaging the two middle elements in the sorted sequence. The addition of two integer values is a classic example of a digital circuit and, for 32-bit integers, consists of 32 full adders. To obtain the mean value, the 33-bit output must be divided by two or—expressed in terms of logic operations—bit-shifted by one. The bit shift, in fact, need not be performed explicitly in hardware. Rather, we can connect the upper 32 bits of the 33-bit sum directly to the operator output.

Overall, the space consumption of the mean operator is 16 slices (two adders per slice).

Sliding Windows. The sliding window of the median operator is implemented as a 32-bit wide linear shift register with depth 8 (see Figure 6). The necessary 8 × 32 flip-flops occupy 128 slices (each slice contains two flip-flops).

5. 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.

5.1 System Overview

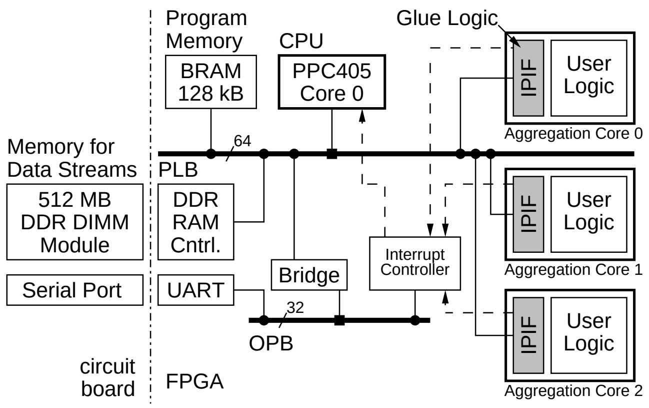


Figure 7: Architecture of the on-chip system: PowerPC core, 3 aggregation cores, BRAM for program, interface to external DDR RAM and UART for terminal I/O.

Using the Virtex-II Pro-based development board described in Section 3.2, we have implemented the embedded system shown in Figure 7. To simplify matters, we only use one of the two available PowerPC cores (our experiments indicate that the use of a second CPU core would not lead to improved throughput). The system further consists of two buses of different width and purpose. The 64-bit wide *processor local bus* (PLB) is used to connect memory and fast peripheral components (such as network cards) to the PowerPC core. The 32 bit-wide *on-chip peripheral bus* (OPB) is intended for slow peripherals, to keep them from slowing down fast bus transactions. The two buses are connected by a bridge. The driver code executed by the PowerPC core (including code for our measurements) is stored in 128 kB block RAM connected to the PLB.

Two soft IP-cores provide controller functionality to access external DDR RAM and a serial UART connection link (RS-232). They are connected to the input/output blocks (IOBs) of the FPGA chip. We equipped our system with 512 MB external DDR RAM and used a serial terminal connection to control our experiments.

Our streaming median operator participates in the system inside a dedicated processing core, dubbed "aggregation core" in Figure 7. As we will elaborate in the experimental part of this work, more than one instance of this component can be created at a time, all of which are connected to the PLB. An aggregation core consists of user logic, as described in detail in the previous section. 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. A similar IPIF component with the same interface on the user-logic side is also available for the OPB. However, since we aim for high data throughput, we chose to attach the aggregation cores to the faster PLB.

5.2 Putting it all together

While the above description is very much focused on the unavoidable hardware aspects of FPGAs, it has important consequences for data processing operations. Many opera-

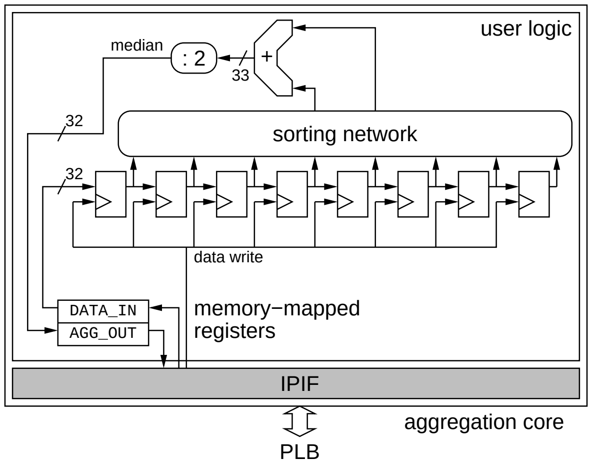


Figure 8: Attachment of aggregation core through memory-mapped registers.

tors involve frequent iteration over the data; data transfers to and from memory; and data acquisition from the network or disks. As in conventional databases, these interactions can completely determine the overall performance. It is thus of critical importance to design the memory/CPU/circuits interfaces so as to optimize performance.

To illustrate the design options and the trade-offs involved, we consider three configurations (attachments of the aggregation core to the CPU) of the FPGA. These configurations are based on registers connected to the input signals of the IP-core and mapped into the memory space of the CPU. Information can then be sent between the aggregation core and the CPU using load/store instructions.

Configuration 1: Slave Registers. The first approach uses two 32-bit registers DATA_IN and AGG_OUT as shown in Figure 8. The IP interface is set to trigger a clock signal upon a CPU write into the DATA_IN register. This signal causes a shift in the shift register (thereby pulling the new tuple from DATA_IN) and a new data set to start propagating through the sorting network. A later CPU read instruction for AGG_OUT then will read out the newly computed aggregate value.

This configuration is simple and uses few resources. However, it has two problems: lack of synchronization and poor bandwidth usage.

In this configuration the CPU and the aggregation core are accessing the same registers concurrently with no synchronization. The only way to avoid race conditions is to add artificial time delays between the access operations.

In addition, each tuple in this configuration requires two 32-bit memory accesses (one write followed by one read). Given that the CPU and the aggregation core are connected to a 64-bit bus (and hence could transmit up to 2×32 bits per cycle), this is an obvious waste of bandwidth.

Configuration 2: FIFO Queues. The second configuration we explore solves the lack of synchronization by introducing *FIFO queues* between the CPU and the aggregation core (Figure 9). Interestingly, this is the same solution as the one adopted in data stream management systems to decouple operators.

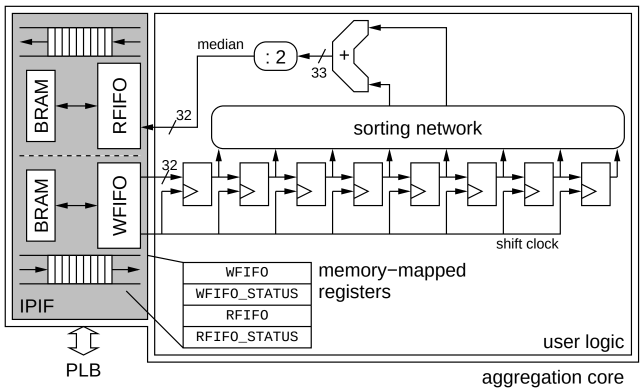


Figure 9: Attachment of aggregation core through Write-FIFO and Read-FIFO queues.

The CPU writes tuples into the Write-FIFO queue (WFIFO) and reads median values from the Read-FIFO queue (RFIFO). The two queues are implemented in the IPIF using additional block RAM components (BRAM). The aggregation core independently dequeues items from the Write-FIFO queue and enqueues the median results into the Read-FIFO queue. Status registers in both queues allow the CPU to determine the number of free slots (write queue) and the number of available result items (read queue).

This configuration avoids the need for explicit synchronization. There is still the drawback that the interface uses only 32 bits of the 64 available on the bus. The *mismatch* between a 64-bit access on the CPU side and a 32-bit width on the aggregation core turns out to be an inherent problem of using a general-purpose FIFO implementation (such as the one provided with the Xilinx IPIF interface). Re-implementing the FIFO functionality in user logic can remedy this deficiency, as we describe next.

Configuration 3: Master Attachment. In the previous configuration, access is through a register than cannot be manipulated in 64-bit width. Instead of using a register through a bus, we can use memory mapping between the aggregation core and the CPU to achieve a full 64 bit transfer width. The memory mapping is now done on the basis of contiguous regions rather than a single address. Two regions are needed, one for input and one for output. These memory regions correspond to local memory in the aggregation core and are implemented using BRAMs.

We can improve on this approach even further by taking advantage of the fact that the transfers to/from these regions can be offloaded to a DMA controller. We have considered two options: one with the DMA controller run by the CPU and one with the DMA controller run in (the IPIF of) the aggregation core. Of these two options, the latter one is preferrable since it frees the DMA controller of the CPU to perform other tasks. In the following, we call this configuration *master attachment*. In Figure 10, we show all the memory mapped registers the CPU uses to set up the transfers, although we do not discuss them here in detail for lack of space. The figure also shows the interrupt line used to notify the CPU that new results are available.

The master attachment configuration has the advantage that the aggregation core can independently initiate the

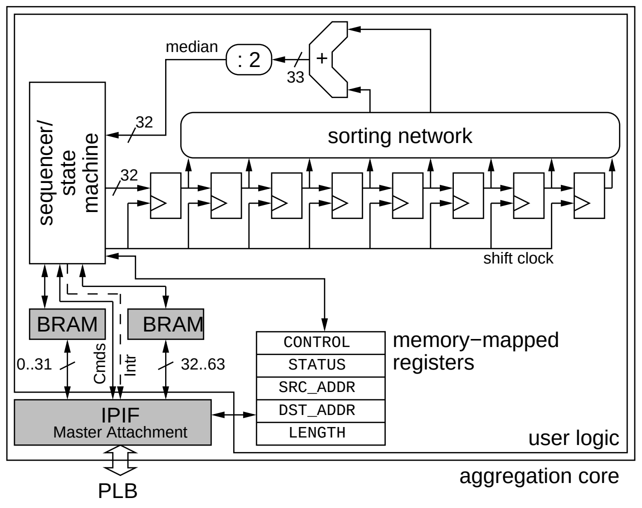


Figure 10: Master attachment of aggregation core supporting through DMA transfers to external memory.

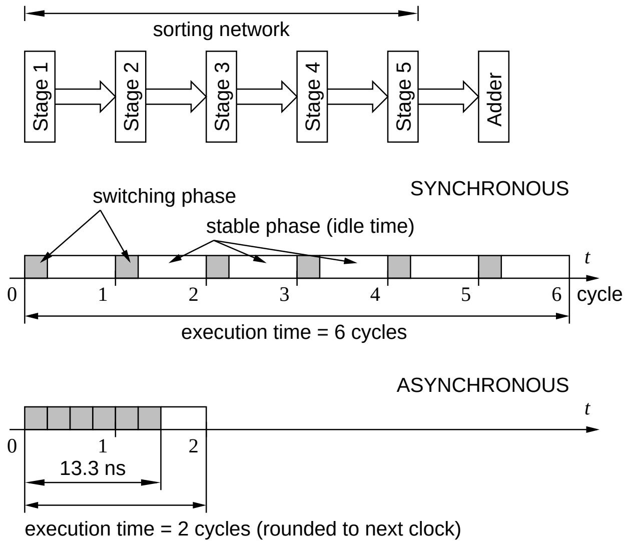


Figure 11: Synchronous implementation of the aggregation core requires 6 clock cycles, i.e., 60 ns. In an asynchronous implementation the output is ready after 13.3 ns (the output signals can be read after 2 cycles).

write-back of results once they are ready, without having to synchronize with an external DMA controller. This reduces latency, uses the full available bandwidth, and gives the aggregation core control over the flow of data, leaving the CPU free to perform other work and thereby increasing the chances for parallelism.

6. EVALUATION

We now proceed to evaluate the different design options we have described in previous sections to ascertain their practical impact when processing data. All experiments have been done on the Xilinx XUPV2P development board. Our focus is on the details of the soft IP-core and we abstract from effects caused for example by I/O (network and disks) by performing all the processing into and out of off-chip memory (512 MB DDR RAM).

6.1 Asynchronous vs. Synchronous Designs

We first consider and evaluate possible implementations of the sorting network discussed in Section 4.1. As indicated, the even-odd merge network is more space efficient so this is the one we consider here. The implementation options are important in terms of the overall latency of the operator which, in turn, will determine how fast data streams can be processed.

Asynchronous design. We start by considering an asynchronous design. The eight 32-bit signals are applied at the input of the sorting network and then ripple down the stages of the sorting network. Until the correct result has stabilized at the output, signals have to traverse up to five comparator stages. The exact latency of the sorting network, the *signal propagation delay*, depends on the implementation of the comparator element and on the on-chip routing between the comparators.

The total propagation delay is determined by the *longest signal path.* For a single comparator, this path starts in the equality comparison LUT, passes through 32 carry logic multiplexers, and ends at one *min*/*max* multiplexer. According to the FPGA data sheet [26] the propagation delay for a single 4-input LUT is 0.28 ns. The carry logic multiplexers and the switching network cause an additional delay. The overall latency for the median output to appear after the input is set can be computed with a simulator provided by Xilinx that uses the post-routing and element timing data of the FPGA.[1]

For our implementation we obtain a latency of 13.3 ns. An interesting point of reference is the performance of a tuned SIMD implementation on current CPU hardware. It has been suggested that 50 CPU cycles is the minimum required to sort 8 elements on a modern general-purpose CPU [6]. For a fast 3.22 GHz processor, this corresponds to ≈ 15 ns, 13 % more than the FPGA used in our experiments. The short latency is a consequence of a deliberate design choice. Our circuit operates in a strictly *asynchronous* fashion, *not* bound to any external clock.

Synchronous design. In a traditional *synchronous* implementation all circuit elements use a common clock. Registers are then necessary between each of the five stages of the sorting network.

A synchronous implementation of the sorting network in Section 4 inherently uses six clock cycles (i.e., 60 ns in a 100 MHz system) to sort eight elements.

Both design choices are illustrated in Figure 11. In this figure, the gray-shaded time intervals indicate switching phases during which actual processing happens (i.e., when signals are changing). During intervals shown in white, signals are stable. The registers are used as buffers until the next clock

[1]One might be tempted to physically measure the latency of the sorting network by connecting the median operator directly to the I/O pins of the FPGA. However, signal buffers at the inputs and outputs (IOBs) of the FPGA and the switching network in between add significant latency (up to 10 ns). Any such measurement is bound to be inaccurate.

cycle. As the figure shows, the switching phase is shorter than the clock length.

Comparison. The latency of the asynchronous design is 13.3 ns. Taking into consideration that the sorting network needs to be connected to other elements that are asynchronous, the effective latency is 2 clock cycles or 20 ns. The latency of the synchronous design is 60 ns or 6 cycles, clearly slower than the asynchronous circuit. On the other hand, the synchronous circuit has a throughput of one tuple per cycle while the asynchronous circuit has a throughput of 1 tuple every 2 cycles. The synchronous implementation requires more space due to the additional hardware (flip-flops) necessary to implement the registers between the comparator stages. The space needed is given by:

$$(5\,\text{stages} \times 8\,\text{elements} + 1\,\text{sum}) \times 32\,\text{bits} =$$
$$1312\,\text{flip-flops/core} \equiv 5\% \text{ of the FPGA/core}\ .$$

The higher complexity of asynchronous circuits has led many FPGA design to rely solely on synchronous circuits [13]. Our results indicate, however, that for data processing there are simple asynchronous designs that can significantly reduce latency (at the cost of throughput). In terms of transforming algorithms into asynchronous circuits, not all problems can expressed in an asynchronous way. From a theoretical point of view, every problem where the only dependence of the output signal are the input signals, can be converted into an asynchronous circuit (a *combinatorial* circuit). The necessary circuit can be of significant size, however (while synchronous circuits may be able to re-use the same logic elements in more than one stage). A more practical criterion can be obtained by looking at the algorithm that the circuit mimics in hardware. As a rule of thumb, algorithms that require a small amount of *control logic* (branches or loops) and have a simple *data flow* pattern are the most promising candidates for asynchronous implementations.

6.2 Median Operator

We now compare two of the configurations discussed in Section 5.2 and then evaluate the performance of the complete aggregation core using the best configuration.

We compare configuration 1 (slave register) with configuration 3 (master attachment). We use maximum-sized DMA transfers (4 kB) between external memory and the FPGA block RAM to minimize the overhead spent on interrupt handling. We do not consider configuration 2 (FIFO queues) because it does not offer a performance improvement over configuration 1.

Figure 12 shows the execution time for streams of varying size up to 64 kB. While we see a linearly increasing execution time for configuration 1, configuration 2 requires a constant execution time of 96 μs for all data sizes up to 4 kB, then scales linearly with increasing data sizes (this trend continues beyond 64 kB). This is due to the latency incurred by every DMA transfer (up to 4 kB can be sent within a single transfer). 96 μs are the total round-trip time, measured from the time the CPU writes to the control register in order to initiate the Read-DMA transfer until it receives the interrupt.

These results indicate that configuration 1 (slave registers) is best for processing small amounts of data or streams with low arrival rates. Configuration 3 (master attachment) is

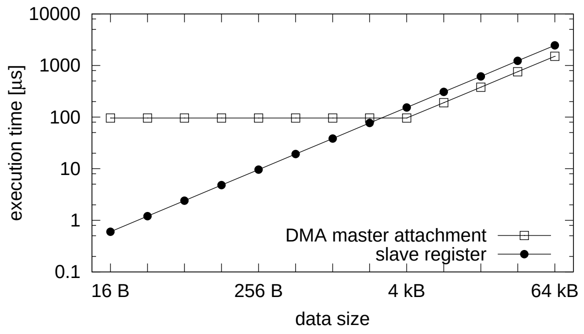


Figure 12: Total execution time to process data streams of different size on the FPGA-based aggregation core.

best for large amounts of data (greater than 4 kB) or data streams with very high arrival rates so that the tuples can be batched.

Using configuration 3, we have also measured the time it takes for the complete median operator to process 256 MB of data consisting of 4-byte tuples. It takes 6.173 seconds to process all the data at a rate of more than 10 million tuples per second. This result is shown as the horizontal line in Figure 13.

6.3 FPGA performance in perspective

As mentioned, 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 when running on a single data stream. Later on we are going to consider parallelism.

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. The eighth is an implementation of the even-odd merge sorting network of Section 4.1 using CPU registers.

We ran the different algorithms on several hardware platforms. We used an off-the-shelf desktop 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 13 shows the wall-clock time obtained when processing 256 MB (as 32-bit tuples) through the median sliding window operator. The horizontal line indicates the execution time of the FPGA implementation. Timings for the merge, quick, and heap sort algorithms on the embedded PowerPC core did not fit into scale (303 s, 116 s, and 174 s,

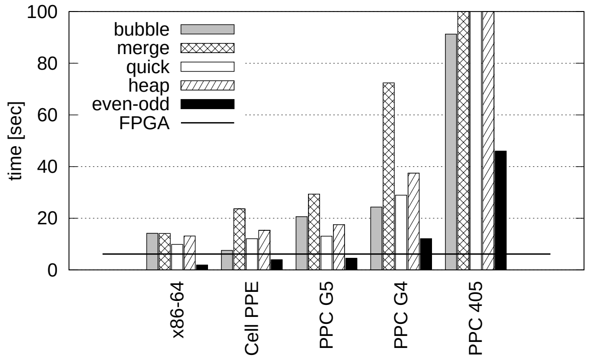


Figure 13: Execution time for processing a single 256 MB data set on different CPUs using different sorting algorithms and on the FPGA.

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 performance observed indicates that the implementation of the operator on the FPGA is comparable to that of conventional CPUs. In the cases where it is worse, it is not significantly slower. Therefore, 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.

6.4 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 the 3.2 GHz x86-64.

It is difficult to reliably measure the power consumption of an isolated chip. Instead, we chose to list some ballpark figures in Table 3. 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") [18]. For the FPGA, a power analyzer provided by Xilinx reports an estimated consumption of 1.3 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 3, the FPGA has a 12-fold advantage (8.3 W over 102 W) compared to the CPU-based solution here.

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 3 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.

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 XUPV2P development board:	
Calculated power estimate (FPGA only)	1.3 W
Measured total power (230 V)	8.3 W

Table 3: Power consumption of an Intel Q6700-based desktop system and the Xilinx XUPV2P FPGA board used in this paper. Measured values are under load when running median computation.

cores	flip-flops	LUTs	slices	%
0	1761	1670	1905	13.9 %
1	3727	6431	4997	36.5 %
2	5684	10926	7965	58.2 %
3	7576	15597	11004	80.3 %
4	9512	20121	13694	100.0 %

Table 4: FPGA resource usage. The entry for 0 cores represents the space required to accommodate all the necessary circuitry external to the aggregation cores (UART, DDR controller, etc.).

6.5 Parallelism: Space Management

Another advantage of FPGAs is their inherent support for parallelism. By instantiating multiple aggregation cores in FPGA hardware, multiple data streams can be processed truly in parallel. The number of instances that can be created is determined both by the size of the FPGA, i.e., its number of slices, and by the capacity of the FPGA interconnect fabric.

We placed four instances of the median aggregation core on the Virtex-II Pro. Table 4 shows the resource usage depending on the number of aggregation cores. We also give the usage in percent of the total number of available slices (13,696). Note that there is a significant difference in size between the space required by the median operator (700 to 900 slices) and the space required by the complete aggregation core (about 3000 slices). This overhead comes from the additional circuitry necessary to put the median operator into the configuration 3 discussed above.

The use of parallelism brings forth another design trade-off characteristic of FPGAs. To accommodate four aggregation cores, the VHDL compiler starts trading latency for space by placing unrelated logic together into the same slice, resulting in longer signal paths and thus longer delays. This effect can also be seen in Figure 14, where we illustrate the space occupied by the four aggregation cores. Occupied space regions are not contiguous, which increases signal path lengths.

The longer path lengths have a significant implication for asynchronous circuits. Without any modification, the median operator produces incorrect results. The longer signal

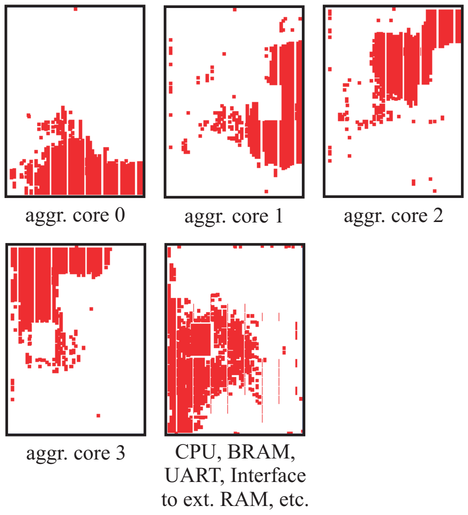


Figure 14: Resource usage on the FPGA by the 4 aggregation cores and the remaining system components.

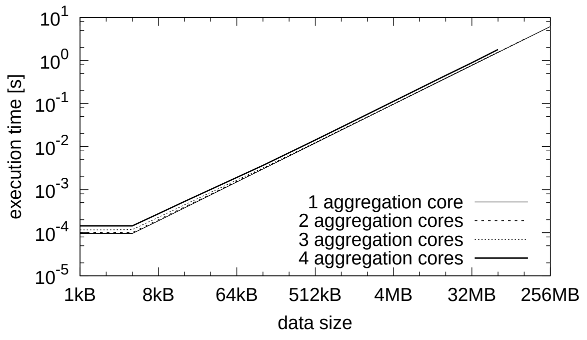


Figure 15: Total execution time to process multiple data streams using concurrent aggregation cores.

paths result in longer switching phases in the sorting network, leading to an overall latency of more than two cycles (20 ns). Incorrect data reading can be avoided by introducing a *wait cycle* and reading the aggregation result three cycles after setting the input signals. This implies that asynchronous circuits need to be treated more carefully if used in high density scenarios where most of the FPGA floor space is used.

Other FPGA models such as the Virtex-5 have significantly larger arrays (7.6 times larger than our Virtex-II Pro) and higher clocks (5.5 times). On such a chip, assuming that a single core requires 3,000 slices, we estimate that ≈ 30 aggregation cores can be instantiated, provided that the memory bandwidth does not further limit this number.

6.6 Parallelism: Performance

We used the four aggregation cores mentioned above to run four independent data streams in parallel. We ran streams of increased size over configurations with an increasing amount of cores. Figure 15 shows the wall-clock execution times for processing multiple data streams in parallel, each on a separate aggregation core. Table 5 summarizes the execution times for a stream of 64 MB.

streams	FPGA	PowerPC 405 seq.	PowerPC 405 alt.	speedup seq.	speedup alt.
1	1.54 s	10.1 s	–	7×	–
2	1.56 s	20.2 s	36.7 s	13×	24×
3	1.58 s	30.4 s	55.1 s	19×	35×
4	1.80 s	40.5 s	73.5 s	22×	41×

Table 5: Execution times for different number of concurrent streams (64 MB data set per stream).

The first important conclusion is that running additional aggregation cores has close to no impact on the other cores. The slight increase with the addition of the fourth core comes from the need to add the wait cycle mentioned above. The second observation is that the execution times scale linearly with the size of the data set as it is to be expected. The flat part of the curves is the same effect observed before for stream sizes smaller than 4 kB. The graph also indicates that since each core is working on a different stream, we are getting linear scale-out in throughput with the number of aggregation cores. It is also interesting to note that with four cores we did not reach the limit in memory bandwidth, neither on the DDR RAM nor on the PLB.

One last question that remains open is whether a similar parallelism could be achieved with a single CPU. Table 5 contains the execution times obtained with a CPU-only implementation for multiple streams, assuming either sequential processing (one stream after the other) or tuple-wise alternation between streams. Cache conflicts lead to a significant performance degradation in the latter case.

Clearly, a single CPU cannot provide the same level of parallelism as an FPGA. Obviously, this could be achieved with more CPUs but at a considerable expense. From this and the previous results, we conclude that FPGAs offer a very attractive platform as data co-processors and that they can be effectively used to run data processing operators.

7. SUMMARY

In this paper we have assessed the potential of FPGAs as co-processor for data intensive operations in the context of multi-core systems. Through an example based on a median operator, we have illustrated the type of data processing operations where FPGAs have performance advantages (through parallelism and low latency) and discuss several ways to embed the FPGA into a larger system so that the performance advantages are maximized. 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 and cost effective manner.

As part of future work we intend to explore a tighter integration of the FPGA with the rest of the computing infrastructure, an issue also at the top of the list for many FPGA manufacturers. Modern FPGAs can directly interface to high-speed bus systems, such as the HyperTransport bus, or even intercept the execution pipeline of general-purpose CPUs, opening up many interesting possibilities for using the FPGA in different configurations.

8. REFERENCES

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2005.

[2] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2), July 2003.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2), June 2006.

[4] Kenneth E. Batcher. Sorting Networks and Their Applications. In *AFIPS Spring Joint Computer Conference*, 1968.

[5] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, December 2008.

[6] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *Proc. VLDB Endowment*, 1(2), 2008.

[7] Netezza Corp. `http://www.netezza.com/`.

[8] David DeWitt. DIRECT—A Multiprocessor Organization for Supporting Relational Database Management Systems. *IEEE Trans. on Computers*, c-28(6), June 1979.

[9] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellSort: High Performance Sorting on the Cell Processor. In *Proc. of the 33rd Int'l Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, September 2007.

[10] Brian T. Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. Accelerating Database Operators Using a Network Processor. In *Int'l Workshop on Data Management on New Hardware (DaMoN)*, Baltimore, MD, USA, June 2005.

[11] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 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, June 2006.

[12] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast Computation of Database Operations Using Graphics Processors. In *Proc. of the 2004 ACM SIGMOD Int'l Conference on Management of data*, Paris, France, 2004.

[13] David Greaves and Satnam Singh. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[14] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *Proc. of the 2005 ACM SIGMOD Int'l Conference on Management of Data*, Baltimore, MD, USA, June 2005.

[15] Shan Shan Huang, Amir Hormati, David Bacon, and Rodric Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *European Conference on Object-Oriented Programming*, Paphos, Cyprus, July 2008.

[16] Xtreme Data Inc. `http://www.xtremedatainc.com/`.

[17] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. 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, September 2007.

[18] Intel Corp. *Intel Core 2 Extreme Quad-Core Processor XQ6000 Sequence and Intel Core 2 Quad Processor Q600 Sequence Datasheet*, August 2007.

[19] Kickfire. `http://www.kickfire.com/`.

[20] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 2nd edition, 1998.

[21] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3), December 2000.

[22] Kemal Oflazer. Design and Implementation of a Single-Chip 1-D Median Filter. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 31, October 1983.

[23] L. Rabiner, M. Sambur, and C. Schmidt. Applications of a Nonlinear Smoothing Algorithm to Speech Processing. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 23(6), December 1975.

[24] John W. Tukey. *Exploratory Data Analysis.* Addison-Wesley, 1977.

[25] Peter D. Wendt, Edward J. Coyle, and Neal J. Gallagher, Jr. Stack Filters. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 34(4), August 1986.

[26] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, v4.2 edition, 2007.

[27] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations using SIMD Instructions. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, Madison, WI, USA, June 2002.