Hardware-Accelerated Stream Processing Using FPGAs

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 streaming 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 use the example of a median filter applied to a sliding window to study how stream 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 significant performance benefits can be obtained by extending conventional engines with the option of implementing operators directly on an FPGA, particularly operators that are CPU-bound.

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 [6]. More recently, graphic processing units (GPUs) are being studied as a way to efficiently implement certain types of operators [9, 10].

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. For instance, some will have full instruction sets, others will have reduced or specialized instruction sets, not all of them will use the same clock frequency or exhibit the same 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) [11]. An example of such heterogeneous system is the Cell Broadband Engine, which contains, in addition to a general-purpose core, multiple special execution cores (synergetic processing elements SPEs).

Given that existing applications and operating systems already have significant problems when dealing with multi-core systems [4], such diversity adds yet another dimension to the complex tasks of adapting data processing software to these new hardware platforms. Unlike in the past, it is no longer 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. As such, in addition to the potential increase in parallelism offered by one more core, they open up the possibility of implementing data processing operators directly in hardware. As a first step in determining how to take advantage of FPGAs, in this paper we study how to implement an apparently simple data stream operator (median) over a small sliding window. We focus on this deceptively simple example to be able to study in detail the trade-offs induced by the use of FPGAs. The contributions of the paper are empirically tested solutions to four key design trade-offs in the use of FPGAs:

(1) FPGAs have clock frequencies that are lower than those of conventional CPUs. In this paper, we show how they can provide high efficiency nonetheless if their circuits are designed to run in an *asynchronous* execution mode, outside the regular system clock.

(2) Asynchronous designs are notoriously more difficult to design than synchronous ones. This has led to a preference for synchronous circuits in studies of FPGA usage [11]. By 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) The usefulness of an FPGA as a database co-processor hinges on its *integration* with the remainder of the system. We demonstrate a working heterogeneous multi-core setup and identify trade-offs in FPGA integration design.

In a detailed experimental study, we let our FPGA prototype compete face to face against general-purpose CPUs and see how a single-threaded low-cost FPGA compares favorably with performance figures published recently for high-end CPUs [5]. Our implementation further provides linear scale-out with respect to additional threads.

Outline. We start our work by setting the context with related work (Section 2). After introducing necessary technical background in Section 3, we illustrate the implementation of a complete database streaming 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., [16]) or the use of vector primitives ("SIMD instructions") in database algorithms [22]. The QPipe [12] 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 networks CPUs, which already 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 [10]. The GPUTeraSort algorithm [9] 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 [14], CellSort [7], and MergeSort [5] 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.* [8]. 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, e.g., the Kiwi [11] and Liquid Metal [13] projects. 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. DeWitt's DIRECT system, e.g., comprises of a number of query processors whose instruction sets embrace common database tasks such as join or aggregate operators [6].

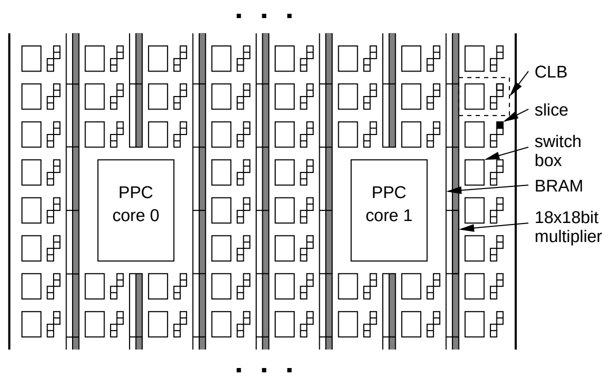


Figure 1: Simplified FPGA architecture: 2D array of CLBs, each consisting of 4 slices and a switch box. Available in silicon: 2 PowerPC cores, BRAM blocks and multipliers.

Database machines ultimately failed purely for economical reasons: with significant up-front costs, the design of customized chips could not compete with the racing clock speeds of general-purpose CPUs. By contrast, our work strives to exploit commodity and widely available components. While we configure FPGA chips for a specific application, the hardware remains general-purpose itself.

FPGAs are being used very successfully in the signal processing domain 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, the computation of a median, has been addressed, e.g., by [20]. The proposed stack filters, however, 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 [17]. 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 (HDL) 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 shows a simplified architecture of the Xilinx Virtex-II Pro XC2VP30 FPGA used in this paper [21]. The FPGA is a 2D array of *configurable logic blocks* (CLBs). Each logic block consists of 4 *slices* that contains the logic gates and a switch box that connects slices to the FPGA interconnect fabric.

In addition to the CLBs, FPGA manufacturers provide frequently-used functionality as discrete silicon components

PowerPC cores	2
Slices	13,696
18 kbit BRAM blocks	136 (=2,448 kbit, usable as 272 kB)
18x18-bit multipliers	136
I/O pads	644
Release year	2002

Table 1: Characteristics of Xilinx XC2VP30 FPGA.

(*hard IP-cores*). Such hard IP-cores include *block RAM* (BRAM) elements (each containing 18 kbit fast storage) as well as 18x18-bit multiplier units. A number of *Input/Output Blocks* (IOB) link to external RAM or networking devices. Two on-chip PowerPC 405 cores are directly wired to the FPGA fabric and to the BRAM components. Each PowerPC core has dedicated 16 kB data and instruction caches. The caches are intended to speed up accesses to external memory of long latency. 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 typically come pre-mounted on circuit board that includes additional peripherals. Quantitative statements in this work are based on a Xilinx XUPV2P development board with a Virtex-II Pro XC2VP30 FPGA chip. Despite being on the market since six years already, this model is still widely used for evaluation and production use. 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 higher at 300 MHz.

Latest FPGAs allow significantly higher clock rates and provide higher bandwidth to external memory. To account for the technology gap, we compare our hardware implementation against the CPU solution on a processor of the same

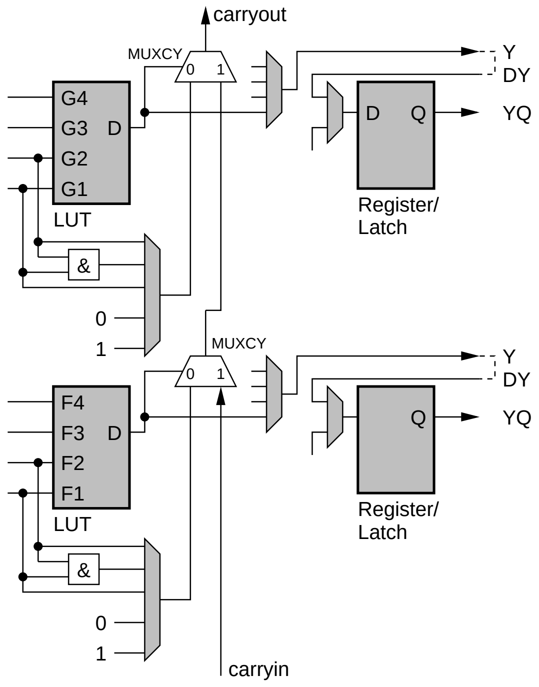


Figure 2: Simplified Virtex-II Pro slice consisting of 2 LUTs and 2 register/latch components. The gray components are configured during programming.

age (the PowerPC core on the FPGA), as well as to modern desktop CPUs. Together, these comparisons give an idea of the potential of FPGA-accelerated data processing. Starting from promising results we can see already, future, faster FPGA models will increase the performance benefit.

4. A STREAMING MEDIAN OPERATOR

To demonstrate some of the considerations in an FPGA-based database co-processor, we prototyped an operator that is simple, yet touches interesting aspects of hardware-accelerated data processing and has useful applications. We used the aforementioned Xilinx development board to implement functionality that could be found in a typical stream processing problem: the computation of a *median* over a count-based *sliding window* (for illustration purposes, we assume a window size of 8 tuples). For an input stream S, such an operator can be described in CQL as

```
Select median(v)
  From S [ Rows 8 ] .                    (Q1)
```

Possible applications of such functionality include, e.g., the elimination of non-Gaussian random noise in sensor readings [18] or data analysis tasks [19].

The semantics of Query Q_1 is 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 or-

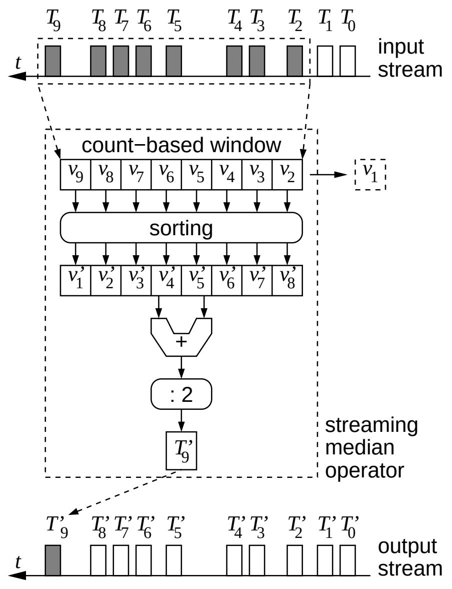


Figure 3: Median aggregate over a count-based sliding window (window size 8).

dered 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 be the middle element of the sorted sequence instead).

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 the 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. Even highly tuned and vectorized implementations [5] require in the order of fifty CPU cycles to sort eight numbers on modern CPUs.

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, e.g., in [7, 9, 14].

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 always the larger value leaves a comparator at the bottom. The *bitonic merge* network (Figure 4(a)) is based on a special property of bitonic sequences (i.e., ones that can be obtained by concatenating two monotonic sequences). A component-wise merging of two such sequences always yields another bitonic sequence, easy to bring 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 [3] or a textbook (e.g., [15]).

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, data paths to output y_0 in Figure 4(b) pass three comparators, whereas those ending in y_3 pass six),

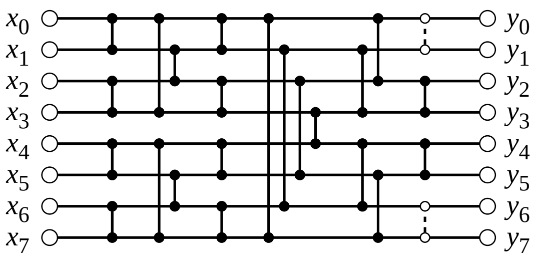
(ii) the number of comparators in each stage is constant (4 comparators per stage for the bitonic merge network, compared to 2–4 for the even-odd merge network).

CPU-Based Implementations. The two properties are the main reason why many successful implementations have opted for a bitonic merge network, despite its higher comparator count (e.g., [7, 9]). 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, e.g., using the SIMD instruction sets of modern CPUs.

4.2 An FPGA Median Operator

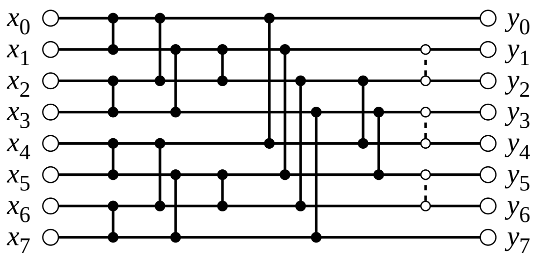
The data flow in Figure 3 can be used to build an implementation in FPGA logic. Each of the solid arrows then corresponds to 32 wires in the FPGA interconnect fabric, carrying the binary representation of a 32-bit integer number. Sorting and mean computation can both be packaged into logic components, whose internals we now look into.

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:



x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7

(a) Bitonic merge sorting network



x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7

(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.

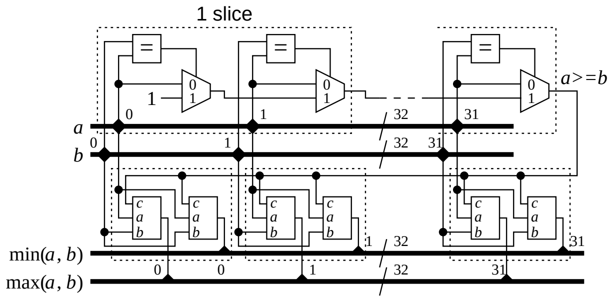


Figure 5: FPGA implementation of a 32-bit comparator. Total space consumption is 48 slices (16 to compare and 32 to select minimum/maximum values).

```
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 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.

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 exclusively depends 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 sorted data sequence is actually more than what we need. 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 us 2 (3) comparators for the bitonic (even-odd) merge sorting network (respectively). Moreover, the even-odd-based network is now shortened by a full stage, reducing the overall execution time of the network. 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

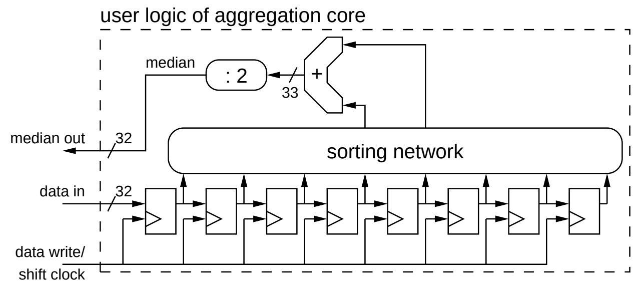


Figure 6: Sliding window implementations as 8 × 32 linear shift register.

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. To function as a co-processing unit of an actual database setup, this component now has to be wired into a system architecture that includes more traditional units such as general-purpose CPUs, memory controllers, or I/O functionality.

The resulting architecture can be built as an embedded system inside the FPGA chip by using the built-in PowerPC CPUs and connecting it to vendor-provided soft IP-cores that implement communication buses or controller components for various purposes. As we will see in a moment, such a system already reveals important design choices that may have a critical impact on the effectiveness of FPGA co-processing.

5.1 System Overview

Using the Virtex-II Pro-based development board described in Section 3.2, we 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 throughput improvements). 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, thereby preventing them to slow down fast bus transactions. The two buses are connected over 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

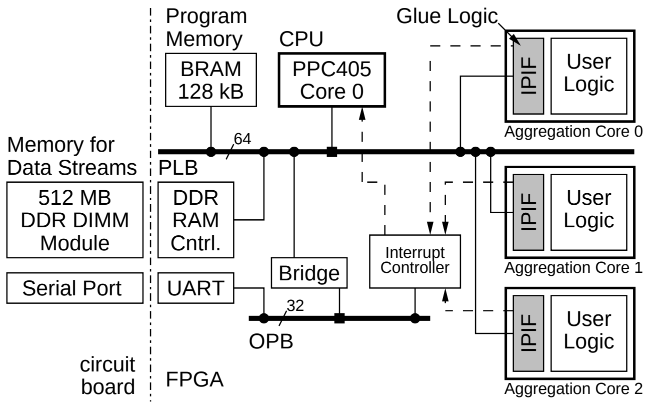


Figure 7: Architecture of the on-chip system: PowerPC core, 3 aggregation cores, BRAM for program, interface to external DDR RAM and UART for terminal I/O.

(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, which are all connected to the PLB. An aggregation core consists of a 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. Since we aim for high data throughput, we chose to attach the aggregation cores to the faster PLB, however.

5.2 Designing a Co-Processor Interface

While the PLB is a rather straightforward choice to establish a fast communication line between the controlling PowerPC core and the hardware-accelerated streaming operator, it is unclear *how* this bus can be used to achieve high bandwidth and low latency. Four attachment strategies are conceivable whose trade-offs turn out to be caused by rather technical system aspects.

All four protocols described in the following make use of registers that are connected to input signals of the IP-core, then mapped into the memory space of the CPU. Information can then be sent from/to the aggregation core by use of CPU load/store instructions.

Method 1: Slave Registers. A very simple 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 starts propagating through the sorting network. A later CPU read instruction for `AGG_OUT` then will read out the newly computed aggregate

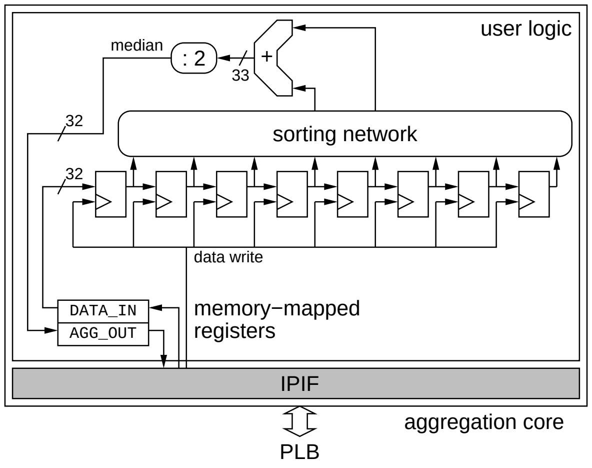


Figure 8: Attachment of aggregation core through memory-mapped registers.

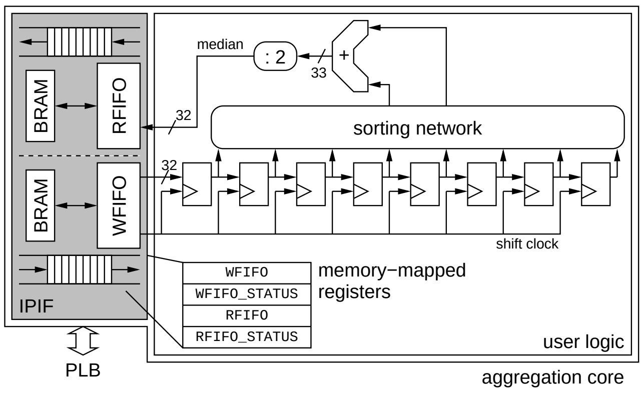


Figure 9: Attachment of aggregation core through Write-FIFO and Read-FIFO queues.

value.

To ensure reading the correct aggregate result, the CPU read operation must occur no earlier than the total computation time, i.e., the combined delay for the shift, sort, and arithmetic operations. To this end, the CPU store and load instructions need to be separated far enough to guarantee proper result reading. One way to enforce such separation is to manipulate the acceleration core's handling of the bus protocol and delay the acknowledge signal for a CPU load until the data is available. A serious side effect is that this delay will result in a complete stall of the embedded CPU (even including non-maskable interrupts).

Each tuple in this setup requires two 32-bit memory accesses to process (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. In addition, successive access to memory-mapped registers requires explicit synchronization, e.g., using the `eioeio` ("enforce in-order execution of I/O") instruction, which introduces additional wait cycles. On the plus side, the slave register is a low-overhead protocol whenever data actually must be processed tuple-by-tuple. If this is not strictly necessary, we can use batching to optimize bandwidth, which inspires our following protocol candidates.

Method 2: FIFO Queues. To decouple the CPU and the aggregation core, i.e., CPU read and write operations from the median computation, the implementation in Figure 9 uses *FIFO queues*. Existing streaming systems use queues in a similar fashion to decouple operators. The CPU can write one or more tuples into the Write-FIFO queue (WFIFO) and independently read median values from the Read-FIFO queue (RFIFO).

The two queues are implemented in the IPIF using additional block RAM components (see Figure 9). Read operations are now consuming, i.e., a CPU load instruction reads and dequeues an item from the RFIFO queue. The aggregation core independently dequeues items from the Write-FIFO queue and enqueues the median result in 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 computed result items (read queue). The CPU is in charge to properly enqueue and dequeue data elements (using the status information provided).

The use of queues avoids the need to explicitly synchronize I/O requests. On the back side, the interface still only uses 32 bits of the available 64-bit bandwidth on the bus. The *mismatch* between a 64-bit access on the CPU side and a 32-bit semantics on the aggregation core side 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.

Method 3: Slave Attachment. By exposing the internal memory structure of the FIFO implementation to the PowerPC CPU, the latter is enabled to send data to the aggregation core in chunks of arbitrary size. Such exposition can be achieved by *memory mapping* the block RAMs into the address space of the CPU. By allocating a sufficient number of BRAM components (two BRAMs with 32 bit word size each) and accessing them in parallel, the full PLB bandwidth can be used to provide the aggregation core with data. To notify the co-processor about new data in the input queue, the CPU writes into a dedicated `CONTROL` register.

An even more efficient way of filling the input queue (or reading the output queue) is then the use of *DMA* (direct memory access). In a *slave attachment* setting, the CPU can instruct its own bus controller to directly move data to/from the aggregation core-internal BRAM. By using *burst access* mechanisms of the PLB, two input tuples can be sent to/from the aggregation core in every bus clock cycle.

Method 4: Master Attachment. Instead of initiating DMA transfers from the CPU side, we can also implement the aggregation core as PLB *master attachment* (shown in Figure 10). In this setup, the aggregation core is responsible for initiating payload data transfers, controlled by *work units* that the CPU writes into memory-mapped configuration registers (`CONTROL` through `LENGTH` in Figure 10). During work unit processing, the CPU is kept free to perform other work.

A work unit consists of a source and destination address in the external memory (`SRC_ADDR` and `DST_ADDR` registers), as well as the number of input tuples at the source address

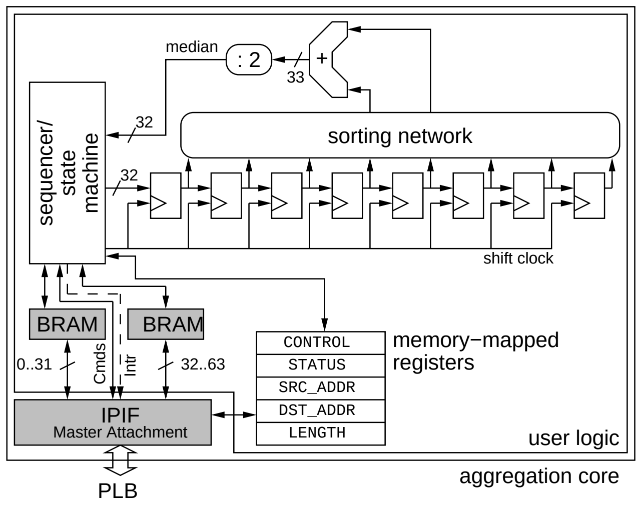


Figure 10: Master Attachment of aggregation core supporting through DMA transfers to external memory.

(register `LENGTH`). Once the CPU has stored the work unit configuration into the aggregation core registers, it sets the `CONTROL` bit to initiate processing. After successful processing of the full unit, the aggregation core sets the `STATUS` bit and triggers a CPU *interrupt* to notify the CPU of the completion of the unit.

An interrupt handler in the CPU code then identifies the source of the interrupt (more than one aggregation core can be instantiated, see Section 6.4) and sets up the next work unit to process the user input stream.

6. EVALUATION

We evaluate the median aggregation core on our Xilinx XUPV2P development board. Because we only focus on the details of the soft IP-core we abstract from effects caused for example by I/O (network and disks). We directly place the buffers for the data streams in the external memory (512 MB DDR RAM). The input stream is initialized with pseudo-random data (uniformly distributed 32-bit values). The result stream is also written back to external memory. The output data is verified using a software implementation of the median operator on the CPU.

We first analyze the impact of different implementation approaches for the median operator, particularly the performance gains that result from an unclocked implementation. Then, we evaluate the complete aggregation core embedded in the system, including effects of the IPIF and the CPU, and compare it with an implementation in software. Finally, we show the performance implications and resource usage when instantiating multiple aggregation cores on a single FPGA.

6.1 Asynchronous vs. Synchronous Designs

To judge the potential of FPGA-based hardware acceleration, we start by looking at the characteristics of an isolated sliding window operator as discussed in Section 4. 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 of the operator, signals have to traverse up to five comparator stages, the main workers that determine the latency of the sorting network. The exact latency, 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 [21] 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 the simulator returns a latency of 13.3 ns. An interesting point of reference is the performance of a tuned SIMD implementation on current CPU hardware. The numbers in [5] indicate a minimum requirement of around 50 CPU cycles to sort 8 elements on a modern general-purpose CPU. For a fast 3.22 GHz processor, this corresponds to $\approx$ 15 ns, 13 % more than our six-year old FPGA. The latency of newer FPGA chips is significantly lower.

Asynchronous Design. 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. In a traditional *synchronous* implementation, by contrast, all circuit elements operate according to a common clock. A set of registers is then required in-between each of the five stages of the sorting network.

A synchronous implementation of the sorting network of Section 4 inherently uses six clocks (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 cycle.

The switching phase is shorter than the clock length. In fact, the length of the longest switching phase determines the maximum system clock. On our board, the maximum system clock is determined by other peripheral IP components (provided by Xilinx) which prevent us from increasing the system clock much above 100 MHz, even though the synchronous sorting network itself could be operated at higher clock rates. Since the signal propagation latency of the asynchronous sorting network implementation (13.3 ns) is slightly larger than a single bus cycle (10 ns), correct values can be read from the output signals after two bus cycles.

In addition to the longer processing time, the synchronous implementation requires additional hardware (flip-flops) to implement the registers between the comparator stages.

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

[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.

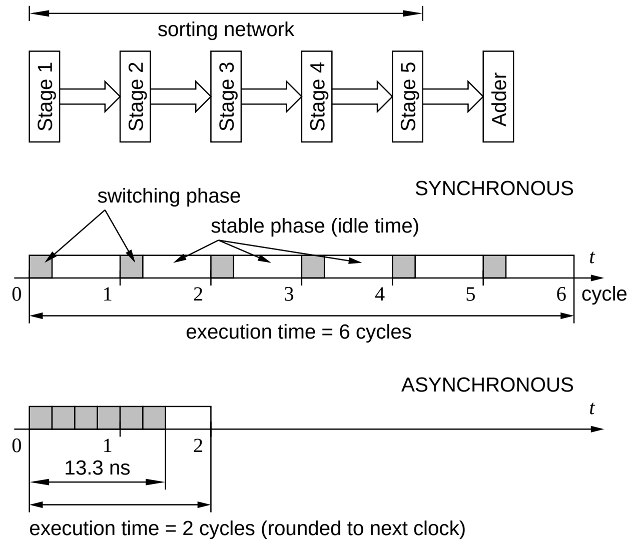


Figure 11: Synchronous implementation of the aggregation core requires 6 clock cycles, i.e., 60 ns. In asynchronous implementation output is ready after 13.3 ns. Therefore, the output signals can be read after 2 cycles.

On the positive side, a synchronous implementation typically leads to a higher issue rate (one tuple per clock cycle, i.e., every 10 ns compared to every 13.3 ns in the asynchronous case). In asynchronous circuits, this can be compensated with multiple instances of the same functional unit.

Design Guidelines. In general, an asynchronous circuit is the design of choice. However, faced with the higher complexity of such circuits, many research projects (e.g., [11]) still keep their hands off asynchronous designs and focus on synchronous circuits instead.

Not all problems can expressed in an asynchronous way. From a theoretical point of view, any pure function, i.e., 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 good asynchronous implementations.

6.2 Single Aggregation Core

We now provide an evaluation of the complete aggregation core, i.e., the combination of user logic and the IPIF. Based on the discussion in Section 5.2, we chose the master attachment method, with a DMA controller on the aggregation core. We use maximum-sized DMA transfers (4 kB) between external memory and the FPGA block RAM to minimize the overhead spent on interrupt handling.

Figure 12 shows the execution time for processing data sets up to a size of 64 kB. Up to a data size of 4 kB, the processing of the full stream requires a constant execution time of 96 μs, then scales linearly with increasing data size (this trend continues beyond 64 kB). The constant execution time 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. This obviously renders the DMA approach unsuitable to process individual tuples. For that case, the slave register method would require only 45 processor cycles or 150 ns. This is the time we measured for a single round-trip between CPU and aggregation core (i.e., a data write followed by a result read).

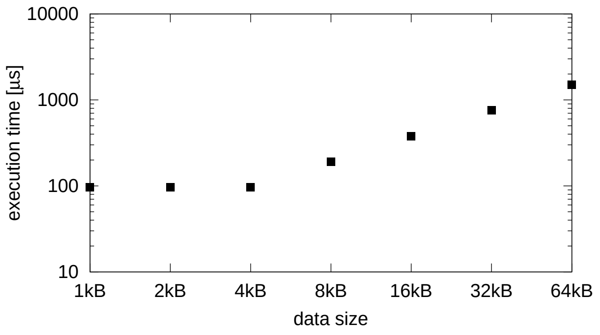


Figure 12: Total execution time to process data streams of different size on the FPGA-based aggregation core.

6.3 CPU-based Implementation

In order to assess the effective gain obtained by using an FPGA-based co-processor in a heterogeneous multi-core setup, we compare our embedded solution with an implementation that is based on a general-purpose CPU only. To ensure a fair comparison, we 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 used two different hardware architectures for the comparison, Intel x86-64 and PowerPC. Neither of them 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 pieces of assembly code implement the comparator operation for PowerPC and x86-64 processors:

$$[r8, r9] := [\min(r8, r9), \max(r8, r9)] \quad .$$

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. We already identified branch-less algorithms to be

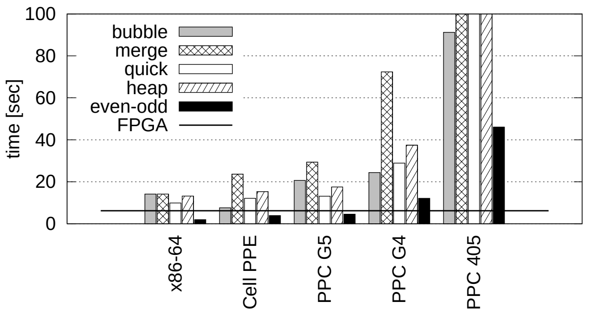


Figure 13: Execution time for processing a single 256 MB data set on different CPUs using different sorting algorithms and on the FPGA.

good candidates for asynchronous FPGA circuits. 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.

Configuration. We ran the different algorithms on several hardware configurations. We used an off-the-shelf desktop Intel x86-64 CPU (2.66 GHz Intel Core2 quad-core Q6700, introduced in 2007) and the following PowerPC CPUs: a 1 GHz G4 (MCP7457, introduced in 2003) and a 2.5 GHz G5 Quad (970MP, introduced in 2005), the PowerPC element (PPE not SPEs, introduced in 2006) of the Cell, and the embedded 405 core of our FPGA (introduced in 2002). All implementations are single-threaded. For illustration purpose, we limit our discussion to the most relevant subset of algorithms.

We also measured the performance of the hardware-accelerated implementation. As before, we opted for the use of master attachment, with DMA memory transfers initiated by the aggregation core.

CPU-Only Results. Figure 13 shows the wall-clock time we observed for processing 256 MB (as 32-bit tuples) through the median sliding window operator. The horizontal line indicates the execution time of the FPGA-accelerated system. 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, respectively).

Observe that the race is not won by any of the classic algorithms. Rather, the CPU-based even-odd merge network excels on all hardware architectures we considered. The good asymptotic complexity of, e.g., heap and merge sort, is not of significance here. In fact, bubble sort shows better performance than merge sort, despite the lower asymptotic complexity of the latter ($O(N^2)$ for bubble sort vs. $O(N \log N)$ for merge sort).

Comparison CPU ↔ FPGA. To process the 256 MB stream, the FPGA implementation requires a total of 6.173 s. Given that we saw a very low signal propagation delay for the isolated median operator circuit in Section 6.1, the FPGA-accelerated system compares rather poorly to the performance of the modern CPUs. In particular, the x86-64 and Cell PPE systems are 3.2 and 1.6 times faster, respectively.

Extrapolating from the latency measured for the isolated median circuit (13.3 ns), one might expect a total runtime $\lesssim 1$ s. This discrepancy is due to the DMA latency, which we already identified as a bottleneck in Section 6.2. Improving on the interfacing with FPGA logic is part of our current research agenda.

The systems in Figure 13 represent a technology time frame of five years, with the FPGA being at the older end. A much more meaningful comparison, therefore, is with the CPU core on the FPGA itself. This core and the FPGA use the same 130 nm fabrication process, the principal factor that determines performance. The FPGA aggregation core outperforms the PowerPC 405 implementation by a factor of seven.

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 3: FPGA resource usage.

6.4 Parallel Aggregation Cores

An advantage of FPGA technology is its 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.

Resource Usage. On the Virtex-II Pro we managed to instantiate four instances of our median aggregation core before we ran out of chip space. Table 3 shows the resource usage depending on the numbers of cores. We also give the usage in percent of the total number of available slices (13,696). The micro-computer system without any aggregation itself uses 13.9 % of the available logic cells for the soft IP-core peripherals (UART, DDR controller, etc.). Each core then adds roughly 3000 slices (22 %). This is almost three times as much as the slice counts we concluded for the median operator in Section 4.2 ($768 + 16 + 128 = 912$ slices).

The remaining space is occupied by the IPIF soft IP-core. In particular, the master attachment that we use contains the full logic of a DMA controller. The IPIF actually consumes the lion's share of the overall logic of the aggregation core.

Note that the 22 % space usage per aggregation core only adds up for up to three instances (an extrapolation to four would yield slightly more than 100 %). In fact, to accommodate four aggregation cores on the chip, the VHDL compiler software has to give up its primary optimization goal: the minimization of latency. It 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 illustrated the space occupation of the four aggregation cores. Occupied space regions are not contiguous, which increases signal

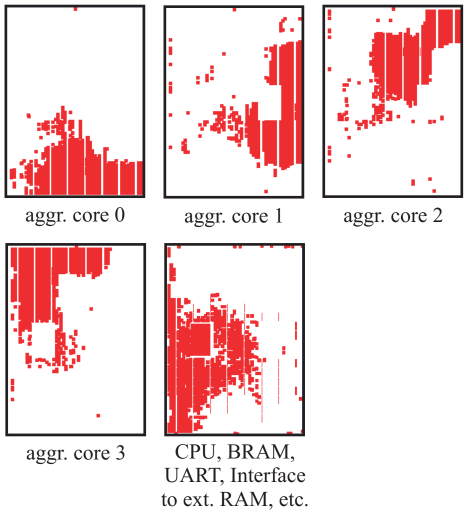


Figure 14: Resource usage on the FPGA by the 4 aggregation cores and the remaining system components.

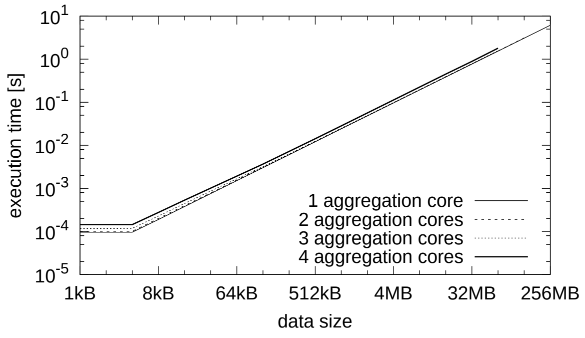


Figure 15: Total execution time to process multiple data streams using concurrent aggregation cores.

path lengths.

For four cores, we observed incorrect result data from the median operator if read after two bus cycles as described in Figure 11. The longer signal 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 another *wait cycle* and reading the aggregation result after three cycles after setting the input signals.

Current FPGAs 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 $\approx$ 30 aggregation cores can be instantiated, provided that the memory bandwidth does not further limit this number.

Performance Evaluation. We used the four cores to run four independent data streams in parallel. Figure 15 shows the wall-clock execution times for processing multiple data streams in parallel, each on a separate aggregation core. Execution times still scale linearly with the size of the data set. Table 4 summarizes the total execution times for a 64 MB

		PowerPC 405		speedup	
streams	FPGA	seq.	alt.	seq.	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 4: Execution times for different number of concurrent streams (64 MB data set per stream).

stream size.

Again, we observe a constant execution time for stream sizes smaller then the 4 kB DMA transfer unit. The constant increases from 96 μs for one aggregation core to 144 μs for four cores. This is mainly due to contention in the interrupt handling by the CPU. There is no significant increase in execution time when going from one to three aggregation cores. We observe a linear scale-out in throughput with the number of aggregation cores.

To accommodate the fourth aggregation core, we had to introduce a wait cycle. This wait cycle is responsible for the gap in execution time when scaling from three to four aggregation cores (1 cycle for each of the 16 mio data tuples amounts to 0.17 ms in total). Nevertheless, with four cores we did not reach a limit in memory bandwidth, neither on the DDR RAM nor on the PLB.

For comparison, Table 4 also contains 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.

7. SUMMARY

Our work assessed the potential of using programmable hardware, namely field-programmable gate arrays, for database processing. By prototyping a simple database streaming scenario, we demonstrated that hardware-accelerated operator implementations can achieve performance characteristics comparable to those obtained in modern high-end CPUs, but by using low-cost hardware only.

To reach this performance, however, the FPGA implementation needs to be engineered with care. We particularly emphasized the importance of an *asynchronous* operator design. The inherent parallelism of FPGA hardware can be leveraged to achieve linear scale-out with respect to data volume or number of streams. This scalability is only limited by the available *chip space*, a resource that can easily be extended by using more or larger chips.

Our experiments indicate that it is not easy to unleash the potential of the isolated FPGA circuit, once it has been integrated into a real, serviceable system. The necessity of a tight integration was lately also recognized by chip manufacturers. Modern FPGAs can directly interface to high-speed bus systems, such as the AMD HyperTransport bus, or even intercept the execution pipeline of general-purpose CPUs, opening up an whole new design space. As part of our future work, we explore the opportunities and limitations for data stream processing that come with this design space.

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

[4] 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.

[5] 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.

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

[7] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellSort: High Performance Sorting on the Cell Processor. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, Vienna, Austria, September 2007.

[8] 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.

[9] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, Chicago, IL, USA, June 2006.

[10] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast Computation of Database Operations Using Graphics Processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, Paris, France, 2004.

[11] David Greaves and Satnam Singh. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *FCCM '08: IEEE Symposium on Field-Programmable Custom Computing Machines*, 2008.

[12] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, Baltimore, MD, USA, June 2005.

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

[14] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, Brasov, Romania, September 2007.

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

[16] 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.

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

[18] 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.

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

[20] 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.

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

[22] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations using SIMD Instructions. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, Madison, WI, USA, June 2002.