Parallel Computation of Skyline Queries

Louis Woods Gustavo Alonso
Systems Group, Dept. of Computer Science
ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

Jens Teubner
DBIS Group, Dept. of Computer Science
TU Dortmund University, Germany
jens.teubner@cs.tu-dortmund.de

***Abstract*—Due to stagnant clock speeds and high power consumption of commodity microprocessors, database vendors have started to explore massively parallel co-processors such as FPGAs to further increase performance. A typical approach is to push simple but compute-intensive operations (*e.g.*, pre-filtering, (de)compression) to FPGAs for acceleration. In this paper, we show how a significantly more complex operation—the computation of the *skyline*—can be holistically implemented on an FPGA. A *skyline* query computes the *pareto optimal* set of multi-dimensional data points. These queries have been studied in software extensively over the last decade but this paper is the first to examine skyline computation in hardware. We propose a methodology that interleaves *data storage* and *computation*, allowing multiple operations to be executed on the same working set in parallel, while accounting for all data dependencies. Our experiments show that we achieve very promising results compared to CPU-based solutions.**

***Keywords*-FPGA, database, pareto optimal, skyline query**

I. Introduction

Recently, a number of projects have suggested to exploit FPGAs for database processing, *e.g.* [1], [2], [3]. On the commercial side, so-called appliances such as [4], [5] successfully use FPGAs to both improve performance and save energy. However, while FPGAs provide high aggregated compute power, it is often difficult to turn their inherent parallelism into true performance for a given database task. Thus, the state-of-the-art is to push only relatively simple operations (*e.g.*, projection/selection-based filtering, (de)compression) to configurable hardware and let commodity CPUs take care of the remaining processing, which is the case, *e.g.*, in IBM/Netezza's data warehouse appliance [4]. Unfortunately, this approach tends to leave much of the true hardware potential unused.

A *skyline* query [6] is a good example of a complex database task that could greatly benefit from hardware acceleration due to its compute-intensive nature. Yet, as we will see, it is not at all obvious how to implement skyline computation on an FPGA in an efficient way. Skyline queries reduce large multi-dimensional data sets to smaller sets of interest by eliminating items that are *dominated* by others, *i.e.*, by computing the set of *pareto optimal* items. Skyline queries are relevant in several areas, *e.g.*, search pruning, decision making, and personalized services. Furthermore, they are related to several other well-known problems such as *convex hull*, *top-K* queries, and *nearest-neighbor* search.

The classical example of a two-dimensional skyline query, is a search for hotels that are cheap and close to the beach. Hotels that are more expensive *and* further away from the beach are referred to as *dominated* and do not need to be further inspected by the user. Since any hotel could potentially dominate any other hotel, there exist data dependencies across the entire data set. This is a challenge, in particular, for an implementation on an FPGA because it requires keeping track of a potentially large state, but on-chip storage resources are limited on FPGAs.

Contributions. We present a solution for solving skyline queries on FPGAs that can handle an arbitrary number of dependencies and has no restrictions on the size of intermediate results. In our approach, it is sufficient to keep a small *working set* of skyline candidate tuples inside the FPGA (with the actual size determined by the available FPGA resources), while the rest of the input tuples are treated as a *data stream* that propagates through the FPGA. We use *pipeline-parallelism* and *nearest neighbor communication* for concurrent manipulation of the active *working set*, combining data organization, computational power, and synchronization into a parallel processing model that naturally leverages the characteristics of FPGAs.

Our solution exhibits high *throughput* and very good *scalability*. In our experiments, we show that throughput scales linearly with the amount of FPGA resources allocated. Using a low-end Virtex-5 FPGA, we clearly outperform a single-threaded CPU-based skyline operator and achieve performance close to the fastest known parallel implementation [7], running on a high-performance 64-core server.

II. Skyline Queries

In this section, we will define skyline queries, a popular software algorithm to solve skyline queries (the BNL algorithm [6]), and our modified version of BNL for parallel execution on an FPGA. Our intention, here, is to picture our approach of parallelizing BNL on a high level, before we discuss technical details later. To do so, we will take the liberty of digressing into the world of Lemmings[1].

A. The Lemming Skyline

Lemmings are primitive creatures that go on migrations in masses. On Lemmings Planet every year a challenge takes

[1]As in the video game "Lemmings": http://www.dmadesign.org/

place among the Lemmings with the goal to identify the "best" Lemmings. Every Lemming has different skills: some are very strong but slow and clumsy, others are agile but neither strong nor fast, then again others are generalists that do not have a particular skill that they are best in but have multiple skills they are pretty good in. As the committee of the competition could not agree on a weighting function that would determine the best Lemmings, all Lemmings that are not *dominated* (cf. Definition 1) by any other Lemming are considered best. In other words, the winners are the (*pareto optimal*) Lemmings that are part of the Lemming *skyline* (cf. Definition 2).

Definition 1. *A Lemming l_i dominates ($\prec$) another Lemming l_j iff every skill (dimension) of l_i is better or equal than the corresponding skill of l_j and at least one skill of l_i is strictly better than the corresponding skill of l_j.*

Definition 2. *Given a set of Lemmings $L = \{l_1, l_2, \dots l_n\}$, the skyline query returns a set of Lemmings S, such that any Lemming $l_i \in S$ is not dominated by any other Lemming $l_j \in L$.*

B. The Competition—1st Year (Best)

When the competition took place for the first time, the committee did have a formal definition for the set of best Lemmings but it was still unclear how to determine this set. Thus, in the absence of sophisticated logistic means, one committee member suggested the following simple algorithm. Initially, all Lemmings queue up in front of a bridge, as illustrated in Figure 1.

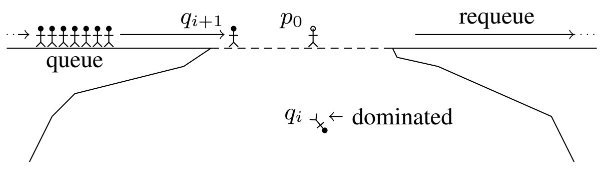

Figure 1. Lemming skyline with *Best* [8].

The first Lemming in the queue q_0 is considered a *potential* skyline Lemming p_0 and can advance onto the bridge. There, the *candidate* Lemming has to battle all other Lemmings in the queue $q_1 \dots q_{n-1}$. A battle can have three possible outcomes. (1) p_0 dominates q_i. In this case, q_i will be pushed from the bridge and p_0 remains on its position to combat q_{i+1}. (2) q_i dominates p_0. Now, p_0 falls from the bridge and q_i becomes the new candidate Lemming p_0, *i.e.*, has to battle q_{i+1}. (3) If neither of the two Lemmings dominates the other, they are considered *incomparable*. In this case, p_0 stays on the bridge and q_i has to requeue.

The candidate Lemming p_0 has to remain on the bridge until it has fought all queued Lemmings once. When a challenger q_j confronts p_0 for the second time, we know that p_0 is not dominated by any other Lemming. Hence, p_0 is part of the Lemming skyline and can leave the bridge safely and q_j becomes the new p_0. The algorithm terminates when the queue is empty, *i.e.*, all dominated Lemmings have fallen from the bridge. The Lemmings still alive all belong to the Lemming *skyline*. This algorithm, known as *Best*, has been formally described in [8].

C. The Competition—2nd Year (BNL)

The following year many new Lemmings were born and it was time to redetermine the Lemming skyline. The previous year some Lemmings complained that they had to spend too much time queuing. In particular, requeing was time-consuming and delayed the entire competition. To improve on this drawback, the set of candidate Lemmings was increased from 1 to w. The modified version of the algorithm is known as *block-nested-loops* (BNL) [6] and illustrated in Figure 2.

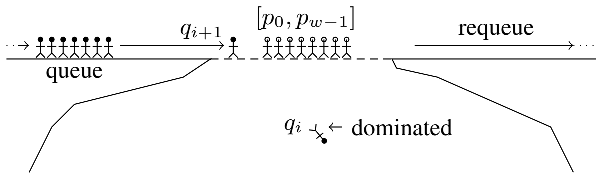

Figure 2. Lemming skyline with *BNL* [6].

On the bridge there is room for a window of w candidate Lemmings. A challenging Lemming q_i from the queue has to battle all candidate Lemmings on the bridge. If the challenging Lemming survives all battles, there are two possibilities. (1) If there are already w other candidate Lemmings on the bridge, q_i has to requeue. (2) Otherwise, q_i becomes a candidate Lemming p_i.

Unfortunately, now it is unclear when exactly a candidate Lemming has been on the bridge long enough to qualify as a true *skyline* Lemming. Luckily, the competition committee found a simple solution to this problem. After a Lemming q_i survives all candidate Lemmings on the bridge, it receives a *timestamp* independent of whether it becomes a candidate Lemming or has to requeue. A candidate Lemming p_i becomes a true *skyline* Lemming (and leaves the bridge) when it either encounters the first challenging Lemming q_j that has a larger timestamp or when the queue is empty. When Lemmings initially queue up for the first time, this timestamp is set to zero. A larger timestamp indicates that two Lemmings must have already competed against each other and since the queue is ordered, all following Lemmings in the queue will also have larger timestamps. More formally, the BNL algorithm is given in Figure 3.

D. The Competition—3rd Year (Parallel BNL)

While the BNL algorithm used in the 2nd year significantly reduced the number of times that Lemmings had to requeue, there were new complaints coming from some

```
foreach Lemming $q_i \in queue$ do
    $isDominated = false$;
    foreach Lemming $p_j \in bridge$ do
        if $q_i.timestamp > p_j.timestamp$ then
            /* $p_j \in Lemming\ skyline$ */
            bridge.movetoskyline($p_j$);
        else if $q_i \prec p_j$ then
            bridge.drop($p_j$);
        else if $p_j \prec q_i$ then
            $isDominated = true$;
            break;
    if not isDominated then
        timestamp($q_i$);
        if bridge.isFull() then
            queue.insert($q_i$);
        else
            bridge.insert($q_i$);
```

Figure 3. BNL Algorithm ($\prec$ means dominates).

Lemmings. In particular, candidate Lemmings criticized that most of the time on the bridge they were idle, waiting for their turn to battle the next challenger. Thus, in favor of higher throughput, the competition committee decided to slightly modify the BNL algorithm. The basic idea is that instead of one challenger q_i now up to w challengers $q_{(i+w-1)} \dots q_i$ are allowed on the bridge, and each challenger can battle a different candidate Lemming in parallel. This version of the algorithm is illustrated in Figure 4.

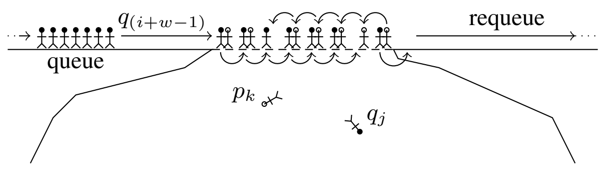

Figure 4. Lemming skyline: parallel BNL for FPGAs.

To avoid chaos on the bridge the procedure is as follows: In each iteration there is a *shift phase* followed by a *evaluation phase*. In the *shift phase* all challenger Lemmings $q_{(i+w-1)} \dots q_i$ move one position to the right to face their next opponent (indicated by the lower arrows in the figure). This frees the leftmost position on the bridge and allows a new Lemming from the queue to step on the bridge every iteration. Then in the *evaluation phase* all w pairs of Lemmings battle concurrently. As can be seen in the figure, in some situations a Lemming will not have an opponent because the corresponding Lemming was previously dominated, *i.e.*, fell from the bridge. In that case, the Lemming does not need to battle in this iteration.

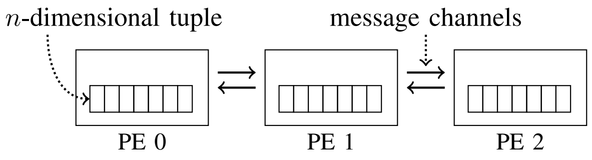

Figure 5. Working set items (*i.e.*, tuples with several dimensions) are distributed over a pipeline of processing elements. Neighboring processing elements are connected via 32-bit message channels.

Once a challenging Lemming q_i safely reaches the right end of the bridge, it qualifies as a candidate Lemming if there is room on the bridge, otherwise it has to requeue. If during the *evaluation phase* a candidate Lemming p_i falls from the bridge, the other Lemmings $p_{i+1} \dots p_{w-1}$ to the right of that Lemming have to move up in the subsequent *shift phase* and fill the gap (indicated by the upper arrows in the figure), making room for new candidate Lemmings that reach the right end of the bridge.

Again, we can use timestamping to decide when candidate Lemmings turn into true skyline Lemmings and can leave the bridge. Since the order among the Lemmings on the bridge is maintained, it is always the leftmost candidate Lemming that may become the newest skyline member. Thus, candidate Lemmings begin on the right end of the bridge and then gradually move towards the left end, where they need to wait until they encounter a challenger with a larger timestamp.

III. Implementation—Parallel BNL with FPGAs

The parallelized BNL version, sketched in the previous section, exhibits properties such as pipeline-parallelism and nearest neighbor communication, which make the algorithm amenable to an FPGA implementation that is efficient and highly scalable, as we will discuss in this section.

A. Pipeline-Parallelism

In BNL, each input tuple needs to be compared against the tuples of the active *working set*. The working set may consist of several hundred tuples but we want to spend only a minimal number of clock cycles on each input tuple in order to achieve high throughput. Hence, we distribute the tuples of the working set over a pipeline of daisy-chained processing elements, as illustrated in Figure 5.

A processing element stores a single tuple of the working set, consisting of multiple 32-bit dimensions. An input tuple is submitted to the first processing element in the pipeline from where it is forwarded to the neighboring processing element after evaluation via the specified message channel. For the sake of argument, let us assume that for now tuples are processed and forwarded atomically.

B. Causality Guarantees

We organized the processing elements such that input tuples are evaluated in a strictly feed-forward oriented way. This has important consequences that we can exploit in order to parallelize the execution over many processing elements while preserving the causality of the corresponding sequential algorithm.

Feed-forward processing implies that the global working set is scanned exactly once in a defined order. What is more, once an input tuple x_i has reached a processing element ν_h, its evaluation cannot be affected by any later input tuple x_j that is evaluated over a preceding processing element ν_d (conversely, the later x_j is guaranteed to see all effects caused by the earlier x_i).

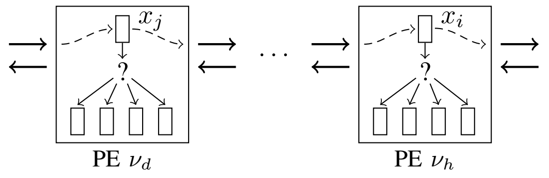

Figure 6. Causality guarantees. The earlier x_i will see no effects caused by the later x_j but x_j sees all effects of x_i.

These causality guarantees hold even if we let the executions of x_i on ν_h and x_j on ν_d run *in parallel* on independent compute resources, as illustrated in Figure 6. For example, once an input tuple x_i reaches the last processing element, we can safely assume that it has been compared against all other working set tuples and invoke appropriate actions.

C. Parallel BNL as Two-Phase Algorithm

As mentioned in Section II-D, we can divide skyline computation into two phases: *(i)* an *evaluation phase (ii)* and a *shift phase*. During the *evaluation phase*, a new state is determined for each processing element; but these changes are not applied before the *shift phase*, which is the phase that allows nearest neighbor communication. In our FPGA-based implementation, those two phases will run synchronously across the chip, as depicted in Figure 7.

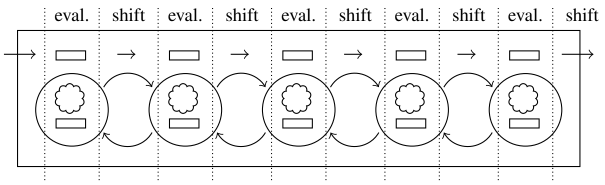

Figure 7. Two-phase processing in parallel BNL.

Evaluation Phase. The partial algorithm executed locally on each processing element in the evaluation phase very closely resembles the global algorithm, *i.e.*, standard BNL. As shown in Figure 8, the only changes to the original algorithm (cf. Figure 3) are that all *side effects* are now handled by the shift phase (cf. Figure 9), and we handle boundary cases more explicitly here.

Shift Phase. All interactions between neighboring processing elements are performed in the shift phase, displayed in

```
on each processing element do
    $q \leftarrow$ current input tuple ;
    $p \leftarrow$ local working set contents ;
    $s \leftarrow$ state of processing element ;
    if $q$.valid then                          /* next challenger */
        if $s = working\ set$ then             /* valid candidate */
            if $q$.timestamp $>$ $p$.timestamp then
                $s \leftarrow$ output ;        /* found skyline tuple */
            else if $q$.data $\prec$ $p$.data then
                $s \leftarrow$ deleted ;       /* drop window tuple */
            else if $p$.data $\prec$ $q$.data then
                $q$.valid $\leftarrow$ false ; /* drop input tuple */
        else if $s$ = free then                /* add input to window */
            timestamp($q$) ;
            $p$.data $\leftarrow$ $q$.data ;
            $s \leftarrow$ working set ;
            $q$.valid $\leftarrow$ false ;
```

Figure 8. Evaluation phase executed on each processing element.

Figure 9, which updates the global algorithm state based on the outcome of the evaluation phase. In essence, all input tuples are forwarded one processing element toward the right, whereas candidate results (working set tuples) move toward the left if there is space available. Since skyline candidates move toward the left, we report them on the leftmost processing element ν_0 once their timestamp condition has been satisfied. Likewise, on the rightmost processing element ν_{w-1}, we write input tuples to the overflow queue in DRAM if they were not invalidated during their move along the pipeline of processing elements, and cannot be inserted into the active working set because there is no space.

D. The State Automaton inside a Processing Element

While in Figures 8 and 9, we phrased parallel BNL as an algorithm in pseudo code, its implementation in hardware boils down to the simple *state automaton* depicted in Figure 10. In this state automaton, each processing element can be in any of four states: F (free), W (working set), X (deleted), and O (output). Initially, all processing elements are in state F. The dashed transitions enable *shifting* of processing elements toward the end or the beginning of the pipeline. To implement shifting, two adjacent processing elements *swap* their state and working set contents. Processing elements in state O are shifted to the beginning of the pipeline, whereas processing elements in state X are shifted to the end, where automatically the (dotted) transition $X \rightarrow F$ is executed. Note that we cannot directly perform the transition $W \rightarrow F$ because that would violate the causality guarantees that we mentioned earlier, *i.e.*, processing elements in state F are required to be at the

```
foreach processing element $\nu_i$ do
    /* all skyline results are emitted on $\nu_0$ */
    if $i = 0 \wedge \nu_i$.state = output then
        emit $\nu_i$.working set.tuple as result ;
        $\nu_i$.state ← deleted ;
    if $i < w - 1$ then /* not last processing element */
        if $\nu_i$.state = deleted then
            /* move up candidates to left */
            $\nu_i$.working set ← $\nu_{i+1}$.working set ;
            $\nu_i$.state ← $\nu_{i+1}$.state ;
            $\nu_{i+1}$.state = deleted ;
        /* challengers move one position to right */
        $\nu_{i+1}$.input tuple ← $\nu_i$.input tuple ;
    else /* the last processing element (physically) */
        if $\nu_i$.state = deleted then
            $\nu_i$.state ← free ;
        if $\nu_i$.input tuple.valid then
            timestamp($\nu_i$.input tuple) ;
            write $\nu_i$.input tuple to overflow queue ;
```

Figure 9. Shift phase. Results are reported on ν_0; candidates and input tuples move to the left and right, respectively; tuples after the last processing element are written to the overflow queue in DRAM.

end of the pipeline. The solid transitions labeled 'insert', 'output', and 'delete' are followed when a corresponding condition (listed below) is satisfied:

(i) *Insert*: when an input tuple reaches the first processing element in state F, it is inserted into the active working set and the respective processing element changes its state accordingly ($F \to W$).

(ii) *Output*: when the timestamp condition of a working set tuple has been met (cf. Figure 8), that tuple is a skyline tuple and is ready for output ($W \to O$).

(iii) *Delete*: working set tuples are deleted when they are dominated by an input tuple ($W \to X$). Output tuples (*i.e.*, skyline tuples) are deleted after they have been output, *i.e.*, processing elements in state O first are shifted to the beginning of the pipeline, where the tuple is output and the state of the processing element is changed ($O \to X$).

E. BRAM-based Component-wise Processing

Up to now, we have assumed atomic processing and forwarding of tuples. However, for performance reasons and because our implementation is based on BRAM, we *stream* all data one dimension at a time through the chain of processing elements. Figure 11 illustrates this for the case of three-dimensional tuples and eight processing elements. Notice that after each tuple, we pass *meta data* such as timestamp information or the *tuple valid* flag.

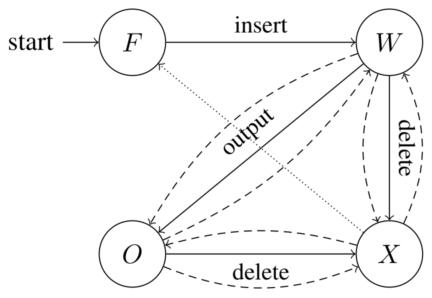

Figure 10. State diagram: processing element.

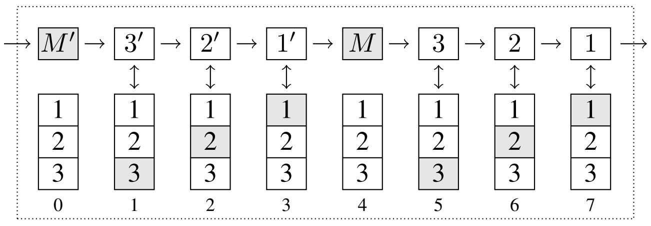

Figure 11. Two tuples streaming by eight processing elements.

We use BRAM for tuple storage within a processing element since potentially large tuples need to be saved in the working set. A BRAM block is big enough to store tuples of any realistic size. As a side-effect, the number of dimensions has minor impact on resource consumption.

To *swap* two adjacent processing elements, we cannot copy entire chunks of memory from one BRAM block to another in a single clock cycle—we have to do this word by word. Nevertheless, as illustrated in Figure 12, copying is still possible without reducing throughput.

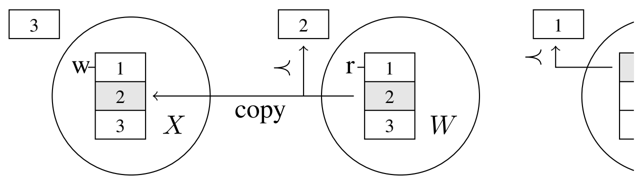

Figure 12. BRAM copy mechanism: three processing elements with a three-dimensional input tuple streaming by above.

In this example, the first processing element, is in state X (deleted), while the subsequent one is in state W (working set), which means they need to be *swapped* so that the deleted processing element can propagate to the end of the pipeline. For the BRAM block of the first processing element the write enable signal is asserted (w-flag). As data is read from BRAM of the second processing element (r-flag) for the dominance test, this data is written proactively to the BRAM of the 'deleted' predecessor processing element. At the end of the dominance test, the relevant BRAM contents have been entirely copied, and the state of both processing elements can be updated appropriately, also taking into account the outcome of the dominance test.

Please note that with this approach it is sufficient to instantiate single-ported BRAM, as opposed to dual-ported BRAM, which provides twice as many available BRAM blocks, enabling a longer pipeline of processing elements.

IV. Experiments

After discussing FPGA resource consumption, we experimentally compare our FPGA-based skyline operator against a single-threaded software implementation, as well as a state-of-the-art multi-threaded skyline implementation [7].

A. Experimental Setup

All experiments were run from main memory. We used the Xilinx XUPV5 development platform with a Virtex-5 FPGA (XC5VLX110T) clocked at 150 MHz and 256 MiB on-board DDR2 memory. The single-threaded CPU experiments were carried out on an Intel Xeon 2.26 GHz server processor (Gainestown, L5520, DDR3 memory). The multicore experiments were conducted on the same 8-core Intel Xeon server, as well as on a 64-core (AMD Bulldozer, 2.2 GHz, DDR3 memory) PowerEdge R815 Server from Dell.

B. Resource Consumption

In Table I, we display resource consumption for different configurations of our circuit using pipelines of 4, 64 and 192 processing elements. A single processing element consumes one out of 296 available single-ported BRAM blocks and roughly 320 LUTs, *i.e.*, 80 slices (post-map measurement).

Table I
Resource Consumption on the Virtex-5 (XC5VLX110T)

	Slices		Flip-Flops		LUTs	
available	17,280	100.0%	69,120	100.0%	69,120	100.0%
4 PEs	3,385	20%	6,371	9%	8,501	12%
64 PEs	9,204	53%	15,495	22%	27,385	40%
192 PEs	17,151	99%	34,951	51%	67,398	98%

A configuration with only four processing elements consumes 20% of the available slices because the measurements also include resources used for the DRAM controller [9] and the Ethernet-based communication framework [10] that we use to move data in and out of the FPGA board.

Furthermore, Table I shows that we are LUT-bound, and that a configuration with 192 processing elements saturates our FPGA. Notice, that even with 99% slice utilization, we were still able to operate the circuit at 150 MHz.

C. Performance Measurements

We now compare our FPGA-based skyline operator against a single-threaded software implementation using three different data distributions to give a better understanding of the performance characteristics of BNL versus our parallel implementation. Synthetic input data was generated with the data generator provided by [6] according to the three different distributions: (1) *random*, (2) *correlated*, and (3) *anti-correlated*. These distributions are commonly used to evaluate skyline operators. The input data consists of 1,024,000 input tuples. A tuple consists of seven dimensions and a timestamp resulting in a total width of 32 bytes, *i.e.*, the size of the entire input set is 31.25 MiB.

Randomly Distributed Data. For our randomly distributed data set, the skyline consists of 15,154 tuples, *i.e.*, 1.48 % of the input data are skyline tuples. This measure is called the *density* of skyline tuples. On the y-axis we display *throughput* (input tuples/sec) and on the x-axis we vary the size of the window used in the BNL algorithm.

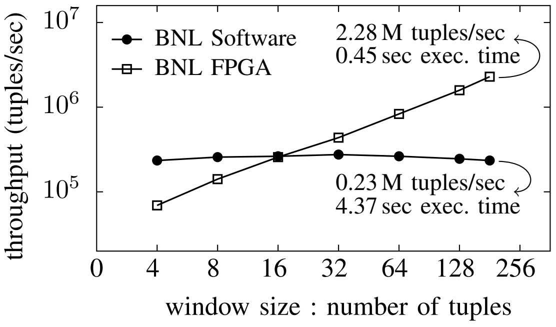

Figure 13. Randomly distributed dimensions → tuples/sec.

As can be seen in Figure 13, the size of the BNL window has little effect in the CPU-based version. On the FPGA, however, throughput increases linearly with the size of the window because a larger window also means more processing elements, *i.e.* a higher degree of parallelism.

Correlated Data. The dimensions of a tuple are correlated if there is a high probability that the values in all dimension are similar. This means that a tuple that is "good" in one dimension is likely to be "good" also in the other dimensions and therefore dominates many tuples. As a result, the skyline is very small, *e.g.*, in this experiment, the skyline consists of only 135 tuples, corresponding to a density of 0.013%.

In Figure 14, the CPU-based version of BNL is faster than the FPGA-based one. Low skyline density favors the CPU-based implementation because parallel compute power no longer is the key criteria for a fast execution. Rather, the CPU-based implementation here benefits from the more efficient memory subsystem: DDR3 plus caches versus DDR2 and no caches. In Figure 14, we display the upper bounds for throughput by dashed lines labeled CPU (63 million tuples/sec) and FPGA (17 million tuples/sec), respectively. These bounds were computed using a data set where the first input tuple is the only skyline tuple, which eliminates all other tuples. This results in a minimal number of tuple comparisons of $n-1$, where n is the number of input tuples, which is in line with the known *best case* complexity of $\mathcal{O}(n)$ for BNL [11].

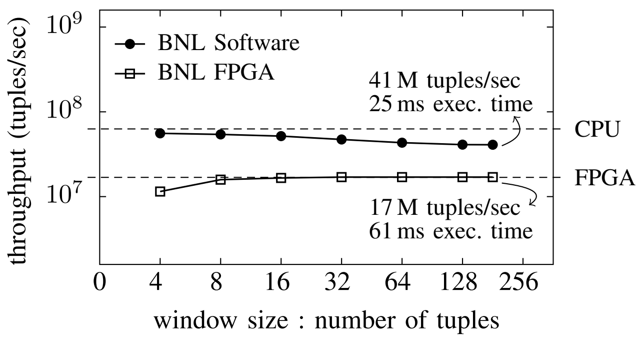

Figure 14. Correlated dimensions → tuples/sec.

While we cannot beat the CPU skyline operator with our FPGA implementation when the skyline tuples have a very low density, it is important to note that in absolute numbers both versions are very fast when dealing with correlated data. For instance, the fastest execution (window size = 4) of the above query on the CPU takes 18 milliseconds and on the FPGA (window size = 192) 61 milliseconds.

Anti-Correlated Data. This experiment is the opposite of the previous one. *Anti-correlated* means that a tuple, which is "good" in one dimensions, is likely to be "bad" in the other dimensions. In this case, a lot more tuples are part of the skyline, *e.g.*, now the skyline consists of 202,701 tuples, which corresponds to a density of 19.80%.

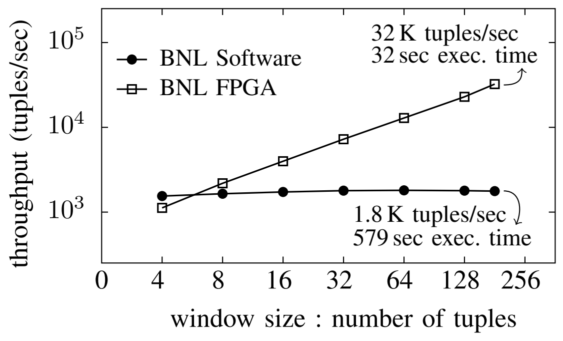

Figure 15. Anti-correlated dimensions → tuples/sec.

The computation of the skyline is now significantly more expensive, *e.g.*, the best execution time of the CPU-based version has gone from 18 milliseconds to almost 10 minutes. This slowdown is due to the increased number of comparisons since all skyline tuples have to be pairwise compared with each other. The number of comparisons among skyline tuples alone is $\frac{1}{2}s(s+1)$, where s is the size of the skyline—hence, the *worst case* complexity for BNL is $\mathcal{O}(n^2)$ [11].

The Curse of Dimensionality. Besides data distribution also dimensionality severely affects performance of skyline queries. Increasing the number of dimensions of the input tuples naturally increases the size of the skyline. For example, in an experiment with 102,400 15-dimensional input tuples following a *random distribution*, the density of the skyline was 74.86%. As can be seen in Figure 16, with a window size of 192, the FPGA executes the query in 5 seconds while it takes the software implementation 115 seconds, resulting in a speedup of 23X.

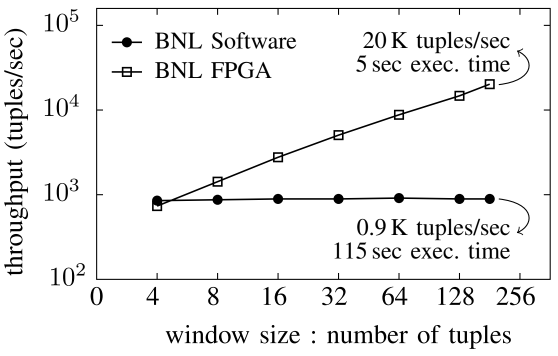

Figure 16. 15 dimensions (randomly distributed) → tuples/sec.

D. FPGA versus Multicore Server

We also compared our FPGA results to *PSkyline* [7], which is the fastest published skyline algorithm for multicore architectures.[2] We ran PSkyline on the same data sets as in the previous experiments that consisted of 1,024,000 seven-dimensional input tuples. We measured the performance of PSkyline on the 8-core (plus hyper-threading) Intel Xeon server used previously, as well as on a 64-core PowerEdge R815 Server from Dell. The FPGA was configured with 192 processing elements. The results are depicted in Table II.

Table II
EXECUTION TIME: FPGA VERSUS MULTICORE.

Data Distribution	FPGA	Intel Xeon	PowerEdge
Random	0.445 sec	0.722 sec	0.433 sec
Correlated	0.061 sec	0.003 sec	0.005 sec
Anti-correlated	31.633 sec	55.104 sec	18.574 sec

On the Intel Xeon server and on the PowerEdge server, best results were obtained using 16 and 64 threads, respectively. Notice that the performance for the compute-intensive workload (anti-correlated) achieved by our $750 (academic price) FPGA is not so far from the performance we measured on the $12,000 PowerEdge 64-core server.

Moreover, with 192 processing elements a throughput of 32,000 tuples/sec (anti-correlated distribution) is reached on the FPGA. This is more than two orders of magnitude below the upper bound of 17 million tuples/sec (cf. Figure 14), *i.e.*, with more real estate, there is still a lot of leeway to further increase performance by adding more processing elements.

[2]We would like to thank H. Im for providing the PSkyline code.

V. Related Work

In recent years, there have been several approaches to execute standard SQL queries on FPGAs (*e.g.*, [1], [2]). Furthermore, FPGA solutions in the context of databases have been proposed for sorting [12], XML filtering [13], or high-speed event processing [14]. Given the potential of FPGAs for data processing, we believe that a number of other database tasks would benefit from FPGA-acceleration.

The introduction of skyline queries in 2001 [6] has created a new direction for research (a comprehensive overview is given in [11]). Recently, there have been a few attempts to exploit parallelism for skyline query processing, *e.g.*, using SIMD instructions [15] or multiple threads [7] on multicore machines. However, the compute-intensive nature of skyline queries suggests that even higher degrees of parallelism are required to effectively tackle this type of problem.

Our approach to solving skyline queries is based on well-studied FPGA concepts such as *stream processing*, *nearest neighbor communication*, and *pipeline-parallelism* (see, *e.g.*, [16], [17]). In this work, we have combined these fundamental techniques into a parallel processing model for skyline queries. However, we are convinced that a number of other (database) problems can be solved in a very similar way, *e.g.*, as was done for *frequent item counting* with FPGAs in [18]. Thus, in future work, we want to define an abstraction that comprises the parallel processing model used here, and exactly characterize the class of problems that can be efficiently solved this way.

VI. Conclusions

FPGAs are becoming increasingly popular in large all-in-one-box database systems for data warehousing—so-called *appliances*. While simpler database tasks, (*e.g.*, pre-filtering, (de)compression) are already successfully off-loaded to FPGAs, hardware implementations of more complex operators like the *skyline operator* still need further investigation. Finding good implementation strategies for such operators can lead to significant gains, as we demonstrated for skyline computation in this paper.

Our experiments show that our implementation on a rather low-end FPGA can significantly outperform a single-threaded software version of BNL and delivers performance results close to those obtained from a highly-optimized parallel skyline algorithm [7] on a modern multicore server using all 64 cores. Our design is very *scalable*, suggesting that on a larger FPGA significantly higher performance gains could be achieved due to the direct relationship between throughput and the number of processing elements.

Acknowledgements

This work was supported by the Swiss National Science foundation via an *Ambizione* grant for Jens Teubner at ETH Zurich and by the Enterprise Computing Center (ECC) of ETH Zurich (http://www.ecc.ethz.ch/).

References

[1] C. Dennl, D. Ziener, and J. Teich, "On-the-fly Composition of FPGA-Based SQL Query Accelerators Using A Partially Reconfigurable Module Library," in *FCCM'12*, Toronto, ON, Canada, 2012.

[2] B. Sukhwani *et al.*, "Database Analytics Acceleration using FPGAs," in *PACT'12*, Minneapolis, MN, USA, 2012.

[3] G. Alonso, D. Kossmann, and T. Roscoe, "SwissBox: An Architecture for Data Processing Appliances," in *CIDR'11*, Asilomar, CA, USA, 2011.

[4] IBM/Netezza. [Online]. Available: http://www.redbooks.ibm.com/abstracts/redp4725.html

[5] T. C. Scofield *et al.*, "XtremeData dbX: An FPGA-Based Data Warehouse Appliance." *Computing in Science and Engineering*, vol. 12, no. 4, pp. 66–73, 2010.

[6] S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," in *ICDE'01*, Heidelberg, Germany, 2001.

[7] S. Park, T. Kim, J. Park, J. Kim, and H. Im, "Parallel Skyline Computation on Multicore Architectures," in *ICDE'09*, Shanghai, China, 2009.

[8] R. Torlone and P. Ciaccia, "Which Are My Preferred Items?" in *Workshop on Recommendation and Personalization in eCommerce (RPEC)*, Malaga, Spain, 2002.

[9] R. Bittner, "The Speedy DDR2 Controller For FPGAs," in *ERSA*, Las Vegas, NV, USA, 2009.

[10] K. Eguro, "SIRC: An Extensible Reconfigurable Computing Communication API," in *FCCM'10*, Charlotte, NC, USA, 2010.

[11] P. Godfrey, R. Shipley, and J. Gryz, "Maximal Vector Computation in Large Data Sets," in *VLDB'05*, Trondheim, Norway, 2005.

[12] D. Koch and J. Torresen, "FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting," in *FPGA'11*, Monterey, CA, USA, 2011.

[13] R. Moussalli *et al.*, "Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs," in *ICDE'11*, Hannover, Germany, 2011.

[14] H. Inoue, T. Takenaka, and M. Motomura, "20Gbps C-Based Complex Event Processing," in *FPL'11*, Chania, Crete, Greece, 2011.

[15] S.-R. Cho *et al.*, "VSkyline: Vectorization for Efficient Skyline Computation," *SIGMOD Rec.*, 2010.

[16] A. Hormati *et al.*, "Optimus: efficient realization of streaming applications on FPGAs," Atlanta, GA, USA, 2008.

[17] G. Kahn, "The Semantics of Simple Language for Parallel Programming," in *IFIP Congress*, Stockholm, Sweden, 1974.

[18] J. Teubner, R. Müller, and G. Alonso, "FPGA Acceleration for the Frequent Item Problem," in *ICDE'10*, Long Beach, CA, USA, 2010.