Shifter Lists—A Data Structure for Massive Parallelism

ABSTRACT

A wide range of technology trends are creating the opportunity to use highly parallel co-processors next to conventional CPUs to improve the performance of data processing systems. However, it is often difficult to exploit the inherent parallelism in these devices. Most systems available focus on ad-hoc implementations of single operators. Rather than focusing on how to parallelize a given operator, in this paper, we propose a new data structure—*shifter lists*—that has been designed to support data processing on massively parallel hardware (hundreds to thousands of processing elements). A shifter list can accommodate processing of arbitrary complexity at each processing stage while guaranteeing throughput-optimized processing. In the paper we present shifter lists, characterize their behavior, show a first implementation (on an FPGA, for generality), and apply the result to a use case (skyline queries) to show how to determine when this type of solution makes practical sense.

1. INTRODUCTION

There has been an increasing amount of research and commercial systems that exploit heterogeneous, low power, and massively parallel co-processors to accelerate data processing operations. For instance, server vendors such as Netezza [20] (acquired by IBM in 2010) and Convey [7] equip their systems with configurable hardware accelerators known as *field-programmable gate arrays* (FPGAs), which are inherently parallel devices.

This new hardware provides aggregated compute power of ever-increasing extent, but at the same time it has become more difficult to turn the hardware's parallelism into true performance. An often-seen answer to this challenge is to build specialized algorithm implementations that addresses modern hardware characteristics for one specific application task, *e.g.*, by fitting the problem to an *input data partitioning* scheme. Not only does this tend to leave much of the true hardware potential unused. Moreover, the lack of suitable abstractions prevents the invested efforts to carry over from one application solution to another.

In this work we provide one such abstraction. We propose a new data structure, *shifter lists*, that helps in the design of massively parallel and *scalable* algorithms for a wide range of problems. Shifter lists combine data organization, computational power, and synchronization into a new parallel processing model that naturally supports the characteristics of emerging parallel hardware. In our model, we think of input data as a data stream that propagates through the shifter list, which itself is distributed over many processing elements. The processing elements are arranged as a pipeline and locally update the shifter list as input data *flows by.* The only communication required is between neighboring processing elements.

We illustrate shifter lists based on a common database problem, the computation of the *skyline.* Skyline computation is a good example where straightforward input data partitioning neither matches the complexity properties of the problem—linear in the input data volume, but quadratic in the (intermediate) skyline result—, nor does it fit with the characteristics of modern parallel hardware. With shifter lists, by contrast, we partition the *working set* of a block-nested-loops (BNL) [4] variant and leverage the data structure's lightweight partitioning mechanisms across many parallel processing units.

Shifter lists are a generic data structure that can be used at different levels of granularity. In this paper, we aim at scalability to very high degrees of parallelism. For evaluation, we thus use *field-programmable gate arrays (FPGAs)*, where we can experiment with degrees of parallelism far beyond those of commodity multi-core hardware (on our FPGA hardware, we could accommodate almost two hundred parallel processing units). Though absolute performance is not the main focus of this paper, our experiments show that we outperform existing CPU-based solutions by almost a factor of 20, even on low-end FPGA hardware. As such, our prototype could readily be used as a *co-processor* to accelerate existing skyline applications.

We present shifter lists as follows. Section 2 motivates our work from the hardware technology side and relates it to existing ideas. In Section 3, we introduce shifter lists. Section 4 illustrates the use of shifter lists in skyline computation in a figurative way, before we realize the idea on concrete (FPGA) hardware in Section 5. We evaluate the characteristics of shifter lists in Section 6 and wrap up in Section 7.

2. RELATED WORK AND MOTIVATION

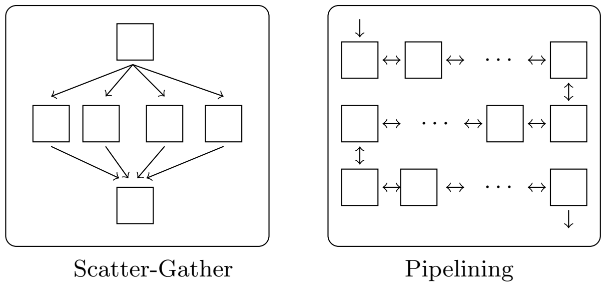

Figure 1: Scatter-Gather versus Pipeline of Cores

The prevalence of parallel hardware is pushing the software side harder and harder to come up with efficient parallel problem solutions. Berkeley researchers phrased this very provocatively in a recent report [1]: "If researchers meet the parallel challenge, the future of IT is rosy. If they don't, it's not."

2.1 Parallel Data Processing

The database community has investigated parallel data processing techniques already long before the current multi-core race. A good overview of parallel databases has been published, *e.g.*, by DeWitt and Gray [9]. Parallelism, thereby, was used in essentially two ways: *(i) data partitioning*, large input data is partitioned to run parallel instances of the same operator on many nodes (*e.g.*, [17]), and *(ii) inter-operator pipelining*, where pipelined query plans are taken literally and individual operators are assigned to distinct processing resources (*e.g.*, [16]).

Interestingly, intra-operator parallelism has been considered useful only if combined with input data partitioning, and pipelining has been used exclusively *across* individual operators (or, at least, across distinct stages of operators like in merge join [14]). As noted in [9], inter-operator pipelining bears a risk of uneven load distribution, because different operators within the same pipeline may have considerable differences in cost.

Modern Hardware. All of the above techniques (and research on parallel databases in general) were an excellent fit for the available hardware at the time. Meanwhile, however, hardware characteristics—in particular with respect to the availability of parallelism—have changed considerably. In this work, we thus propose to *re-think* parallel algorithm design and devise a new programming model that matches the actual trends in hardware.

Most importantly, this affects the interplay of parallelism with *communication.* With growing core counts, the average on-chip *distance* grows between arbitrary communication partners, requiring *additional energy* and increasing *latency* [3]. What is more, the necessary *routing logic* scales quadratically in the number of compute nodes, which limits the observed bandwidth for arbitrary communication patterns.

Communication Patterns. As illustrated in Figure 1, algorithms based on *scatter-gather* mechanisms are particularly affected by the cost of communication. Negative effects can be avoided, however, if the communication follows very simple topologies, such as *pipelining* along a series of parallel units (Figure 1), or *ring* and *tree* topologies.

Existing algorithms for parallel database processing barely reflect these effects. Most recently, *e.g.*, Kim et al. [15] or Blanas et al. [2] devised parallel variants of *join algorithms* for multi-core hardware. But while the primary focus of these techniques—cache awareness—can be viewed as one particular type of communication (to main memory), neither technique is really aware of the implied inter-core communication.

One of the few exceptions that do address core-to-core communication is the work of Gao et al. [11]. *QPipe* places database operators on individual cores and explicitly pipelines data between cores. The proposed scheme is still limited, however, to *inter-operator* pipelining, hence is not prepared for really large degrees of parallelism.

With shifter lists, we address the design of parallel database algorithms in two important ways: *(i)* shifter lists can be used as a generic implementation strategy to build parallel algorithms—no need to re-start platform optimization for each new problem instance; *(ii)* shifter lists have the awareness of communication cost built-in. It is applied by bringing pipelining to the *inside* of individual operators.

Our work shares an interesting characteristic with the work of Wang et al. [24] on large-scale parallelization of behavioral simulations. By running parallel code in two phases ("query phase" and "update phase" in [24]), algorithms can be phrased in a way that is intuitive, yet efficient to parallelize.

2.2 Heterogeneous Hardware

The trend toward increasing degrees of parallelism is complemented with a notion of hardware *heterogeneity.* Specialized co-processors and even programmable logic are already used successfully to assist general-purpose cores on compute- or data-intensive tasks.

A particular instance of this trend are *field-programmable gate arrays (FPGAs)*, which proved already very successful in the database domain. FPGAs are digital logic devices that can be used to realize any hardware circuit by a mere (software-based) device re-configuration. The configured logic can then solve a particular problem at very high speeds and with favorable energy consumption properties.

IBM/Netezza's *TwinFin* [20] is probably the most prominent example of a commercial FPGA-powered database appliances. On the research side, FPGA solutions have been proposed for sorting [19], XML filtering [18], or high-speed event processing [22]. Nevertheless, those examples all confirm the observation of Chung et al. [6]: FPGAs still lack essential abstractions that have become pervasive in general-purpose computers; rather, most systems are developed in an ad-hoc manner for just one particular problem setting. With shifter lists, we work toward one such abstraction. In this report, we illustrate how shifter lists can be used to build parallel FPGA solutions for database tasks.

FPGAs are interesting for our research also because of their inherent massive parallelism. The degree of parallelism is limited mainly by the amount of available *chip space* on the given piece of hardware. At the same time, the observed effects of increased parallelism are representative for what we can expect from future commodity hardware. Most importantly, FPGAs can be used to illustrate the interplay of parallelism and communication on hardware that is already available today.

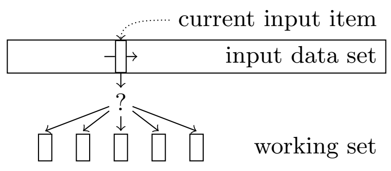

Figure 2: Typical application pattern for a shifter list: For each input item, the working set is accessed and possibly modified.

2.3 Skyline Processing

After skyline queries were first introduced in 2001 [4], a decade of research has produced a variety of different approaches to solving skyline queries (a comprehensive overview of the general directions of approaches is given in [13]). To demonstrate shifter lists, we revisit block-nested-loops (BNL) [4], one of the early skyline algorithms, which is simple and yet still effective today.

Recently, there have also been some approaches to exploit parallelism for skyline queries. On multi-core machines, the main problem is that multi-threaded skyline algorithms using traditional approaches often only scale up to a few cores [21]. Using SIMD, the dominance test of skyline queries [5] can be improved. However, maximal theoretical speedup is limited by the vector size of the SIMD registers (to 4 without AVX and to 8 otherwise).

3. SHIFTER LISTS

The main target of our shifter list data structure are application patterns as illustrated in Figure 2. From a given input data set, all items are consumed in turn. The single input item is evaluated against many or even all the items in an in-memory *working set.* Possibly, this evaluation results in an update to the working set, such as adding the current input item to the set or removing others.

Many common database tasks match this pattern. For instance, stream processing engines often operate on *windows of tuples.* For each input tuple, the window is consulted (*e.g.*, to compute a join) and also updated (to implement sliding-window semantics). Top-k algorithms—to name another example—typically maintain an in-memory set of candidates. For each input item, the candidate list is inspected and updated if the input item is found to be closer than any of the existing candidates.

In this work, we use *skyline computation* to showcase shifter lists. We are going to base our work on the *block nested loops (BNL)* algorithm [4] that, for each input item, examines and updates a working set—consistent with the pattern in Figure 2.

In all these examples, note that the processing *order* may influence the operation's final result. Many applications demand that this *causality* implied by in-order processing is preserved also in a parallel execution scheme.

3.1 Shifter Lists as a Data Structure

The high-level structure of shifter lists is illustrated in Figure 3. Working set items are held in a number of *nodes* (each of which we will later assign to a separate compute resource). There is a defined total *order* among all nodes ν_i in a shifter list. Oftentimes, node contents themselves will have a defined order, resulting in a total order across all working set items.

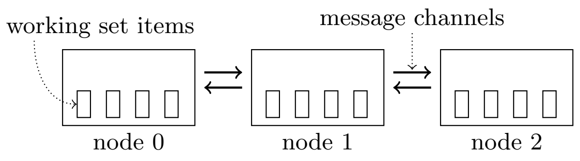

Figure 3: Shifter lists group working set items into nodes. Nodes are connected by asynchronous message channels.

Nodes are organized independently, but communicate with each other through well-defined *message channels.* As illustrated in Figure 3, these channels constrain communication to *neighbor-to-neighbor messaging.* Besides for application-defined messages, the channels will also be used to propagate input data and to *exchange* working set items between nodes (which ultimately results in a dynamic re-partitioning of the working set).

Ready for Modern Hardware. A working set organized as in Figure 3 is well prepared for the runtime characteristics of modern and future hardware. Grouping and neighbor-to-neighbor communication both ensure spacial *locality.* Awareness of communication locality is exactly among the properties that Borkar and Chien demand from the software side if we want to see continuous performance growth also on future hardware generations [3].

A possible way to implement messaging channels on commodity systems is to use *asynchronous FIFO queues.* Such queues were shown to match the capabilities of modern multi-core systems particularly well [12] and—if organized in a linear structure like shifter lists—scale to large core counts almost trivially. In FPGA designs, the point-to-point nature of the channels avoids costly multiplexing logic and reduces circuit complexity.

3.2 Shifter Lists are for Data Processing

To process input, we *shift* each input item into the leftmost list node. There, the item is evaluated against all local working set items, then shifted on to the right neighbor where the process repeats. Effectively, a sequence of input items flows through all nodes in a pipeline fashion.

The *actions* performed at each node depend on the specific task that is to be solved with the shifter list. Application code may decide to alter the local working set partition (*e.g.*, by dropping, inserting, or re-arranging items); drop the input item from the pipeline; or send and/or receive messages along the two message queues.

Self-Similarity. While the concrete action code has to be written specifically for each shifter list use case, we often see a "self-similarity" effect. Thereby, the local action code resembles very closely the superordinate algorithm that solves the overall application task. Typically, only *side effects* have to be modified to obtain the code for node-local execution. For instance, node-local "overflow tuples" in the BNL skyline algorithm have to be forwarded to the next shifter list node, rather than be written to an overflow file (more details later).

This self-similarity property not only eases application development. It also means that we can slice the original

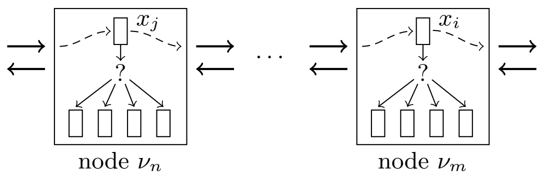

Figure 4: Shifter list causality guarantees. The earlier x_i will see no effects caused by the later x_j, while x_j sees all effects of x_i.

task into smaller and smaller units in a hierarchical fashion. Again, this fits well to what we see on the hardware side where, *e.g.*, functional units are combined to form one CPU core, several CPU cores make one chip die, dies are packaged into processors, etc. (*e.g.*, [8]). Conceivably, shifter lists could be applied across all these levels, or even up to the network and data center scale.

3.3 Shifter Lists are for Parallelism

As described above, input items are evaluated over the individual node contents in a strictly forward-oriented fashion. This has important consequences that we can exploit to parallelize the execution over many processing units and still preserve the causality of the un-parallelized code.

Causality Guarantees. Forward-only processing implies that the global working set is scanned exactly once (even in a defined order if that is a desirable property by the algorithm). What is more, once an input item x_i has reached a shifter list node ν_m, its evaluation cannot be affected by any later input item x_j that is evaluated over a preceding node $\nu_n, n < m$ (while conversely, the later x_j is guaranteed to see all effects caused by the earlier x_i).

These causality guarantees hold even if we let the executions of x_i on ν_m and x_j on ν_n run *in parallel* on independent compute resources, as illustrated in Figure 4. To uphold the guarantees, x_j only must never *overtake* x_i in the processing pipeline, a requirement that is easy to meet if all message channels are implemented as *FIFO queues*.

The preservation of causality hides much of the parallelization difficulties from the application developer. But the two-way interaction between two neighboring shifter list nodes may still bear a risk for race conditions. More specifically, an item x_i might be affected by the evaluation of x_j if x_j follows too closely in the processing pipeline and the interaction code is not engineered carefully. Explicit *barriers*, placed between successive input items, are an easy method to prevent such risks.

Application-Level Guarantees and Invariants. Specific applications may use the shifter lists' causality guarantees to further establish their own guarantees and invariants. When solving the skyline problem in Section 4, for instance, we add new items to the working set only on the right-most position. Since items never overtake each other, this guarantees that the oldest working set element can always be found at the front of node 0 (the left-most shifter list node).

3.4 Shifter Lists are Data *and* Logic

The intended use of shifter lists is to keep chunks of data—the contents of a node—strictly co-located with the processing logic that uses it. In a sense, this blurs the classical separation of logic and data.

Releasing this strict separation indeed makes sense in the light of ongoing hardware trends. There is a general consensus that power and heat dissipation problems will force a move toward *heterogeneous* system architectures, which might even soon be dominated by highly specialized co-processors or configurable hardware [3, 10]. In such designs, data structures can be wrapped right into the corresponding processing logic to further improve energy efficiency and speed.

In the experimental part of this work, we make the integration of data and logic very explicit by implementing shifter lists on top of FPGAs. The circuit that we propose would be ready to become one part of a heterogeneous multi-core architecture.

4. USE CASE: SKYLINE QUERIES

In a data warehouse *appliance* equipped with configurable hardware, *e.g.*, IBM/Netezza's Twin Fin [20], a significant performance optimization can be achieved when *compute-intensive*, long-running queries are outsourced to a dedicated co-processor previously loaded onto the configurable hardware. *Skyline* queries, discussed in this section, are typically very compute-intensive and are related to many other well-known database problems, *e.g.*, *top-K* or *convex hulls*, making skyline queries a good candidate for shifter lists.

4.1 The Lemming Skyline

To figuratively explain *skyline queries*, the BNL algorithm [4], and the modified version of BNL using shifter lists, we digress into the world of Lemmings. Lemmings[1] are primitive creatures that go on migrations in masses. On Lemmings Planet every year a challenge—*Lemmings got Talent*—takes 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 are pretty good in multiple skills. 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* (see Definition 1) by any other Lemming are considered best. In other words, the winners are those Lemmings that are part of the Lemming *skyline* (see 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 $L = \{l_1, l_2, \ldots l_n\}$, the skyline query returns a set of Lemmings S, such that any Lemming $l_i \in S$ is not dominated by any other Lemming $l_j \in L$.*

4.2 The Competition—1st Year (*Best*)

When the competition took place for the first time, the committee had a definition for the set of best Lemmings (see previous section) but it was still unclear how to determine this set. Thus, in the absence of sophisticated logistic

[1]As in the video game "Lemmings" originally developed by DMA Design: http://www.dmadesign.org/

means, one committee member suggested the following simple algorithm. Initially all Lemmings queue up in front of a bridge, as illustrated in Figure 5.

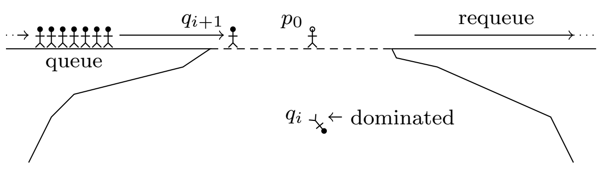

Figure 5: Lemming Skyline with *Best* [23]

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 *potential* skyline 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 *potential* skyline 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 Lemmings 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 [23].

4.3 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 *potential* skyline Lemmings was increased from 1 to w. The modified version of the algorithm is known as *block-nested-loops* (BNL) [4] and illustrated in Figure 6.

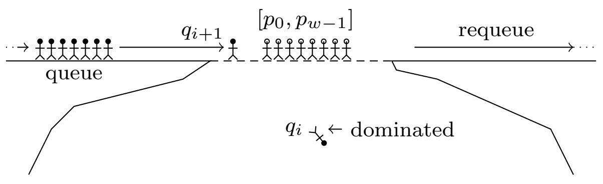

Figure 6: Lemming Skyline with *BNL* [4]

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

Unfortunately, now it is unclear when exactly a *potential* skyline 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 *potential* skyline Lemmings on the bridge, it receives a *timestamp* independent of whether it becomes a *potential* skyline Lemming or has to requeue. A *potential* skyline Lemmings p_i can now be said to be a true *skyline* Lemming (and leave the bridge) when it 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 the 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 7.

```
1  foreach Lemming $q_i \in queue$ do
2      $isDominated = false$;
3      foreach Lemming $p_j \in bridge$ do
4          if $q_i.timestamp > p_j.timestamp$ then
               /* $p_j \in$ Lemming skyline */
5              bridge.movetoskyline($p_j$);
6          else if $q_i \prec p_j$ then
7              bridge.drop($p_j$);
8          else if $p_j \prec q_i$ then
9              $isDominated = true$;
10             break;
11     if not isDominated then
12         timestamp($q_i$);
13         if bridge.isFull() then
14             queue.insert($q_i$);
15         else
16             bridge.insert($q_i$);
```

Figure 7: BNL Algorithm ($\prec$ means dominates)

4.4 The Competition—3rd Year (Shifter List)

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 Lemmings. In particular, potential skyline 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 using the *shifter list* approach. 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 *potential* skyline Lemming in parallel. This version of the algorithm is illustrated in Figure 8.

To avoid chaos on the bridge the procedure is as follows: In each iteration there is a *shift phase* followed by a *battle 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

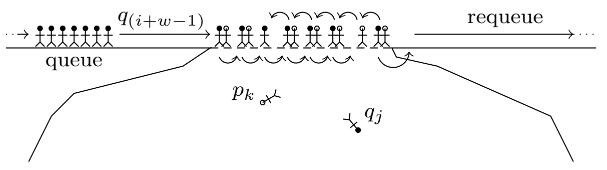

Figure 8: Lemming Skyline: BNL & Shifter List

iteration. Then in the *battle 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.

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

As in standard BNL, we can also use timestamping to decide when *potential* skyline 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 *potential* skyline Lemmings that can go first. Thus, *potential* skyline Lemmings get on the bridge on the right end and then gradually move towards the left end again, where they need to wait until they encounter a challenger with a larger timestamp.

5. PARALLEL BNL WITH FPGAS

When Börzsönyi et al. [4] first proposed the block-nested-loops (BNL) algorithm, their main motivation was to support skyline computation for problem sizes that would require external (and slow) memory. With today's hardware characteristics, the bandwidth to read input or overflow data (typically from large main memories) limits skyline computation only for extremely small working set sizes (few potential skyline tuples). In most practical cases, CPU load becomes the bottleneck when computing skylines.

This makes FPGAs a good alternative to conventional CPUs, because the relevant window sizes (say, 100 skyline candidates) conveniently fit into on-chip memories. Here we show how shifter lists then help to parallelize the computation within the FPGA to make best use of its available compute capacity. As our results in Section 6 show, this brings the algorithm back to memory-bound behavior, which in turn restores the original characteristics of the algorithm, where reducing the number of overflow tuples translates into faster execution of the algorithm—the larger the window, the better the performance.

5.1 BNL Using Shifter Lists

Given the opportunity for very fine-granular parallelism inside FPGAs, we configure our shifter list implementation such that each node holds only a single working set item. This allows us to fully leverage the available hardware parallelism and also helps us illustrate how shifter lists scale to very high degrees of parallelism.

```
on each node do
    q ← current input item ;
    p ← local working set content ;
    if ¬p.empty then  /* only if working set is valid */
        if q.timestamp > p.timestamp then
            p.isSkyline ← true ;
        else if q.valid then
            if q.data ≺ p.data then
                p.empty ← true ;  /* drop wnd tuple */
            else if p.data ≺ q.data then
                q.valid ← false ;  /* drop input tuple */
    else if p.isLast ∧ q.valid then
        p.data ← q.data ;      /* add input to window */
        p.empty ← false ;              /* bookkeeping */
        q.valid ← false ;
```

Figure 9: Evaluation phase of shifter list-based BNL implementation.

As illustrated in the previous section, data processing in shifter lists can be viewed as a two-phase algorithm: during the *evaluation phase* (the "battle" phase in the previous section), a new state is determined for each shifter list node; but these changes are not applied before the *shift phase*, which is the phase that also allows neighbor-to-neighbor communication. In our FPGA-based implementation, those two phases will run synchronously across the chip (since this is easy and efficient to realize on FPGAs).

Thanks to the *self-similarity* property of shifter lists, the partial algorithm executed locally on each shifter list node in the evaluation phase very closely resembles the global algorithm—BNL in our case. As shown in Figure 9, the only changes to the original algorithm (Figure 7) are that all *side effects* are now handled by the shift phase (and we handle boundary cases more explicitly here). In this code, dropping a "bridge" Lemming is implemented by setting the node's working set content to *empty*; "challengers" are dropped off the bridge by setting their *valid* flag to false.

All interactions between neighboring nodes are performed in the following shift phase, which updates the global algorithm state based on the outcome of the evaluation phase. In essence, all input items are forwarded one shifter list node toward the right, whereas candidate results (working set items) move toward the left if there is space available, *i.e.*, the left-next working set is marked as empty. Since candidate results move toward the left, we report them on the left-most node ν_0 once their timestamp condition has been satisfied. Likewise, on the right-most node ν_{w-1}, we write input items to the overflow file if they were not invalidated during their move along the shifter list pipeline, and cannot be inserted into the shifter list because it is already full.

5.2 FPGA Specifics

Taken literally, the shift phase illustrated in Figure 10 passes items atomically between neighboring shifter list nodes. In practice and for multi-dimensional input, this would lead to very high bandwidth demand (*e.g.*, 15 dimensions × 32 bits

```
1  foreach node $\nu_i$ do
       /* all skyline results are emitted on $\nu_0$ */
2      if $i = 0 \wedge \nu_i$.working set.isSkyline then
3          emit $\nu_i$.working set.tuple as result ;
4          $\nu_i$.working set.empty ← true ;
5          $\nu_i$.working set.isSkyline ← false ;
6      if $i < w - 1$ then          /* any but the last node */
7          if $\nu_i$.working set.empty then
               /* move up candidates to left */
8              $\nu_i$.working set ← $\nu_{i+1}$.working set ;
9              $\nu_{i+1}$.working set.empty ← true ;
           /* challengers move one position to right */
10         $\nu_{i+1}$.current item ← $\nu_i$.current item ;
11     else
12         if $\nu_i$.working set.empty then
13             $\nu_i$.working set.isLast ← true ;
14         if $\nu_i$.current item.valid then
15             write $\nu_i$.current item to overflow file ;
```

Figure 10: Shift phase of shifter list-based BNL implementation. Result tuples are reported on ν_0; candidates and input items move to left and right; items after last node are written to the overflow file.

× 150 MHz clock frequency = 9 GB/s in- *and* outgoing traffic), which is far out of reach for the hardware that we use. Instead, our implementation *streams* all data one dimension at a time, following best practices for hardware design.

Data Representation. Figure 11 illustrates this for the case of three-dimensional data and a shifter list configuration of ten nodes. After each data point, we pass *input bits* (such as timestamp information or the *valid* flag) as two additional 32-bit words I_1 and I_2.

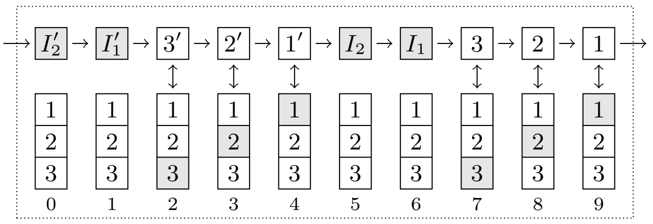

Figure 11: Dimension-at-a-time processing. Two tuples streaming by ten shifter list nodes.

Further, in a massively parallel environment like shifter lists, *timestamps* are not a suitable means to impose a stable total order on input data—which is what timestamps effectively do in [4]. Thus, we mark each input item with an immutable *sequence number* instead and introduce a *round counter* that is incremented whenever the tuple is written to the overflow file. Together, they allow us to test whether a candidate has seen all other input items without a need for actual timestamps.

BNL in Logic. While phrased in Figure 10 as an algorithm in pseudo code, its implementation in logic boils down to a *state automaton.* It turns out that most of the transitions in this automaton are not specific to a particular application problem, but are determined by the general data processing

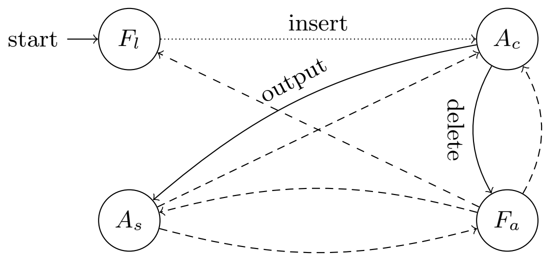

Figure 12: State diagram to implement shifter list nodes on top of FPGAs.

pattern that we saw in Figure 2. Only three transitions really depend on the concrete skyline problem, labeled 'insert', 'output', and 'delete' in Figure 12.

In this state automaton, each shifter list node can be in any of four states:

F_l: ***"free and last in pipeline"*** This state indicates that all following nodes are also free. An input item that has reached this point will be written to the local node (transition 'insert' in Figure 12), which means that a new candidate is added to the global working set.

F_a: ***"free and anywhere in pipeline"*** A working set item has been dropped along the pipeline (transition 'delete'). Another candidate can move up from the right.

A_c: ***"allocated; contains candidate tuple"*** An item in the working set that has not (yet) been identified as a skyline result.

A_s: ***"allocated; contains skyline result tuple"*** The content of this node has been identified as a skyline result. The item will be output to the user once it reaches the left-most node.

State automata of this type are well supported by actual FPGA hardware and, as we will see in the following section, can run at high throughput rates.

Managing State. Up to now we have assumed that potential skyline tuples can be stored in memory local to a shifter list node but we have not yet said anything about what this memory looks like in detail. Modern FPGAs typically ship with two types of on-chip memory: *registers* and *block RAM* (BRAM).

Registers are composed from flip-flops and the configurable logic resources on the FPGA. Registers are very fast and can be customized precisely to special application needs, such as wide word widths and high throughput, but they are also an expensive and scarce resource. Therefore they are best used for storing *small* amounts of data. Registers are automatically inferred from hardware description language (HDL) code by the tools that generate the configuration file for the FPGA.

Block RAM on the other hand needs to be explicitly instantiated by the hardware designer. Block RAM refers to blocks of dedicated memory with address, data, and control ports. A typical size for a single block of dual-ported BRAM is 36 kbit or 18 kbit for single-ported BRAM respectively. BRAM blocks are distributed across the FPGA chip, and their count varies depending on the FPGA between several hundred and a few thousand.

Shifter lists can be implemented using both memory types. Which memory type is more suitable depends on the application. For example, in our BNL skyline algorithm, BRAMs with a word width of 32 bits provide sufficient bandwidth and since our tuples can be quite large, *e.g.*, consisting of 15 dimensions and more, we would quickly run into space problems using registers for storing potential skyline tuples. Hence, in this case BRAM is the adequate choice.

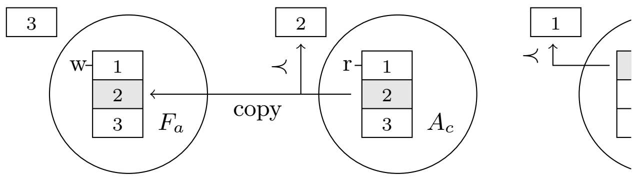

Figure 13: BRAM-Copy-Mechanism

Neighbor-to-Neighbor Communication. In the BNL algorithm, skyline tuples are output in the same order as they were inserted into the window of potential skyline tuples. This means that allocated nodes to the left of a given nodes will be deallocated *before* that node's content will be output. Thus, we never have the situation, where we need to swap to allocated nodes. We only swap free nodes with allocated ones and hence only need to copy data in one direction and then swap the state. The state however is stored in a register and swapping the contents of two registers in one clock cycle is not an issue. Figure 13 depicts the above described copy-mechanism in detail.

In the figure, the first node (on the left) is in state F_a and the second node is in A_c, which means that by shifter list semantics these two nodes need to swap their contents. The node in state A_c is performing dominance tests against input tuples streaming by (each numbered box above the nodes corresponds to one dimension of the input tuple). As data is read from BRAM of the allocated node for the dominance tests at the same time this data can be copied to the BRAM of the free predecessor. After an entire input tuple has been processed by the allocated node, all dimensions have also been copied to the free predecessor. Now, if the input tuple did not dominate the tuple, which is stored at the allocated node, the states can be swapped. Otherwise the allocated node is also freed since it was just dominated by the input tuple.

6. EXPERIMENTS

In our experiments, we compare the FPGA-based skyline operator against a software implementation. On the one hand, we demonstrate the *scalability* of our approach and show how *throughput* increases as we add more shifter list nodes. On the other hand, our results are put in relation to throughput measurements of a (faster clocked) CPU-based BNL implementation.

On the FPGA side we use an XUPV5-LX110T from Xilinx. In all experiments the FPGA was clocked at *151.1 MHz*. Our CPU-based experiments were carried out on a *2.26 GHz* Xeon L5520 Intel machine. For more details on the hardware that we used please refer to Appendix A.

6.1 Characteristics of BNL

Before we present our throughput measurements in the next section, we first want to explain some characteristics of the *block-nested-loops* (BNL) algorithm, which should help to better understand the measurements discussed later.

The main objective of original BNL [4] as an *external* algorithm was to reduce I/O operations. The larger the window, the fewer runs are needed, as can be seen in Figure 14, where the number of *overflow tuples* decreases almost linearly as we increase the size of the window. Thus, with I/O being the main bottleneck, a larger window directly translates into a higher throughput. However, observe that the number of tuple *comparisons* (see Figure 14) does not improve with a larger window. In fact, the number of comparisons might even slightly increase. Hence, if we run BNL in main memory the size of the window has little effect on runtime.

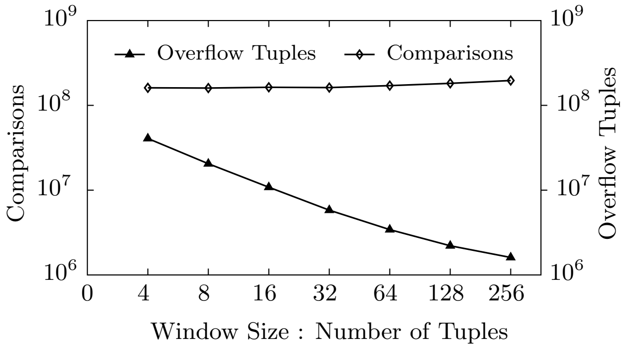

Figure 14: BNL: Comparisons vs. Overflow Tuples

Nevertheless, BNL as a main memory algorithm still exhibits reasonable performance. For the experiment in Figure 14, the skyline was computed over 1,024,000 input tuples of seven dimensions each following a uniform random distribution. It took *3.7 seconds* to compute the resulting skyline of 15,154 tuples on this data. As a comparison, we also ran a newer block-nested-loops algorithm designed for fast in-memory processing called *SSkyline*, which was recently presented in [21]. For *SSkyline* we measured an execution time of *3.3 seconds*. While *SSkyline* here is indeed a bit faster (speedup = 1.12X), it is not orders of magnitude better than BNL, and if we increase the number of dimensions or the correlation of the tuple dimensions, the two algorithms run practically at same speed.

In Figure 14 the number of comparisons increases with a larger window. This would suggest that a window size of one—BNL with a window size of one is referred to as an algorithm called *Best* [23]—should yield the best results. However, despite more comparisons BNL with a small window size executes faster than *Best*. In such a configuration, the system becomes bottlenecked by memory bandwidth, because of a very large number of overflow tuples. In the following measurements we will always indicate for which window size BNL achieved the best results.

6.2 Effects of Data Distribution

In the following experiments we evaluate *throughput* performance of our FPGA-based skyline operator versus a CPU-based *block-nested-loops* (BNL) implementation. Again, the input data consists of 1,024,000 seven-dimensional input tuples. Each dimension is represented by a 32-bit integer. Thus, together with a 32-bit *sequence number* a single tuples is 32 bytes wide and the size of the entire input set is

31.25 MiB. We use synthetic input data with three different distributions: (1) *random*, (2) *correlated*, and (3) *anti-correlated*. These distributions are commonly used to evaluate skyline operators. To generate the data we used the data generator[2] provided by [4].

6.2.1 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* as number of input tuples processed per second and on the x-axis we vary the size of the window used in the BNL algorithm.

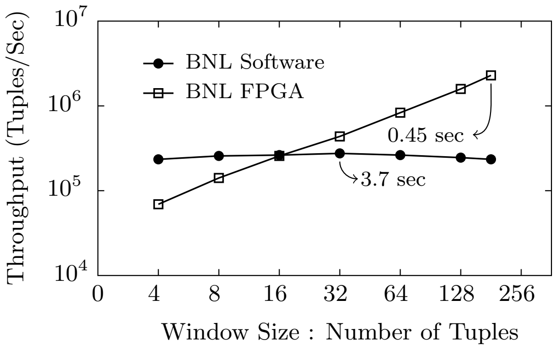

Figure 15: Random Distr. → Tuples/second

As we already noted before, 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. This is not surprising because in the FPGA case a larger window also means more shifter list nodes or higher degrees of parallelism. Since the BNL algorithm here is compute-bound, we can significantly increase throughput by performing more dominance tests in parallel. Also notice that the frequency of the CPU with 2.26 GHz is roughly 15 times higher than that of the FPGA, which is 151.1 MHz.

Here, BNL executes the fastest (3.7 seconds) with a window size of 32. The best FPGA results are at a window size of 192 with an execution time of 0.45 seconds. The break-even-point between the two versions is at a window size of 16.

6.2.2 The Correlated Case

In the second experiment, we compute the skyline on data that favors the CPU-based implementation. Our CPU runs at higher clock speed and has a faster memory subsystem than our FPGA. If the computational effort per input tuple is very low, aggregated compute power is no longer the key criteria for a fast execution of the algorithm and the CPU will be faster than the FPGA. This is the case, when the dimensions of the input tuples are strongly correlated, *i.e.*, when a tuple is "good" in one dimension, it is likely to be "good" also in the other dimensions. As a result, the skyline is very small. For example, in our experiment, depicted in Figure 16, the skyline consists of only 135 tuples, which corresponds to a skyline tuple density of *0.013%*.

[2]http://www.pubzone.org/pages/publications/showWiki.do?task=showComment&commentId=201&publicationId=298353&versionId=298378

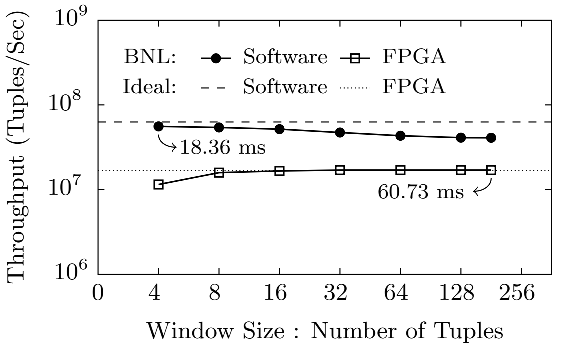

Figure 16: Correlated Distr. → Tuples/second

The first observation is that the CPU-based version is indeed faster than the FPGA-based version. This is not surprising since the upper bound for throughput is also higher for the CPU than for the FPGA because of the reasons mentioned above. The upper bounds are depicted in the figure for both CPU and FPGA by a dashed line and a dotted line 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 [13].

For the CPU version throughput decreases slightly with a larger window. The reason for this is that a larger window means that more unnecessary tuple comparisons are performed. By contrast, in the FPGA case, the additional comparisons due to the larger window do not hurt since they are computed in parallel. Therefore throughput increases until we hit the limits of our memory subsystem.

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 above query took *18.36 ms* on the CPU and *60.73 ms* on the FPGA. Thus, for many use cases with a reasonable number of input tuples, when the data is strongly correlated, this performance difference will not be very noticeable. However, as we will see in the next section, with an increased *density* of skyline tuples the computation of the skyline becomes extremely expensive and calls for an optimized solution.

6.2.3 The Anti-Correlated Case

This experiment is the inverse of the previous experiment. Here, the dimensions of the input tuples are *anti-correlated* meaning 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%*.

Now, the computation of the skyline has become significantly more expensive, *e.g.*, the best execution time of the CPU-based version has gone from *18.36* milliseconds to almost *10* minutes, *i.e.*, more than four orders of magnitude slower than the correlated case. The slowdown can be explained by the increase in number of comparisons since all

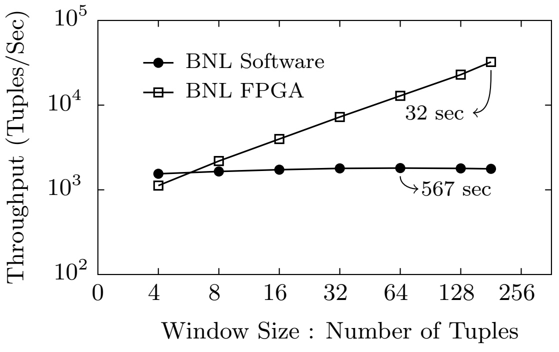

Figure 17: Anti-Correlated Distr. → Tuples/second

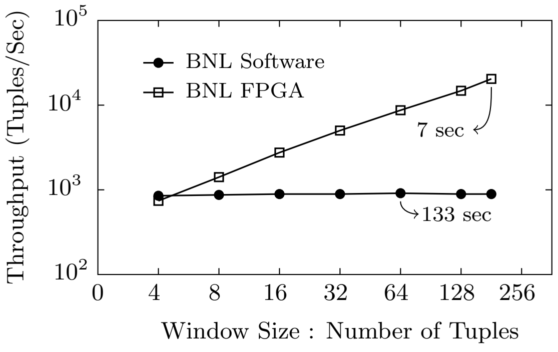

Figure 18: Fifteen Dimensions → Random Distr.

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 $\mathcal{O}(n^2)$ [13]. Therefore, the number of comparisons becomes the dominating factor as the size of the skyline increases.

With this many comparisons—we measured an average of $\sim 25,000$ comparisons per input tuple—the skyline query becomes compute-bound. In the FPGA case, this cost can be reduced with every additional shifter list node. This observation is also confirmed by the measurements in Figure 17: y-axis and x-axis are in logarithmic scale and the throughput increases linearly with the size of the window, *i.e.*, the number of shifter list nodes.

With a 192 shifter list nodes we reach a throughput of $\sim$ 32 thousand tuples/second. This is almost two orders of magnitude below the upper bound of $\sim$ 16 million tuples/second for throughput (coming from the memory subsystem). Therefore there is still a lot of leeway to further increase performance by adding more shifter list nodes. The number of shifter list nodes that we can put on an FPGA is limited by FPGA real estate. Our results suggest that a larger FPGA would further increase throughput, as (below the upper bound) there is a one-to-one relationship between chip space and throughput performance.

6.3 The Curse of Dimensionality

By now we know that the *size of the skyline* severely impacts the performance of the BNL algorithm. Increasing the number of dimensions of the input tuples naturally increases the size of the skyline. This phenomenon is known as *the curse of dimensionality* [25]. Here, we show an experiment with tuples of 15 dimensions each. The dimensions of every tuple follow a random distribution. Because of the increased computational intensity we had to reduce the input data set by a factor of ten. Out of the 102,400 input tuples the skyline here consists of 76,657 tuples which translates to a density of *74.86%*. Our results are displayed in Figure 18.

The graph above looks similar to the one in Figure 17. The break-even-point has moved even a bit further to the left, close to a window size of four. Thus, even though the CPU is clocked about 15 times faster than the FPGA, it cannot do 15 times more comparisons per clock cycle, otherwise the break-even-point should never be below a window size of 15. A single comparison of two 15-dimensional tuples on the FPGA takes 17 clock cycles, *i.e.*, 112.5 ns at a 151.1 MHz clock. We also measured how long a comparison takes on our CPU and the result was 34.4 ns. Thus, we need at least $\lceil\frac{112.5}{34.4}\rceil = 4$ shifter list nodes to achieve better results than the CPU.

6.4 Concluding Remarks

The important lesson to be learned from the experiments in this section is that the *computational intensity* of skyline queries is mainly driven by the *density* of the skyline tuples, *i.e.*, the size of the skyline. The *density* of skyline tuples depends on how the dimensions of the tuples are distributed and more importantly on the number of dimensions. The FPGA-based skyline operator improves performance most when the *density* of skyline tuples is high as then the parallel compute power of the FPGA is used most effectively. For instance, with a *density* of 74.86% in the experiment in Section 6.3, the FPGA with 192 shifter list nodes achieves a 17.7X speedup over the CPU based version.

7. CONCLUSIONS

Nearly ten years ago the laws of physics changed the way in which microprocessor architectures evolved. While miniaturization of the transistors according to *Moore's Law* still prevails, this no longer translates into faster clock speeds. Thus, the only hardware solution to further increasing performance of algorithms seems to be parallelization. However, parallelization of algorithms is often a non-trivial task, especially when the problem is not *embarrassingly parallel.* In particular, when reasoning about parallelizing a particular task the focus is often on the computation while the communication overhead is often overlooked. But the communication overhead can become a severe bottleneck especially as the core count increases.

Until now, the usual way to address the parallelism challenge is to build a specialized parallel algorithm for each and every given problem. With *shifter lists*, we propose a general-purpose solution instead. A shifter list is a novel data structure tailored at tightly-coupled many-core systems. A special property of shifter lists is that they can be used to unify algorithmic data processing components with the data structure itself. We used an FPGA as a test platform for a many-core system of up to 192 processing elements and showed by example of a skyline algorithm how shifter lists help accessing this massive parallelism in a highly scalable way.

8. REFERENCES

[1] Krste Asanovic, Rastislav Bodík, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David A. Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine A. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[2] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *SIGMOD Conference*, pages 37–48, Athens, Greece, June 2011.

[3] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Commun. ACM*, 54:67–77, May 2011.

[4] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.

[5] Sung-Ryoung Cho, Jongwuk Lee, Seung-Won Hwang, Hwansoo Han, and Sang-Won Lee. VSkyline: vectorization for efficient skyline computation. *SIGMOD Rec.*, 39:19–26, December 2010.

[6] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: an in-fabric memory architecture for FPGA-based computing. In *FPGA*, pages 97–106, 2011.

[7] Convey Computer Corp. http://www.conveycomputer.com.

[8] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, March 2010.

[9] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 35(6):85–98, 1992.

[10] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proc. of the 38th Int'l Symposium on Computer Architecture (ISCA)*, pages 365–376, San Jose, CA, USA, June 2011.

[11] Kun Gao, Stavros Harizopoulos, Ippokratis Pandis, Vladislav Shkapenyuk, and Anastassia Ailamaki. Simultaneous Pipelining in QPipe: Exploiting Work Sharing Opportunities Across Queries. In *ICDE*, page 162, 2006.

[12] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for Efficient Pipeline Parallelism. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 43–52, Salt Lake City, UT, USA, February 2008.

[13] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal vector computation in large data sets. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 229–240. VLDB Endowment, 2005.

[14] Rikard Hultén, Christoph W. Kessler, and Jörg Keller. Optimized On-Chip-Pipelined Mergesort on the Cell/B.E. In *Euro-Par (2)*, pages 187–198, 2010.

[15] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[16] Ming-Ling Lo, Ming-Syan Chen, Chinya V. Ravishankar, and Philip S. Yu. On Optimal Processor Allocation to Support Pipelined Hash Joins. In *SIGMOD Conference*, pages 69–78, 1993.

[17] Manish Mehta and David J. DeWitt. Managing Intra-operator Parallelism in Parallel Database Systems. In *VLDB*, pages 382–394, 1995.

[18] Roger Moussalli, Mariam Salloum, Walid A. Najjar, and Vassilis J. Tsotras. Massively parallel XML twig filtering using dynamic programming on FPGAs. In *ICDE*, pages 948–959, 2011.

[19] René Müller, Jens Teubner, and Gustavo Alonso. Data Processing on FPGAs. *PVLDB*, 2(1):910–921, August 2009.

[20] Netezza Corp. http://www.redbooks.ibm.com/abstracts/redp4725.html.

[21] Sungwoo Park, Taekyung Kim, Jonghyun Park, Jinha Kim, and Hyeonseung Im. Parallel Skyline Computation on Multicore Architectures. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 760–771, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Mohammad Sadoghi, Hans-Arno Jacobsen, Martin Labrecque, Warren Shum, and Harsh Singh. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. *PVLDB*, 3(2):1525–1528, 2010.

[23] R. Torlone and P. Ciaccia. Which are my preferred items? In *Workshop on Recommendation and Personalization in eCommerce (RPEC)*, 2002.

[24] Guozhang Wang, Marcos Vaz Salles, Benjamin Sowell, Xin Wang, Tuan Cao, Alan Demers, Johannes Gehrke, and Walker White. Behavioral Simulations in MapReduce. *PVLDB*, 3(1):952–963, 2010.

[25] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

APPENDIX

A. HARDWARE USED IN EXPERIMENTS

All of our FPGA experiments were conducted on a low-cost ($750.00 university price) Virtex-5 FPGA from Xilinx clocked at *151.8 MHz*. Commonly available resource types hosted by FPGAs include *lookup tables* (LUTs) to realize combinational logic, on-chip storage in terms of *Block RAM* (BRAM) and *flip-flops*, and a configurable *interconnect network*. Some selected characteristics are displayed in Table 1.

LUTs (6-to-1 lookup tables)	69,120
Flip-flops (1-bit registers)	69,120
BRAM (total kbit)	5,328
BRAM (dual-ported 36-kbit blocks)	148
BRAM (single-ported 18-kbit blocks)	296
Process	65 nm
Transistors	1.1 billion

Table 1: Xilinx Virtex-5 FPGA (XC5VLX110T)

All of the software experiments were run on an Intel Xeon *2.26 GHz* server processor (Gainestown also known as Nehalem-EP). Further processor specifications are listed in Table 2.

Number of Cores	4@2.26 GHz
L1 data & instr.	32 KiB
L2 unified cache	256 KiB
L3 shared cache	8 MiB
Process	45 nm
Transistors	731 million

Table 2: Intel Xeon Gainestown CPU (L5520).