Massively Parallel Skyline Computation For Processing-In-Memory Architectures

Abstract

Processing-In-Memory (PIM) is an increasingly popular architecture aimed at addressing the 'memory wall' crisis by prioritizing the integration of processors within DRAM. It promotes low data access latency, high bandwidth, massive parallelism, and low power consumption. The skyline operator is a known primitive used to identify those multi-dimensional points offering optimal trade-offs within a given dataset. For large multidimensional dataset, calculating the skyline is extensively compute and data intensive. Although, PIM systems present opportunities to mitigate this cost, their execution model relies on all processors operating in isolation with minimal data exchange. This prohibits direct application of known skyline optimizations which are inherently sequential, creating dependencies and large intermediate results that limit the maximum parallelism, throughput, and require an expensive merging phase.

In this work, we address these challenges by introducing the first skyline algorithm for PIM architectures, called DSky. It is designed to be massively parallel and throughput efficient by leveraging a novel work assignment strategy that emphasizes load balancing. Our experiments demonstrate that it outperforms the state-of-the-art algorithms for CPUs and GPUs, in most cases. DSky achieves 2× *to* 14× *higher throughput compared to the state-of-the-art solutions on competing CPU and GPU architectures. Furthermore, we showcase DSky's good scaling properties which are intertwined with PIM's ability to allocate resources with minimal added cost. In addition, we showcase an order of magnitude better energy consumption compared to CPUs and GPUs. Despite our focus on the skyline problem, our work provides also the skeleton for a general parallel framework suitable for developing other important data processing applications on PIM systems.*

1. Introduction

The skyline computation is crucial for multi-criteria analysis on large dataset. Initially introduced as a new relational algebra operator [9], skyline has evolved beyond its original specification due to its close resemblance to Pareto Optimality to support applications ranging from data exploration [10], database preference queries [4], route planning [19], multi-objective optimization [32], web information [28] and user recommendation systems [6]. Computing the skyline requires identifying all the points from a given dataset D that are not dominated by any other point within it. A point p dominates another point q, if it is equal or better on all dimensions and there exists at least one dimension for which it is strictly better. In order to identify the dominance relationship between two points, it is common to perform a Dominance Test (DT) [11] by comparing all their attributes/dimensions.

	CPU	GPU	PIM
Cores (c)	10	3584	2048
Bandwidth (GB/s)	68	480	4096
Power (W/c)	10.5	0.17	0.04

Table 1: Single node specification comparison for CPU (Xeon E5-2650), GPU (TITAN X) and PIM (UPMEM) architectures.

When the input dataset is large and multidimensional, computing the skyline is costly, since in theory each unprocessed point needs to be compared against all the existing skyline points. In order to reduce this cost, most sequential algorithms rely on established optimization techniques such as in-order processing [12] and space partitioning [9], both of which aim at reducing the total number of point-to-point comparisons.

Modern processors leverage the integration of many compute cores on a single chip to mitigate the effects of processing large dataset. This trend necessitates the redesign of popular skyline algorithms to take advantage of the additional hardware. Recent work on skyline computation relies on modern parallel platforms such as multi-core CPUs [11] and many-core GPUs [7]. These solutions attempt to address the unprecedented challenges associated with maintaining algorithmic efficiency while maximizing throughput. Despite these efforts, the widening gap between memory and processor speed contributes to a high execution time, as the maximum attainable throughput is constrained by the data movement overhead that is exacerbated by the low computation to data movement ratio evident in the core (i.e. dominance test, Section 4) skyline computation.

Processing-In-Memory (PIM) architectures [2, 13, 16, 17, 20, 23, 27, 29, 30, 34] present a viable alternative for addressing this bottleneck leveraging on many processing cores that are embedded into DRAM. Moving processing closer to where data reside offers many advantages including but not limited to higher processing throughput, lower power consumption and increased scalability for well designed parallel algorithms (Table 1). In this paper we rely on UPMEM's architecture [20] , a commercially available PIM implementation that incorporates several of the aforementioned characteristics. Our skyline implementation presents a practical use case, that captures the important challenges associated with designing complex data processing algorithms using the PIM programming model. UPMEM's architectural implementation follows closely the fundamental characteristics of previous PIM systems [2, 13, 16, 17, 20, 23, 27, 29, 30, 34], offering in addition an FPGA-based testing environment [1].

Computing the skyline using a PIM co-processor comes

with its own set of non-trivial challenges, related to both architectural and algorithmic limitations. Our goal is to identify and overcome these challenges through the design of a massively parallel skyline algorithm, that is optimized for PIM systems and adheres to the computational efficiency and throughput constraints established on competing architectures. Our contributions are summarized below:

- We outline the challenges associated with developing an efficient skyline algorithm on PIM architectures (Sections 3, 5.1, 5.2).
- We propose a nontrivial assignment strategy suitable for balancing the expected skyline workload amongst all available PIM processors (Section 5.3).
- We present the first massively parallel skyline algorithm (i.e. *DSky*), optimized for established PIM architectures (Section 5.4).
- We provide a detailed complexity analysis, proving that our algorithm performs approximately the same amount of parallel work, as in the sequential case (Section 5.4).
- We successfully incorporate important optimizations, that help maintain algorithmic efficiency without reducing the maximum attainable throughput (Section 5.4.1).
- Our experimental evaluation demonstrates 2× to 14× higher throughput (Section 6.5), good scalability (Section 6.6), and an order of magnitude better energy consumption (Section 6.7) compared to CPUs and GPUs.

Although this work deals explicitly with the skyline problem, it also presents the skeleton of a general parallel framework for PIM architectures upon which other data intensive applications can be developed. This can be achieved by a small adaptation of our proposed processing stages, namely: data partitioning (Section 5.3), local batch processing (Section 5.4) and global intermediate result merging with emphasis on masking communication latency (Section 5.4).

2. Related Work

The skyline operator was first introduced by Borzsony et al. [9], who also proposed a brute-force algorithm known as Block Nested Loop (BNL) to compute it. Sort-Filter-Skyline (SFS) [12] relied on topological sorting to choose a processing order, that maximizes pruning and reduces the overall work associated with computing the skyline set. Related variants such as LESS [15] and SALSA [5] proposed the use of optimizations like pruning while sorting the data or determining when to stop early.

Sort-based solutions are optimized towards maximizing dominance and reducing the overall work by half. However, on certain distributions where the majority of points are incomparable [22], they are proven to be less effective. In contrast, space partitioning strategies [22] have been proven to perform better at identifying incomparability.

The *BSkyTree* [21] algorithm facilitates index-free partitioning by using a single pivot point. This point is calculated iteratively during processing through the use of a heuristic that

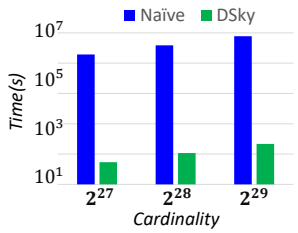


Figure 1: Runtime snapshot for 16 dimension skyline.

aims at achieving a balance between maximizing incomparability and dominance. *BSkyTree* is the current state-of-the-art sequential algorithm for computing the skyline regardless of the dataset distribution.

Despite their proven usefulness, previous optimizations cannot be easily adapted on modern parallel platforms. Related research concentrated mainly on developing parallel skyline algorithms that are able to maintain the same level of efficiency as their sequential counterparts. The *PSkyline* algorithm [25] is based on the Branch & Bound Skyline (BBS) and exploits multi-core architectures to improve performance of the sequential BBS. For data distributions that are more challenging to process, it creates large intermediate results that require merging which causes a noticeable drop in performance. *BSkyTree-P* [21] is a parallel variant of the regular *BSkyTree* algorithm. Although, generally more robust on challenging data distributions, *BSkyTree-P* is also severely restricted during the merging of intermediate results, an operation that entails lower parallelism.

The current state-of-the-art multi-core algorithm is *Hybrid* [11] and is based on blocked processing, an idea used extensively for a variety of CPU-based applications to achieve good cache locality. Sorting based on a monotone function is used to reduce the total workload by half. For more challenging distributions, the algorithm employs a simple space partitioning mechanism, using cheap filter tests which effectively reduce the cost for identifying incomparable points. *Hybrid* is specifically optimized for multi-core platforms, the performance of which depends heavily on cache size and memory bandwidth. Data distributions that generate an arbitrarily large skyline limit processing performance. Therefore, multi-core CPUs are limited when it comes to large scale skyline computation.

Accelerators present the most popular solution when dealing with data parallel applications such as computing the skyline set. Previous solutions include using GPUs [7] or FPGAs [33]. The FPGA solution relies on streaming to implement a variant of BNL. Although, it showcases better performance compared to an equivalent software solution, it is far from the efficiency achieved by *Hybrid*. On GPUs, the current state-of-the-art algorithm is *SkyAlign* [7]; it aims at achieving work-efficiency through the use of a data structure that closely resembles a quad tree. *SkyAlign* strives towards reducing the overall workload at the expense of lower throughput that is caused by excessive thread divergence. Furthermore, load balancing issues and irregular data accesses coupled with restrictions in

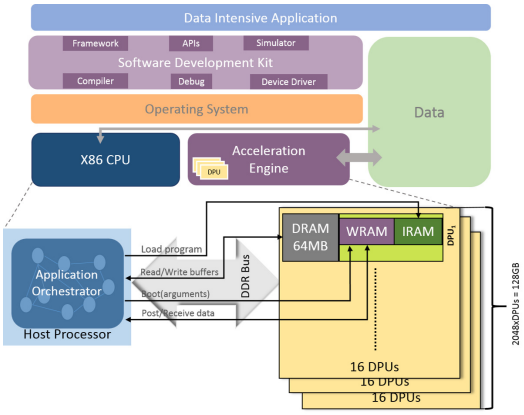


Figure 2: UPMEM's PIM Architecture Overview

memory size and bandwidth result in significant performance degradation when processing large dataset.

Our solution is based on PIM architectures which rely on integrating a large collection of processors in DRAM. This concept offers higher bandwidth, lower latency and massive parallelism. In short, it is perfectly tailored for computing the skyline, a data intensive application. In UPMEM's PIM architecture, each processor is isolated having access only to their local memory. This restriction makes previously proposed parallel solutions and their optimizations nontrivial to apply. In fact, our initial attempts to directly apply optimizations used in the state-of-the-art CPU and GPU solutions on UPMEM's PIM architecture, resulted in noticeable inferior performance (Figure 1). We attribute this behavior to low parallelism, unbalanced workload assignment and a high communication cost. In the following sections, we discuss these challenges in detail and describe how to design a parallel skyline algorithm suitable for this newly introduced architecture.

3. Architecture Overview & Challenges

UPMEM's Processing-In-Memory (PIM) technology promotes integration of processing elements within the memory banks of DRAM modules. UPMEM's programming model assumes a host processor (CPU), which acts as an orchestrator performing read/write operations directly to each memory module. Once the required data is in-place, the host may initiate any number of transformations to be performed on the data using the embedded co-processors. This data-centric model favors the execution of fine grained data-parallel tasks [20]. Figure 2 illustrates the UPMEM's PIM architecture.

A 16 GBs UPMEM DIMM contains 256 embedded processors called Data Processing Units (*DPUs*). Depending on the number of DIMMs, it is possible to have hundreds of *DPUs* operating in parallel. Each one owns 64 MBs which are part of the DRAM, referred to as Main RAM(*MRAM*). The UPMEM DPU is a triadic RISC processor with 24 32-bits registers per thread. The *DPU* processors are highly multi-threaded, supporting a maximum of 24 threads. Fast context switching allows for effective masking of memory access latency[1]. Dedicated Instruction RAM (IRAM) allows for individual *DPUs* to execute their own program as initiated by the host. Additionally, each *DPU* has access to a fast working memory (64 KB) called Work RAM (WRAM), which is used as a cache/scratchpad memory during processing and is globally accessible from all active threads running on the same *DPU*. This memory can be used to transfer blocks of data from the MRAM and is managed explicitly by the application.

From a programing point of view, two different implementations must be specified: (1) the host program that will dispatch the data to the co-processors' memory, sends commands, and retrieves the results, and (2) the *DPU* program/kernel that will specify any transformations that need to be performed on the data stored in memory. The UPMEM architecture offers several benefits over conventional multi-core chips including but not limited to increased bandwidth, low latency and massive parallelism. For a continuously growing dataset, it can offer additional memory capacity and proportional processing throughput since new DRAM modules can be added as needed.

PIM systems promote a data-centric processing model [14] that offers the potential to improve performance for many data parallel applications. However, this technology is rather an enabler than a solution, especially in the context of computing the skyline. The best practices established for CPU- or GPU-centric processing are not directly applicable to PIM systems [30]. For example, in-order processing, although useful for reducing complexity, creates dependencies that limit parallelism and subsequently lower throughput. Furthermore, relying on globally accessible space partitioning data structures [11], results in excessive communication with the host CPU nullifying any benefits offered by PIM systems.

Although PIM architectures resemble a distributed system, they are far from being one since they do not allow for direct communication between DPUs (i.e. slave-nodes). For this reason, algorithms relying on the MapReduce framework [26] are not directly applicable since they will involve excessive bookkeeping to coordinate execution and necessary data exchange for each DPU. Additionally, the MapReduce framework involves only a few stages of computation (i.e. chained map-reduce transformations) which may not be enough to effectively mask communication latency when the intermediate results between local skyline computations are prohibitively large. Despite these limitations, we can still rely on Bulk Synchronous Processing (BSP) to design our algorithm, giving greater emphasis on good partitioning strategies that provide opportunities to mask communication latency and achieve load balancing. The most prominent solutions in that field include the work of Vlachou et al. [31] and Köhler et al. [18]. Both advocate towards partitioning the dataset using each points' hyperspherical coordinates. Although, this methodology is promising, it does not perform well on high dimensional data (i.e. $d > 8$), because it creates large local skylines, resulting in a single expensive merging phase [24]. Additionally, calcu-

[1] Switching is performed at every clock cycle between threads

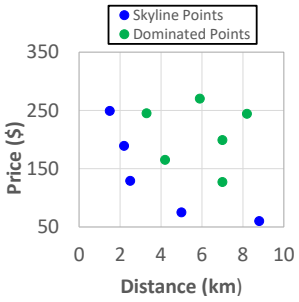


Figure 3: Skyline set on toy dataset (hotel price vs distance).

lating each points' hyperspherical coordinates is a computationally expensive step [18]. For this reasons, we purposefully avoid using the aforementioned partitioning schemes. Instead, we present a simpler partitioning scheme which emphasizes load balancing and masking communication latency during the merging of all intermediate results.

4. Skyline Definitions

We proceed with the formal mathematical definition of the skyline operator. Let D be a set of d-dimensional points such that $p \in D$ and $p[i] \in \mathbb{R}$, $\forall i \in [0, d-1]$. The concept of dominance between two points is used to identify those that are part of the skyline set. As mentioned, a point p *dominates* a point q, if it has *"better"* or equal value for all dimensions and there exists at least one dimension where its value is strictly *"better"*. The meaning of *"better"* corresponds to the manner in which we choose to rank the values for each dimension, being smaller or larger, although the ranking should be consistent amongst all dimensions. For this work, we regard smaller values as better, therefore the mathematical definition of dominance becomes:

Dominance: Given $p, q \in D$, p dominates q, written as $p \prec q$ if and only if $\forall i \in [0, d-1]$ $p[i] \leq q[i]$ and $\exists j \in [0, d-1]$ such that $p[j] < q[j]$.

Any point that is not dominated from any other in the dataset, will be part of the skyline set (Fig. 3) and can be identified through a simple comparison called Dominance Test (DT).

Skyline: The skyline S of set D is the collection of points $S = \{\forall p \in D | \nexists q \in \ s.t\ q \prec p\}$.

Clearly $S \subseteq D$. The definition of *dominance* acts as the basic building block for designing skyline algorithms. The BNL algorithm relies naïvely on brute force to compute the skyline set. This method is quite inefficient, resulting in $O(n^2)$ DTs and a proportional number of memory fetches. To avoid unnecessary DTs, previous solutions used in-order processing based on a user defined monotone function. It considers all query attributes, reducing the point to a single value that can be used for sorting. Such a function is formally defined as:

Monotone Function: A monotone scoring function F with respect to $\mathbb{R}^d$ takes as input a given point $p \in D$ and maps it to $\mathbb{R}$ using k monotone increasing functions $(f_1, f_2, ...f_k)$. Therefore, for $p \in D$, $F(p) = \sum_{i=1}^{k} f_i(p[i])$.

The ordering guarantees that points which are already determined to be part of the skyline, will not be dominated by any other which are yet to be processed. This effectively reduces the number of DTs by half.

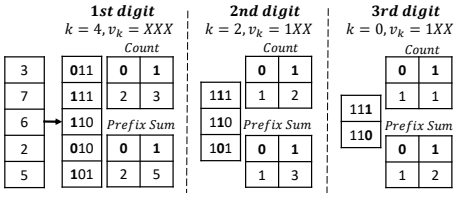


Figure 4: Radix-select example using radix-1.

Another important optimization aimed at reducing the total number of DTs uses a so-called stopping point [5] to determine when it is apparent that no other point is going to be added in the skyline. Thus a number of DTs are avoided by stopping early. Each time a new point is added to the skyline, it is checked to see if it can be used as a stopping point. Regardless of the chosen monotone function, we can optimally select that point using the following update MiniMax [5] equation:

$$\mathbf{p_s} = \underset{p_i \in S}{\operatorname{argmin}} \left\{ \max_{j \in [0, d-1]} \{p_i[j]\} \right\} \tag{1}$$

5. DSky Algorithm Overview

We present a high level overview of our novel algorithm which we call *DPU Skyline* (DSky), followed by a detailed complexity analysis. The algorithm operates in two stages, the *preprocessing* stage where points are grouped into blocks/partitions and assigned to different *DPUs*, and a *main processing* stage spanning across multiple iterations within which individual blocks are compared in parallel against other previously processed blocks.

5.1. Parallel Radix-Select & Block Creation

Maintaining the efficiency of sequential skyline algorithms, requires processing points in-order based on a user-defined monotone function. Due to architectural constraints, sorting the input to establish that order, contributes to a significant increase in the communication cost between host and *DPUs*. Our algorithm relies on parallel radix-select [3] to find a set of pivots which can be used to split the dataset into a collection of blocks/partitions. Radix-select operates on the ranks/scores that are generated for each point from a user defined monotone function. In our implementation, we assume the use of L_1 norm. Computing the rank of each point is relatively inexpensive, highly parallel and can be achieved by splitting the data points evenly across all available *DPUs*.

Radix-select closely resembles radix-sort, in that it requires grouping keys by their individual digits which share the same significant position and value. However, it differs as its ultimate goal is to discover the k-th largest element and not sort the data. This can be accomplished by building a histogram of the digit occurrences, for each significant position across multiple iterations, and iteratively construct the digits for the k-th largest element. An example for $k = 4$ is shown in Fig. 4. The digits are examined in groups of 1 (i.e. radix-1) starting

from the most significant digit (MSD). At the first iteration, there are 2 and 3 occurrences of 0 and 1, respectively. The prefix sum of these values indicates that the 4-th element starts with 1. We update k by subtracting the number of elements at the lower bins. This process repeats at the next iteration for elements that match to $1XX$. After 3 iterations the k-th largest value will be $v_k = 110$.

The pseudocode for the *DPU* kernel corresponding to radix-select is shown in Algorithm 1. In our implementation, we use radix-4 (i.e. examine 4 digits at a time) which requires 16 bins per thread. For 32-bit[2] values, we require 8 iterations that consist of two phases. First, each *DPU* thread counts the digit occurrences for a given portion of the data. At a given synchronization point the threads cooperate to accumulate partial results into a single data instance. In the second phase, the host will gather all intermediate results and calculate the corresponding digit of the k-th value while also updating k. The new information is then made available to all *DPUs* at the next iteration. This whole process is memory bound, although highly parallel and with a low communication cost (i.e. only few KB need to be exchanged), fitting nicely to the PIM paradigm. Therefore, it is suitable for discovering the splitting points between partitions.

Algorithm 1 Radix-select Kernel

R = *Precomputed Rank vector.*
K = *Splitting Position.*
V_k = *Digits of Current Pivot.*

1: **for** $digit \in [7,0]$ **do**
2: *Set* $B_t = \{0\}$ ▷ Set thread bins to zero.
3: **for all** $r \in R$ *in parallel* **do**
4: **if** $prefix(r, V_k)$ **then** ▷ Match prefix.
5: $B_t[digit]++$
6: **end if**
7: **end for**
8: $B = sum(Bt)$ ▷ Aggregate Partial Counts.
9: $(V_k, K) = search(B, K)$ ▷ Update P & K.
10: **end for**

Assuming a partition size, denoted with P_{size}, and N number of points, we require $P_{vt} = P - 1 = \frac{N}{P_{size}} - 1$ pivots to create partitions $\{C_0, C_1, C_2 ... C_{P-2}, C_{P-1}\}$. In Algorithm 2, we present the pseudocode for assigning points to their corresponding partitions. As indicated in Line 3, we concentrate on the rank of a given point to identify the range of pivots that contain it, after which we assign it to the partition with the corresponding index. The presented partitioning method guarantees that no two points p, q exist, such that $p \in C_i$ and $q \in C_j$, where $i < j$ and $F(p) > F(q)$. Points within a partition do not have to be ordered with respect to their rank, given a small partition which allows for parallel brute force point-to-point comparison.

Blocked processing has been used before for CPU based skyline computation [11] to improve cache locality. Our solution differs, since it supports blocking while avoiding the high cost of completely sorting the input data. Furthermore, we utilize blocking to introduce a nontrivial work assignment strategy which enables us to design a highly parallel and throughput optimized skyline algorithm for PIM architectures. This strategy aims at maximizing parallel work through maintaining good load balance across all participating *DPUs*, as compared to the optimal case.

[2]Floating-point types can be processed through a simple transformation to their IEEE-754 format.

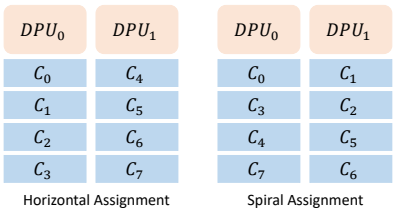


Figure 5: Assignment strategies of 8 partitions on 2 *DPUs*.

Algorithm 2 Radix-select Partitioning

D = *Input dataset*
R_p = *Pivots vector*

1: $R_p = radix\_select(D)$ ▷ Calculate pivots.
2: **for all** $p \in D$ **do**
3: **if** $R_p[j] < F(p) \leq R_p[j+1]$ **then**
4: $C_j = C_j \cup \{p\}$ ▷ Assign p to C_j.
5: **end if**
6: **end for**

5.2. Horizontal Partition Assignment

In this section, we concentrate on introducing a simple horizontal assignment strategy, the performance of which motivates our efforts to suggest a better solution. Our goal is to establish the lower bound associated with the parallel work for computing the skyline, measured in partition-to-partition (*p2p*) comparisons, and suggest a strategy along with the algorithm that is able to attain it.

We start by introducing some definitions. Given a partition C_j, we define its pruned equivalent partition, the set of all points that appear in C_j which will be eventually identified as being part of the final skyline set. We denote this pruned partition as $\widetilde{C}_j \subseteq C_j$. Assuming a collection of P partitions, which can be ordered using radix-select partitioning, such that for $i, j \in [0, P-1]$ and $i < j$, then $C_i \prec C_j$ (i.e. C_i precedes C_j), it is possible to compute P pruned partitions iteratively:

a. $\widetilde{C}_0 = p2p(C_0, C_0)$
b. $\widetilde{C}_1 = p2p(\widetilde{C}_0, p2p(C_1, C_1))$
c. $\widetilde{C}_2 = p2p(\widetilde{C}_0, p2p(\widetilde{C}_1, p2p(C_2, C_2)))$

The *p2p* function denotes a single partition-to-partition comparison operation, checking if any points exist in C_i that dominate those in C_j. More details related to the implementation of *p2p*, are presented in Section 5.4.1. We observe that using the pruned partition definition, we can calculate the skyline set using the following formula:

$$S = \bigcup_{i \in [0, P-1]} \left(\widetilde{C}_i \right) \quad (2)$$

Eq. 2, indicates that it is possible to compute the skyline us-

ing the union of all pruned partitions. Therefore, it is possible to maintain and share information about the skyline *without* using a centralized data structure. Additionally, once $\widetilde{C}_j$ is generated, all remaining partitions with index larger than j may use it to prune points from their own collection. In fact, performing this work is "embarrassingly" parallel and depending on the partition size and the input dataset size, it can be scaled to utilize thousands of processing cores. However, we observe that assigning work to DPUs naïvely could potentially hurt performance, due to the apparent dependencies between partitions and the fact that latter partitions require more *p2p* comparisons to be pruned.

Assuming all partitions are processed in sequence, we can calculate the number of total *p2p* comparisons by examining each partition separately. For example, C_0 will need 1 self-comparison (i.e. $p2p(C_0,C_0)$), C_1 will need 2 *p2p* comparisons, C_2 3 and so on. In fact, the total number of *p2p* comparisons, assuming P partitions is given by the following equation:

$$M_{seq} = \frac{P \cdot (P+1)}{2} \tag{3}$$

Ideally, with D_p *DPUs* at our disposal, we would like to evenly distribute the workload among them, maintaining a *p2p* comparison count which is roughly equal to $\frac{M_{seq}}{D_p}$. A fairly common and easily implementable strategy, is to divide the partitions ($P_D = \frac{P}{D_p}$ per *DPU*) horizontally across *DPUs* as indicated in Figure 5. However, if we attempt to follow this strategy, the *DPU* responsible for the last collection of partitions will have to perform at least $(P - P_D) \cdot P_D + \frac{P_D \cdot (P_D+1)}{2}$ *p2p* comparisons a number $P \cdot P_D$ times higher than the DPU responsible for the first collection of partitions. Obviously, this assignment mechanism suffers from several issues, the most important of which is poor load balancing. In fact during processing, the majority of the participating *DPUs* will be idle waiting for pruned partitions to be calculated and transmitted. Additionally, the limited memory space available to each DPU, makes it hard to amortize the cost of communication, since processing needs to complete before exchanging any data. To overcome the problems set forth by horizontal partitioning, we introduce the concept of *spiral partition* assignment.

5.3. Spiral Partition Assignment

Commonly, data intensive algorithms rely on Bulk Synchronous Processing (BSP) to iteratively apply transformations on a given input across many iterations, between which a large portion of the execution time is dedicated to processing rather than communication. This process aims to maintain good load balance and reducing communication to effectively minimize each processor's idle time. In this section, we introduce a nontrivial assignment strategy which allows for the design of an iterative algorithm that follows the aforementioned properties.

Our assignment strategy relies on the observation that for a collection of $2 \cdot D_p$ ordered partitions with respect to a user-provided monotone function, we can always group them together creating non-overlapping pairs, all of which when processed individually, require the same *p2p* comparison count. The pairing process considers partitions in opposite locations with respect to the monotone function ordering, resulting in the creation of D_p pairs in total. For example, assuming the existence of partitions $\{C_0, C_1 ..., C_{2 \cdot D_p - 1}\}$, we will end up with the following pairs:

$$\left\{\left\langle C_0, C_{2 \cdot D_p - 1}\right\rangle, \left\langle C_1, C_{2 \cdot D_p - 2}\right\rangle ..., \left\langle C_{D_p - 1}, C_{D_p}\right\rangle\right\} \tag{4}$$

In Figure 5, we showcase our novel assignment strategy, which we call *spiral partitioning*, next to the naïve horizontal partitioning scheme. In contrast to the horizontal partitioning mechanism which requires $4 \cdot 4 + \frac{4 \cdot 5}{2} = 26$ *p2p* comparisons from a single *DPU*, our spiral partitioning scheme requires only 18 (i.e., $(1+4+5+8) = DPU_0$, $(2+3+6+7) = DPU_1$) most of which can be performed in parallel. This number is equivalent to $\frac{M_{seq}}{D_p} = \frac{36}{2}$, indicating that our spiral partitioning strategy splits evenly the expected workload across all participating *DPUs*.

In our analysis, we assumed the number of partitions P to be equal to $2 \cdot D_p$. In the general case, we can choose P and D_p, in order for P to be expressed as multiple of $2 \cdot D_p$ such that $K = \frac{P}{2 \cdot D_p}$. For each one of the K collections, we can individually apply the spiral partitioning algorithm and assign one pair from each collection to a distinct *DPU*. Following this assignment process, we calculate the total p2p comparison count per *DPU* based on the following formula:

$$\begin{gathered} M_{opt} = (1 + 2 \cdot D_p) + (1 + 6 \cdot D_p) + \\ (1 + 10 \cdot D_p) + ... = D_p \cdot (2 + 6 + 10 + 14...) + \\ \frac{P_D}{2} = D_p \cdot \frac{(4 + 4 \cdot (\frac{P_D}{2} - 1))}{2} \cdot \frac{P_D}{2} + \frac{P_D}{2} = \\ \frac{P_D}{2} \cdot \left[2 \cdot \frac{P_D}{2} \cdot D_p + 1\right] = \frac{P}{2 \cdot D_p} [P+1] \Rightarrow \\ M_{opt} = \frac{P \cdot (P+1)}{2 \cdot D_p} \end{gathered} \tag{5}$$

The aforementioned formula is based on the observation that for each collection, the number of *p2p* comparisons per *DPU* is equal to the *p2p* comparisons required for the first and last partition of that collection. Therefore, for the first collection we need $1 + 2 \cdot D_p$ *p2p* comparisons, for the second $2 \cdot D_p + 1 + 4 \cdot D_p$, for the third $4 \cdot D_p + 1 + 6 \cdot D_p$ and so on.

In theory, it is possible to utilize at most $\frac{P}{2}$ *DPUs* for processing when using spiral partitioning. However, in practice, it might not be beneficial to reach this limit, since at that point the work performed within each *DPU* will not be enough to amortize the cost of communication or minimize the idle time. Additionally, due to the existing dependencies between partitions, increasing the number of *DPUs* will result in less work being performed in parallel. In the next section, we present more details regarding these issues and present the intrinsic characteristics of our main algorithm.

Partitions/Iteration	$i=0$	$i=1$	$i=2$
$\{C_0, C_3\}$	$\tilde{C}_0$: (C_3)	$\tilde{C}_1$: (C_3)	$\tilde{C}_2$: (C_3)
$\{C_1, C_2\}$	$\tilde{C}_0$: (C_1, C_2)	$\tilde{C}_1$: (C_2)	–

(A)

Partitions/Iteration	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
$\{C_0, C_3, C_4, C_7\}$	$\tilde{C}_0$: (C_3, C_4, C_7)	$\tilde{C}_1$: (C_3, C_4, C_7)	$\tilde{C}_2$: (C_3, C_4, C_7)	$\tilde{C}_3$: (C_4, C_7)	$\tilde{C}_4$: (C_7)	$\tilde{C}_5$: (C_7)	$\tilde{C}_6$: (C_7)
$\{C_1, C_2, C_5, C_6\}$	$\tilde{C}_0$: (C_1, C_2, C_5, C_6)	$\tilde{C}_1$: (C_2, C_5, C_6)	$\tilde{C}_2$: (C_5, C_6)	$\tilde{C}_3$: (C_5, C_6)	$\tilde{C}_4$: (C_5, C_6)	$\tilde{C}_5$: (C_6)	–

(B)

Figure 6: Number of comparisons across iterations when assigning (A) 2 partitions per *DPU* vs (B) 4 partitions per *DPU*

5.4. DSky Main Processing Stage

Leveraging on spiral partitioning, we introduce a new algorithm for computing the skyline set on PIM architectures. Once each partition has been assigned to their corresponding *DPU*, we can start calculating each pruned partition within two distinct phases as indicated in Algorithm 3. In the first phase, each *DPU* performs a "self-comparison" for all partitions assigned to it. This step is "embarrassingly" parallel and does not require any data to be exchanged. The second phase consists of multiple iterations across which the pruned partitions are computed. At iteration i, the pruned partition $\widetilde{C}_i$ has already been computed and is ready to be transmitted across all *DPUs*. Once the broadcast is complete, all *DPUs* have access to $\widetilde{C}_i$ which they use as a window to partially prune any of their own C_j partitions in parallel, where $j > i$ is based on the established ordering of partitions.

Our implementation uses a collection of flags, denoted with F_i for partition $\widetilde{C}_i$, to mark which points have been dominated during processing. We indicate with 0 those points that have been pruned away and with 1 those that are still tentatively skyline candidates. The whole process is orchestrated by the host (CPU), who keeps track of which partition needs to be transmitted at the end of each iteration. It is important to note that broadcasting individual partitions can be expensive. For this reason, we need to carefully choose the partition size in order to overlap data exchange with actual processing. Additionally, we propose to further reduce this cost by preemptively broadcasting m partitions at each iteration before they are actually needed, thus increasing the computation-communication overlap window. Nevertheless, we still need to wait for the F_i bit-vector to become available before starting the next iteration. However, once the corresponding F_i bit-vector is calculated we can inexpensively transmit it to all *DPUs*, since it is inversely proportional to the point dimensions and partition size.

Assuming an optimal *p2p* kernel, we measure the complexity of DSky in terms of *p2p* comparisons per *DPU*. For the first phase, each *DPU* is responsible for self-comparing their assigned partitions, requiring P_D comparisons to complete. The second stage is slightly more complex. Within iteration i, the corresponding partition $\widetilde{C}_i$ will be compared against all C_j partitions having a higher index. Starting from $\widetilde{C}_0$ and for the next $D_p - 1$ iterations, each *DPU* will perform P_D comparisons. Once $\widetilde{C}_{D_p}$ is computed, only partitions with index larger than D_p will need to be considered, resulting in at most $P_D - 1$ comparisons for iterations D_p to $2 \cdot D_p - 1$. This process is repeated multiple times until all partitions within each *DPU* have been checked. Adding the complexity of each phase together, we end up with the following formula:

$$\begin{aligned} M_{par} = [(D_p - 1) \cdot P_D + D_p \cdot (P_D - 1) + \\ D_p \cdot (P_D - 2) \ldots + D_p \cdot 1] + P_D \Rightarrow \\ M_{par} = D_p \cdot [\frac{P_D \cdot (P_D + 1)}{2}] \end{aligned} \tag{6}$$

Algorithm 3 DSky Algorithm

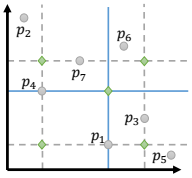
B_j = Region bit vectors for C_j.
F_j = Flags indicating active skyline points for C_j.

```
1:  for all DPUs in parallel do
2:      for all C_j ∈ DPU_i do
3:          P2P(C_j, B_j, C_j, B_j)            ▷ Self compare partitions.
4:      end for
5:  end for
6:  for all i ∈ [0, P − 1] do
7:      copy(C̃_i, B̃_i, F_i)                    ▷ Broadcast pruned partition info.
8:      for all DPUs in parallel do
9:          for all j > i do
10:             P2P(C̃_i, B̃_i, C_j, B_j)        ▷ Prune C_j using C̃_i
11:         end for
12:     end for
13: end for
```

From Eq. 6 and Eq. 5, if we replace $P_D = \frac{P}{D_p}$, we get the following ratio:

$$\frac{M_{par}}{M_{opt}} = \frac{1 + \frac{D_p}{P}}{1 + \frac{1}{P}} \tag{7}$$

From Eq. 7, we can observe how different values for P and D_p affect the complexity of *DSky* with respect to the optimal case. When $P \to \infty$, then $\frac{M_{par}}{M_{opt}} \to 1$. Intuitively, when the number of partitions assigned per *DPU* is significantly larger than its collective number, the observed idle time constitutes a smaller portion of the actual processing time. In Figure 6 using two *DPUs*, we present an example where 2 or 4 partitions are assigned per *DPU*. In the first case, we require 3 p2p comparisons and within iterations $i = 0, 2$, DPU_0 or DPU_1 will do 1 less comparison than the other, respectively. Therefore, $\frac{1}{4}$ of the time each *DPU* will be idle. In the second case, the total comparisons across iterations will be 14 and the corresponding idle time within iterations $i = 0, 2, 4, 6$ is 2. Hence, the idle time per *DPU* will be $\frac{2}{16}$, half of what was observed for the previous example. At this point, it is important to note that creating more partitions does not depend on the input size,



	p_1	p_2	p_3	p_4	p_5	p_6	p_7
$Level_1$	10	01	10	01	10	11	01
$Level_2$	00	01	11	00	10	01	11

Figure 7: Median pivot multi-level partitioning example.

but instead on the number of pivots calculated during radix-select partitioning. Although, this may seem like having a partition size equal to 1 is the best case, in practice there are several trade-offs to consider, such as the preprocessing time required to calculate each partition and the communication overhead when small data are transmitted frequently and not in bulk. Through experimentation, we are able to identify the specific parameters contributing to these trade-offs, allowing us to successfully fine tune the partition size.

5.4.1. P2P Kernel In this section, we discuss three specific optimizations that can be integrated into our *p2p* kernel to ensure algorithmic efficiency. Although, their application on PIM systems created unprecedented challenges, our novel assignment strategy made possible to overcome them.

Optimization I: The points within each partition are sorted based on their rank. This optimization can be embarrassingly parallel and less expensive than globally sorting all the points. It aims at reducing the expected number of DTs for each DPU by half [11].

Algorithm 4 P2P Function Kernel

R_j = *Rank vector for* C_j.
B_j = *Region bit vectors for* C_j.
F_j = *Flags indicating active skyline points for* C_j.
(g_s, p_s) = *Global stop level and point.*

```
1: if stop(g_s,p_s,R_j[0],C_j[0]) then
2:     return F_j ← 0                                  ▷ Prune partition.
3: end if
4: for all q ∈ C_j in parallel do
5:     if F_j[q] ≠ 0 then                              ▷ q is alive.
6:         for all p ∈ C_i do
7:             if F_i[p] ≠ 0 then                      ▷ p is alive.
8:                 if B_i[p] ⊀ B_j[q] then             ▷ Incomparable.
9:                     continue
10:                end if
11:                if p ≺ q then
12:                    F_j[q] ← 0                      ▷ Set flag for q to zero.
13:                    break
14:                end if
15:            end if
16:        end for
17:    end if
18:    if F_j[q] = 1 then                              ▷ Point is not dominated.
19:        l_s[id] = MiniMax(q,l_s[id])                ▷ Thread stop level.
20:        p_s[id] = q                                 ▷ Thread stop point.
21:    end if
22: end for
23: (g_s, p_s) = update_ps(l_s[id], p_s[id])           ▷ DPU stop info.
24: merge F_j
```

Optimization II: For more challenging distributions (i.e. anti-correlated), space partitioning is preferable since it can help with identifying incomparable point through cheap filter tests [11]. Similarly to previous work [7], we exploit a recursive space partitioning scheme to detect incomparability. This technique requires calculating bit-vectors for each point, indicating the region of space it resides. They are determined through a virtual pivot, constructed from the median value of its subspace.

An example of this is shown in Figure 7. There, we determine the values for the median level virtual pivot by taking the projection of p_1 in the x-axis and p_4 in the y-axis. Each point is assigned a bit vector based on its relative position to the virtual point. For example, p_1 is assigned 10 because it is $\geq$ and $<$ in the x and y-axis, respectively, compared to the pivot. For each quartile, we can repeat this process multiple times. However, it has been shown empirically [7] that doing it twice is sufficient to gain good algorithmic efficiency. We use radix-select to calculate the median value for each subspace and construct the corresponding pivots.

In related work [7], a centralized data structure is used to manage the bit vectors and establish a good order of processing. Due to architectural limitations (i.e. expensive global access), our implementation uses a flat array to pack both bit vectors in a single 32-bit value for each point. Our spiral partitioning scheme is responsible for maintaining the good order of processing. Additionally, it is designed around optimizing local access and minimizing communication while, also, promoting the seamless incorporation of the bit vector information within a partition.

Optimization III: Based on the work in [5], we use Eq. 1 to update the stopping level and point, and then compare this information with the point of the smallest rank within each partition to determine if it is dominated. Due to lack of space, we omit details on why this optimization works, although we discuss how it can be applied in our paradigm. The stopping information is updated locally within each *DPU*. The host is responsible for merging the local results at each step of *DSky's* second stage (Algorithm 3). This process requires only a few KBs to be exchanged, thus its communication overhead is low.

Algorithm 4 presents the implementation of our *p2p* kernel. Each *DPU* allocates memory for P_D partitions, plus two remote partitions to support double buffering. In Line 1, we compare the smallest rank within the given partition to the global stopping value to determine if the whole partition is dominated. When this test fails, we need to check all the points within the partition. For each point in the local partition, we only examine the points that are still skyline candidates (Line 5) against those of the remote partition that satisfy the same property (Line 7). Using the corresponding bit vectors, if the two points are incomparable (Line 8) we skip to the next point in the remote partition, otherwise we need to perform a full DT (Line 11). For all points in the local partition that are not dominated (Line 18), we update the local stop point information. At the end of the for-loop (Lines 23 – 24), we merge the

local stop point information and update the local partition's flags to indicate which points have been dominated.

Framework generality: Note that the intrinsic characteristics of PIM architectures imply the need to define three main stages in order to develop an efficient PIM algorithm: (i) data partitioning, (ii) local batch processing, and (iii) intermediate result merging. These stages are pivotal for overcoming the physical or logical isolation of the embedded processors, inherent to most prominent PIM systems [14]. In fact, these stages are interchangeable with only minor variations when it comes to developing algorithms for other data intensive applications (i.e. Top-K, Group-by aggregation, Joins etc.). An important difference between such applications and the skyline problem, is that some of the former might not produce large intermediate results, which require careful design of the third stage in order to deal with the excessive communication cost. However, the first two stages are universally applicable to various data intensive applications, that seek to achieve effective load balancing and massive parallelism on PIM. Hence our analysis can serve as a general parallel framework upon which solutions for related analytical workloads can be proposed.

6. Experimental Evaluation

In this section, we present an in-depth analysis of *DSky*, comparing against the state-of-the-art sequential [21], multi-core [11] and many-core [7] algorithms.

6.1. Setup Configuration

CPU Configuration: For the CPU algorithms, we conducted experiments on an Intel Xeon E5-2650 2.3 GHz CPU with 64 GB memory. We used readily available C++ implementations of *BSkyTree* [21] and *Hybrid* [8].

GPU Configuration: For the GPU, we used the latest NVIDIA Titan X (Pascal) 1.53 GHz 12 GB main memory GPU with CUDA 8.0. We conducted experiments using the readily available C++ implementation of *SkyAlign* [8] which is the current state-of-the-art algorithm for GPUs. For a fair comparison, we present measurements using clock frequencies 0.75 and 1.53 GHz.

DPU Configuration: We implemented both phases of *DSky*, including the preprocessing steps, using UPMEM's C-based development framework [1] and dedicated compiler. Our experiments were performed on UPMEM's Cycle Accurate Simulator (CAS) using the binary files of the corresponding implementation. The simulation results were validated using an FPGA implementation [1] of the *DPU* pipeline. Based on the reported clock cycle count that includes pipeline stalls associated with the corresponding data accesses, and a base clock of 0.75 GHz for each *DPU*, we calculated the exact execution time for a single node system using 8 to 4096 *DPUs*. For a fair comparison against the GPU, we limit the number of *DPUs* in accordance to the available cuda cores (i.e. 3584).

6.2. Dataset

Similarly to previous work [7], we rely on the standard skyline dataset generator [9] to create common data distributions (i.e., correlated, independent, anticorrelated). We compare against the CPU and GPU implementations using queries with dimensionality $d \in \{4, 8, 16\}$ and for dataset of cardinality $n \in [2^{20}, 2^{26}]$ [3]. Additional experiments are presented on PIM only for cardinality $n \in [2^{20}, 2^{29}]$.

6.3. Experiments & Metrics

For all implementations, our measurements include the cost of preprocessing and data transfer (where it is applicable) across PCIE 3.0 (i.e. GPU) or broadcast between *DPUs*. We benchmarked the aforementioned algorithms with all of their optimizations enabled. For the performance evaluation, we concentrate on the following metrics:

Runtime Performance: This metric is used to evaluate at a high level the performance of *DSky* against previous solutions. It showcases the overall capabilities of the given architecture coupled with the chosen algorithm.

Algorithmic Efficiency & Throughput: Due to several hidden details within the runtime performance, we focus on the algorithmic efficiency by studying the number of full DTs conducted by each algorithm. Our ultimate goal is to showcase the ability of *DSky* to successfully incorporate known skyline optimizations and indicate their contribution towards achieving high throughput on the UPMEM-PIM architecture.

Scaling: An important property of the UPMEM-PIM architecture is the ability to easily increase resources when the input grows beyond capacity. However, doing so requires a well designed parallel algorithm that avoids any unnecessary overheads caused by excessive communication or load imbalance. With this metric, we indicate *DSky's* ability to scale when resources increase proportionally to the input size.

In addition, our experiments on comparing the system utilization between GPU and PIM architectures, indicated an upward trend of 75% for PIM against 40% for GPUs (figures omitted due to lack of space). Moreover, we provide measurements indicating superior energy efficiency when comparing our solution to state-of-the-art algorithms on CPUs and GPUs (Section 6.7).

6.4. Run-Time Performance

Correlated data contribute to a smaller skyline set which contains only a few dominator points. Therefore, during processing the main performance bottleneck is the memory bandwidth. Figure 8 illustrates the runtime performance for all algorithms on correlated data. *DSky* outperforms previous state-of-the-art algorithms for all tested query dimensions. This happens because it relies on radix-select, an inherently memory bound operation, to lower the preprocessing cost. Moreover, the main processing stage terminates early due to the discovery of a suitable stopping point. *BSkyTree* and *Hybrid* under-utilize the available bandwidth, since a single point requires only few comparisons to be pruned away. Therefore, prefetching data into cache will result in lower computation to communica-

[3] Due to restrictions in GPU memory, the maximum dataset for comparison purposes was set to 2^{26}.

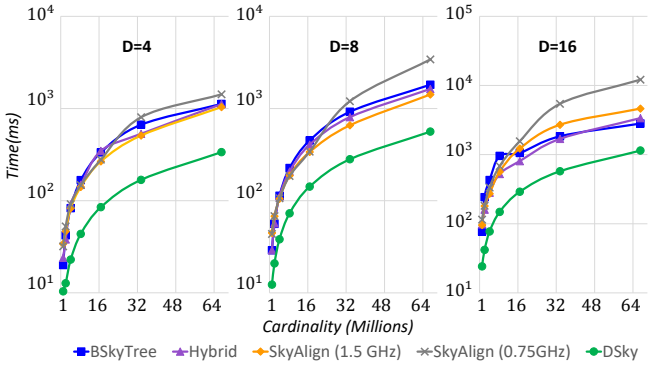


Figure 8: Execution time (log(t)) using correlated data.

tion ratio and higher execution time. *SkyAlign* is limited by the overhead associated with launching kernels on the GPU, which in this case is high relative to the cost of the processing and preprocessing stages.

Figure 9 presents the runtime performance for all methods using independent data. We observe that *DSky* outperforms previous implementations for query dimensions (i.e. $d = \{4, 8\}$) that reflect the needs of real-world applications. *Hybrid* and *BSkyTree* are restricted by the cache size, since increasing dimensionality contributes to a larger skyline. This results in a higher number of direct memory accesses leading to higher runtime. Compared to *DSky*, *SkyAlign* exhibits higher runtime on 4 and 8 dimension queries, due to achieving lower throughput as a result of irregular memory accesses and thread divergence. On 16 dimensions, these limitations have a lesser effect on runtime, due to the increased workload which contributes towards masking memory access latency when more threads execute in parallel. However, concentrating on measurements using 0.75 GHz clock frequency, we observe that *DSky* outperforms *SkyAlign* approximately by a factor of 2. Intuitively, this indicates that *DSky* is throughput efficient compared to *SkyAlign*, as the latter fails to sustain same runtime for equal specification. In fact, experiments with higher frequency indicate a trend that predicts better performance for *DSky* on sufficiently large input (beyond 16 million points *SkyAlign* would crash, probably due to implementation restrictions and limited global memory).

Finally, Figure 10 illustrates the measured runtime for anti-correlated distributions. As before, *DSky* outperforms CPU-

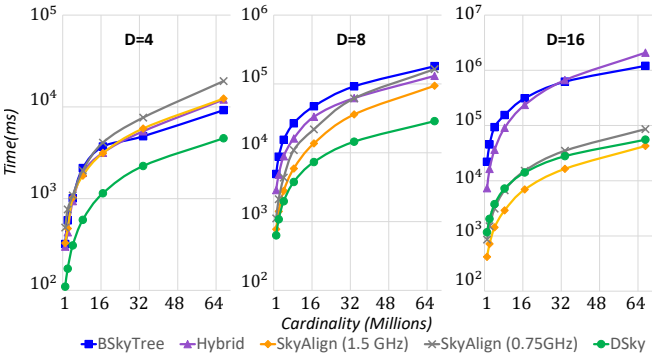


Figure 9: Execution time (log(t)) using independent data.

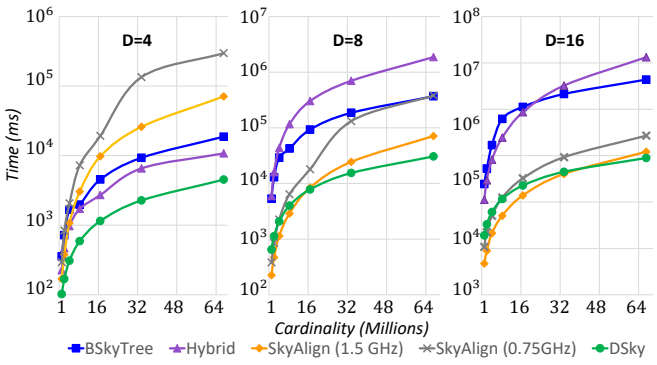


Figure 10: Execution time (log(t)) using anticorrelated data.

based methods which are restricted by the cache size. The only noticeable difference relates to the runtime of *SkyAlign* which is closer to that of *DSky* on 8 and 16 dimensions for higher clock frequency. The increased workload associated with anti-correlated distributions makes optimizing for work-efficiency a good strategy but only for a relatively small number of points.

6.5. Algorithmic Efficiency & Throughput

Figure 11 illustrates the number of full DTs performed by all algorithms. We concentrate on independent and anticorrelated distributions and omit DTs performed on correlated data as their limited number has a lesser impact on throughput. Our experiments indicate that *DSky* exhibits remarkable efficiency for queries on 8 dimensions, outperforming the state-of-the-art parallel algorithms. In fact, its performance is closer to *BSkyTree* in terms of total DT count, indicating its ability to achieve balance between efficient pruning and detecting incomparability. This results from the optimizations related to in-order processing, early stopping and cheap filter tests using space partitioning. On 16 dimensions, *DSky* remains as efficient or slightly better than the CPU-based methods. In contrast to *SkyAlign*, *DSky* requires more DTs to compute the skyline, since the former relies on a centralized data structure to decide the ordering in which points are processed. Avoiding such a data structure comes at a trade-off, which offers opportunities for high parallelism and subsequently high throughput at the expense of doing more work.

In order to support our claims, we present in Figure 12

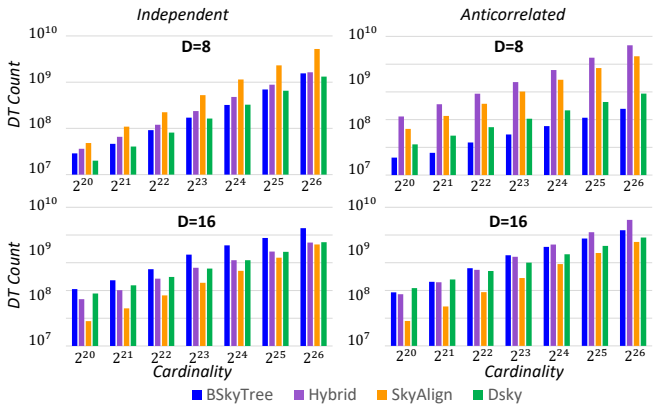


Figure 11: Number of executed DTs per algorithm.

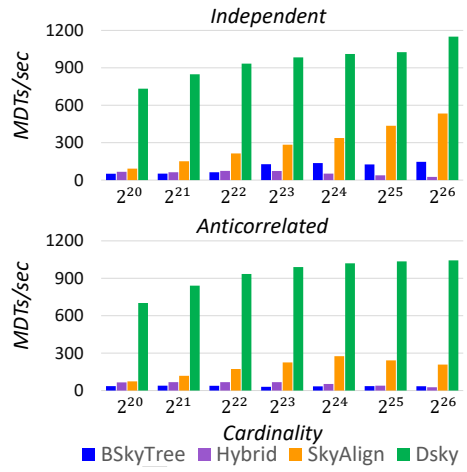


Figure 12: MDTs/sec for each algorithm on 16 dimensions.

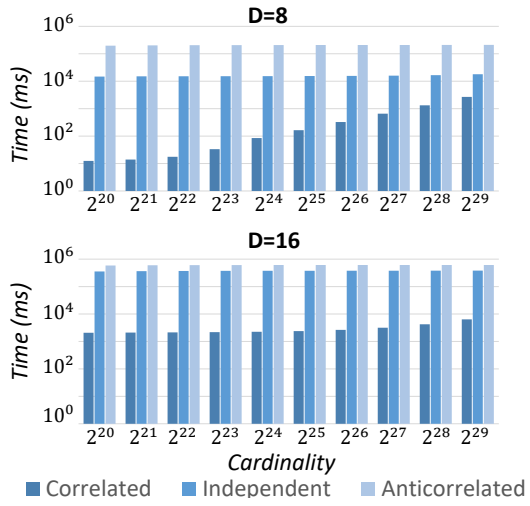


Figure 13: Execution time scaling with additional *DPUs*.

the throughput measured in million DTs per second for all implementations. We focus on the higher workload 16 dimension queries that allow for accurate throughput measurements. In our experiments, we observe that *DSky* is able to consistently maintain a higher throughput than previous state-of-the-art algorithms. Despite requiring a higher number of DTs, *DSky* maintains a higher processing rate relative to *SkyAlign* when using the same clock frequency. Intuitively, this can be attributed to a less rigid parallel execution model which allows for irregular processing, and higher bandwidth achieved through processing-in-memory. *DSky* leverages on these two properties towards being throughput efficient.

6.6. Scaling

We evaluate scalability by measuring the execution time, while the number of available *DPUs* increases proportionally (i.e. 8 to 4096) to the input size. Figure 13 contains the results of our experiments for all distributions. We focus on 8 and 16 dimension queries, which are the most compute and communication intensive case studies. Experiments with correlated data demonstrate a constant increase in execution time regardless of the query dimensions. We attribute this behavior to the higher cost of communication relative to processing. In practice, doubling the number of *DPUs* will improve performance only when the computation cost is sufficiently large. Low processing time offers minimal improvements over the increase in communication which dominates the overall execution time.

Independent and anticorrelated distributions require more time for processing than transmitting data, thus adding resources contributes to a higher reduction of the total execution time. In fact, as we increase the number of *DPUs* proportionally to the number of points, the execution time remains fairly constant regardless of the distribution or query dimension. This showcases the ability of *DSky* to scale comfortably with respect to growing input. It is also noteworthy to mention that selecting a suitable partition size, contributes to achieving good scalability. This offers more opportunities for parallelism, while minimizing the work overhead associated with dependencies which arise from in-order processing.

6.7. Energy Consumption

As seen from our experimental evaluation, in most cases *DSky* achieves same or better execution time than state of the art solutions while being more throughput efficient and easily scalable. Moreover, *DSky* runs on an architecture that uses around 25% of the energy requirements (Table 1). Overall, this translates to more than an order of magnitude better energy consumption per unit of work in comparison to the corresponding CPU and GPU solutions, as seen in Table 2.

	CPU	GPU	PIM
Independent	0.715	1.124	0.140
Anticorrelated	1.562	2.177	0.153

Table 2: Energy per unit of work (μJ/DT).

7. Conclusion

In this work, we presented a massively parallel skyline algorithm for PIM architectures, called *DSky*. Leveraging on our novel work assignment strategy, we showcased *DSky's* ability to achieve good load balance across all participating *DPUs*. We proved that by following this methodology, the total amount of parallel work is asymptotically equal to the optimal case. Furthermore, combining spiral partitioning with blocking enabled us to seamlessly incorporate optimizations that contribute towards respectable algorithmic efficiency. Our claims have been validated by an extensive set of experiments that showcased *DSky's* ability to outperform the state-of-the-art implementations for both CPUs and GPUs. Moreover, *DSky* maintains higher processing throughput and better resource utilization. In addition, we showcased that *DSky* scales well with added resources, a feature that fits closely the capabilities of PIM architectures. Finally, our solution improves by more than an order of magnitude the energy consumption per unit of work, as compared to CPUs and GPUs.

Acknowledgement: We would like to thank UPMEM for providing the SDK and related simulation tools to evaluate our algorithms.

References

[1] UPMEM SDK, 2015. http://www.upmem.com/wp-content/uploads/2017/02/20170210_SDK_One-Pager.pdf.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proc. ISCA*, pages 105–117. IEEE, 2015.

[3] T. Alabi, J. D. Blanchard, B. Gordon, and R. Steinbach. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 17:4–2, 2012.

[4] W.-T. Balke and U. Güntzer. Multi-objective query processing for database systems. In *Proc. VLDB*, pages 936–947, 2004.

[5] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *TODS*, 33(4):31, 2008.

[6] C. Beecks, I. Assent, and T. Seidl. Content-based multimedia retrieval in the presence of unknown user preferences. *Advances in Multimedia Modeling*, pages 140–150, 2011.

[7] K. S. Bøgh, S. Chester, and I. Assent. Work-efficient parallel skyline computation for the gpu. *VLDB*, 8(9):962–973, 2015.

[8] K. S. Bøgh, S. Chester, D. Šidlauskas, and I. Assent. Template skycube algorithms for heterogeneous parallelism on multicore and gpu architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 447–462. ACM, 2017.

[9] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. ICDE*, pages 421–430. IEEE, 2001.

[10] S. Chester, M. L. Mortensen, and I. Assent. On the suitability of skyline queries for data exploration. In *EDBT/ICDT*, pages 161–166, 2014.

[11] S. Chester, D. Šidlauskas, I. Assent, and K. S. Bøgh. Scalable parallelization of skyline computation for multi-core processors. In *Proc. ICDE*, pages 1083–1094. IEEE, 2015.

[12] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting: Theory and optimizations. In *Intelligent Information Processing and Web Mining*, pages 595–604. Springer, 2005.

[13] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, et al. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th international conference on Supercomputing*, pages 14–25. ACM, 2002.

[14] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. The mondrian data engine. In *Proc. ISCA*, pages 639–651. ACM, 2017.

[15] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *VLDB*, 16(1):5–28, 2007.

[16] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

[17] Q. Guo, N. Alachiotis, B. Akin, F. Sadi, G. Xu, T. M. Low, L. Pileggi, J. C. Hoe, and F. Franchetti. 3d-stacked memory-side acceleration: Accelerator and system design. In *WoNDP*, 2014.

[18] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 85–96. ACM, 2011.

[19] H.-P. Kriegel, M. Renz, and M. Schubert. Route skyline queries: A multi-preference path planning approach. In *Proc. ICDE*, pages 261–272. IEEE, 2010.

[20] D. Lavenier, J. F. Roy, and D. Furodet. DNA mapping using processor-in-memory architecture. In *Proc. BIBM*, pages 1429–1435. IEEE, 2016.

[21] J. Lee and S.-w. Hwang. Bskytree: scalable skyline computation using a balanced pivot selection. In *Proc. EDBT*, pages 195–206. ACM, 2010.

[22] K. C. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in z order. In *Proc. VLDB*, pages 279–290. VLDB Endowment, 2007.

[23] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 457–468. IEEE, 2017.

[24] A. Nasridinov, J.-H. Choi, and Y.-H. Park. A two-phase data space partitioning for efficient skyline computation. *Cluster Computing*, 20(4):3617–3628, 2017.

[25] S. Park, T. Kim, J. Park, J. Kim, and H. Im. Parallel skyline computation on multicore architectures. In *Proc. ICDE*, pages 760–771. IEEE, 2009.

[26] Y. Park, J.-K. Min, and K. Shim. Efficient processing of skyline queries using mapreduce. *IEEE transactions on knowledge and data engineering*, 29(5):1031–1044, 2017.

[27] P. Siegl, R. Buchty, and M. Berekovic. Data-centric computing frontiers: A survey on processing-in-memory. In *Proceedings of the Second International Symposium on Memory Systems*, pages 295–308. ACM, 2016.

[28] D. Skoutas, D. Sacharidis, A. Simitsis, and T. Sellis. Serving the sky: Discovering and selecting semantic web services through dynamic skyline queries. In *Proc. ICSC*, pages 222–229. IEEE, 2008.

[29] L. Song, X. Qian, H. Li, and Y. Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 541–552. IEEE, 2017.

[30] E. Upchurch, T. Sterling, and J. Brockman. Analysis and modeling of advanced pim architecture design tradeoffs. In *Proc. SC*, page 12. IEEE Computer Society, 2004.

[31] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 227–238. ACM, 2008.

[32] S. Wang, Q. Sun, H. Zou, and F. Yang. Particle swarm optimization with skyline operator for fast cloud-based web service composition. *Mobile Networks and Applications*, 18(1):116–121, 2013.

[33] L. Woods, G. Alonso, and J. Teubner. Parallel computation of skyline queries. In *Proc. FCCM*, pages 1–8. IEEE, 2013.

[34] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. Top-pim: throughput-oriented programmable processing in memory. In *Proc. HPDC*, pages 85–98. ACM, 2014.