Work-Efficient Parallel Skyline Computation for the GPU

Kenneth S. Bøgh, Sean Chester, Ira Assent
Data-Intensive Systems Group, Aarhus University
{ksb,schester,ira}@cs.au.dk

ABSTRACT

The skyline operator returns records in a dataset that provide optimal trade-offs of multiple dimensions. State-of-the-art skyline computation involves complex tree traversals, data-ordering, and conditional branching to minimize the number of point-to-point comparisons. Meanwhile, GPGPU computing offers the potential for parallelizing skyline computation across thousands of cores. However, attempts to port skyline algorithms to the GPU have prioritized throughput and failed to outperform sequential algorithms.

In this paper, we introduce a new skyline algorithm, designed for the GPU, that uses a global, static partitioning scheme. With the partitioning, we can permit *controlled branching* to exploit transitive relationships and avoid most point-to-point comparisons. The result is a non-traditional GPU algorithm, `SkyAlign`, that prioritizes work-efficiency and respectable throughput, rather than maximal throughput, to achieve orders of magnitude faster performance.

1. INTRODUCTION

The skyline [3] is a well-studied operator for selecting the most competitive points from a multi-dimensional dataset. Consider the canonical example of selecting from amongst a set of hotels such as those in Table 1. Here, hotel C is clearly worse than A, because it is *both* more expensive *and* lower rated. The *skyline* is the subset of data points that are not clearly worse than any others, in this case {A, B}.

The skyline is expensive to compute. So, like several other database operators (c.f., [7, 8, 11]), it could benefit from co-processor acceleration. The GPU, in particular, offers high throughput from extreme parallelism, running tens of thousands of threads on thousands of cores to hide memory and instruction latencies. Indeed, GPU skyline algorithms already exist [2, 5] and often approach the device's maximum theoretical compute throughput.

However, this throughput comes at a cost: in comparison to state-of-the-art sequential algorithms [12, 20], the most efficient of these GPU algorithms, `GGS` [2], does up to 650× more work (see Section 6). Even with 2688 cores, the parallelism in our modern GPU is insufficient to overcome this volume of work; for benchmark datasets, computing skylines sequentially is up to 3× faster than using the GPU. That is to say, it is better not to use the GPU at all than to use current GPU skyline algorithms.

Hotel	Price	Rating
A	\$45/nt	***
B	\$75/nt	****
C	\$50/nt	**

Table 1: Sample hotel dataset. A and B are both in the skyline, but C is not because it is dominated by (i.e., has less desirable values for every attribute than) A.

Improving the algorithms, however, is non-trivial, because they are compute-bound. Therefore, in order to outperform sequential computation, GPU skyline algorithms *must do less work*. Thus is our challenge, to achieve greater work-efficiency on an architecture that thrives on throughput.

Even for multi-core [4], designing work-efficient, parallel skyline algorithms is non-trivial. State-of-the-art sequential skyline algorithms [12,20] derive performance from extensive use of trees, recursion, strict ordering of computation, and unpredictable branching. Many of these techniques are not conducive to parallel performance, in general. On the GPU, where threads are executed in groups (called *warps*) such that blocks of warps execute in arbitrary order and threads within a warp always execute the same instruction (i.e., are step-locked), recursive data structures, ordered computation, and branch divergence are debillitating.

So, we introduce a new approach to skylines wherein we globally, statically partition the data into a grid defined by quartiles in the dataset. We recognize that the efficiency of the recursive partitioning algorithms [12, 20] does not come primarily, as thought, from their point-based partitioning, but rather from the memoization of pairwise relationships whenever two points are compared. (Section 4.1 reviews this.) The memoization can still be done with our static grid. Moreover, we can assign homogeneous work to step-locked threads by allocating it in alignment with the static grid cells. The result is hundreds-fold less work than `GGS` [2].

The compromise is that, even with our static grid scheme, skipping point-to-point comparisons still incurs branch divergence. Our non-traditional trade-off, then, is throughput: we do not maximize throughput, but we are work-efficient—running an order of magnitude faster than state-of-the-art multicore [4]—while remaining bound by the availability of

Proceedings of the VLDB Endowment, Vol. 8, No. 9

id	x_0	x_1	x_2	Dominated?	Pruned?
p_1	**2**	2	1		
p_0	1	2	**3**		
p_2	2	**4**	1	✓	
p_3	**3**	3	3	✓	✓

Table 2: Example of skyline definitions and of GPU-friendly prefilter. Points p_2 and p_3 are dominated, so not part of the skyline. The max of each row is bolded. The min of these, 2, is used as a threshold to prefilter points (just p_3 here).

physical compute resources. So, our proposed `SkyAlign` algorithm will scale elegantly with the increase in available parallelism of next-generation GPUs.

Outline We introduce the skyline operator formally, along with background on GPU computing in Section 2. In Section 3 we detail key algorithmic developments in skyline literature. We describe a global, static partitioning scheme in Section 4. Our proposed GPU algorithm, `SkyAlign` is presented in Section 5 and evaluated against state-of-the-art sequential, multicore, and GPU algorithms in Section 6. Finally, we conclude in Section 7.

2. BACKGROUND

In this section, we introduce the notations and assumptions that formalize the work in this paper.

To begin, let P be a dataset consisting of $n = |P|$ points in d dimensions. Arbitrary points in P are denoted by p_i, p_j, or p_k. The i^{th} point in P, under the current ordering of P, is denoted $P[i]$. The value of p_i (or $P[i]$) in the δ^{th} dimension is denoted $p_i[\delta]$ (or $P[i][\delta]$). For example, in Table 2, $n = 4$, $d = 3$, and $p_1[2] = P[0][2] = 1$.

Next, we describe skyline-related concepts (Section 2.1) and then key characteristics of GPUs (Section 2.2).

2.1 Skyline Computation

The skyline is defined through the concept of *dominance*. A point p_i dominates another point p_j if the points are distinct and p_j does not have a smaller value[1] than p_i:

Definition 1 (Dominance [3])**.**
Point p_i *dominates* point p_j, denoted $p_i \prec p_j$ iff:

$$(\exists \delta \in [0,d), p_i[\delta] \neq p_j[\delta]) \wedge \left(\nexists \delta' \in [0,d), p_j[\delta'] < p_i[\delta']\right).$$

We denote the right half of the expression by $p_i \prec_{\text{distinct}} p_j$. This is useful when the distinctness of p_i and p_j can be safely assumed, for it is twice cheaper to evaluate than all of Definition 1. If neither $p_i \prec p_j$ nor $p_j \prec p_i$, we say that p_i and p_j are *incomparable*, denoted $p_i \nprec p_j$. Note that dominance is transitive (i.e., $p_i \prec p_j \wedge p_j \prec p_k \implies p_i \prec p_k$), but that incomparability is not.

Given an input dataset, P, the skyline is the subset of P that is not dominated:

Definition 2 (Skyline [3])**.**
The *skyline* of P, denoted $\text{SKY}(P)$, is the set:

$$\{p_i \in P : \nexists p_j \in P, p_j \prec p_i\}.$$

Considering Table 2 again, both p_2 and p_3 are dominated by p_1, because none of the points are equivalent and neither p_2 nor p_3 has a smaller value on any dimension than does p_1. Point p_0, on the other hand, is not dominated by p_1, because $p_0[0] < p_1[0]$. Since $p_1 \nprec p_0$ transitively implies that neither p_2 nor p_3 dominate p_0 either, the skyline is $\{p_0, p_1\}$.

Measuring skyline "work" Determining whether $p_i \prec p_j$ requires evaluating Definition 1, often called a *dominance test* (DT). Although DTs are cheap, each requiring $6d+4$ instructions,[2] they are more expensive than the surrounding computation, which consists mostly of control flow. Furthermore, each DT loads $2d$ floats into registers, a transfer cost that is poorly amortized by the $6d+4$ instructions and susceptible to memory-boundedness for parallel skylines.

Consequently, the performance of skyline algorithms is often measured in terms of the number of DTs executed [12, 20]. Minimizing this reduces both the compute and memory workload of an algorithm. Compared to the $n(n-1)$ DTs used in a brute-force, quadratic algorithm, DTs can be avoided with transitivity relative to a common "pivot" point, which can be ascertained with a *mask test* (MT) that loads just two integers and conducts just 3 instructions (more on this in Section 4).

We define the number of these two high-level operations as the *work* done by a skyline algorithm:

Definition 3 (Skyline work)**.** The *work* done by algorithm $\mathcal{A}$ to compute $\text{SKY}(P)$ using D DTs and M MTs is:

$$\text{W}(\mathcal{A}, P) = (3\text{M} + (6d+4)\text{D}).$$

An algorithm requiring a low amount of work is called *work-efficient*. While the exact cost of DTs vs. MTs depends on cache hit ratios, instruction-level parallelism, etc., *work* provides an abstract measure of performance of skyline algorithms. Unlike previous work (e.g. [4, 12]), which only counts DTs, work recognizes that, while MTs are much cheaper that DTs, they are also significantly more frequent.

2.2 GPU Computation

With thousands of physical processing cores, the GPU offers tremendous opportunity for parallelism, especially as a co-processor to simultaneous CPU computation. However, it has important architectural differences from CPUs. Here, we review those most relevant to this paper.

2.2.1 Computational model

The GPU can reach teraflop-level computational throughput from a combination of rapid context switching and thousands of (relatively slow) physical cores. Threads are grouped into *warps* of size 32, and warps are grouped into *thread blocks* (of tuneable size). Warps are rapidly switched out for others when waiting for memory transfers in order to hide latencies. So, throughput depends on launching enough warps that some are always ready for execution.

Within a warp, all threads are *step-locked*; i.e., they all execute the same instruction at the same time (although some can idle instead). *Branch divergence* results when two threads within the same warp evaluate a condition differently, and consequently must execute separate instructions. This serializes computation, because some threads execute one branch while the others idle, after which the

[1] We assume to prefer smaller values to simplify exposition.

[2] Obtained by counting low-level operations in Algorithm 1 of `GGS` [2] (the branch-free dominance test). Branching DTs are ill-suited to GPUs and have unpredictable, variable cost.

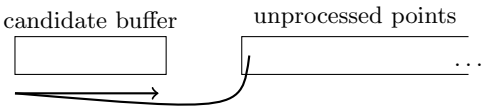


Figure 1: Control flow for sort-based skyline algorithms. As-yet-unprocessed points are iterated sequentially and compared to current solution from left-to-right.

other threads execute the second branch while the first ones idle. The cost of branch divergence can be minimized by ensuring conditions have only one branch (i.e., no `ELSE` statement), but this still idles threads, affecting throughput.

Thread blocks are launched concurrently and the order in which they are executed is controlled at the hardware level. Therefore, there are two means of introducing order into computation: either at the thread-level (the sequence of instructions executed by each thread) or with synchronization points (where all active thread blocks finish before any new ones are launched). Ordered computation within a thread reduces throughput by limiting opportunities for instruction-level parallelism. Synchronization also reduces throughput, because the last thread blocks are unlikely to finish at the same time, leaving physical resources idle.

2.2.2 Memory model

It is important to be aware of the GPU memory model, because control flow that maximizes cache utilization is critical to achieving high memory throughput on the device.

Similar to the CPU, the GPU has a highly stratified memory hierarchy. There is a high latency in copying data from the CPU (host) to the GPU (device) and back. The GPU memory hierarchy consists of *global memory* shared among all resources (6GB on our device), an L2 cache (1.5MB on our device), and three lower-level caches. The read-only texture cache (48KB) has the lowest latency when reading from L2 and is shared among all active thread blocks. 64KB is available for shared memory and the L1 cache, and the proportion devoted to each is configurable. Shared memory is local to a thread block and the L1 cache is local to the set of thread blocks that share a multiprocessor. While L2, and L1 behave like a naive cache, one can deliberately specify which data should be read through texture cache. Shared memory can be used like a heap, with allocations and memory accesses controlled from software.

Finally, each thread can use up to 255 registers, subject to the constraint that at most 65536 registers can be used by the thread blocks sharing the 192 cores of a multiprocessor.

3. RELATED WORK

The skyline operator was introduced by Börzsönyi et al. [3] and has sinced received considerable research attention. Here, we are primarily interested in milestone algorithmic developments in main-memory and parallel skyline computation. We review a key selection of them below.

3.1 Sort-based (and GPU) skyline algorithms

Sort-based skyline algorithms obtain efficiency from monotonicity and transitivity. Figure 1 illustrates the basic control flow, introduced as the *block-nested-loops* (BNL)[3] algorithm [3]. Each unprocessed point p_i is compared with DTs against each point p_j in the current solution. If $p_j \prec p_i$, then p_i is discarded and control passes to the next point. If $p_i \prec p_j$, then p_j is removed from the current solution. BNL can be parallelized (e.g., on FPGAs [19]), but is inefficient in terms of work.

The *sort-first skyline* (SFS) [6] algorithm sorts the points by Manhattan Norm[4] prior to executing BNL. The sort key ensures $P[i+x] \not\prec P[i]$, for any positive x. In other words, once a point is added to the solution, it will never be removed. Furthermore, the sort order loosely correlates with the probability of dominating a random point; so, dominated points are discarded faster. The SaLSa [1] extension changes the sort key to `min` attribute value, which permits halting once the smallest `max` seen in the buffer is less than the `min` at the head of the unprocessed list. Points can also be sorted by z-order [13], another monotonic sort key.

Existing GPU skyline algorithms are adaptations of these ideas for the GPU. The `GNL` [5] algorithm launches a thread for every point. The thread working on behalf of p_i treats the half of the dataset following p_i (wrapping around to the beginning) as the candidate window. For any point p_j determined to be dominated by p_i, the thread increments p_j's global counter (initialized to zero). If $p_j \prec p_i$, then p_i's counter is incremented. Afterwards, any points with non-zero counters are not in the solution. These counters avoid any synchronization; so, `GNL` achieves very high throughput.

The `GGS` algorithm [2], similarly to SFS, first sorts the data by Manhattan Norm. For each iteration, the first α sorted points are declared the candidate buffer. A thread is launched for every point and it compares its point to every point in the buffer. Each iteration is suceeded by a synchronization step in which dominated points are removed, non-dominated points in the α-block are output as skyline points, and the remaining data is re-coalesced. In addition to the advantage of monotonic sorting, this block-wise processing achieves very good spatial locality.

The principal disadvantage of sort-based algorithms is that when the skyline is large, so too is the candidate buffer, causing performance to degrade to brute-force quadratic.

3.2 Partition-based skyline algorithms

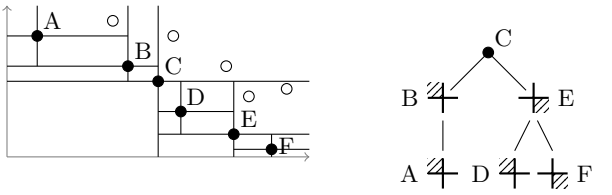
Another class of skyline algorithms partitions the dataset or data space. The first such algorithm was recursive Divide-and-Conquer [3] that halved the dataspace by the median of an arbitrarily chosen dimension and solved each half. The results are merged when backtracking out of the recursion.

A non-recursive version of this is found in many parallel algorithms, which vertically cut the data file, solve each slice on a worker, and then merge the results (e.g., `PSkyline` [10]). In such setups, a key consideration is how the file is cut (e.g., so that points within the same slice are cosine similar [18]), to better balance workload distribution. This approach is common for distributed skyline computation (see the survey by Hose et al. [9]), but does not enable one-word mask tests.

Sequential partition-based algorithms have evolved towards recursive, *point-based* partitioning [12, 20]. For each (recursive) partition, a skyline point, called *the pivot*, is found, and the other points are partitioned based on their relationship to the pivot. A search tree is constructed from the pivots to accelerate the merge phase, which is typically the bottleneck of partition-based approaches. These methods vary on how

[3]We assume "large" memory to simplify the algorithm.

[4]Manhattan Norm is the sum of all attribute values.



(a) Point-based partitions (b) Corresponding tree

Figure 2: Point-based partitioning methods [12, 20].

the pivot is selected, either as a random skyline point [20] or as the one whose attribute values have the smallest normalized range [12], and in the exact layout of the search tree. The `Hybrid` multi-core algorithm [4] is a point-based method that flattens the tree into an array structure for better access patterns and processes points in blocks of size α to improve parallelism. We describe this class in more detail in section 4.1. The proposed `SkyAlign` algorithm is a partition-based method, but it is unlike other algorithms in this class, because it is not recursive and has no merge.

3.3 Other key skyline algorithms

A few algorithms do not fit well into the categorization above. The index-based methods (using B$^+$-Trees [17] or R-Trees [15]) are not especially interesting in our context because GPU support for indexes is weak and they cannot be applied within arbitrary positions of a query plan.

A number of MapReduce algorithms have recently emerged (e.g., [14, 16]). The more recent of these [14] partitions each dimension into m even-width cells to produce a grid with m^d cells. They encode a bitstring of length m^d with 1's for and only for non-empty cells. They then use a method similar to `PSkyline` [10], but exploiting bitstring encodings to avoid regions of points that cannot include skyline points.

4. GPU-FRIENDLY PARTITIONING

The (work-)efficiency of skyline algorithms comes from skipping DTs. The incomparability of two points p_i, p_j can often be ascertained by transitivity if the relationship to a third point, p_k, is known for both p_i and p_j. We call this a *mask test* (MT). The relationship of $p_i[\delta]$ and $p_k[\delta]$ (and also of $p_j[\delta]$ and $p_k[\delta]$) is represented with one bit for each $\delta \in [0, d)$. The incomparability of p_i and p_j can sometimes be detected by comparing the masks: if each has a different bit set, then they each have a dimension on which, by transitivity, they are preferable to each other. A MT is much cheaper than a DT, because it only requires (loading and) comparing 2 values, rather than $2d$ values.

MTs have been shown to drastically improve performance by reducing DTs [4, 12, 20]. In this section, we briefly review the recursive, point-based approach that introduced MTs in literature, including its limitations (Section 4.1). We then describe our GPU-friendly static grid method (Section 4.2).

4.1 The case against recursive partitioning

A review of point-based methods Point-based, recursive partitioning methods induce a quad-tree partitioning of the data set and record skyline points as they are found in a tree. A skyline point (called a *pivot*) is discovered and used to split the partition into 2^d sub-partitions. Each sub-partition is then handled recursively. Figure 2a illustrates the partitioning of space by a set of two-dimensional points and Figure 2b shows the resultant quad-tree of skyline points. Each tree node contains one point p_i and (except for the root) a bitmask that records on which dimensions p_i is worse than its parent. We represent the bitmasks graphically in Figure 2b.

The tree is built incrementally, point by point. When processing the next point, p_i, the quad-tree that has so far been built can be used to eliminate DTs for p_i. First, p_i builds a new bitmask recording its dimension-wise relationship to the root of the tree. If all bits are set, then p_i is dominated. Otherwise, only children of the root for which the bitmask of p_i and the bitmask of the child do not infer incomparability need to be visited. For example, consider when point F is added in Figure 2b. It is first partitioned to the lower-right of root point C. Since all points to the lower right of C are incomparable to all those to the upper left of C, F need not be compared to any point in the subtree rooted at B. The bitmask generated against E similarly permits skipping the subtree rooted at D.

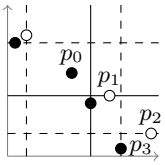
A deeper tree, therefore, permits skipping more DTs. If a point p_i is to compare to a point p_j at depth h in the quad-tree, it uses h cheap MTs to try and infer incomparability before resigning to a DT against p_j. Furthermore, the higher the height of a point p_j for which p_i infers incomparability, the more points p_i can skip entirely.

High divergence The recursively defined, point-based methods are unsurprisingly poorly suited for the branching-sensitive GPU architecture. We discuss challenges for both the traversal of the trees and the partitioning itself.

Traversal We illustrate the challenge with an example. Consider two subtrees, L and R, of a quad-tree and their lowest common ancestor, A. When the points in the subtree rooted at A were partitioned was the last time that points in L and points in R were partitioned using the same boundaries; afterwards, they are sub-partitioned independently based on their own subset of points.

Points are added incrementally to the tree (in depth-first manner). Consider when the points in R (not yet added to the tree) are to compare to the points in L (which are in the tree). First, a DT with the root of L is conducted for each point in R, generating a bitmask. These bitmasks are then used to determine which branches of L each point of R should traverse. Because the root of L is chosen independently of R, the results of all the MTs diverge sporadically. Without some form of global alignment, this will happen for any pair of partitions that are not siblings in the quad-tree.

Partitioning Even just partitioning the points is hard to do effectively. Each partition is sub-partitioned relative to its own pivot, independent of all other partitions. The pivot needs to be a skyline point. Therefore, for each level of the recursion, each subset of points need to independently select a representative skyline point, preferably a "balanced" one. The independent partitions must each do this in a data-parallel fashion to avoid incurring copious branch divergence, while still utilizing the thousands of physical cores on the GPU. We posit that a global partitioning scheme, with a common pivot to all partitions at each level of recursion, can achieve much better utilization and data-parallelism.



Median/Quartile Masks

$M_0 = 01$	$Q_0 = 10$
$M_1 = 11$	$Q_1 = 00$
$M_2 = 10$	$Q_2 = 11$
$M_3 = 10$	$Q_3 = 10$

Figure 3: Static median/quartile-based partitioning of data. Solid lines denote medians and dashed lines denote quartiles.

High dimensions Although the quad-tree partitioning can be effective at reducing DTs, it does not scale well with dimensionality. Consider the effect of adding another dimension to the example in Figure 2. Each bitmask will carry one extra bit of information; so, the probability of a random MT inferring incomparability will rise from $1 - \frac{3}{4}^2$ to $1 - \frac{3}{4}^3$. On the other hand, the branching factor of the quad-tree will double, drastically shortening the tree. So, the number of MTs that, on average, shield any given point from a DT will decrease from $\log_4 n$ to $\log_8 n$. Also, the number of points in any given subtree will decrease; so, the value of skipping subtrees with MTs will decrease. In the ultimate case, a tree with n nodes consists of one root and $n-1$ children, and the average number of masks per point is less than one.

Note that the poor scalability with respect to dimensionality has been noted before [12]; the proposed (although not investigated) solution was to ignore a subset of dimensions—and, ergo, information. We posit that a static grid that assigns a constant number of masks to every point scales elegantly with the information gain in higher dimensionality.

4.2 A static grid alternative

The recursive, point-based methods do not scale with (or otherwise fail to capture the increasing information with) dimensionality. They also lead to heavy, GPU-unfriendly branch divergence. We introduce a static grid that avoids these issues, yet retains—or even expands—the value of MTs.

We first give an overview of the partitioning mechanism (Section 4.2.1). We then describe how to statically generate bitmasks from the partitioning (Section 4.2.2) and how to use the the bitmasks for MTs (Section 4.2.3).

4.2.1 Median/Quartile as a partitioning scheme

Our static grid is conceptually simple: we split each dimension based on the quartiles in the dataset. We define three *global* pivots, one corresponding to each quartile boundary. The middle quartile, the median of the dataset, provides a coarser resolution. For each point p_i, one can define a bitmask relative to the median for a coarse-grain perspective, and one bitmask relative to either the first or third quartile (whichever is relevant), to provide a finer-grain perspective. Then, every point has exactly two bitmasks (and two possible MTs), irrespective of the input parametres. Importantly, all partition boundaries are defined relative to three global pivots, so the boundaries of partitions are *aligned* with each other.

Conceptually, this is similar to the quad-tree decomposition, albeit with virtual pivots. All points are partitioned at the first level by their relationship to the median of the dataset. At the second level of recursion, all points are partitioned by their relationship to the first and third quartiles. This produces a tree with branching factor 2^d, with virtual points in the inner nodes, and sets of points in the leaves. Of course, using consistent splitting points at all second-level vertices is the distinct difference that makes the scheme GPU-friendly.

Indeed, it is worth questioning whether a third level of resolution (octiles) would be worthwhile. Naturally, this depends on both cardinality and dimensionality. Two layers of resolution already provide 4^d distinct partitions. At $d = 10$, this is enough to give a unique partition to one million points (assuming a perfect partitioning). A third level partitioning in 10 dimensions would produce 8^{10}, or slightly more than a billion, distinct partitions. Current GPU memory is well shy of the 40GB that a $10d$ dataset with a point for every octile-level partition would occupy.

4.2.2 Definition of masks

We outlined previously the challenges with assigning the recursively-defined bitmasks to points on the GPU. Here, we describe how it can be done quite easily when using the static grid. Let $\text{quart}_i[\delta]$ denote the i'th quartile for the δ'th dimension; quart_2 is, of course, the median of the dataset. Considering Figure 3 as an example, $\text{quart}_1[1] = p_2[1]$ and $\text{quart}_2[1] = p_1[1]$. The quartiles are virtual: every dimensional value of a quartile corresponds to a value of a data point, but no quart_i is likely to be equal on all dimensions to any single point.

We denote by M_i the median-level-resolution bitmask for point p_i, and we denote by Q_i point p_i's quartile-level-resolution bitmask. For dimension δ, $M_i[\delta]$ is set iff p_i is larger than or equal to the *median* on dimension δ. Q_i is similarly defined. Formally, we have:

$$
\begin{aligned}
M_i[\delta] = 0, Q_i[\delta] = 0 &\iff p_i[\delta] < \text{quart}_1[\delta] \\
M_i[\delta] = 0, Q_i[\delta] = 1 &\iff p_i[\delta] \in [\text{quart}_1[\delta], \text{quart}_2[\delta]) \\
M_i[\delta] = 1, Q_i[\delta] = 0 &\iff p_i[\delta] \in [\text{quart}_2[\delta], \text{quart}_3[\delta]) \\
M_i[\delta] = 1, Q_i[\delta] = 1 &\iff p_i[\delta] \geq \text{quart}_3[\delta]
\end{aligned}
$$

Consider Figure 3 again. The thick lines denote the medians and the thin lines denote the quartiles. The median-level masks are set depending on to which side of the thick lines a point lays and the quartile-level masks are set depending on to which side of the relevant thin line a point lays. Specifically, $M_0 = 01$ because p_0 is less than the x-median and greater than the y-median. By contrast, $M_2 = M_3 = 10$, because p_2 and p_3 are greater than the x-median, but less than the y-median. However, they differ on quartile masks, where $Q_2 = 11$ and $Q_3 = 10$. Finally, $Q_0 = 10$, $M_1 = 11$ and $Q_1 = 00$.

4.2.3 Statically-defined MT-based incomparability

Given the bitmasks defined in the previous subsection, we can define a series of non-dominance implications from their *orders* (i.e., number of bits set) and bitwise relationships. We define these equations at both levels of resolution. The finer, quartile-level resolution assumes knowledge of median-level resolution MTs to elegantly simplify the equations.

Median-level resolution Let p_i, p_j be points with median relationships M_i, M_j. Also, let $|M_i|$ denote the order of M_i. Often, inspecting M_i and M_j is sufficient to reveal that $p_j \not\prec p_i$. We define three rules for this purpose. The rules rely on transitivity with respect to the

median: if $M_i[\delta] < M_j[\delta]$, then $p_i[\delta] < \text{quart}_2[\delta] \leq p_j[\delta]$ and therefore $p_j \not\prec p_i$. Note that the reverse is not true: $p_i[\delta] < p_j[\delta] \not\Longrightarrow M_i[\delta] < M_j[\delta]$, because both $p_i[\delta], p_j[\delta]$ could be less/greater than the median.

$$(M_j \mid M_i) > M_i \implies p_j \not\prec p_i. \quad (1)$$

$$|M_i| < |M_j| \implies p_j \not\prec p_i. \quad (2)$$

$$|M_i| = |M_j|,\ M_i \neq M_j \implies p_j \nprec p_i. \quad (3)$$

Equation 1 The first equation directly expresses this transitivity simultaneously for all bits. It checks whether M_j has *any* bits set that are not also set in M_i. If so, then $\exists\delta$ s.t. $p_i[\delta] < p_j[\delta]$, and, consequently, $p_j \not\prec p_i$. In Figure 3, we can deduce that $p_1 \not\prec p_0$ from the median masks alone. Since $M_0 = 01$ and $M_1 = 11$, the first bit is set in M_1, not in M_0, and $M_1 \mid M_0 = 11 > M_0 = 01$. Therefore, the antecedent of the rule evalutes true, and we have $p_1 \not\prec p_0$.

Equation 2 Equation 2 is a special case of Equation 1. If M_j has more 1's set than does M_i, then it necessarily contains one that is not set in M_i. In such a case, Equation 1 is trivially true. Considering Figure 3 again, we could actually determine $p_1 \not\prec p_0$ from this easier special case: $|M_0| = 1 < |M_1| = 2$.

Equation 3 Finally, Equation 3 identifies another special case, when $|M_i| = |M_j|$, of Equation 1. If M_i, M_j have the same order, then the only condition under which all bits set in M_j are also set in M_i is if the masks are identical. If the masks are not identical, then neither $p_i \prec p_j$ nor $p_j \prec p_i$, because both necessarily have bits set that the other does not. This rule is exemplified between p_2 and p_0 in Figure 3. Both points are partitioned to the same level ($|M_0| = |M_2| = 1$); however, they do not appear in the same partition ($M_0 = 01$, but $M_2 = 10$). Therefore, the points and bitmasks are both incomparable to each other ($M_0 \nsim M_2$ *and* $p_0 \nsim p_2$).

Quartile-level resolution If M_i, M_j are insufficient to determine that $p_j \not\prec p_i$ (i.e., the precedent does not hold in Equations 1-3), the quartile-level masks, Q_i, Q_j, may suffice.

Here, we have two cases: Equation 4 corresponds to a false precedent in Equation 1 and Equation 5, in Equation 3.

$$M_j \preceq M_i, \big(\big((M_j \mid \sim M_i)\ \&\ Q_j\big) \mid Q_i\big) > Q_i \implies p_j \not\prec p_i \quad (4)$$

$$M_i = M_j, \big(Q_j \mid Q_i\big) > Q_i \implies p_j \not\prec p_i \quad (5)$$

Equation 4 Equation 4 bears similarity to Equation 1. The primary difference is that some bits of Q_j are cleared first. This is the incorporation of knowledge from the median-level MT, that $M_j \preceq M_i$. (Note that without the condition on the median-level MT, the equation is incorrect!) The expression $M_j \mid \sim M_i$ yields a result with bit δ set iff $M_i[\delta] = M_j[\delta]$, because $M_j \preceq M_i$ enforces that $M_j[\delta] \leq M_i[\delta]$. In other words, $M_j \mid \sim M_i$ indicates on which dimensions p_i and p_j lay to the same side of the median. These are the dimensions unresolved by the median-level masks (the others, it is safe to assume, indicate $p_j < p_i$). So, the expression $(M_j \mid \sim M_i)\ \&\ Q_j$ selects exactly those bits that were unresolved by median-level masks and for which p_j is greater than the median. Should p_i be less than the median on any of these dimensions, then $p_j \not\prec p_i$.

Algorithm 1 `SkyAlign`: $P \longrightarrow \text{SKY}(P)$

```
1: τ ← min_{p∈P} max_{i∈[0,d)} p[i]
2: P ← {p ∈ P | ∃i ∈ [0,d) : p[i] ≤ τ}
3: for all dimensions δ ∈ [0,d) do
4:    Sort P by dimension δ
5:    quart_i[δ] ← P[⌊i * |P|/4⌋][δ], i ∈ {1,2,3}
6: for all points p_i ∈ P (in parallel) do
7:    for all dimensions δ ∈ [0,d) do
8:       M_i[δ] ← (p_i[δ] > quart_2[δ])
9:       Q_i[δ] ← (p_i[δ] > (M_i[δ]?quart_3[δ] : quart_1[δ]))
10: Sort P by ⟨|M|, M⟩
11: for all levels l ∈ [0,d) do
12:    Record start index of all nonempty partitions
13:    for all points p_i ∈ P (in parallel) do
14:       if |M_i| > l then
15:          for all M : |M| = l ∧ (M | M_i) = M_i do
16:             for all points p_j ∈ P, M_j = M do
17:                if (((M_j | ~M_i) & Q_j) | Q_i) > Q_i then
18:                   if p_j ≺_distinct p_i then
19:                      Mark p_i dominated; terminate thread
20:       else
21:          for all points p_j ∈ P, M_j = M_i do
22:             if (Q_j | Q_i) = Q_i then
23:                if p_j ≺ p_i then
24:                   Mark p_i dominated; terminate thread
25:    Remove dominated points from P
26:    SKY(P) ← SKY(P) ∪ {p_i ∈ P : |M_i| = l}
27: Return SKY(P)
```

Equation 4 is illustrated in Figure 3 to determine that C $\not\prec$ B. Here, $M_C \prec M_B$. Specifically, C is better than B on the second dimension and equal on the first dimension (w.r.t. relationship to the median). Thus, we get $M_C \mid \sim M_B = 10$. However, the quartile masks reveal that $10\ \&\ Q_C = 10$, which contains a bit (namely, the first) that is not in Q_B. Hence, the quartile-level masks permit skipping a dominance test that otherwise would have been conducted.

Equation 5 Equation 5 is a special case of Equation 4, corresponding to when the median-level masks are equal (as in Equation 3). Then, $M_j \mid \sim M_i = [1]^d$, the *identity mask*, and so $(M_j \mid \sim M_i)\ \&\ Q_j = Q_j$. The median masks in this case provide no information for the quartile-level test, thus do not appear in the equation.

This last equation is illustrated in Figure 3 with C and D, for which $M_C = M_D = 10$. Since they are in the same median-level partition, nothing is known of their relationship. We use the entire quartile-level masks to determine, since $Q_C \mid Q_D = 11 \mid 10 = 11 > Q_D$, that C $\not\prec$ D.

5. THE SKYALIGN ALGORITHM

Here, we introduce `SkyAlign` (Algorithm 1), a work-efficient GPU skyline algorithm that uses static partitioning (Section 4). The key algorithmic idea in `SkyAlign` is the manner in which we introduce order. Data points are physically sorted by grid cell and threads are mapped onto that sorted layout. The actual computation, however, is loosely ordered with d carefully-placed synchronizations. This use of order simultaneously achieves good spatial locality, homogeneity within warps, and independence among threads.

At a high level, `SkyAlign` consists of d iterations. In the l'th iteration, remaining points are compared, each by its

own thread, to all points with order l, using MTs and DTs as necessary. After each phase, we remove dominated points and move all surviving points with order l into the solution. We also repack the remaining data to improve locality and repack the warps, since some threads may have already determined that their points are dominated.

We detail Algorithm 1 in the following subsections. We describe the initialization (Section 5.1), the data layout and allocation of work to threads and warps (Section 5.2), how Equations 1-5 are utilized to improve work-efficiency (Section 5.3), and the thread-level control flow (Section 5.4).

5.1 GPU-friendly initialization

Our initialization, once the data is resident in GPU memory,[5] consists of a pre-filter, the assignment of masks/grid cells, and sorting of the data points.

Pre-filter The pre-filter (Lines 1–2 of Algorithm 1) eliminates points that are easy to identify as not in the skyline prior to the calculation of quartiles and other sorting operations. A similar idea was used in the `Hybrid` multi-core algorithm [4]. The technique in [4] is to precede the main algorithm by first identifying the β points with the smallest sum of attributes, and then comparing every other point to these β points. For the GPU, however, those β points are difficult to identify without sorting; the technique in `Hybrid` [4] uses priority queues (not available on a GPU).

Instead, `SkyAlign` uses a parallel reduction to identify a threshold, τ, as the min of max values. This threshold has already been shown to be effective at eliminating many comparisons [1]. However, we use the threshold differently here: in [1], τ is used to halt execution; we instead employ it before execution to remove some non-skyline points from the input and minimize the costs of subsequent sorting.

Recall the example in Table 2. Here, we use the parallel reduction to identify a threshold of $\tau = 2$, p_1's largest value and the smallest largest value in the data. Next, each thread is responsible for one point, checking whether it has any values less than the threshold (or has all values equal to the threshold). Here, p_3 can be eliminated because it has no values less than τ. Notice that, although p_2 is also dominated by p_1, it is not caught by the prefilter.

Mask assignment Section 4.2.2 described how masks are assigned to each point, given the quartiles of the dataset for each dimension (Lines 6-9). To compute quartiles, we use the extremely parallel built-in GPU radix sort on each dimension independently (Lines 3-5). This is not so expensive: each sort only considers the n floats for the relevant dimension. In some cases, the (approximate) quartiles may even be known, but `SkyAlign` does not make this assumption.

Data sorting Our final initialization step, Line 10, sorts the data points to improve subsequent access patterns. We first sort the ids of the points by integer representation of their median-level bitmasks. We then sort these bitmasks by their cardinality (i.e., $|\mathrm{M}|$). Finally, we reorganize the data to match this two-level sort, which also matches the control flow described in the next subsections.

5.2 Data Layout and Thread Allocations

[5]either via PCIe transfer from host memory or because the previous GPU operator in the query plan completes

`SkyAlign` uses three elements per data point, the attribute values, median-level masks, and quartile-level masks. Here, we describe how we represent these elements and how threads map onto them.

The data itself is stored in a long one-dimensional array of the form $[p[0][0], p[0][1], \ldots, p[n-1][d-1]]$ with padding to fit cache lines. This format best supports coalesced reads, because, when an infrequent, ad-hoc DT is required between $P[i]$ and $P[j]$, we load the d values offset from $i * d$ and offset from $j * d$, which are on the same or consecutive cache lines. The quartile masks are stored in an array of length n, sorted in the same order as the data points. Therefore, the quartile mask for $P[i]$ is at the i'th position of the quartile mask array.

Because there are fewer median-level masks, we represent them with two arrays. One stores at position i the i'th distinct median-level bitmask that is used. The other array contains the start index in the quartile mask array of the first incidence of the i'th median-level bitmask (i.e., a prefix sum). This permits indexing directly into the quartile mask array when a median-level MT fails.

All three data structures are repacked (i.e., values are shifted left to cover points that have been removed) at each synchronization to maintain contiguousness and alignment.

The threads then are allocated in order. Thread t_i works on point $P[i]$. Because the data points are sorted by median-level partition, threads are similarly sorted. In other words, the threads working within any given warp work on a small set of partitions, so have minimized divergence with respect to median-level MTs (Line 15).

5.3 Work-efficiency

The work-efficiency of `SkyAlign` comes from the static partitioning described in Section 4. The five equations introduced in that section are used to substitute DTs for MTs.

We employ Equation 2 first, on Lines 10-11 and 26. Due to the iteration order, a thread processing point p_i will only consider points p_j such that $|\mathrm{M}_i| \geq |\mathrm{M}_j|$. Once p_i has finished processing all points with order $\leq \mathrm{M}_i$, it can be progressively added to the solution: no point with higher order can possibly dominate p_i.

On Line 14, we branch on the order of the point. Note that at most 1 warp diverges at this line for any given iteration. The points with order $> l$ branch into Lines 15-19, where they conduct both median-level and (possibly) quartile-level MTs. Line 15 branches on fewer than 2^d warps. The median-level MT occurs on Line 16, where a thread processing point p_i only considers other median masks for which Equation 1 does not hold. If this MT fails, we then load and iterate the quartile-level masks and use Equation 4 on Line 18 to ascertain which points will require a DT.

Conversely, if at Line 14 a thread branches towards order $= l$ (note that $< l$ is impossible), then Lines 21-24 are executed. In this case, Equation 3, used on Line 21, permits skipping median-level tests by only comparing a point p_i to other points p_j with $\mathrm{M}_i = \mathrm{M}_j$. The quartile-level MT on Line 22 invokes Equation 5 to decide if a DT is necessary.

5.4 Thread-level control flow

Here, we combine the previous subsections with our running example from Figure 3. Figure 4 illustrates, for the first (and, in this case, only) iteration, the control flow of each thread through the data structures described earlier.

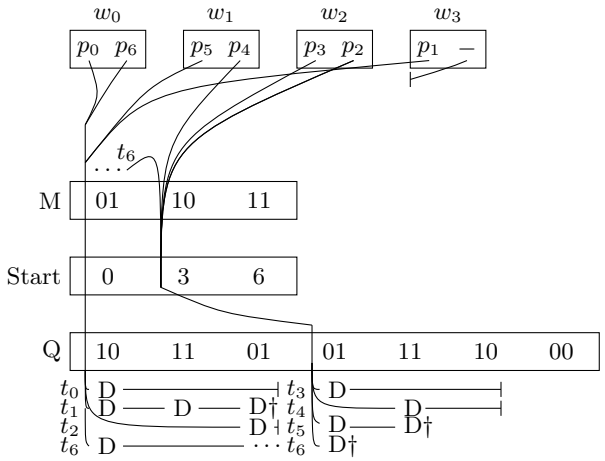


Figure 4: `SkyAlign`'s thread flow for the data in Figure 3

The seven points are sorted by median-level mask: masks with 1 bit set appear first; the mask with 2 bits set appears afterwards. There is no sort with respect to quartile-level mask; however, the quartile-mask array is ordered the same as the list of points. Thread t_i is responsible for point $P[i]$. Threads are grouped into warps (of size 2 for illustrative purposes). For example, thread t_1 from warp w_0 processes point $p[1] = p_6$. The path of a thread through the data structures is traced with a line. All threads run concurrently.[6]

Threads t_0–t_5 process points with order 1, the same as the current iteration. Thus, they index directly into and iterate the three quartile masks for their own point's partition (Lines 21–24). When a quartile MT fails, for example in all cases for t_1, a DT is conducted, denoted by a D under the quartile mask. When a point is dominated, the thread dies (denoted by a dagger). If thread t_i reaches the end of its point's partition without discovering dominance, such as with t_0, t_2, t_3, t_4, point $P[i]$ will be added to the solution.

Thread t_6, on the other hand, works on point p_1 with order 2. So, it iterates all the median-level masks with order 1 (Lines 15–19). For each median-level MT that fails (in this case, both), the thread iterates the corresponding set of quartile masks. When the quartile-level MT fails, such as at position 0 and 3, a *distinct-value DT* is conducted. Were this thread to reach the end of the iteration without being dominated, it would survive to the next iteration. However, p_1 here is dominated on its second DT, against p_4.

Note that only one warp, w_1, diverges on the median-level MT. This is the only warp that contains points of multiple median-level grid cells. In general, because there are at most $2^d - 1$ median-level grid cell boundaries, at most $2^d - 1$ warps can contain threads that diverge at this line. This is in sharp contrast to point-based partitioning, which would branch sporadically. Also note that, because the threads are step-locked, whenever multiple threads need to access data, they are always accessing the same data. Consequently, all share the same cache line.

6. EXPERIMENTAL EVALUATION

[6]Strictly speaking, some warps will run concurrently while others queue, and the order in which they are queued is unpredictable.

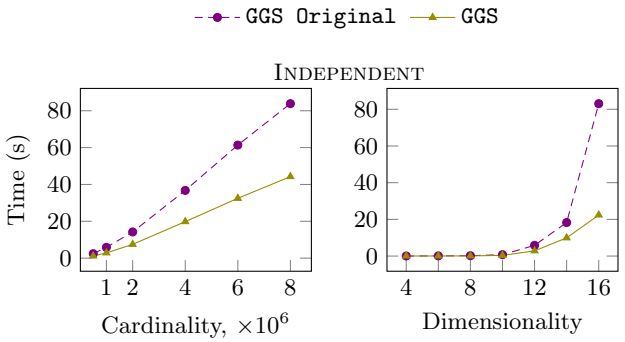


Figure 5: Unoptimized vs. optimized `GGS` implementations.

Here, we conduct thorough experimentation, evaluating `SkyAlign` against state-of-the-art sequential, multi-core, and GPU skyline algorithms and investigating `SkyAlign` in detail. We begin by describing the setup of the experiments (Section 6.1) and then discuss the results (Section 6.2).

6.1 Setup and configuration

Software For CPU algorithms, we use publicly available [4] C++ implementations of sequential `BSkyTree` [12] and multi-core `Hybrid` [4]. These algorithms have both been recently shown to be state-of-the-art by their respective authors for their respective platforms. We use our own implementation of `GGS` [2], the state-of-the-art GPU algorithm; however, we optimize it (described below) to utilize features in our newer graphics card. We implement our proposed algorithm, `SkyAlign` (Section 5), in CUDA 7.

Our implementation of `SkyAlign` and our optimized version of `GGS` will both be released publicly upon publication.[7]

GPU Optimizations We optimize `GGS` by unrolling loops to make better use of registers and to increase instruction-level parallelism (using C++ templates). The original implementation tiled data into shared memory, but we instead load it through lower-latency, read-only texture cache (which was not available on the graphics card used in [2]). Finally, we augment the `GGS` algorithm with distinct-value DTs, since the `BSkyTree`, `Hybrid`, and `SkyAlign` implementations all take advantage of distinctness: In `GGS`, points with different *Manhattan scores* are certainly distinct; comparing them can use the twice-cheaper distinct-value DT. Figure 5 shows the speed-up provided by these optimizations. We make the same optimizations to `SkyAlign`.

Datasets To measure trends, we use synthetic data, generated with the standard skyline dataset generator [3], to produce datasets that are correlated, independent, and anti-correlated. We vary dimensionality $d \in \{4, 6, \ldots, 16\}$ and cardinality $n \in \{\frac{1}{2}, 1, 2, 4, 6, 8\} * 10^6$. Following literature [4, 12], we assume default values of $d = 12$ and $n = 1 * 10^6$.

Environment All experiments are run on a quad core Intel i7-3770 at 3.40GHz, with 16GB of RAM, running Fedora 21. The GPU algorithms use a dedicated Nvidia GTX Titan graphics card. Timings are measured with C++ libraries inside the software and do not include reading input files into CPU memory, but do include transfer times from the CPU

[7]http://cs.au.dk/research/research-areas/data-intensive-systems/repository/

host to the GPU device. The GPU implementations are compiled using the nvcc 7.0.17 compiler; the CPU implementations are compiled using `g++` 4.9.2 with the `-O3` flag. `Hybrid` is run with eight threads (hyper-threading enabled).

Experiments We conduct four experiments, the first two comparing the performance of the four algorithms to each other, and the next two investigating the performance of `SkyAlign` in more depth. We describe each of them below:

Run-time performance We compare the relative performance and trends in execution times of the four algorithms over variations in distribution, dimensionality, and cardinality (Section 6.2.1).

Work-efficiency As with run-time performance, we compare the relative performance and trends of the four algorithms, this time with respect to DTs and work (Definition 3), and compare this to run-time (Section 6.2.2).

`SkyAlign` variants To isolate the effect of `SkyAlign`'s algorithmic contributions, we disable features of `SkyAlign` and observe the resultant performance. Specifically, we examine the effects of synchronization, padded partitions, and quartile-level vs. median-level partitioning (Section 6.2.3).

Per-phase performance `SkyAlign` processes data in d phases, synchronizing after each. We examine the run time of and points pruned by each phase, illustrating which skyline points most impact efficiency and the final result (Section 6.2.4).

6.2 Results and Discussion

Here, we report and discuss the experimental results.

6.2.1 Run-time performance

Figure 6 shows run times in milliseconds on a logarithmic scale for each of the four algorithms on the three distributions (decreasing correlation from top to bottom).

Cardinality The subfigures on the left show trends with respect to increasing cardinality ($d = 12$). We note first that `GGS` is slower than `BSkyTree` on most workloads and only marginally faster on the exceptions (low cardinality, independent). It is always much slower than `Hybrid`. This justifies the need for this research, that state-of-the-art GPU skyline algorithms are simply too slow to warrant use.

By contrast, our proposed `SkyAlign` consistently computes the skyline fastest, the only exceptions being the easier low cardinality, correlated workloads, where all methods except `GGS` terminate within one hundred milliseconds.

Considering the multi-core algorithm, `Hybrid`, we do not see the same margin of improvement relative to `BSkyTree` as reported in [4], but we have fewer cores in our configuration.

All methods exhibit the same basic trend of increasing running times with respect to increases in cardinality, irrespective of distribution. `GGS` has the highest rate of growth. This is an unsurprising result: previous literature (c.f., [3, 12, 20]) has noted this distinction between partition-based and non-partition-based (i.e, sort-based) methods, and `GGS` is the only method here that fits into the latter class. Sort-based methods typically do more work per point, so suffer worst when more points must be processed.

Dimensionality The subfigures on the right show trends with respect to increasing dimensionality ($n = 10^6$). Note that, unlike with cardinality, not all methods have the same

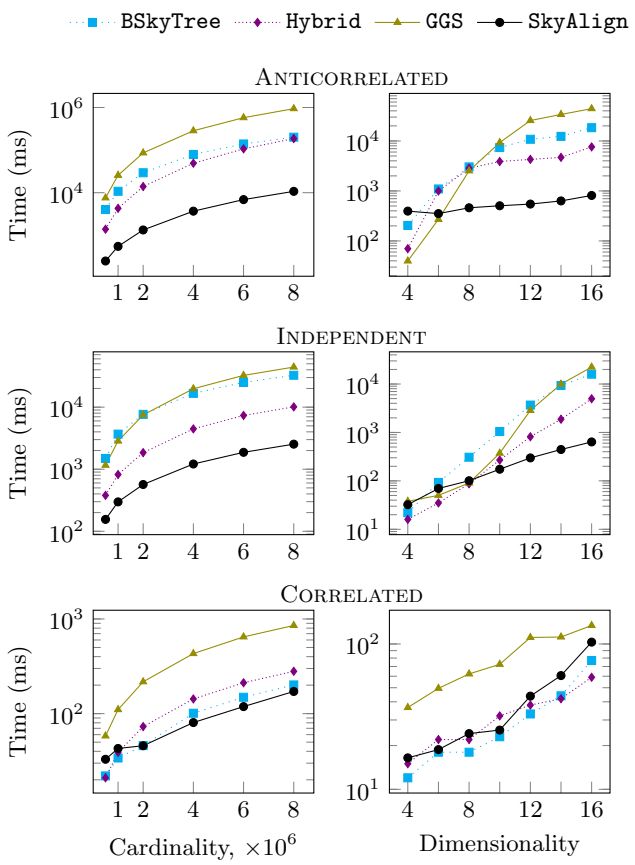


Figure 6: Execution time (ms) of `Hybrid`, `GGS`, `BSkyTree`, and `SkyAlign` as a function of n (left) and d (right).

trends: `SkyAlign` behaves differently on anticorrelated data. In general, `SkyAlign` has a slow growth rate. This is a consequence of the static, rather than recursive, partitioning, as will be elucidated in the experiment on work-efficiency.

Finally, for all distributions, neither GPU algorithm is suitable for very low dimensional (i.e., $d < 6$) data: the computation is not challenging enough to provide the opportunity to amortize the cost of transferring the data to the GPU. With one million correlated points, increases in dimensionality are still insufficient to increase the workload to beyond tens of milliseconds. On the other hand, `SkyAlign` achieves nearly an order of magnitude improvement over the next competitor for high dimensional, anticorrelated data.

6.2.2 Work-efficiency

Dominance tests Figure 7 plots the number of DTs, normalized per point, conducted by `GGS`, `BSkyTree`, `Hybrid`, and `SkyAlign` as a function of n and d for anticorrelated and independent distributions. The slow performance of high-throughput `GGS` observed in Section 6.2.1 is clearly explained by the plots: `GGS` consistently performs significantly more DTs, a difference of nearly *five orders of magnitude* relative to `SkyAlign` in the most extreme case (anticorrelated, $d = 16$). Without any mechanism to avoid DTs, other than Manhattan Norm sort, and a very large skyline that limits the avoidance of DTs through transitivity, `GGS` necessarily degrades to quadratic. The extreme parallelism on the GPU

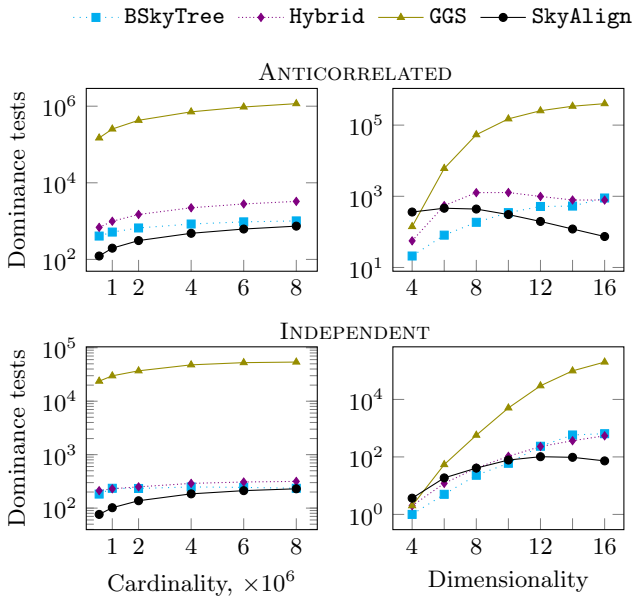


Figure 7: DTs conducted by each algorithm

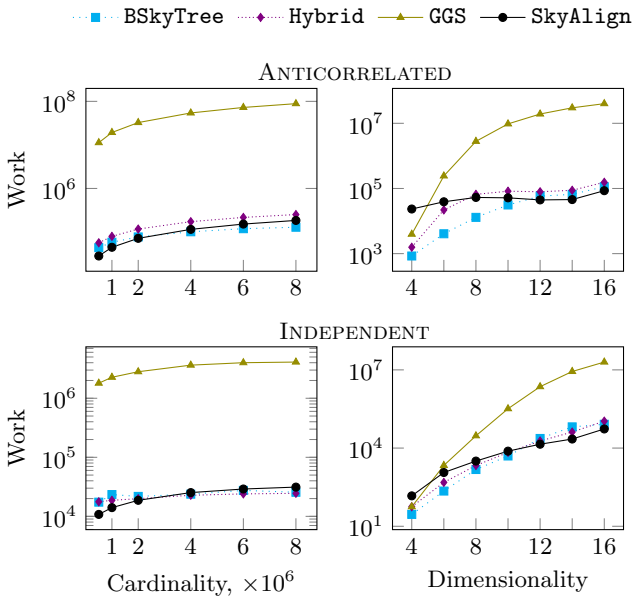


Figure 8: Work-efficiency of each algorithm

counters much, but not enough, of this difference in DTs to make GGS nearly competitive with sequential BSkyTree. However, compared to the other parallel algorithms, which have much better work-efficiency, GGS is uncompetitive.

By contrast, lower-throughput SkyAlign consistently uses within 5% of the fewest DTs for $d \geq 10$, often less work even than the sequential algorithm. This is astounding, since parallel algorithms, like Hybrid can be seen to do here, typically trade off work-efficiency for more parallelism. These plots enforce that, while throughput is crucial for parallel—especially GPU—algorithms, so too is work-efficiency.

The trends with respect to cardinality (on the left) show a convergence among Hybrid, BSkyTree, and SkyAlign. The recursively partitioned algorithms do not require many additional DTs per point when the number of points increases, because the resultant quad tree simply becomes deeper; more meta-data (in terms of MTs) is available for avoiding DTs among the new points. The growth rate of DTs for SkyAlign is slow, since quartile-level partitioning is still sufficient to give nearly every point its own quartile-level cell.

It is worth comparing the cardinality plots for independent data in Figures 6 and 7. The consistency among algorithms in the shape of their trendlines permits really observing the effect of the parallel architectures. Despite such a significant gap in DTs between GGS and BSkyTree, we observe roughly equal performance: the impact of high-throughput GPU computing is really massive. On the other hand, the work-efficient algorithms have roughly the same performance with respect to DTs; we see the expected several factors improvement for multicore Hybrid over BSkyTree, and the order of magnitude improvement, despite the lower throughput, for GPU SkyAlign over BSkyTree.

Work A curious effect is observed when comparing DTs in the dimensionality plots in Figure 7 to execution times in Figure 6. For SkyAlign, the number of DTs per point actually decreases with increases in d; however, the running time continues to climb (albeit slowly). The reason, naturally, is that the standard counting of DTs does not take into account the work done in evaluating MTs. The plots in Figure 8 of work (Definition 3), which also constantly climb at a slow rate, better match the observed performance, so are perhaps a better measure of performance than DTs.

Here we observe that GGS is not as extremely outperformed in work as the DT plots imply, because it does *no* MTs. We also see that the recursively-partitioned methods have very similar work to SkyAlign for $d \geq 10$; while SkyAlign manages to avoid DTs that BSkyTree and Hybrid cannot, it uses many MTs to do so. As is more intuitive, we see that the sequential algorithm, by this definition of work, is approximately as work-efficient as SkyAlign.

In summary, we expect low d to advantage recursive partitioning, where it is nonetheless outperformed on account of parallelism. Conversely, the static grid method becomes relatively more work-efficient with increasing d, and so, with massive parallelism, clearly outperforms the competition.

6.2.3 SkyAlign *variants*

We study a few algorithmic choices in SkyAlign that are perhaps surprising relative to typical GPU algorithms. Synchronization often limits parallelism; so, to evaluate our use of it, we run a version, NoSync, where Line 11 is removed from Algorithm 1. Similarly, MTs create divergence within warps. We study a SkyAlign variant that only uses median-level partitioning (NoQuart) to remove quartile-level MT divergence. We also study a variant, Padded, wherein every median-level partition is padded with data to align the partition boundaries with warp boundaries. Padded thus completely avoids median-level MT divergence. Figure 9 plots the variants. The slower a variant is relative to SkyAlign, the more effective is its disabled feature.

The most striking of the variants is NoSync, which has a profound impact on low (anticorrelated) to moderate (independent) dimensionality. Recall, the value of the synchronization is to improve data access patterns by recompressing

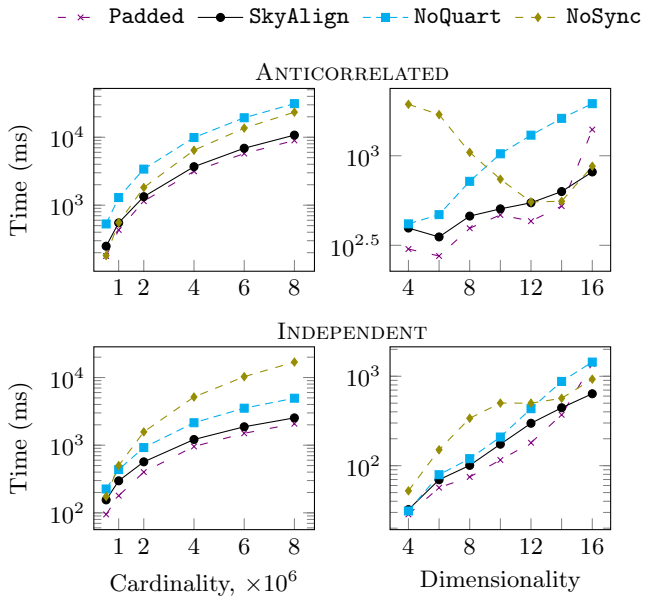


Figure 9: Run-time of `SkyAlign` with features disabled

the data structures to exclude data that has already been eliminated and to improve resource utilization by allocating new work to otherwise retired threads.

The number of synchronizations is the same as the dimensionality, so cost clearly grows with d. Also, as d increases, so, too, does the size of the skyline; so, the number of points being removed between synchronization points decreases. This presents less value in the synchronization. Thus, we see decreasing payoff with increasing dimensionality.

On the independent data, however, the trend is delayed. Note that utilization can only be improved if there is enough work to utilize the resources. In low-dimensional, independent data, much of the data is pruned early. If there are not at least 28672[8] points left, we cannot fill enough threads anyway, and so the value of repacking warps is compromised. It is after 8 dimensions where the skyline becomes large enough that the graphics card can be utilized well enough to really take advantage of the synchronizations.

With respect to increasing cardinality, we always see added value in synchronization. More data points leads to larger skylines, larger working sets, and thus better utilization.

The effect of the quartile-level MTs is notable and consistent. The `NoQuart` variant always performs worse than `SkyAlign`, and moreso with increases in either input parametre, d or n. This ratifies the observations we made in the previous subsection on work-efficiency. MTs are much cheaper than DTs, and `SkyAlign` trades the latter for the former.

Our final variant, `Padded`, is quite interesting. It generally outperforms `SkyAlign` by a small margin by achieving higher throughput from less divergence. However, this does not continue into higher dimensions, where the performance spikes to worse than `SkyAlign`. Note that with more dimensions, the same number of points are scattered over more median-level partitions. Consequently, each partition has fewer points and more padding. Thus, the throughput decreases after a critical point, because the gains from homogeneous work are overcome by the percentage of resources idled by each warp. Many (up to 32×) more warps are ultimately launched, resulting in a slow-down. It is for the consistency and reliability that we prefer `SkyAlign` to `Padded`. Nonetheless, it is interesting to observe the trade-off in two elements of GPU throughput: avoiding branch divergence versus busying all the resources with meaningful work.

[8]Amount of concurrently running threads on our GPU

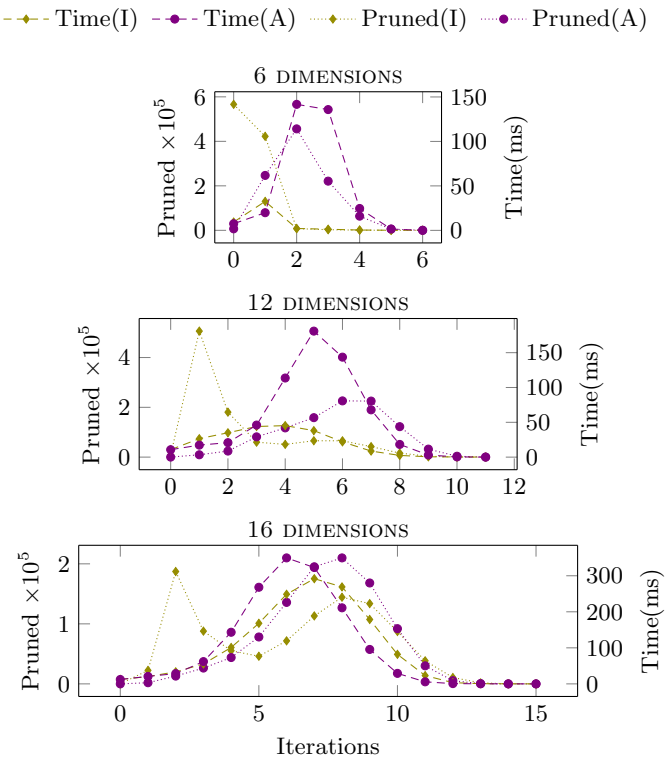


Figure 10: Run-time, pruning of `SkyAlign`, by iteration.

6.2.4 *Per-iteration performance*

Our final plots, Figure 10, show a break-down of `SkyAlign` per each of the $\leq d$ iterations. We also show "iteration 0," which includes transfer to device, the pre-filter, and median/partition calculations. The dotted lines depict, on the left y-axis, the number of points pruned in each iteration; the dashed-lines depict, on the right y-axis, the execution time of each iteration. We show independent and anticorrelated data in the plots. There are three plots, one each for low ($d = 6$), default ($d = 12$), and high ($d = 16$) dimensionality.

The 12d plot illustrates the difference between the distributions quite nicely. For independent data, the number of points pruned spikes quickly in the first couple of iterations. Thereafter follows a spike in execution time over the next few iterations (#3-5) where the majority of remaining points are verified as members of the solution. Conversely, for the anticorrelated data, execution time spikes first, right in the middle where the most (i.e., $\binom{d/2}{d}$) partitions are. The spike in points pruned follows thereafter (iterations #6-8).

This effect explains the success on correlated and independent data of prioritizing points with low Manhattan Norm [2, 6], Z-order [13], and our bitmask cardinality. All these heuristics lead to comparing first against points that are

in our earlier iterations, which quickly reduces input size and improves algorithmic performance. However, these results also suggest exploring alternatives for anticorrelated data, where the majority of point pruning occurs after the majority of the running time. Although the MTs make the processing time faster, alternatives could reduce the input size enough to make for less processing overall.

The independent data is particularly interesting. On low dimensionality, we see that the pre-filter is especially effective. More than half of the points that are eventually pruned are pruned by the pre-filter (#0). The pre-filter is also very fast, always taking $<$ 10ms to compute on any workload. This behaviour is especially dramatic for the correlated data (not shown), where the pre-filter nearly solves the skyline.

On the other end of the scale, the independent data behaves quite unusually in high dimensions. The points pruned have a bimodal distribution, exhibiting characteristics of both independent and anticorrelated data. Still, true to independent data, there is a spike in points pruned in the first couple of iterations, but it is only half the amplitude as on the default dimensionality. Thereafter, as before, follows the spike in execution time. However, this is also followed by a second spike in points pruned, which is nearly as dramatic as the first. The second spike, which follows the majority processes, is reflective of anticorrelated data.

Lastly, recall the performance of `NoSync` in Figure 9. For low d, the synchronization was especially effective. Considering the $6d$ plot in Figure 10, we see that a lot of points, both for anticorrelated and independent data, are pruned in the early iterations. So, it is intuitive that repacking the data and warps to physically remove these pruned points would have a dramatic impact on the access patterns of the several subsequent iterations. For the $12d$ data, on the other hand, the distributions behave quite differently already. The independent data still prunes many points early and thus benefits well from the synchronization; however, the anticorrelated data sees less impact, having not pruned many points until the mid- to late-iterations. Finally, at high dimensions ($d = 16$), both distributions prune a large percentage of their points after the majority of processing time has completed; so, the synchronization is less impactful.

6.3 Summary

To summarize our findings, measuring *work* is a more accurate reflection of running times than is counting DTs. The work of recursively-partitioned methods scales well with n, but not d. The work of our statically-partitioned `SkyAlign` scales very well with d and reasonably well with n. Consequently, `SkyAlign`, being the most parallel of the work-efficient methods, is the most run-time efficient. The state-of-the-art GPU competitor, `GGS`, struggles even to match sequential computation because of its poor work-efficiency.

Looking deeper, we see that the synchronization and branch divergence, which is generally ill-advised for GPU algorithms, pays off for `SkyAlign` because of the resultant work-efficiency. The only alternative that showed promise, that of padding partitions to fit the size of warps, does not scale well with d. Looking more generally at when skyline points are pruned, we distinctly see that independently distributed points are generally pruned by other points that are better than the median across most dimensions, whereas anticorrelated points are generally pruned by other points that are worse than the median on more than half the dimensions. As d increases, the distinction between the distributions blurs, and the independent data exhibits hybrid distribution characteristics.

7. CONCLUSION

In this paper, we investigated skyline computation on the GPU. We showed that existing algorithms, although utilizing the GPU card well, lack the work-efficiency to justify the use of the co-processor. We introduced a new static, global partition-based method, `SkyAlign`, that achieves lower throughput, but that does orders of magnitude less work. This serves as an example of how sophisticated algorithms can outperform high-throughput, but relatively naive, algorithms, even on the massively parallel GPUs. Our static partitioning reduces dominance tests beyond even what is achieved by sequential algorithms for high dimensions. These results suggest that exploring more work-efficient query algorithms on the GPU is a promising direction for database research.

8. ACKNOWLEDGMENTS

This research was supported in part by the Danish Council for Strategic Research, grant 10-092316.

9. REFERENCES

[1] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *TODS*, 33(4):31:1–49, November 2008.
[2] K. S. Bøgh, I. Assent, and M. Magnani. Efficient gpu-based skyline computation. In *Proc. DaMoN*, pages 5:1–6, 2013.
[3] S. Börzsönyi, D. Kossman, and K. Stocker. The skyline operator. In *Proc. ICDE*, pages 421–430, 2001.
[4] S. Chester, D. Šidlauskas, I. Assent, and K. S. Bøgh. Scalable parallelization of skyline computation for multi-core processors. In *Proc. ICDE*, 2015.
[5] W. Choi, L. Liu, and B. Yu. Multi-criteria decision making with skyline computation. In *Proc. IRI*, pages 316–323, 2012.
[6] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proc. ICDE*, pages 717–719, 2003.
[7] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *TODS*, 34(4):1–39, 2009.
[8] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *Proc. SIGMOD*, pages 511–524, 2008.
[9] K. Hose and A. Vlachou. A survey of skyline processing in highly distributed environments. *VLDB J*, 21(3):359–384, 2012.
[10] H. Im, J. Park, and S. Park. Parallel skyline computation on multicore architectures. 36(4):808–823, 2011.
[11] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *Proc. DaMoN*, pages 55–62, 2012.
[12] J. Lee and S.-w. Hwang. Scalable skyline computation using a balanced pivot selection technique. *Information Systems*, 39:1–24, January 2014.
[13] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in Z order. In *Proc. VLDB*, pages 279–290, 2007.
[14] K. Mullesgaard, J. L. Pedersen, H. Lu, and Y. Zhou. Efficient skyline computation in MapReduce. In *Proc. EDBT*, pages 37–48, 2014.
[15] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, March 2005.
[16] Y. Park, J.-K. Min, and K. Shim. Parallel computation of skyline and reverse skyline queries using MapReduce. *PVLDB*, 6(14):2002–2011, 2013.
[17] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proc. VLDB*, pages 301–310, 2001.
[18] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proc. SIGMOD*, pages 227–238, 2008.
[19] L. Woods, G. Alonso, and J. Teubner. Parallel computation of skyline queries. In *Proc. FCCM*, pages 1–8, 2013.
[20] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *Proc. SIGMOD*, pages 483–494, 2009.