Noname manuscript No.
(will be inserted by the editor)

SkyAlign: A Portable, Work-Efficient Skyline Algorithm for Multicore and GPU Architectures

Kenneth S. Bøgh · Sean Chester · Ira Assent

Received: date / Accepted: date

Abstract The skyline operator determines points in a multi-dimensional dataset that offer some optimal trade-off. State-of-the-art CPU skyline algorithms exploit quad-tree partitioning with complex branching to minimize the number of point-to-point comparisons. Branch-phobic GPU skyline algorithms rely on compute throughput rather than partitioning, but fail to match the performance of sequential algorithms.

In this paper, we introduce a new skyline algorithm, `SkyAlign`, that is designed for the GPU, and a GPU-friendly, grid-based tree structure upon which the algorithm relies. The search tree allows us to dramatically reduce the amount of work done by the GPU algorithm by avoiding most point-to-point comparisons at the cost of some compute throughput. This trade-off allows `SkyAlign` to achieve orders of magnitude faster performance than its predecessors. Moreover, a NUMA-oblivious port of `SkyAlign` outperforms native multicore state-of-the-art on challenging workloads by an increasing margin as more cores and sockets are utilised.

Keywords work-efficiency · GPGPU · multicore · parallel algorithms · skyline operator · data structures

K. S. Bøgh
Aarhus University
Åbogade 34, Aarhus N, Denmark 8200
E-mail: ksb@cs.au.dk

S. Chester
Norwegian University of Science and Technology (NTNU)
Sem Sælandsvei 7-9, Trondheim, Norway 7491
E-mail: sean.chester@idi.ntnu.no

I. Assent
Aarhus University
Åbogade 34, Aarhus N, Denmark 8200
E-mail: ira@cs.au.dk

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.

1 Introduction

The skyline [4] is a well-studied operator for selecting the most competitive points from a multi-dimensional dataset. Table 1 gives a canonical example, wherein one is to select from amongst a set of hotels (two-dimensional points). The task can be simplified by first eliminating uncompetitive options. In this example, hotel C is clearly worse than A, because it is *both* more expensive *and* lower rated. The *skyline* is the subset of data points, $\{A, B\}$, that are not clearly worse than any others; it filters the uncompetitive points, such as C.

The skyline is expensive to compute, so, like several other database operators (c.f., [9, 10, 13]), could benefit from co-processor acceleration. The GPU, in particular, offers high compute throughput from extreme parallelism, running tens of thousands of threads on thousands of cores to hide memory latencies. Indeed, GPU skyline algorithms already exist [2, 7] and approach the device's maximum theoretical compute throughput.

However, this throughput comes at a cost: compared to state-of-the-art sequential algorithms [14, 23], the most efficient of these GPU algorithms, `GGS` [2], does up to 650× more work (Section 6). Even modern GPUs offer insufficient parallelism to overcome this volume of work; for benchmark datasets, computing skylines sequentially is up to 3× faster than on the GPU. Frankly,

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 dominance and the GPU-friendly prefilter. Points p_2 and p_3 are dominated by p_1, 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.

it is better not to use the GPU at all than to use current GPU skyline algorithms. Because the algorithms are already high-throughput, they *must do less work* if they are to outperform sequential computation. Thus the challenge: to improve work-efficiency on an architecture that thrives on compute throughput.

Designing work-efficient, parallel skyline algorithms is already non-trivial on more flexible multi-core architectures. Efficient sequential skyline algorithms [14, 23] 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 can be debilitating.

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 recognise that the efficiency of the recursive partitioning algorithms [14, 23] does not come primarily, as thought, from their point-based partitioning, but rather from the memoisation of pairwise relationships whenever two points are compared. (Section 4.1 reviews this.) The memoisation 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 hardware throughput: we maximise neither instruction nor memory throughput, but we are work-efficient, and can therefore run an order of magnitude faster than state-of-the-art multicore while remaining bound by the availability of physical compute resources. So, our proposed algorithm will scale elegantly with the increase in available parallelism of next-generation GPUs.

Compared to the earlier version of this work [3], this paper investigates further impacts of work-efficiency. With the larger cache sizes, out-of-order execution, and superscalar processing on modern CPUs, multicore architectures can exploit parallelism that is difficult to expose to the GPU. We create a NUMA-oblivious port of `SkyAlign` and reclaim much of the presumed-forfeited compute throughput. While a similar port of `GGS` [2] obtains the best instruction throughput on the CPU, `SkyAlign` leverages the *combination* of work-efficiency and throughput to obtain state-of-the-art performance on the CPU, as well. Extensive new experiments show that the GPU-specific considerations account for the algorithm's excellent architectural portability. In all, this paper presents the following concrete contributions:

- a GPU-friendly, grid-based search tree, similar to the quad-tree used by CPU skyline algorithms;
- a novel, work-efficient GPU skyline algorithm that outperforms both multicore and GPU state-of-the-art by at least an order of magnitude; and
- state-of-the-art multicore skyline performance, with better parallel scalability, by porting our work-efficient GPU algorithm to the CPU.

This paper is organised as follows. Section 2 formalises the skyline query and describes key differences between GPU and CPU computing. Section 3 details key algorithmic advances in skyline literature. We describe in Section 4 the global, static partitioning scheme according to which our search tree is constructed and present our proposed GPU algorithm, `SkyAlign`, in Section 5. Section 6 evaluates `SkyAlign` against state-of-the-art sequential, multicore, and GPU algorithms and Section 7 evaluates the multicore performance of ports of GPU skyline algorithms. Section 8 concludes.

2 Background

2.1 Skyline computation

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, with 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 $p_1[2] = P[0][2] = 1$. The skyline is defined via 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 [4]) Point p_i *dominates* point p_j, denoted $p_i \prec p_j$ iff:

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

[1] Without loss of generality and to simplify exposition, we assume smaller values are better, but to handle mixed preferences (e.g., Table 1) is a straight-forward adaptation.

By $p_i \prec_{\text{distinct}} p_j$ we denote the right-hand clause of that expression. This is useful when the distinctness of p_i and p_j can be safely assumed, for it has only half the cost of the full expression. If neither $p_i \prec p_j$ nor $p_j \prec p_i$, we say that p_i and p_j are *incomparable*, denoted $p_i \not\asymp p_j$. 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 incomparability is not.

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

Definition 2 (Skyline [4]) The *skyline* of P, denoted $\text{SKY}(P)$, is the set:

$$\text{SKY}(P) = \{p_i \in P : \nexists p_j \in P, p_j \prec p_i\}.$$

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 $p_1 \not\prec p_0$ 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, called a *dominance test* (DT). Although DTs are cheap, especially when vectorised as in [6], their $6d + 4$ instructions[2] is still more expensive than the surrounding computation, which consists mostly of control flow. More importantly, each DT loads $2d$ floats into registers, a cost that is poorly amortised by the $6d + 4$ instructions. As such, the cost of loading data for DTs can be a constrain compute throughput, especially if the loads are random accesses and the DTs are unpredictable.

As such, the performance of skyline algorithms is often measured in terms of the number of DTs executed [14, 23]. Minimising DTs 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. This 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 versus MTs depends on cache hits, cache line reuse, vectorisation, and other architectural events, *work* provides a reasonable abstract measure of performance of skyline algorithms. Unlike previous work (e.g. [5, 14]), which only counts DTs, our work measure captures that, although MTs are much cheaper that DTs, they are also significantly more frequent in work-efficient algorithms.

2.2 Differences between GPU and CPU architectures

The GPU offers tremendous parallelism; however, it has important architectural differences from the CPU. We briefly review those pertinent to this paper.

Computational throughput on the GPU can reach teraflops by running tens of thousands of threads on thousands of (relatively slow) cores. The 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 loads in order to hide latencies. So, compute throughput depends on launching enough warps that some are always ready for execution. In contrast, the CPU hides latencies by executing instructions when they are ready, not necessarily in order (*out-of-order execution*) and with advanced branch prediction in order to prefetch data.[3]

All threads in a warp are *step-locked*, i.e., they execute the same instruction at the same time (although some can idle instead). *Branch divergence* results when two threads within a warp evaluate a condition differently, and thus must execute separate instructions. This serialises computation: first some threads idle while others execute, and then the first execute while the others idle. The cost of branch divergence can be minimised by ensuring conditions have only one branch (i.e., no `ELSE` statement), but this still idles threads, affecting compute throughput. In contrast, threads on a CPU are fully independent; however, a large number of conditional statements may still negatively impact the branch prediction accuracy. Each misprediction requires a *machine clear*, which costs several cycles and any execution of mispredicted instructions is wasteful.

Thread blocks are launched concurrently and the order in which they are executed is controlled by the hardware. 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 synchronisation points (where all active thread blocks

[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.

[3] Hyper-threading also hides latencies, but it is not as impactful as the features that we will explicitly analyse.

finish before new ones are launched). Ordered computation within a thread reduces compute throughput by limiting opportunities for instruction-level parallelism. Synchronisation also reduces compute throughput, because the last thread blocks are unlikely to finish at the same time, leaving physical resources idle.

Maximising cache utilisation is important for the GPU as well as the CPU to minimise latencies, and they both have highly stratified memory hierarchies. Nonetheless, there are some important distinctions.

The GPU memory hierarchy consists of *global memory* (6GB on our device) and an L2 cache (1.5MB on our device) that is shared among all resources, while clusters of 192 cores each have three local lower-level caches. The read-only texture cache (48KB) has the lowest latency. 64KB is available for shared memory and the L1 cache, and the proportion devoted to each is configurable. A portion of shared memory is allocated to each thread block and the L1 cache is local to the set of thread blocks sharing a cluster of cores. While L2 and L1 behave like naive, inclusive caches, one can choose to read (read-only) data through the faster texture cache. Shared memory can be used as a heap, with allocations and accesses controlled from software.

Our CPUs have a three-level, inclusive cache hierarchy, where L1/L2 are local to each core and L3 is shared by all cores on a socket. Memory is distributed across sockets, which leads to non-uniform latencies for each core (i.e., NUMA effects): memory operations across sockets incur higher latencies. NUMA-oblivious implementations, e.g., those in Section 7, rely on caching and prefetching to circumvent the cost, rather than explicitly managing on which socket memory is allocated.

3 Related work

The skyline operator was introduced by Börzsönyi et al. [4] and has since received considerable research attention. We are primarily interested in milestone developments in main-memory and parallel skyline algorithms and review a key selection of them below.

3.1 Sort-based (and GPU) skyline algorithms

Sort-based skyline algorithms use transitivity and monotonicity to obtain efficiency using a *block-nested-loops* (BNL) algorithm [4] over sorted data.[4] BNL iterates over points p_i, conducting a DT between p_i and all p_j, $j < i$, that is 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 parallelised (e.g., on FPGAs [21]), but it is inefficient in terms of work. The *sort-first skyline* (SFS) [8] algorithm sorts the points by Manhattan Norm[5] 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 [15], another monotonic sort key.

Existing GPU skyline algorithms are adaptations of these ideas for the GPU. The `GNL` [7] 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 (initialised 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 synchronisation; so, `GNL` achieves very high compute 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, comparing its point to every point in the buffer. Each iteration is succeeded by a synchronisation step in which dominated points are removed, non-dominated points in the α-block are output as skyline points, and the remaining data is recoalesced. This block-wise processing (i.e., tiling) augments the monotonic sort with good spatial locality.

The main disadvantage of sort-based algorithms is a large skyline generates a large candidate buffer, causing performance to degrade to brute-force quadratic.

3.2 Partition-based skyline algorithms

Another class of skyline algorithms partitions the data space or dataset. The first of these was recursive Divide-and-Conquer [4] that halves the data space at an arbitrary dimension's median and solves each half. Results are merged when backtracking from 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` [12]). In such setups, a key consideration is

[4] we assume "large" memory to simplify the algorithm.

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

how the file is cut (e.g., so that points within the same slice are cosine similar [20]), to better balance workload distribution. This approach is common for distributed skyline computation (see the survey by Hose et al. [11]), but does not enable one-word mask tests.

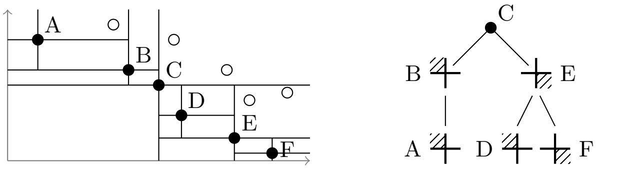
Sequential partition-based algorithms have evolved towards recursive, *point-based* partitioning [14, 23]. For each (recursive) partition, a skyline point, called *the pivot*, is found, and the other points are partitioned based on their relationship to the pivot. A search tree is constructed from the pivots to accelerate the merge phase, which is typically the bottleneck of partition-based approaches. These methods vary on how the pivot is selected, either as a random skyline point [23] or as the one whose attribute values have the smallest normalised range [14], and in the exact layout of the search tree. The `Hybrid` multi-core algorithm [5] 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 these point-based methods 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 split statically and has no merge.

3.3 Other key skyline algorithms

A few algorithms do not fit well into the categorisation above. The index-based methods (using B$^+$-Trees [19] or R-Trees [17]) are not interesting in our context because they cannot be applied at arbitrary positions of a query plan. A number of MapReduce algorithms have recently emerged (e.g., [16, 18]). The more recent of these [16] 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` [12], but exploiting bitstring encodings to avoid regions of points that cannot include skyline points. DTs are accelerated by vectorisation in the `VSkyline` [6] method.

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), since it is done with representative bitmasks. The relationship of $p_i[\delta]$ to $p_k[\delta]$ (and also of $p_j[\delta]$ to $p_k[\delta]$) is represented with one bit for each $\delta \in [0, d)$. The masks are sometimes sufficient to determine that p_i and p_j are incomparable: if a bit is set in the mask of p_i but not the mask of p_j, and vice versa, then we can be certain that p_j and p_i are preferable to each other on those respective dimensions.



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

Fig. 1: Point-based partitioning methods [14, 23].

An MT is cheaper than a DT, as only 2 values are (loaded and) compared, rather than $2d$ values. Substituting MTs for DTs has been shown to drastically improve performance [5, 14, 23]. Section 4.1 briefly reviews the recursive, point-based approach that introduced MTs in literature, with its limitations. Section 4.2 describes our GPU-friendly static grid tree.

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 1a illustrates the partitioning of space by a set of two-dimensional points (skyline points are solid, the rest are dominated) and Figure 1b 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. Figure 1b presents the bitmasks graphically.

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 with bitmasks that dominate or are equal to the bitmask of p_i need to be visited. For example, consider when point F is added in Figure 1b. 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. Since neither C nor E dominated F and the rest of the tree was skipped, F is in the skyline.

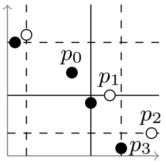
A deeper tree, therefore, permits skipping more DTs. If a point p_i is to compare to a point p_j at depth h in the quad-tree, it uses h cheap MTs to try and infer incomparability before resigning to a DT against p_j. Furthermore, the higher the height of a point p_j for which p_i infers incomparability,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

High divergence Since the point-based methods are recursively defined, they are poorly suited to the branch-sensitive GPU architecture. We discuss challenges for both the tree traversal 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 Just efficiently partitioning the points is hard on the GPU. Each partition is sub-partitioned relative to its own pivot, a skyline point, independent of all other partitions. Therefore, for each recursive call, a subset of points must independently select a “balanced,” representative skyline point. The independent partitions must each do this in a data-parallel fashion to avoid incurring copious branch divergence, while still utilising the thousands of physical cores on the GPU. We will introduce a globally-defined alternative, with pivots common to all partitions at each level of recursion, that is designed to be data-parallel (Section 4.2).

High dimensions Although 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 1. 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 doubles, 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 decreases; so, the value of skipping subtrees with MTs decreases. In the ultimate case, a tree with n nodes consists of one root and $n - 1$ children: the average number of masks per point is less than one.



Median/Quartile Masks

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

Fig. 2: Static median/quartile-based partitioning of data. Solid (dashed) lines denote medians (quartiles). The x-axis (y-axis) is the first (second) dimension.

Note that the poor scalability with respect to dimensionality has been noted before [14]; the proposed (although not investigated) solution was to ignore a subset of dimensions—and, ergo, information. Our static grid assigns a constant number of masks to every point, leading to fewer DTs in higher dimensions (Section 6.2.2).

4.2 A static grid alternative

The recursive, point-based methods do not scale with (i.e., fail to capture the increasing information with) dimensionality and lead to heavy, GPU-unfriendly branch divergence. We introduce a static grid that avoids these issues, yet retains—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. Every point has exactly two bitmasks (and two possible MTs), irrespective of 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/third quartiles. This produces a tree with branching factor 2^d, virtual points in the inner nodes, and sets of points in the leaves. The use of virtual median pivots has been independently validated for recursive partitioning [22]; using non-recursive (i.e., static) splitting points at all second-level vertices is the distinct difference that makes our scheme branch more predictably.

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. We investigate the quality of the partitioning in the experiments of Section 7.3.

4.2.2 Definition of masks

Section 4.1 described the challenges to recursively defining bitmasks on the GPU. Here, we describe how bitmasks can be assigned using the static grid. Let $\mathrm{quart}_i[\delta]$ denote the i'th quartile for the δ'th dimension; quart_2 is, of course, the median of the dataset. Figure 2 shows an example, with the space partitioning on the left (skyline points are again solid; the rest are dominated) and the masks for select points on the right. The quartiles are based on the dataset, e.g., $\mathrm{quart}_1[1] = p_2[1]$ and $\mathrm{quart}_2[1] = p_1[1]$. The quartiles are virtual: every dimensional value of a quartile corresponds to a data point's value, but no quart_i is likely to be equal on all dimensions to any data point.

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

$$\begin{aligned}
\mathrm{M}_i[\delta] = 0, \mathrm{Q}_i[\delta] = 0 &\iff p_i[\delta] < \mathrm{quart}_1[\delta] \\
\mathrm{M}_i[\delta] = 0, \mathrm{Q}_i[\delta] = 1 &\iff p_i[\delta] \in [\mathrm{quart}_1[\delta], \mathrm{quart}_2[\delta]) \\
\mathrm{M}_i[\delta] = 1, \mathrm{Q}_i[\delta] = 0 &\iff p_i[\delta] \in [\mathrm{quart}_2[\delta], \mathrm{quart}_3[\delta]) \\
\mathrm{M}_i[\delta] = 1, \mathrm{Q}_i[\delta] = 1 &\iff p_i[\delta] \geq \mathrm{quart}_3[\delta]
\end{aligned}$$

Consider Figure 2 again. The solid lines denote the medians and the dashed lines denote the quartiles. The median-level masks are set depending on to which side of the solid lines a point lays and the quartile-level masks are set depending on to which side of the relevant dashed line a point lays. Specifically, $\mathrm{M}_0 = 01$ because p_0 is less than the x-median and greater than the y-median. By contrast, $\mathrm{M}_2 = \mathrm{M}_3 = 10$, 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 $\mathrm{Q}_2 = 11$ and $\mathrm{Q}_3 = 10$. Finally, $\mathrm{Q}_0 = 10$, $\mathrm{M}_1 = 11$ and $\mathrm{Q}_1 = 00$.

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 *order* (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 $\mathrm{M}_i, \mathrm{M}_j$. Also, let $|\mathrm{M}_i|$ 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 $\mathrm{M}_i[\delta] < \mathrm{M}_j[\delta]$, then $p_i[\delta] < \mathrm{quart}_2[\delta] \leq p_j[\delta]$ and therefore $p_j \not\prec p_i$. Note that the reverse is not true: $p_i[\delta] < p_j[\delta] \not\Longrightarrow \mathrm{M}_i[\delta] < \mathrm{M}_j[\delta]$, because both $p_i[\delta], p_j[\delta]$ could be less/greater than the median.

$$\begin{aligned}
(\mathrm{M}_j \mid \mathrm{M}_i) > \mathrm{M}_i &\implies p_j \not\prec p_i. && (1)\\
|\mathrm{M}_i| < |\mathrm{M}_j| &\implies p_j \not\prec p_i. && (2)\\
|\mathrm{M}_i| = |\mathrm{M}_j|,\ \mathrm{M}_i \neq \mathrm{M}_j &\implies p_j \not\asymp p_i. && (3)
\end{aligned}$$

Equation 1 The first equation expresses 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 2, we deduce that $p_1 \not\prec p_0$ from the median masks alone. Since $\mathrm{M}_0 = 01$ and $\mathrm{M}_1 = 11$, the first bit is set in M_1, not in M_0, and $\mathrm{M}_1 \mid \mathrm{M}_0 = 11 > \mathrm{M}_0 = 01$. So, the antecedent of the rule is 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 2 again, we could actually determine $p_1 \not\prec p_0$ from this easier special case: $|\mathrm{M}_0| = 1 < |\mathrm{M}_1| = 2$.

Equation 3 Finally, Equation 3 identifies another special case, when $|\mathrm{M}_i| = |\mathrm{M}_j|$, of Equation 1. If $\mathrm{M}_i, \mathrm{M}_j$ 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 2. Both points are partitioned to the same level ($|\mathrm{M}_0| = |\mathrm{M}_2| = 1$); however, they do not appear in the same partition ($\mathrm{M}_0 = 01$, but $\mathrm{M}_2 = 10$). Therefore, the points and bitmasks are both incomparable to each other ($\mathrm{M}_0 \not\prec\!\!\succ \mathrm{M}_2$ *and* $p_0 \not\prec\!\!\succ p_2$).

Quartile-level resolution If $\mathrm{M}_i, \mathrm{M}_j$ do not determine that $p_j \not\prec p_i$ (i.e., the precedent does not hold in Equations 1-3), quartile-level masks, $\mathrm{Q}_i, \mathrm{Q}_j$, may suffice. Here, we have two cases: Equation 4 maps to a false precedent in Equation 1 and Equation 5, in Equation 3.

$$\mathrm{M}_j \preceq \mathrm{M}_i, \Big(\big((\mathrm{M}_j \mid {\sim}\mathrm{M}_i) \mathbin{\&} \mathrm{Q}_j\big) \mid \mathrm{Q}_i\Big) > \mathrm{Q}_i \implies p_j \not\prec p_i \quad (4)$$

$$\mathrm{M}_i = \mathrm{M}_j, \big(\mathrm{Q}_j \mid \mathrm{Q}_i\big) > \mathrm{Q}_i \implies p_j \not\prec p_i \quad (5)$$

Equation 4 Equation 4 resembles Equation 1. The main difference is that some bits of Q_j are cleared first. This incorporates knowledge from the median-level MT, that $\mathrm{M}_j \preceq \mathrm{M}_i$. (Without the condition on the median-level MT, the equation is incorrect.) The expression $\mathrm{M}_j \mid {\sim}\mathrm{M}_i$ yields a result with bit δ set iff $\mathrm{M}_i[\delta] = \mathrm{M}_j[\delta]$, because $\mathrm{M}_j \preceq \mathrm{M}_i$ enforces that $\mathrm{M}_j[\delta] \leq \mathrm{M}_i[\delta]$. In other words, $\mathrm{M}_j \mid {\sim}\mathrm{M}_i$ 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 indicate $p_j < p_i$). So, the expression $(\mathrm{M}_j \mid {\sim}\mathrm{M}_i) \mathbin{\&} \mathrm{Q}_j$ 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 $p_j \not\prec p_i$.

Equation 4 is illustrated in Figure 2 to determine that $p_2 \not\prec p_1$. Here, $\mathrm{M}_2 \prec \mathrm{M}_1$. Specifically, p_2 is better than p_1 on the second dimension and equal on the first dimension (w.r.t. relationship to the median). Thus, we get $\mathrm{M}_2 \mid {\sim}\mathrm{M}_1 = 10$. However, the quartile masks reveal that $10 \mathbin{\&} \mathrm{Q}_2 = 10$, which contains a bit (the first) that is not in Q_1. Hence, the quartile-level masks permit skipping a DT that otherwise was otherwise necessary.

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, $\mathrm{M}_j \mid {\sim}\mathrm{M}_i = [1]^d$, the *identity mask*, and so $(\mathrm{M}_j \mid {\sim}\mathrm{M}_i) \mathbin{\&} \mathrm{Q}_j = \mathrm{Q}_j$. 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 2 with p_2 and p_3, for which $\mathrm{M}_2 = \mathrm{M}_3 = 10$. Since both are in the same median-level partition, their relationship is unknown. We use the full quartile-level masks to determine that $p_2 \not\prec p_3$, since $\mathrm{Q}_2 \mid \mathrm{Q}_3 = 11 \mid 10 = 11 > \mathrm{Q}_3$.

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

```
$\tau \leftarrow \min_{p \in P} \max_{i \in [0,d)} p[i]$
$P \leftarrow \{p \in P | \exists i \in [0,d) : p[i] \leq \tau\}$
for all dimensions $\delta \in [0,d)$ do
    Sort $P$ by dimension $\delta$
    $\mathrm{quart}_i[\delta] \leftarrow P[\lfloor i * |P|/4 \rfloor][\delta], i \in \{1,2,3\}$
for all points $p_i \in P$ (in parallel) do
    for all dimensions $\delta \in [0,d)$ do
        $\mathrm{M}_i[\delta] \leftarrow (p_i[\delta] > \mathrm{quart}_2[\delta])$
        $\mathrm{Q}_i[\delta] \leftarrow (p_i[\delta] > (\mathrm{M}_i[\delta] ? \mathrm{quart}_3[\delta] : \mathrm{quart}_1[\delta]))$
Sort $P$ by $\langle |\mathrm{M}|, \mathrm{M} \rangle$
for all levels $l \in [0,d)$ do
    Record start index of all nonempty partitions
    for all points $p_i \in P$ (in parallel) do
        if $|\mathrm{M}_i| > l$ then
            for all $\mathrm{M} : |\mathrm{M}| = l \wedge (\mathrm{M} \mid \mathrm{M}_i) = \mathrm{M}_i$ do
                for all points $p_j \in P, \mathrm{M}_j = \mathrm{M}$ do
                    if $\Big(\big((\mathrm{M}_j \mid {\sim}\mathrm{M}_i) \mathbin{\&} \mathrm{Q}_j\big) \mid \mathrm{Q}_i\Big) > \mathrm{Q}_i$ then
                        if $p_j \prec_{\text{distinct}} p_i$ then
                            Mark $p_i$ dominated; terminate thread
        else
            for all points $p_j \in P, \mathrm{M}_j = \mathrm{M}_i$ do
                if $\big(\mathrm{Q}_j \mid \mathrm{Q}_i\big) = \mathrm{Q}_i$ then
                    if $p_j \prec p_i$ then
                        Mark $p_i$ dominated; terminate thread
    Remove dominated points from $P$
    $\mathrm{SKY}(P) \leftarrow \mathrm{SKY}(P) \cup \{p_i \in P : |\mathrm{M}_i| = l\}$
Return $\mathrm{SKY}(P)$
```

5 The SkyAlign algorithm

In this section, 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. 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 synchronisation points. 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. At the end of each iteration, we remove dominated points from consideration and add non-dominated points with order l to the solution. We repack non-dominated points with order $> l$ to improve locality and we repack warps, since some threads may have determined their points to be dominated.

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

5.1 GPU-friendly initialization

Our initialisation, once the data is resident in GPU memory,[6] consists of a pre-filter, the assignment of bitmasks (i.e., 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 [5]. The technique in [5] 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` [5] 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 value 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 order (i.e., $|\mathrm{M}|$). Finally, we reorganize the data to match this two-level sort, which also matches the control flow described later.

5.2 Data Layout and Thread Allocations

`SkyAlign` uses three elements per data point, the attribute values, a median-level mask, and a quartile-level mask. 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 after each point to fit cache lines. This format best supports coalesced reads, because this data is only used to conduct ad-hoc DTs, so is accessed randomly: these are the reads `SkyAlign` is designed to avoid. The quartile masks are stored in an unpadded array of length n, sorted in the same order as the data points: the quartile mask for $P[i]$ is at the i'th position of the quartile mask array. The quartile mask array is read sequentially, so these cache lines get high reuse.

As 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 moved left to occupy space vacated by points that have been dominated) at each synchronisation point in order to maintain contiguity and alignment.

The threads 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 in any given warp work on a small set of partitions, so have minimised divergence with respect to median-level MTs (Line 15).

5.3 Work-efficiency

The work-efficiency of `SkyAlign` comes from the static partitioning of 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[7] to the solution: no point with higher order can possibly dominate p_i.

On Line 14, we branch on the order of the point, but at most 1 warp diverges at this line on any iteration. Points with order $> l$ branch into Lines 15-19, where

[6] Either via PCIe transfer from CPU (host) memory or because the previous GPU operator in the query plan completes.

[7] Progressive skyline algorithms [17] can output solution points as they are discovered, in contrast to an algorithm that must fully complete before any solution point can be confirmed.

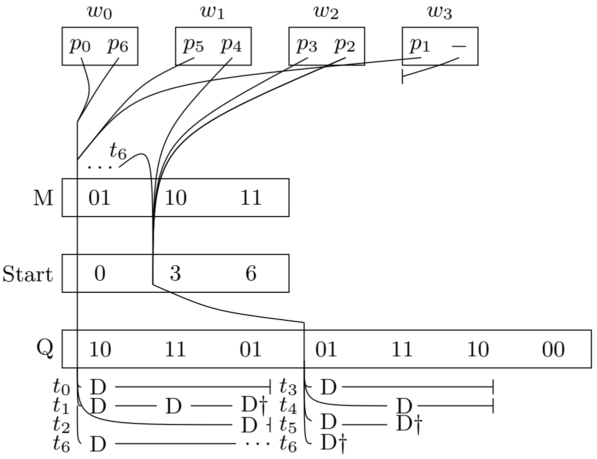


Fig. 3: `SkyAlign`'s thread flow for the data in Figure 2. The three arrays of the quad-tree are represented by M (median masks), Start (start indexes), and Q (quartile masks). The lines trace which values are read by each of the 8 threads. The threads (t_0-t_7) are assigned to warps from left to right. D indicates a DT and †, dominance.

they conduct both median-level and (possibly) quartile-level MTs. Fewer than 2^d warps diverge at Line 15, since there are at most $2^d - 1$ median-level partition boundaries. 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 2. Figure 3 illustrates, for the first (and, in this case, only) iteration, the control flow of each thread through our data structures.

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.[8]

Threads t_0–t_5 process points with order $l = 1$, the value of 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* ($\prec_{\text{distinct}}$) 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 whenever multiple (step-locked) threads need to access data, they always access the same data, therefore sharing the same (read-only) cache line.

6 Experimental evaluation on the GPU

In this section, we evaluate `SkyAlign` relative to state-of-the-art GPU [2], multi-core [5], and sequential [14] algorithms. Section 6.1 describes the setup of the experiments; Section 6.2 presents and discusses the results. Our findings are summarised in Section 6.3.

6.1 Setup and configuration

Software We use publicly available [5] `C++` implementations of sequential `BSkyTree` [14] and multicore `Hybrid` [5] CPU algorithms.[9] These algorithms have both been recently shown to be state-of-the-art by their respective authors for their respective platforms. We

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

[9] `https://github.com/sean-chester/SkyBench`

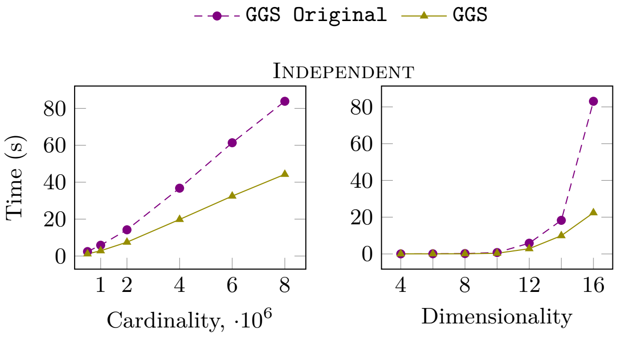


Fig. 4: Unoptimised vs. optimised `GGS` implementations.

use our own implementation of `GGS` [2], the state-of-the-art GPU algorithm; however, we optimise 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 optimised version of `GGS` are unchanged from [3].

GPU Optimizations We optimise `GGS` by unrolling loops (using `C++` templates) to make better use of registers and to increase instruction-level parallelism. 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 ($\prec_{\text{distinct}}$), 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 4 shows the speed-up provided by these optimisations. The same optimisations are present in `SkyAlign`.

Datasets To measure trends, we use synthetic data, generated with the standard skyline dataset generator [4]. The datasets have uniformly-distributed dimensions which are *correlated* (C), *independent* (I), or *anticorrelated* (A). Grids, which `SkyAlign` uses, are often susceptible to data skew; so, we also create *pareto* (P) datasets, wherein each attribute is independent from the others and distributed according to a Pareto distribution with $x_m = 1$ and $\alpha = 1$, by generating uniform random numbers $r \in [0, 1]$ and applying an inverse transform of $x = (1-r)^{-1}$ (the inverse of the cumulative distribution function). This produces a heavy skew towards lower (i.e., better) values. Once generated, we normalise the data to the range $[0, 1]$. For all datasets, we vary dimensionality $d \in \{4, 6, \ldots, 16\}$ and cardinality $n \in \{\frac{1}{2}, 1, 2, 4, 6, 8\}\cdot 10^6$. Following literature [5, 14], we assume default values of $d = 12$ and $n = 1\cdot 10^6$.

Environment All experiments in this section are run on a quad core Intel i7-3770 at 3.40GHz, with 16GB of RAM, running Fedora 21. To differentiate this machine from those in Section 7, we call it `Desktop`. `Desktop` favours the sequential algorithm, `BSkyTree`, given the high clock frequency. Section 7 shows the performance of `Hybrid` with more, but slower, cores. 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 host to the GPU device. The GPU implementations are compiled using the `nvcc 7.0.17` compiler; CPU implementations are compiled using `g++ 4.9.2` with the `-O3` flag. `Hybrid` is run with eight (hyper-threaded) threads.

6.2 Results and discussion

We conduct four experiments in this section, the first two comparing the performance of the four algorithms to each other and the next two investigating `SkyAlign` in more depth. Section 6.2.1 evaluates the relative run time performance and scalability of the algorithms, and Section 6.2.2 contrasts the algorithms in terms of DTs and work conducted. Section 6.2.3 disables features of `SkyAlign` in order to isolate and independently study the effect of each of our algorithmic contributions. Finally, Section 6.2.4 examines `SkyAlign` by iteration, illustrating which iterations most impact execution time and which iterations prune the most skyline points.

6.2.1 Run-time performance

Figure 5 shows run times in milliseconds on a logarithmic scale for each algorithm on the four distributions (increasing 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 and pareto). It is always much slower than `Hybrid`. This justifies the need for this research: state-of-the-art GPU skyline algorithms are simply too slow. By contrast, our proposed `SkyAlign` consistently computes the skyline fastest, the only exceptions being the easier low cardinality, correlated data, where only `GGS` fails to terminate within 100 ms.

We observe a smaller improvement margin for `Hybrid` relative to `BSkyTree` than reported in [5], but `Desktop` has fewer and faster cores than the machine used in [5].

All methods exhibit the same basic trend of increasing running times with respect to increases in cardinality, irrespective of distribution. `GGS` has the highest

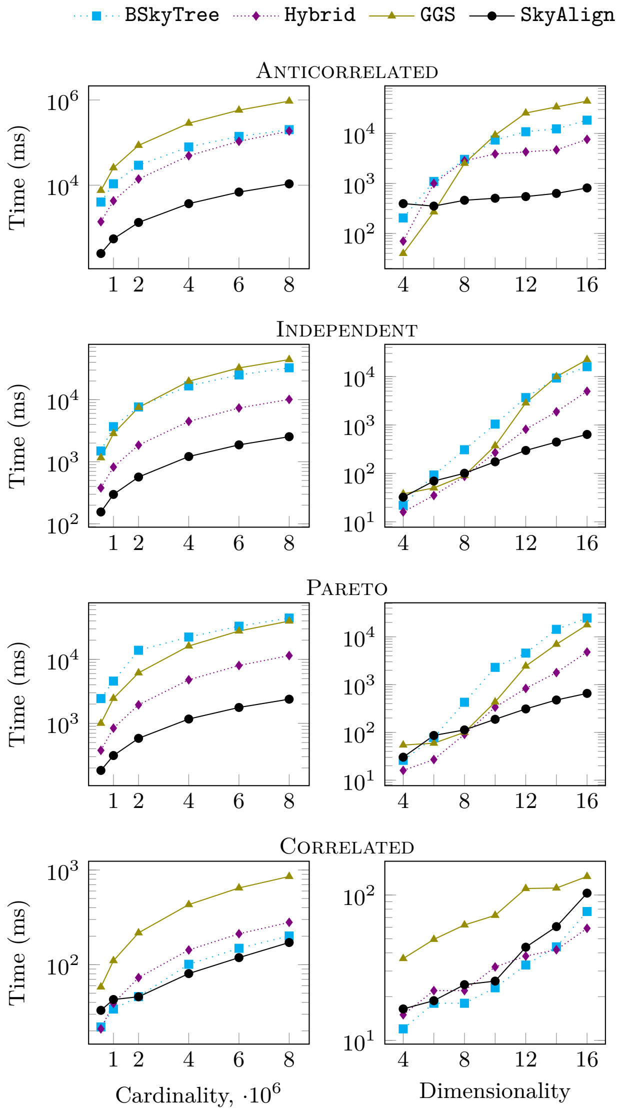


Fig. 5: Execution time (ms) of Hybrid, GGS, BSkyTree, and SkyAlign as a function of n (left) and d (right).

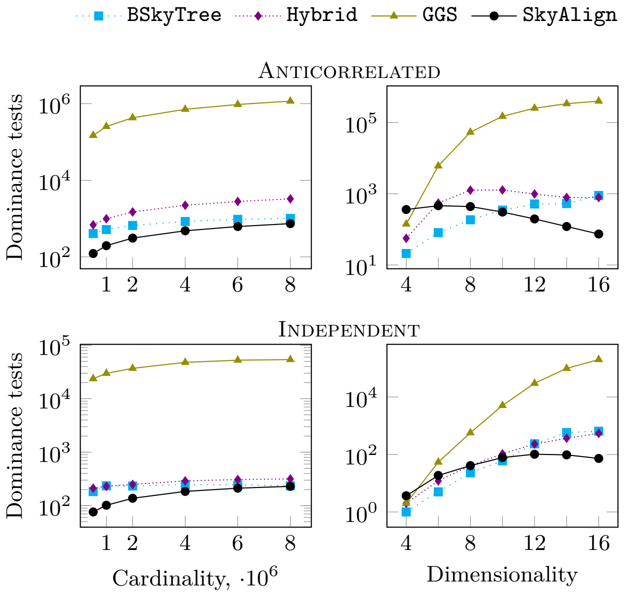


Fig. 6: DTs conducted by each algorithm

rate of growth. This is an unsurprising result: previous literature (c.f., [4,14,23]) 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 worse when more points must be processed. The skew of the pareto distribution, relative to the independent data, has a minor effect that we attribute to the randomness with which the data was generated. The fixed-cost prefilter used by Hybrid (and studied more in Section 7.3) removes 60 % of the independent data points, but only 58 % of those in the pareto distribution, leaving the algorithm with more work to complete. The GPU algorithms use a different filter which provides a slight relative improvement on the pareto data. Evidently, basing the SkyAlign grid on the quartiles of the dataset is sufficient to overcome the data skew.

Dimensionality The subfigures on the right show trends with respect to increasing dimensionality ($n = 10^6$). Unlike with cardinality, not all methods have the same trends: SkyAlign behaves differently on anticorrelated data and generally has a slow growth rate. This results from our static, rather than recursive, partitioning. We elaborate on this in the next experiment.

Comparing results on the independent and pareto distributions again, we see an inconsistent effect of data skew for datasets of different dimensionalities. This supports the conclusion that the variation results from randomness in the data. 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 amortise 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. As we show in Section 7.2, the efficacy of our pre-filters are predominantly responsible for the fast run times; so, we exclude correlated data from the more detailed analysis in the rest of this section. In contrast, 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 6 plots the number of DTs, normalised per point, conducted by each algorithm as a function of n and d for anticorrelated and independent distributions. We omit correlated and pareto distributed data from the remainder of this section, since the former is handled very fast by all algorithms and the latter exhibit trends that are very similar to those of the independent data. 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 degrades to quadratic performance. The extreme parallelism on the GPU 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.

In contrast, lower-throughput SkyAlign uses ≤ 5 % of the fewest DTs for $d \geq 10$, often fewer even than the sequential algorithm. This is astounding, since parallel algorithms, as Hybrid does here, usually trade work-efficiency for more parallelism. These plots enforce that, while compute throughput is crucial for parallel (and especially GPU) algorithms, so too is work-efficiency.

The trends with respect to cardinality (on the left) show a convergence of SkyAlign towards Hybrid and BSkyTree. The recursively partitioned algorithms do not require many additional DTs per point when the number of points increases, because the resultant quadtree 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 5 and 6. The consistency among algorithms in the shape of their trend lines 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 compute throughput, for GPU SkyAlign over BSkyTree.

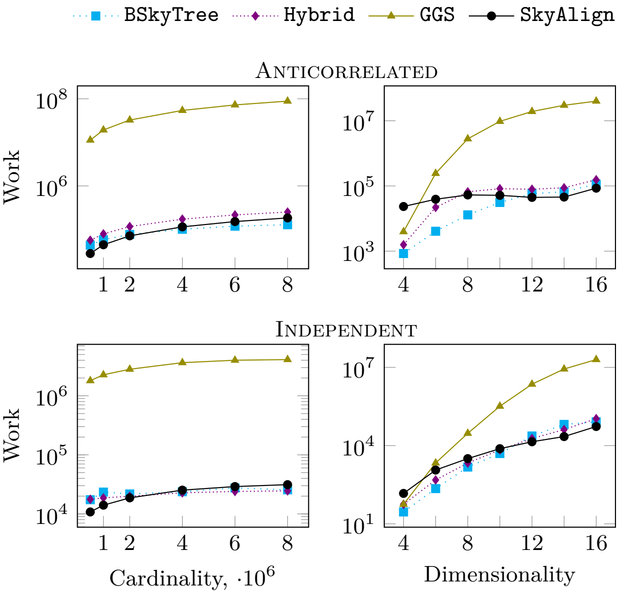


Fig. 7: Work-efficiency of each algorithm

Work A curious discrepancy is observable between the DTs in the dimensionality plots in Figure 6 and the execution times in Figure 5. For SkyAlign, the number of DTs per point *decreases* with increasing d; yet, the running time continues to climb (slowly). The reason is that the standard counting of DTs ignores the work done in evaluating MTs. The plots in Figure 7 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 work-inefficient, and hence slow, as the DT plots imply, because it does *no* MTs. Also the recursively-partitioned methods have very similar work to SkyAlign for $d \geq 10$; SkyAlign manages to avoid DTs that BSkyTree and Hybrid cannot, but it does so with many MTs. 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, our static grid partitioning gains information and 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. Synchronisation often limits parallelism; so, to evaluate our use of it, we run a version, NoSync, where

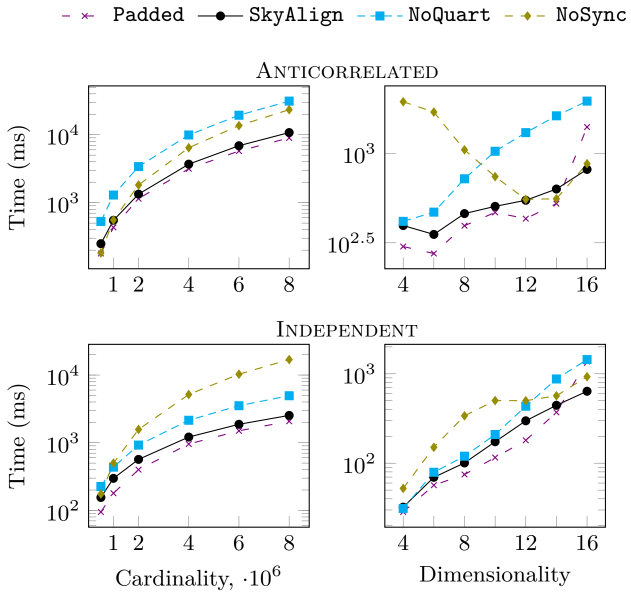


Fig. 8: Run-time of `SkyAlign` with features disabled

Line 11 is removed from Algorithm 1. Similarly, MTs create divergence within warps. We study a 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 8 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 synchronisation is to improve data access patterns by recompressing the data structures to exclude data that has already been eliminated and to improve resource utilisation by allocating new work to otherwise retired threads. The number of synchronisations 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 synchronisation points decreases. This presents less value in the synchronisation. Thus, we see decreasing payoff with increasing dimensionality.

On the independent data, however, the trend is delayed. Note that utilisation can only be improved if there is enough work to utilise the resources. In low-dimensional, independent data, much of the data is pruned early. If there are not at least 28 672 points left,[10] we cannot fill enough threads anyway, and so the value of repacking warps is nullified. It is after 8 dimensions where the skyline becomes large enough that the graphics card can be utilised well enough to really take advantage of the synchronisations.

With respect to increasing cardinality, we always see added value in synchronisation. More points lead to larger skylines and working sets, so better utilisation.

The effect of the quartile-level MTs is notable and consistent. The `NoQuart` variant always performs worse than `SkyAlign`, and more so with increases in either input parameter, d or n. This ratifies our observations 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 compute throughput from less divergence. However, the performance degrades at higher dimensions. 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 compute 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. We prefer the consistency and reliability of `SkyAlign`. 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.

6.2.4 Per-iteration performance

Figure 9 shows a break-down of `SkyAlign` per each of the $\leq d$ iterations. It also shows "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).

[10] This is the number of concurrent threads on our GPU.

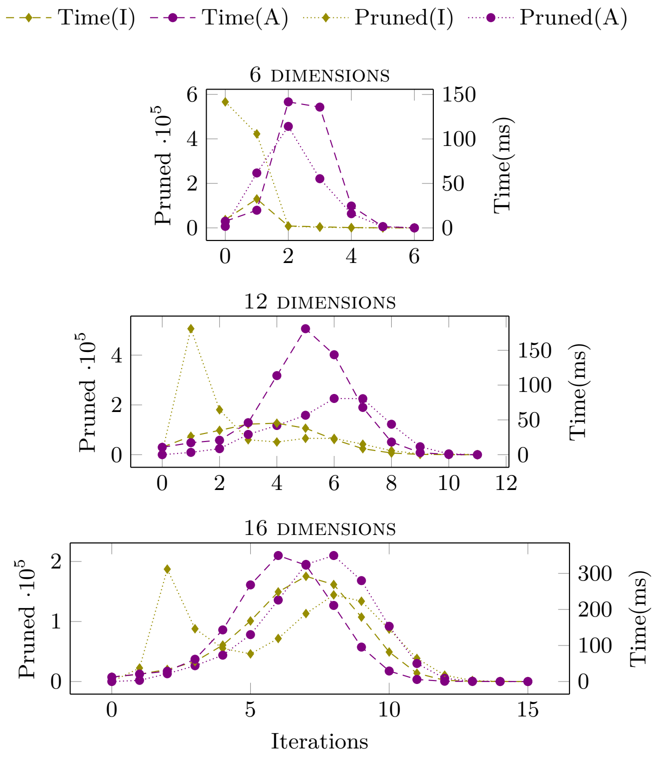


Fig. 9: Run-time, pruning of `SkyAlign`, by iteration.

This effect explains the success on correlated and independent data of prioritizing points with low Manhattan Norm [2,8], Z-order [15], 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 < 10 ms 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 8. For low d, the synchronisation was especially effective. Considering the $6d$ plot in Figure 9, 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 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 synchronisation; 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 synchronisation is less impactful.

6.3 Summary

To summarise 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 grid-partitioned `SkyAlign` scales very well with d and reasonably well with n, even in the presence of data skew. Consequently, `SkyAlign`, being the most parallel of the work-efficient methods, is the most run-time efficient. Given its poor work-efficiency, the state-of-the-art GPU competitor, `GGS`, struggles even to match sequential computation.

Looking closer, synchronisation and branch divergence, generally ill-advised for GPU algorithms, pay off for `SkyAlign` because of the resultant work-efficiency. The only promising alternative, that of padding partitions to fit the warp size, scales poorly with d. Looking abstractly at when points are pruned, we see that independently distributed points are generally pruned by others that are better than the median across most dimensions, whereas anticorrelated points are generally pruned by others 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 characteristics.

7 Portability of the GPU algorithms

We have stated that *work-efficient* parallel algorithms should not do much more work than a single-threaded algorithm. Figure 7 shows that this is true of SkyAlign and Hybrid. Although our intention has been to investigate whether work-efficiency helps on a throughput-hungry architecture such as the GPU, it is reasonable to ask whether these observations hold when the GPU algorithms are ported to multicore CPUs.

We argued in Section 4.1 that the existing CPU algorithms are ill-suited for the GPU. In particular, the state-of-the-art multicore algorithm, Hybrid, dynamically constructs a quad tree that consists only of skyline points. The tree is constructed on-the-fly, incrementally, and sequentially: frequent synchronisation points are necessary to accommodate the sequential insert phase, which would cripple performance on the GPU, and pre-building the tree, as in SkyAlign, requires first knowing the query result. Moreover, the uncontrolled branching in the tree traversal makes it impossible to assign points to warps in a way that would not serialise execution within each warp on account of branch divergence.

On the other hand, because the CPU has a less restrictive threading model than the GPU, the GPU algorithms can easily be ported. This section attempts to answer three related questions: *whether, why, and when* the GPU algorithms port well.

Section 7.2 repeats the experiments of Section 6.2.1, this time comparing CPU implementations of the algorithms on a 28-core machine and also studying parallel scalability. We investigate the reasons for the performance trends in Section 7.3, which looks closer at the CPU-based prefilter and the quartile-based partitioning scheme, and in Section 7.4, which studies low-level metrics of compute throughput, branching behaviour, and cache performance. Finally, Section 7.5 looks closer at NUMA behaviour by repeating some key experiments on a quad-socket machine.

7.1 CPU setup and configuration

This section uses a newer generation dual-socket and a previous-generation quad-socket Intel machine, called Dual and Quad, respectively. Dual has 188 GB of 2 GHz DDR4 memory and 2 Intel Xeon E5-2683 v3 2.00 GHz (Haswell) processors, each with 35 MB L3 cache and 14 cores. Quad has 1 TB of 1 GHz DDR3 memory and 4 Intel Xeon E7-4820 v2 2.00 GHz (Ivy Bridge) processors, each with 16 MB L3 cache and 8 cores. The use of Quad is contained in Section 7.5. The key differences of Quad relative to Dual are that:

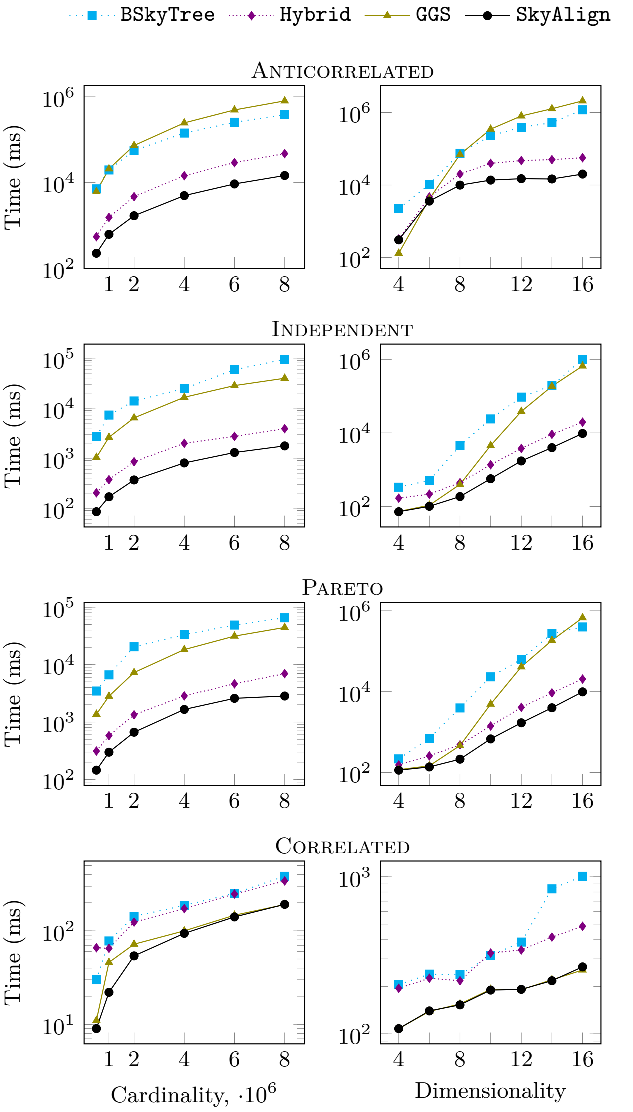


Fig. 10: Execution time on the CPU as a function of n (left) and d (right) ($d = 12$, $n = 8{\cdot}10^6$, $t = 28$, Dual).

- the speed of memory is slower;
- there is 2.0 (not 2.5 MB) of L3 cache per core (and less than half the L3 cache per socket);
- there are four more cores, but twice as many sockets;
- Ivy Bridge processors use AVX, rather than AVX2, so the DTs are up to 2× slower (depending on d);
- the QPI intersocket link is 7.2 GT/s vs. 9.6 GT/s.

As our objective is to assess the performance characteristics of GGS and SkyAlign when trivially ported to the CPU, we modify the implementations used in the previous section as minimally as possible[11] while still being fair. To adapt to the coarse-grained parallelism

[11] CUDA 7 and C++ are similar enough that converting from the former to the latter is trivial.

	1	2	4	7	8	14	28
SkyAlign	37.0 s	1.95×	3.74×	6.43×	7.28×	11.71×	-
	-	1.96×	3.82×	6.30×	7.16×	12.31×	20.77×
GGS	811.1 s	1.95×	3.67×	6.40×	7.36×	12.27×	-
	-	1.98×	3.92×	6.39×	7.29×	12.55×	24.15×
Hybrid	67.6 s	1.95×	3.60×	5.98×	6.71×	10.45×	-
	-	1.93×	3.75×	5.92×	6.63×	10.43×	16.35×

Table 3: Execution time (n=8·10^6, d=12, dist=I, Dual) relative to the number of cores. Single-socket performance is listed above double-socket performance.

	1	2	4	7	8	14	28
SkyAlign	376 s	1.98×	3.73×	6.51×	7.46×	13.02×	-
	-	1.98×	3.90×	6.43×	7.34×	12.93×	24.15×
GGS	19912 s	1.93×	3.67×	6.41×	7.31×	12.26×	-
	-	1.99×	3.91×	6.37×	7.27×	12.59×	23.97×
Hybrid	877 s	1.96×	3.71×	6.26×	7.09×	11.05×	-
	-	1.97×	3.82×	6.16×	7.06×	11.34×	17.82×

Table 4: Execution time (n=8·10^6, d=12, dist=A, Dual) relative to the number of cores. Single-socket performance is listed above double-socket performance.

of the CPU while remaining loyal to the original GPU algorithms, we make the following modifications:

- We assign an OpenMP dynamic schedule of size 32 to match the warp size assumed by GGS and SkyAlign;
- DTs are vectorised for both algorithms (as in Hybrid and BSkyTree) rather than manually unrolled (as they were on the GPU);
- We use the more aggressive CPU prefilter from [5] rather than the GPU-friendly one;
- The NoSyncs version of SkyAlign is used, because there are no warps to repack on the CPU.

Otherwise, the publicly-available implementations[12] of SkyAlign and GGS are virtually identical to those in Section 6 and are deliberately NUMA-oblivious.

7.2 Run-time performance and scalability

Workload scalability In Figure 10, we repeat the experiments from Section 6.2.1, this time with the GPU algorithms (SkyAlign and GGS) ported to the CPU; additionally, we exploit having more memory by increasing the default cardinality to $n = 8{\cdot}10^6$. As with Figure 5, Figure 10 shows run times on a logarithmic scale for each of the four algorithms on each distribution (increasing correlation from top to bottom).

In comparison to Figure 5, the trends remain unchanged. This is strong evidence in favour of portability of the GPU algorithms, since their behaviour is consistent on both architectures. Nonetheless, there are several interesting contrasts to Figure 5. First, BSkyTree runs slower on this machine, given the lower clock frequency, whereas Hybrid runs faster, because it benefits from the extra 24 cores. Given the noteable differences between Dual and Desktop, we encourage due caution in directly comparing values across the figures.

On low-dimensional and correlated data, GGS now typically performs best. It runs faster on the CPU than on the GPU in these cases because it no longer needs to amortise the cost of host-to-device transfer. On these simpler workloads, it benefits from its simplicity, whereas the greater sophistication in the memoisation-based algorithms (BSkyTree, Hybrid, and SkyAlign) is ineffective overhead. This matches the observation in [5], that more naive algorithms do best on correlated data.

While BSkyTree was consistently afflicted by the data skew on the previous machine, the extra 27 MB of L3 cache available to the single-threaded algorithm here dampens the effect and produces faster run times on skewed data than uniform, independent data *for some of the cardinalities.* The parallel algorithms, which now all use the same pre-filter, are all minorly slowed down.

Having increased the workload size for correlated data, the effect of vectorising DTs is salient, especially for Hybrid. Since the relative frequency of DTs as a percentage of computation time is high on correlated data, the cost of each DT is more important. When dimensionality aligns with the 128-bit (256-bit) AVX(2) registers, there is a clear boost in performance.

In general, we observe that SkyAlign outperforms the state-of-the-art Hybrid on the CPU by a small but consistent factor. This is an excellent result for an algorithm designed for a different architecture, especially since Figure 7 shows that SkyAlign and Hybrid have roughly equal work-efficiency. Subsequent experiments will identify why this performance difference exists.

Parallel scalability Tables 3 and 4 show how the algorithms scale with parallelism for independent and anticorrelated data, respectively. (None of the algorithms scale well on correlated data, as will be evident later.)

The leftmost column gives single-threaded run times. The subsequent columns indicate speed-up relative to single-threaded performance for a given thread count. The top row for each algorithm shows single-socket performance; the bottom row, dual-socket. For example, on anticorrelated data (Table 4), GGS uses 19912 s/12.59 = 1582 s on fourteen cores distributed over two sockets.

GGS scales best, achieving ≈ 24× speed-up on 28 cores for both distributions. However, as the theme of

[12] If the paper is accepted, code will be published as a newer release of https://github.com/sean-chester/SkyBench.

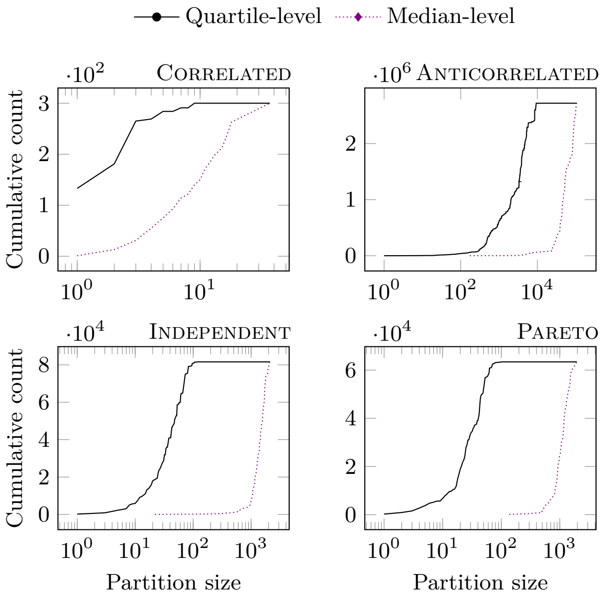


Fig. 11: Cumulative count of data points relative to the number of points in each partition ($n = 8{\cdot}10^6, d = 6$).

this paper has suggested, it also has the worst performance for these workloads on account of its poor work-efficiency. `SkyAlign` achieves a couple- to several-factor improvement over `Hybrid`, and parallelises better, particularly as the thread count gets relatively high. Interestingly, on this dual-socket machine, none of the algorithms are noticeably affected by NUMA; especially at 14 threads, dual-socket is often faster. When only half the cores of each socket are used, there is twice as much L3 cache available, which compensates for the additional latency to migrate data across NUMA nodes.

Anticorrelated data produces the best parallel scalability, because MTs are more often false. Therefore, fewer DTs need to be conducted and branch prediction is easier. Moreover, there is more work to parallelise.

7.3 Common pre-filtering and grid-based partitioning

Figure 11 analyses the efficacy of the quartile-based partitioning for each distribution at $d = 6$: the trends become more severe as d increases. The x-axis (on a log scale to illuminate the more interesting small values) specifies how many points are in a partition, i.e., the *partition size*. These cumulative plots show, for a given x value, how many points are in a partition that contains $\leq x$ points, in order to indicate how well the partitioning method disperses the dataset into separate partitions. Observe that, as the partitioning occurs after the pre-filtering, the maximum y value is precisely the number of points that were not pre-filtered.

The correlated data is shown in the top-left subfigure, with the dotted line representing the partitions at the first level of the tree (i.e., only medians) and the solid line, the second level (i.e., quartiles). In general, quartile partitions contain fewer points; so, the solid line reaches higher values earlier. Observe that the peak value, $y = 300$, is only 0.00375 % of the dataset. The pre-filter is extremely effective on the correlated data, given that each point is very likely dominated by another point that is close to the origin. Even at $d = 12$ (not shown), only 7 573 points remain after pre-filtering. As a result, there is not much work to parallelise, making the sequential `BSkyTree` competitive[13] and the overall difference between all algorithms quite minor. Given the dearth of points, it is unsurprising that no quartile-level partition contains more than nine of them.

The anticorrelated data (top-right) expresses the opposite patterns. The prefilter leaves 2 719 184 $6d$ and 6 983 464 $12d$ points, because there are fewer positive dominance relationships in the dataset for the prefilter to exploit. However, this copious incomparability is precisely what the mask tests (MTs) are leveraging effectively. Naturally, the partition sizes are much larger, since 2 719 184 points are distributed among only $4^6 =$ 4 096 partitions; however, already at $d = 12$ (not shown) 88 % of data points are in partitions of size ≤ 9.

Considering the independent attributes, with uniform (bottom-left) and pareto (bottom-right) distributions, we expect the latter data points to cluster closer to the origin. However, the quartiles of the dataset then similarly cluster closer to the origin, producing partition sizes that are roughly the same. The most notable difference between the distributions is instead the efficacy of the prefilter, which prunes 25 % more of the pareto data, but this is not a consistent trend: at $d = 12$, the prefilter leaves 2 199 119 pareto points and 1 965 026 independent points. Because the attributes are independent of each other, even for the pareto data, the points occupy more of the data space and are thus well dispersed. Already at $d = 6$, all points are in partitions with ≤ 125 points; by $d = 12$, no partition contains more than 5 points. This dispersion provides near-unique masks for each point, ergo more effective MTs, which enables the work-efficiency of `SkyAlign`.

7.4 Performance profiling

Section 7.2 showed that `SkyAlign` and `GGS` scale well when ported to multicore, and that all three algorithms are fairly resistant to NUMA effects (although they suf-

[13] `BSkyTree` does not use the pre-filter that the other three methods use, but its pivot selection routine has a similar effect.

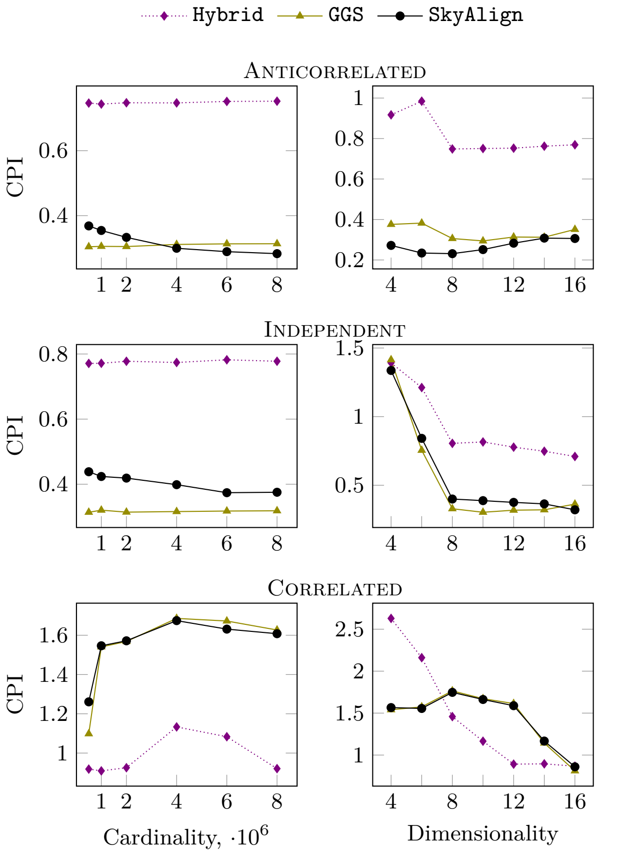


Fig. 12: Cycles per instruction (CPI) as a function of n (left) and d (right) ($t = 14$, Dual, 1 socket).

fer diminishing returns with more active cores, even on a single socket). It answered *whether*. Here, we endeavour to explain *why* these trends exist by profiling the hardware for low-level, concrete explanations.

To derive these explanations, we use the PAPI library to measure hardware counters from within the software. Profiling begins immediately before and ends immediately after the timer of the previous sections; i.e., it includes the building of the SkyAlign data structure, running the pre-filters, etc., but not reading of data from disk to memory. We count the values for each thread independently, summing them at the end to obtain the values that we report.

A primary goal of these low-level investigations is to observe not only how these algorithms behave, but also how the specific GPU restrictions impact CPU hardware throughput. We profile in three categories, first looking at instruction throughput in general, then branching behaviour, and lastly cache performance.

7.4.1 Instruction throughput analysis

This paper proposes trading compute throughput for work-efficiency. Here, we quantify that trade off for the CPU implementations. Generally speaking, we expect that instruction throughput will be a bit higher on the CPU, because it supports out-of-order execution. That is to say, while some instructions are stalled by load latencies, other non-dependent instructions, even if they occur later, can be executed in the meantime. This increases instruction-level parallelism (ILP) relative to the GPU and helps to hide the load latencies for the DTs. We profile this with three separate experiments.

First, Figure 12 shows the cycles per instruction retired (CPI) by each algorithm as the workload changes. Low CPI values indicate high compute throughput (few cycles are required for each instruction). Values below 1 are possible, because modern computers can retire multiple instructions at once (via *instruction-level parallelism*). On Intel Haswell architectures the theoretical minimum average CPI is 0.25, since a maximum of four instructions can be delivered by the front-end per cycle.[14] We measure CPI on a single socket to focus first on performance profiles without NUMA complications.

We see immediately that Hybrid does not achieve the same level of compute throughput as the GPU algorithms for most workloads (not correlated and $d > 4$). This explains why, with the same work-efficiency as SkyAlign, it does not achieve the same execution times: it requires $\geq 2\times$ as many cycles to retire each instruction. On the other hand, GGS consistently has near-best or best compute throughput of the algorithms, typically ≤ 0.30 for workloads that are not correlated and for which $d > 6$. Yet, on these very workloads it is the slowest running algorithm. Together, these observations indicate the mutual importance of compute throughput and work-efficiency.

Looking closer at cardinality for non-correlated data, GGS and Hybrid are very stable; SkyAlign exhibits improved CPI with increasing workload size, even surpassing GGS on high-cardinality or anticorrelated data.

We observe different behaviour with respect to dimensionality. Here all algorithms improve on correlated data, as the amount of computational work increases. Note that even at $d = 16$, the CPI is barely under 1.0 and worse than the other distributions. This partly explains the universally poor scalability on the distribution. The CPI of SkyAlign degrades on anticorrelated data as d increases, but still remains ≤ 0.35.

[14] Up to eight instructions can be retired in a cycle if they are ready for execution, but the front-end (the instruction fetch-decode cycle) populates the queue at a slower rate.

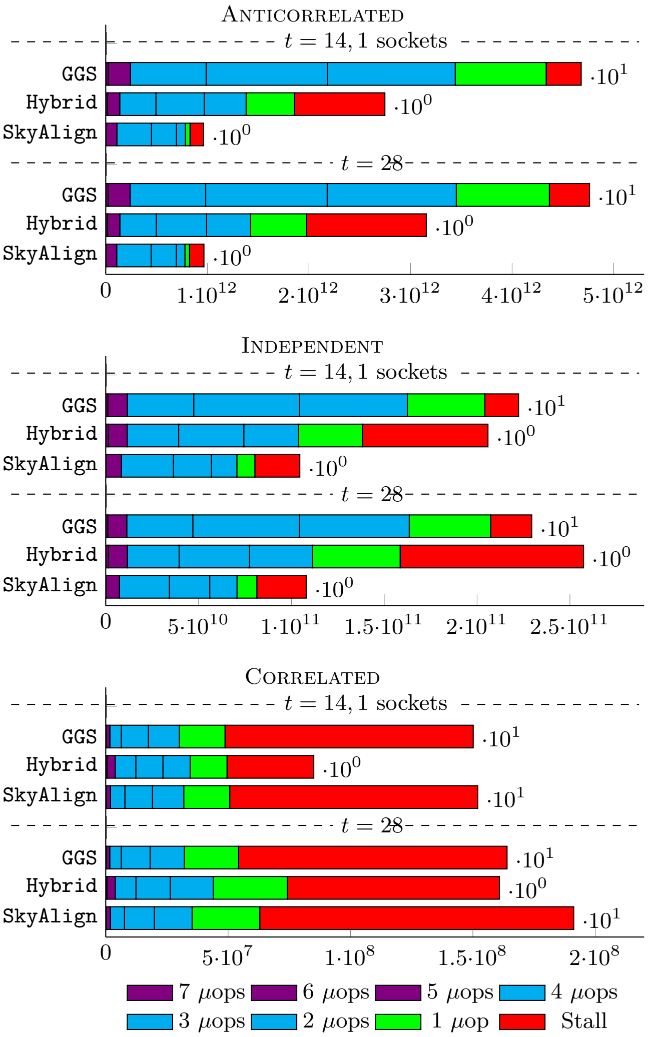


Fig. 13: Cycles by number of μops executed (`Dual`, $n = 8{\cdot}10^6, d = 12$, $t = 28$). Bars have increasing μops towards the left and are coloured by superscalar performance: violet is outpacing the front-end; cyan is superscalar; green exhibits no ILP; and red is stalled.

Figure 13 provides another view of compute throughput by directly reporting instruction-level parallelism (ILP). The cycle count is plotted on the x-axis, broken down by how many micro-operations (μops) were retired on each cycle. Retiring more μops in a cycle requires more ILP and lowers the CPI (and run time). Note that, to represent `Hybrid` and `SkyAlign` more clearly, `GGS` is compressed by an order of magnitude in all cases and `SkyAlign` is similarly compressed on correlated data. On red cycles, no μops were executed (i.e., execution was *stalled*); green cycles retire only one instruction, obtaining poor throughput; cyan to violet cycles indicate increasing ILP.

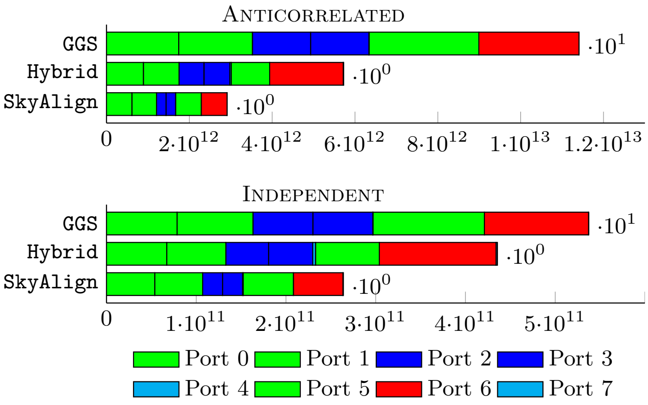


Fig. 14: Breakdown of μops by port ($n = 8{\cdot}10^6, d = 12$, $t = 28$, `Dual`), ordered by port number from left to right. Ports are coloured by similarity of function.

We can see a general alignment between Figure 12 and Figure 13: non-correlated data, which feature lower CPI values, have significantly fewer stalled cycles than correlated data. On correlated data, for all algorithms, more than half of cycles are stalled. As a result, adding more cores does not improve performance much, since existing cores are stalled anyway.

`Hybrid` has fewer superscalar cycles (cycles with more than one μop retired). In constrast, `SkyAlign` on anticorrelated data, in particular, retires at least 3 μops on 75 % of cycles. This high ILP drives the relatively lower CPI of `SkyAlign`.

Figure 13 also provides the first insight into parallel scalability: for each distribution, values for 14 threads on one socket are given above those for 28 threads on two sockets. Observe that for `SkyAlign` and `GGS`, neither the overall number of cycles nor the distribution of μops retired changes when extra threads are added (for non-correlated data). On the other hand, `Hybrid` gains cycles and the new cycles almost exclusively retire ≤ 1 μop. Adding extra cores for `Hybrid` also adds extra cycles (with no ILP) in overhead. For correlated data, `GGS` still maintains roughly the same profile, but `Hybrid` and `SkyAlign` gain considerable stalled and non-ILP cycles.

Figure 14 presents our last view of compute throughput. Here we show how μops are distributed on the different execution ports. Intel Haswell architectures have eight execution ports, and each is responsible for only a subset of possible μops. While ILP depends on a diverse use of executions ports, a high ratio of compute operations (on ports 0, 1, 5, and 6) to other operations minimises reliance on the higher latency memory subsystem. Thus, this figure give both a view of potential ILP and of compute-ratio. The green ports correspond to ones primarily responsible for compute and vector

	Correlated	Independent	Anticorrelated
SkyAlign	$1.7\cdot10^{6}$ (0.92 %)	$1.1\cdot10^{9}$ (1.85 %)	$6.4\cdot10^{9}$ (0.85 %)
GGS	$6.4\cdot10^{5}$ (0.49 %)	$1.3\cdot10^{9}$ (0.10 %)	$1.1\cdot10^{10}$ (0.04 %)
Hybrid	$7.5\cdot10^{5}$ (2.85 %)	$2.1\cdot10^{9}$ (3.55 %)	$2.3\cdot10^{10}$ (3.02 %)

Table 5: Number of branch instructions mispredicted (with misprediction rate in parentheses) relative to data distribution (n=$8\cdot10^{6}$, d=12, $t = 28$, `Dual`).

	Correlated	Independent	Anticorrelated
SkyAlign	$1.0\cdot10^{8}$(5.84 %)	$5.9\cdot10^{10}$(23.2 %)	$3.6\cdot10^{11}$(11.9 %)
GGS	$7.1\cdot10^{7}$(2.77 %)	$1.2\cdot10^{11}$(1.97 %)	$5.9\cdot10^{11}$(0.49 %)
Hybrid	$5.2\cdot10^{7}$(24.4 %)	$1.7\cdot10^{11}$(36.2 %)	$2.0\cdot10^{12}$(35.9 %)

Table 6: Number of μops issued but not retired (and as a percentage of issued μops) relative to data distribution (n=$8\cdot10^{6}$, d=12, $t = 28$, `Dual`).

μops, where, for example, DTs and MTs are executed. The blue ports are primarily responsible for memory operations (stores and loads), and the red port 6 is a newer port on Intel Haswell architectures that, in the case of these algorithms, does MTs and branching.

We make two interesting observations here: first, there is a disproportionately large number of μops being executed on Port 6 for `Hybrid`, which does not promote ILP. This suggests there may be quite a lot of branching logic, which, relative to the GPU algorithms, we might expect. Second, `SkyAlign` has an especially high ratio of μops on compute ports (0, 1, 5, and 6) relative to memory ports (2 and 3). Generally speaking, a good compute-ratio (i.e., a large number of compute operations per memory operation) is favourable for parallelism. This is what we would hope to see by swapping DTs for MTs, that the overall load on the machine for memory-related operations would decrease.

To summarise our instruction throughput findings, the throughput of `Hybrid` is quite good, with a CPI less than 1 for the majority of workloads. Consequently, it obtains a performance boost from adding cores. However, the addition of cores adds more stall cycles, so the return on parallel scalability diminishes at higher thread counts. By contrast, the algorithms developed for the GPU clearly obtain good instruction throughput on the CPU, where additional ILP comes from out-of-order execution. They achieve CPI values close to the theoretical optimum on harder workloads, partly because they have a diverse distribution of μops onto different ports, which leads to more superscalar cycles.

These three experiments have demonstrated that the GPU algorithms port well in terms of throughput onto multicore, but the central question of *why* remains unanswered. The next experiments look into what causes the high ILP or, conversely, the stalls.

7.4.2 Branch prediction

A central tenet of GPU algorithms (c.f., Section 2.2) is to minimise branching. While the GPU requires branch-free code on account of step-locked threads, the CPU also benefits from having better branch prediction. Here we look at how significantly the branching aversion of the GPU algorithms impacts performance on the CPU.

Table 5 shows branch mispredictions for the algorithms, both as absolute numbers and rates. Unsurprisingly, we observe the misprediction rates of the GPU algorithms are low. `GGS` is particularly impressive: despite an order of magnitude more branch instructions than the other algorithms (on account of its poor work-efficiency), it has fewer mispredictions than `Hybrid` consistently, and never more than twice that of `SkyAlign`.

`Hybrid` and `SkyAlign` necessarily have more branching, because they contain more logic to avoid unnecessary work. `SkyAlign` has consistently better branch prediction rates than `Hybrid`. Recall that both algorithms feature a tree traversal. Neither tree exhibits clearly strided behaviour in patterns of whether or not to visit child nodes; so, branch prediction will be difficult in both cases. However, since `SkyAlign` is a static partitioning, points are grouped by their relation (i.e., median and quartiles masks) relative to the data space and branches are determined by the masks of the groups. Consequently, there are likely to be “runs” of consecutively branching in the same direction. This is much less likely with `Hybrid`, where the relationship between a point and the root of the current subtree is determined dynamically; in this case, such “consecutive runs” are likely to be broken quickly and often.

Table 6 presents a different view of branch mispredictions: it describes the number of μops that are issued but never retired, again both in absolute numbers and as a rate. This roughly quantifies the *bad speculation*, μops that exist downstream on a branch misprediction prior to its being recognised as such. Between the machine clears required per branch misprediction and the number of badly speculated instructions in this table, we can capture the rough cost of branch misprediction.

The general patterns are, unsurprisingly, the same as in Table 5, except that `GGS` has more badly speculated μops than `Hybrid` on correlated data. Also, the ratios between algorithms change. These changing ratios suggest a difference in how deep are the pipelines that need to be cleared per branch misprediction. In the case of `GGS`, where the branch misprediction rate is very low, each misprediction’s cost is relatively higher.

	Correlated	Independent	Anticorrelated
`SkyAlign`	$7.1{\cdot}10^{8}$(32.3 %)	$1.2{\cdot}10^{11}$(46.9 %)	$1.9{\cdot}10^{12}$(71.5 %)
`GGS`	$7.3{\cdot}10^{8}$(67.5 %)	$5.5{\cdot}10^{12}$(93.3 %)	$1.1{\cdot}10^{14}$(97.0 %)
`Hybrid`	$1.2{\cdot}10^{7}$(5.74 %)	$9.0{\cdot}10^{9}$ (1.95 %)	$2.2{\cdot}10^{11}$(4.00 %)

Table 7: Number of μops delivered from the Loop Stream Detector (and as a percentage of all issued μops) relative to distribution (n=$8{\cdot}10^6$, d=12, $t = 28$, `Dual`).

Finally, Table 7 illustrates an interesting side-effect of good branch prediction. The *loop stream detector* (LSD) on a modern Intel machine can cache the front-end work in fetching and decoding instructions for small loops. When the LSD is active, the unused front-end components of the execution pipeline are temporarily disabled, saving both cycles and energy. The LSD requires loops with ≤ 28 instructions and is de-activated at the first misprediction of the loop's condition. Table 7 shows how many μops are delivered from the LSD rather than the usual, full-fledged front-end pipeline.

Remarkably, `GGS` is almost entirely satisfied by the LSD, and `SkyAlign` obtains one third to three quarters of its μops from the LSD. None of the three algorithms are front-end bound, so, except for the potential energy savings, LSD utilisation is perhaps not significant itself. However, such extensive use of the LSD requires highly predictable branching in the most frequently executed inner loops. Therefore, it presents alternative evidence that, not only is the branching behaviour good for the GPU algorithms, but it is also good exactly where the majority of the μops are being delivered.

As for `Hybrid`, it is not just the higher branch misprediction rates that render the LSD unusable. `Hybrid` also features a larger inner loop that exceeds the 28 instruction limit. This is directly a result of needing to conduct DTs in order to traverse the tree, which provides enough additional instructions to surpass the LSD limit. (Recall `SkyAlign` only conducts DTs at the leaves of the tree, not at the inner nodes of the traversal.)

7.4.3 The memory subsystem

Figure 13 indicated that all algorithms have stalled cycles. Even in the best case, `SkyAlign` on anticorrelated data had 25 % of cycles retiring two or fewer μops (i.e., at fifty percent capacity or less). This subsection looks at the performance of the memory hierarchy to identify the source of the stalled cycles. Our design choices for the layout of the data structures (principally, padded data points and non-padded bitmasks) are tested here.

Table 8 describes the number of cache misses for each algorithm, both in absolute numbers and as a rate. For each algorithm, the table contains two rows, one for the L2 cache that is local to each core (above) and one for the shared L3 cache (below). L3 misses are always fewer than L2 misses, because Haswell machines have inclusive caches: any cache line in L2 is also in L3.

	Correlated	Independent	Anticorrelated
`SkyAlign`	$1.6{\cdot}10^{7}$(85.9 %)	$1.2{\cdot}10^{9}$(44.4 %)	$9.4{\cdot}10^{9}$ (66.0 %)
	$6.2{\cdot}10^{6}$(37.9 %)	$3.3{\cdot}10^{7}$(2.77 %)	$2.4{\cdot}10^{8}$ (2.51 %)
`GGS`	$1.6{\cdot}10^{7}$(77.3 %)	$7.1{\cdot}10^{8}$(0.13 %)	$6.0{\cdot}10^{9}$ (0.05 %)
	$6.0{\cdot}10^{6}$(36.2 %)	$1.3{\cdot}10^{8}$(18.4 %)	$2.7{\cdot}10^{9}$ (44.8 %)
`Hybrid`	$3.3{\cdot}10^{5}$(12.7 %)	$9.0{\cdot}10^{9}$(72.3 %)	$1.5{\cdot}10^{11}$(76.9 %)
	$5.2{\cdot}10^{4}$(15.7 %)	$1.3{\cdot}10^{7}$(0.14 %)	$9.4{\cdot}10^{8}$ (0.62 %)

Table 8: Number of cache misses (and as a percentage of cache accesses) relative to data distribution (n=$8{\cdot}10^6$, d=12, $t = 28$, `Dual`). L2 misses are listed above L3.

Considering non-correlated data, we see that `GGS` has significantly better usage of its L2 cache than do either `SkyAlign` or `Hybrid`, in fact having fewer absolute misses despite orders of magnitude more accesses. This testifies to the efficacy of the *tiling* in `GGS`: a block of α points are loaded into L2 cache and then thoroughly processed. `Hybrid` also attempts tiling, but has a much more difficult time of it. The tiling is easy for `GGS`, because it simply loads a block of points and then conducts DTs between those points and non-dominated points. However, `Hybrid` attempts to skip work at the same time. If α is chosen too small for `Hybrid`, then there is not enough work to parallelise. On the other hand, a large value of α and some bad luck leads to cache eviction, a difficult and data-dependent balance to strike. This `NoSync` version of `SkyAlign` foregoes attempts at tiling, so has poor L2 performance. There is no `NoSync` option for `Hybrid`, because `Hybrid` constructs its tree on-the-fly, thereby requiring the tiling and synchronisation to construct its data structure.

`Hybrid` outperforms `SkyAlign` in terms of cache miss rates, but `SkyAlign` has significantly fewer absolute L2 misses (on non-correlated data) on account of having fewer L2 accesses. This testifies to the success of `SkyAlign` in minimising memory operations by replacing DTs with MTs in the tree traversal. Although each data point fits on a cache line, each cache line fits *sixteen bitmasks*; so, each L2 access provides the masks required for the next sixteen sequential MTs. `Hybrid` needs to fetch the actual data points in order to construct the bitmasks for the MTs, rather than pre-computing them as in the global, static partitioning scheme of `SkyAlign`.

`Hybrid` has the best off-core (> L2) cache performance, at least as a percentage. This is important, because while off-core latencies are highest, they are also the most susceptible to NUMA effects: using additional sockets leads to a last level cache that is no longer

	Correlated	Independent	Anticorrelated
SkyAlign	$1.4 \cdot 10^9$(56.0 %)	$1.4 \cdot 10^{10}$(13.1 %)	$8.0 \cdot 10^{10}$(8.20 %)
	$1.6 \cdot 10^9$(68.2 %)	$1.9 \cdot 10^{10}$(18.1 %)	$1.1 \cdot 10^{11}$(11.0 %)
GGS	$8.5 \cdot 10^8$(52.2 %)	$2.6 \cdot 10^{10}$(1.15 %)	$6.3 \cdot 10^{11}$(1.33 %)
	$1.0 \cdot 10^9$(61.4 %)	$1.7 \cdot 10^{11}$(7.53 %)	$3.5 \cdot 10^{12}$(7.35 %)
Hybrid	$9.6 \cdot 10^7$(39.0 %)	$2.9 \cdot 10^{10}$(11.6 %)	$5.3 \cdot 10^{11}$(17.8 %)
	$1.0 \cdot 10^9$(48.3 %)	$4.4 \cdot 10^{10}$(18.0 %)	$6.5 \cdot 10^{11}$(21.5 %)

Table 9: Number of cycles stalled with pending cache requests (and as a percentage of cycles) relative to data distribution (n=$8 \cdot 10^6$, d=12, t = 28, Dual). L2 stalls are above those for the entire memory subsystem.

	1	2	4	8	16	32
SkyAlign	48.1 s	1.89×	3.64×	7.04×	-	-
	-	1.94×	3.69×	6.94×	13.23×	-
	-	-	3.77×	6.77×	12.47×	23.61×
GGS	1196.9 s	1.78×	3.43×	7.34×	-	-
	-	1.85×	3.54×	6.77×	14.35×	-
	-	-	3.67×	6.91×	13.32×	27.79×
Hybrid	84.7 s	1.88×	3.55×	6.60×	-	-
	-	1.94×	3.60×	6.52×	11.20×	-
	-	-	3.70×	6.57×	11.17×	16.54×

Table 10: Execution time (n=$8 \cdot 10^6$, d=12, dist=I, Quad) relative to the number of cores. Single-, dual-, and quad-socket performance are listed vertically.

shared among all cores. For Hybrid, which writes to its last-level cache as it updates its data structure on-the-fly, this also leads to cache-coherency issues.

The good L3 cache performance of Hybrid arises from its traversal strategy. Nodes near the top of the tree are visited more often; so, the points corresponding to the roots of large sub-trees are frequently accessed and likely L3-resident. In contrast, SkyAlign accesses raw data points very unpredictably, so these are much less likely to be L3-resident; L3-resident points are those in partitions with very low median and quartile bitmasks, against which many points are required to conduct DTs. For GGS, which has the worst L3 cache performance, each data point must be fetched from memory when the thread first begins work on it. However, subsequent L2 cache performance compensates this cost.

Table 8 indicated how frequent cache misses were, but that does not necessarily mean that they are responsible for the stalls; all three algorithms have mechanisms for hiding latency. GGS has very good branch prediction, enabling the pre-fetcher to retrieve the data points well in advance. SkyAlign has very good ILP, so the out-of-order execution engine can busy itself with other work. Hybrid has good L3 cache performance to minimise the frequency of the worst-case memory latencies. Table 9 describes how often cache misses coincide with stalled cycles, giving a truer estimate of how often the execution ports are starved by missing operands.

Like before, stalls coincident with L2 pending requests are shown above those for the entire memory subsystem. Unlike before, L2 stalls are a subset of memory stalls and therefore always less frequent. A low value of L2 relative to memory stalls would indicate L2-bound computation. This is not true of any of the algorithms, which instead are stalled by L3/memory load requests.

The poor compute throughput on correlated data is explained by a very high percentage of cycles waiting for memory loads. Clearly, then, adding additional cores on correlated data is ineffective, as it will simply increase demand for memory loads. We can also observe that the ratio of cache stalls between SkyAlign and Hybrid is roughly proportionate to the difference in their running times. An interesting observation is that Hybrid suffers fewer L3 misses but more memory stalls than SkyAlign. This relates back to its high compute-ratio: With more compute μops per cache line read and ILP for the out-of-order execution engine to exploit, SkyAlign can hide latencies better. By contrast, latencies are not hidden as well in Hybrid nor especially in GGS, because all cores stop and synchronise at the same time when done their tiles, which creates a *bursty* behaviour to the reads from (potentially off-socket) memory. Thus, we conclude that it is the combination of increased number of absolute L2 accesses (resulting in more L2 misses and moreover more L2 stalls) and decreased ability to hide the resultant latencies that explains the difference in performance between the algorithms. The re-use of cache lines containing quartile-level masks, rather than the dynamic computation of bitmasks, reduces the total number of memory accesses.

GGS does remarkably on these metrics too, often with a competitive number of absolute L2 and memory stalls. However, it is the sheer volume of work done by the algorithm that prevents its good compute throughput from materialising as raw performance, except on correlated data where no algorithm has much work.

7.5 Scalability across NUMA architectures

Given that the memory subsystem stalls are limiting the compute throughput on many workloads, we investigate here the effect that the higher NUMA latencies on the Quad machine might have on performance.

7.5.1 Parallel scalability

Tables 10 and 11 repeat on the quad-socket machine the experiments in Tables 3 and 4 for independent and anticorrelated data, respectively. They show single-threaded

	1	2	4	8	16	32
SkyAlign	492 s	1.89×	3.66×	7.29×	-	-
	-	1.97×	3.76×	7.18×	14.20×	-
	-	-	3.88×	7.29×	14.07×	27.25×
GGS	24606 s	1.77×	3.42×	7.34×	-	-
	-	1.84×	3.53×	6.76×	14.40×	-
	-	-	3.67×	6.93×	13.32×	28.03×
Hybrid	1298 s	1.85×	3.54×	6.82×	-	-
	-	1.91×	3.47×	6.42×	11.57×	-
	-	-	3.59×	6.35×	11.20×	18.07×

Table 11: Execution time (n=8·10^6, d=12, dist=A, Quad) relative to the number of cores. Single-, dual-, and quad-socket performance are listed vertically.

execution time (leftmost column) and parallel scalability at various thread and socket counts. Compared to before, each algorithm is given an extra (lower-most) row to show performance on four sockets.

Comparing the tables directly, single-threaded execution of each algorithm is slower on the quad-socket machine. This is unsurprising given the slower DDR3 memory (thus higher memory latencies) and that the single active core has less L3 cache at its dispense. Each DT also requires an extra AVX instruction at $d = 12$.

We can also directly compare speed-ups for 2, 4, and 8 cores on 1 and 2 sockets. Consistently, we see that parallel scalability is slightly reduced on the quad-socket machine. This is explained (and shown in the next experiments) exactly as for single-threaded performance: utilising more cores places greater stress on the increasingly scarce L3 cache and does not alleviate the higher latencies of outstanding cache misses.

For $t < 16$, using extra sockets produces faster run times than fewer sockets with more L3 sharing. Thus, the NUMA latencies at low core counts are compensated by the higher availability of L3 cache per active socket. At $t = 16$, all four sockets are half-occupied, and competition for all resources becomes greater. The trend reverses, and NUMA latencies are no longer compensated by doubly-available L3 cache.

As in Table 3 and 4 we see that all algorithms scale better on anticorrelated data. The GPU algorithms still scale reasonably well, with both getting $> 25\times$ speed-up on 32 cores. Hybrid struggles to utilise the fourth socket, obtaining minimally better speedup on 32 cores as on the 28 cores of the dual-socket machine.

7.5.2 Performance profiling

Table 12 repeats on Quad the experiment from Table 9, using all 32 cores. It reports the number of stalled cycles (no instructions executed) coincident with a pending L2 (top row) or any memory (bottom row) load request. The number of memory stalls as a percentage of all cycles is given in parentheses. Because there is less cache per core, we expect a higher L3 miss ratio. Because cores are distributed across four sockets, we expect a higher latency for L3 misses and for reading from memory. Finally, because memory is half the speed on Quad, we expect that latencies in general should go up.

	Correlated	Independent	Anticorrelated
SkyAlign	$8.9\cdot10^{8}$(40.1 %)	$1.8\cdot10^{10}$(13.7 %)	$1.2\cdot10^{11}$(9.09 %)
	$1.1\cdot10^{9}$(48.5 %)	$2.4\cdot10^{10}$(19.0 %)	$1.5\cdot10^{11}$(11.7 %)
GGS	$8.5\cdot10^{8}$(49.0 %)	$4.3\cdot10^{10}$(1.38 %)	$6.5\cdot10^{11}$(1.05 %)
	$9.6\cdot10^{8}$(55.2 %)	$2.6\cdot10^{11}$(8.27 %)	$4.3\cdot10^{12}$(6.89 %)
Hybrid	$1.0\cdot10^{8}$(36.7 %)	$4.9\cdot10^{10}$(15.1 %)	$1.3\cdot10^{12}$(31.6 %)
	$1.0\cdot10^{9}$(48.3 %)	$7.5\cdot10^{10}$(23.1 %)	$1.5\cdot10^{12}$(35.5 %)

Table 12: Memory stalls (n=8·10^6, d=12, $t = 32$, Quad). L2 stalls are above the whole memory subsystem's.

Comparing the tables directly, we see that GGS and SkyAlign do not exhibit substantial differences between the tables (except that SkyAlign suffers fewer stalls on correlated data). Hybrid, on the other hand, suffers quite dramatically on non-correlated data, spending up to one third of cycles stalled on memory requests. Since the increased general memory stalls are also observed as L2 stalls, the responsible loads are off-core, which aligns with our expectations. This demonstrates that the better memory access patterns of the GPU algorithms (i.e., tiling for GGS and cache-line reuse for SkyAlign) is increasingly important as latencies increase.

A couple other interesting differences arise between the machines, which we briefly describe. Ivy Bridge has two fewer execution ports than Haswell; so, we investigate whether this affects instruction-level parallelism of the algorithms (c.f., Figure 14). Naturally, the loss of an arithmetic ALU functional unit on Port 6 means that μops must be redistributed to other ports on the Ivy Bridge architecture, reducing the number of simultaneous compute operations that can be executed. Still, the compute-ratios are roughly the same and only a small percentage of cycles in Figure 14 had retired more than 5 instructions, anyway. LSD utilisation is higher for all algorithms on the quad-socket machine, and quite notably so for Hybrid: compared to Table 7, 20.0 % and 32.0 % of μops are delivered by the LSD for independent and anticorrelated data, respectively. Nonetheless, this is still substantially less than the GPU algorithms.

7.6 Summary

In summary, the GPU algorithms obtain very good performance when ported to the CPU, even in a NUMA-oblivious manner. The tiling in GGS leads to excellent utilisation of L2, and the predictable nature and specific placement of the branch instructions leads to low branch misprediction rates and activation of the loop stream detector. For `SkyAlign`, the sequential reading of bitmasks utilises each cache line well thereby requiring significantly fewer cache accesses. `GGS` cannot be shielded from its poor work-efficiency, but the work-efficient `SkyAlign` obtains a multiplicative improvement over existing state-of-the-art multicore skyline computation (namely `Hybrid`) on account of its throughput. The trade-off on the GPU hardly showed on the CPU.

The quad-tree design of the `Hybrid` algorithm hurt its parallel scalability, especially on NUMA: the more complex loop does not fit in the LSD; the reading of entire data points, rather than multiple bitmasks compressed onto one cache line, leads to more cache accesses and, inevitably, stalls; and the cache-sharing for modified cache lines in the data structure degrades in the presence of NUMA effects, because of cache-coherency.

However, `SkyAlign` also is not perfect: its L2 miss rate is volatile and high, since it does not utlise tiling; it suffers from a large number of badly speculated μops when it mispredicts branches; and it struggles to recuperate the cost of constructing its data structure when the amount of subsequent computation is quite small. However, for a direct port, it obtains excellent performance, both in terms of raw run time and hardware metrics, yielding a new, more scalable state-of-the-art.

8 Conclusion

In this paper, we first investigated skyline computation on the GPU. We showed that existing algorithms, may utilise the GPU card well, but 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 compute 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. We then studied how well the GPU algorithms perform when ported to multicore. By exposing a lot instruction-level parallelism and minimising memory stalls, `SkyAlign` obtains higher compute throughput than the existing multicore state-of-the-art, which leads to better raw performance. Thus, emphasising work-efficiency led not just to better GPU performance, but also to a portable parallel algorithm.

Acknowledgments

This research was supported through the WallViz (Danish Council for Strategic Research) and ExiBiDa (Norwegian Research Council) projects. The authors thank the Harvard DASlab for the use of their quad-socket machine and the anonymous reviewers for their helpful comments and suggestions of informative experiments.

References

1. Bartolini, I., Ciaccia, P., Patella, M.: Efficient sort-based skyline evaluation. TODS **33**(4), 31:1–49 (2008)
2. Bøgh, K.S., Assent, I., Magnani, M.: Efficient gpu-based skyline computation. In: Proc. DaMoN, pp. 5:1–6 (2013)
3. Bøgh, K.S., Chester, S., Assent, I.: Work-efficient skyline computation for the GPU. PVLDB **8**(9), 962–973 (2015)
4. Börzsönyi, S., Kossman, D., Stocker, K.: The skyline operator. In: Proc. ICDE, pp. 421–430 (2001)
5. Chester, S., Šidlauskas, D., Assent, I., Bøgh, K.S.: Scalable parallelization of skyline computation for multi-core processors. In: Proc. ICDE (2015)
6. Cho, S.R., Lee, J., Hwang, S.W., Han, H., Lee, S.W.: VSkyline: Vectorization for efficient skyline computation. SIGMOD Rec. **39**(2), 19–26 (2010)
7. Choi, W., Liu, L., Yu, B.: Multi-criteria decision making with skyline computation. In: Proc. IRI, pp. 316–323 (2012)
8. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: Proc. ICDE, pp. 717–719 (2003)
9. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. TODS **34**(4), 1–39 (2009)
10. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational joins on graphics processors. In: Proc. SIGMOD, pp. 511–524 (2008)
11. Hose, K., Vlachou, A.: A survey of skyline processing in highly distributed environments. VLDB J **21**(3), 359–384 (2012)
12. Im, H., Park, J., Park, S.: Parallel skyline computation on multicore architectures. Inf. Syst. **36**(4), 808–823 (2011)
13. Kaldewey, T., Lohman, G., Mueller, R., Volk, P.: GPU join processing revisited. In: Proc. DaMoN, pp. 55–62 (2012)
14. Lee, J., Hwang, S.w.: Scalable skyline computation using a balanced pivot selection technique. Information Systems **39**, 1–24 (2014)
15. Lee, K.C.K., Zheng, B., Li, H., Lee, W.C.: Approaching the skyline in Z order. In: Proc. VLDB, pp. 279–290 (2007)
16. Mullesgaard, K., Pedersen, J.L., Lu, H., Zhou, Y.: Efficient skyline computation in MapReduce. In: Proc. EDBT, pp. 37–48 (2014)
17. Papadias, D., Tao, Y., Fu, G., Seeger, B.: Progressive skyline computation in database systems. TODS **30**(1), 41–82 (2005)
18. Park, Y., Min, J.K., Shim, K.: Parallel computation of skyline and reverse skyline queries using MapReduce. PVLDB **6**(14), 2002–2011 (2013)
19. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient progressive skyline computation. In: Proc. VLDB, pp. 301–310 (2001)
20. Vlachou, A., Doulkeridis, C., Kotidis, Y.: Angle-based space partitioning for efficient parallel skyline computation. In: Proc. SIGMOD, pp. 227–238 (2008)
21. Woods, L., Alonso, G., Teubner, J.: Parallel computation of skyline queries. In: Proc. FCCM, pp. 1–8 (2013)
22. Zhang, K., Yang, D., Gao, H., Li, J., Wang, H., Cai, Z.: VMPSP: Efficient skyline computation using VMP-based space partitioning. In: Proc. DASFAA Workshops, pp. 179–193 (2016)
23. Zhang, S., Mamoulis, N., Cheung, D.W.: Scalable skyline computation using object-based space partitioning. In: Proc. SIGMOD, pp. 483–494 (2009)