Efficient GPU-based skyline computation

Kenneth S. Bøgh
Aarhus University, Denmark
u071354@cs.au.dk

Ira Assent
Aarhus University, Denmark
ira@cs.au.dk

Matteo Magnani
ISTI, CNR, Italy
matteo.magnani@isti.cnr.it

ABSTRACT

The skyline operator for multi-criteria search returns the most interesting points of a data set with respect to any monotone preference function. Existing work has almost exclusively focused on efficiently computing skylines on one or more CPUs, ignoring the high parallelism possible in GPUs. In this paper we investigate the challenges for efficient skyline algorithms that exploit the computational power of the GPU. We present a novel strategy for managing data transfer and memory for skylines using CPU and GPU. Our new sorting based data-parallel skyline algorithm is introduced and its properties are discussed. We demonstrate in a thorough experimental evaluation that this algorithm is faster than state-of-the-art sequential sorting based skyline algorithms and that it shows superior scalability.

1. INTRODUCTION

The skyline operator finds all points which optimize any monotone scoring function [4]. As opposed to top-k or ranking approaches in general, the scoring function does not need to be specified. The skyline consists of those data points that are not dominated by other data points in the data set, i.e., their values are at least as good in all dimensions, and strictly better in at least one dimension.[1]

EXAMPLE 1. *Consider buying a car based on the criteria mileage and price (in Fig. 1 cars are 2d points). Cars with low mileage tend to have a high cost and vice versa. The acceptable trade-off between mileage and price depends on user preferences, and is often unknown at query time. The best options for any monotone preference function are the skyline points d1, d3, d7, d12. Note that while d3 and d7 do not minimize either of the dimensions, none of the other points have at the same time lower mileage and price. Non-skyline points such as d8 are dominated by (skyline) points such as d7 in both dimensions.*

[1]We assume better means smaller without loss of generality.

DaMoN'13, June 24 2013, New York, NY, USA

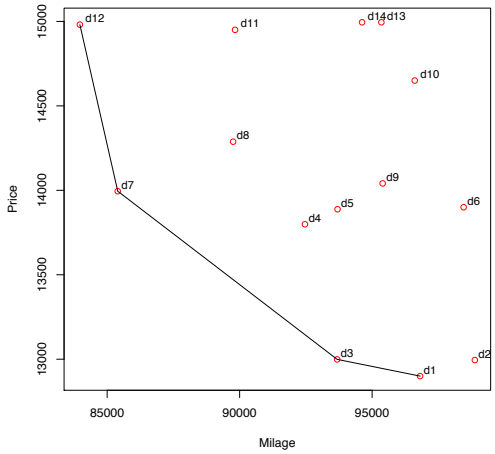


Figure 1: Example skyline on car database

Although several efficient skyline algorithms have been proposed, computing a skyline is still challenging for specific data distributions, high dimensionalities and interactive data analysis scenarios where query results are usually expected in a very short time [10].

Parallelization has been a successful approach for efficient query processing for many years, and research was done even before computers with this ability existed [13]. Also General Purpose computation on the GPU (GPGPU) was introduced in 1978 [7], and has been studied in many computationally intensive areas. This research has led to increased hardware support with GPGPU frameworks such as CUDA and openCL. While skyline computations are compute-intense, a principled study for GPGPU does not exist.

In this paper we present methods for computing the skyline efficiently on the GPU, and thus reduce the time needed to compute the skyline. Our contributions include:

1. an analysis of skyline computation tasks that are suitable for GPU parallelization
2. a novel strategy for managing data transfer and memory for skylines using CPU and GPU
3. a new sort-based skyline algorithm taking this strategy
4. a thorough experimental comparison with state-of-the-art sequential sorting based skyline algorithms.

2. RELATED WORK

Since the skyline operator was first introduced to the database community in 2001 [4] it has received a lot of research attention, and many different approaches to efficiently computing

the skyline have been proposed. These can be roughly divided into three categories, which we outline here.

The first category is based on divide-and-conquer. These algorithms recursively divide the data by some heuristic until the partitions have reached a certain size. The local skylines of the partitions are computed, and then merged to produce the final skyline. This type of algorithms has been implemented on distributed systems [8,14,15] and multicore CPUs [12]. While it might seem a good candidate for translation to the GPU, there are some issues. First, computing the local skylines in parallel is difficult without resorting to the brute-force solution of comparing all data points to all others. Secondly, a lot of thread management is needed in order to fully utilize the GPU when merging the local skylines, whose size typically varies a lot. Also, it is not trivial to merge two local skylines without comparing all data points from one skyline to all data points in the other.

The second category makes use of indexing structures. These algorithms typically utilize a hierarchical index of all data points, either directly in e.g. an R-tree [11] or by indexing the data in a B-tree according to some scoring function [9]. The advantage of these approaches is that once the index has been built, the skyline can be computed without accessing the entire dataset. Pruning is possible whenever subtrees are dominated. To their disadvantage they require that an index can be built and maintained, which is difficult on the GPU. Moreover, the pruning power deteriorates with the dimensionality since multidimensional indexing structures become less effective with higher dimensionality.

The third category is based on sorting. The idea, which was first introduced in [6], is that if data is sorted according to a monotone scoring function, then many dominance tests can be avoided. This is due to the fact that, in the sorted order, a data point can only be pruned by its predecessor, not its successor. These algorithms thus presort the data, and then maintain a window of skyline points. The data is scanned exactly once, with each data point being compared to the current skyline points in the window, and added if it is not pruned [6]. It was shown that if the monotone function is chosen carefully, then a full scan may not be needed after presorting, since at some point in the scan, none of the remaining points can be a skyline point [3]. A problem with sort-based algorithms is that they have to sort the data, which can be expensive. However, when it is not feasible to maintain an index and a distributed network of computers is not available, these methods are efficient.

To the best of our knowledge, only one work on GPU skyline algorithms is accessible. This approach translates the block nested loop approach to the GPU [4]. Basically, the GPU-parallel skyline algorithm (GNL) runs nested parallel loops, which compare all data points to all other data points [5]. Thus it is a simple parallel implementation of the brute force algorithm. We demonstrate in our experimental evaluation that our approach outperforms GNL.

3. BACKGROUND

In this section we define the skyline, review properties relevant for its computation, and provide background on using the GPU as a coprocessor.

3.1 Skylines

The skyline consists of all points for which no better point exists in the database as formalized in the following:

DEFINITION 1. ***Skyline***
*Given a set of d-dimensional points D, the **skyline** is the subset of all points that are not dominated:*

$$S = \{p \in D | \nexists q \in D \text{ such that } q \prec p\}$$

*A point p **dominates** q, written $p \prec q$, iff*

$$\forall i = 1, \ldots, |D|\ p_i \leq q_j \wedge \exists i \in \{1, \ldots, |D|\}\ p_i < q_j$$

*p, q are **incomparable**, written $p \prec\succ q$, iff*

$$p \not\prec q \wedge q \not\prec p$$

Thus, as illustrated in Figure 1, the skyline consists of points where no point exists that is at least as good in all attributes and strictly better in at least one attribute.

3.2 GPUs as coprocessors

In GPGPU computing the GPU is used as a so-called coprocessor. That is, the host (CPU) transfers data to the memory of the device (GPU) and then instructs the device to execute a special kind of function known as a kernel in a specified set of parallel threads. These threads have ids to identify themselves in relation to other threads, as well as offsets when working on data. Unlike for the CPU, these threads are lightweight software threads rather than hardware threads. The device is thus capable of efficiently executing an order of magnitude more threads than it has cores, which is used to fully utilize the hardware.

To obtain maximal throughput two major factors should be addressed: branch divergence and memory management.

3.2.1 Branch divergence

Due to the hardware design of GPUs threads are executed step-locked in groups of 32 called warps. This means that the same instruction is executed for each thread in the same clock cycle. This assumes that the threads are all set to execute the same instruction. If this is not the case, then the warp is split into as many groups as there are different instructions to execute. These groups then execute sequentially one by one. This is known as branch divergence. As this limits parallelism it can seriously hurt performance, especially if many steps are performed before the warp converges to the same instruction stream again. Therefore branching within warps should be avoided whenever possible.

3.2.2 Memory management

For the memory management, three major issues must be addressed. First the transfer of data to and from the device. Although memory of both device and host is fast, the PCIe bus connection is a bottleneck and one should therefore aim to minimize traffic between host and device. Secondly the latency for reading data from the global memory on the device is relatively slow at 200-400 clock cycles [1]. Special on chip memory, called shared memory, can be accessed by groups of threads known as blocks. This can be used to manually cache data close to the threads, but there is only 48KB per 192 GPU-cores so it is a scarce resource. Nevertheless the shared memory should be utilized whenever possible. Thirdly, the data is fetched from global memory at 32 or 128 bytes at a time, depending on the global memory cache configuration [1]. So care should be taken to make sure that data is read coalesced whenever possible to maximize usage of the bytes transferred. That is, when fetching data from

global memory, the data should be fetched in the same order as it is stored in the memory.

4. SKYLINE COMPUTATION ON THE GPU

In this section we present the main challenges of computing the skyline on the GPU, our solutions to those challenges, as well as our sort-based parallel skyline algorithm.

4.1 Sharing work between CPU and GPU

Based on the observations in the previous section, it is of prime importance in designing an efficient GPGPU algorithm to share the work between CPU and GPU in a manner that makes use of their respective capabilities, while limiting the amount of data that needs to be transferred.

We therefore propose to use the CPU to keep track of what data should be processed and the result set, while the GPU processes batches of a fixed size. In our method, the CPU keeps track of filtered and skyline points as well as what data is to be processed next which is difficult and ineffective to do on the GPU. Meanwhile the GPU does the actual dominance tests which are identical operations for all points and would be expensive if performed on the CPU.

In the following, we introduce a new method for branch free dominance tests, and explain which principles are exploited in our skyline algorithm, before detailing the effect of different batch sizes.

4.2 Branch free dominance test

Dominance tests are the most frequently executed operations in any skyline algorithm. It is therefore important to perform efficient dominance tests on the device. Since branch divergence reduces the efficiency, we need to minimize the degree of branching when comparing data points. To this end we have developed an optimized branch free algorithm. The core idea when comparing two points p and q is to avoid branching by maintaining two boolean variables indicating whether $p \prec q$ or $q \prec p$, respectively. The dimensions are processed sequentially and the variables are updated accordingly. The algorithm's pseudo code is given in Algorithm 1, where lines 5-6 contain the update rule. If $p \prec q$ then p must be better in at least one dimension and q can never be better in any dimension, which is exactly what we capture with the two boolean variables. To efficiently compute a domination relationship we would normally use `if` statements. However, this would result in branching, which as said decreases the performance of GPU computation. Therefore, we use this branch-free function.

Algorithm 1: BranchFreeDominance(p,q)

```
1 begin
2     bool p_better ← false;
3     bool q_better ← false;
4     for i ∈ d do
5         p_better ← p_i < q_i ∨ p_better;
6         q_better ← q_i > q_i ∨ q_better;
7     return ¬q_better ∧ p_better ;          // p ≺ q ?
```

4.3 GPGPU skyline algorithm

As discussed above, it is of crucial importance for an efficient GPGPU skyline algorithm that branching is limited. It is therefore necessary to devise a processing order that is data independent, i.e., determining whether a point is a skyline element or not should be independent of the test for another point. Most existing skyline algorithms are inherently data dependent and are therefore not efficiently executable on the GPU. This is because they maintain a temporary skyline set, against which they compare each data point. Differently, in our approach we want to determine if a point has been pruned without any knowledge of whether other points belong to the skyline or not. Our work is based on the observation that underlying principles in sorting-based skyline algorithms may be re-used [3, 6]. The main idea in sorting based approaches, as described in Section 2, is to run dominance tests in the order defined by some monotone scoring function. Based on the fact that a data point cannot be dominated by a successor in the sorted order (proven e.g. in [6]), data points may be pruned from dominance testing.

This main idea is exploited also in our GPGPU skyline algorithm (GGS) 2. In lines 1-3 the skyline set is initialized and the dataset is transferred to the device. It is then sorted with respect to the Manhattan norm of the data points ($||p|| = \sum_{i=1,...|D|} |p_i|$) in line 4. Next, in lines 6-12 we compare each point in D to the first α points of D. Since D is sorted according to the Manhattan norm and there is a thread per point we only need to do dominance tests one way (i.e., per point p, check if $p \prec q$). If a data point is dominated, the corresponding thread adds it to the set of dominated points and returns. Once the parallel execution is done, the host removes the dominated points from D in line 13. In lines 14-15 the host records the non-dominated points among the current batch of α points in the skyline result set and removes them from the dataset. Please note that these points do not contain false positives, since they have been compared in Manhattan norm order to all points that might dominate them. We iterate over the continuously smaller data set, until all data have been processed. Since each batch removes at least α points from the data set, (efficient) termination is ensured.

As we can see from this description, data dependence issues have been eliminated, since the dominance tests are performed on a per point basis, without any dependence on the other results. The CPU takes care of filtering (i.e., removing processed dominated points) and assigning the next data to be processed which are the steps that involve branching, whereas the GPU takes care of the highly parallelized branch free dominance tests. In this manner, we make best use of the respective strengths of CPU and GPU.

As any sorting-based skyline algorithm, the efficiency depends on the sorting function. Existing work [3] found that Manhattan norm performed best among several different scoring functions with respect to their pruning power of non-skyline points. We have therefore chosen this as the sorting function in our experiments as well, but in principle any monotone sorting function may be used in our algorithm.

4.4 Setting the batch size

The parameter α controls the batch size, i.e., how many data points each point is compared to on the device in each iteration, before the host takes over and filters the dominated points from the dataset. This affects the performance in two respects. First, each iteration reduces the remaining data set, which in turn accelerates the parallel performance since less threads need to be started. On the other hand,

Algorithm 2: GGS(data set D, batch size α)

```
1  begin
2      Sky ← ∅;
3      transfer D to device memory;
4      sort D w.r.t the Manhattan norm in parallel;
5      while D ≠ ∅ do
6          D' ← {p_i ∈ D | i ∈ 0...α};
7          D_dom ← ∅;
8          for ∀p ∈ D do in parallel
9              for q_i ∈ {D | i ∈ 0...α} do
10                 if BranchFreeDominance(q_i,p) then
11                     D_dom ← p ;        // p is dominated
12         D ← D \ D_dom;
13         Sky ← Sky ∪ (D' \ D_dom);
14         D ← D \ D';
15     D ← Sky ;                          // skyline of D
```

there is overhead in terms of data transfer to and from the device, and in terms of initializing threads associated with each iteration. Secondly, the performance is affected differently for different data distributions. For correlated data, the skyline tends to be small. Therefore, a small batch size can prune many non-skyline points early. For anti-correlated data which tends to produce a large skyline, a large batch size is preferred. As many points are skyline points, we expect to prune fewer in each iteration. In order to maintain a favourable relationship between the overhead per iteration and the number of points that can be pruned per iteration, the batch size should thus be larger.

Preliminary experiments indicate that the largest effect of a small batch size α on correlated data is in the first iteration, and that a small batch size in the first iteration on anti-correlated data does not yield substantial differences in execution time. Thus, we have optimized the algorithm towards both types of distributions by comparing data points against $\frac{\alpha}{4}$ points in the first iteration, and against α in all the remaining iterations. Please notice that α does not need to be modified with respect to the size of the input data, and should thus not be considered as a parameter that users should explicitly provide and tune.

4.5 Sorting data in parallel

In order to fully utilize the GPU, we sort the data in parallel. This is done by first launching a thread for each point. These threads record their index into one array, compute the manhattan norm of their point and record it into another array. The array containing the indexes is then sorted in parallel with respect to the manhattan norm array, using the sorting function from the thrust library [2]. The sorted array is then used to reorganize the data on the device.

4.6 Memory management

As mentioned, the key factors for an efficient GPU algorithm are memory management and avoiding branch divergence. We have taken care of the latter by introducing a branch free dominance check, which is utilized in an algorithm that only branches when the thread is done. We will now discuss our optimizations to the memory management that were identified in Section 3.2 as important in a CPU / GPGPU setup.

As discussed earlier, the main bottleneck in data transfer is the PCIe bus. We reduce the amount of data to be transferred between host and device by only transferring indexes and not actual data points. More specifically, the set D_{dom} in algorithm 2 only contains data point identifiers. The skyline is then built on the host, by utilizing the fact that a copy of the data remains in the host memory.

In every iteration of Algorithm 2 the data becomes fragmented, since some data points are pruned. To ensure that the data can be read in coalesced manner, we do a parallel order preserving de-fragmentation of the data after every iteration. The de-fragmentation is done in two steps. First, while the CPU is filtering the dataset for the pruned points, it records where non-pruned data points are located in the data array on the device. This information is then transferred to the device, and the data is moved from the original positions to the beginning of the array in batches of α. This is possible since the first α points have either been pruned, or recorded as skyline points after each iteration. It is worth noting that only the array with data indices is transfered from and to the device, and not the data, preventing a potential bottleneck. The de-fragmented array allows reading of the data in a coalesced manner, and thus a full utilization of the memory bandwidth.

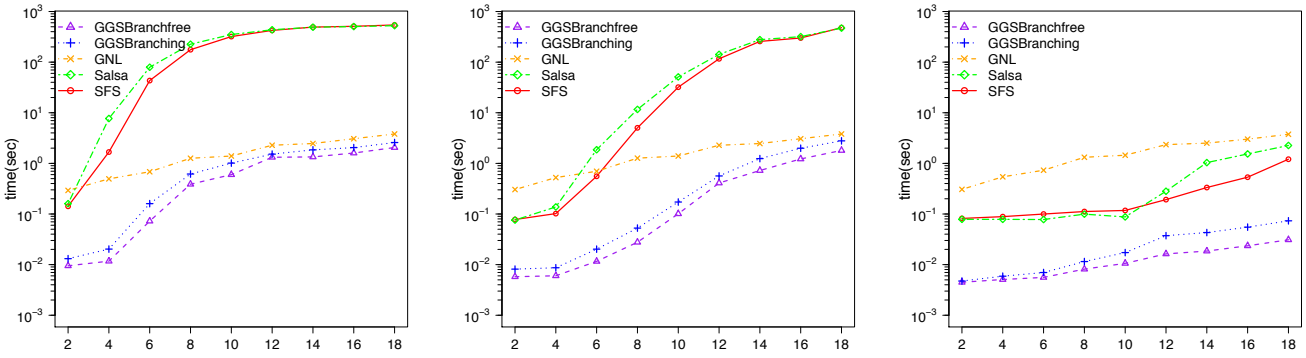
As mentioned, threads are divided into blocks that can access a very fast shared memory. Since each data point is compared to α other points in each iteration, it is repeatedly accessed. To minimize fetches from the global memory, the threads of a block load the points they are responsible for into shared memory at the beginning of each iteration. To further reduce fetches from global memory, each of the α data points are also loaded into the shared memory before dominance test is performed. This allows the threads to do dominance tests on data that are entirely contained in the low-latency shared memory, rather than having to load the data from the high-latency global memory. Further more, since all threads in a block have access to the same shared memory, each of the α datapoints only needs to be fetched from the global memory once for each block.

5. EXPERIMENTS

All experiments were executed on an Intel i7-2600 3.4Ghz quad core processor with 8GB 1333Mhz ram, and NVIDIA gtx670 graphics card with 4GB of ram. The system ran Ubuntu 12.04, with CUDA driver and CUDA runtime system in version 5.0. The parallel algorithm was implemented in the CUDA-C programming language, both with and without the branch free dominance test algorithm. All GPU experiments was executed with global memory caching enabled, thus making the load granularity 128 bytes.

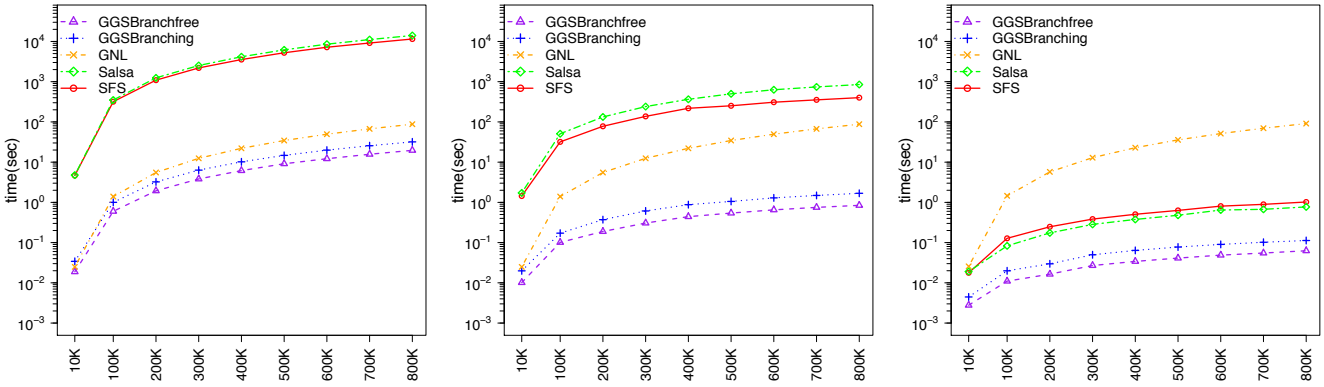
The intent of our work is to produce results fast enough to be used in an interactive environment, where the data may not be known before query time, and the user may modify it on the fly before the skyline computation is executed. Index based solutions do not perform well in such settings since the index needs to be built before the skyline can be calculated, for each query. Also, as dimensionality increases, indexes deteriorate. Therefore we have chosen to compare our sort-based GPGPU solution to index-free sequential algorithms in addition to the existing GPGPU method.

In summary, in order to evaluate our solution we implemented the state-of-the-art sort-based algorithm SaLSa [3] as well as the original sort-based algorithm SFS [6] in C++



(a) Anticorrelated (b) Uniform Independent (c) Correlated

Figure 2: Varying dimensionality with n=100K



(a) Anticorrelated (b) Uniform Independent (c) Correlated

Figure 3: Varying cardinality with d=10

(both being pure CPU, simple-threaded algorithms), and we also implemented the GPU parallel algorithm GNL [5] in CUDA-C, with branch free dominance test. For both sort-based approaches we implemented the same sorting functions identified as the best choices in the original papers. For SaLSa, this function sorts the data on the minimum attribute value, and uses the sum of attributes to solve ties. For SFS, we implemented the sorting function based on entropy. We used the sum of coordinates (Manhattan norm) for sorting the data for our GGS approach as discussed before, and the data was sorted in parallel on the device for the parallel algorithms. Notice that sum was also considered in [3]. We have chosen to implement GGS with both our new branching free dominance test algorithm, and the normal comparison minimizing dominance test, to show the difference between the two strategies. For the choice of batch size α we conducted extensive empirical studies, and concluded that 4000 is the best trade-off value for this setup.

As is the de-facto standard in skyline research, we test the algorithms on synthetic data generated with correlated, anticorrelated and uniform distributions, as well as real data on NBA player statistics.[2] For all datasets we assume that they fit in both the host and device memory, so that SaLSa and SFS only do a single pass over the data, and the parallel algorithms have access to all data on the device. All reported runtimes are an average over 10 runs and include transferring data to and from the device for GGS and GNL, as well as sorting for all sort based algorithms. Timing is started just after the data have been loaded into the host memory.

[2]http://www.basketballreference.com/

5.1 Synthetic data

The first experiment studies how the algorithms respond to different dimensionality of the data, since high-dimensional data is known to increase the complexity, due to the added dimensions in dominance test as well as increasing the skyline size. Figure 2 shows the results of this experiment for correlated, anti-correlated and uniform synthetic data with 100,000 data points. As the figure shows our algorithm scales orders of magnitude better in terms of dimensionality. For the anti-correlated data, both our algorithm and GNL are much faster than the sequential algorithms. However, since a large percentage of the data becomes part of the skyline for high-dimensional anti-correlated data, we end up being only slightly better than the brute force GNL. On

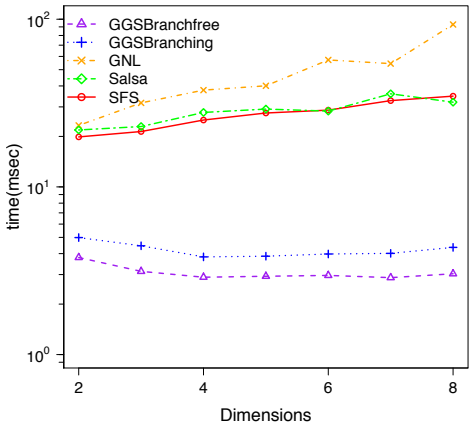


Figure 4: Varying dimensionality, NBA data

the other hand for correlated data, both the sequential and GGS are substantially faster than GNL, since all three algorithms prune parts of the data early, unlike the brute force GNL. Overall it is clear to see that GGS scales very well with dimensionality, regardless of distribution.

Next, we look at how the algorithms scale with respect to the cardinality of the data. Again we have the three data distributions, and we vary the cardinality from 10,000 to 800,000 10-dimensional data points. A similar pattern as for dimensionality emerges. For anti-correlated data GNL and GGS are again orders of magnitude faster, but relatively close to each other, even though our GGS is still a factor 4 faster. The reason that the difference is less pronounced is that large parts of the data are in the skyline, thus necessitating many dominance tests. We also see that the sequential algorithms once more outperform GNL for the correlated data, and that GGS in turn outperforms the sequential algorithms by more than a factor 12 for the largest dataset. For the uniformly distributed data we see a similar tendency, although the raw computing power of GNL allows it to beat the sequential algorithms. In general GGS outperformed the other algorithms in all cases. We have provided a speedup factor of more than 600 for SFS and more than 700 for SaLSa on anti-correlated data while also being consistently better on the correlated data, unlike GNL. Looking at our two implementations for dominance tests we see that the branch-free dominance test is faster by a more or less constant logarithmic factor.

6. REAL WORLD DATA

We evaluate performance also on real world data, using those 8 dimensions of NBA player statistics that have been recorded at all times. This dataset has been chosen for several reasons. First, it is the same data as in [6]. Secondly, it is relatively small with mostly correlated dimensions, which favors the sorting based sequential approaches. The data has been normalized and negated in order to maximize the dimensions which is more intuitive for this data. This preprocessing is not included in the runtimes. Please note that we report the times in milliseconds in Fig. 5.1.

Even though this data favors the sequential algorithms, our parallel algorithm still outperforms both of them by a factor between 6 and 11. GNL is again outperformed due to the correlated nature of the data. Thus, our algorithm is not only superior on synthetic, but also on real world data.

7. CONCLUSION

We have analyzed major challenges for efficient skyline computations on the GPU, and presented novel solutions to them. Our completely branch-free method for dominance tests between data points has been demonstrated to be consistently faster on the GPU than the traditional data point comparison. We have presented a novel algorithm for efficient skyline computations on the GPU and have shown that it consistently outperforms state-of-the-art sequential sorting based algorithms, as well as the existing GPU parallel skyline. Thorough experiments on both synthetic and real data show that GGS is a good choice for on-the-fly skyline computations, regardless of data distribution, dimensionality and size.

8. ACKNOWLEDGEMENT

This work has been supported in part by the Danish Council for Strategic Research, grant 10-092316.

9. REFERENCES

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2012.

[2] Thrust parallel datastructure library. http://docs.nvidia.com/cuda/thrust/, 2012.

[3] I. Bartolini, P. Ciaccia, and M. Patella. Salsa: computing the skyline without scanning the whole sky. In *CIKM*, pages 405–414, 2006.

[4] S. Börzsöny, D. Kossmann, and K. Stocker. The Skyline operator. In *ICDE*, pages 421–430, 2001.

[5] W. Choi, L. Liu, and B. Yu. Multi-criteria decision making with skyline computation. In *IRI*, 2012.

[6] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting: Theory and optimizations. In *IIPWM*, pages 595–604, 2005.

[7] J. N. England. A system for interactive modeling of physical curved surface objects. pages 336–340, 1978.

[8] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD*, pages 85–94, 2011.

[9] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in z order. In *VLDB*, 2007.

[10] M. Magnani, I. Assent, and M. Mortensen. Anytime skyline query processing for interactive systems. In *DBRANK*, 2012.

[11] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

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

[13] W. Shooman. Parallel computing with vertical data. In *IRE-AIEE-ACM*, pages 111–115, 1960.

[14] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *SIGMOD*, 2008.

[15] P. Wu, C. Zhang, Y. Feng, B. Zhao, D. Agrawal, and A. El Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT*, pages 112–130. 2006.