B-line to the skyline. The fastest query in all the land.

This paper investigates skyline queries and how to optimise the speed in which these queries are resolved. This increase in efficiency is examined for both single core and multi core approaches on a CPU and GPU.

What is a skyline query?

To best describe the skyline query let us assume that each node represents a stock in the stock market. In this example the only relevant information about a stock is its risk and return.

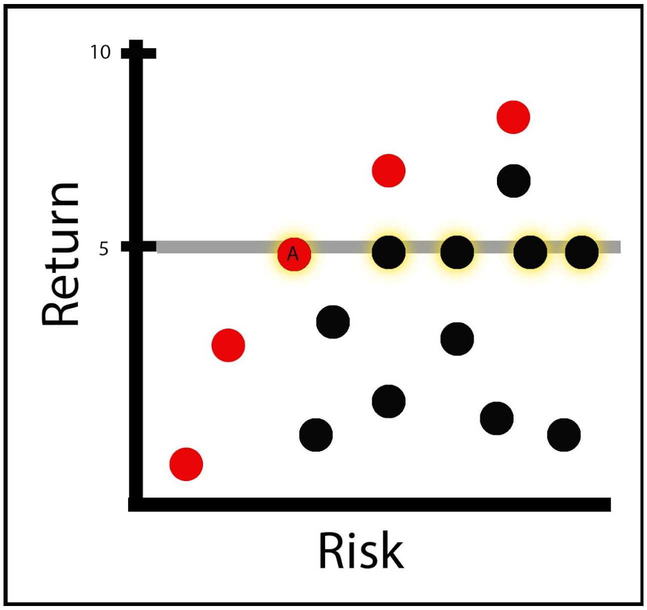
Given a set of nodes a skyline query will return nodes that are not dominated by any other nodes. For a node to dominate another node it has to be better on one dimension (risk), and better-than-or-equal-to on the other dimension (return). The returned nodes from a skyline query are known as Pareto set and/or Pareto frontier.

In the example on the right, all the red stocks would be returned by a skyline query: the Pareto set. In this example, the red nodes are the only stocks that a trader should be interested in. Suppose one wanted a stock with a return of 5%. Figure 1. demonstrates there are five stocks that meet this requirement. Out of these five stocks *A* has the lowest risk. If anyone were to buy a highlighted stock other than *A*, they would be assuming extra risk for no extra return. Any stock that gives the optimal return compared to risk will be in the Pareto set.

Why are Pareto sets useful and where are they used?

A skyline query has the ability to drastically reduce the set of data for any problem without removing any optimal solutions. This ability can be used to reduce the size of databases. It can also help with multicriteria decision-making applications. If each node represents a possible choice a decision-making algorithm needs to analyse, then reducing the number of nodes will drastically speed up the algorithm. This speed up is especially important for real-time decision making algorithms, which are always pressed for time. As a skyline query gets faster and more efficient it also becomes more versatile.

Figure 1. Risk versus return, where the desirable stocks have high return and low risk:



Hardware used:

All solutions proposed in this paper were benchmarked using an Azure instance. The Azure instance has 6 CPUs with 32K of L1d and L1i cache, 256K of L2 cache and 35840K of L3 cache. It also has an NVIDIA Tesla V100 GPU with 16 GB of memory.

Single Core

All algorithms take a collection of nodes as input. Each node has a x , y and payload value. X and y are initially stored as ints, and payload is stored as a string of length four. For certain applications nodes may have information that is not used by the skyline query. To represent this case the payload was chosen to be approximately the size of a pointer, this pointer can reference all non essential information regarding particular nodes.

Pseudo Code:

```
Node Best [ ]
Update = True
While( Update )
    For( i < input.size i++)
        Update = False
        Add = True
// If nothing dominates the input node add it to best
        For(k< Best.size k++)
            If( Best[k] Dominates input[ i ]
            OR equal)
                Add = False
            if( Add )
                Best.add( input [ i ] )
                Update = True
// Remove anything the input node dominates
                For( j = 0 ; j < Best.size ; j++)
                        if( input[ i ] Dominates Best[ j ] )
                                    Remove Best [ j ]
Return Best
```

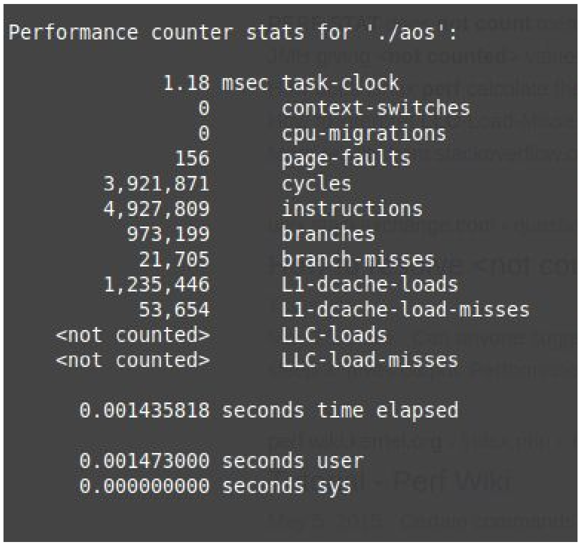
Baseline algorithm

This creates a set of undominated nodes, this set is referred to as *best*. The algorithm loops through all of the input nodes. If the current node is not dominated by any nodes in *best* it is then added. Once a node is added to *best* it is compared to all other *best* nodes. If it dominates any other node the dominated node is removed. The algorithm returns best and finishes once it has iterated through the input without adding a new node to *best*.

The Bottleneck

Perf is a performance analysis tool in Linux that gives the user access to various cpu counters. The following shows perf results for the three solutions using an identical dataset of 1000 nodes that was included as a *.hpp* file at compile time.

Figure 2: Perf statistic running the baseline algorithm



```
Performance counter stats for './aos':

            1.18 msec task-clock
               0      context-switches
               0      cpu-migrations
             156      page-faults
       3,921,871      cycles
       4,927,809      instructions
         973,199      branches
          21,705      branch-misses
       1,235,446      L1-dcache-loads
          53,654      L1-dcache-load-misses
   <not counted>      LLC-loads
   <not counted>      LLC-load-misses

     0.001435818 seconds time elapsed

     0.001473000 seconds user
     0.000000000 seconds sys
```

The ultimate goal when looking to improve a solution is to decrease the number of cycles and instructions required. Over a 5th of all instructions are *L1-dcache-loads* as shown in Figure 2. The bottleneck is memory access, currently the CPU spends a significant amount of its time loading data to L1. The next section will focus on reducing the amount of working data to overcome this bottleneck.

Baseline algorithm optimisation using Structure of Array (SoA)

The Array of Structures (AoS) used up until this point contains the following :

Integer x
Integer y
String payload

When computing the skyline solutions, only x and y values are used. The payload of the nodes is not needed for any computation, however in the original AoS approach all three values would need to be loaded into cache. By switching to an SoA, hot and cold data can be separated. This separation reduces the amount of data pulled into cache. The SoA Node structure contains:

Array x[ ] // x coordinates
Array y[ ] // y coordinates
String name[ ] // name of data

Unlike the AoS which will have many individual objects the SoA will have separate arrays for each x, y, and payload.

Figure 4: Structure of arrays

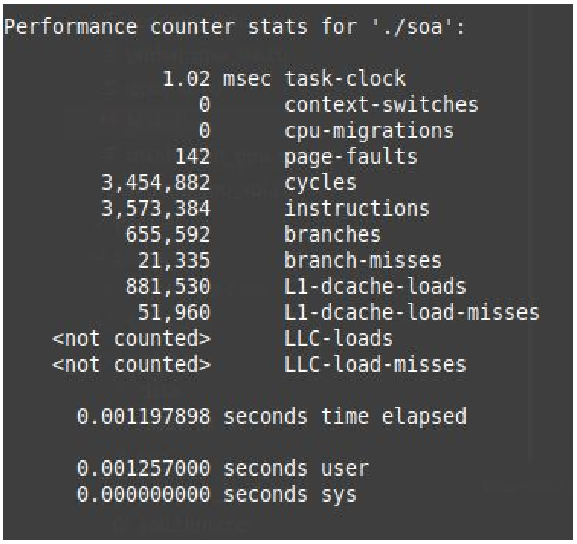
X1	X2	X3	...	Xn

Y1	Y2	Y3	...	Yn

Payload 1	Payload2	...	Payload n

Figure 3 has 880,000 L1 cache loads, this is a significant decrease compared to the 1,230,000 cache loads in Figure 2. This decrease heavily reduced the number of instructions, cycles, and time required to complete the skyline query. At the end of this section there is a graph (Graph 1) that shows the speed up obtained here.

Figure 3: Perf statistic running our SoA



```
Performance counter stats for './soa':

             1.02 msec task-clock
                0      context-switches
                0      cpu-migrations
              142      page-faults
        3,454,882      cycles
        3,573,384      instructions
          655,592      branches
           21,335      branch-misses
          881,530      L1-dcache-loads
           51,960      L1-dcache-load-misses
    <not counted>      LLC-loads
    <not counted>      LLC-load-misses

      0.001197898 seconds time elapsed

      0.001257000 seconds user
      0.000000000 seconds sys
```

This was then later improved when the Node structure was modified once again.

Figure 5: SoA XY

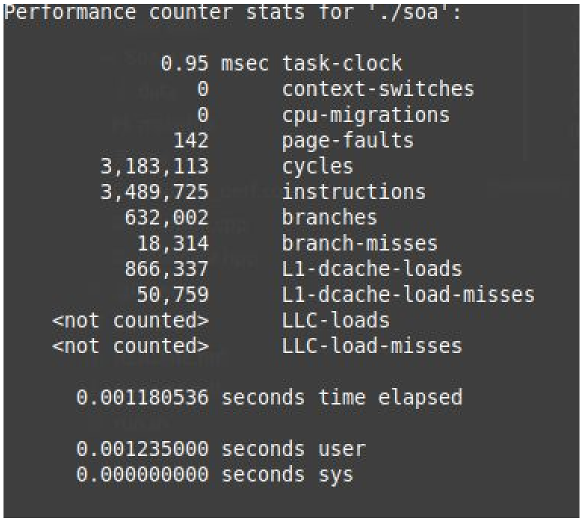
X1,Y1	X2,Y2	X3,Y3	...	Xn,Yn

Payload 1	Payload2	...	Payload n

Algorithm clean up

Grouping XY values showed a small speed up but the next big improvement came when the algorithm itself was simplified. The improved algorithm works in a similar way to the baseline algorithm. The main difference is when a node is compared to *best* and is dominated the inner for-loop is broken. This reduces the number of overall instructions. The perf image below (Figure 4) shows a decrease of 100,000 instructions compared to the previous image of perf (Figure 3). The difference in number of instructions increases exponentially as time increases, demonstrated in table 1 and graph 1.

Figure 4: Perf statistic running our SoA XY



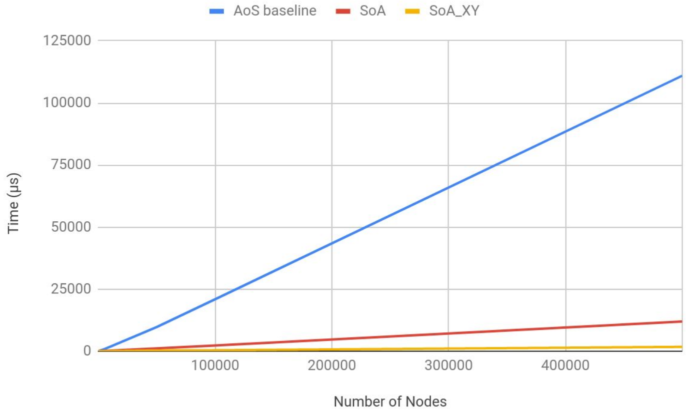
```
Performance counter stats for './soa':

          0.95 msec task-clock
             0      context-switches
             0      cpu-migrations
           142      page-faults
     3,183,113      cycles
     3,489,725      instructions
       632,002      branches
        18,314      branch-misses
       866,337      L1-dcache-loads
        50,759      L1-dcache-load-misses
   <not counted>      LLC-loads
   <not counted>      LLC-load-misses

   0.001180536 seconds time elapsed

   0.001235000 seconds user
   0.000000000 seconds sys
```

Table 1: Benchmarking data for single core comparison

Number of Nodes	AoS baseline (us)	SoA(us)	SoA XY grouped (us)
100	18	10	2
500	70	20	6
1000	158	32	10
5000	793	127	29
10000	1755	228	50
50000	9717	1147	304
100000	20913	2275	376
500000	110968	11978	1741

The table above shows the benchmarking results for the three solutions. With an increase in the number of nodes, the relative speedup also increases. With 100 nodes, the relative speedups between the AoS solution and SoA is 1.8, between basic SoA and XY group a 5 times speedup is achieved. As the number of nodes increases to 500000, the relative speedups increase to 9.3 and 6.9. A graph is shown below illustrates these speedups. When the number of nodes increases the gap between them increases as well.

Graph 1: Comparison of single core solutions



Multi core

Parallel Algorithm Description

The multicore solution is very similar to the single core SoA XY solution. First, threads are spawned, one for each core, these threads are assigned an equal section of input node. Each thread will loop over its input nodes and try to add them to *best*. In this solution *best* is stored in shared memory, this memory can be accessed by any core or thread. Race condition can occur when multiple threads try to read/write to the same location in shared memory.

Areas that have these race conditions are labeled ”Critical section” in bold.

Pseudo Code:

Algorithm description:

```
Node best [ ]
Spawn threads
For( i < input.size i++)
   Bool Written = false
   Bool Dominated = false
   For( k < best.size k++)

         if( input.xy[i] dominates best.xy[k])
              Critical section
              best.add( input[i] )
              Written = true

         Else if( best.xy[k] dominates input.xy[i] OR is equal)
              Dominated = true
              Break

   If (!Dominated AND !Written)
           Critical section
           best.pushback( input[i] )
```

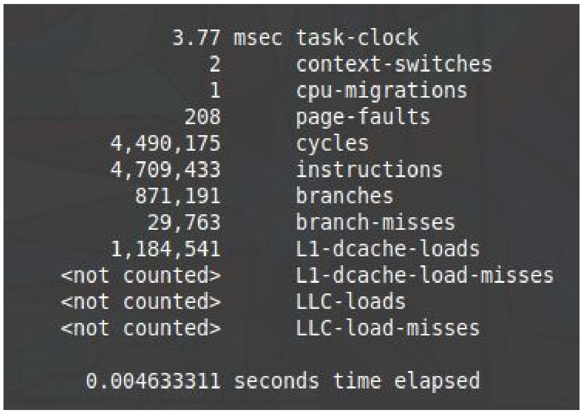
How are race conditions resolved

The race conditions occur when threads are accessing and writing to the same shared index of *best*. To handle this, critical sections are placed in the code. A critical section can only be accessed by one thread at a time. If a thread reaches a critical section it will wait/stall until the critical section is available. Once a thread enters a critical section it will check that the previous “if” statement still holds. The thread will then either exit and release control of the critical section or write to *best.*

Work efficiency compared to the single core case

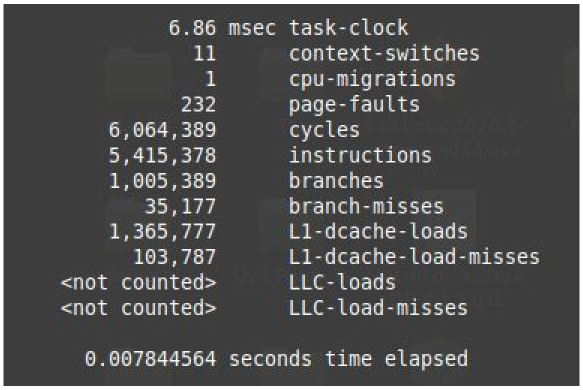
Using 1 core and the same input, the parallel solution is 15% less work efficient then the optimized single core solution, SoA XY. The 15% comes from the cycle counts found in figures 5 and 6.

Figure 5: Perf SoA XY



```
          3.77 msec task-clock
             2      context-switches
             1      cpu-migrations
           208      page-faults
     4,490,175      cycles
     4,709,433      instructions
       871,191      branches
        29,763      branch-misses
     1,184,541      L1-dcache-loads
 <not counted>      L1-dcache-load-misses
 <not counted>      LLC-loads
 <not counted>      LLC-load-misses

   0.004633311 seconds time elapsed
```

Figure 6: Perf parallel



```
          6.86 msec task-clock
            11      context-switches
             1      cpu-migrations
           232      page-faults
     6,064,389      cycles
     5,415,378      instructions
     1,005,389      branches
        35,177      branch-misses
     1,365,777      L1-dcache-loads
       103,787      L1-dcache-load-misses
 <not counted>      LLC-loads
 <not counted>      LLC-load-misses

   0.007844564 seconds time elapsed
```

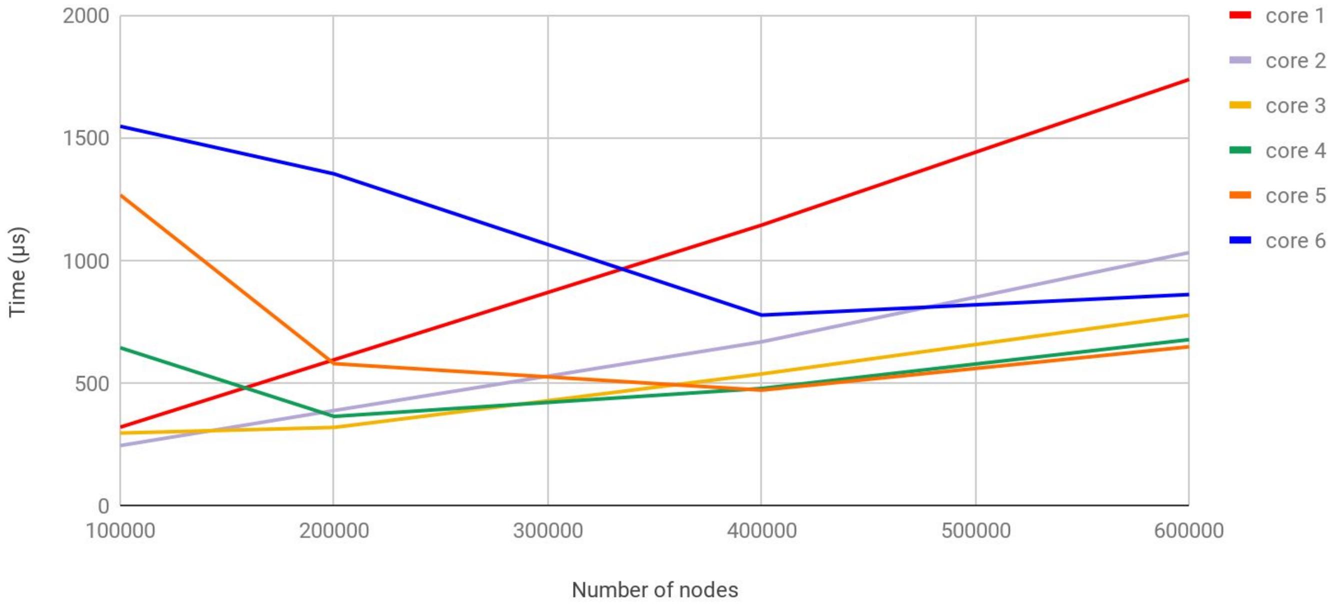
Scalability

The table below represents how the performance is affected based on the number of cores and number of nodes.

Table 2: Core comparisons

Number of threads (t)	10^3	10^4	10^5	2 * 10^5	4 * 10^5	6 * 10^5
1 core	20	60	319	594	1143	1737
2 core	157	167	244	387	667	1031
3 core	470	605	295	318	536	776
4 core	931	988	643	363	477	676
5 core	1236	1258	1266	578	470	647
6 core	1583	1514	1546	1352	776	860

Graph 2: Solution time compared to input size for multiple cores.



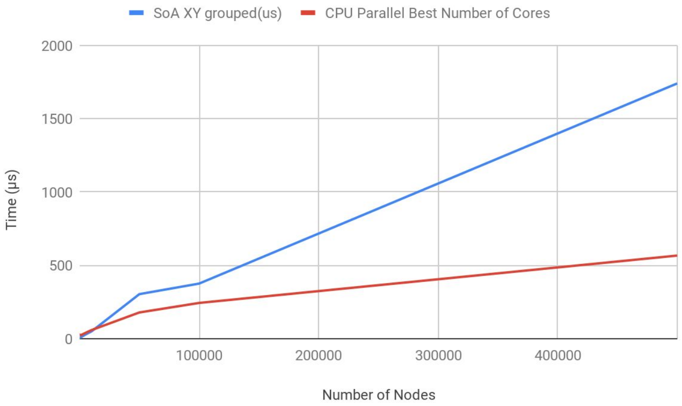
Graph 2 and table 2 show that the optimal number of cores increases with the number of input nodes, this may be because of the critical sections. The number times a thread will wait for access to a critical section is based on three main factors:

1. Amount of data
2. Distribution of data
3. Number of cores/threads

The data distribution was not changed in the previous results. Looking at stall counts would be valuable when testing this hypothesis.

The 6 cores solution slope is the lowest when looking at the rightmost section of graph 2. As input size increases the time needed to solve said input will also increase. The 6 core solutions time grows at the slowest rate. Once the input reaches a certain size, all cores less than 6 will be outperformed. Past this critical point 6 cores will increasingly outperform lower core solutions.

Graph 3: Single Core vs Multicore



Graph 3 shows how the parallel solution outperforms the optimized single core solution. At 200000 nodes the parallel solution had a 2 times speed up compared with the single core solution. At 500000 Nodes this speed up increases to 3. As the input size increases, the performance gap will also linearly increase thus allowing for more than 3 times speedup.

The GPGPU

GPGPU stands for general purpose graphical processing unit. The results of this paper are collected using cuda on a NVIDIA Tesla V100 with 16GB of memory.

GPGPU basic algorithm description

First the algorithm spawns a thread for each node. A thread will travel along the input comparing input nodes to itself, whenever it finds a input node that dominates itself it will set a flag and return. The flag shows that the node assigned to that particular thread is not part of the solution. When a flag is set the thread will return. If a thread is not dominated it will continue comparing its node to the input. Skyline nodes will be the only ones left unmarked. These unmarked nodes will be printed by *Namemaker*. *Namemaker* simply spawns a thread for each node and if that node's flag has not been set the thread will print out its name.

Pseudo Code:

```
N = numbeer_of_nodes
Global memory  XY de_xy[N]

Solv(FlagArray[ ])
   Index = thredID + BlockID
   for(i =0 , i<N, i++)
           if( de_data[ i] Dominates de_xy[ index ])
                     FlagArray[index] = set
                     return
Namemaker(FlagArray[ ], Names[ ])
   Index = thredID + BlockID
   if(FlagArray[Index] Not set)
           Out-put ( Name[index])
Main()
   Copy data from CPU to GPU
   Solv<< num block, num_threads >>(FlagArray)
   Synchronise (wait for all threads to finish)
   Namemaker<< num block, num_threads>>(FlagArray, Names)
    Synchronize
    Free memory
```

There are no race conditions in this code. A thread only writes to its unique *index* of the FlagArray and this location of data is not assessed again until Namemaker is called. The GPGPU solution is completely different from the multi core solution discussed in this paper.   Unlike the multi core solution GPGPU solution has no critical sections and does not compare nodes to *best*.

Global or constant memory

Like the CPU the GPU has a data hierarchical storage system, meaning that memory access times vary depending on where said data is stored. Data access patterns are critical in effectively using any GPGPU. There are two types of memory that can be accessed by any thread, global and constant. Global memory is initialized using __device__ and does not need to be static. Constant memory __constant__ is unchanging but has faster access time compared with global memory [x]. Although Constant memory is faster it is much smaller than Global memory. Global memory can hold 500000 nodes and Constant memory can hold 5000.

Branch free domination

Branching tends to be harmful on GPUs' performance, when a branch occurs multiple threads must wait. Up until this point the *Dominate* function was not branch free. A branch free *Dominate* is discussed in a paper called Efficient GPU-based skyline computation[2]. When the two *Dominate* functions were compared on the GPU, there was little difference in the overall performance. From this point on all GPU solutions in this paper will use the branch free *Dominate* function.

Get names and printing

After generating the Pareto set, each solution prints the payloads of each node from that set. As mentioned above, the GPU solutions keep track of the nodes that are dominated by setting a flag. The size of the flag array is equal to the size of the dataset. Each index of the flag array must be checked. This can be done in the host (CPU) or in the GPU (namemaker). The CPU solution is to simply loop over the flag array. Namemaker will spawn a thread for each node and if that node flag is not set, it will print the output. Table 3 shows performance comparison between the two. Namemaker tends to be slightly faster when run on large data sets and will be used for all GPU solutions in this paper.

Table 3: Namemaker vs Loop

Nodes	Main Func Using for loop	Namemaker GPU
10000	528	604
100000	9352	9274
500000	63344	62564

Shared memory

Reducing memory access time is the focus of this “Shared memory” solution. A thread is spawned for every node, these threads are grouped into even sized blocks. Each thread in a block copies a node from global memory to shared block memory, if this memory already has a value it is overwritten. Each block has block memory. Block memory is small and fast and can be accessed by any thread in said block. Once all the threads in a block have finished copying, each thread will compare its assigned node to the nodes in its block's shared memory. This process repeats until all nodes in global have been copied and compared in block memory. Like the other solutions discussed in the, a flag is set whenever a thread's assigned node is dominated, this thread will no longer compare its input to other nodes and will only copy from global to the block.

Pseudo Code:

```
N = numbeer_of_nodes
Global memory  XY de_xy[N]

Solv(FlagArray[ ])
   Index = thredID + BlockID
   Shared [Num thread per block]
   for(i =0 , i<Num blocks, i++)
           Copy from global
           Sync threads
           for(k=0, k<Num thread per block)
                     if( Shared[k] Dom My_node)
                               FlagArray[index] = set
                               Sync threads
                               for()
                                         Sync
                                         Copy from
                               global
                                         sync
                               return
           Sync threads

Namemaker(FlagArray[ ], Names[ ])

Main()
   Copy data from CPU to GPU
   Solv<< num block, num_threads >>(FlagArray)
   Synchronise (wait for all threads to finish)
   Namemaker<< num block,
num_threads>>(FlagArray, Names)
    Synchronize
    Free memory
```

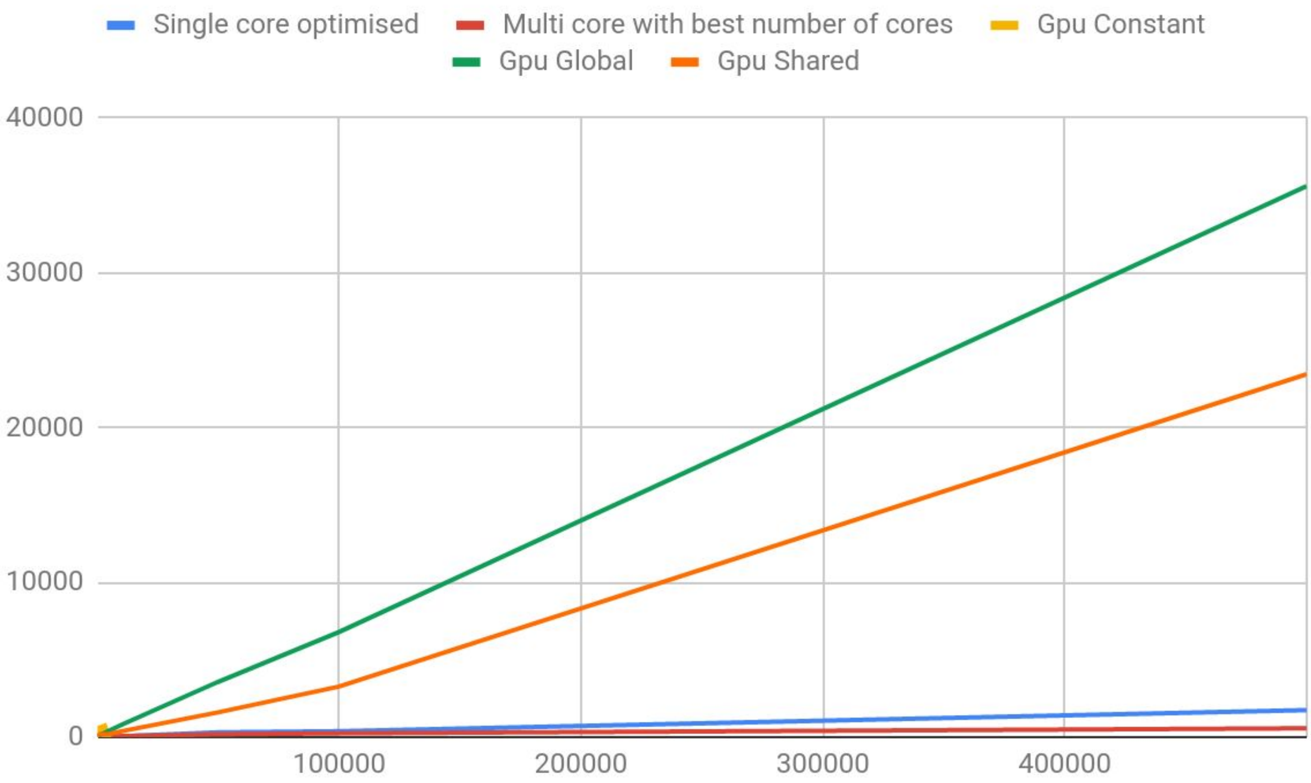
Table 4: Final benchmarking results

Nodes	Single core optimised	Multi core CPU	Gpu Con stant	Gpu Global	Gpu Shared
100	2	20	78	76	77
500	6	26	105	99	116
1000	10	24	137	127	183
5000	29	40	366	420	196
10000	50	59	NA	788	371
50000	304	179	NA	3570	1600
100000	376	244	NA	6791	3266
500000	1741	567	NA	35606	23453

Conclusion

When comparing the results above, it is clear that the CPU parallel solution outperforms all the other solutions. The GPU shared solution is 30 - 50% faster than the GPU global solution. This being said the GPU solutions perform significantly worse then the CPU solutions. In order to have a GPU solution that can outperform the CPU, the solution needs to be much more parallelizable. If more dimensions were used the GPU solutions might perform relatively better. There might be potential for speed up even with the ultra fast CPU solutions. Potential improvements are mentioned in the Further improvements section below.

Graph 4: Final comparison



Evidence of Correctness & Explanation on how benchmarking is done

The following section outlines the methodology used to benchmark the solutions and generate the tables and plots in this paper and ensuring that the results are accurate.

1. All the scripts have the same predefined default values.
2. Benchmarking is done by running the solutions 1000 times and then calculating the average time. This eliminates any bias and thus increases the accuracy. Note that the number of test instances to run can be passed in as a command line argument and thus can be increased if more accurate results are needed.
3. In order to generate accurate *perf* results, no data generator was used as that would affect the results. Instead a precompiled list of nodes was written in a *.hpp* file and was included into the scripts.
4. Validating outputs was done in 2 ways:
    a. Black Box - Scripts have a compare mode that allows the user to pass in a seed value. A single seed value will always generate the same nodes. When the same seed is passed to multiple solutions, the output of each solution should be the same.
    b. Predetermined Values - A data *data-sanity-check* file with known values and solutions is used for testing on each solution.

5. All benchmarking was done on the same hardware.

There is also a readme attached that explains in detail how to compile and run the codes so that anyone can reproduce these results.

Further improvements

Our team was focused on improving our ability to optimise code. Some of the results in this paper and papers referenced in this text, show that there are potentially more efficient algorithms to tackle Skyline queries. Sort based solutions showed a lot of promise in the initial tests of this paper.

Depending on dimension and data distribution results may vary. The results in this paper used only two dimensions and a normal distribution of node values, the Pareto sets frequently were very small. If more dimensions were used it would be interesting to see how the solutions performed, little improvements such as branch free *domination* might have a large impact. SIMD and SIMT registers show promise, especially when more dimensions are used.

Data access seemed to be the bottleneck of all the solutions mentioned in this paper. Tiling might be able fix this and allow for significant speed up. If this paper was to be revisited Tiling would be the first potential improvement explored.

References:

1. Tiakas, E., Papadopoulos, A. N., & Manolopoulos, Y. (n.d.). Skyline Queries: an Introduction. Retrieved April 22, 2020, from http://delab.csd.auth.gr/papers/IISA2015tpm.pdf

2. Bøgh, K. S., Magnani, M., & Assent, I. (2013, June). Efficient GPU-based skyline computation. Retrieved from https://dl.acm.org/doi/10.1145/2485278.2485283

3. Chester, S., Sidlauskas, D., Assent, I., & Bøgh, K. S. (2015). Scalable Parallelization of Skyline Computation for Multi-core Processors. In *31st IEEE International Conference on Data Engineering (ICDE 2015)* (pp. 1083 - 1094). IEEE Computer Society Press. https://doi.org/10.1109/ICDE.2015.7113358

Contributions

Ben and Akash are a solid team! Both spent time keeping each other motivated. C++ and CUDA programming was new for the members of this team. Peer coding and sharing resources to learn these languages was essential for the duo's success/improvement. The development of this paper was very much a team effort.

Ben Shapland focused on algorithm design. He explained and discussed design choices with his partner. Akash Charitar has a lot of technical knowledge, he created bash scripts and set-up/troubleshoot environment issues.

This paper was written by both members of the team. Ben did the copy editing while akash collected data. Both members of this team are extremely happy with their joint performance.

By Akash Charitar (V00875728)
and Ben Shapland(V00819775)
April 22, 2020

Link to Git repository

https://github.com/BenShapland/skyline_problem