GPGPU Programming Assignment 2014 ”The Asteroid Miner”

Harry Long

***Abstract*—The purpose of this research is to attempt to parallelize the ”Asteroid Miner Problem” in which a drone must find the most efficient path to go from its drop position back to its base station in the minimum steps whilst collecting the most minerals possible. In an attempt to do so we first examine the problem and outline its parallelism potential. We then outline the algorithms developed to tackle the problem and benchmark the speed ups gained from running it on the GPU rather than its more sequential counter part - the CPU.**

Note: All code is freely accessible on GitHub: http://github.com/HarryLong/CudaMiner

I. Introduction

In order to understand the context, it is important to get to grips with the asteroid miner problem. The asteroid miner is a drone whose task it is to mine the rings of Saturn for minerals. The drone is released at a given position $\mathbf{P_{drone}}$ on a 2 dimensional square grid. For simplicity, this point is referred to as the origin of the 2 dimensional grid (i.e $\mathbf{P_{drone}}$**=** $\{\mathbf{0,0}\}$). The aim is for the drone to reach it's base station $\mathbf{P_{base}}$ in the minimum amount of moves whilst collecting the maximum amount of minerals. The drone can only move in step size increments in 2 directions: horizontally to the right and vertically downwards. At each step location, the drone's tractor beam can pick up minerals surrounding the drone if the the distance of the mineral from the drone is less than or equal to half the step size.

The mineral data is presented as a binary file broken down into triplets of single precision floating point values **x, y and v** where:

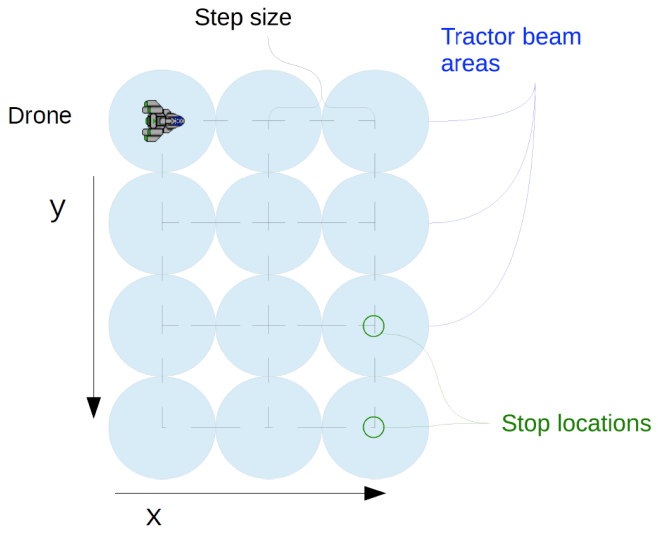
- **x**: x position of the mineral on the grid.
- **y**: y position of the mineral on the grid.
- **v**: value of the given mineral.

The only exception is the first triplet of the file which represents $\mathbf{P_{base}}$, the location of the base station (with a value of zero).

Finding the optimal path through the grid is a two step process:

1) **Binning**: During the binning process, each mineral must be processed and matched to its closest stop location. During this stage, minerals more distant than a half step size from their closest stop location can be discarded ad they are unreachable.
2) **Find best path**: Now that each step location has a matching value, it is necessary to find the optimal path. A path is deemed optimal if:
   - The number of steps to reach the base station is the minimum possible.
   - The aggregated value of all minerals collected when the drone reaches the base station is more than (or equal to) all other possible paths the drone can follow.

Fig. 1: Grid layout



The aim of this paper is to find an algorithm to optimize this two step process to run on the GPU rather than the ordinary CPU. The language we use is CUDA which, once compiled, runs solely on Nvidia GPUs. The problem is suited to run on a GPU as algorithms can be found in which the problem can be split into multiple work loads and ran in parallel. More on this in the ”Design and Implementation” section

II. Design and Implementation

In this section we will describe the algorithms used for the two steps of the problem: the binning and finding the optimal path. Note that reading the raw data from the input file is performed on the CPU prior to the binning.

A. Binning

The purpose of the binning stage is to match a mineral value to each stop location on the grid. For this, each mineral location $\mathbf{P_{mineral}}$ must be processed and matched to the closest stop location $\mathbf{P_{stop}}$ accessible by the drone. Once $\mathbf{P_{stop}}$ is found, it is necessary to calculate the distance $\mathbf{D_{mineral}}$ from $\mathbf{P_{mineral}}$ to $\mathbf{P_{stop}}$. Once $\mathbf{P_{mineral}}$ is calculated one of two possibilities can occur:

- **If** $D_{\textbf{mineral}} \leq {}^{stepsize}/_{2}$: Add the minerals value to its corresponding $\mathbf{P_{stop}}$
- **If** $D_{\textbf{mineral}} > {}^{stepsize}/_{2}$: Discard the mineral as it is unreachable.

Algorithm

The mineral data from the input file with the base station data removed is copied onto the GPU's global memory after which a thread is spawned to deal with each mineral. Each thread then:

- Reads it's corresponding mineral data triplet (x,y,v)
- Calculates it's corresponding closest stop location
- Calculates whether or not the mineral is reachable and then:
  - **if reachable**: Adds the mineral's value to its corresponding stop location in the pre-allocated grid.
  - **if unreachable**: Discards the mineral

Data layout

Two pieces of data are allocated on the GPU:

1) Mineral data: refer to figure 2

2) Grid data: refer to figure 3

Thread layout

Refer to figure 4 for a detailed description of the thread layout on the GPU for the binning.

B. Accumulating

After the binning is performed, we have a weighted grid where, for each stopping point, we have its corresponding mineral value. What is needed now is to find the minimum path which would result in the most mineral gain and result in the drone being at its base station.

Algorithm

A way to do so, which also spawns parallelizing potential, is to create an accumulated grid. To create an accumulated grid, the initial grid must be iterated over through its diagonals starting with the diagonal passing through through point [0,0]. At each intersecting stopping point on the diagonal we add the value of the maximum between the two unique points from which the point can be accessed (i.e the point directly to the left and the point directly up).

Using figure 5 as a reference, to calculate the aggregated value of point C would need to add to its current value the maximum between points A and B. I.e:

$$C_{\text{aggregated-value}} = C_{\text{value}} + max(A_{\text{aggregated-value}}, B_{\text{aggregated-value}})$$

Although the diagonals **must** be processed sequentially, the aggregated value of each stopping point on the diagonal can be calculated in parallel. The number of threads which can be spawned increased as we reach the center diagonal, is at its maximum on the center diagonal, then decreases again until we reach the final diagonal. For any diagonal, the number of threads $N_{(}$threads) which can be spawned can be calculated as follows:

$$N_{\text{threads}} = (diagonal_{\text{current}} + 1) - (2 * max(0, diagonal_{\text{current}} - diagonal_{\text{center}}))$$

Using the grid from figure 5, we get:

TABLE I: Spawnable threads per diagonal

$\mathbf{diagonal_{current}}$	$\mathbf{N_{threads}}$
0	1
1	2
2	3
3	4
4	5
5	4
6	3
7	2
8	1

Fig. 2: Memory layout of the mineral data

Representation	Array of floats
Type of memory used	Global memory
Size	$n_{\text{minerals}} * sizeof(float) * 3$.
Initialization	Allocated and data copied across from host
Optimizations comments	As each floating point value is read only once from this array, there would be no optimization in copying this data to a faster access memory medium (e.g shared memory).

Fig. 3: Memory layout of the Grid data

Representation	Although conceptually a 2-d array, for efficiency this is represented in memory as a flat array of floats
Type of memory used	Global memory
Size	$(grid_{\text{width}} + 1) * (grid_{\text{length}} + 1) * sizeof(float)$.
Initialization	Allocated from host. All elements of array set to 0 (using cudamemset). This is essential as it is not guaranteed that each index will be processed (possibility that no mineral will be close enough to a given stop location) and when an index is processed, its value is incremented rather than explicitly set.
Optimizations comments	As different minerals can be linked to the same stopping point it is possible for multiple threads to attempt to access the same location in global memory. For this reason an atomic add is used which, unfortunately, slows performance.

Fig. 4: Thread layout for the binning

Block size	256 * 1 * 1
Thread count	Number of minerals
Accessing mineral element	(blockIdx.x * blockDim.x) + threadIdx.x * 3
Accessing grid element	($\text{grid}_{\text{length}}$ * (grid_x+1)) + (grid_y+1)

Data layout

No new data usage is necessary. As it is necessary to process the diagonals sequentially, the initial grid can simply be overwritten.
The left-side and top-side padding can be better understood now: it is an optimization mechanism to prevent costly branching code on the GPU. The stopping points for which X = 0 and Y = 0 have only one point from which they can be accessed (top and left respectively), therefore branching would be necessary on the GPU to check whether X = 0 or Y = 0 for the current point. All padding points are set to a negative value and therefore:

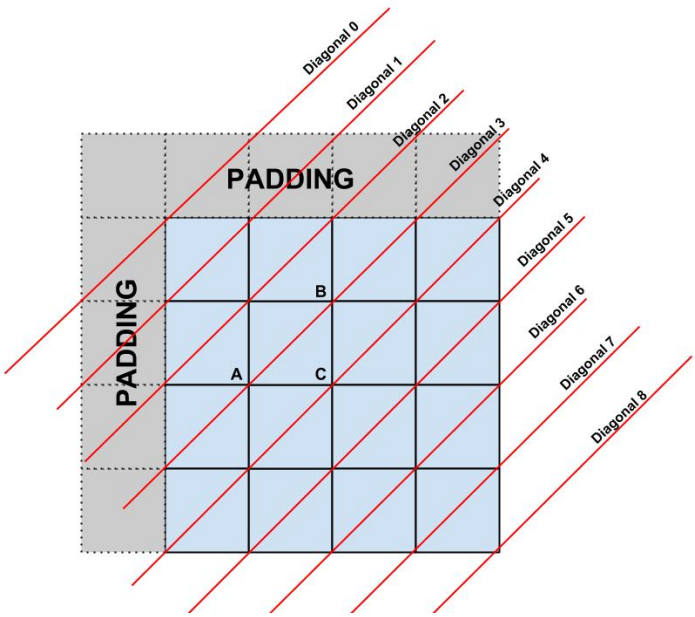
- Each valid stopping point (excluding padding) can access a valid memory location for the stopping point straight above and straight to the left.
- A padding point will **never** be chosen as the best of the two source points as all valid points have a corresponding value of at least zero.

Note that for optimization purposes, the value for the padding was chosen as -1.498. The reason for this is that it can be set with a more efficient cudaMemset using 0xbf as the byte value. This is much more efficient that using the costly alternative, a memory copy.

Thread layout

Refer to figure 6 for a detailed description of the thread

Fig. 5: Accumulated grid with diagonals



layout on the GPU for the grid accumulating.

C. Second implementation on the GPU

In an attempt to tackle the main bottleneck of this algorithm - the atomic add during the binning, an algorithm was developed which differed to this one in the following ways:

- Rather than having a single grid, each thread has its own grid.
- When a thread calculated its corresponding grid location, it sets the value on its own grid (no atomic add necessary).
- Once each thread has finished processing it's mineral and set its corresponding grid value, a parallel reduction of each threads grid occurs in order to retrieve the final grid.

A problem quickly became apparent with this implementation however: the memory usage quickly becomes unmanageable even for reasonable small grid sizes (see table 3 below).

TABLE II: Binning Algorithm 2 data usage based on grid size (with fixed mineral count of 1000)

Grid width	Size (in MB)
100	38.1468
200	152.5872
300	343.3212
400	610.3488
500	953.67
600	1373.2848
700	1869.1932
800	2441.3952
900	3089.8908
1000	3814.68

As a result of this huge memory usage, this second implementation was dropped in favour of the first.

III. Results and Discussion

The main purpose of running this problem on the highly parallel GPU architecture was to gain performance other running it on the more sequential equivalent - the CPU. In this section we will outline the performance gain, if any, as well as discuss the testing that was performed to ensure our findings were accurate.

A. Results

In order to grasp the performance increase of running the problem on the GPU the algorithms were also implemented to run on a regular CPU. For reference, below are the CPU and GPU specifications of the machine on which the benchmark tests were performed:

TABLE III: CPU specifications

Name	Intel i7 4th generation
Architecture	64 bit
Cores	4
Threads per core	2

TABLE IV: GPU specifications

Name	Nvidia GTX 760
Number of cores	1152
Global memory size	2048 MB

As stated in the previous section, the algorithm is split in two parts: the binning and the accumulation. The benchmarking was performed for both parts separately.

Benchmarking the binning

For the binning, the number of threads that are spawned is equal to the number of raw minerals. Therefore, we expect the performance gained by running it on the

Fig. 6: Thread layout for grid accumulating

Block size	256 * 1 * 1
Thread count	See above
Accessing corresponding memory element $grid_x$	(blockIdx.x * blockDim.x) + threadIdx.x) + max(0, $diagonal_{current}$ - $diagonal_{center}$)
Accessing corresponding memory element $grid_y$	($diagonal_{current}$ - ((blockIdx.x * blockDim.x) + threadIdx.x) + max(0, $diagonal_{current}$ - $diagonal_{center}$)

GPU to increase with the number of minerals passed through as input. In order to test this, a benchmarking application was implemented which gradually increases the number of minerals passed as input whilst keeping the grid size to a constant size of 250 by 250. This benchmarking application performs 5 runs for each configuration and calculates the average run time based on the 5 separate runs. This is to prevent any anomalies in the data caused by, for example, heavy CPU or GPU usage by a third-party application during the benchmarking tests.

Based on the results (plotted in figure 7) we can see that there is a significant increase in performance when the binning is performed on the GPU. The GPU overtakes the CPU in terms of performance as soon as we pass approximately 70000 minerals. Before this point, the overhead of allocating memory on the GPU and copying data to and from the GPU outweighs that of the performance increase obtained from performing the actual number crunching on the GPU. As expected, the speed up increases with the number of minerals which are to be placed on the grid. The maximum speed up occurs for the maximum number of minerals tested - 560'000. With this number of minerals, the GPU implementation runs almost 3 times faster.

In order to see what is taking the most GPU time, a breakdown was performed of all the kernel calls, memory allocations and memory copies which constitute the bulk of the GPU binning algorithm. Based on this data (refer to figure 8), we can rank them in order of resource usage:

1) **Mineral allocation:** Allocation of the grid in the GPU's global memory.
2) **Grid padding:** Setting the padding indices of the grid in GPU's global memory,
3) **Mineral copy:** Copy of the raw mineral data into GPU's global memory,
4) **Binning kernel:** The kernel call in which the binning is performed.
5) **Grid memset:** All grid values are set to 0 (except for the padding indices),
6) **Grid allocation:** The grid is allocated on GPU's global memory.
7) **Mineral free:** In order to re-use the space previously taken by the mineral data and no longer needed once the grid is created, the memory it reserved is freed.

Benchmarking the Aggregation

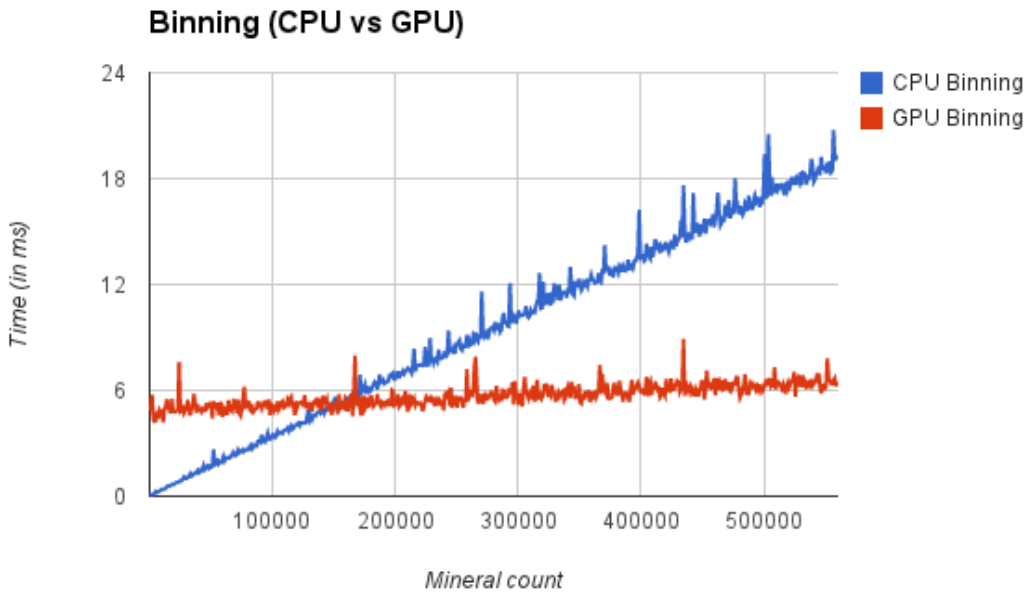
For the aggregation, the maximum number of threads which can be spawned is equal to the number of stopping points which intersect with the center diagonal. Therefore, the number of spawned threads increases with the size of the grid. As a result of this, we expect the performance gain of the GPU to increase with the size of the grid. In order to test this, the same benchmarking application was configured to gradually increases the grid size whilst keeping the number of mineral count to a constant 5000. As for the binning benchmarking, the results are averaged on 5 separate runs to prevent statistical anomalies.

Based on the results (plotted in figure 9) we can see that there is a significant increase in performance when the aggregation is run on the GPU. The GPU overtakes the CPU in terms of performance as soon as we use a grid size with over 1500 stopping points (vertically and horizontally). As expected, the speed up continues to increase with the size of the grid. The maximum speed up obtained was for the largest grid size tested (width and length = 19000). In this configuration, the GPU implementation runs almost 2.5 times faster.

Similarly to the binning benchmarking, a timing breakdown of the various elements which constitute the GPU algorithm was performed. Based on this data (refer to figure **??**), we can rank them in order of resource usage:

1) **The accumulation kernel:** This is the kernel in which the data is processed and the accumulation

Fig. 7: Binning aggregation benchmark



grid is built.

2) **Grid retrieval:** This is when the final accumulated grid is copied from the device back to the host.

B. Testing

In order to ensure the calculated optimal path is correct it is essential to perform some tests. Two types of tests were performed to ensure the algorithm worked as planned: unitary tests on individual core components of the algorithm and results comparison between the results obtained from the CPU and the GPU.

Comparing results from the CPU and the GPU

As discussed in the Results section, a benchmarking application was developed in order to compare speed ups with different input minerals and grid size configurations. In order to test the results at the same time, each run (5 runs per configuration) also ensured the calculated optimal path returned by the CPU run and the GPU run were identical.

A problem which becomes apparent with the above test if is both the CPU and GPU output the same incorrect optimal path. There needs to be a way of ensuring that the calculated optimal path is not only identical for the CPU and GPU code, but also correct. In order to do so a functionality was implemented in the testing application to generate an input file with a preconfigured optimal path. That is, given the step size, the minerals are placed in such a way that the optimal path will always follow the diagonal of the grid (right, left, right, left, etc...). By using this pre-configured input file along with the comparison between the CPU and GPU outputs we can ensure that:

- The CPU and GPU generate the same optimal path.
- The generated path is correct.

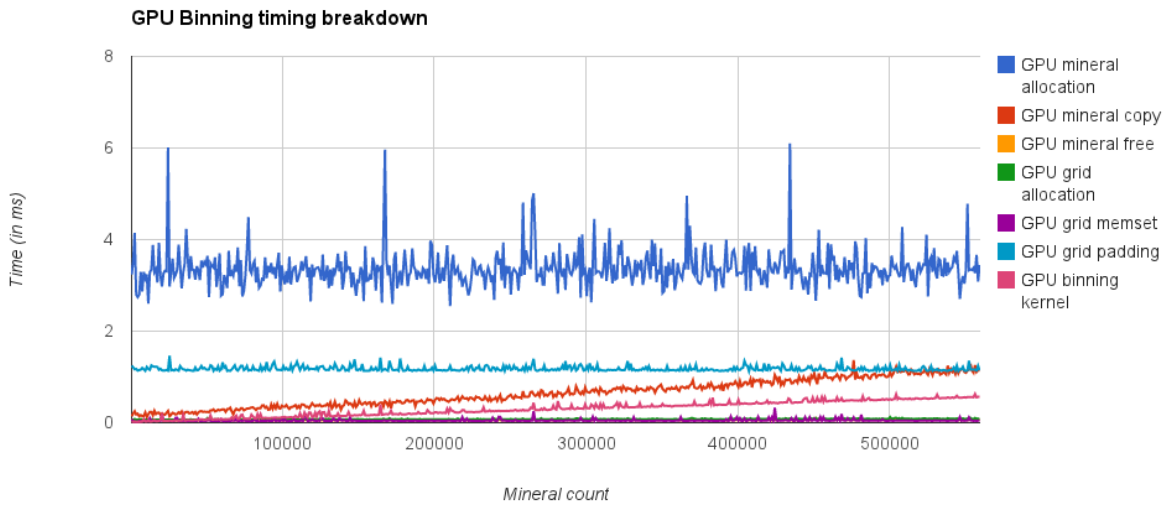
Unit testing

A unitary test was developed in order to test the two core functions of the application - the binning and the aggregating.

To test the binning, it is a simple matter of ensuring each mineral is placed in the correct grid index. To ensure this, a pre-configured optimal path input file is used with the minimum number of minerals (one for each stopping point on the diagonal). The resulting binned grid is then written to a file for examination. The test is deemed successful if only the grid indices on the diagonals have values other than zero.

To test the grid aggregation a different unitary test was developed: The grid which is passed to the grid accumulation kernel has all values set to 0 except the stopping point at index [0,0] which is set to 1. The resulting accumulated grid is then written to a file for

Fig. 8: Binning Timing Breakdown



examination. The test is deemed successful if each stopping point on the grid index is the sum of the stopping point straight above and straight below. Note that, in order for this test to work, the padding points were exceptionally set to 0 (rather than -1.498). Please refer to figure 11 for a successful accumulated grid test for a 4 by 4 grid.

IV. Conclusions and Future Work

As seen in the results section, the GPU implementation lead to a significant increase in performance. This performance increase was notably superior for the binning over the grid accumulation. The reason for this is most definitely because a lot of the work for the grid accumulation needs to be performed sequentially (i.e each diagonal). The binning, however, is greatly suited to high parallelism as each individual thread can deal with a single mineral.

It would potentially be possible to increase the general performance by finding ways to remove the main bottlenecks of the algorithm:

- The atomic add during the binning.
- Whilst creating the accumulated grid, it is possible for two threads to attempt to access (for read purposes) the same grid index.
- The branching on the GPU during the binning stage: When processing all the minerals for binning purposes, the GPU tests whether of not the mineral is reachable. Depending on the outcome of this test, the GPU either adds the mineral's value or ignores it as it is deemed unreachable. If a way is found to avoid this branching, performance could be improved yet further.

V. Aknowledgements

In no particular order I would like to thank:

- The University of Cape Town's computer science department for putting an Nvidia GPU at my disposal to play with. Without which this work would not be possible.
- Chris Laider for his support throughout the course.
- John Stone, Manuel Ujaldn, Bruce Merry & Simon Perkins for their invaluable talks on the subject.

Fig. 9: Grid aggregation benchmark

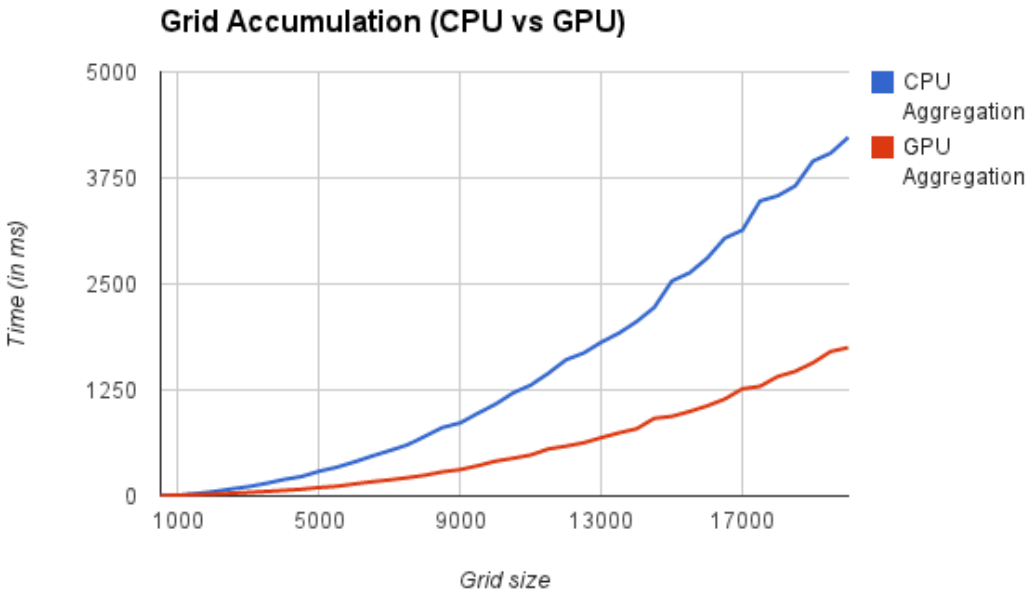


Fig. 10: Grid accumulation breakdown

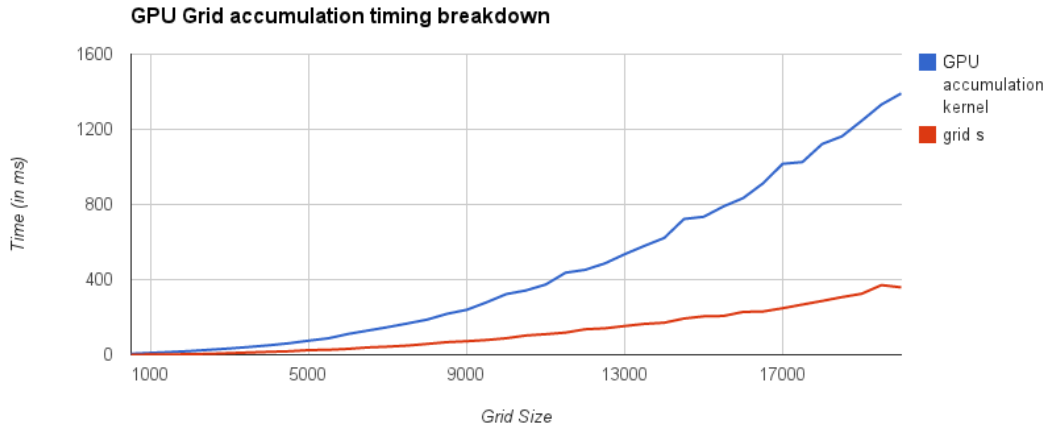


Fig. 11: Successful Accumulated Grid Test