ITS Cluster Finding Benchmark on GPU

Summer Student Report
06/08/2014

Boonyarit Changaival
Department of Computer Engineering
King Mongkut's University of Technology Thonburi, Thailand

Supervisors
Sylvain Chapeland
ALICE, CERN

Assoc. Prof. Dr. Tiranee Achalakul
Department of Computer Engineering
King Mongkut's University of Technology Thonburi, Thailand

Introduction

ALICE is planning an upgrade in 2018. After which, the amount of data readout from ALICE detectors is expected to exceed 1 TB/s. Therefore, those data must be compressed before feeding to a farm of processing nodes. "ITS Cluster Finding" is the first algorithm that is used to reduce the data from ITS detector. The data is represented by X and Y coordinates of hits on chips in the ITS detector. The function of this algorithm is to group the adjacent hits together into clusters. Then the center of gravity of each cluster is calculated to represent its coordinates. Provided that the data is fed in a continuous stream, the throughput rate (number of event processed in a single amount of time) is the metric measuring the implementation performance. To survey a variety of platforms, this algorithm had been implemented on x86, GPU and Xeon-Phi. These platforms promote parallelism to increase the overall throughput. The result will help selecting the platform(s) for the final system design.

My scope of work during the internship was to implement and benchmark the cluster finding algorithm on GPU using C and CUDA library. GPU is known for its overwhelming number of processing cores. Even though those cores are not powerful compared to a typical CPU, but with such number, the task can be broken down into small pieces and being processed concurrently. The performance gain is distinctively high in various kinds of task such as matrix manipulation. With a given format of data and the algorithm function, GPU seemed to be a very good platform since the algorithm can be derived into a very fine grain level, possibly one thread for one hit on the chip because of its massive amount of processing cores. For example, in GTX780, there are 2304 cuda cores comparing to i7 with just 4 cores (but, of course, more powerful). In addition, GPU was shown to be very efficient with another cluster finding algorithm for ALICE TPC detector. This made GPU became even more attractive and promising.

Given that GPU has a different architecture and memory management, I could not follow the existing x86 code. Hence, new method was proposed and compared with the existing x86 code for benchmark. This report is organized as follow. In section II, the context of the problem is introduced and shows how GPU could be adapted to the problem with the new implementation. In section III, the result of the implementation in comparison to the existing code is shown and discussion is presented to explain the outcome of the result. Finally the conclusion is provided.

2. ITS Cluster Finding

2.1 Data Format and Structure

Cluster finding algorithm is the first reduction step in the ITS detector processing. The data is categorized by event number and chip number. In this simulated data, there are 430 chips in a single event with each chip contains from 70 to 1000 hits. The dimension of each chip is 650x1500 pixels. These coordinates (or hits) are read and kept in a one dimension array with a mark to keep track of the event they belong. With an array format, it is easier and faster to transfer these coordinates onto the GPU.

Event 1

ChipNo.	DigitNo.	Column	Row
0	0	504	519
0	1	505	519
0	2	506	519
0	3	507	519
...	...	...	...

Figure 1: Data Format

There are two modes of clustering those hits; 1) Diagonal Off and 2) Diagonal On. In this experimenting phase, only Diagonal Off mode was implemented. In this mode, hits are grouped with other adjacent hits in four directions (up, down, left, right). In Diagonal On, the adjacent directions increased to eight (diagonal directions included). Once every hit in the chip belongs to a cluster, the center of gravity of a cluster is computed and this is the data that will be kept. Figure 2 show the example of the output. This output will be cross checked with the expected output for validation purpose.

#1

ChipNo.	X coordinate	Y coordinate	Pixels
0	+0.4875	-0.4880	8
507	520		
506	520		
505	520		
...	...		

Figure 2: Output format

2.2 Flow Chart

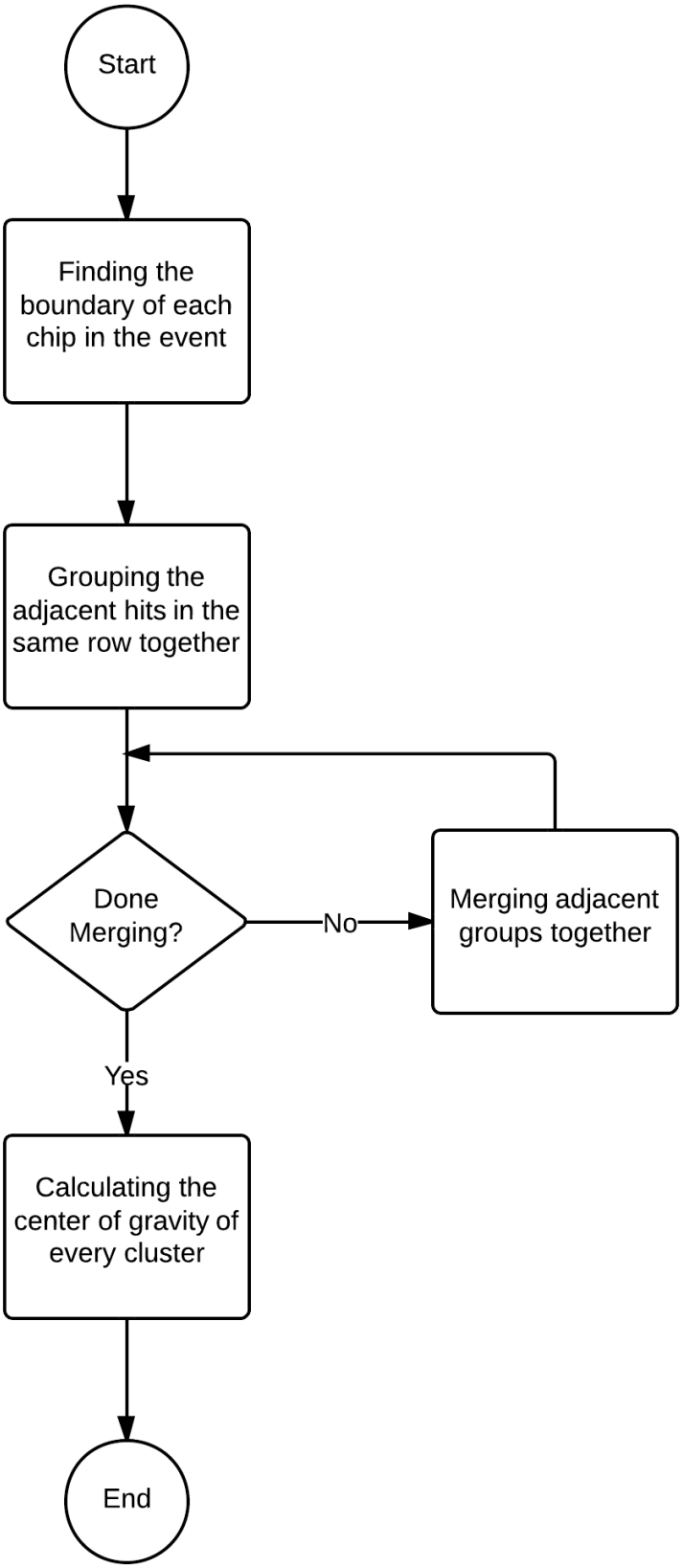


Figure 3: Algorithm flowchart

2.3 Function Description

On a GPU, a barrier can block only threads in the same block. To process an event as fast as possible, multiple blocks with multiple threads inside are launched to process a single chip. These hits on the same chip depend on each other in the clustering process. With no barrier, some threads might finish faster than others. Hence, there is a chance that those threads will use incorrect data for further processing which lead to an incorrect result. Due to this synchronization problem on GPU that can only be done in a small scale, I needed to write small functions instead of two or three large functions. There are 6 steps in cluster finding algorithm on GPU and they are listed below.

1. Finding Boundary
2. Pre-grouping
3. Assigning Cluster
4. Joining Cluster
5. Cluster Summation
6. Calculating Center

2.3.1 Finding Boundary

The input data are kept in a one dimensional array for transferring purpose. Only the start and end of the event are marked. In a single event, there are 430 chips and the number of hits in the chip is varied. This function is used to find the boundary of chips in an event, so that once the whole single event is put on the GPU for processing, those hits will not be mixed up and cause an error. From figure 1, the table demonstrate the boundary of each chip. Green is for chip number 1 and so on.

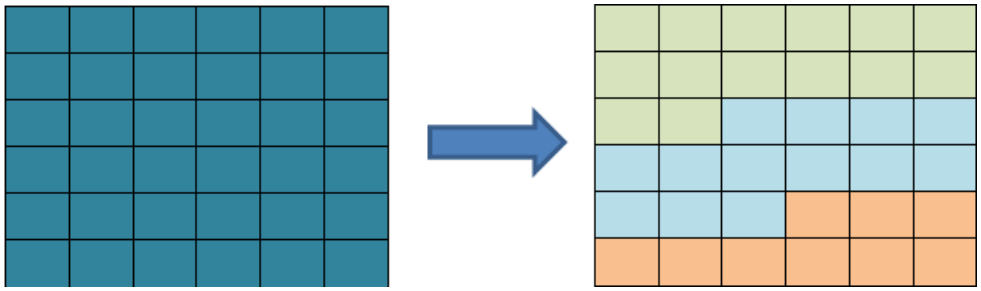


Figure 4: Boundary of each chip in an event

2.3.2 Pre-grouping

This function groups the adjacent hits together. The thread synchronization in GPU be done in a very small scale. To prevent the race condition to happen, the hits are needed to be assigned with the initial ID first. If all hits in the chip were to be arranged in a table, it would look like figure 5. The white squares represent the missing coordinate on the chip.



Figure 5: Adjacent hits in a group on a single chip

2.3.3 Assigning Cluster ID & Joining Cluster

These two functions are called to merge adjacent clusters. Clusters are first marked to be merged with another in "Assigning Cluster ID" and then merged in the "Joining Cluster". In order to completely merge all adjacent clusters, these two functions are executed iteratively (average at 4 times per event).

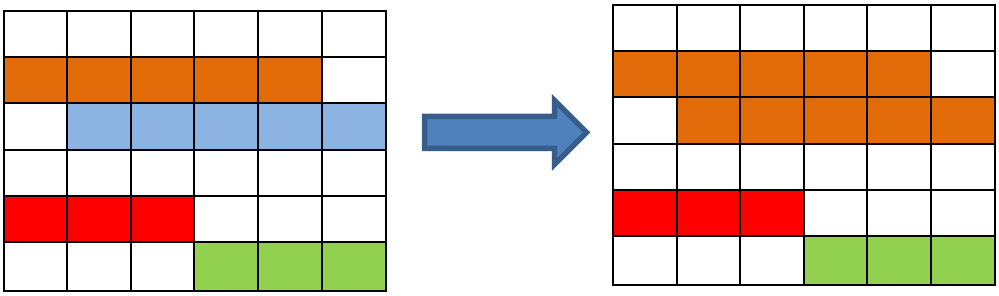


Figure 5: After Merging groups

2.3.4 Cluster Summation and Calculating Center

In order to calculate the center of gravity of the cluster, the sum of all X and Y coordinates is collected first with the numbers of hit in the cluster in "Cluster Summation". After the "Cluster Summation" process, the center of gravity of each cluster is calculated in "Calculating Center". Only these resulting coordinates will be used and the rest can be discarded.

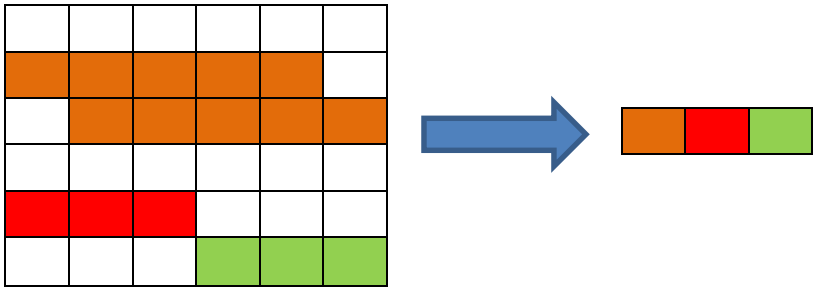


Figure 6: Cluster Representation

2.4 Optimization Approaches

Optimizing the algorithm is not a trivial task at all. Considering all constraints on GPU, I could come up with some ideas to experiment. In this part, some of the notable ideas are mentioned. Each approach is branched from the base line version except the pipeline approach. It was built on the success of "Thread number reduction" approach. Please note that the result of each approach depends on GPU types.

2.4.1 Multiple GPU

This approach is very straight forward and so the result. With this approach, the work is distributed evenly to each GPU in the system and since events are independent, there was no problem doing this. The more GPUs in the system, the higher throughput. However, this approach is still futile, since there is a limit to the number of GPU that can be used.

2.4.2 Zero Copy and Non-blocking data Transfer

This approach stemmed from the code profiling. The data movement took a lot of time to initiate and can cause a bottleneck. Therefore, to be able to remove this part out would result in an improved performance. Zero Copy is introduced to the context. According to NVIDIA, Zero Copy memory is allocated on the host machine, but can be seen and accessed by the GPU directly. Asynchronous transfer or Non-blocking communication is then used to speed up the transferring process. Despite the convenience, non-pageable memory which is a kind of memory that cannot be moved out of memory is needed for both methods. The excessive use can lead to a performance degradation of the whole system.

2.4.3 Reduce the number of thread per block

Following the example in the documentation of CUDA, the kernel also takes time to initialize the entire environment on the GPU. Hence, sometimes, using a lot of thread can lead to performance degradation. Using a lot of thread can also prevent coalesce memory access. Reducing the number of thread will cut down some initialization time and promote coalesce memory access on GPU.

2.4.4 Pipelining (using stream)

There are actually three phases in kernel execution. The first phase is copying data from host to device. The second phase is kernel execution and the last one is to copy back the data from device to host. With pipelining, while the first kernel is being executed, the second event can start the first phase right away. The pipelining can be extended to three phases which is the copying from device to host. However, this feature is limited to some GPUs only.

3. Result

Machine Specification: CERN SLC 6.5 64 bit, CPU: Intel Xeon E5-2650,
GPU: Nvidia Tesla K20Xm

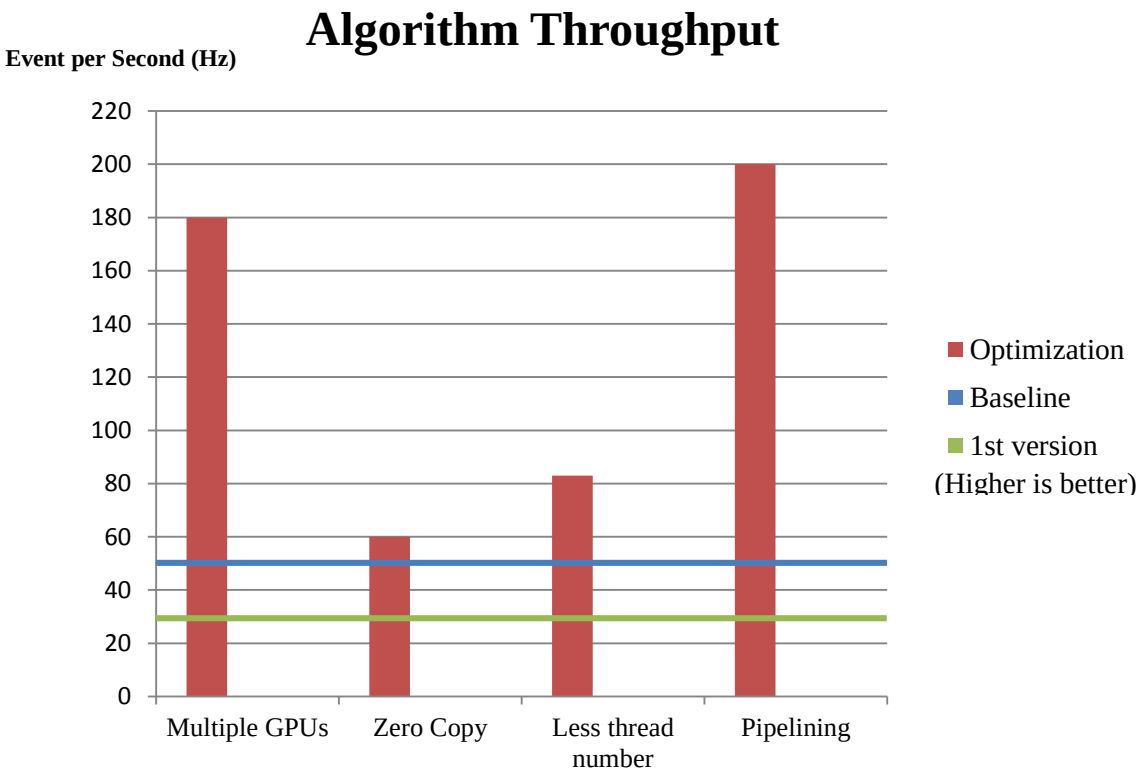


Figure 8: Baseline versus other approaches throughput (Tesla K20Xm)

3.1 Runtime per Kernel

In order to measure the run time per kernel, the Nvidia profiling tool is used. On Linux based OS, it can be used by typing the following command.

```
nvprof ./Application_Name
```

The profile being used to show the runtime of each kernel was taken from the "Less Thread Number" approach due to the fact that it processed the data in an event by event fashion like Baseline version, but faster. The time showing here is the average time for each kernel.

Tasks	**Runtime**	
	Time (ms)	**Percentage**
Finding Boundary	0.014	0.1%
Pre-grouping	0.07	0.4%
Assigning Cluster	2	15%
Joining Cluster	0.2	1.4%
Cluster Summation	8	62%
Calculating Center	0.015	0.1%
Overhead (data transfer, initialization, tasks on CPU, etc.)	2.7	21%
Total	13	100%

The *Cluster Summation* process is defined as a hot spot in this algorithm. It is inevitable for this kernel to contain a big loop because sum of X and Y coordinates and numbers of hits in clusters are needed to be collected. I tried to use multiple blocks to speed up the process, but it resulted in an erroneous output, because of the race condition in writing to the memory. Some tasks are performed on CPU due to synchronization on GPU and contributed to the overhead in the table.

3.2 Result Explanation

Figure 3 shows the runtime comparison between baseline version and other optimization approaches. The higher the bar, the better the result. The first bar shows that using multiple GPUs definitely boosts up the throughput by n times where n is the number of GPUs involved in the system. However, this result is speculated from several past experiments with the same scenarios, but low-end GPUs. Therefore, I believe that it is safe to state that the throughput will be increased to the factor of a number of GPUs in the system.

In the second bar, no difference can be observed in the second bar. This is due to the synchronization issue. The GPU needs time to write back on the host memory. Therefore, no performance was gained from this approach.

In the third bar, though, there is a surge in performance. Tesla K20Xm has two copy engines. Only one engine was used in this case. At first, I planned to use the second engine for copying data from host to device. The step would be converted those data to zero and then transfer it to the device. However, this approach did not show any significant improvement over the *memset* function offered by CUDA library. Even though, only one copy engine were used, the throughput was increased by a factor of four because the data transfer is overlapped. I would like to note though, the result of this approach is varied from model to model. I have

tried on two low-end models (NVS 300 and GT 550M), but they showed no sign of improvement. Although, another top-tier GPU such as GTX 780 showed an impressive throughput result at a factor of three.

The next approach is using the less thread number for each kernel. The idea is to reduce the environment initialization time before kernel execution. While this approach did not show the best result on high-end models such as Tesla K20Xm and GTX780, the throughput can be gained by 50% on both low-end models. I believed the reason to be the number of cuda cores on the device. Tesla K20Xm and GTX 780 have over 2000 cores, while there are 16 and 64 cores on NVS 300 and GT 550M respectively.

3.3 Discussion

GPU version performs around 5 milliseconds producing the throughput of 200 Hz at best. While, the x86 version can achieve the throughput of up to 500 Hz per thread on ivybridge. The gap in performance is large since it will be used in an online environment which requires much higher throughput. To investigate the performance, not only time was used, but also profiling tool. Using Nvidia profiler, the program was shown to be able to occupy the device for more than 80% most of the time. In this section, I would like to discuss why this time gap occurred and how could we move closer to the target.

First of all, this task is not computing intensive, but much of a data intensive. There is not much floating point calculation in the code. Most portions are just a load-compare-store. This is not one of GPU specialties which lie in complex computation of floating point. In this algorithm, the hotspot is in assigning cluster process because hits are dependent with others. While in the process of calculating the center of gravity, it can be executed fast because there is no dependency between hits in this kernel.

Secondly, it is inevitable for kernels to contain a loop and branch inside due to the dependency among hits. By introducing a loop inside a kernel, the performance is deteriorated. Furthermore, not all the thread executed the same amount of time in loop, even the thread that finishes first need to wait for the other threads in the block which lead to another bottleneck. The same is applied to the branch case. However, this problem can be partly solved by tuning the number of thread per block as you can see from the result showing in table 1.

Thirdly, the data transferring between host and device took long to initialize. This could be observed from the code cleaning process. I removed excessive transferring instructions and was able to reduce the runtime by 15%. Following this success, I tried the Zero Copy approach to eliminate most of the data movement. The result was not expected. It was slower because the synchronization was needed for GPU to write on the host memory. With more research, Zero Copy will only achieve “Zero Copy” only when a GPU share the same memory with the CPU. The discrete GPU has its own memory, so the data movement still occur. However, it still benefits from this through the GPU high bandwidth because Zero Copy memory is a non-pageable memory that is mapped to the GPU address.

Lastly, the synchronization on GPU is in a very small scale. The synchronization can only be done in a block scale, but multiple blocks can be launched at once on a piece of data. This means that threads in different blocks cannot be synched leading to the race condition on data. Also from the experiment, putting loops inside the kernel is not efficient and should be avoided. This cause the algorithm to be implemented with six small kernels to prevent race condition between threads and some kernels were needed to be called several times. To make the matter worse, it takes more time to launch a GPU kernel than launching a function on CPU. Due to the nature of this algorithm which has a lot of dependencies, barriers are put after each kernel to ensure the correctness. This prevents the pipelining approach from achieving its full potential because it cannot move on to the next kernel right away and the largest event in the

queue will serve as the bottleneck. However, as you can see from the result, pipelining is our best bet.

In conclusion, due to several constraints from memory management and architecture and the nature of the algorithm that has a lot of dependencies, only one GPU might not enable us to achieve the target throughput and it would require intensive tuning to a specific model for it to work well. This implies that for the next upgrade and in the case that the system has GPUs changed, the algorithm must be revised over again. In addition, several reasons were mentioned above. Data movement between host and device contributes to overhead, but what I believe to hinder the performance the most is the synchronization problem. While CPU has mutex and can perform loop at a much higher rate, GPU does not offer a large scale barrier and cannot handle the loop that well. As you can also see from the result that the best GPU version still cannot be compared to the CPU version which portrayed the difference in performance quite well.

4. Conclusion

After the upgrade of ALICE, many tasks currently done offline will be moved to online. This put the next system design to focus on achieving higher throughput in real time fashion. Not to mention that the amount of raw data from detector will be at 1TB/s. Hence, data compression algorithms are needed. Several platforms are studied in order to achieve the most suitable design for the upgraded system. Studied platforms included CPU (thread), GPU and Xeon Phi. All choices promote parallelism to improve the performance. Given that the data comes in a form of continuous stream, any delay can create the need for bigger buffer on the machine. Since the significance of such data cannot be known before analysis, the risk of discarding the important data is high. My work here is to implement and benchmark the performance a compression algorithm for ITS detector on GPU. GPUs were already successfully used with TPC detector.

In section 2, the cluster finding algorithm and optimization approaches were proposed. The algorithm was implemented in C using CUDA library. The algorithm is separated in 6 small functions (or kernels) to group all adjacent hits on the same chip together. Then, the centre of gravity of each group or cluster is calculated to represent all hits in the cluster. By doing this, the data per event can be compressed by 4 times. In addition, the whole grouping process needs to be done online and hence has to cope with the continuous stream of data. Achieving the target throughput is not trivial. Several optimizations were done such as Zero Copy, Non-blocking transfer of data, parameters tuning and pipelining which helped improving performance by a factor of 4.

Through the experiment, it can be observed that the GPU version is slower than the x86 version. This is because GPUs still have some constraints which I believe to be a hindrance from achieving the target throughput. The first and foremost is the synchronization problem. This forced me to write several small kernels which introduce an overhead to the algorithm since launching a kernel on GPU takes longer than launching a function on CPU. Another reason is the algorithm itself. This algorithm does not do much of a calculation. In most kernels, they only did the load, compare and store which is not a GPUs' specialty. Plus, there are a lot of dependencies among the hits as well. These factors contributed to the overhead and cause difficulties for GPU to reach its full potential. However, I still believe with fine tuning and may be slight changes in some parts of the algorithm to fit the target device model might be able to improve the performance as it had been tested on Tesla K20Zm, GTX780 and GT550M. Moreover, this occurrence implied that once the system change the specification, the algorithm is needed to be revised all over again to let it match perfectly with the target model to reach its full potential, which is not promoting sustainability.