THE AUSTRALIAN NATIONAL UNIVERSITY

College of Engineering and Computer Science

Inter-block synchronization on a GPGPU

SHAOXUAN SHEN (SHAWN)

Project Supervisor

Dr. Eric McCreath

November 2013

A thesis submitted in partial fulfilment of the requirements for the degree of
Master of Computing
At The Australian National University

Acknowledgements

I would like to thank my supervisor Dr. Eric McCreath for his guidance, assistance and patience during the entire project period. And, I would also like to appreciate the course co-ordinator, Dr. Weifa Liang, for sharing his valuable experiences in writing report and giving presentations. In addition, I would like to express my gratitude to Dr. Peter Strazdin for spending time to exam my paper. Last but not least, thanks to my family and friends for their support.

Abstract

With the invention of multi-core processing unit technology, the graphics processing unit has evolved from single core graphic processing unit to multi-core programmable graphics processing units. Because of the GPUs' architecture, people found that it is not only good at processing graphics related data, but also suitable for performing general purpose parallel computations.

However, since global synchronization plays a big part in parallel programming, the adoption of the GPU as a general purpose computation device was limited because there was no explicit support for global synchronization on GPU without going back to the host CPU. Recently, Xiao and Feng proposed three GPU global synchronization algorithms, implemented them on NVIDIA's CUDA platform and tested the performances. Also there exist a few other CUDA GPU synchronization implementations which are publicly available. However, none of them were implemented on AMD's OpenCL platform.

In this project, I implemented two GPU-based global synchronization algorithms, the Decentralized Barrier Algorithm and Centralized Barrier Algorithm, on the OpenCL platform. Performances of these two algorithms were recorded and compared against the traditional CPU based global synchronization. From experiment result, both GPU-based global synchronization approaches work better than CPU-based global synchronization when the synchronization method is used for more than one time in an application. Moreover, the performance of Decentralized Barrier Synchronization is higher than that of Centralized Barrier Synchronization. However, there are some restrictions when applying Decentralized Barrier Synchronization.

Contents

1. Introduction

1.1 Overview

In the past few years, the computing speed of computers has been improved by using more processing cores rather than increasing the clock speed of one core. The reason was because of power consumption and heat generating of hardware. The graphics processing unit (GPU) took the advantage of multi-core technology and has evolved from single core graphics processing unit to multi-core graphics processing units (GPUs).

At first, GPU was designed to process graphic related data/applications. However, after the invention of GPUs, people found that GPUs are also capable of doing general purpose computation because of its parallel processing architecture and the significant number of programmable processors in GPUs. Thus, general-purpose GPU (GPGPU) technology showed up which is the utilization of GPU to perform computation in applications traditionally handled by CPU [2]. Programming models such as NVIDIA's CUDA[3] and AMD/ATI's OpenCL[4] appeared right after, and made it faster to process applications by mapping the applications to GPUs and perform parallel processing.

1.2 Motivation

Currently, GPUs only work well with applications which require minimal or no communication between blocks. This is because there is no explicit support for communications between blocks on GPUs and such communication can only be achieved by using global memory and barrier synchronization or via going back to host CPU. AMD/ATI's OpenCL does support intra-block synchronization which allows all threads running in the same block to communicate with each other. These threads run on the same physical hardware in parallel and share the same local memory.

According to "Inter-Block GPU Communication via Fast Barrier Synchronization"[1], Shucai Xiao and Wu-chun Feng have worked out 3 different algorithms for the purpose of global synchronization on GPGPU, and these three algorithms have been implemented and tested using NVIDIA's CUDA. Moreover, some GPGPU algorithms were published online, but not in detail.

This project is conducted to explore the possibility on implementing the exiting algorithms on AMD's OpenCL. If it is possible, then implement some of the algorithms and test the performance. An additional task of this project is to resolve the deadlock problem which happened during algorithm implementation using NVIDIA's CUDA. The details of this problem are mentioned in Tao Wang's paper [5].

The execution time of a general purpose computation application with inter-block synchronization involves three phases, kernel launch time, computation time and synchronization time. However, in this project, these phases are tested as a whole rather than to be tested separately. Each selected algorithm is tested in N iterations. But the difference is, CPU synchronization goes back to CPU to get global synchronization done at the end of each loop, while other algorithms stay in GPU and use the algorithm to finish global synchronization.

1.3 Project Scope

This project is divided into five parts.

1. Learn and get start with AMD's OpenCL parallel programming language.
2. Study existing GPGPU inter-block synchronization algorithms and explore the possibility to implement them using OpenCL language.
3. Implementation of chosen algorithms and try to resolve deadlock problem at the same time.
4. Test and record the performance of implemented algorithms.
5. Report writing and project delivering.

Figure 1-1 shows the detailed schedule,

Week	Task
1	Finalize project specification with supervisor.
2	Start learning Opencl.
3	Install Opencl and write simple test program.
4	Develop program involving local synchronization.
5	Research on GP-GPU global synchronization.
6	Develop program to achieve GP-GPU global synchronization.
break	Develop program to achieve GP-GPU global synchronization.
7	Develop program to achieve GP-GPU global synchronization.
8-9	Test and record code performance.
10	Start writing final report.
11	Write final report and prepare for seminar.
12	Finalize and submit final report, give seminar.

Figure 1-1 Project Schedule

1.4 Contribution

The contributions of this project includes,

- The basic task of this project is to learn AMD's OpenCL parallel language and implement some of the existing inter-block algorithms using OpenCL. Thus, the outcome of this project could be considered as a very good starting point if they decided to explore in this area further. This project provides an understanding of the deadlock problem.

- The deadlock problem was caused by hardware limitation. This is a bottleneck in GPGPU inter-block synchronization domain. If this problem is resolved, it can be a big break through this area.

- Understand and measured the overhead of inter-block synchronization using OpenCL. Comparisons are done between CPU Synchronization, Centralized Barrier Synchronization and Decentralized Barrier Synchronization.

1.5 Report Structure

This report is structured as follows,

- Chapter 2 provides the background of this project including hardware specification, coding platform and tools used, some important concepts in this project as well as the left over from previous work

- Chapter 3 explains how both CPU-based and GPU-based synchronization algorithms work and how they were implemented. In the last section of this chapter, an incomplete deadlock solution is discussed.

- Chapter 4 explains how the tests were conducted and the testing results are analysed.

- Chapter 5 provides the conclusion of this project, possible future works and lessons learnt from this project.

2. Background

2.1 Hardware Information

The project is to be conducted on the student's laptop. It is equipped with ATI Mobility Radeon HD 5870 graphics card produced by AMD which is a high-end graphics card in ATI Mobility Radeon HD 5800 series, and the graphics card is AMD APP SDK supported. In this chapter, we discuss about the architecture of the GPU from OpenCL perspective.

2.1.1 ATI Mobility Radeon HD 5870 Graphics Card

The specification of ATI Mobility Radeon HD 5870 graphics card is the same as that of ATI Radeon HD 5770 Juniper XT. It has 10 compute units with engine speed clocked at 850MHz. Each of these compute units consists of 32 Kb of local shared memory which can be used to perform synchronization within each group, and 16 stream cores which execute program on independent data streams. Further down, there are 5 processing units in each of the stream cores which are in charge of executing fundamental basic instructions. Figure 2-1 shows the structure of this graphics card.

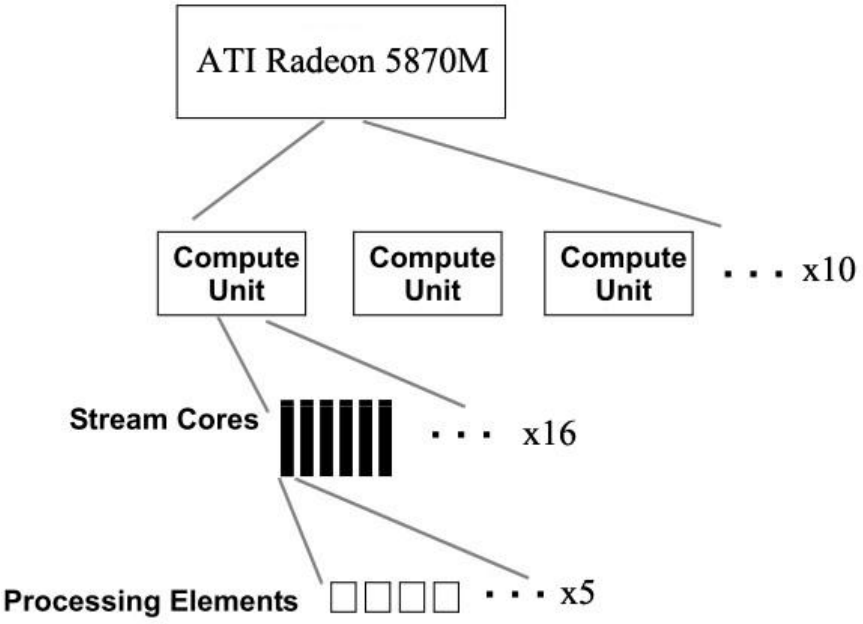

Figure 2-1 Project graphics card structure [6]

This graphics card has a 1GB GDDR5 global memory with the speed of 1200MHz. Since it can be accessed by threads in all groups, it is possible to use it to communicate between groups or even accomplish inter-block synchronization.

2.1.2 GPU Memory and communication with CPU

The AMD Accelerated Parallel Processing system recognizes 6 different kinds of memories, they are, private memory, local memory, global memory, constant memory, host memory and PCIe memory. Figure 2-2 illustrates the interrelationship of these memories.

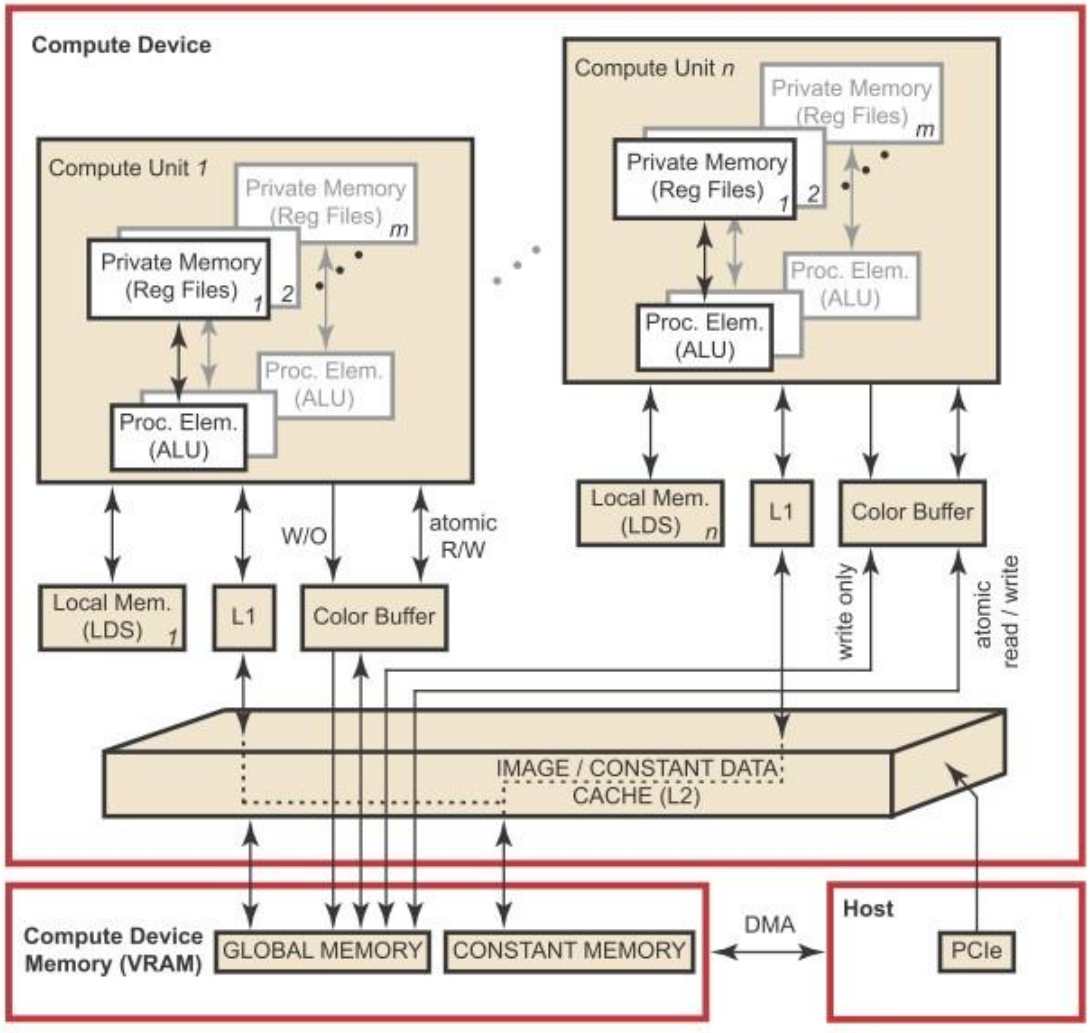

Figure 2-2 GPU Memory domains architecture [7]

The private memory is a block of memory specific to a work-item. Any variable that is defined in a work-item as private variable is not visible to other work-items. The local memory is a block of memory specific to a work-group. Variables defined as a local variable can only be seen by all work-items in the same work-group. The global memory is accessible to all work-items and all work-groups as well as to the host. It allows read and write operation to all work-items in all work-groups and map commands to host. The constant memory is a block of memory that is read only. It is allocated and initiated by host before kernel launch. And it is not allowed to be modified during execution of the kernel.

PCI Express (PCIe) memory is part of the host memory which is accessible and modifiable by both host program and GPU compute device. Thus, modifying this memory requires synchronization between GPU and CPU. In other words, modifying this memory consumes some time. Comparing the synchronization time between using global memory and using host PCIe memory would be one of the tasks in this project. Figure 2-3 illustrates the standard dataflow between CPU and GPU.

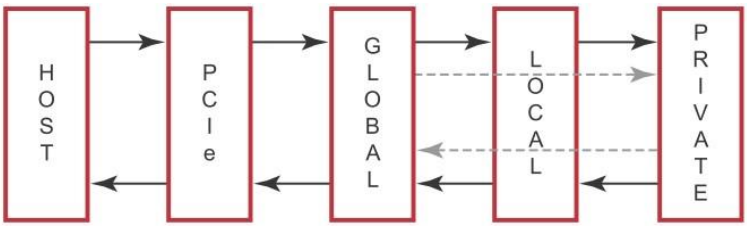

Figure 2-3 Memory dataflow between CPU and GPU [8]

2.2 Software Information

2.2.1 AMD APP SDK

AMD provides the AMD APP SDK programming model and OpenCL software environment. They allow the programmer to quickly and easily develop applications accelerated by AMD APP technology using a semi-C++ coding style.

In general, kernel code consists of the works that the programmer wants to get done in parallel. In OpenCL, the kernel code is to be read and compiled to the device's instruction set first, and then it is to be launched before it can be executed.

There is explicit support for local memory synchronization. This can be done by calling the barrier() function in kernel. This function ensures proper work-item execution order within one work-group as well as updates data in local/global memory.

The steps of executing a GPGPU OpenCL application are as follows:

- Host compiles the kernel code
- Define constant, local and global memory usage of the application
- Allocate memory spaces and initiate constant, local and global variables if necessary
- Launch the kernel in an N-dimensional fashion, and each item in the dimension represents a work-item
- All work-items in this N-dimension execute the same kernel code, but on different data streams.
- Wait for all work-items to complete and do global synchronization on host

2.2.2 Work-item and Work-group

In a GP-GPU, when a kernel is launched, many threads are launched at the same time. We call these threads 'Work-items'. These work items are to be wrapped into different small groups called 'Wave-fronts' automatically, and work-items in the same wave-front are to be executed together in strict parallel. After that, these wave-fronts are to be grouped into 'Work-groups'. All work-items in the same work-group share the same local memory and there is explicitly support to get them synchronized. Figure 2-4 illustrates the relationship between work-item, wave-front and work-group.

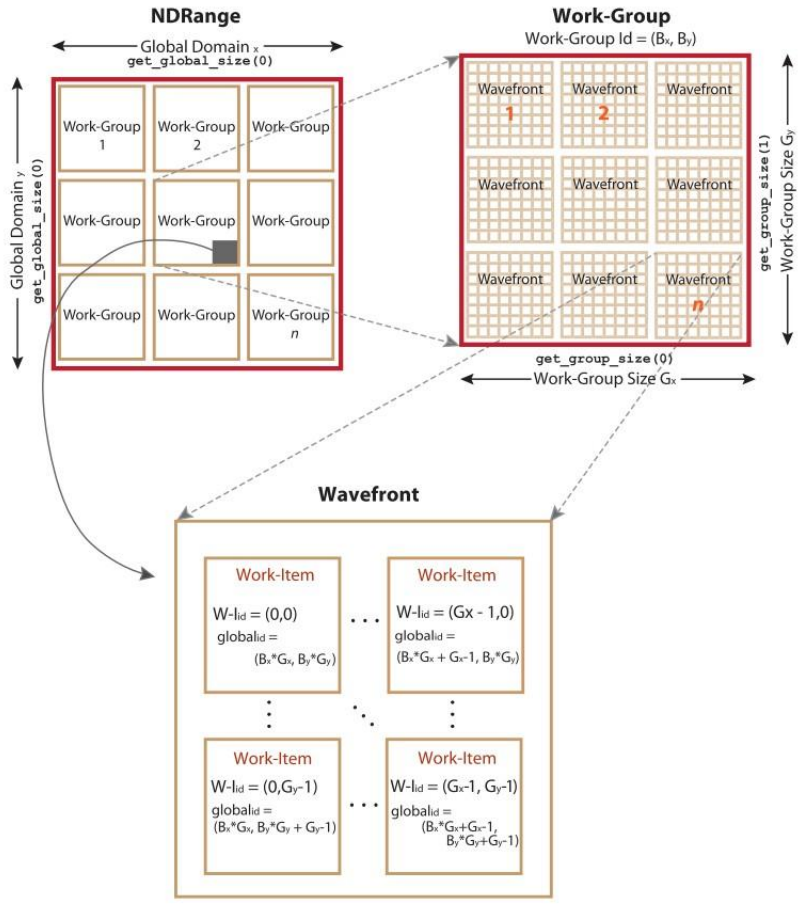

Figure 2-4 Relationship between work-items, wave-fronts and work-groups [9]

2.3 Inter-block Synchronization

In terms of inter-block synchronization, instead of going back to CPU to complete the synchronization between work-groups, algorithms are applied to GPU in order to perform communication between work-groups.

The program execution time includes kernel launch time, computation time and synchronization time. By theory, global synchronization on GPU instead of going back to CPU has the advantage of reducing the kernel launch time because the kernel is only launched once. However, the synchronization time might either enlarge or reduce the difference. Figure 2-5 illustrates the difference between perform global synchronization on GPU and perform global synchronization back on CPU.

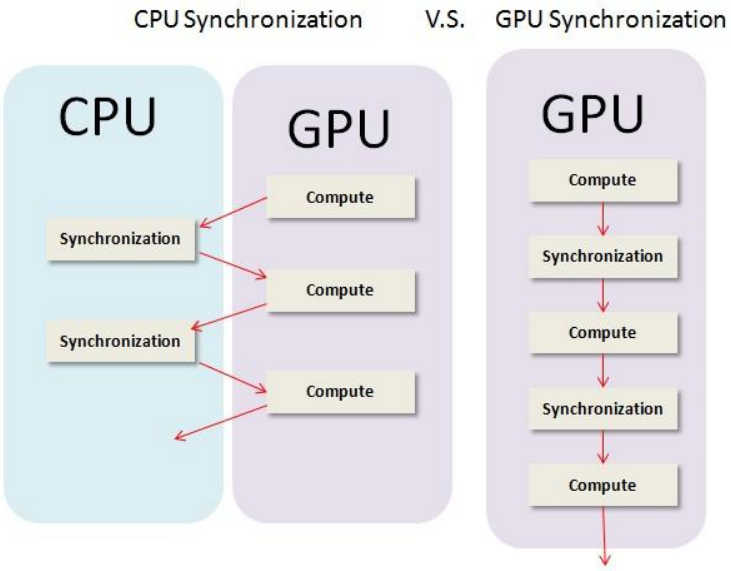

Figure 2-5 CPU synchronization and GPU synchronization

2.4 Previous Work

A student, Tao Wang, did a relevant project back in 2010 called "Synchronization between blocks on CUDA GPU" [5]. In his project, he implemented and tested three algorithms from Shicai Xiao and Wu-chun Feng's paper[1] using NVIDIA's CUDA.

He mentioned in his paper that, there would be a deadlock if the total number of work-groups launched exceeds the hardware limitation. In other words, those algorithms are not going to work if all launched work-groups cannot be executed together. Thus, resolving the deadlock was assigned as an additional task for this project. This problem is further discussed in the next chapter. Figure 2-6 illustrates the details of this deadlock problem.

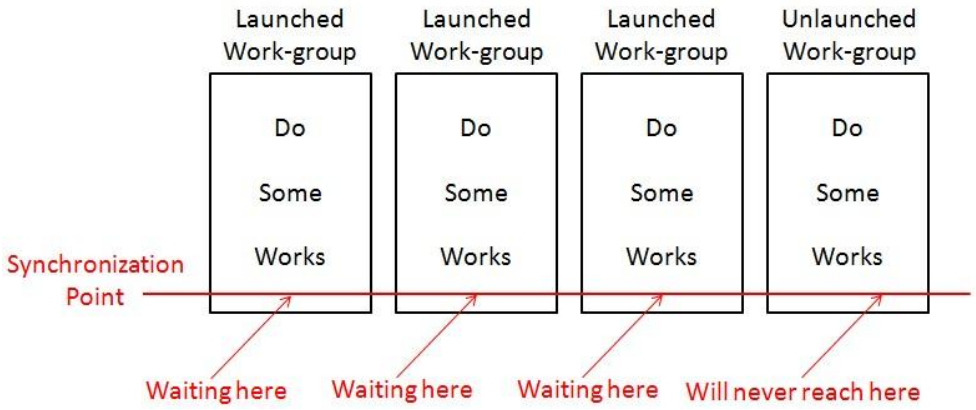

Figure 2-6 Deadlock in GPGPU inter-block synchronization

3. GPU Global Synchronizations

In this project, two barrier algorithms, the decentralized barrier and the centralized barrier algorithm based on an algorithm proposed in Shicai Xiao and Wu-chun Feng's paper [1] have been implemented. In addition, explicit CPU synchronization has been implemented in order to compare the performance with the other two algorithms. The implementations were done using AMD APP SDK in OpenCL. Moreover, an incomplete deadlock solution is also discussed in this chapter. These algorithms mentioned above are explained in detail in this chapter.

3.1 Synchronization using CPU

```
1 //CPU Explicit Synchronization
2 main(){
3         launch kernel
4         wait for kernel to finish and synchronize
5 }
```

Figure 3-1 Pseudo code of Explicit CPU Synchronization

Figure 3-1 shows the pseudo code of CPU explicit synchronization. Please refer to Appendix A for source code. In this algorithm, CPU keeps launching a kernel right after the previous one has been synchronized and has finished. In OpenCL, a kernel can be launched by calling clEnqueueNDRangeKernel() function and clFinish() call can be used to synchronize after a kernel has finished execution.

The pseudo code above in Figure 3-1 is coded in main function which will be executed by CPU. According to the pseudo code, the execution time T can be represented as Equation (1), where $T_{l,}$ is the kernel launch time, T_c is the computation time and T_{cs} is the synchronization time, i.e.

$$T = T_l + T_c + T_{cs} \qquad (1)$$

3.2 Synchronization using GPU

Different from using explicit support to perform synchronization back on CPU, synchronization on GPU requires making use of global memory and implementations of different algorithms to ensure proper communication and correct execution order between all working threads. Two different algorithms, decentralized and centralized barrier algorithm, are introduced in section 3.2.2 and 3.2.3 below.

3.2.1 Generic structure of inter-block synchronization

```
1 //generic GPU Sync Structure
2 __kernel void function_name(parameters){
3         do some computations
4         do local synchronization
5 }
```

Figure 3-2 generic structure of synchronization on GPU (pseudo code)

Figure 3-2 illustrates the generic structure of synchronization on GPU. Please refer to Appendix B for source code. This block of code is written in a file which is separate from the file that contains main function, and after compilation, the code in this file runs on GPU.

The execution time of the code above can be represented as Equation (2), where T_l is the kernel launch time, T_c is the computation time and T_{gs} is the time required to perform global synchronization.

$$T = T_l + T_c + T_{gs} \qquad (2)$$

Compared to CPU synchronization, the advantage to perform synchronization on GPU is that the kernel launch time is greatly reduced. And, whether synchronization time is able to enlarge or reduce the difference is going to be discussed in analysis section of this paper.

3.2.2 Centralized Barrier Algorithm

```
1  //Centralized Barrier Algorithm
2  void sync_(_global int goalVal){
3      do local synchronization
4      if
5          this is the master thread in a group
6      then
7          atomic decrease goalVal by 1
8          loop until goalVal become 0
9      end
10     do local synchronization
11 }
```

Figure 3-3 Pseudo code of Centralized Barrier Algorithm

Figure 3-3 above illustrates the pseudo code of Centralized Barrier Algorithm. In this approach, the first work-item of each work-group decreases a global variable by one atomically and then waits until the global variable reaches zero while the rest of the work-items are waiting at a local barrier. The global variable is initialized as the total number of work-groups, so that, when the variable reaches zero, it means that all work-items have reached this point and they are allowed to be synchronized. Please refer to Appendix C for source code.

According to pseudo code above, the total execution time consists of two parts, atomic operation and the time needed to wait until global variable reaches zero. The time consumed by atomic operation can be represented by $X*T_a$, where X is the total number of work-groups and T_a is the time needed for each of the atomic operations. We assume that the time needed for waiting is T_w, and then we get Equation (3) which calculates the synchronization time T_{gs} of Centralized Synchronization algorithm,

$$T_{gs} = X*T_a + T_w \qquad (3)$$

The advantages of this algorithm are, it is pretty simple, easy to understand, easy to implement and works well for all sorts of situations. However, the disadvantage of this approach is, it requires atomic operations which are very time consuming. From Equation (3) above, we can see that the atomic operation time plays a big part in the time calculation. Moreover, if this algorithm needs to be applied for multiple times, same approach has to be implemented again to reset the counter array which doubles the time. In the next section, an algorithm without using atomic algorithm is introduced.

3.2.3 Decentralized Barrier Algorithm

This algorithm resolves the problem in Centralized Barrier Algorithm. In order to get rid of atomic operations in the algorithm, modifying the same address in global memory must be prevented. To achieve this, an array of flags is introduced to reflect the execution status of each work-group. Moreover, it is very convenient of this algorithm that the flags are reset back to zero after synchronization and the function is ready-to-go again without any further operations.

Figure 3-4 illustrates the pseudo code of Decentralized Barrier Algorithm. In this approach, the first work-item of each work-group sets array[X] to one (X is the work-group number). Then the Jth work-item in the first work-group waits until the Jth position of array is set to one. After that, all work-items in the first work-group wait on a barrier to ensure all work-items in this group reach here. In the next step, the Jth work-item in the first group resets array[J] back to zero and all first work-items of each work-groups waits until array[X] is set to zero. In the end, all threads wait on a barrier to ensure proper communication. Please refer to Appendix D for source code.

```
1  //Decentralized Barrier Algorithm
2  void sync_(parameters){
3      do local synchronization
4      if
5          this is a master thread in each work-group
6      then
7          set array[group id] to 1
8      end
9      if
10         this is a thread in master thread
11     then
12         if
13             thread id of this thread is smaller than totoal number of groups
14         then
15             loop until array[thread id] is set to one
16         end
17         do local synchronization
18         if
19             thread id of this thread is smaller than totoal number of groups
20         then
21             reset array[thread id] to 0
22         end
23     end
24     if
25         this is a master thread in each work-group
26     then
27         loop until array[group id] is set to 0
28     end
29     do local synchronization
30 }
```

Figure 3-4 Pseudo code of Decentralized Barrier Algorithm

According to the pseudo above, the total execution time consists of the following 4 parts, the time needed to wait for the flag to be set to one and back to zero, T_{w1} and T_{w2}. As well as the time spent to set the flags, T_{f1} and T_{f2}. Then we get the Equation (4) to calculate the synchronization time T_{gs},

$$T_{gs} = T_{w1} + T_{w2} + T_{f1} + T_{f2} \qquad (4)$$

As we can see from the equation, after getting rid of atomic operation, the synchronization time become fixed value instead of an equation which is linearly increasing. Thus, by theory, synchronization time of Decentralized Barrier Algorithm should be shorter than that of Centralized Barrier Algorithm. The detailed comparison is to be shown in analysis section of this page.

The advantage of this algorithm is that it is relatively fast. However, compare to Centralized Barrier Algorithm, it is a little bit hard to understand and it only applies when the number of work-items is larger than the number of work-groups.

3.3 Resolve Deadlock Problem

The deadlock problem introduced in Tao Wang's paper [5] happens when the number of work-groups launched is larger than the actual number of work-groups that the hardware can handle. i.e. assume we have 10 work-groups launched, but the GPU is only able to process maximum 8 work-groups at the same time. When the kernel is launched, 8 executing work-groups wait at the synchronization point for the 2 work-groups that have not yet started. However, the 2 groups will never start since all compute units are occupied.

After analysed the problem, I decided to use one of the implemented inter-block synchronization algorithm to resolve the deadlock. Figure 3-5 illustrates the pseudo code of the implementation using Centralized Barrier Algorithm.

```
1  //Double Centralized Barrier
2  __kernel void function_name(parameters){
3      do some computations
4      do centralized barrier synchronization for the first X groups
5      loop through the flag array
6          if
7              flag array has an empty slot
8          then
9              set the flag to 1 atomically
10             do the work that is supposed to be done by the work-group
11             which associated with the flag
12         end
13     end
14     do centralized barrier synchronization for X groups
15 }
```

Figure 3-5 Pseudo code of the implementation to solve deadlock problem

In this approach, the first Centralized Barrier Algorithm at line 4 is used to make sure all running work-groups have finished their own jobs. From line 5 to line 13, all threads try to find out any unfinished job, set the associated flag atomically and do the job. At line 14, after all jobs are done, Centralized Barrier Algorithm is used again to make sure all work-groups have reached this point.

This approach works properly if the job that needs to be done is hard coded at line 10. I spent a lot of time trying to work out a way to find what job is actually assigned to a particular group and I did test a few methods. However, those tests failed and the project time constraint made me no choice but to give up researching in order to keep up with project schedule. Looking at the bright side, the idea is delivered and this can be a good starting point for the researchers who would like to explore further in this area. Please refer to Appendix E for source code.

4. Results and Analysis

4.1 Testing Strategies

In this project, three different inter-block synchronization methods are implemented and tested, they are,

- CPU Explicit Synchronization (CPU Sync)
- Centralized Barrier Synchronization (CB Sync)
- Decentralized Barrier Synchronization (DCB Sync)

Experiments are conducted from three perspectives to compare the performance of the above three algorithms,

- Time consumption based on different number of iterations looped
- Time consumption based on different number of work-groups used
- Memory usage on GPU based on different work-groups used

Figure 4-1 illustrates the pseudo code of the strategy used on CPU Sync to get time consumption based on different number of iterations looped. And Figure 4-2 shows the pseudo code of the strategy used on CB Sync and DCB Sync to get the time consumption based on different number of iterations looped. Please refer to Appendix F and Appendix G respectively for source code.

```
1 //CPU Explicit Synchronization Test
2 main(){
3     loop for N iterations
4         launch Kernel
5         wait for kernel to finish and synchronize
6     end
7 }
```

Figure 4-1 Pseudo code of CPU Sync test strategy

According to Figure 4-1, when CPU Sync needs to be used for more than one time, the kernel has to be launched by host for multiple times as well. Thus, by modifying Equation (1), we get Equation (5) which represents the amount of time needed to use CPU Sync for N times,

$$T = N * (T_l + T_c + T_{cs}) \qquad (5)$$

```
1 //generic GPU Sync Structure Test
2 __kernel void function_name(parameters){
3     loop for N iterations
4         do some computations
5         do local synchronization;
6     end
7 }
```

Figure 4-2 Pseudo code of GPU Sync test strategy

The pseudo code in Figure 4-2 is written as kernel code. Thus, no matter how many times the synchronization is needed, kernel is launched only once. This is one of the advantages of implementing global synchronization on GPU instead of implement it back on CPU. By modifying Equation (2), we get Equation (6) which calculates the amount of time needed to synchronize N times on GPU,

$$T = T_l + N * (T_c + T_{gs}) \qquad (6)$$

For testing purpose, the same computation is used for all tests. Figure 4-3 gives this computation.

```
for (int i=0;i<20000;i++){
    avg = (a+b)/2;
}
```

Figure 4-3 Computation for testing purpose

4.2 Performance based on number of work-groups

According to OpenCL Specification [10], by using cl_event variable type and clGetEventProfilingInfo function, we are able to get the values of two parameters, CL_PROFILING_COMMAND_START and CL_PROFILING_COMMAND_END. These two parameters are 64-bits values indicating the start time and end time of kernel execution in nanoseconds.

Figure 4-4 illustrates the time – workgroup relationship for the implemented synchronization algorithms. Please refer to Appendix H for experiment result. The times are measured using above parameters and synchronizations are performed once during each implementation. The x-axis represents the execution time in nanoseconds, and the y-axis represents the number of work-groups. Each time node is the average value of 10 test results under same circumstance.

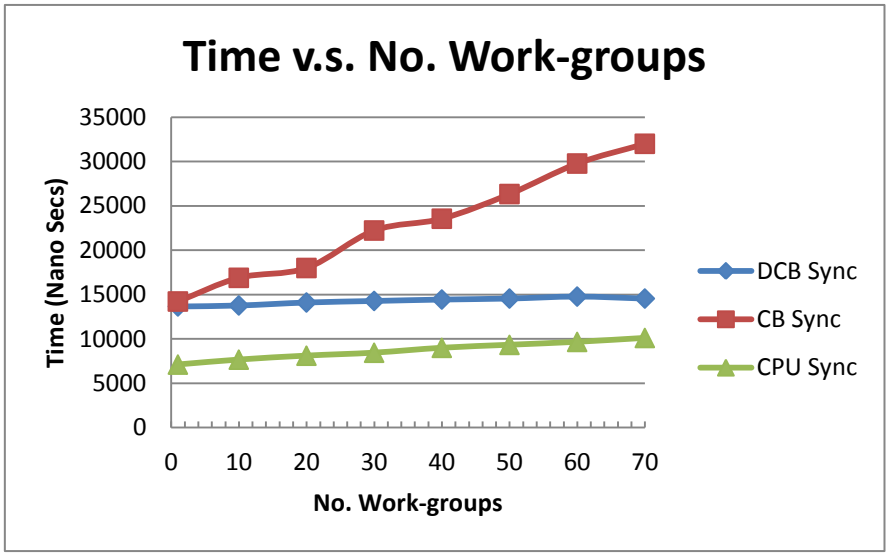

Figure 4-4 Time and Work-group relationship

As we can see from Figure 4-4, for single synchronization, CPU Sync has a lower starting point and the two GPU synchronizations have almost the same starting point. The execution time of DCB Sync remains stable at around 14000 nanoseconds. While the execution time of CPU Sync increases slowly and keeps approaching DCB Sync while the execution time of CB Sync increases significantly from 14000 nanoseconds. Generally speaking, for single synchronization, CPU Sync performs better than DCB Sync and CB Sync when the number of work-groups is lower than 70.

4.3 Performance based on number of iterations

From this point onwards, instead of using parameters to measure the timings, a Visual Studio plug-in application called AMD APP Profiler is used. This application is a performance analysis tool that gathers data from the OpenCL run-time and AMD Radeon GPUs during the execution of an OpenCL application [11]. This application can be used to discover bottlenecks in applications and find ways to optimize the application's performance for AMD platforms. It is easier to use, the results are clearer to read and, most importantly, the results are more accurate.

Figure 4-5 illustrates the time – iteration relationship of three implemented algorithms. However, since this figure is not clear enough to show the difference between DCB Sync and CB Sync, Figure 4-6 is used to make the differences easier to read. Please refer to Appendix I for experiment result. In this experiment, number of work-groups is fixed at 70 and number of iterations varies from 10 to 5000. And the times measured are in milliseconds (ms).

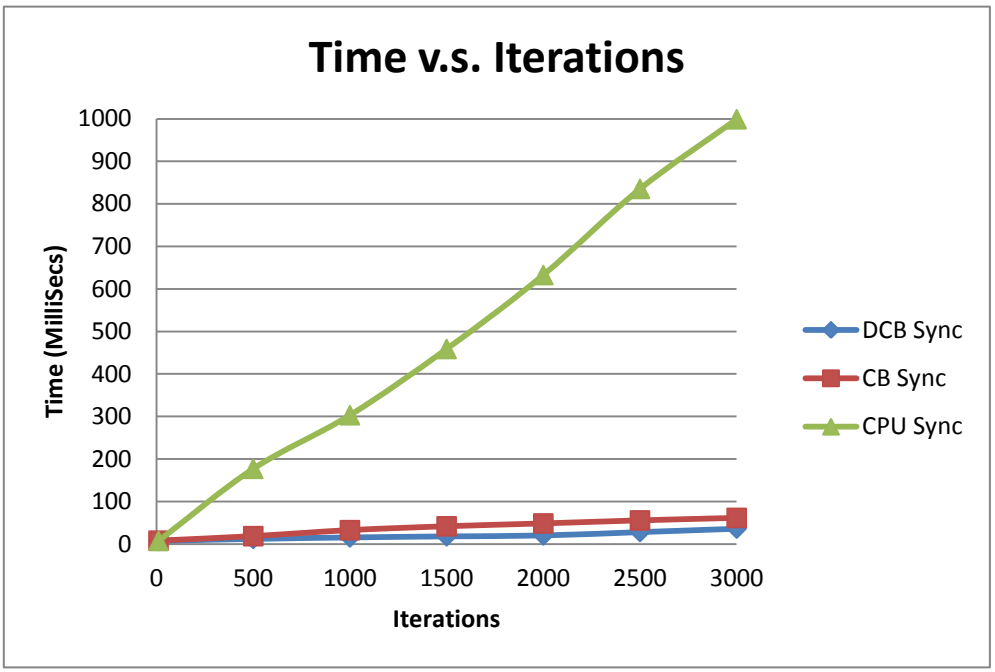

Figure 4-5 Time – Iteration relationship (3 algorithms)

In Figure 4-5, the implementations are tested with 10, 500, 1000, 1500, 2000, 2500 and 3000 iterations. Please refer to Figure 4-1 and Figure 4-2 for detailed testing strategies. With the increase of the number of iterations, the times of DCB Sync and CB Sync keep constant under 150ms. Although the time of CPU Sync is almost the same as the other two algorithms when there is no iteration, it shoots up substantially right after the starting point. Overall, the CPU Sync performs much worse than two GPU Synchronizations if it needs to be used for multiple times.

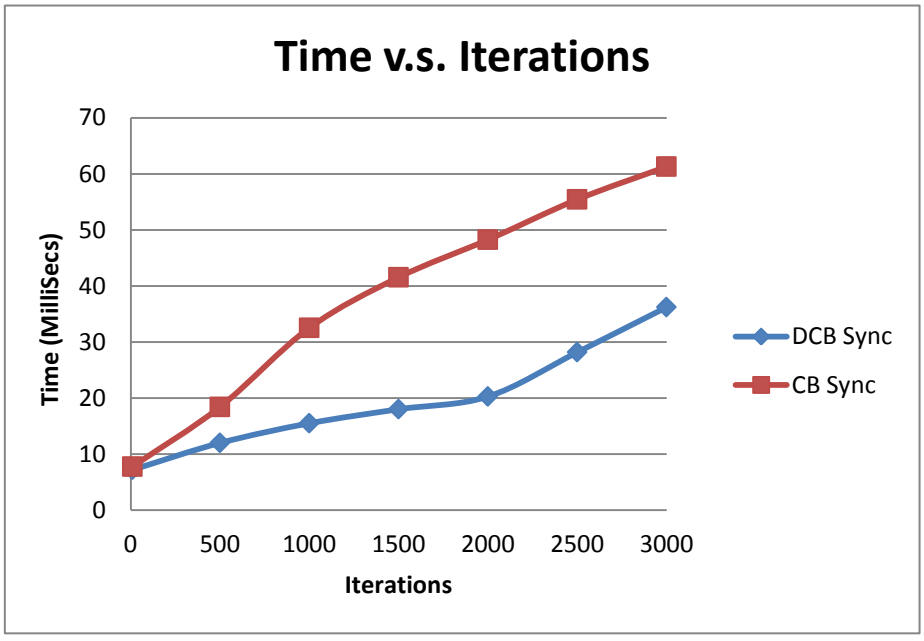

Figure 4-6 Time – Iteration relationship (2 algorithms)

In Figure 4-6, the lines for DCB Sync and CB Sync are both increasing. According to Equation (3) and Equation (4), DCB Sync contains atomic operation which is time consuming while CB does not. Not surprisingly, in the figure, the line representing DCB Sync has a lower increasing rate than the one representing CB Sync. Overall, DCB Sync performs better than CB Sync in all circumstances. (Except those circumstances that do not apply to DBC Sync)

4.4 Memory usage based on number of work-groups

The memory usages are compared for three implemented synchronizations. In this experiment, work-groups vary from 10 to 70 with a 10-groups gap between each experiment. Data is collected using AMD APP Profiler. Figure 4-7 illustrates the relationship between data usage and number of work-groups launched. Please refer to Appendix J for experiment result.

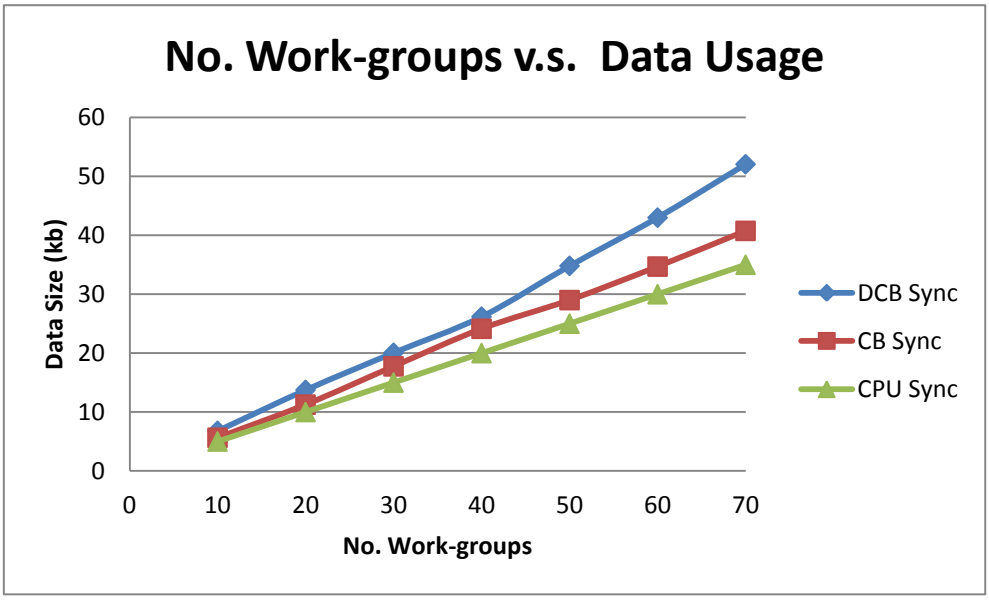

Figure 4-7 Data usage – number of work-groups relationship

In Figure 4-7, when the same number of work-groups is launched, the line representing DCB Sync is above CB Sync and the line representing CB Sync is always above CPU Sync. All in all, within these three inter-block synchronization algorithms, the faster the algorithm, the more data memory it uses.

5. Conclusion

5.1 Works Done

This project explored the area of performing inter-block synchronization on a GP-GPU using OpenCL. In this project, I implemented three existing inter-block synchronizations, CPU Explicit Synchronization (CPU Sync), Centralized Barrier Synchronization (CB Sync) and Decentralized Barrier Synchronization (DCB Sync). The performances are tested, compared and recorded. Within these three synchronization approaches, if the approach needs to be used for multiple times, CB Sync and DCB Sync perform much better than CPU Sync.

In general situations, DCB Sync works faster than CB Sync because CB Sync contains atomic operation and DCB Sync does not. However, the disadvantage of DCB Sync is that it only applies when the number of threads is larger than number of work-groups.

Overall, the purpose of implementing a GP-GPU inter-block synchronization is to ensure proper communication and proper communication between work-groups, so that the application is able to do more work accurately. Thus, the synchronization approach is very likely to be used for later work. Since there is a high chance to perform multiple synchronizations, GPU inter-block synchronizations (CB and DCB Sync) are recommended.

5.2 Future Work

5.2.1 More inter-block synchronization algorithms

From the experiment results, it is obvious that performing global synchronization is on GPU itself is much faster than do it back on CPU. Thus, possible topics of future researches in this area would be creating more GPGPU inter-block synchronization approaches.

5.2.2 Resolve the deadlock problem

As I discussed earlier, the deadlock problem has not been completely resolved. And, this problem greatly limited the application of the algorithms. Thus, another possible future research would be using this project as a starting point and get the deadlock problem explained in section 3.3 solved.

5.3 Lessons Learnt

- The biggest lesson I learnt was that I have to manage and balance my project time in order to keep up with the schedule. I could have resolved the deadlock problem if I had managed project time in a better way.

- Weekly meetings with Eric are held during the entire project, I gained some experience on project progress reporting to either supervisor or client.

- During workshops held by Weifa, I learnt how to write a quality research report and give a good project seminar.

Reference

[1] Shucai Xiao, Wu-chun Feng. GPU Communication via Fast Barrier Synchronization. In Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Atlanta, Georgia, USA, April 2010.

[2] Wikipedia. 2013. General-purpose computing on graphics processing units. http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units. [Accessed 01 October 13].

[3] AMD Brook+ Presentation. In SC07 BOF Session. November 2007.

[4] NVIDIA. 2006. What is CUDA. [ONLINE] Available at: https://developer.nvidia.com/what-cuda. [Accessed 31 October 13].

[5] Tao Wang. Synchronize between blocks on CUDA GPU. 2010. Canberra: The Australian National University.

[6] AMD, ATI Stream Computing OpenCL Programming Guide, 2010. (Fig. 1.1)

[7] AMD, AMD Accelerated Parallel Processing OpenCL programming Guide, 2012. (Fig. 1.8)

[8] AMD, AMD Accelerated Parallel Processing OpenCL programming Guide, 2012. (Fig. 1.9)

[9] AMD, ATI Stream Computing OpenCL Programming Guide, 2010. (Fig. 1.5)

[10] Aaftab Munshi. The OpenCL Specification Version: 1.2. June 2011.

[11] AMD. 2013. APP Profiler. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/archived-tools/amd-app-profiler/. [Accessed 31 October 13].

Appendices

Appendix A

```
err1 = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_work_size, &local_work_size, 0, NULL,&ceEvent);
if (err1 != CL_SUCCESS)
{
    printf("launch kernel error!\n");
}
```

Appendix B

```
//Do some computations
for (int i=0;i<20000;i++){
    avg = (a+b)/2;
}

//Synchronization
gpu_sync();
```

Appendix C

```
void gpu_sync(volatile __global int *counter){
    int thread_id = get_local_id(0);
    barrier(CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE);

    if(thread_id == 0){
        atom_dec(&counter[0]);
        while (counter[0]%70 != 0);
    }
    barrier(CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE);
}
```

Appendix D

```
void gpu_sync(volatile __global int *Flgs){
    const size_t thread_id = get_local_id(0);
    const size_t workgroup_id = get_group_id(0);

    if (thread_id == 0) {
        Flgs[workgroup_id] = 1;
    }

    if (workgroup_id == 0) {
        if (thread_id < get_num_groups(0)) {
            while (Flgs[thread_id] != 1) ;
        }
        barrier(CLK_GLOBAL_MEM_FENCE);

        if (thread_id < get_num_groups(0)) {
            Flgs[thread_id] = 0;
        }
    }

    if (thread_id == 0) {
        while (Flgs[workgroup_id] != 0) ;
    }
    barrier(CLK_GLOBAL_MEM_FENCE);
}
```

Appendix E

```
//Master thread doing global syn
if (thread_id == 0){
    atom_inc(&x[0]);
    if(goal_value == 70){
        while (x[0]<70 * (j+1));
    } else {
        while (x[0]<goal_value * (j+1));
    }
    atom_inc(&group_finish[group_id]);
}

barrier(CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE);
//group 0-70 has done their own job

//TODO - Do what is supposed to be done by next waiting group and finish it
//TODO - Loop until no more waiting group

//after all jobs are done, use master threads to syn again
if (thread_id == 0){
    atom_inc(&y[0]);
    if(goal_value == 70){
        while (y[0]<70 * (j+1));
    } else {
        while (y[0]<goal_value * (j+1));
    }
}

//Indicate the group has finished one task
if (thread_id == 0){
    atom_inc(&finish[0]);
}

barrier(CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE);
//group 0-70 have done the rest of the work in this loop at this point
```

Appendix F

```
for (int z=0;z<N;z++){
    err1 = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_work_size, &local_work_size, 0, NULL,&ceEvent);
    if (err1 != CL_SUCCESS)
    {
        printf("launch kernel error!\n");
    }
    clFinish( queue );
}
```

Appendix G

```
for (c=0;c<N;c++){
    //Do some computations
    for (int i=0;i<20000;i++){
        avg = (a+b)/2;
    }

    //Synchronization
    gpu_sync(counter);
}
```

Appendix H

No of groups	Time (Nanoseconds)		
	DCB Sync	CB Sync	CPU Sync
1	13666	14223	7111
10	13778	16889	7666
20	14111	18000	8111
30	14284	22223	8444
40	14444	23555	8998
50	14556	26334	9333
60	14778	29777	9667
70	14555	32000	10111

Appendix I

Iteration	Time(Milliseconds)		
	DCB Sync	CB Sync	CPU Sync
10	7.203	7.749	7.207
500	11.987	18.402	176.852
1000	15.495	32.518	302.903
1500	18.004	41.543	458.671
2000	20.267	48.254	632.585
2500	28.174	55.443	835.143
3000	36.216	61.3	999.043

Appendix J

No. of groups	Data size (KB)		
	DCB Sync	CB Sync	CPU Sync
10	6.81	5.63	5
20	13.75	11.25	10
30	20.06	17.75	15
40	26.19	24.13	20
50	34.81	29	25
60	43	34.69	30
70	52.06	40.75	35