Analysis of OpenCL on CUDA Platform

A BACHELOR'S THESIS

submitted in partial fulfillment

of the requirements for the award of the degree

of

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY
(B.Tech in IT)

Submitted by

Abhijeet Kumar Singh

IIT2006011

Under the Guidance of:

Dr. Pavan chakraborty
Associate Professor
IIIT-Allahabad

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY
ALLAHABAD – 211 012 (INDIA)

May, 2010

CANDIDATE'S DECLARATION

I hereby declare that the work presented in this thesis entitled "**Analysis Of OpenCL On CUDA Platform**", submitted in the partial fulfillment of the degree of Bachelor of Technology (B.Tech), in Information Technology at Indian Institute of Information Technology, Allahabad, is an authentic record of my original work carried out under the guidance of **Dr. Pavan Chakraborty** due acknowledgements have been made in the text of the thesis to all other material used. This thesis work was done in full compliance with the requirements and constraints of the prescribed curriculum.

Place: Allahabad
Date: 21-5-2010

Abhijeet kumar Singh
IIT2006011

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date: 21-5-2010
Place: Allahabad

Dr. Pavan Chakraborty
Associate Professor, IIITA

Committee on Final Examination for Evaluation of the Thesis

_____ _____

_____ _____

ACKNOWLEDGEMENTS

I would like to express my deep sense of gratitude and profound feeling of admiration to my guide, Dr. **Pavan Chakraborty**. Dr. **Pavan Chakraborty** has been an advisor and a well wisher to me. His guidance and knowledge have been invaluable. He let me make my choices and learn from my mistakes. It is his supervision which made this project complete. I am ever grateful for his patience and the time he devoted for my work.

I would like to thank my friend, Mayuri Gupta, Phd Scholar, NTNU Norway, for exposure to researches in the areas of High Scale Computation. Also I'm grateful to my parents and my well wishers who have supported me a lot and gave me strength to do this project.

Place: Allahabad
Date: 21/5/2010

Abhijeet Kumar Singh
B Tech Final Year, IIITA

ABSTRACT

Aim of this project is to study the new GPGPU platform **OpenCL** which has recently been launched. In Past three four years there hav been a huge development in the field of **High Scale Computation (HSC)**, particularly in the field of GPGU (General Purpose Graphical Processing Unit). OpenCL being platform independent is obviously expected to create more demand then the previously existing technology of CUDA, which can do computation only on CUDA compatible NVIDIA graphic cards (Platform Dependent). Ofcourse for non cuda, heterogeneous system OpenCL is the only option for high scaling. But in CUDA we have two option Opencl and CUDA. In this project I'm comparing the performance of the two on A Nvidia CUDA enabled system. And will try to Analyse the performance and will reach on the conclusion which which will suite the demand of High Scale computation better..

Table of Contents

1. Introduction

The whole project is based on OpenCL implementation for HSC and harnessing multiple thread computation on CUDA GPGPU. So before I go anyfurther I would like to make you All familiar with All these terms.

- **CUDA (Compute Unified Device Architecture**) is a parallel computing architecture owned by NVIDIA. It is the computing engine in NVIDIA's (GPUs) that is accessible to developers through industry standard programming languages. Developers use CUDA C (C with NVIDIA extensions), compiled through a Open64 C compiler to code algorithms for execution on the GPU. CUDA gives developers access to the native instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest NVIDIA GPUs efficiently become, like CPU, an open architectures. Moreover, GPUs have a parallel "multi-core" architecture, with each core capable of running hundreds of threads (say 'processes') simultaneously. If any application suits to this kind of architecture, the GPGPU CUDA can offer large performance benefits. For Example, In the virtual gaming, in addition to render graphics, GPUs are used in "game physics calculation" (effects like shades,leaves,crowd etc). CUDA provides both a low level and a higher level API (Application programming interface). CUDA works with all NVIDIA GPUs from the G8X series onwards, to the recently launched Fermi. Due to binary compatibility programs developed for the GeForce 8 series will also work on all future Nvidia video cards without modification.

- **OpenCL** :(**Open C**omputing **L**anguage) is a framework for writing programs that execute across multiple platforms (heterogeneous platforms) consisting of GPUs and Core CPUs. OpenCL includes a language called OpenCL (based upon C99) for writing *kernels* (functions executing on GPU devices) and APIs that are used for defining and then controlling the platforms. OpenCL gives any application access to the Graphical Processing Unit for non-graphical computing for the GPU which had previously been available for graphical applications only. OpenCL is managed by Khronos Group. NVIDIA and Apple have too ben involved in its development.

- **GPGPU**: (General Purpose Computing on Graphical Processing Unit) is the technique of using a GPU, which were for typicalling handling computer graphic computation, for performing computation in applications traditionally handled by the CPU. It is made possible by the addition of higher precision arithmetic and programmable stages to the rendering pipeline, which allows stream processing for software developers on non-graphics data..

1.1 Currently existing technologies

CUDA is the only existing technology for GPGPU. CUDA follows SIMD technology ie Singe Instruction Multiple Device. A huge pool of data are sent to device in the form of threads. Each threads contain data and the instruction of operation in it. Since all the threads contain the same instruction with unique threadId. The Idx is the key factor in deciding the processing of the data.

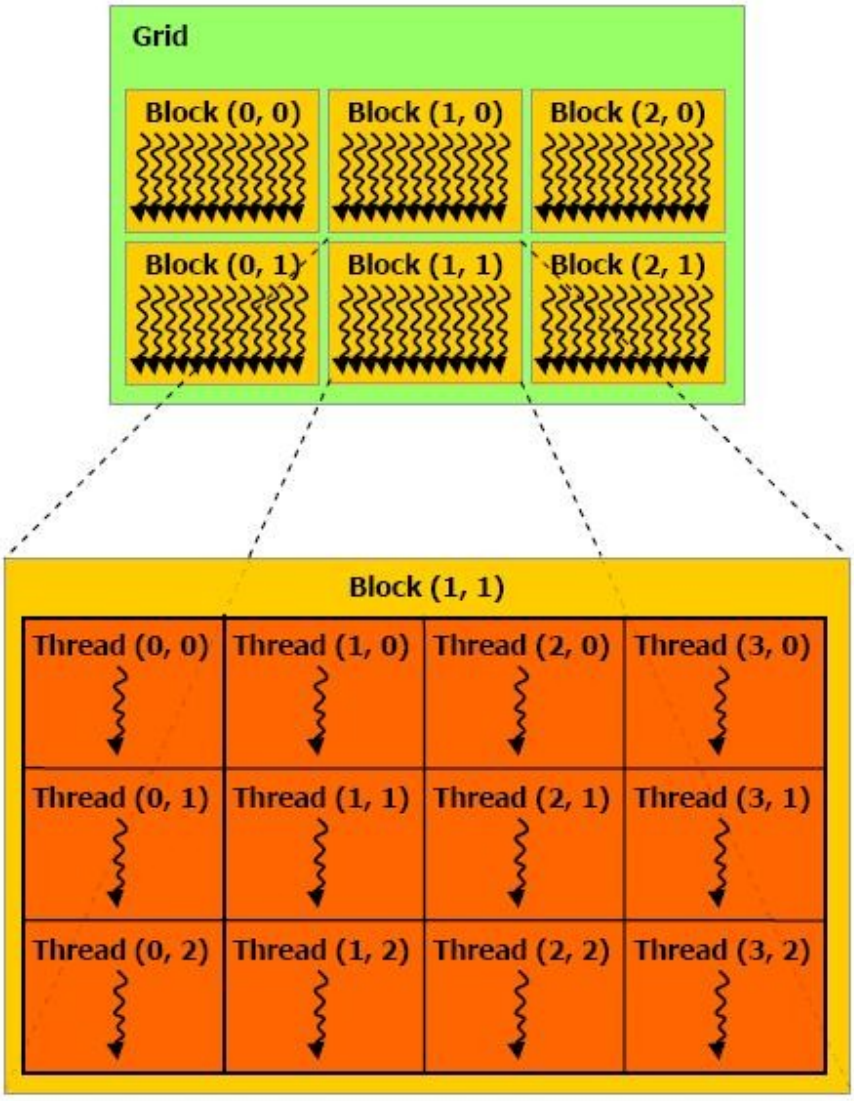
int idx = blockIdx.x * blockDim.x + threadIdx.x;

where blockIdx is the block Id. A block is the collection of thread which are executed on the single device processor. blockDim.x is the block dimension. It tells the size of the block which contains the threas. A collection of thread is called grid, which encompases all the threads. The threads in the block can communicate with each other. While the threads of two different cann't. The block has a shared memory which can be used for communication by the threads in the block.

The kernel function is invocated like this

prime_check <<< n_blocks, block_size >>> (a_d1,a_d, N); //kernel function call

where prime_check is the function.n_block is the number of block and block_size is the size of the block. Rest are the sent parameters.



1.2 An insight On OpenCL Technology

OpenCl is the heterogeneous API. It configure itself according to the device, Multi-CORE, (eg intel duo core) or Sigle CORE and Graphic devices(ATI, Nvidia). This is why called Context free.

In configuring itself for the particular device it calls for certain functions

1. ciErr1=clGetDeviceIDs(cpPlatform,CL_DEVICE_TYPE_GPU,1,&cdDevice,NULL);

```
2. cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);

3. cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);

4. cPathAndName = shrFindFilePath(cSourceFile, argv[0]);

5. cSourceCL = oclLoadProgSource(cPathAndName, "", &szKernelLength);

6. cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char **)&cSourceCL, &szKernelLength, &ciErr1);

7. ckKernel = clCreateKernel(cpProgram, "kernal function name", &ciErr1);

8. clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL, &szGlobalWorkSize, &szLocalWorkSize, 0, NULL, NULL);
```

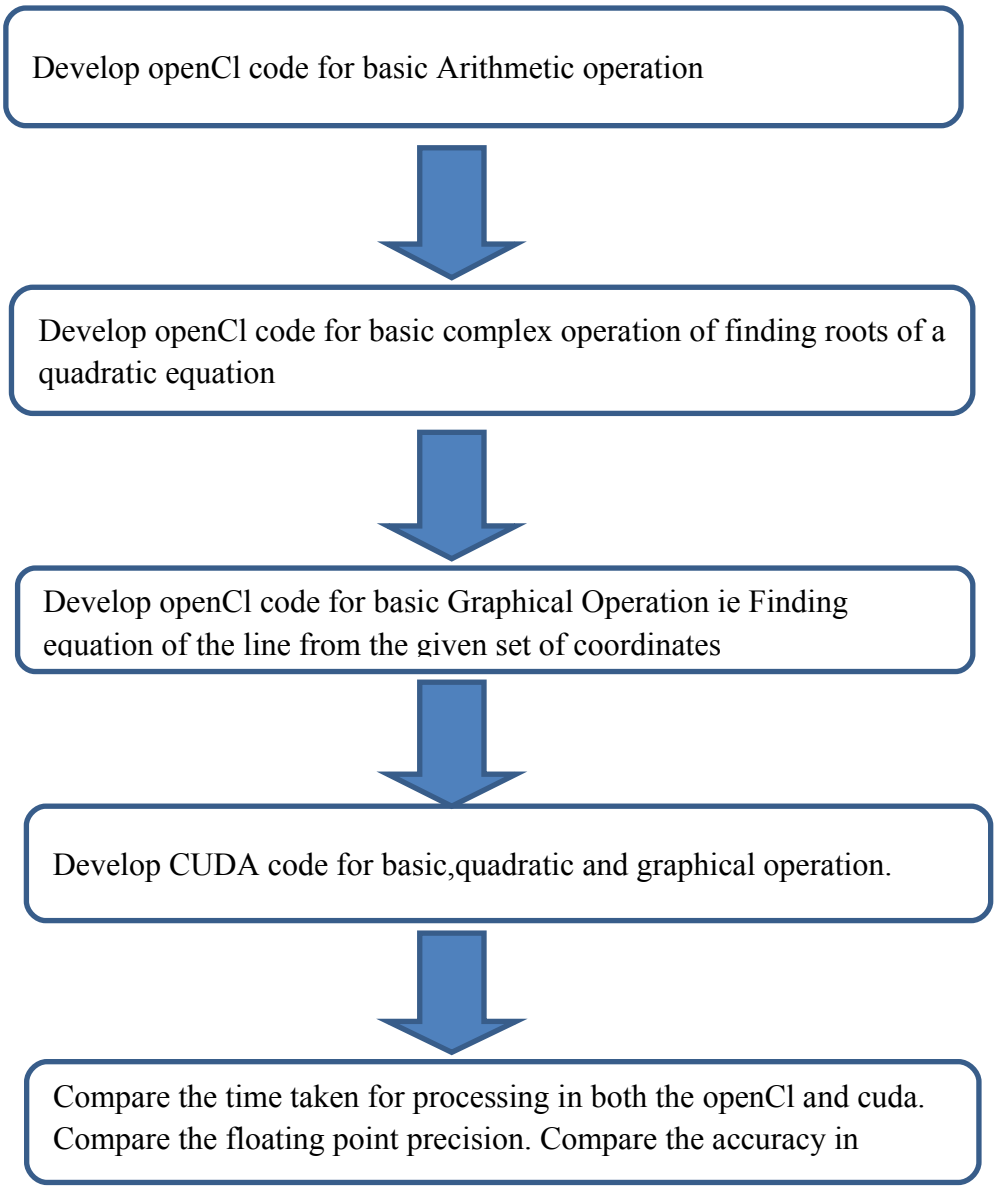
1.3 Problem definition and scope

Our objective is to Analyse Open Cl floating operation on CUDA. All the possible floating point arithmetic operation will be conducted on cuda with a large no of floating data. The Analysis will be on the three major metric.

1. The processing Speed: the amount of time taken for computation
2. The precision in floating point operation
3. The accuracy in floating point operation

1.4 Formulization of present problem

OpenCL analysis consist of the series of steps which are shown in the following flowchart given below:



1.5 Organization of the thesis

Till now we have discussed the currently existing technologies and algorithm. From now chapter 2 will discuss the hardware and software needed to complete this project and develop the software tool. After that chapter 3 will deal with theoretical analysis of the project and steps to follow in order to perform the analysis. In chapter 4 steps involved in development of software are discussed, getting multiple programs. After that chapter 5 deals with result and analysis of the software tool. In chapter 6 I have concluded the project giving the proper need and use of this software tool. Later in chapter 7, I have discussed the various subparts of my project where further work can be done and tool can be made much more efficient. At the end of the project thesis there is an appendix in which code of the project is explained.

2. Description of hardware and software used

While doing this project various hardware and software used are following:

1.1 Hardware:

a) NVIDIA Tesla GRAPHIC CARD. Model No. C870

b) Computer: I have developed and implemented this software using core2 duo processor with 3gb ram but it can be executed on any standard desktop computer.

1.2 Software :

a) Microsoft Visual Studio.

b) Open CL and CUDA SDK.

c) NVIDA CUDA DRIVER.

d) Platform: I have used windows platform for the development of the software tool.

3. Theoretical Tools – Analysis and Development

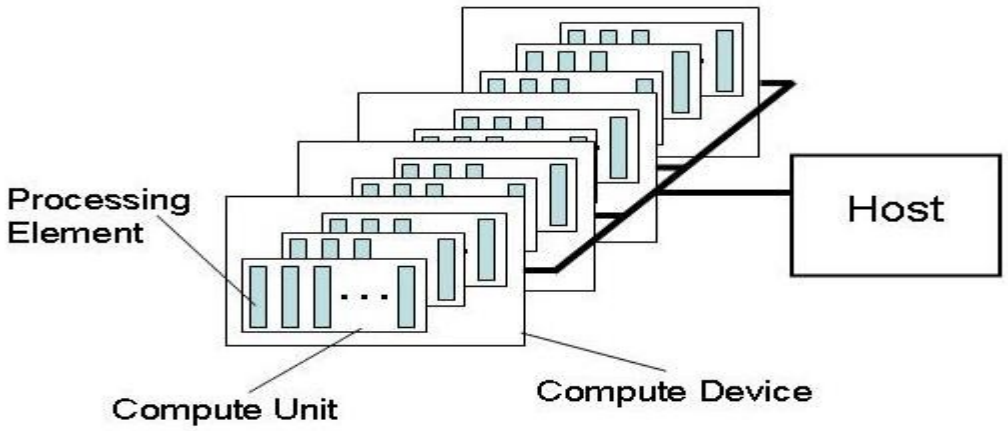
The OpenCL Architecture

OpenCL has these as hierarchy of models:

1. Platform Model
2. Memory Model
3. Exicution Model
4. Programming Model

Platform Model

one or more OpenCL devices are connected to host. An OpenCL device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements.



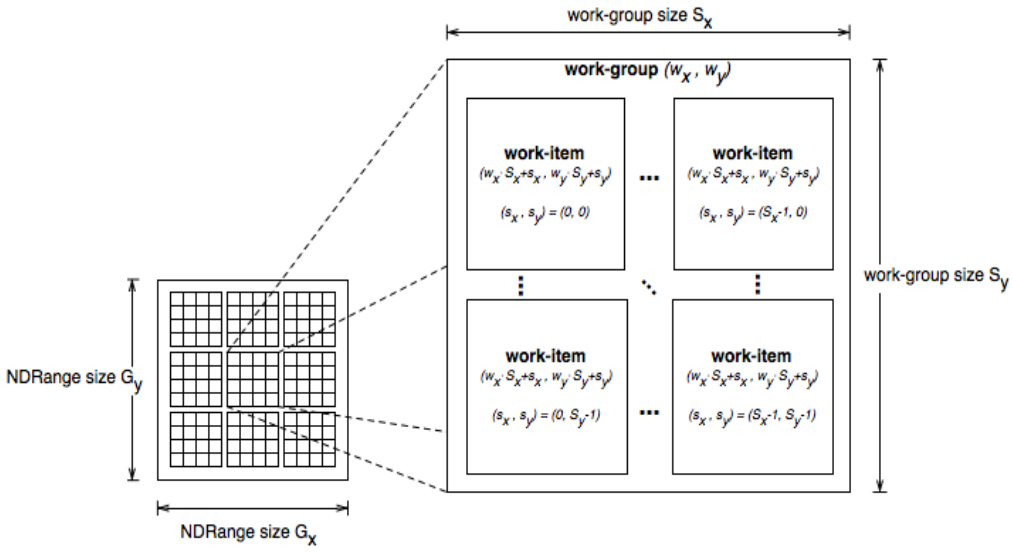
Exicution Model

Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the

context for the kernels and manages their execution. The core of the OpenCL execution model is defined by how the kernels execute. When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item..

The index space supported in OpenCL 1.0 is called an NDRange. An NDRange is an N-dimensional index space, where N is one, two or three. An NDRange is defined by an integer array of length N specifying the extent of the index space in each dimension. Each work-item's global ID and local ID are N-dimensional tuples. The global ID components are values in the range from zero to the number of elements in that dimension minus one.

Work-groups are assigned IDs using a similar approach to that used for work-item global IDs. An array of length N defines the number of work-groups in each dimension. Work-items are assigned to a work-group and given a local ID with components in the range from zero to the size of the work-group in that dimension minus one. Hence, the combination of a work-group ID and the local-ID within a work-group uniquely defines a work-item. Each work-item is identifiable in two ways; in terms of a global index, and in terms of a work-group index plus a local index within a work group.



Execution Model: Context and Command Queues

The host defines a context for the execution of the kernels. The context includes the following resources:

1. Devices: The collection of OpenCL devices to be used by the host.

2. Kernels: The OpenCL functions that run on OpenCL devices.

3. Program Objects: The program source and executable that implement the kernels.

4. Memory Objects: A set of memory objects visible to the host and the OpenCL devices.

Memory objects contain values that can be operated on by instances of a kernel. The context is created and manipulated by the host using functions from the OpenCL API. The host creates a data structure called a command-queue to coordinate execution of the kernels on the devices. The host places commands into the command-queue which are then scheduled onto the devices within the context.

These includes:-

Kernel execution commands: Execute a kernel on the processing elements of a device.

Memory commands: Transfer data to, from, or between memory objects, or map and unmap memory objects from the host address space.

Synchronization commands: Constrain the order of execution of commands.

Execution Model: Categories of Kernels

The OpenCL execution model supports two categories of kernels:

OpenCL kernels are written with the OpenCL C programming language and compiled with the OpenCL compiler. All OpenCL implementations support OpenCL kernels. Implementations may provide other mechanisms for creating OpenCL kernels.

The OpenCL Framework

The OpenCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

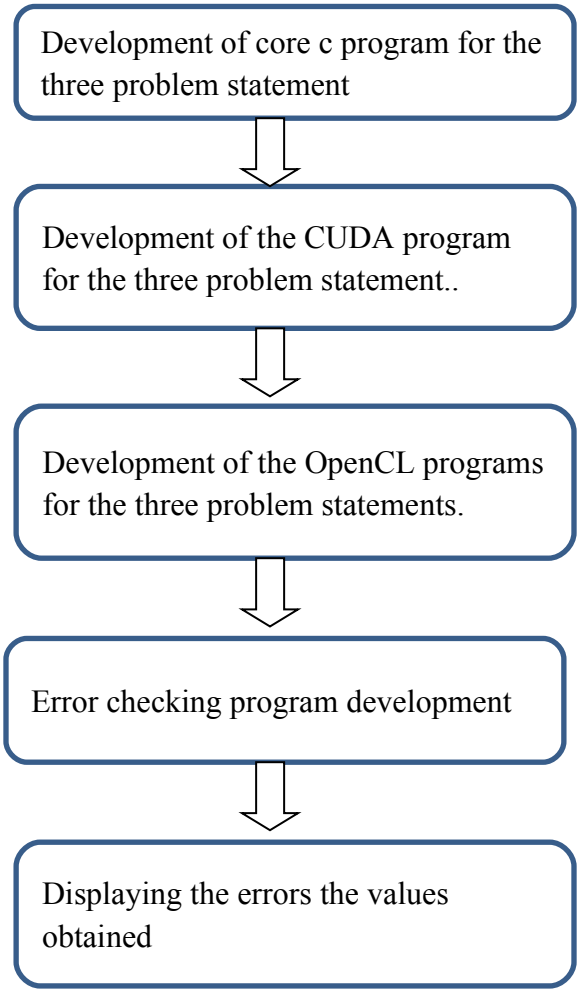
OpenCL Platform layer: The platform layer to discover OpenCL devices and their capabilities and then to create contexts.

OpenCL Runtime: The runtime allows the host program to manipulate contexts once they have been created.

OpenCL Compiler: The OpenCL compiler creates program executables that contain OpenCL kernels. The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism

4. Development of Software

4.1Steps involved in software development are as follow:



5. Testing and Analysis

Block SIZE - 256

Basic Operation

	CUDA			OPENCL		
No. of threads	Speed(ms)	Precision(%)	Accuracy(%)	Speed(ms)	Precision(%)	Accuracy(%)
11444777	1734	0.000003	71.76	3109	0.000002	76.47
11444000	1728	0.000003	72.34	3089	0.000002	79.03
1144400	1603	0.000002	76.29	2947	0.000001	85.78
114440	1473	0.000001	78.31	2789	0.000000	89.54

Quadratic operation

	CUDA			OPENCL		
No. of threads	Speed(ms)	Precision(%)	Accuracy(%)	Speed(ms)	Precision(%)	Accuracy(%)
11444777	2176	1.#nan00	15.6745%	3904	0.000004	57.7031%
11444000	2163	1.#nan00	17.8952%	3889	0.000003	59.0611%
1144400	1921	0.4345	18.3841%	3693	0.000002	67.8715%
114440	1834	0.3904	21.8342%	3367	0.000001	75.5631%

Coorinate operation

	CUDA			OPENCL		
No. of threads	Speed(ms)	Precision(%)	Accuracy(%)	Speed(ms)	Precision(%)	Accuracy(%)
11444777	2765	1.#nan00	57.3451%	3828	0.000003	62%
11444000	2695	1.#nan00	57.3451%	3748	0.000003	67%
1144400	2479	0.5936	57.3451%	3149	0.000002	73%
114440	2132	0.3852	57.3451%	2849	0.000001	77%

6. Conclusion

The above data on the performance of openCL on CUDA gives the following conlusion

- The openCl compiler is about 1.5 times slower then native CUDA compiler
- The floating point precession is higher in OpenCL then in CUDA by 2-3 times.
- For basic arithmetic operations precision and accuracy depends on operation for multiplication, division, log, sqrt, exponential etc the precision is low. While for addition or subtraction its high.
- On lower the No. of data the precision of both OpenCL and CUDA gets similar. While for large amount the difference in accuracy and precision becomes observable.
- For very low No. of data (in thousands) the processing Speed of both becomes close.

Thus for high precision computational operation on GPGPU platform with optimal processing speed OpenCL should be preferred and for low precision computational operation with high preference to processing speed CUDA should be preferred.

7. Recommendations and Future Work

Since OpenCL is heterogeneous. So definitely it will be supported on mass scale on the by the GPGPU community. But considering its limitation on processing speed on cuda platform with respect to pre-existing CUDA, the developer need to find the further improve the OpenCL compiler.

Thus future work will be analyzing the limitation of the OpenCL in computation speed. Obtaining the possible reason behind this limitation in it and trying to rectify it. It can be obtained by further optimize the programs written on this language or can be done by improving the design of its compiler.

Appendix - Explanation of the Source Code

CUDA CODE:

CODE for Basic Operation:

```
//#include "stdafx.h"
#include <stdio.h>      //Including the main input output headerfile
#include <cuda.h>             //Including the cuda header file
#include <time.h>                                                  //time in
processing
#include <cutil.h>
#include <math.h>

__global__ void float_operation( float *a,float *b,float *Mu,float *D,float
*P,float *Mi,int N)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < N)
    {Mu[i] = a[i] * b[i];
    D[i] = a[i] / b[i];
    P[i] = a[i] + b[i];
    Mi[i] = a[i] - b[i];}
}


void printFloatArray(float* sA,float* sB,float* sMul,float* sD,float*
sP,float* sMi, char* name, int length)
{
printf("%s:\n", name);
float mcnt=0.0f,dcnt=0.0f,pcnt=0.0f,micnt=0.0f;
for(int i=0;i<length;i++)
{
if(sA[i]*sB[i]>sMul[i])
mcnt=(mcnt) +(sA[i]*sB[i]-sMul[i])/(sA[i]*sB[i]);
else
mcnt=(mcnt) -(sA[i]*sB[i]-sMul[i])/(sA[i]*sB[i]);

if(sA[i]/sB[i]>sD[i])
dcnt=(dcnt)+(sA[i]/sB[i]-sD[i])/(sA[i]/sB[i]);
else
dcnt=(dcnt)-(sA[i]/sB[i]-sD[i])/(sA[i]/sB[i]);

if((sA[i]+sB[i])>sP[i])
```

```
pcnt=(pcnt)+((sA[i]+sB[i])-sP[i])/(sA[i]+sB[i]);
else
pcnt=(pcnt)-((sA[i]+sB[i])-sP[i])/(sA[i]+sB[i]);

if((sA[i]-sB[i])>sMi[i])
{if((sA[i]-sB[i])!=0)
micnt=micnt + ((sA[i]-sB[i])-sMi[i])/(sA[i]-sB[i]) ;
}
else
{if((sA[i]-sB[i])!=0)
micnt=micnt - ((sA[i]-sB[i])-sMi[i])/(sA[i]-sB[i]);
}
}
printf("precesion in multiplication %f\n",(mcnt * 100)/(float)length);
printf("precesion in devision %f\n",(dcnt* 100)/(float)length) ;
printf("precesion in addition %f\n",(pcnt* 100)/(float)length) ;
printf("precesion in subtraction %f\n",(micnt *100)/(float)length);
printf("\n\n");
}


clock_t EndTimer(clock_t begin)
{
    clock_t End;
    End = clock() * CLK_TCK;   //stop the timer
    return End;
}


#define N 11444777
#define block_size 256

int main(void)
{


  float *a_h,*b_h,*Mu_h,*D_h,*P_h,*Mi_h,*a_d,*b_d,*Mu_d,*D_d,*P_d,*Mi_d;


   int size = N*sizeof(float);


   a_h = ( float *)malloc(size);
   b_h = ( float *)malloc(size);
   Mu_h = ( float *)malloc(size);
   D_h = ( float *)malloc(size);
   P_h = ( float *)malloc(size);
   Mi_h = ( float *)malloc(size);

   cudaMalloc((void **) &a_d, size);
   cudaMalloc((void **) &b_d, size);
   cudaMalloc((void **) &Mu_d, size);
   cudaMalloc((void **) &D_d, size);
   cudaMalloc((void **) &P_d, size);
   cudaMalloc((void **) &Mi_d, size);

   for (int i=0; i<N; i++)
```

```
    {a_h[i]=(float)((rand()%100 + 1)*(1.3));b_h[i]=(float)((rand()%100
+1)*(1.7));}


  double begin = clock() * CLK_TCK;
  double elapTicks;




  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(Mu_d, Mu_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(D_d, D_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(P_d, P_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(Mi_d, Mi_h, size, cudaMemcpyHostToDevice);


  int n_blocks = N/block_size + 1;



  float_operation <<< n_blocks, block_size >>> (a_d,b_d,Mu_d,D_d,P_d,Mi_d,
N);

  cudaMemcpy(a_h, a_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(b_h, b_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(Mu_h, Mu_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(D_h, D_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(P_h, P_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(Mi_h, Mi_d, size, cudaMemcpyDeviceToHost);

  elapTicks = (double)EndTimer(begin);
  printf("Processing time: %f (ms) \n", elapTicks/1000);




     printFloatArray(a_h,b_h,Mu_h,D_h,P_h,Mi_h, "result",N);

  free(a_h);free(b_h);free(Mu_h);free(D_h);free(P_h);free(Mi_h);
  cudaFree(a_d);cudaFree(b_d); cudaFree(Mu_d);cudaFree(D_d);cudaFree(P_d);
cudaFree(Mi_d);

  system("pause");
}
```

The accuracy calculator program in basic operation of cuda

```
void printFloatArray(float* sA,float* sB,float* sMul,float* sD,float*
sP,float* sMi, char* name, int length)
{
printf("%s:\n", name);
int mcnt=0,dcnt=0,pcnt=0,micnt=0;
```

```
for(int i=0;i<length;i++)
{
if(sA[i]*sB[i]==sMul[i])
mcnt++;
if(sA[i]/sB[i]==sD[i])
dcnt++;
if(sA[i]+sB[i]==sP[i])
pcnt++;
if(sA[i]-sB[i]==sMi[i])
micnt++;
}
printf("precesion in multiplication %f\n",(mcnt*100)/(float)length);
printf("precesion in devision %f\n",(dcnt*100)/(float)length);
printf("precesion in addition %f\n",(pcnt*100)/(float)length);
printf("precesion in subtraction %f\n",(micnt*100)/(float)length);
printf("\n\n");
}
```

The Quadratic operation CODE for CUDA

```
//#include "stdafx.h"
#include <stdio.h>      //Including the main input output headerfile
#include <cuda.h>             //Including the cuda header file
#include <time.h>                                                  //time in
processing
#include <cutil.h>
#include <math.h>

// Kernel that executes on the CUDA device
__global__ void float_operation( float* a, float* b, float* c, float* d,
float* e, int iNumElements)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < iNumElements)
    {d[i] = (- b[i]+sqrt(double((b[i]*b[i])-(4*a[i]*c[i]))))/2*a[i];
    e[i] = (- b[i]+sqrt(double((b[i]*b[i])-(4*a[i]*c[i]))))/2*a[i];}
}


void printFloatArray(float* sA,float* sB,float* sC,float* sD,float* sP, char*
name, int length)
{
printf("%s:\n", name);
float dcnt=0.0f,pcnt=0.0f;float d, p;
for(int i=0;i<length;i++)
{
d=(- sB[i]+(float)sqrt(double((sB[i]*sB[i])-(4*sA[i]*sC[i]))))/2*sA[i];
p=(- sB[i]- (float)sqrt(double((sB[i]*sB[i])-(4*sA[i]*sC[i]))))/2*sA[i];
if(d>sD[i] && d!=0)
```

```
dcnt=dcnt+ ((d-sD[i])/d);
else
dcnt=dcnt-((d-sD[i])/d);

if(p>sP[i] && p!=0)
pcnt=pcnt+((p-sD[i])/p);
else
pcnt=pcnt-((p-sD[i])/p);

}
printf("precesion in 1st root %f\n",(dcnt*100)/(float)length) ;
printf("precesion in 2nd root %f\n",(pcnt* 100)/(float)length) ;

printf("\n\n");
}


clock_t EndTimer(clock_t begin)
{
    clock_t End;
    End = clock() * CLK_TCK;   //stop the timer
    return End;
}


#define N 11444777
#define block_size 256

int main(void)
{


  float *a_h,*b_h,*c_h,*D_h,*P_h,*a_d,*b_d,*c_d,*D_d,*P_d;



   int size = N*sizeof(float);


   a_h = ( float *)malloc(size);
   b_h = ( float *)malloc(size);
   c_h = ( float *)malloc(size);
   D_h = ( float *)malloc(size);
   P_h = ( float *)malloc(size);


   cudaMalloc((void **) &a_d, size);
   cudaMalloc((void **) &b_d, size);
   cudaMalloc((void **) &c_d, size);
   cudaMalloc((void **) &D_d, size);
   cudaMalloc((void **) &P_d, size);


   for (int i=0; i<N; i++)
    {a_h[i]=(float)((rand()%100)*(1.3));b_h[i]=(float)((rand()
%100)*(1.7));c_h[i]=(float)((rand()%100)*(1.9));}
```

```
  double begin = clock() * CLK_TCK;
  double elapTicks;


  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(c_d, c_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(D_d, D_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(P_d, P_h, size, cudaMemcpyHostToDevice);
  //cudaMemcpy(Mi_d, Mi_h, size, cudaMemcpyHostToDevice);


  int n_blocks = N/block_size + 1;



  float_operation <<< n_blocks, block_size >>> (a_d,b_d,c_d,D_d,P_d, N);

  cudaMemcpy(a_h, a_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(b_h, b_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(c_h, c_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(D_h, D_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(P_h, P_d, size, cudaMemcpyDeviceToHost);
  //cudaMemcpy(Mi_h, Mi_d, size, cudaMemcpyDeviceToHost);


 elapTicks = (double)EndTimer(begin);
 printf("Processing time: %f (ms) \n", elapTicks/1000);


     printFloatArray(a_h,b_h,c_h,D_h,P_h, "result",N);

  free(a_h);free(b_h);free(c_h);free(D_h);free(P_h);
  cudaFree(a_d);cudaFree(b_d); cudaFree(c_d);cudaFree(D_d);cudaFree(P_d);

  system("pause");
}
```

The Accuracy Calculator program

```
void printFloatArray(float* sA,float* sB,float* sC,float* sD,float* sP, char*
name, int length)
{
printf("%s:\n", name);
int dcnt=0,pcnt=0;float d, p;
for(int i=0;i<length;i++)
```

```
{
d=(- sB[i]+(float)sqrt(double((sB[i]*sB[i])-(4*sA[i]*sC[i]))))/2*sA[i];
p=(- sB[i]- (float)sqrt(double((sB[i]*sB[i])-(4*sA[i]*sC[i]))))/2*sA[i];
if(d==sD[i])
dcnt++;
if(p==sP[i])
pcnt++;
}
printf("Accuracy in 1st root %f\n",(dcnt*100)/(float)length) ;
printf("Accuracy in 2nd root %f\n",(pcnt* 100)/(float)length) ;

printf("\n\n");
}
```

The Coordinates operation on CUDA

```
//#include "stdafx.h"
#include <stdio.h>      //Including the main input output headerfile
#include <cuda.h>             //Including the cuda header file
#include <time.h>                                                  //time in
processing
#include <cutil.h>
#include <math.h>

// Kernel that executes on the CUDA device
__global__ void float_operation(  float* x1,  float* y1, float* x2, float*
y2, float* d,  float* e, int iNumElements)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < iNumElements)
    {d[i] = (y2[i]-y1[i])/(x2[i]-x1[i]);
    e[i] = y2[i]-(d[i]*x2[i]);}
}


void printFloatArray(float* sx1,float* sy1,float* sx2,float* sy2,float*
sD,float* sP, char* name, int length)
{
printf("%s:\n", name);
float dcnt=0.0f,pcnt=0.0f;float d, p;
for(int i=0;i<length;i++)
{
d=(sy2[i]-sy1[i])/(sx2[i]-sx1[i]);
p=sy2[i]-(d*sx2[i]);

if(d!=0)
{if(d>sD[i])
```

```
dcnt=dcnt+((d-sD[i])/d);
else
dcnt=dcnt-((d-sD[i])/d);
}

if(p!=0)
{if(p>sP[i])
pcnt=pcnt+((p-sP[i])/p);
else
pcnt=pcnt-((p-sP[i])/p);
}}
printf("precesion in slope %f\n",(dcnt*100)/(float)length) ;
printf("precesion in coefficient %f\n",(pcnt* 100)/(float)length) ;

printf("\n\n");
}

clock_t EndTimer(clock_t begin)
{
    clock_t End;
    End = clock() * CLK_TCK;   //stop the timer
    return End;
}



#define N 11444777
#define block_size 256

int main(void)
{


  float *x1_h,*y1_h,*x2_h,*y2_h,*D_h,*P_h,*x1_d,*y1_d,*x2_d,*y2_d,*D_d,*P_d;



   int size = N*sizeof(float);


   x1_h = ( float *)malloc(size);
   y1_h = ( float *)malloc(size);
   x2_h = ( float *)malloc(size);
   y2_h = ( float *)malloc(size);
   D_h = ( float *)malloc(size);
   P_h = ( float *)malloc(size);


   cudaMalloc((void **) &x1_d, size);
   cudaMalloc((void **) &y1_d, size);
   cudaMalloc((void **) &x2_d, size);
   cudaMalloc((void **) &y2_d, size);
   cudaMalloc((void **) &D_d, size);
   cudaMalloc((void **) &P_d, size);
```

```
   for (int i=0; i<N; i++)
    {
    x1_h[i]=(float)((rand()%100)*(1.3));y1_h[i]=(float)((rand()%100)*(1.7));
    x2_h[i]=(float)((rand()%100)*(1.1));y2_h[i]=(float)((rand()%100)*(1.9));
    }



  double begin = clock() * CLK_TCK;
  double elapTicks;


  cudaMemcpy(x1_d, x1_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(y1_d, y1_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(x2_d, x2_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(y2_d, y2_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(D_d, D_h, size, cudaMemcpyHostToDevice);
  cudaMemcpy(P_d, P_h, size, cudaMemcpyHostToDevice);
  //cudaMemcpy(Mi_d, Mi_h, size, cudaMemcpyHostToDevice);


  int n_blocks = N/block_size + 1;



  float_operation <<< n_blocks, block_size >>> (x1_d,y1_d,x2_d,y2_d,D_d,P_d,
N);

  cudaMemcpy(x1_h, x1_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(y1_h, y1_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(x2_h, x2_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(y2_h, y2_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(P_h, P_d, size, cudaMemcpyDeviceToHost);
  cudaMemcpy(D_h, D_d, size, cudaMemcpyDeviceToHost);
  //cudaMemcpy(Mi_h, Mi_d, size, cudaMemcpyDeviceToHost);


 elapTicks = (double)EndTimer(begin);
  printf("Processing time: %f (ms) \n", elapTicks/1000);


     printFloatArray(x1_h,y1_h,x2_h,y2_h,D_h,P_h, "result",N);

  free(x1_h);free(y1_h);free(x2_h);
  free(y2_h);free(D_h);free(P_h);
  cudaFree(x1_d);cudaFree(y1_d);
cudaFree(x2_d);cudaFree(y2_d);cudaFree(D_d);cudaFree(P_d);

  system("pause");
}
```

The ERROR CALCULATOR PROGRAM

```
void printFloatArray(float* sx1,float* sy1,float* sx2,float* sy2,float*
sD,float* sP, char* name, int length)
{
printf("%s:\n", name);
int dcnt=0,pcnt=0;float d, p;
for(int i=0;i<length;i++)
{
d=(sy2[i]-sy1[i])/(sx2[i]-sx1[i]);
p=sy2[i]-(d*sx2[i]);
if(d==sD[i])
dcnt++;
if(p==sP[i])
pcnt++;
}
printf("Accuracy in slope %f",(dcnt/(float)length) *100);
printf("Accuracy in coefficient %f",(pcnt/(float)length) * 100);

printf("\n\n");
  }
```

OPENCL CODE

```
#include <oclUtils.h>
#include <time.h>

const char* cSourceFile = "VectorAdd.cl";

void *sA, *sB, *sMul,*sD,*sP,*sMi;


cl_context cxGPUContext;
cl_command_queue cqCommandQueue;
cl_platform_id cpPlatform;
cl_device_id cdDevice;
cl_program cpProgram;
cl_kernel ckKernel;
cl_mem DevsA;
cl_mem DevsB;
cl_mem DevsMul;
cl_mem DevsD;
cl_mem DevsP;
cl_mem DevsMi;
size_t szGlobalWorkSize;
size_t szLocalWorkSize;
size_t szParmDataBytes;
size_t szKernelLength;
cl_int ciErr1, ciErr2;
```

```
char* cPathAndName = NULL;
char* cSourceCL = NULL;
void printFloatArray(float* sA,float* sB,float* sMul,float* sD,float*
sP,float* sMi, char* name, int length)
{
printf("%s:\n", name);
float mcnt=0.0f,dcnt=0.0f,pcnt=0.0f,micnt=0.0f;
for(int i=0;i<length;i++)
{
if(sA[i]!=0 && sB[i]!=0)
{if(sA[i]*sB[i]>sMul[i])
mcnt=(mcnt) +(sA[i]*sB[i]-sMul[i])/(sA[i]*sB[i]);
else
mcnt=(mcnt) -(sA[i]*sB[i]-sMul[i])/(sA[i]*sB[i]);
}

if(sA[i]!=0 && sB[i]!=0)
{if(sA[i]/sB[i]>sD[i])
dcnt=(dcnt)+(sA[i]/sB[i]-sD[i])/(sA[i]/sB[i]);
else
dcnt=(dcnt)-(sA[i]/sB[i]-sD[i])/(sA[i]/sB[i]);
}

if(sA[i]!=0 && sB[i]!=0)
{if((sA[i]+sB[i])>sP[i])
pcnt=(pcnt)+((sA[i]+sB[i])-sP[i])/(sA[i]+sB[i]);
else
pcnt=(pcnt)-((sA[i]+sB[i])-sP[i])/(sA[i]+sB[i]);
}
if(sA[i]!=0 && sB[i]!=0)
{if((sA[i]-sB[i])>sMi[i])
{if((sA[i]-sB[i])!=0)
micnt=micnt + ((sA[i]-sB[i])-sMi[i])/(sA[i]-sB[i]) ;
}
else
{if((sA[i]-sB[i])!=0)
micnt=micnt - ((sA[i]-sB[i])-sMi[i])/(sA[i]-sB[i]);
}
}
}
printf("precesion in multiplication %f\n",(mcnt * 100)/(float)length);
printf("precesion in devision %f\n",(dcnt* 100)/(float)length) ;
printf("precesion in addition %f\n",(pcnt* 100)/(float)length) ;
printf("precesion in subtraction %f\n",(micnt *100)/(float)length);
printf("\n\n");
}


int iNumElements = 11444777;
shrBOOL bNoPrompt = shrFALSE;


void Cleanup (int iExitCode);


clock_t EndTimer(clock_t begin)
{
```

```
    clock_t End;
    End = clock() * CLK_TCK;    //stop the timer
    return End;
}

int main(int argc, char **argv)
{

    szLocalWorkSize = 256;
    szGlobalWorkSize = shrRoundUp((int)szLocalWorkSize, iNumElements);

    sA = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
    sB = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
    sMul = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
      sD = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
      sP = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
    sMi  = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
        shrFillArray((float*)sA, iNumElements);
    shrFillArray((float*)sB, iNumElements);


       double begin = clock() * CLK_TCK;
     double elapTicks;

      ciErr1 = clGetPlatformIDs(1, &cpPlatform, NULL);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clGetPlatformID, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }



    //Get the devices
    ciErr1 = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice,
NULL);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clGetDeviceIDs, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    //Create the context
    cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateContext, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Create a command-queue
    cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
    if (ciErr1 != CL_SUCCESS)
    {
```

```
        shrLog("Error in clCreateCommandQueue, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }


    DevsA = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) *
szGlobalWorkSize, NULL, &ciErr1);
    DevsB = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) *
szGlobalWorkSize, NULL, &ciErr2);
    ciErr1 |= ciErr2;
    DevsMul = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,
sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr2);
    ciErr1 |= ciErr2;
    DevsD = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, sizeof(cl_float)
* szGlobalWorkSize, NULL, &ciErr2);
    ciErr1 |= ciErr2;
      DevsP = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,
sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr2);
    ciErr1 |= ciErr2;
      DevsMi = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,
sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr2);
    ciErr1 |= ciErr2;

    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateBuffer, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Read the OpenCL kernel in from source file
    cPathAndName = shrFindFilePath(cSourceFile, argv[0]);
    cSourceCL = oclLoadProgSource(cPathAndName, "", &szKernelLength);

    // Create the program
    cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char
**)&cSourceCL, &szKernelLength, &ciErr1);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateProgramWithSource, Line %u in file %s !!!\n\
n", __LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Build the program with 'mad' Optimization option
    #ifdef MAC
        char* flags = "-cl-fast-relaxed-math -DMAC";
    #else
        char* flags = "-cl-fast-relaxed-math";
    #endif
    ciErr1 = clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clBuildProgram, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
```

```
    }

    // Create the kernel
    ckKernel = clCreateKernel(cpProgram, "VectorAdd", &ciErr1);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateKernel, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Set the Argument values
    ciErr1 = clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&DevsA);
    ciErr1 |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&DevsB);
    ciErr1 |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&DevsMul);
      ciErr1 |= clSetKernelArg(ckKernel, 3, sizeof(cl_mem), (void*)&DevsD);
      ciErr1 |= clSetKernelArg(ckKernel, 4, sizeof(cl_mem), (void*)&DevsP);
      ciErr1 |= clSetKernelArg(ckKernel, 5, sizeof(cl_mem), (void*)&DevsMi);
    ciErr1 |= clSetKernelArg(ckKernel, 6, sizeof(cl_int),
(void*)&iNumElements);

    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clSetKernelArg, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }


    ciErr1 = clEnqueueWriteBuffer(cqCommandQueue, DevsA, CL_FALSE, 0,
sizeof(cl_float) * szGlobalWorkSize, sA, 0, NULL, NULL);
    ciErr1 |= clEnqueueWriteBuffer(cqCommandQueue, DevsB, CL_FALSE, 0,
sizeof(cl_float) * szGlobalWorkSize, sB, 0, NULL, NULL);

    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clEnqueueWriteBuffer, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Launch kernel
    ciErr1 = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL,
&szGlobalWorkSize, &szLocalWorkSize, 0, NULL, NULL);

    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clEnqueueNDRangeKernel, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Synchronous/blocking read of results, and check accumulated errors
    ciErr1 = clEnqueueReadBuffer(cqCommandQueue, DevsMul, CL_TRUE, 0,
sizeof(cl_float) * szGlobalWorkSize, sMul, 0, NULL, NULL);
    ciErr1 = clEnqueueReadBuffer(cqCommandQueue, DevsD, CL_TRUE, 0,
sizeof(cl_float) * szGlobalWorkSize, sD, 0, NULL, NULL);
```

```
      ciErr1 = clEnqueueReadBuffer(cqCommandQueue, DevsP, CL_TRUE, 0,
sizeof(cl_float) * szGlobalWorkSize, sP, 0, NULL, NULL);
      ciErr1 = clEnqueueReadBuffer(cqCommandQueue, DevsMi, CL_TRUE, 0,
sizeof(cl_float) * szGlobalWorkSize, sMi, 0, NULL, NULL);

    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clEnqueueReadBuffer, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

      elapTicks = (double)EndTimer(begin);
      printf("Processing time: %f (ms) \n", elapTicks/1000);

      printFloatArray((float *)sA,(float *)sB,(float *)sMul,(float *)sD,(float
*)sP,(float *)sMi, "Result", iNumElements);


    Cleanup (EXIT_SUCCESS);
}

void Cleanup (int iExitCode)
{
    // Cleanup allocated objects

    if(cPathAndName)free(cPathAndName);
    if(cSourceCL)free(cSourceCL);
      if(ckKernel)clReleaseKernel(ckKernel);
    if(cpProgram)clReleaseProgram(cpProgram);
    if(cqCommandQueue)clReleaseCommandQueue(cqCommandQueue);
    if(cxGPUContext)clReleaseContext(cxGPUContext);
    if(DevsA)clReleaseMemObject(DevsA);
    if(DevsB)clReleaseMemObject(DevsB);
    if(DevsMul)clReleaseMemObject(DevsMul);
     if(DevsD)clReleaseMemObject(DevsD);
       if(DevsP)clReleaseMemObject(DevsP);
       if(DevsMi)clReleaseMemObject(DevsMi);
       //if(DevsA)clReleaseMemObject(DevsA);
    // Free host memory
    free(sA);
    free(sB);
    free (sMul);
      free (sD);
      free (sP);
      free (sMi);
    //free(Golden);

    // finalize logs and leave
    if (bNoPrompt)
    {
        shrLogEx(LOGBOTH | CLOSELOG, 0, "oclVectorAdd.exe Exiting...\n");
    }
    else
    {
        shrLogEx(LOGBOTH | CLOSELOG, 0, "oclVectorAdd.exe Exiting...\nPress
<Enter> to Quit\n");
```

```
        getchar();
    }
    exit (iExitCode);
}
```

Opencl Program for Quadratic operation

```
#include <oclUtils.h>
#include <time.h>
#include <math.h>
// Name of the file with the source code for the computation kernel
// *****************************************************************
const char* cSourceFile = "VectorAdd.cl";

// Host buffers for demo
// *****************************************************************
void *sA, *sB, *sC,*sD,*sP;        // Host buffers for OpenCL test

cl_context cxGPUContext;
cl_command_queue cqCommandQueue
cl_platform_id cpPlatform;
cl_device_id cdDevice;
cl_program cpProgram;
cl_kernel ckKernel;
cl_mem DevsA;
cl_mem DevsB;
cl_mem DevsC;
cl_mem DevsD;
cl_mem DevsP;
size_t szGlobalWorkSize;
size_t szLocalWorkSize;
size_t szParmDataBytes;
size_t szKernelLength;
cl_int ciErr1, ciErr2;
char* cPathAndName = NULL;
char* cSourceCL = NULL;

void printFloatArray(float* sA,float* sB,float* sC,float* sD,float* sP, char*
name, int length)
{
printf("%s:\n", name);
float dcnt=0,pcnt=0;float d, p;
for(int i=0;i<length;i++)
{
d=(- sB[i]+(float)sqrt(double((sB[i]*sB[i])-(4*sA[i]*sC[i]))))/2*sA[i];
p=(- sB[i]- (float)sqrt(double((sB[i]*sB[i])-(4*sA[i]*sC[i]))))/2*sA[i];
```

```
if(d!=0 && sA[i]!=0)
{if(d>sD[i])
dcnt=(dcnt)+ (d-sD[i])/d;
else
dcnt=(dcnt)+ (d-sD[i])/d;}

if(p!=0 && sA[i]!=0)
{if(p>sP[i])
pcnt=(pcnt)+(p-sD[i])/p;
else
pcnt=(pcnt)+(p-sD[i])/p;
}
}
printf("precesion in 1st root %f\n",(dcnt/(float)length) *100);
printf("precesion in 2nd root %f\n",(pcnt/(float)length) * 100);

printf("\n\n");
}


// demo config vars
int iNumElements = 11444777;
shrBOOL bNoPrompt = shrFALSE;

void Cleanup (int iExitCode);


clock_t EndTimer(clock_t begin)
{
    clock_t End;
    End = clock() * CLK_TCK;   //stop the timer
    return End;
}

int main(int argc, char **argv)
{

    szLocalWorkSize = 256;
    szGlobalWorkSize = shrRoundUp((int)szLocalWorkSize, iNumElements);
shrLog("Global Work Size \t\t= %u\nLocal Work Size \t\t= %u\n# of Work Groups
\t\t= %u\n\n",
           szGlobalWorkSize, szLocalWorkSize, (szGlobalWorkSize %
szLocalWorkSize + szGlobalWorkSize/szLocalWorkSize));


    shrLog( "Allocate and Init Host Mem...\n");
    sA = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
    sB = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
    sC = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
      sD = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
      sP = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);

    shrFillArray((float*)sA, iNumElements);
    shrFillArray((float*)sB, iNumElements);
      shrFillArray((float*)sC, iNumElements);

    double begin = clock() * CLK_TCK;
```

```
     double elapTicks;

      ciErr1 = clGetPlatformIDs(1, &cpPlatform, NULL);

    shrLog("clGetPlatformID...\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clGetPlatformID, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }


    ciErr1 = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice,
NULL);
    shrLog("clGetDeviceIDs...\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clGetDeviceIDs, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

        cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
    shrLog("clCreateContext...\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateContext, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }


    cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
    shrLog("clCreateCommandQueue...\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateCommandQueue, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }


    DevsA = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) *
szGlobalWorkSize, NULL, &ciErr1);
      DevsB = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float)
* szGlobalWorkSize, NULL, &ciErr2);
    DevsC = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) *
szGlobalWorkSize, NULL, &ciErr1);
    DevsD = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, sizeof(cl_float)
* szGlobalWorkSize, NULL, &ciErr2);
    DevsP = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, sizeof(cl_float)
* szGlobalWorkSize, NULL, &ciErr1);


    shrLog("clCreateBuffer...\n");
    if (ciErr1 != CL_SUCCESS)
```

```
    {
        shrLog("Error in clCreateBuffer, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Read the OpenCL kernel in from source file
    shrLog("oclLoadProgSource (%s)...\n", cSourceFile);
    cPathAndName = shrFindFilePath(cSourceFile, argv[0]);
    cSourceCL = oclLoadProgSource(cPathAndName, "", &szKernelLength);

    // Create the program
    cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char
**)&cSourceCL, &szKernelLength, &ciErr1);
    shrLog("clCreateProgramWithSource...\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateProgramWithSource, Line %u in file %s !!!\n\
n", __LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Build the program with 'mad' Optimization option
    #ifdef MAC
        char* flags = "-cl-fast-relaxed-math -DMAC";
    #else
        char* flags = "-cl-fast-relaxed-math";
    #endif
    ciErr1 = clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
    shrLog("clBuildProgram...\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clBuildProgram, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Create the kernel
    ckKernel = clCreateKernel(cpProgram, "VectorAdd", &ciErr1);
    shrLog("clCreateKernel (VectorAdd)...\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateKernel, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Set the Argument values
    ciErr1 = clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&DevsA);
    ciErr1 |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&DevsB);
    ciErr1 |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&DevsC);
      ciErr1 |= clSetKernelArg(ckKernel, 3, sizeof(cl_mem), (void*)&DevsD);
      ciErr1 |= clSetKernelArg(ckKernel, 4, sizeof(cl_mem), (void*)&DevsP);
    ciErr1 |= clSetKernelArg(ckKernel, 5, sizeof(cl_int),
(void*)&iNumElements);
    shrLog("clSetKernelArg 0 - 6...\n\n");
    if (ciErr1 != CL_SUCCESS)
```

```
    {
        shrLog("Error in clSetKernelArg, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }


    ciErr1 = clEnqueueWriteBuffer(cqCommandQueue, DevsA, CL_FALSE, 0,
sizeof(cl_float) * szGlobalWorkSize, sA, 0, NULL, NULL);
    ciErr1 |= clEnqueueWriteBuffer(cqCommandQueue, DevsB, CL_FALSE, 0,
sizeof(cl_float) * szGlobalWorkSize, sB, 0, NULL, NULL);
      ciErr1 |= clEnqueueWriteBuffer(cqCommandQueue, DevsC, CL_FALSE, 0,
sizeof(cl_float) * szGlobalWorkSize, sC, 0, NULL, NULL);
    shrLog("clEnqueueWriteBuffer (SrcA and SrcB)...\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clEnqueueWriteBuffer, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Launch kernel
    ciErr1 = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL,
&szGlobalWorkSize, &szLocalWorkSize, 0, NULL, NULL);
    shrLog("clEnqueueNDRangeKernel (VectorAdd)...\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clEnqueueNDRangeKernel, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Synchronous/blocking read of results, and check accumulated errors

    ciErr1 = clEnqueueReadBuffer(cqCommandQueue, DevsD, CL_TRUE, 0,
sizeof(cl_float) * szGlobalWorkSize, sD, 0, NULL, NULL);
      ciErr1 = clEnqueueReadBuffer(cqCommandQueue, DevsP, CL_TRUE, 0,
sizeof(cl_float) * szGlobalWorkSize, sP, 0, NULL, NULL);

      shrLog("clEnqueueReadBuffer (Dst)...\n\n");
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clEnqueueReadBuffer, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }
   elapTicks = (double)EndTimer(begin);
  printf("Processing time: %f (ms) \n", elapTicks/1000);


    printFloatArray((float *)sA,(float *)sB,(float *)sC,(float *)sD,(float
*)sP, "Result", iNumElements);


    Cleanup (EXIT_SUCCESS);
}
```

```
void Cleanup (int iExitCode)
{
    // Cleanup allocated objects
    shrLog("Starting Cleanup...\n\n");
    if(cPathAndName)free(cPathAndName);
    if(cSourceCL)free(cSourceCL);
      if(ckKernel)clReleaseKernel(ckKernel);
    if(cpProgram)clReleaseProgram(cpProgram);
    if(cqCommandQueue)clReleaseCommandQueue(cqCommandQueue);
    if(cxGPUContext)clReleaseContext(cxGPUContext);
    if(DevsA)clReleaseMemObject(DevsA);
    if(DevsB)clReleaseMemObject(DevsB);
    if(DevsC)clReleaseMemObject(DevsC);
     if(DevsD)clReleaseMemObject(DevsD);
       if(DevsP)clReleaseMemObject(DevsP);

    free(sA);
    free(sB);
    free (sC);
      free (sD);
      free (sP);

    if (bNoPrompt)
    {
        shrLogEx(LOGBOTH | CLOSELOG, 0, "oclVectorAdd.exe Exiting...\n");
    }
    else
    {
        shrLogEx(LOGBOTH | CLOSELOG, 0, "oclVectorAdd.exe Exiting...\nPress
<Enter> to Quit\n");
        getchar();
    }
    exit (iExitCode);
}
```

The OpenCL CODE for Coordinate operation

```
#include <oclUtils.h>
#include <time.h>

const char* cSourceFile = "VectorAdd.cl";


void *sx1, *sy1,*sx2,*sy2,*sD,*sP;


cl_context cxGPUContext;
cl_command_queue cqCommandQueue;
cl_platform_id cpPlatform;
cl_device_id cdDevice;
cl_program cpProgram;
cl_kernel ckKernel;
cl_mem Devsx1;
cl_mem Devsy1;
```

```
cl_mem Devsx2;
cl_mem Devsy2;
cl_mem DevsD;
cl_mem DevsP;

size_t szGlobalWorkSize;
size_t szLocalWorkSize;
size_t szParmDataBytes;
size_t szKernelLength;
cl_int ciErr1, ciErr2;
char* cPathAndName = NULL;
char* cSourceCL = NULL;



void printFloatArray(float* sx1,float* sy1,float* sx2,float* sy2,float*
sD,float* sP, char* name, int length)
{
printf("%s:\n", name);
float dcnt=0.0f,pcnt=0.0f;float d, p;
for(int i=0;i<length;i++)
{

d=(sy2[i]-sy1[i])/(sx2[i]-sx1[i]);
p=sy2[i]-(d*sx2[i]);

if(d!=0 && (sx2[i]!=sx1[i]))
{if(d>sD[i])
dcnt=dcnt+((d-sD[i])/d);
else
dcnt=dcnt-((d-sD[i])/d);
}

if((sx2[i]!=sx1[i]))
{if(p>sP[i])
pcnt=pcnt+((p-sP[i])/p);
else
pcnt=pcnt-((p-sP[i])/p);
}}
printf("precesion in slope %f\n",(dcnt*100)/(float)length) ;
printf("precesion in coefficient %f\n",(pcnt* 100)/(float)length) ;

printf("\n\n");
}



int iNumElements = 11444777;
shrBOOL bNoPrompt = shrFALSE;


void Cleanup (int iExitCode);


clock_t EndTimer(clock_t begin)
{
    clock_t End;
```

```
    End = clock() * CLK_TCK;   //stop the timer
    return End;
}

int main(int argc, char **argv)
{

    szLocalWorkSize = 256;
    szGlobalWorkSize = shrRoundUp((int)szLocalWorkSize, iNumElements);  //
rounded up to the nearest multiple of the LocalWorkSize



    sx1 = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
    sy1 = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
      sx2 = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
      sy2 = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
      sD = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);
      sP = (void *)malloc(sizeof(cl_float) * szGlobalWorkSize);

    shrFillArray((float*)sx1, iNumElements);
    shrFillArray((float*)sy1, iNumElements);
      shrFillArray((float*)sx2, iNumElements);
    shrFillArray((float*)sy2, iNumElements);

     double begin = clock() * CLK_TCK;
     double elapTicks;

      ciErr1 = clGetPlatformIDs(1, &cpPlatform, NULL);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clGetPlatformID, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }



    //Get the devices
    ciErr1 = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice,
NULL);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clGetDeviceIDs, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    //Create the context
    cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateContext, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }
```

```
    // Create a command-queue
    cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateCommandQueue, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Allocate the OpenCL buffer memory objects for source and result on the
device GMEM
    Devsx1 = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float)
* szGlobalWorkSize, NULL, &ciErr1);
    Devsy1 = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float)
* szGlobalWorkSize, NULL, &ciErr2);
    Devsx2 = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float)
* szGlobalWorkSize, NULL, &ciErr1);
    Devsy2 = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float)
* szGlobalWorkSize, NULL, &ciErr2);
      ciErr1 |= ciErr2;
    DevsD = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, sizeof(cl_float)
* szGlobalWorkSize, NULL, &ciErr2);
    ciErr1 |= ciErr2;
      DevsP = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,
sizeof(cl_float) * szGlobalWorkSize, NULL, &ciErr2);
    ciErr1 |= ciErr2;


    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateBuffer, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Read the OpenCL kernel in from source file
    cPathAndName = shrFindFilePath(cSourceFile, argv[0]);
    cSourceCL = oclLoadProgSource(cPathAndName, "", &szKernelLength);

    // Create the program
    cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char
**)&cSourceCL, &szKernelLength, &ciErr1);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateProgramWithSource, Line %u in file %s !!!\n\
n", __LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Build the program with 'mad' Optimization option
    #ifdef MAC
        char* flags = "-cl-fast-relaxed-math -DMAC";
    #else
        char* flags = "-cl-fast-relaxed-math";
    #endif
    ciErr1 = clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
    if (ciErr1 != CL_SUCCESS)
```

```
    {
        shrLog("Error in clBuildProgram, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Create the kernel
    ckKernel = clCreateKernel(cpProgram, "VectorAdd", &ciErr1);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clCreateKernel, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Set the Argument values
    ciErr1 = clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&Devsx1);
    ciErr1 |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&Devsy1);
    ciErr1 = clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&Devsx2);
    ciErr1 |= clSetKernelArg(ckKernel, 3, sizeof(cl_mem), (void*)&Devsy2);
      ciErr1 |= clSetKernelArg(ckKernel, 4, sizeof(cl_mem), (void*)&DevsD);
      ciErr1 |= clSetKernelArg(ckKernel, 5, sizeof(cl_mem), (void*)&DevsP);
    ciErr1 |= clSetKernelArg(ckKernel, 6, sizeof(cl_int),
(void*)&iNumElements);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clSetKernelArg, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    ciErr1 = clEnqueueWriteBuffer(cqCommandQueue, Devsx1, CL_FALSE, 0,
sizeof(cl_float) * szGlobalWorkSize, sx1, 0, NULL, NULL);
    ciErr1 |= clEnqueueWriteBuffer(cqCommandQueue, Devsy1, CL_FALSE, 0,
sizeof(cl_float) * szGlobalWorkSize, sy1, 0, NULL, NULL);
      ciErr1 = clEnqueueWriteBuffer(cqCommandQueue, Devsx2, CL_FALSE, 0,
sizeof(cl_float) * szGlobalWorkSize, sx2, 0, NULL, NULL);
    ciErr1 |= clEnqueueWriteBuffer(cqCommandQueue, Devsy2, CL_FALSE, 0,
sizeof(cl_float) * szGlobalWorkSize, sy2, 0, NULL, NULL);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clEnqueueWriteBuffer, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }

    // Launch kernel
    ciErr1 = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL,
&szGlobalWorkSize, &szLocalWorkSize, 0, NULL, NULL);
    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clEnqueueNDRangeKernel, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }
```

```
    ciErr1 = clEnqueueReadBuffer(cqCommandQueue, DevsD, CL_TRUE, 0,
sizeof(cl_float) * szGlobalWorkSize, sD, 0, NULL, NULL);
      ciErr1 = clEnqueueReadBuffer(cqCommandQueue, DevsP, CL_TRUE, 0,
sizeof(cl_float) * szGlobalWorkSize, sP, 0, NULL, NULL);


    if (ciErr1 != CL_SUCCESS)
    {
        shrLog("Error in clEnqueueReadBuffer, Line %u in file %s !!!\n\n",
__LINE__, __FILE__);
        Cleanup(EXIT_FAILURE);
    }


  elapTicks = (double)EndTimer(begin);
  printf("Processing time: %f (ms) \n", elapTicks/1000);

      printFloatArray((float *)sx1,(float *)sy1,(float *)sx2,(float *)sy2,
(float *)sD,(float *)sP, "Result", iNumElements);


    Cleanup (EXIT_SUCCESS);
}

void Cleanup (int iExitCode)
{


    if(cPathAndName)free(cPathAndName);
    if(cSourceCL)free(cSourceCL);
      if(ckKernel)clReleaseKernel(ckKernel);
    if(cpProgram)clReleaseProgram(cpProgram);
    if(cqCommandQueue)clReleaseCommandQueue(cqCommandQueue);
    if(cxGPUContext)clReleaseContext(cxGPUContext);
    if(Devsx1)clReleaseMemObject(Devsx1);
    if(Devsy1)clReleaseMemObject(Devsy1);
      if(Devsx2)clReleaseMemObject(Devsx2);
    if(Devsy2)clReleaseMemObject(Devsy2);
     if(DevsD)clReleaseMemObject(DevsD);
       if(DevsP)clReleaseMemObject(DevsP);

    free(sx1);
    free(sy1);
    free(sx2);
    free(sy2);
      free (sD);
      free (sP);

    // finalize logs and leave
    if (bNoPrompt)
    {
        shrLogEx(LOGBOTH | CLOSELOG, 0, "oclVectorAdd.exe Exiting...\n");
    }
    else
    {
```

```
        shrLogEx(LOGBOTH | CLOSELOG, 0, "oclVectorAdd.exe Exiting...\nPress
<Enter> to Quit\n");
        getchar();
    }
    exit (iExitCode);
}
```

The Kernel programs in OpenCL For Basic operation

```
__kernel void VectorAdd(__global const float* a, __global const float* b,
__global float* c, __global float* d, __global float* e, __global float* f,
int iNumElements)
{
    // get index into global data array
    int iGID = get_global_id(0);

    // bound check (equivalent to the limit on a 'for' loop for
standard/serial C code
    if (iGID >= iNumElements)
    {
        return;
    }

    // add the vector elements
    c[iGID] = a[iGID] * b[iGID];
    d[iGID] = a[iGID] / b[iGID];
    e[iGID] = a[iGID] + b[iGID];
    f[iGID] = a[iGID] - b[iGID];
}
```

The Kernel Program for Quadratic operation.

```
__kernel void VectorAdd(__global const float* a, __global const float* b,
__global float* c, __global float* d, __global float* e, int iNumElements)
{
    // get index into global data array
    int iGID = get_global_id(0);

    // bound check (equivalent to the limit on a 'for' loop for
standard/serial C code
    if (iGID >= iNumElements)
    {
        return;
    }

    // add the vector elements

    d[iGID] = (- b[iGID]+sqrt(double((b[iGID]*b[iGID])-
(4*a[iGID]*c[iGID]))))/2*a[iGID];
    e[iGID] = (- b[iGID]+sqrt(double((b[iGID]*b[iGID])-
(4*a[iGID]*c[iGID]))))/2*a[iGID];
```

```
}
```

The Kernel programs in OpenCL For Coordinate operation

```
__kernel void VectorAdd(__global const float* x1, __global const float*
y1,__global const float* x2, __global const float* y2, __global float* d,
__global float* e, int iNumElements)
{
    // get index into global data array
    int iGID = get_global_id(0);

    // bound check (equivalent to the limit on a 'for' loop for
standard/serial C code
    if (iGID >= iNumElements)
    {
        return;
    }

    // add the vector elements
    if(x2[iGID]!=x1[iGID])
    d[iGID] = (y2[iGID]-y1[iGID])/(x2[iGID]-x1[iGID]);
    e[iGID] = y2[iGID]-(d[iGID]*x2[iGID]);

}
```

Bibliography

[1] **TAN Cai-feng, MA An-guo, X Zuo-cheng**, “Parallel implementation of Genetic Algorithm on CUDA”

[2] **A Bakhoda, GL Yuan, WWL Fung, H Wong**, “Analysing CUDA workload using a detailed GPU simulator”.

[3] **I Castaño**, “High Quality DXT compression using CUDA”.

[4] **J Breitbart, C Fohry**, “OpenCL an effective programming model for data parallel computations at the cell broadband engine”

[5] **MJ Gutmann - Journal of Applied Crystallography**, **“**Accelerated computation of diffuse scattering pattern and application to magnetic neutron scattering”.

.