OpenCL

Brian LaRochelle
Computer Science Department
San Jose State University
San Jose, CA, 95192
408-924-1000
brlarochelle@gmail.com

Abstract

The purpose of this paper is to provide an overview of the OpenCL language. This includes the purpose and design goals of the language, the architecture of the language, and some core features of OpenCL. It will also go over some areas where the language excels based on real world performance data.

1. Introduction

OpenCL is a framework maintained by the Kronos group with industry support from the industry, including, Nvidia, AMD, Apple, Intel, Samsung, ARM holdings, and many others. The primary goal of the framework is to be able to write programs that will execute in a heterogeneous programming environment [1]. Modern computers no longer consist of one general purpose CPU, they now usually have a GPU which is capable of running general purpose code. Each device on a system has certain things that it may excel at, and the design of OpenCL is built to take advantage of each devices strengths in the system. The framework is primarily built for CPUs (Central Processing Units), GPUs (Graphics Processing Units), and DSPs (Digital Signal Processors) [1]. This paper will focus on using OpenCL on CPUs and GPUs, but the concepts may be allowed to carry over to other devices used by the framework.

1. OpenCL Standard

1.1 Memory Architecture

To first understand the OpenCL framework it is key to understand the architecture. To start this paper will first go over the memory model. As seen in figure 1 below, Memory is explicitly divided between the main host process, and all other areas. Memory must be explicitly transferred between host memory, global/constant memory, or local memory. Host memory is directly form the CPU, global memory is on the device, and local memory is group of the device.

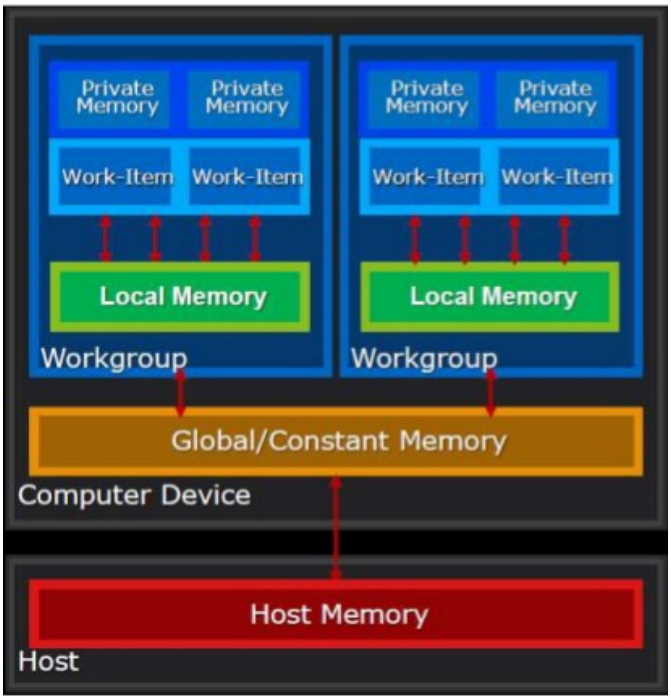

Figure 1. OpenCL Memory Model[2]

1.2 Execution Architecture

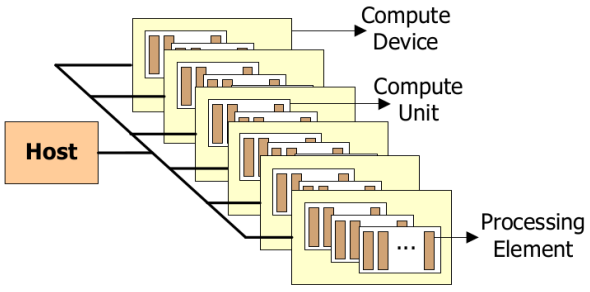

Figure 2. OpenCL Execution Model[2]

As shown in figure 2, the host process controls multiple devices. This devices may or may not consist of multiple compute elements. These compute elements then consist of on or more processing elements. The way these are divided are determined

exclusively by device hardware. One thing to note, however devices may be grouped together in OpenCL if the programmer wishes to do so.

As can be seen by the OpenCL execution model, it inherently designed with parallelism in mind. Not only in dividing work among devices, but also on the device itself among compute units, and processing elements. OpenCL handles parallelism in two different ways. It can do either task parallelism, or data parallelism.

1.3 OpenCL Structures

The architecture also has two main memory structures, the buffer and the image. The buffer is the standard type that most programmers are probably familiar with. It is just a simple one dimensional structure, given as a pointer. The other memory structure is an image. This is either a one or two dimensional memory structure. It is often compared to a 2d or 3d texture. The use case is designed to take advantage of device hardware, such as that on a GPU, that has been built specifically for these kinds of structures. An image can of course be passed as a buffer, but there may be performance disadvantages to doing so, as the programmer will be unable to take advantage of specific language tools designed for the image object.

1.4 OpenCL Kernels

OpenCL kernels are the main component of any OpenCL program. The kernel of an OpenCL program is what actually runs on each device in the program. A kernel is analogous to a function in any other programming language. OpenCL kernels as of OpenCL 1.2 are written in an extension of C99. The C99 language has primarily been extended to include the OpenCL data types. There are constraints to OpenCL kernels, they are unable to make system calls, they must return void, and recursion may not be used. There are also a few other restrictions mentioned in the OpenCL specification. An example kernel is shown in figure 3.

```
__kernel void matrix_mul(__global unsigned int *output,

__global unsigned int *matrix1,

__global unsigned int *matrix2,

 const unsigned int dimension)
{
   unsigned int x = get_global_id(0); //get x
   unsigned int y = get_global_id(1);//get y

   unsigned int element=0;
   unsigned int i;
   for(i = 0; i < dimension; ++i)
   {
      element += matrix1[y * dimension + i] *
matrix2[i*dimension + x];
   }
   output[y*dimension+x] =element;
}
```

Figure 3. OpenCL kernel for matrix multiplication

The kernel above is designed to perform a matrix multiplication on two square matrices. Here it can be seen the parameters for the buffers are global as they reside in global memory, and the dimension is declared as a constant since as a single variable it is in the constant memory. OpenCL is designed to have each processing element running executing the kernel on its own. The processing element gets it's position in the overall execution of the program by the get_global_id function. For comparison a typical c++ matrix multiplication is shown in figure 4.

```
for(unsigned int j=0; j<length; ++j)//rows
{
  for(unsigned int k=0; k<width; ++k)//columns
  {
    for(int i=0; i<min(length,width); ++i)
    {
      output[j*length+k+=input1[i*length+k]*input2[j*length+i];
    }
  }
}
return output;
```

Figure 4. C++ Matrix Multiplication

Comparing the c++ matrix multiplication in figure four with figure one it can be seen that the global_ids are the outer loops of the C++ function. While this is a simple OpenCL kernel it shows that they are not necessarily complex.

OpenCL has the challenge however of being required to run these types of kernels on varying devices which can be of not only very different hardware configurations, but also different manufactures that may have radically different instruction sets for their hardware. For programs it is not uncommon to see them distributed with a few CPU types, and a few operating system types, it would be impractical or impossible to distribute software for every possible combination of that with every possible OpenCL GPU. For this reason the host program can still be compiled and can be distributed like the traditional method, but the kernel can not, because the kernel is what will be running on the specific device, and it is improbable to know the device beforehand. OpenCL's solution to this problem is to compile the kernel at runtime after the device is known.

1.5 Initialization of an OpenCL program

This first program any programmer runs is of course hello world, and to demonstrate the basic setup of an OpenCL program is no different. It is perhaps even more relevant in OpenCL, since OpenCL requires a lot of initialization and setup compared to many other languages. In this section, some of the basic but important OpenCL initialization functions and structures for passing data between devices will be explained. The Hello World program will be broken up in this section to more clearly explain it line by line, but is available in a contiguous form in the appendix[4]. Some of the declaration of variable may also be left out. The language used for the main program is C, however APIs for other languages are available.

```
FILE *fp;
char fileName[] = "./hello.cl";
char *source_str;
size_t source_size;
fp = fopen(fileName, "r");
if (!fp) {
   fprintf(stderr, "Failed to load kernel.\n");
   exit(1);
}
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);
```

Figure 5. Opening an OpenCL Kernel

Above it is shown in figure 5, the kernel for the program is opened in order to be compiled. It is opened just like any other file one might open in C and is read into a character array. From here the program becomes mostly boilerplate. The next step is to select the OpenCL device that will be used.

```
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);

ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1,
&device_id, &ret_num_devices);
```

Figure 6. Identifying the Platform and Devices

Many of the OpenCL functions shown from here have a return value of an error code. The actual information wanted form the function is typically passed through a variable to the function. The first function listed gives the platform id and the number of different platforms on the host system. These are typically separated by device manufacturer. The second function returns the number of devices specified running on that platform. In the case given above it is using the parameter CL_DEVICE_TYPE_GPU. There are several others such as DEFAULT, CPU, ACCELERATOR, and ALL [3].

```
context = clCreateContext(NULL, 1, &device_id, NULL, NULL,
&ret);

command_queue = clCreateCommandQueue(context, device_id,
0, &ret);
```

Figure 7. Creating a Context and a Command Queue

The next step is to create the context. A context is a group of one or more devices that will be executing commands. It is important to note that in the example above &device_id is an array of devices. The importance of grouping devices in a context is that while each device receives its own queue to receive commands, as seen by the second function, devices within the same context have the ability to share memory. Devices in different contexts do not have this ability[1].

```
program = clCreateProgramWithSource(context, 1, (const char
**)&source_str,(const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL,
NULL);
 kernel = clCreateKernel(program, "hello", &ret);
```

Figure 8. Creating, building a program, and kernel

In figure 8 the three steps for creating the binary of a kernel are shown. First, because a context consists of unique devices, the program must be built for that specific context. Then the created program must be built for each device as they may be of different architectures. A kernel object must then be created.

```
memobj = clCreateBuffer(context,
CL_MEM_READ_WRITE,MEM_SIZE * sizeof(char), NULL,
&ret);
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void
*)&memobj);
ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);
```

Figure 9. Creating a Buffer and Passing it to a Kernel

In figure 9 the first function creates the buffer. A buffer is needed because it is an array that the kernel will be returning. The size of the buffer is specified by MEM_SIZE*sizeof(char). The buffer is then set to be the first argument of the kernel. The task is then enqueue to the command queue. One interesting thing to note is if a kernel creates an output array, the array from the source program is not passed in at this time.

```
ret = clEnqueueReadBuffer(command_queue, memobj,
CL_TRUE, 0,
   MEM_SIZE * sizeof(char),string, 0, NULL, NULL);
```

Figure 10. Reading the result

In figure 10 the function to receive the result from the kernel is shown. The important parameters are as follows, the queue it was sent to, the string we are writing the result to, and the CL_TRUE. The last parameter defines this as a blocking read from the kernel. We do not necessarily have to have the main program block and wait for the kernel to finish. However, since this is the end of the program, we are having the program wait. The kernel simply put the characters “Hello World in the order of the array.

This basic boilerplate is all that is required for many simple OpenCL programs. It also demonstrates the numerous steps required for even something as simple as Hello World. However it does not get increasingly complex as the program becomes more complex, and it demonstrates how much abstraction is built into the framework in order to handle different devices and different groupings of devices.

2.0 OpenCL Performance Analysis

Given the complexity the hello world program it brings into question whether it is even worth using OpenCL at all. After seeing how many source lines of code it was to do Hello World that is a fair question. For the first performance comparison a matrix multiply will be tested. A very basic matrix multiply will be done, this basic implementation has a time complexity in big O notation of $O(n^3)$. So while not the most efficient implementation it should be adequate in demonstrating the performance differences of OpenCL, versus a single thread C++ implementation, and a multi-threaded C++ multiplication.

2.1 Hardware and Software Used

The hardware and software used for testing was as follows.

ASUS G73JH-RBBX05
Intel Core i7-720QM - 4 physical cores - 8 logical – 1.60GHz - max turbo 2.8GHz
ATI Mobility Radeon HD 5870 – 1GB GDDR5 – Core Clock 700 MHz Memory Clock – 1GHz- 10 OpenCL compute units

Wavefront of 64
6GB DDR3
fglrx driver 13.35.5 (March 12 2014)
Ubuntu 13.10
gcc 4.8
clang++ 3.2
AMD APP (Accelerated parallel processing) SDK 2.9

2.2 Testing Methodology

For all of the programs, a square matrix was used. It was used because different sized matrices should not impact performance testing much. Implementing it was also slightly easier. This does mean doubling one side of the matrix is increasing the number of elements by four. Since it is an n^3 algorithm, this gives a 64x multiple to doubling the dimensions. Since it is a square matrix, dimension will be referring to the length of only one of the sides. The average of three runs for each dimension was used to eliminate random performance errors.

2.3 Single Threaded Matrix Multiplication

Dimensions	Time in seconds
16	0
32	0
64	.01
128	.03
256	.21
512	1.87
1024	37.99
2048	324.44

Table 1. Single Thread Matrix Multiply Performance

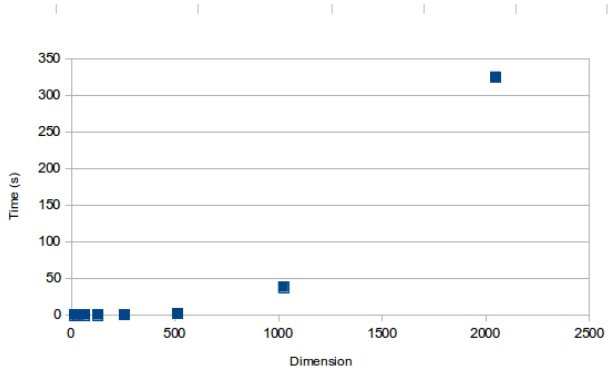

Figure 11. Graph of Single Thread Performance

The single thread implementation was written in C++, and compiled used Clang++. As can be seen from the graph and table, it did become slower very quickly as expected. It did not increase at quite the rate I thought it would, but that may be due to compiler optimizations. Times less than 64 were too fast to gather anything less than zero.

2.3 Multi-Thread Matrix Multiplication

The multi-thread matrix multiplication was done using C++11 using the STL implementation of threads, and clang++ which implemented that using the pthread library. Testing showed that the optimal number of threads to use was 8. While there were not 8 physical cores to use, I assume that the hyper-threaded meant that another thread could execute if the operating system choose to context switch, leading to the better performance. The performance from 8 over 4 was minor but consistently evident.

Dimensions	Time in seconds
16	0
32	0
64	.00125
128	.005
256	.08
512	.705
1024	12.25
2048	112.025

Table 2. Multi-Threaded Matrix Multiply Performance

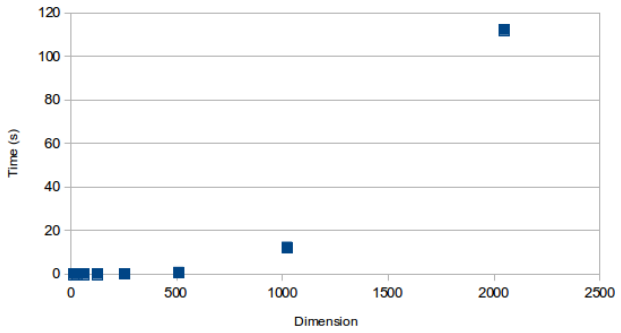

Figure 12. Graph of Multi-Thread Performance

As can be seen from table 2 and figure 12, the multi-threaded performance was far better than the single thread implementation. That being said though, for times as many cores were being used, but there was only an average speedup of around 3.16. This is not the four that we would like. This implementation however, did always pick the optimal number of threads, since C++11 has support for querying the maximum number of concurrent threads. Also if there was a penalty for creating the 8 threads, it could not be seen given matrices under 64 did not show any time.

2.3 OpenCL Matrix Multiplication CPU

The matrix multiplication for both the CPU and the GPU were both from the exact same source code, with the exception of when querying devices, a CPU was requested instead of the GPU for CPU testing and vice versa for GPU testing. The number of threads for execution did not need to be specified for the program, but was only outputted for testing purposes. Time to compile the program and kernel were also included, due to that being steps that the C++ implementation did not have to due, and does add to the OpenCL execution time.

Dimensions	Time in seconds
16	0.06125
32	0.0625
64	0.0625
128	0.065
256	0.0825
512	0.17875
1024	3.4025
2048	35.8025

Table 3. OpenCL CPU Matrix Multiply Performance

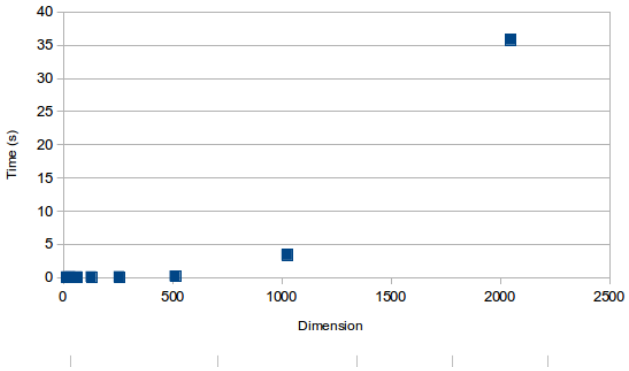

Figure 13. Graph of OpenCL CPU Performance

The results from the OpenCL CPU testing were not what I expected to get, with the exception of the low dimension matrices. There it was seen that there was approximately a 0.06 second overhead just to create the OpenCL environment. It did not come close to matching the single or multi-threaded performance until 256, where it was slightly slower than the C++ implementation. What it did miss out on with small matrices it more than made up with larger matrices. With larger matrices it was around 10x faster than the single thread performance, and was around 3x faster than the multi-threaded C++ implementation. The only thing that it must have done, since it was the same number of threads on the same hardware is very heavy performance optimization, since each kernel was meant to run on each cell of the matrix.

2.3 OpenCL Matrix Multiplication GPU

The exact same program used for the CPU OpenCL implementation was used, just the device was specified to be the GPU. It is worth noting that the GPU has 10 compute units compared to the CPU's 8 compute units. However, the GPU has a wavefront of 64. AMD determines the wavefront number as the number of processing elements times the length of their pipeline. The pipeline is always 4, therefore this GPU has 16 processing elements. AMD defines the wavefront as such however, because if that processing element does not have 4 threads associated with it, it will be underutilized. Therefore the wavefront number of AMD GPU's is more useful than the processing elements. Therefore the GPU used can handle 640 threads among all of its OpenCL compute units.

Dimensions	Time in seconds
16	0.51
32	0.51
64	0.52
128	0.59
256	0.5
512	0.51
1024	0.58
2048	1.01
4096	4.8
8196	33.99

Table 4. OpenCL GPU Matrix Multiply Performance

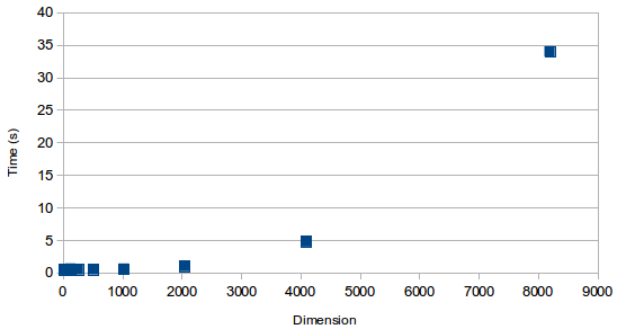

Figure 14. Graph of OpenCL GPU Performance

Here OpenCL shows where the GPU really shines. It is a task with no branching, and is highly parallelized. On the 2048 dimension the GPU outperforms the OpenCL CPU implementation by nearly 35x. Going up to 8196, which would not have been possible in the single thread example in any reasonable amount of time was required to see how far was needed to push the GPU to a longer execution time. This is enormous. On the other hand for any matrix smaller than 1024 the GPU performed worse than the OpenCL CPU. This is most likely due to memory needing to be copied and possibly compilation done on the GPU, instead of being located in the main system memory of the CPU. An anomaly that could be consistency reproduced was the time going done after 128 before going back up again at 2048. A reason for this occurring could not be determined.

3. Summary and Conclusion

In summary, I found through my matrix multiplication OpenCL analysis that OpenCL accomplishes what the Khronos Group wanted the language to be. I was able to switch between the CPU and the GPU with almost no changes to my code. Not only that but it also outperformed my C++ implementations with no extra work required once the language was learned. Outperformed is almost an understatement with even the CPU. The GPU on the other hand outperforms the OpenCL CPU by orders of

magnitude. Once the framework is adjusted to, it provides a fairly simple to use, excellently performing environment for heterogeneous systems as is intended.

4. References

[1] Khronos OpenCL Working Group, The OpenCL Specification v1.2, https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[2] Trevett, Neil; OpenCL Introduction SiggraphAsia 2013, https://www.khronos.org/assets/uploads/developers/library/2013-siggraph-asia/OpenCL%20Intro%20SIGGRAPH%20Asia%20Nov13.pdf

[3] OpenCL References Pages, https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/

[4] Tsuchiyama, Royoji; Nakamura, Takashi; et al, The OpenCL Programming Book, 4/26/2012, http://www.fixstars.com/en/opencl/book/

[5] Tompson, Jonathon; Schlachter, Kristofer, An Introduction to the OpenCL Programming model http://www.cs.nyu.edu/~lerner/spring12/Preso07-OpenCL.pdf

Appendix:

Single threaded matrix multiplication source:

```
#include <iostream>
#include <ctime>
#include <algorithm>
using namespace std;
template <typename Type> Type* matrix_muiltiply(Type*,Type*,unsigned int, unsigned int);
int main(int argc, char **argv)
{
	unsigned int *matrix1, *matrix2;
	unsigned int length, width;
	cout<<"enter the length/width of the square matrix" <<endl;
	cin >> length;
	width=length;
	matrix1 = new unsigned int [length*width];
	matrix2 = new unsigned int [length*width];
	for(unsigned int i=0; i<length*width; ++i)
	{
		matrix1[i]=i;
		matrix2[i]=i;
	}
	srand(time(0));
	clock_t start=clock();
	unsigned int *resultant_matrix = matrix_muiltiply(matrix1,matrix2,length,width);
	clock_t end=clock();
	if(max(length,width) <6)
	{
		for(unsigned int i=0; i<length; ++i)
		{
			for(unsigned int j=0; j<width; ++j)
			{
				cout<<resultant_matrix[i*length+j] << " ";
			}
			cout<< endl;
		}
	}
	std::cout<<"Time taken:"<< (double)(end-start)/CLOCKS_PER_SEC<< " seconds" << std::endl;
	return 0;
}

template <typename Type>
Type* matrix_muiltiply(Type *input1, Type *input2,unsigned int length, unsigned int width)
{
	Type *output = new Type [length*width];
	for(unsigned int j=0; j<length; ++j)//rows
	{
		for(unsigned int k=0; k<width; ++k)//columns
		{
			for(int i=0; i<min(length,width); ++i)
			{
				output[j*length+k]+=input1[i*length+k]*input2[j*length+i];
			}
		}
	}
	return output;
}
```

Multi-Threaded Matrix Multiplication Source:

```
//clang++ matrixthreaded.cxx -std=c++11 -pthread
#include <iostream>
```

```
#include <ctime>
#include <algorithm>
#include <thread>
#include <vector>
using namespace std;
template <typename Type> Type* matrix_muiltiply(Type*,Type*,unsigned int, unsigned int, unsigned int);
template <typename Type> void matrix_muiltiply_thread(Type*,Type*,Type*,unsigned int,unsigned int, unsigned int, unsigned int);
int main(int argc, char **argv)
{
	unsigned int *matrix1, *matrix2;
	unsigned int length, width,num_threads;
	cout<< "maximum supported concurrant threads is " << std::thread::hardware_concurrency()<<endl;
	cout<<"enter the length/width of the square matrix" <<endl;
	cin >> length;
	width=length;
	cout<<"enter the number of threads" <<endl;
	cin>> num_threads;
	matrix1 = new unsigned int [length*width];
	matrix2 = new unsigned int [length*width];
	for(unsigned int i=0; i<length*width; ++i)
	{
		matrix1[i]=i;
		matrix2[i]=i;
	}
	srand(time(0));
	clock_t start=clock();
	unsigned int *resultant_matrix = matrix_muiltiply(matrix1,matrix2,length,width,num_threads);
	clock_t end=clock();
	if(max(length,width) <6)
	{
		for(unsigned int i=0; i<length; ++i)
		{
			for(unsigned int j=0; j<width; ++j)
			{
				cout<<resultant_matrix[i*length+j] << " ";
			}
			cout<< endl;
		}
	}
	std::cout<<"Time taken:"<< (double)(end-
start)/CLOCKS_PER_SEC/min(num_threads,std::thread::hardware_concurrency())<< " seconds" << std::endl;
	return 0;
}

template <typename Type>
void matrix_muiltiply_thread(Type *input1, Type *input2,Type *output,unsigned int length, unsigned int width,unsigned int offset,
unsigned int num_threads)
{
	for(unsigned int j=offset; j<length; ++j)//rows
	{
		for(unsigned int k=0; k<width; ++k)//columns
		{
			for(int i=0; i<min(length,width); i+=num_threads)
			{
				output[j*length+k]+=input1[i*length+k]*input2[j*length+i];
			}
		}
	}
}
template <typename Type>
Type* matrix_muiltiply(Type *input1, Type *input2,unsigned int length, unsigned int width, unsigned int num_threads)
{
	Type *output = new Type[length*width];
	vector<thread> *thread_group = new vector<thread>;
	for(unsigned int i=0; i<num_threads; ++i)
```

```
		{
			thread_group->push_back(thread(matrix_muiltiply_thread<Type>,input1,input2,output,length,width,i,num_threads));
		}
		for(unsigned int i=0; i<num_threads; ++i)
		{
			thread_group->at(i).join();
		}
		return output;
}
```

Hello World OpenCL source[4]:

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>


#define MEM_SIZE (128)
#define MAX_SOURCE_SIZE (0x100000)

int main()
{
		cl_device_id device_id = NULL;
		cl_context context = NULL;
		cl_command_queue command_queue = NULL;
		cl_mem memobj = NULL;
		cl_program program = NULL;
		cl_kernel kernel = NULL;
		cl_platform_id platform_id = NULL;
		cl_uint ret_num_devices;
		cl_uint ret_num_platforms;
		cl_int ret;

   char string[MEM_SIZE];

   FILE *fp;
   char fileName[] = "./hello.cl";
   char *source_str;
   size_t source_size;

   /* Load the source code containing the kernel*/
   fp = fopen(fileName, "r");
   if (!fp) {
		fprintf(stderr, "Failed to load kernel.\n");
		exit(1);
   }
   source_str = (char*)malloc(MAX_SOURCE_SIZE);
   source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
   fclose(fp);

   /* Get Platform and Device Info */
   ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
   ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &ret_num_devices);

   /* Create OpenCL context */
   context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);

   /* Create Command Queue */
   command_queue = clCreateCommandQueue(context, device_id, 0, &ret);

   /* Create Memory Buffer */
   memobj = clCreateBuffer(context, CL_MEM_READ_WRITE,MEM_SIZE * sizeof(char), NULL, &ret);

   /* Create Kernel Program from the source */
   program = clCreateProgramWithSource(context, 1, (const char **)&source_str,(const size_t *)&source_size, &ret);
```

```
    /* Build Kernel Program */
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    /* Create OpenCL Kernel */
    kernel = clCreateKernel(program, "hello", &ret);

    /* Set OpenCL Kernel Parameters */
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobj);

    /* Execute OpenCL Kernel */
    ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);

    /* Copy results from the memory buffer */
    ret = clEnqueueReadBuffer(command_queue, memobj, CL_TRUE, 0,
    MEM_SIZE * sizeof(char),string, 0, NULL, NULL);

    /* Display Result */
    puts(string);
    int i=2;
            printf("%d\n",i*2);
    /* Finalization */
    ret = clFlush(command_queue);
    ret = clFinish(command_queue);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(memobj);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);

    free(source_str);

            return 0;
}
```

Hello World Kernel:

```
__kernel void hello(__global char* string)
{
            string[0] = 'H';
            string[1] = 'e';
            string[2] = 'l';
            string[3] = 'l';
            string[4] = 'o';
            string[5] = ',';
            string[6] = ' ';
            string[7] = 'W';
            string[8] = 'o';
            string[9] = 'r';
            string[10] = 'l';
            string[11] = 'd';
            string[12] = '!';
            string[13] = '\0';
}
```

Matrix Muiltiplication OpenCL source:

```
//gcc matrixmul_cl.c -I$AMDAPPSDKROOT/include -L$AMDAPPSDKROOT -lOpenCL  -o matmultcl.out
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <CL/cl.h>
#include <string.h>
#define MAX_SOURCE_SIZE (0x100000)

int main()
```

```
{
	//create matrixes
	printf("0 for CPU 1 for GPU\n");
	int mode;
	scanf("%d", &mode);
	printf("enter dimension of matrix\n");
	unsigned int dimension;
	scanf("%u",&dimension);

	unsigned int* matrix1 = (unsigned int*) malloc(dimension*dimension*sizeof(unsigned int));
	unsigned int* matrix2 = (unsigned int*) malloc(dimension*dimension*sizeof(unsigned int));
	unsigned int* resultant_matrix = (unsigned int*) malloc(dimension*dimension*sizeof(unsigned int));
	memset(resultant_matrix, 0, dimension*dimension*sizeof(unsigned int));
	unsigned int i;
	unsigned int j;

	for(i=0; i<dimension*dimension; ++i)
	{
		matrix1[i]=i;
		matrix2[i]=i;
	}
	if(dimension<6)
	{
		for(i=0; i<dimension; ++i)
		{
			for( j=0; j<dimension; ++j)
			{
				printf("%u ",matrix1[i*dimension+j]);
			}
			printf("\n");
		}
	}


	clock_t start = clock();
	//opencl starts here


	cl_device_id device_id = NULL;
	cl_context context = NULL;
	cl_command_queue command_queue = NULL;
	cl_mem memobj = NULL;
	cl_program program = NULL;
	cl_kernel kernel = NULL;
	cl_platform_id platform_id = NULL;
	cl_uint ret_num_devices;
	cl_uint ret_num_platforms;
	cl_int ret;

FILE *fp;
char fileName[] = "./matrixmul_cl.cl";
char *source_str;
size_t source_size;

/* Load the source code containing the kernel*/
fp = fopen(fileName, "r");
if (!fp) {
	fprintf(stderr, "Failed to load kernel.\n");
	exit(1);
}
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

/* Get Platform and Device Info */
```

```
   ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
            if(!mode)
            {
            ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, 1, &device_id, &ret_num_devices);
   }
            else
            {
                        ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &ret_num_devices);
            }
            //query number of compute units
            cl_int num_compute_units;
            ret = clGetDeviceInfo(device_id,CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_int), &num_compute_units, NULL);
            printf("number of compute units is:%d\n",num_compute_units);

   /* Create OpenCL context */
   context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);

   /* Create Command Queue */
   command_queue = clCreateCommandQueue(context, device_id, 0, &ret);

   /* Create Memory Buffer for pointers */
            cl_mem buf_output, buf_matrix1,buf_matrix2;
   buf_output = clCreateBuffer(context, CL_MEM_READ_WRITE,dimension*dimension*sizeof(unsigned int), NULL, &ret); //for
output
   buf_matrix1 = clCreateBuffer(context, CL_MEM_READ_ONLY , dimension*dimension*sizeof(unsigned int), NULL, &ret);//matrix 1
            buf_matrix2 = clCreateBuffer(context, CL_MEM_READ_ONLY, dimension*dimension*sizeof(unsigned int), NULL, &ret);

            //give pointers to buffers
            clEnqueueWriteBuffer(command_queue, buf_matrix1, CL_TRUE,0,sizeof(unsigned int)*dimension*dimension,
matrix1,0,NULL,NULL);
            clEnqueueWriteBuffer(command_queue, buf_matrix2, CL_TRUE,0,sizeof(unsigned int)*dimension*dimension,
matrix2,0,NULL,NULL);
   /* Create Kernel Program from the source */
   program = clCreateProgramWithSource(context, 1, (const char **)&source_str,(const size_t *)&source_size, &ret);

   /* Build Kernel Program */
   ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

   /* Create OpenCL Kernel */
   kernel = clCreateKernel(program, "matrix_mul", &ret);

            //work item size per unit
            size_t workgroup_size;
            ret= clGetKernelWorkGroupInfo(kernel, device_id, CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE,
sizeof(size_t), &workgroup_size, NULL);
            printf("workgroup size should be %zu\n", workgroup_size);
   /* Set OpenCL Kernel Parameters */
   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), &buf_output);
            ret |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &buf_matrix1);
            ret |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &buf_matrix2);
            ret |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &dimension);

   /* Execute OpenCL Kernel */
            size_t globalWorkSize[2];
            globalWorkSize[0]=dimension;
            globalWorkSize[1]=dimension;
   ret = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL, globalWorkSize, NULL, 0, NULL, NULL);

            clFinish(command_queue);
   /* Copy results from the memory buffer */
   ret = clEnqueueReadBuffer(command_queue, buf_output, CL_TRUE, 0, dimension*dimension*sizeof(unsigned int), resultant_matrix,
0, NULL, NULL);

            clock_t diff = clock()-start;
            if(mode)
```

```
        {
                printf("time taken:%f\n", (double)diff/CLOCKS_PER_SEC);
        }
        else
        {
                printf("time taken:%f\n", (double)diff/CLOCKS_PER_SEC/num_compute_units);
        }
    /* Display Result */
        if(dimension<6)
        {
                for( i=0; i<dimension; ++i)
                {
                        for(j=0; j<dimension; ++j)
                        {
                                printf("%u ", resultant_matrix[i*dimension+j]);
                        }
                        printf("\n");
                }
        }
    /* Finalization */
    ret = clFlush(command_queue);
    ret = clFinish(command_queue);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(memobj);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);

    free(source_str);

        return 0;
}
```

Matrix Multiplication Kernel

```
//matrixmul_cl.cl
__kernel void matrix_mul(__global unsigned int *output,
                                                        __global unsigned int *matrix1,
                                                        __global unsigned int *matrix2,
                                                        const unsigned int dimension)
{
        unsigned int x = get_global_id(0); //get x
        unsigned int y = get_global_id(1);//get y

        unsigned int element=0;
        unsigned int i;
        for(i = 0; i < dimension; ++i)
        {
                element += matrix1[y * dimension + i] * matrix2 [i*dimension + x];
        }
        output[y*dimension+x] =element;
}
```