Dustin Ma
ME 2054
November 7th, 2018

Project 1: Finite Difference Solution of a Vibrating 2D Membrane

Introduction

Graphics processing unit (GPU) computing, also known as parallel computing is utilizing Nvidia's graphics computing codes to parallelize the serial codes that occurs on our central processing units (CPU). The benefits of parallelizing a serial portion of the code is to divide up the task for multiple different fast processing graphics cards to return a speed up in time needed for a large size calculation. Although CPU computing is sufficient for most of today's codes, scientists who are dealing with a large amount of data and needing to do numerous calculations can benefit from using GPU computing. By parallelizing a program, GPU computing carries out processes in a parallel manner (simultaneously), instead of the serial way of the CPU. Due to the ability to handle and compute large amount of data in a short amount of time, GPU computing has major contributions when it comes to performing large scale computations such as Fluid dynamics.

Usually these GPUs are kept inside a well ventilated cool room. Figure 1, shows an example of one of the supercomputing center in the world.

Figure 1. IBM's Gene/P massively parallel supercomputer.

Getting Started With CUDA

To parallelize one's code, the programmer must understand the basic functions that differentiates calling a function on the CPU and GPU. Figure 2, shows some basic functions and syntax programmers must use and learn to program using CUDA.

1) `__global__`
2) `cudaMallocManage();`
3) `cudaMemCpy();`
4) `Kernel<<<nBlocks,threadsPerBlock>>>();`

Figure 2. Examples of general codes and functions to know in CUDA.

While the new language may seem daunting, coders should know that CUDA works with C, therefore converting their daily C code into CUDA only requires an extra understanding of how the GPU computes, and process information.

Inside the GPU

Inside the GPU, there are two sub categories. The first one being blocks, and second ones being threads. Blocks and threads can be specified by the user. Nvidia gives their best practice value of 256 threats per block to have the most computational increase in speed up. While a desired number is given for a performance boost, users can set their own values to test the speed up in their code.

Dustin Ma
ME 2054
November 7th, 2018

Procedures

In this project, we implemented the wave function of a 2d membranes whose boundaries are clamped. This 2d membrane vibrates up and down with the given formula below,

$$\phi_{i,j}{}^{t+1} = 2\phi_{i,j}{}^{t} - \phi_{i,j}{}^{t+1} + \frac{c^2\Delta t^2}{H^2}(\phi_{i+1,j}{}^{t} + \phi_{i-1,j}{}^{t} + \phi_{i,j+1}{}^{t} + \phi_{i,j-1}{}^{t} - 4\phi_{i,j}{}^{t})$$

We had to first initialized the wave function with the formula below,

$$\phi(x, y, t) = 0.1(4x - x^2)(2y - y^2)$$

After initialization, boundary conditions should be applied to all x and y that are at the boundary. In this problem the membrane size is Lx = 4 (units), and Ly = 2 (units). Therefor for all x = 0, x = Lx, y = 0, and y = Ly, the amplitude should be zero.

Methods

I first wrote my serial code and ran it on the CPU. For all Cases, I am running with an end time of 1 second. This gives me the first maximum amplitude. While coding, I used a mesh size of 41x21 to ensure that my exact solution matches with that of my numerical serial code. Initial condition visualization is shown in figure 2.

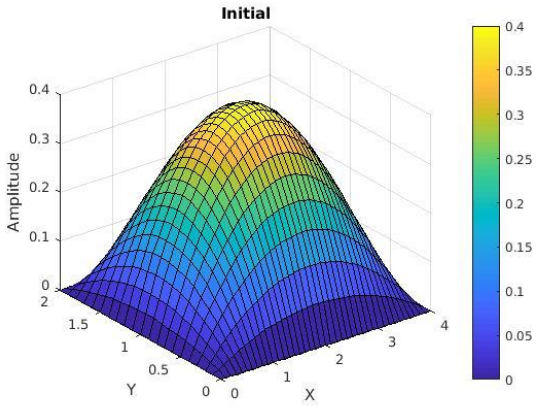

Figure 2. Initial condition of 2d vibrating membrane

After getting the CPU portion of the code working. I started implementing the GPU portion. In this portion I parallelized the code so that it will still only calculate within the boundaries.

Noticeable Speed Up

Speed up tables are shown below. With Table 1 comparing wall time between different sized meshes, and Table two changing in size block size. From Table 1, speed up from the CPU and GPU is significant. While mesh size gets larger and CPU time gradually takes longer, the GPU solves it in less than 5 times of the previous computation time. In Table2, code is ran using a mesh size of 1026 x 514. Comparing the different mesh sizes, speed up can be observed when larger block size is used to solve the mesh. 32x16 is more efficient than 32 x 32. This can be due to part of the problem parameter. Having a block size more closely to that of the problem statement may increase the efficiency of the program overall.

Table 1. Comparing different CPU Mesh size execution time, End time = 1 [s].

Mesh Size	CPU Time [ms]	GPU Time [ms]
514 x 258	3,342.0	22.5
1026 x 514	26,677.8	70.6
2050 x 1026	213,878.7	348.2

Table 2. Comparing Execution configuration of different Block Sizes

Block Size	GPU Time [ms]
8 x 8	82.7
16 x 16	72.5
32 x 16	67.4
32 x 32	69.2

In the PDF below, is my result for 3 different points on the membrane at 4 different end times.

Dustin Ma
ME 2054
November 7th, 2018

Conclusion

In the lab, I implemented GPU parallelism into the oscillating wave formula for a 2d vibrating membrane. The results have shown that GPU is most beneficial when a large problem is at hand. I also noticed that when setting up the GPU environment, it is beneficial to design the environment to match some physical aspect of your problem to gain some speed up.

Some hiccups that I've encountered during this project was understand the usage of cudaMallocManage Function. I figured out that with such function, there is no need to copy memory back and from the GPU to read and receive data. Pointer address swapping also proved to be very useful. I learned how to linearly index a 2d array within the CUDA environment

End time		1	End time		5	End time		7
0.000,	0.000,	0	0.000,	0.000,	0	0.000,	0.000,	0
-0.031,	-0.033,	-0.033	0.031,	0.030,	0.03	-0.028,	-0.030,	-0.03
-0.050,	-0.053,	-0.053	0.039,	0.036,	0.036	-0.046,	-0.049,	-0.049
-0.031,	-0.033,	-0.033	0.031,	0.030,	0.03	-0.028,	-0.030,	-0.03
-0.063,	-0.065,	-0.065	0.062,	0.060,	0.06	-0.056,	-0.060,	-0.06
-0.101,	-0.104,	-0.104	0.078,	0.074,	0.074	-0.091,	-0.097,	-0.097
-0.063,	-0.065,	-0.065	0.062,	0.060,	0.06	-0.056,	-0.060,	-0.06
-0.094,	-0.094,	-0.094	0.093,	0.090,	0.09	-0.082,	-0.088,	-0.088
-0.152,	-0.152,	-0.152	0.116,	0.111,	0.111	-0.134,	-0.143,	-0.143
-0.094,	-0.094,	-0.094	0.093,	0.090,	0.09	-0.082,	-0.088,	-0.088
-0.122,	-0.121,	-0.121	0.122,	0.119,	0.119	-0.107,	-0.116,	-0.116
-0.195,	-0.195,	-0.195	0.152,	0.147,	0.147	-0.174,	-0.186,	-0.186
-0.122,	-0.121,	-0.121	0.122,	0.119,	0.119	-0.107,	-0.116,	-0.116
-0.145,	-0.144,	-0.144	0.151,	0.147,	0.147	-0.133,	-0.142,	-0.142
-0.233,	-0.232,	-0.232	0.187,	0.182,	0.182	-0.215,	-0.226,	-0.226
-0.145,	-0.144,	-0.144	0.151,	0.147,	0.147	-0.133,	-0.142,	-0.142
-0.164,	-0.163,	-0.163	0.177,	0.175,	0.175	-0.158,	-0.165,	-0.165
-0.263,	-0.262,	-0.262	0.220,	0.217,	0.217	-0.253,	-0.263,	-0.263
-0.164,	-0.163,	-0.163	0.177,	0.175,	0.175	-0.158,	-0.165,	-0.165
-0.178,	-0.177,	-0.177	0.201,	0.199,	0.199	-0.179,	-0.186,	-0.186
-0.286,	-0.285,	-0.285	0.251,	0.248,	0.248	-0.286,	-0.295,	-0.295
-0.178,	-0.177,	-0.177	0.201,	0.199,	0.199	-0.179,	-0.186,	-0.186
-0.189,	-0.188,	-0.188	0.221,	0.217,	0.217	-0.196,	-0.201,	-0.201
-0.303,	-0.302,	-0.302	0.277,	0.270,	0.27	-0.311,	-0.318,	-0.318
-0.189,	-0.188,	-0.188	0.221,	0.217,	0.217	-0.196,	-0.201,	-0.201
-0.195,	-0.194,	-0.194	0.236,	0.228,	0.228	-0.206,	-0.210,	-0.21
-0.313,	-0.313,	-0.313	0.296,	0.284,	0.284	-0.327,	-0.331,	-0.331
-0.195,	-0.194,	-0.194	0.236,	0.228,	0.228	-0.206,	-0.210,	-0.21
-0.197,	-0.196,	-0.196	0.240,	0.231,	0.231	-0.210,	-0.213,	-0.213
-0.317,	-0.316,	-0.316	0.302,	0.289,	0.289	-0.332,	-0.336,	-0.336
-0.197,	-0.196,	-0.196	0.240,	0.231,	0.231	-0.210,	-0.213,	-0.213
-0.195,	-0.194,	-0.194	0.236,	0.228,	0.228	-0.206,	-0.210,	-0.21
-0.313,	-0.313,	-0.313	0.296,	0.284,	0.284	-0.327,	-0.331,	-0.331
-0.195,	-0.194,	-0.194	0.236,	0.228,	0.228	-0.206,	-0.210,	-0.21
-0.189,	-0.188,	-0.188	0.221,	0.217,	0.217	-0.196,	-0.201,	-0.201
-0.303,	-0.302,	-0.302	0.277,	0.270,	0.27	-0.311,	-0.318,	-0.318
-0.189,	-0.188,	-0.188	0.221,	0.217,	0.217	-0.196,	-0.201,	-0.201
-0.178,	-0.177,	-0.177	0.201,	0.199,	0.199	-0.179,	-0.186,	-0.186
-0.286,	-0.285,	-0.285	0.251,	0.248,	0.248	-0.286,	-0.295,	-0.295
-0.178,	-0.177,	-0.177	0.201,	0.199,	0.199	-0.179,	-0.186,	-0.186
-0.164,	-0.163,	-0.163	0.177,	0.175,	0.175	-0.158,	-0.165,	-0.165
-0.263,	-0.262,	-0.262	0.220,	0.217,	0.217	-0.253,	-0.263,	-0.263
-0.164,	-0.163,	-0.163	0.177,	0.175,	0.175	-0.158,	-0.165,	-0.165
-0.145,	-0.144,	-0.144	0.151,	0.147,	0.147	-0.133,	-0.142,	-0.142
-0.233,	-0.232,	-0.232	0.187,	0.182,	0.182	-0.215,	-0.226,	-0.226
-0.145,	-0.144,	-0.144	0.151,	0.147,	0.147	-0.133,	-0.142,	-0.142

-0.122,	-0.121,	-0.121	0.122,	0.119,	0.119	-0.107,	-0.116,	-0.116
-0.195,	-0.195,	-0.195	0.152,	0.147,	0.147	-0.174,	-0.186,	-0.186
-0.122,	-0.121,	-0.121	0.122,	0.119,	0.119	-0.107,	-0.116,	-0.116
-0.094,	-0.094,	-0.094	0.093,	0.090,	0.09	-0.082,	-0.088,	-0.088
-0.152,	-0.152,	-0.152	0.116,	0.111,	0.111	-0.134,	-0.143,	-0.143
-0.094,	-0.094,	-0.094	0.093,	0.090,	0.09	-0.082,	-0.088,	-0.088
-0.063,	-0.065,	-0.065	0.062,	0.060,	0.06	-0.056,	-0.060,	-0.06
-0.101,	-0.104,	-0.104	0.078,	0.074,	0.074	-0.091,	-0.097,	-0.097
-0.063,	-0.065,	-0.065	0.062,	0.060,	0.06	-0.056,	-0.060,	-0.06
-0.031,	-0.033,	-0.033	0.031,	0.030,	0.03	-0.028,	-0.030,	-0.03
-0.050,	-0.053,	-0.053	0.039,	0.036,	0.036	-0.046,	-0.049,	-0.049
-0.031,	-0.033,	-0.033	0.031,	0.030,	0.03	-0.028,	-0.030,	-0.03
0.000,	0.000,	0	-0.000,	0.000,	0	0.000,	0.000,	0

End time	20	
0.000,	0.000,	0
-0.041,	-0.043,	-0.043
-0.060,	-0.063,	-0.063
-0.041,	-0.043,	-0.043
-0.083,	-0.088,	-0.088
-0.120,	-0.128,	-0.128
-0.083,	-0.088,	-0.088
-0.124,	-0.131,	-0.131
-0.179,	-0.191,	-0.191
-0.124,	-0.131,	-0.131
-0.164,	-0.172,	-0.172
-0.237,	-0.251,	-0.251
-0.164,	-0.172,	-0.172
-0.202,	-0.210,	-0.21
-0.294,	-0.306,	-0.306
-0.202,	-0.210,	-0.21
-0.238,	-0.241,	-0.241
-0.346,	-0.351,	-0.351
-0.238,	-0.241,	-0.241
-0.268,	-0.266,	-0.266
-0.390,	-0.387,	-0.387
-0.268,	-0.266,	-0.266
-0.289,	-0.283,	-0.283
-0.421,	-0.412,	-0.412
-0.289,	-0.283,	-0.283
-0.302,	-0.293,	-0.293
-0.440,	-0.426,	-0.426
-0.302,	-0.293,	-0.293
-0.306,	-0.296,	-0.296
-0.446,	-0.431,	-0.431
-0.306,	-0.296,	-0.296
-0.302,	-0.293,	-0.293
-0.440,	-0.426,	-0.426
-0.302,	-0.293,	-0.293
-0.289,	-0.283,	-0.283
-0.421,	-0.412,	-0.412
-0.289,	-0.283,	-0.283
-0.268,	-0.266,	-0.266
-0.390,	-0.387,	-0.387
-0.268,	-0.266,	-0.266
-0.238,	-0.241,	-0.241
-0.346,	-0.351,	-0.351
-0.238,	-0.241,	-0.241
-0.202,	-0.210,	-0.21
-0.294,	-0.306,	-0.306
-0.202,	-0.210,	-0.21

-0.164,	-0.172,	-0.172
-0.237,	-0.251,	-0.251
-0.164,	-0.172,	-0.172
-0.124,	-0.131,	-0.131
-0.179,	-0.191,	-0.191
-0.124,	-0.131,	-0.131
-0.083,	-0.088,	-0.088
-0.120,	-0.128,	-0.128
-0.083,	-0.088,	-0.088
-0.041,	-0.043,	-0.043
-0.060,	-0.063,	-0.063
-0.041,	-0.043,	-0.043
0.000,	-0.000,	0

```
/*
 * ME 2054 Parallel Scientific Computing
 * Project 1 - Finite Difference Solution of a Vibrating 2D Membrane on a GPU
 * Due: November 6,2018
 *
 * Author: Dustin (Ting-Hsuan) Ma
 *
 * Compile: nvcc -O2 GPUmembrane.cu -o GPUrun.exe
 * Clang: ../clang-format -i GPUmembrane.cu
 */

#include "timer.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

#define LX 4.0f
#define LY 2.0f
#define NX 41.0f
#define NY 21.0f
#define DX LX / (NX - 1)
#define DY LY / (NY - 1)
#define H DX

#define C sqrt(5.f)
#define DT 0.4f * (H / C)

#define ENDTIME 20.0f
#define INFINITE 200

// Making calculation part easier
#define IC i + j *NX
#define IP1 (i + 1) + j *NX
#define IM1 (i - 1) + j *NX
#define JP1 i + (j + 1) * NX
#define JM1 i + (j - 1) * NX

#define BLOCKSIZE 16

typedef float REAL;
typedef int   INT;

__global__ void GPU_calculatingWave(REAL *now, REAL *old, REAL *out)
{
        // Row index and Colum index
        INT i = blockIdx.x * blockDim.x + threadIdx.x;
        INT j = blockIdx.y * blockDim.y + threadIdx.y;

        // Linear indexing
        INT ic = i + j * NX, ip1 = (i + 1) + j * NX, im1 = (i - 1) + j * NX, jp1 = i + (j + 1) * NX,
            jm1 = i + (j - 1) * NX;

        __syncthreads();

        if (i > 0 && i < NX - 1) {
                if (j > 0 && j < NY - 1) {
                        out[ic] = 2.f * now[ic] - old[ic]
                                  + ((C * C * DT * DT) / (H * H))
                                    * (now[ip1] + now[im1] + now[jp1] + now[jm1] - 4.f * now[ic]);
                }
        }
}

void CPU_calculatingWave(REAL *now, REAL *old, REAL *out)
{
        for (INT j = 1; j < NY - 1; j++) {
                for (INT i = 1; i < NX - 1; i++) {
                        INT ic = IC, ip1 = IP1, im1 = IM1, jp1 = JP1, jm1 = JM1;
                        out[ic] = 2.f * now[ic] - old[ic]
                                  + ((C * C * DT * DT) / (H * H))
                                    * (now[ip1] + now[im1] + now[jp1] + now[jm1] - 4.f * now[ic]);
                }
        }
}

void initializeMatrices(REAL *in)
{
        INT  i, j, idx;
        REAL x, y = 0.0f;

        for (j = 0; j < NY; j++) {
                x = 0.0f;
                for (i = 0; i < NX; i++) {
                        idx = IC;
                        // Eq 2.
```



```
                        in[idx] = 0.1f * (4.f * x - (x * x)) * (2.f * y - (y * y));
                        x += DX;
                }
                y += DY;
        }
}

void applyingBoundary(REAL *in)
{
        INT i, j, idx;
        for (j = 0; j < NY; j++) {
                for (i = 0; i < NX; i++) {
                        idx = IC;
                        // Eq 4 - 7
                        if (i == 0 || i == NX) in[idx] = 0.0f;
                        if (j == 0 || j == NY) in[idx] = 0.0f;
                }
        }
}

void initializeSolution(REAL *in, REAL *out)
{
        for (INT j = 1; j < NY - 1; j++) {
                for (INT i = 1; i < NX - 1; i++) {
                        INT ic = IC, ip1 = IP1, im1 = IM1, jp1 = JP1, jm1 = JM1;

                        out[ic] = in[ic]
                                  + (0.5 * (C * C * DT * DT) / (H * H))
                                    * (in[ip1] + in[im1] + in[jp1] + in[jm1] - 4.f * in[ic]);
                }
        }
}

void analyticalSolution(REAL *out)
{
        INT  idx, i, j;
        REAL x, m, n, y = 0;

        for (j = 0; j < NY; j++) {
                x = 0.f;
                for (i = 0; i < NX; i++) {
                        idx = IC;
                        for (m = 1.f; m <= INFINITE; m += 2.f) {
                                for (n = 1.f; n <= INFINITE; n += 2.f) {
                                        out[idx] += 0.426050f / (m * m * m * n * n * n)
                                                    * cos(ENDTIME * sqrt(5.f) * M_PI / 4.f * sqrt(m
* m + 4.f * n * n))
                                                    * sin(m * M_PI * x / 4.f) * sin(n * M_PI * y / 2
.f);
                                }
                        }
                        x += DX;
                }
                y += DY;
        }
}

void outputMatrix(REAL *in)
{
        INT i, j, idx;
        for (j = 0; j < NY; j++) {
                for (i = 0; i < NX; i++) {
                        idx = i + j * NX;
                        printf("%7.3f ", in[idx]);
                }
                printf("\n");
        }
        printf("\n");
}

void outputComparisonResult(REAL *exact, REAL *CPU, REAL *GPU)
{
        INT i, j, idx;
        for (j = 0; j < NY; j++) {
                for (i = 0; i < NX; i++) {
                        idx = i + j * NX;
                        if (i == 10 || i == 20 || i == 30) {
                                printf("%7.3f, %7.3f, %7.3f \n", exact[idx], CPU[idx], GPU[idx]);
                        }
                }
        }
}

INT main()
{
        printf("End time = %f\n", ENDTIME);
```

```
	/*
	    // Running CFL check
	    if (sqrt(C) * DT / H < 1.0f) {
	        printf("CFL condition is met\n");
	    } else {
	        printf("CFL condition is not met, try again\n");
	        return EXIT_SUCCESS;
	    }
	*/
	// Calculating Analitical Solution
	REAL *Exact_phi = (REAL *) calloc(NX * NY, sizeof(*Exact_phi));
	analyticalSolution(Exact_phi);
	/*    printf(
	    "==============================Exact Solution==============================\n");
	    outputMatrix(Exact_phi);
	*/
	// Allocating memory for CPU
	REAL *phi     = (REAL *) calloc(NX * NY, sizeof(*phi));
	REAL *phi_old = (REAL *) calloc(NX * NY, sizeof(*phi_old));
	REAL *phi_new = (REAL *) calloc(NX * NY, sizeof(*phi_new));

	// Initializing Mesh, boundaries, and solution
	initializeMatrices(phi);
	applyingBoundary(phi);
	initializeSolution(phi, phi_old);

	// Starting time
	double start, finish;

	GET_TIME(start);

	// Solving time function until endtime is reached
	REAL time = 0.0f, *tmp_CPU;
	while (time < ENDTIME) {
	        CPU_calculatingWave(phi, phi_old, phi_new);

	        tmp_CPU = phi;
	        phi     = phi_new;
	        phi_new = phi_old;
	        phi_old = tmp_CPU;

	        time += DT;
	}
	GET_TIME(finish);
	/*
	    // Outputing CPU solution
	    printf("==========================Final Solution CPU==========================\n");
	    outputMatrix(phi);
	    printf("elapsed wall time (Host) = %3.1f ms\n", (finish - start) * 1000);
	    printf("\n");
	*/
	// Allocating memory for GPU
	REAL *phi_d, *phi_old_d, *phi_new_d;
	cudaMallocManaged(&phi_d, NX * NY * sizeof(*phi_d));
	cudaMallocManaged(&phi_new_d, NX * NY * sizeof(*phi_new_d));
	cudaMallocManaged(&phi_old_d, NX * NY * sizeof(*phi_old_d));

	// Restarting Problem for GPU
	initializeMatrices(phi_d);
	applyingBoundary(phi_d);
	initializeSolution(phi_d, phi_old_d);

	// Setting up device enviromnet
	dim3 dimBlock(BLOCKSIZE, BLOCKSIZE);
	dim3 dimGrid(NX / dimBlock.x + 1, NY / dimBlock.y + 1);

	// Time enviorment to test cuda speed up
	REAL        elapsedTime;      // records in [ms]
	cudaEvent_t timeStart, timeStop; // cudaEvent_t initializes variable used in event time
	cudaEventCreate(&timeStart);
	cudaEventCreate(&timeStop);
	cudaEventRecord(timeStart, 0);

	// Solving time function until endtime is reached
	REAL Itertime = 0.0f, *tmp_GPU;

	while (Itertime < ENDTIME) {
	        GPU_calculatingWave<<<dimGrid, dimBlock>>>(phi_d, phi_old_d, phi_new_d);
	        cudaDeviceSynchronize();

	        tmp_GPU   = phi_d;
	        phi_d     = phi_new_d;
	        phi_new_d = phi_old_d;
	        phi_old_d = tmp_GPU;

	        Itertime += DT;
```



```
	}

	cudaEventRecord(timeStop, 0);
	cudaEventSynchronize(timeStop);
	cudaEventElapsedTime(&elapsedTime, timeStart, timeStop);
	/*
	    // Outputting Matrix
	    printf("==========================Final Solution GPU==========================\n");
	    outputMatrix(phi_d);
	    printf("elapsed wall time (Device) = %3.1f ms\n", elapsedTime);
	*/
	outputComparisonResult(Exact_phi, phi, phi_d);

	// Deallocating memory
	cudaEventDestroy(timeStart);
	cudaEventDestroy(timeStop);

	free(phi);
	free(phi_old);
	free(phi_new);
	free(Exact_phi);
	cudaFree(phi_d);
	cudaFree(phi_new_d);
	cudaFree(phi_old_d);
	phi       = NULL;
	phi_old   = NULL;
	phi_new   = NULL;
	Exact_phi = NULL;

	return EXIT_SUCCESS;
}
```

References

Web.archive.org. (2018). HMPP Competence Center | Addressing the Many-Core Computing Challenge. [online] Available at: https://web.archive.org/web/20130616205308/http://openhmpp.org/ [Accessed 7 Nov. 2018].

W. H. (www.winkhosting.com), "OpenCL Compiler, Embedded Source, Encryption," ClusterChimps.org. [Online]. Available: https://web.archive.org/web/20111101184143/http://www.clusterchimps.org/ocltools.html. [Accessed: 07-Nov-2018].

NVIDIA Developer, 10-Oct-2018. [Online]. Available: https://developer.nvidia.com/nvidia-developer-zone. [Accessed: 07-Nov-2018]