Praise be to Allah

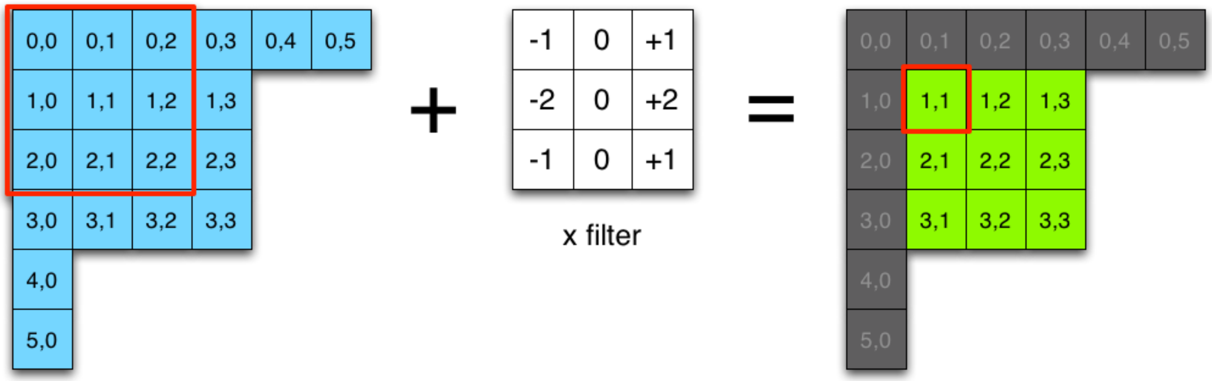
Parallel Implementation of Sobel Filter

Ahmad Siavashi
Email: a.siavosh@yahoo.com
Ali Kianinejad
Email: af.kianinejad@gmail.com
Department of Computer Science and Engineering
School of Engineering, Shiraz University, Shiraz, Iran

Submitted to—
Prof. Farshad Khunjush
Department of Computer Science and Engineering
School of Engineering, Shiraz University, Shiraz, Iran

Winter 2014

Sobel Operator is among the most popular edge detection algorithms. Besides being popular it is easy to implement. Thus it is important to optimize the algorithm so that it takes less time and memory to accomplish its task.
The way it works can be summarized in this picture:



Original image

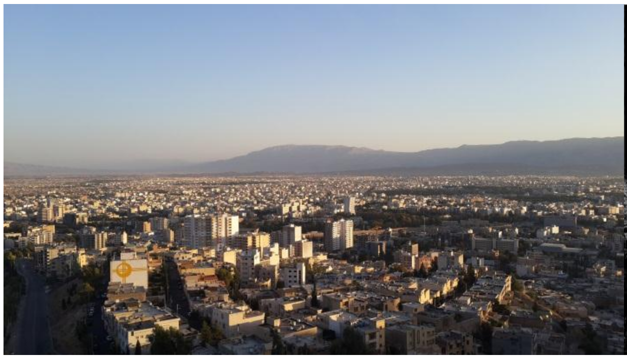
Edge detected image

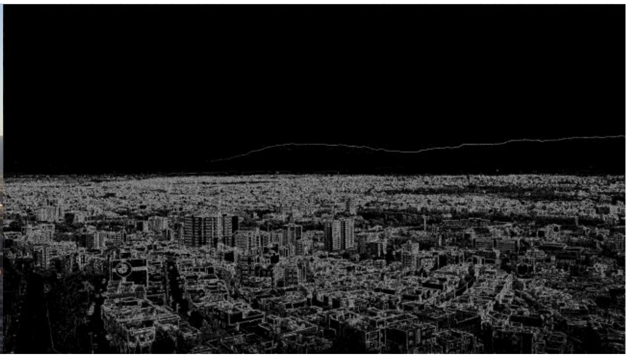
We have made several attempts to optimize the filter taking advantage of our limited knowledge in Parallelization.

All of the following implementations use a single picture as their standard input. For simplicity purposes inputs are converted to greyscale images using FreeImage library.

Input

Output





Name	Dimensions	Type	Bit Depth	Color Representation	Size
shiraz.jpg	4128*2322	JPEG	24	sRGB	3,774,591 bytes

System specifications

CPU	Intel core i3 3110m 2.40GHz
GPU	nVidia GeForce 635m
RAM	4GB DDR3

All of the implementations' code can be found in appendix A.

Sequential Implementation on CPU

- **Average Elapsed Time: 3.834s**

Initial Kernel Launch Configuration
Grid Dim { Width / 32 , Height / 32 }
Block Dim { 32 , 32 }

CUDA Implementation

- **Naïve Implementation**

  Average Elapsed Time: 0.508

- **Using Pinned Memory**

  Average Elapsed Time: 0.402

- **Using 2D Batched Pinned Memory**

  Average Elapsed Time: 0.398~0.402s

- **Using Zero Copy Memory**

  Average Elapsed Time: 0.384s

- **Kernel Modifications ($\sqrt{Gx^2 + Gy^2} = \sqrt{Gx * Gx + Gy * Gy}$) , Branch Elimination, Using More Registers in Kernel**

  Average Elapsed Time: 0.036s

- **Using Texture Memory and cudaArray**

  Due to pictures spatial locality, using texture memory improves the overall performance. Unfortunately duo to technical flaws we were not able to use texture memory.

Guidelines which we have used during host-device data transfer optimization, according to nVidia developers FAQ:

- Minimize the amount of data transferred between host and device when possible, even if that means running kernels on the GPU that get little or no speed-up compared to running them on the host CPU.
- Higher bandwidth is possible between the host and the device when using page-locked (or "pinned") memory.
- Batching many small transfers into one larger transfer performs much better because it eliminates most of the per-transfer overhead.
- Data transfers between the host and device can sometimes be overlapped with kernel execution and other data transfers.

OpenCL Implementation

NDRange Size {Width, Height} , Work-Group Size {is left for the runtime environment to decide}

- **Naïve Implementation**

  Average Elapsed Time: 1.886

- **Using Pinned Memory**

  OpenCL applications do not have direct control over whether memory objects are allocated in page-locked memory or not, but they can create objects using CL_MEM_ALLOC_HOST_PTR flag and such objects are likely to be allocated in page-locked memory by the driver for best performance.

  Average Elapsed Time: 1.885

- **Using Zero Copy Memory (Mapped Memory Objects)**

  Average Elapsed Time: 0.887s

- **Kernel Modifications ($\sqrt{Gx^2 + Gy^2} = \sqrt{Gx * Gx + Gy * Gy}$) , Branch Elimination, Using More Registers in Kernel**

  Average Elapsed Time: 0.466s

- **Using Pinned Texture Memory (Image Memory Objects) + Vector Data Types**

  Average Elapsed Time: 0.668

  Duo to the nature of nVidia architectures in comparison to AMD architectures, they show a significant increase in time while working with vector data types.

- **Using Zero Copy Texture Memory (Image Memory Objects) + Vector Data Types**

  Average Elapsed Time: 0.650

  We could not achieve any improvements duo to our limited knowledge in using Vector Data Types and Texture Memory.

	I3-3110M	**CUDA 635M**	**OpenCL 635M**	**OpenMP i3-3110M**
Naïve	**2.834**	**0.508**	**1.886**	
Naïve + Kernel modification + 4 threads				**1.512**
Pinned Memory		**0.402**	**1.885**	
2D Batched Pinned Memory		**0.402**		
Zero-Copy Memory		**0.384**	**0.887**	
Kernel Modification		**0.036**	**0.466**	
Pinned Texture Memory + Vector Data Types			**0.668**	
Zero-Copy Texture Memory + Vector Data Types			**0.650**	

Best speed up was achieved using CUDA: 78.7X Speed-Up in Comparison to the sequential implementation.

We could not work further on the OpenMP version ☹
We also planned to run the OpenCL kernel on "Intel HD 4000" GPU and "Intel Core i3-3110M" CPU that together form an APU (Which reduces the data transfer stage in the program), but unfortunately we were overloaded by several other projects ☹

Appendix A

Sequential Code (C style C++)

```cpp
#include <iostream>
#include <cmath>
#include <ctime>
#include <Windows.h>
#include <FreeImage.h>
#include <string>

using namespace std;

unsigned int height,width,pitch;

void sobel(BYTE *image,BYTE *G){
       for(int i = 1 ; i <  width - 1 ; i++){
              for(int j = 1; j < height -1 ; j++){
                     int Gx = image[(i+1) + (j-1)*pitch] + 2*image[(i+1) + (j)*pitch] + 
image[(i+1) + (j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i-1) + (j)*pitch] - 
image[(i-1) + (j+1)*pitch];
                     int Gy = image[(i-1) + (j+1)*pitch] + 2*image[(i) + (j+1)*pitch] + 
image[(i+1) + (j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i) + (j-1)*pitch] - 
image[(i+1) + (j-1)*pitch];
                     G[i + j*width] = sqrtf(powf(Gx,2) + powf(Gy,2));
              }
       }
}


int main(){
       string fileName;
       clock_t t;
       FREE_IMAGE_FORMAT fif;
       FIBITMAP *bitmap;

       BYTE * in_picture,* out_picture;


       cin >> fileName;
       fif = FreeImage_GetFileType(fileName.c_str());
       bitmap = FreeImage_Load(fif,fileName.c_str());
       pitch = FreeImage_GetPitch(bitmap);
       height = FreeImage_GetHeight(bitmap);
       width = FreeImage_GetWidth(bitmap);
       bitmap = FreeImage_ConvertToGreyscale(bitmap);
       in_picture =  (BYTE *) malloc(sizeof(BYTE)*pitch*height);
       out_picture =  (BYTE *) calloc(sizeof(BYTE),width*height);

       FreeImage_ConvertToRawBits(in_picture,bitmap,pitch,8,0,0,0,TRUE);

       /**************/
       t = clock();

       sobel(in_picture,out_picture);
```

```
        cout << "Elapsed Time = " << ((float)clock()-t)/CLOCKS_PER_SEC << "s" << endl;
        /**************/

        FIBITMAP *dst =
FreeImage_ConvertFromRawBits(out_picture,width,height,width,8,0,0,0,TRUE);
        string outputName = fileName + "_output.";
        outputName.append(FreeImage_GetFormatFromFIF(fif));
        FreeImage_Save(fif,dst,outputName.c_str(),0);

        // Free memory allocation.
        free(in_picture);
        free(out_picture);
        FreeImage_Unload(bitmap);

        return EXIT_SUCCESS;
}
```

OpenCL – Naïve

```
#include <iostream>
#include <string>
#include <ctime>
#include <CL\cl.h>
#include <FreeImage.h>

using namespace std;

int main(){
        cl_platform_id platform_id;
        cl_uint num_platforms;
        cl_device_id device_id;
        cl_uint num_devices;
        cl_context context;
        cl_context_properties context_properties[3];
        cl_command_queue command_queue;
        cl_program program;
        cl_kernel kernel;
        cl_mem inputImage,outputImage;
        cl_int err;

        size_t global[2];

        char *kernelSource;

        FILE *pFile;
        int fileLen,readLen;

        char buffer[1024];
        char buildBuffer[2048];
        size_t buildLogLen;

        FREE_IMAGE_FORMAT fif;
        FIBITMAP *bitmap;
        unsigned int height,width,pitch;
        BYTE *picture;
```

```
       if(clGetPlatformIDs(1,&platform_id,&num_platforms) != CL_SUCCESS)
              printf("[-] Error : GetPlatformIDs\n");
       if(clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_GPU,1,&device_id,&num_devices) !=
CL_SUCCESS)
              printf("[-] Error : GetDeviceIDs\n");
       clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(buffer), buffer, NULL);
       cout << buffer << endl;
       context_properties[0] = CL_CONTEXT_PLATFORM;
       context_properties[1] = (cl_context_properties) platform_id;
       context_properties[2] = 0;
       context = clCreateContext(context_properties,1,&device_id,NULL,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateContext\n");

       command_queue = clCreateCommandQueue(context,device_id,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateCommandQueue\n");

       pFile = fopen("sobel_kernel.cl","r");
       fseek(pFile,0L,SEEK_END);
       fileLen = ftell(pFile);
       rewind(pFile);
       kernelSource = (char *) malloc(sizeof(char) * fileLen);
       readLen = fread(kernelSource,1,fileLen,pFile);
       kernelSource[readLen] = '\0';
       fclose(pFile);

       program = clCreateProgramWithSource(context,1,(const char
**)&kernelSource,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateProgramWithSource\n");
       if(clBuildProgram(program,0,NULL,NULL,NULL,NULL) != CL_SUCCESS){
              printf("[-] Error : BuildProgram\n");

       clGetProgramBuildInfo(program,device_id,CL_PROGRAM_BUILD_LOG,sizeof(buildBuffer),b
uildBuffer,&buildLogLen);
              printf("%s\n",buildBuffer);
       }

       free(kernelSource);

       kernel = clCreateKernel(program,"sobel",&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateKernel\n");

       //////////////////////
       string fileName;
       cin >> fileName;
       fif = FreeImage_GetFileType(fileName.c_str());
       bitmap = FreeImage_Load(fif,fileName.c_str());
       pitch = FreeImage_GetPitch(bitmap);
       height = FreeImage_GetHeight(bitmap);
       width = FreeImage_GetWidth(bitmap);
       bitmap = FreeImage_ConvertToGreyscale(bitmap);
       picture =  (BYTE *) malloc(pitch*height);
       FreeImage_ConvertToRawBits(picture,bitmap,pitch,8,FI_RGBA_RED_MASK,FI_RGBA_GREEN_M
ASK,FI_RGBA_BLUE_MASK,TRUE);
       //////////////////////
```

```
        double t = clock();


        inputImage = clCreateBuffer(context,CL_MEM_READ_ONLY ,sizeof(cl_char) * pitch *
height,NULL,&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : CreateBuffer\n");
        if(clEnqueueWriteBuffer(command_queue,inputImage,CL_TRUE,0,sizeof(cl_char)*pitch*h
eight,picture,0,NULL,NULL) != CL_SUCCESS)
                printf("[-] Error : EnqueueWriteBuffer\n");
        outputImage = clCreateBuffer(context,CL_MEM_WRITE_ONLY,sizeof(cl_char) * width *
height,NULL,&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : CreateBuffer\n");

        if(
                clSetKernelArg(kernel,0,sizeof(cl_mem),&inputImage) ||
                clSetKernelArg(kernel,1,sizeof(cl_mem),&outputImage) ||
                clSetKernelArg(kernel,2,sizeof(cl_int),&width)   ||
                clSetKernelArg(kernel,3,sizeof(cl_int),&pitch)
                )
                printf("[-] Error : SetKernelArg\n");

        global[0] = width - 1;
        global[1] = height - 1;
        if(clEnqueueNDRangeKernel(command_queue,kernel,2,NULL,global,0,0,NULL,NULL) !=
CL_SUCCESS)
                printf("[-] Error : EnqueueNDRangeKernel\n");

        clFinish(command_queue);

        if(clEnqueueReadBuffer(command_queue,outputImage,CL_TRUE,0,sizeof(cl_char)*width*h
eight,picture,0,NULL,NULL) != CL_SUCCESS)
                printf("[-] Error : EnqueueReadBuffer\n");


        // Elapsed time calculation.
        cout << "Elapsed Time = " << (clock()-t)/CLOCKS_PER_SEC << "s" << endl;

        //////////////////////////
        FIBITMAP *dst =
FreeImage_ConvertFromRawBits(picture,width,height,width,8,FI_RGBA_RED_MASK,FI_RGBA_GREEN_
MASK,FI_RGBA_BLUE_MASK,TRUE);
        string outputName = fileName + "_output.";
        outputName.append(FreeImage_GetFormatFromFIF(fif));
        FreeImage_Save(fif,dst,outputName.c_str(),0);

        free(picture);
        FreeImage_Unload(bitmap);
        clReleaseMemObject(inputImage);
        clReleaseMemObject(outputImage);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(command_queue);
        clReleaseContext(context);

        return EXIT_SUCCESS;
```

```
}
```

OpenCL – Naïve – Kernel

```
__kernel void sobel(__global unsigned char *image, __global unsigned char *G, const
unsigned int width,const unsigned int pitch){
       int i = get_global_id(0)+1;
       int j = get_global_id(1)+1;
       int Gx = image[(i+1) + (j-1)*pitch] + 2*image[(i+1) + (j)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i-1) + (j)*pitch] - image[(i-1) +
(j+1)*pitch];
       int Gy = image[(i-1) + (j+1)*pitch] + 2*image[(i) + (j+1)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i) + (j-1)*pitch] - image[(i+1) +
(j-1)*pitch];
       G[i + j*width] = sqrtf((float)powf(Gx,2) + powf(Gy,2));
 }
```

OpenCL – Pinned Memory

```
#include <iostream>
#include <string>
#include <ctime>
#include <CL\cl.h>
#include <FreeImage.h>

using namespace std;

int main(){
       cl_platform_id platform_id;
       cl_uint num_platforms;
       cl_device_id device_id;
       cl_uint num_devices;
       cl_context context;
       cl_context_properties context_properties[3];
       cl_command_queue command_queue;
       cl_program program;
       cl_kernel kernel;
       cl_mem inputImage,outputImage;
       cl_int err;

       size_t global[2];

       char *kernelSource;

       FILE *pFile;
       int fileLen,readLen;

       char buffer[1024];
       char buildBuffer[2048];
       size_t buildLogLen;

       FREE_IMAGE_FORMAT fif;
       FIBITMAP *bitmap;
       unsigned int height,width,pitch;
       BYTE *picture;
```

```
        if(clGetPlatformIDs(1,&platform_id,&num_platforms) != CL_SUCCESS)
                printf("[-] Error : GetPlatformIDs\n");
        if(clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_GPU,1,&device_id,&num_devices) !=
CL_SUCCESS)
                printf("[-] Error : GetDeviceIDs\n");
        clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(buffer), buffer, NULL);
        cout << buffer << endl;
        context_properties[0] = CL_CONTEXT_PLATFORM;
        context_properties[1] = (cl_context_properties) platform_id;
        context_properties[2] = 0;
        context = clCreateContext(context_properties,1,&device_id,NULL,NULL,&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : CreateContext\n");

        command_queue = clCreateCommandQueue(context,device_id,NULL,&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : CreateCommandQueue\n");

        pFile = fopen("sobel_kernel.cl","r");
        fseek(pFile,0L,SEEK_END);
        fileLen = ftell(pFile);
        rewind(pFile);
        kernelSource = (char *) malloc(sizeof(char) * fileLen);
        readLen = fread(kernelSource,1,fileLen,pFile);
        kernelSource[readLen] = '\0';
        fclose(pFile);

        program = clCreateProgramWithSource(context,1,(const char
**)&kernelSource,NULL,&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : CreateProgramWithSource\n");
        if(clBuildProgram(program,0,NULL,NULL,NULL,NULL) != CL_SUCCESS){
                printf("[-] Error : BuildProgram\n");

        clGetProgramBuildInfo(program,device_id,CL_PROGRAM_BUILD_LOG,sizeof(buildBuffer),b
uildBuffer,&buildLogLen);
                printf("%s\n",buildBuffer);
        }

        free(kernelSource);

        kernel = clCreateKernel(program,"sobel",&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : CreateKernel\n");

        //////////////////////
        string fileName;
        cin >> fileName;
        fif = FreeImage_GetFileType(fileName.c_str());
        bitmap = FreeImage_Load(fif,fileName.c_str());
        pitch = FreeImage_GetPitch(bitmap);
        height = FreeImage_GetHeight(bitmap);
        width = FreeImage_GetWidth(bitmap);
        bitmap = FreeImage_ConvertToGreyscale(bitmap);
        picture =  (BYTE *) malloc(pitch*height);
        FreeImage_ConvertToRawBits(picture,bitmap,pitch,8,FI_RGBA_RED_MASK,FI_RGBA_GREEN_M
ASK,FI_RGBA_BLUE_MASK,TRUE);
```

```
        //////////////////////
        double t = clock();


        inputImage = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR
,sizeof(cl_char) * pitch * height,NULL,&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : CreateBuffer\n");
        if(clEnqueueWriteBuffer(command_queue,inputImage,CL_TRUE,0,sizeof(cl_char)*pitch*h
eight,picture,0,NULL,NULL) != CL_SUCCESS)
                printf("[-] Error : EnqueueWriteBuffer\n");
        outputImage = clCreateBuffer(context,CL_MEM_WRITE_ONLY,sizeof(cl_char) * width *
height,NULL,&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : CreateBuffer\n");

        if(
                clSetKernelArg(kernel,0,sizeof(cl_mem),&inputImage) ||
                clSetKernelArg(kernel,1,sizeof(cl_mem),&outputImage) ||
                clSetKernelArg(kernel,2,sizeof(cl_int),&width)   ||
                clSetKernelArg(kernel,3,sizeof(cl_int),&pitch)
                )
                printf("[-] Error : SetKernelArg\n");

        global[0] = width - 1;
        global[1] = height - 1;
        if(clEnqueueNDRangeKernel(command_queue,kernel,2,NULL,global,0,0,NULL,NULL) !=
CL_SUCCESS)
                printf("[-] Error : EnqueueNDRangeKernel\n");

        clFinish(command_queue);

        if(clEnqueueReadBuffer(command_queue,outputImage,CL_TRUE,0,sizeof(cl_char)*width*h
eight,picture,0,NULL,NULL) != CL_SUCCESS)
                printf("[-] Error : EnqueueReadBuffer\n");


        // Elapsed time calculation.
        cout << "Elapsed Time = " << (clock()-t)/CLOCKS_PER_SEC << "s" << endl;

        //////////////////////////
        FIBITMAP *dst =
FreeImage_ConvertFromRawBits(picture,width,height,width,8,FI_RGBA_RED_MASK,FI_RGBA_GREEN_
MASK,FI_RGBA_BLUE_MASK,TRUE);
        string outputName = fileName + "_output.";
        outputName.append(FreeImage_GetFormatFromFIF(fif));
        FreeImage_Save(fif,dst,outputName.c_str(),0);

        free(picture);
        FreeImage_Unload(bitmap);
        clReleaseMemObject(inputImage);
        clReleaseMemObject(outputImage);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(command_queue);
        clReleaseContext(context);

        return EXIT_SUCCESS;
```

```
}
```

OpenCL – Zero Copy Memory

```
#include <iostream>
#include <string>
#include <ctime>
#include <CL\cl.h>
#include <FreeImage.h>

using namespace std;

int main(){
	cl_platform_id platform_id;
	cl_uint num_platforms;
	cl_device_id device_id;
	cl_uint num_devices;
	cl_context context;
	cl_context_properties context_properties[3];
	cl_command_queue command_queue;
	cl_program program;
	cl_kernel kernel;
	cl_mem inputImage,outputImage;
	cl_int err;

	size_t global[2];

	char *kernelSource;

	FILE *pFile;
	int fileLen,readLen;

	char buffer[1024];
	char buildBuffer[2048];
	size_t buildLogLen;

	FREE_IMAGE_FORMAT fif;
	FIBITMAP *bitmap;
	unsigned int height,width,pitch;
	BYTE *picture;
	BYTE *out_picture;

	if(clGetPlatformIDs(1,&platform_id,&num_platforms) != CL_SUCCESS)
		printf("[-] Error : GetPlatformIDs\n");
	if(clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_GPU,1,&device_id,&num_devices) != 
CL_SUCCESS)
		printf("[-] Error : GetDeviceIDs\n");
	clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(buffer), buffer, NULL);
	cout << buffer << endl;
	context_properties[0] = CL_CONTEXT_PLATFORM;
	context_properties[1] = (cl_context_properties) platform_id;
	context_properties[2] = 0;
	context = clCreateContext(context_properties,1,&device_id,NULL,NULL,&err);
	if(err != CL_SUCCESS)
		printf("[-] Error : CreateContext\n");
```

```
       command_queue = clCreateCommandQueue(context,device_id,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateCommandQueue\n");

       pFile = fopen("sobel_kernel.cl","r");
       fseek(pFile,0L,SEEK_END);
       fileLen = ftell(pFile);
       rewind(pFile);
       kernelSource = (char *) malloc(sizeof(char) * fileLen);
       readLen = fread(kernelSource,1,fileLen,pFile);
       kernelSource[readLen] = '\0';
       fclose(pFile);

       program = clCreateProgramWithSource(context,1,(const char
**)&kernelSource,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateProgramWithSource\n");
       if(clBuildProgram(program,0,NULL,NULL,NULL,NULL) != CL_SUCCESS){
              printf("[-] Error : BuildProgram\n");

       clGetProgramBuildInfo(program,device_id,CL_PROGRAM_BUILD_LOG,sizeof(buildBuffer),b
uildBuffer,&buildLogLen);
              printf("%s\n",buildBuffer);
       }

       free(kernelSource);

       kernel = clCreateKernel(program,"sobel",&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateKernel\n");

       //////////////////////
       string fileName;
       cin >> fileName;
       fif = FreeImage_GetFileType(fileName.c_str());
       bitmap = FreeImage_Load(fif,fileName.c_str());
       pitch = FreeImage_GetPitch(bitmap);
       height = FreeImage_GetHeight(bitmap);
       width = FreeImage_GetWidth(bitmap);
       bitmap = FreeImage_ConvertToGreyscale(bitmap);
       //////////////////////
       inputImage = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR
,sizeof(cl_char) * pitch * height,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateBuffer\n");
       picture = (BYTE *)
clEnqueueMapBuffer(command_queue,inputImage,CL_TRUE,CL_MAP_READ,0,sizeof(cl_char) * pitch
* height,0,NULL,NULL,NULL);
       FreeImage_ConvertToRawBits(picture,bitmap,pitch,8,FI_RGBA_RED_MASK,FI_RGBA_GREEN_M
ASK,FI_RGBA_BLUE_MASK,TRUE);

       outputImage = clCreateBuffer(context,CL_MEM_WRITE_ONLY |
CL_MEM_ALLOC_HOST_PTR,sizeof(cl_char) * width * height,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateBuffer\n");
       out_picture = (BYTE *)
clEnqueueMapBuffer(command_queue,outputImage,CL_TRUE,CL_MAP_WRITE,0,sizeof(cl_char) *
width * height,0,NULL,NULL,NULL);
```

```
        //////////////////////
        double t = clock();

        if(clEnqueueWriteBuffer(command_queue,inputImage,CL_FALSE,0,sizeof(cl_char)*pitch*
height,picture,0,NULL,NULL) != CL_SUCCESS)
                printf("[-] Error : EnqueueWriteBuffer\n");

        if(
                clSetKernelArg(kernel,0,sizeof(cl_mem),&inputImage) ||
                clSetKernelArg(kernel,1,sizeof(cl_mem),&outputImage) ||
                clSetKernelArg(kernel,2,sizeof(cl_int),&width)   ||
                clSetKernelArg(kernel,3,sizeof(cl_int),&pitch)
                )
                printf("[-] Error : SetKernelArg\n");

        global[0] = width - 1;
        global[1] = height - 1;
        if(clEnqueueNDRangeKernel(command_queue,kernel,2,NULL,global,0,0,NULL,NULL) !=
CL_SUCCESS)
                printf("[-] Error : EnqueueNDRangeKernel\n");

        clFinish(command_queue);

        if(clEnqueueReadBuffer(command_queue,outputImage,CL_FALSE,0,sizeof(cl_char)*width*
height,out_picture,0,NULL,NULL) != CL_SUCCESS)
                printf("[-] Error : EnqueueReadBuffer\n");

        // Elapsed time calculation.
        cout << "Elapsed Time = " << (clock()-t)/CLOCKS_PER_SEC << "s" << endl;

        //////////////////////////
        FIBITMAP *dst =
FreeImage_ConvertFromRawBits(out_picture,width,height,width,8,FI_RGBA_RED_MASK,FI_RGBA_GR
EEN_MASK,FI_RGBA_BLUE_MASK,TRUE);
        string outputName = fileName + "_output.";
        outputName.append(FreeImage_GetFormatFromFIF(fif));
        FreeImage_Save(fif,dst,outputName.c_str(),0);

        FreeImage_Unload(bitmap);
        clReleaseMemObject(inputImage);
        clReleaseMemObject(outputImage);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(command_queue);
        clReleaseContext(context);

        return EXIT_SUCCESS;
}
```

OpenCL – Kernel Modification

```
__kernel void sobel(__global unsigned char *image, __global unsigned char *G, const
unsigned int width,const unsigned int pitch){
        int i = get_global_id(0)+1;
        int j = get_global_id(1)+1;
```

```
       int Gx = image[(i+1) + (j-1)*pitch] + 2*image[(i+1) + (j)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i-1) + (j)*pitch] - image[(i-1) +
(j+1)*pitch];
       int Gy = image[(i-1) + (j+1)*pitch] + 2*image[(i) + (j+1)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i) + (j-1)*pitch] - image[(i+1) +
(j-1)*pitch];
       G[i + j*width] = sqrtf(Gx*Gx + Gy*Gy);
 }
```

OpenCL – Pinned Texture Memory + Vector Data Type

```
#include <iostream>
#include <string>
#include <ctime>
#include <CL\cl.h>
#include <FreeImage.h>

using namespace std;

int main(){
       cl_platform_id platform_id;
       cl_uint num_platforms;
       cl_device_id device_id;
       cl_uint num_devices;
       cl_context context;
       cl_context_properties context_properties[3];
       cl_command_queue command_queue;
       cl_program program;
       cl_kernel kernel;
       cl_mem inputImage,outputImage;
       cl_int err;

       size_t global[2];

       char *kernelSource;

       FILE *pFile;
       int fileLen,readLen;

       char buffer[1024];
       char buildBuffer[2048];
       size_t buildLogLen;

       FREE_IMAGE_FORMAT fif;
       FIBITMAP *bitmap;
       unsigned int height,width,pitch;
       BYTE *picture;
       BYTE *out_picture;


       // The program always select the first platform and its first device.
       if(clGetPlatformIDs(1,&platform_id,&num_platforms) != CL_SUCCESS)
              printf("[-] Error : GetPlatformIDs\n");
       if(clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_GPU,1,&device_id,&num_devices) !=
CL_SUCCESS)
              printf("[-] Error : GetDeviceIDs\n");
       clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(buffer), buffer, NULL);
       cout << buffer << endl;
       context_properties[0] = CL_CONTEXT_PLATFORM;
       context_properties[1] = (cl_context_properties) platform_id;
       context_properties[2] = 0;
       context = clCreateContext(context_properties,1,&device_id,NULL,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateContext\n");

       command_queue = clCreateCommandQueue(context,device_id,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateCommandQueue\n");

       pFile = fopen("sobel_kernel.cl","r");
       fseek(pFile,0L,SEEK_END);
       fileLen = ftell(pFile);
       rewind(pFile);
       kernelSource = (char *) malloc(sizeof(char) * fileLen);
       readLen = fread(kernelSource,1,fileLen,pFile);
       kernelSource[readLen] = '\0';
       fclose(pFile);

       program = clCreateProgramWithSource(context,1,(const char
**)&kernelSource,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateProgramWithSource\n");
       if(clBuildProgram(program,0,NULL,NULL,NULL,NULL) != CL_SUCCESS){
              printf("[-] Error : BuildProgram\n");

       clGetProgramBuildInfo(program,device_id,CL_PROGRAM_BUILD_LOG,sizeof(buildBuffer),b
uildBuffer,&buildLogLen);
              printf("%s\n",buildBuffer);
       }

       free(kernelSource);

       kernel = clCreateKernel(program,"sobel",&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateKernel\n");

       //////////////////////
       string fileName;
       cin >> fileName;
       fif = FreeImage_GetFileType(fileName.c_str());
       bitmap = FreeImage_Load(fif,fileName.c_str());
       pitch = FreeImage_GetPitch(bitmap);
       height = FreeImage_GetHeight(bitmap);
       width = FreeImage_GetWidth(bitmap);
       picture = (BYTE*) malloc(sizeof(cl_float)*height*pitch);
       out_picture = (BYTE*) malloc(sizeof(BYTE)*height*width);
       bitmap = FreeImage_ConvertToGreyscale(bitmap);
       //////////////////////
       FreeImage_ConvertToRawBits(picture,bitmap,pitch,8,FI_RGBA_RED_MASK,FI_RGBA_GREEN_M
ASK,FI_RGBA_BLUE_MASK,TRUE);
       cl_image_format image_format = {CL_R,CL_UNSIGNED_INT8};

       double t = clock();
```

```
    inputImage = clCreateImage2D(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR |
CL_MEM_ALLOC_HOST_PTR,&image_format,pitch,height,0,picture,&err);

    if(err != CL_SUCCESS){
        printf("[-] Error : CreateImage2D\n");
        switch(err){
        case CL_INVALID_CONTEXT:
            cout << "Invalid Context" << endl;
            break;
        case CL_INVALID_VALUE :
            cout << "Invalid Value" << endl;
            break;
        case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
            cout << "Invalid Image Format Description" << endl;
            break;
        case CL_INVALID_IMAGE_SIZE:
            cout << "Invalid Image size" << endl;
            cout << "Max Dims = " << CL_DEVICE_IMAGE2D_MAX_WIDTH << " * " <<
CL_DEVICE_IMAGE2D_MAX_HEIGHT << endl;
            cout << "Your Dims = " << pitch << " * " << height << endl;
            break;
        case CL_INVALID_HOST_PTR:
            cout << "Invalid Host Ptr" << endl;
            break;
        case CL_IMAGE_FORMAT_NOT_SUPPORTED:
            cout << "Image Format Not Supported" << endl;
            break;
        case CL_MEM_OBJECT_ALLOCATION_FAILURE:
            cout << "Object Allocation Failure" << endl;
            break;
        case CL_INVALID_OPERATION:
            cout << "Invalid Operation" << endl;
            break;
        case CL_OUT_OF_HOST_MEMORY:
            cout << "Out of Host Memory" << endl;
            break;
        default:
            cout << "Unknown Error." << endl;
        }
    }

    outputImage = clCreateBuffer(context,CL_MEM_WRITE_ONLY,sizeof(cl_char) * width *
height,NULL,&err);
    if(err != CL_SUCCESS)
        printf("[-] Error : CreateBuffer\n");
    //////////////////////

    if(
        clSetKernelArg(kernel,0,sizeof(cl_mem),&inputImage) ||
        clSetKernelArg(kernel,1,sizeof(cl_mem),&outputImage) ||
        clSetKernelArg(kernel,2,sizeof(cl_int),&width)   ||
        clSetKernelArg(kernel,3,sizeof(cl_int),&pitch)
        )
        printf("[-] Error : SetKernelArg\n");

    global[0] = width - 1;
    global[1] = height - 1;
```

```
       if(clEnqueueNDRangeKernel(command_queue,kernel,2,NULL,global,0,0,NULL,NULL) !=
CL_SUCCESS)
              printf("[-] Error : EnqueueNDRangeKernel\n");

       clFinish(command_queue);

       if(clEnqueueReadBuffer(command_queue,outputImage,CL_FALSE,0,sizeof(cl_char)*width*
height,out_picture,0,NULL,NULL) != CL_SUCCESS)
              printf("[-] Error : EnqueueReadBuffer\n");

       // Elapsed time calculation.
       cout << "Elapsed Time = " << (clock()-t)/CLOCKS_PER_SEC << "s" << endl;

       //////////////////////////
       FIBITMAP *dst =
FreeImage_ConvertFromRawBits(out_picture,width,height,width,8,FI_RGBA_RED_MASK,FI_RGBA_GR
EEN_MASK,FI_RGBA_BLUE_MASK,TRUE);
       string outputName = fileName + "_output.";
       outputName.append(FreeImage_GetFormatFromFIF(fif));
       FreeImage_Save(fif,dst,outputName.c_str(),0);

       FreeImage_Unload(bitmap);
       clReleaseMemObject(inputImage);
       clReleaseMemObject(outputImage);
       clReleaseKernel(kernel);
       clReleaseProgram(program);
       clReleaseCommandQueue(command_queue);
       clReleaseContext(context);

       return EXIT_SUCCESS;
}
```

OpenCL – Pinned Texture Memory + Vector Data Type – Kernel

```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP_TO_EDGE |
CLK_FILTER_NEAREST;
__kernel void sobel(__read_only image2d_t image,
       __global unsigned char *G,
       const unsigned int width,
       const unsigned int pitch
       ){
              const int2 pos = {get_global_id(0) + 1, get_global_id(1) + 1};
              const int2 ur = {(pos.x+1) , (pos.y-1)};
              const int2 dr = {(pos.x+1) , (pos.y+1)};
              const int2 ul = {(pos.x-1) , (pos.y-1)};
              const int2 dl = {(pos.x-1) , (pos.y+1)};
              int Gx = read_imageui(image,sampler,ur).x +
2*read_imageui(image,sampler,(int2)((pos.x+1) , (pos.y))).x +
read_imageui(image,sampler,dr).x - read_imageui(image,sampler,ul).x -
2*read_imageui(image,sampler,(int2)((pos.x-1) , (pos.y))).x -
read_imageui(image,sampler,dl).x;
              int Gy = read_imageui(image,sampler,dl).x +
2*read_imageui(image,sampler,(int2)((pos.x) , (pos.y+1))).x +
read_imageui(image,sampler,dr).x - read_imageui(image,sampler,ul).x -
2*read_imageui(image,sampler,(int2)((pos.x) , (pos.y-1))).x -
read_imageui(image,sampler,ur).x;
```

```
                G[pos.x + pos.y*width] = sqrt((float)Gx*Gx + Gy*Gy);
}
```

OpenCL – Zero-Copy Texture Memory + Vector Type

```
#include <iostream>
#include <string>
#include <ctime>
#include <CL\cl.h>
#include <FreeImage.h>

using namespace std;

int main(){
        cl_platform_id platform_id;
        cl_uint num_platforms;
        cl_device_id device_id;
        cl_uint num_devices;
        cl_context context;
        cl_context_properties context_properties[3];
        cl_command_queue command_queue;
        cl_program program;
        cl_kernel kernel;
        cl_mem inputImage,outputImage;
        cl_int err;

        size_t global[2];

        char *kernelSource;

        FILE *pFile;
        int fileLen,readLen;

        char buffer[1024];
        char buildBuffer[2048];
        size_t buildLogLen;

        FREE_IMAGE_FORMAT fif;
        FIBITMAP *bitmap;
        unsigned int height,width,pitch;
        BYTE *picture;
        BYTE *out_picture;


        // The program always select the first platform and its first device.
        if(clGetPlatformIDs(1,&platform_id,&num_platforms) != CL_SUCCESS)
                printf("[-] Error : GetPlatformIDs\n");
        if(clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_GPU,1,&device_id,&num_devices) !=
CL_SUCCESS)
                printf("[-] Error : GetDeviceIDs\n");
        clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(buffer), buffer, NULL);
        cout << buffer << endl;
        context_properties[0] = CL_CONTEXT_PLATFORM;
        context_properties[1] = (cl_context_properties) platform_id;
        context_properties[2] = 0;
        context = clCreateContext(context_properties,1,&device_id,NULL,NULL,&err);
```

```
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateContext\n");

       command_queue = clCreateCommandQueue(context,device_id,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateCommandQueue\n");

       pFile = fopen("sobel_kernel.cl","r");
       fseek(pFile,0L,SEEK_END);
       fileLen = ftell(pFile);
       rewind(pFile);
       kernelSource = (char *) malloc(sizeof(char) * fileLen);
       readLen = fread(kernelSource,1,fileLen,pFile);
       kernelSource[readLen] = '\0';
       fclose(pFile);

       program = clCreateProgramWithSource(context,1,(const char
**)&kernelSource,NULL,&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateProgramWithSource\n");
       if(clBuildProgram(program,0,NULL,NULL,NULL,NULL) != CL_SUCCESS){
              printf("[-] Error : BuildProgram\n");

       clGetProgramBuildInfo(program,device_id,CL_PROGRAM_BUILD_LOG,sizeof(buildBuffer),b
uildBuffer,&buildLogLen);
              printf("%s\n",buildBuffer);
       }

       free(kernelSource);

       kernel = clCreateKernel(program,"sobel",&err);
       if(err != CL_SUCCESS)
              printf("[-] Error : CreateKernel\n");

       //////////////////////
       string fileName;
       cin >> fileName;
       fif = FreeImage_GetFileType(fileName.c_str());
       bitmap = FreeImage_Load(fif,fileName.c_str());
       pitch = FreeImage_GetPitch(bitmap);
       height = FreeImage_GetHeight(bitmap);
       width = FreeImage_GetWidth(bitmap);
       out_picture = (BYTE*) malloc(sizeof(BYTE)*height*width);
       bitmap = FreeImage_ConvertToGreyscale(bitmap);
       //////////////////////
       cl_image_format image_format = {CL_R,CL_UNSIGNED_INT8};

       double t = clock();

       inputImage = clCreateImage2D(context,CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR
,&image_format,pitch,height,0,NULL,&err);
       if(err != CL_SUCCESS){
              printf("[-] Error : CreateImage2D\n");
              switch(err){
              case CL_INVALID_CONTEXT:
                     cout << "Invalid Context" << endl;
                     break;
              case CL_INVALID_VALUE :
```

```
                        cout << "Invalid Value" << endl;
                        break;
                case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
                        cout << "Invalid Image Format Description" << endl;
                        break;
                case CL_INVALID_IMAGE_SIZE:
                        cout << "Invalid Image size" << endl;
                        cout << "Max Dims = " << CL_DEVICE_IMAGE2D_MAX_WIDTH << " * " <<
CL_DEVICE_IMAGE2D_MAX_HEIGHT << endl;
                        cout << "Your Dims = " << pitch << " * " << height << endl;
                        break;
                case CL_INVALID_HOST_PTR:
                        cout << "Invalid Host Ptr" << endl;
                        break;
                case CL_IMAGE_FORMAT_NOT_SUPPORTED:
                        cout << "Image Format Not Supported" << endl;
                        break;
                case CL_MEM_OBJECT_ALLOCATION_FAILURE:
                        cout << "Object Allocation Failure" << endl;
                        break;
                case CL_INVALID_OPERATION:
                        cout << "Invalid Operation" << endl;
                        break;
                case CL_OUT_OF_HOST_MEMORY:
                        cout << "Out of Host Memory" << endl;
                        break;
                default:
                        cout << "Unknown Error." << endl;
                }
        }

        size_t origin[] = {0,0,0};
        size_t region[] = {pitch,height,1};

        picture = (BYTE *)
clEnqueueMapImage(command_queue,inputImage,CL_TRUE,CL_MAP_READ,origin,region,0,0,NULL,NUL
L,NULL,&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : EnqueueMapImage\n");
        FreeImage_ConvertToRawBits(picture,bitmap,pitch,8,FI_RGBA_RED_MASK,FI_RGBA_GREEN_M
ASK,FI_RGBA_BLUE_MASK,TRUE);

        if((err =
clEnqueueWriteImage(command_queue,inputImage,CL_FALSE,origin,region,0,0,(void
*)picture,0,NULL,NULL)) != CL_SUCCESS){
                printf("[-] Error : EnqueueWriteImage\n");
                switch(err){
                case CL_INVALID_COMMAND_QUEUE:
                        cout << "CL_INVALID_COMMAND_QUEUE" << endl;
                        break;
                case CL_INVALID_CONTEXT:
                        cout << "CL_INVALID_CONTEXT" << endl;
                        break;
                case CL_INVALID_MEM_OBJECT:
                        cout << "CL_INVALID_MEM_OBJECT" << endl;
                        break;
                case CL_INVALID_VALUE:
                        cout << "CL_INVALID_VALUE" << endl;
                        break;
                case CL_INVALID_EVENT_WAIT_LIST :
                        cout << "CL_INVALID_EVENT_WAIT_LIST" << endl;
                        break;
                case CL_MEM_OBJECT_ALLOCATION_FAILURE:
                        cout << "CL_MEM_OBJECT_ALLOCATION_FAILURE " << endl;
                        break;
                case CL_OUT_OF_HOST_MEMORY:
                        cout << "CL_OUT_OF_HOST_MEMORY" << endl;
                        break;
                default:
                        cout << "Unknown Error." << endl;
                }
        }

        outputImage = clCreateBuffer(context,CL_MEM_WRITE_ONLY |
CL_MEM_ALLOC_HOST_PTR,sizeof(cl_char) * width * height,NULL,&err);
        if(err != CL_SUCCESS)
                printf("[-] Error : CreateBuffer\n");
        //////////////////////

        if(
                clSetKernelArg(kernel,0,sizeof(cl_mem),&inputImage) ||
                clSetKernelArg(kernel,1,sizeof(cl_mem),&outputImage) ||
                clSetKernelArg(kernel,2,sizeof(cl_int),&width)   ||
                clSetKernelArg(kernel,3,sizeof(cl_int),&pitch)
                )
                printf("[-] Error : SetKernelArg\n");

        global[0] = width - 1;
        global[1] = height - 1;
        if(clEnqueueNDRangeKernel(command_queue,kernel,2,NULL,global,0,0,NULL,NULL) !=
CL_SUCCESS)
                printf("[-] Error : EnqueueNDRangeKernel\n");

        clFinish(command_queue);

        if(clEnqueueReadBuffer(command_queue,outputImage,CL_FALSE,0,sizeof(cl_char)*width*
height,out_picture,0,NULL,NULL) != CL_SUCCESS)
                printf("[-] Error : EnqueueReadBuffer\n");

        // Elapsed time calculation.
        cout << "Elapsed Time = " << (clock()-t)/CLOCKS_PER_SEC << "s" << endl;

        //////////////////////////
        FIBITMAP *dst =
FreeImage_ConvertFromRawBits(out_picture,width,height,width,8,FI_RGBA_RED_MASK,FI_RGBA_GR
EEN_MASK,FI_RGBA_BLUE_MASK,TRUE);
        string outputName = fileName + "_output.";
        outputName.append(FreeImage_GetFormatFromFIF(fif));
        FreeImage_Save(fif,dst,outputName.c_str(),0);

        FreeImage_Unload(bitmap);
        clReleaseMemObject(inputImage);
        clReleaseMemObject(outputImage);
        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(command_queue);
```

```
	clReleaseContext(context);

	return EXIT_SUCCESS;
}
```

CUDA Naïve Implementation

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <cuda.h>
#include <iostream>
#include <cmath>
#include <ctime>
#include <Windows.h>
#include <FreeImage.h>
#include <string>

using namespace std;

__global__ void sobel(BYTE *image,BYTE *G, unsigned int height, unsigned int width,
unsigned int pitch){
	int i = blockDim.x*blockIdx.x+threadIdx.x+1;
	int j = blockDim.y*blockIdx.y+threadIdx.y+1;
	int Gx = image[(i+1) + (j-1)*pitch] + 2*image[(i+1) + (j)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i-1) + (j)*pitch] - image[(i-1) +
(j+1)*pitch];
	int Gy = image[(i-1) + (j+1)*pitch] + 2*image[(i) + (j+1)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i) + (j-1)*pitch] - image[(i+1) +
(j-1)*pitch];
	G[i + j*width] = sqrtf(powf(Gx,2) + powf(Gy,2));
}


int main(){
	string fileName;
	clock_t t;
	FREE_IMAGE_FORMAT fif;
	FIBITMAP *bitmap;

	BYTE * in_picture,* out_picture;

	cin >> fileName;
	fif = FreeImage_GetFileType(fileName.c_str());
	bitmap = FreeImage_Load(fif,fileName.c_str());
	unsigned int pitch = FreeImage_GetPitch(bitmap);
	unsigned int height = FreeImage_GetHeight(bitmap);
	unsigned int width = FreeImage_GetWidth(bitmap);
	bitmap = FreeImage_ConvertToGreyscale(bitmap);
	in_picture =  (BYTE *) malloc(sizeof(BYTE)*pitch*height);
	out_picture =  (BYTE *) calloc(sizeof(BYTE),width*height);

	FreeImage_ConvertToRawBits(in_picture,bitmap,pitch,8,0,0,0,TRUE);

	/********** GPU Part ***********/
	BYTE *d_in,*d_out;
```

```
        t = clock();

        /**************** Memory Allocation ****************/
        cudaMalloc(&d_in,sizeof(BYTE)*pitch*height);
        cudaMemcpy(d_in,in_picture,sizeof(BYTE)*pitch*height,cudaMemcpyHostToDevice);
        cudaMalloc(&d_out,sizeof(BYTE)*width*height);

        /****************** Kernel Launch Configuration *************/
        dim3 gridDim(width/32,height/32);
        dim3 blockDim(32,32);

        /********* Kernel Launch *************/
        sobel<<<gridDim,blockDim>>>(d_in,d_out,height,width,pitch);
        cudaThreadSynchronize();

        /********** Taking the result back from GPU **************/
        cudaMemcpy(out_picture,d_out,sizeof(BYTE)*width*height,cudaMemcpyDeviceToHost);

        // Elapsed time calculation.
        cout << "Elapsed Time = " << ((float)clock()-t)/CLOCKS_PER_SEC << "s" << endl;

        /*********************************/

        /******** Producing the output *********/
        FIBITMAP *dst = 
FreeImage_ConvertFromRawBits(out_picture,width,height,width,8,0,0,0,TRUE);
        string outputName = fileName + "_output.";
        outputName.append(FreeImage_GetFormatFromFIF(fif));
        FreeImage_Save(fif,dst,outputName.c_str(),0);

        // Free memory allocation.
        free(in_picture);
        free(out_picture);
        FreeImage_Unload(bitmap);
}
```

CUDA Pinned Memory

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <cuda.h>
#include <iostream>
#include <cmath>
#include <ctime>
#include <Windows.h>
#include <FreeImage.h>
#include <string>

using namespace std;

__global__ void sobel(BYTE *image,BYTE *G, unsigned int height, unsigned int width, 
unsigned int pitch){
        int i = blockDim.x*blockIdx.x+threadIdx.x+1;
        int j = blockDim.y*blockIdx.y+threadIdx.y+1;
```

```
	int Gx = image[(i+1) + (j-1)*pitch] + 2*image[(i+1) + (j)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i-1) + (j)*pitch] - image[(i-1) +
(j+1)*pitch];
	int Gy = image[(i-1) + (j+1)*pitch] + 2*image[(i) + (j+1)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i) + (j-1)*pitch] - image[(i+1) +
(j-1)*pitch];
	G[i + j*width] = sqrtf(powf(Gx,2) + powf(Gy,2));
}


int main(){
	string fileName;
	clock_t t;
	FREE_IMAGE_FORMAT fif;
	FIBITMAP *bitmap;

	BYTE * in_picture,* out_picture;

	cin >> fileName;
	fif = FreeImage_GetFileType(fileName.c_str());
	bitmap = FreeImage_Load(fif,fileName.c_str());
	unsigned int pitch = FreeImage_GetPitch(bitmap);
	unsigned int height = FreeImage_GetHeight(bitmap);
	unsigned int width = FreeImage_GetWidth(bitmap);
	bitmap = FreeImage_ConvertToGreyscale(bitmap);
	cudaMallocHost(&in_picture,sizeof(BYTE)*pitch*height);
	cudaMallocHost(&out_picture,sizeof(BYTE)*width*height);
	FreeImage_ConvertToRawBits(in_picture,bitmap,pitch,8,0,0,0,TRUE);

	/********** GPU Part ***********/
	BYTE *d_in,*d_out;
	cudaMalloc(&d_in,sizeof(BYTE)*pitch*height);
	cudaMalloc(&d_out,sizeof(BYTE)*width*height);

	t = clock();

	/**************** Memory Allocation ****************/
	cudaMemcpy(d_in,in_picture,sizeof(BYTE)*pitch*height,cudaMemcpyHostToDevice);

	/****************** Kernel Launch Configuration *************/
	dim3 gridDim(width/32,height/32);
	dim3 blockDim(32,32);

	/********* Kernel Launch *************/
	sobel<<<gridDim,blockDim>>>(d_in,d_out,height,width,pitch);
	cudaThreadSynchronize();

	/********** Taking the result back from GPU **************/
	cudaMemcpy(out_picture,d_out,sizeof(BYTE)*width*height,cudaMemcpyDeviceToHost);

	// Elapsed time calculation.
	cout << "Elapsed Time = " << ((float)clock()-t)/CLOCKS_PER_SEC << "s" << endl;

	/*********************************/

	/******** Producing the output *********/
	FIBITMAP *dst =
FreeImage_ConvertFromRawBits(out_picture,width,height,width,8,0,0,0,TRUE);
```

```
	string outputName = fileName + "_output.";
	outputName.append(FreeImage_GetFormatFromFIF(fif));
	FreeImage_Save(fif,dst,outputName.c_str(),0);

	// Free memory allocation.
	cudaFreeHost(in_picture);
	cudaFreeHost(out_picture);
	cudaFree(d_in);
	cudaFree(d_out);
	FreeImage_Unload(bitmap);
}
```

CUDA Zero-Copy Memory

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <cuda.h>
#include <iostream>
#include <cmath>
#include <ctime>
#include <Windows.h>
#include <FreeImage.h>
#include <string>

using namespace std;

__global__ void sobel(BYTE *image,BYTE *G, unsigned int height, unsigned int width,
unsigned int pitch){
	int i = blockDim.x*blockIdx.x+threadIdx.x+1;
	int j = blockDim.y*blockIdx.y+threadIdx.y+1;
	int Gx = image[(i+1) + (j-1)*pitch] + 2*image[(i+1) + (j)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i-1) + (j)*pitch] - image[(i-1) +
(j+1)*pitch];
	int Gy = image[(i-1) + (j+1)*pitch] + 2*image[(i) + (j+1)*pitch] + image[(i+1) +
(j+1)*pitch] - image[(i-1) + (j-1)*pitch] - 2*image[(i) + (j-1)*pitch] - image[(i+1) +
(j-1)*pitch];
	G[i + j*width] = sqrtf(powf(Gx,2) + powf(Gy,2));
}


int main(){
	string fileName;
	clock_t t;
	FREE_IMAGE_FORMAT fif;
	FIBITMAP *bitmap;

	BYTE * in_picture,* out_picture;
	BYTE *d_in,*d_out;

	cin >> fileName;
	fif = FreeImage_GetFileType(fileName.c_str());
	bitmap = FreeImage_Load(fif,fileName.c_str());
	unsigned int pitch = FreeImage_GetPitch(bitmap);
	unsigned int height = FreeImage_GetHeight(bitmap);
	unsigned int width = FreeImage_GetWidth(bitmap);
	bitmap = FreeImage_ConvertToGreyscale(bitmap);
	cudaHostAlloc(&in_picture,sizeof(BYTE)*pitch*height,cudaHostAllocMapped);
	cudaHostAlloc(&out_picture,sizeof(BYTE)*width*height,cudaHostAllocMapped |
cudaHostAllocWriteCombined);

	FreeImage_ConvertToRawBits(in_picture,bitmap,pitch,8,0,0,0,TRUE);

	/********** GPU Part ***********/

	t = clock();

	/**************** Memory Allocation ****************/
	cudaHostGetDevicePointer(&d_in,in_picture,0);
	cudaHostGetDevicePointer(&d_out,out_picture,0);

	/****************** Kernel Launch Configuration *************/
	dim3 gridDim(width/32,height/32);
	dim3 blockDim(32,32);

	/********* Kernel Launch *************/
	sobel<<<gridDim,blockDim>>>(d_in,d_out,height,width,pitch);
	cudaThreadSynchronize();

	/********** Taking the result back from GPU **************/

	// Elapsed time calculation.
	cout << "Elapsed Time = " << ((float)clock()-t)/CLOCKS_PER_SEC << "s" << endl;

	/*********************************/

	/******** Producing the output *********/
	FIBITMAP *dst =
FreeImage_ConvertFromRawBits(out_picture,width,height,width,8,0,0,0,TRUE);
	string outputName = fileName + "_output.";
	outputName.append(FreeImage_GetFormatFromFIF(fif));
	FreeImage_Save(fif,dst,outputName.c_str(),0);

	// Free memory allocation.
	cudaFree(in_picture);
	cudaFree(d_in);
	cudaFree(d_out);
	cudaFree(out_picture);
	FreeImage_Unload(bitmap);
}
```

CUDA Zero-Copy Memory + Modified Kernel

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <cuda.h>
#include <iostream>
#include <cmath>
#include <ctime>
#include <Windows.h>
#include <FreeImage.h>
#include <string>
```

```
using namespace std;

__global__ void sobel(BYTE *image,BYTE *G, unsigned int width, unsigned int pitch){
       int i = blockDim.x*blockIdx.x+threadIdx.x+1;
       int j = blockDim.y*blockIdx.y+threadIdx.y+1;
       int ur = (i+1) + (j-1)*pitch;
       int dr = (i+1) + (j+1)*pitch;
       int ul = (i-1) + (j-1)*pitch;
       int dl = (i-1) + (j+1)*pitch;
       int Gx = image[ur] + 2*image[(i+1) + (j)*pitch] + image[dr] - image[ul] - 
2*image[(i-1) + (j)*pitch] - image[dl];
       int Gy = image[dl] + 2*image[(i) + (j+1)*pitch] + image[dr] - image[ul] - 
2*image[(i) + (j-1)*pitch] - image[ur];
       G[i + j*width] = sqrtf(Gx*Gx + Gy*Gy);
}


int main(){
       string fileName;
       clock_t t;
       FREE_IMAGE_FORMAT fif;
       FIBITMAP *bitmap;

       BYTE * in_picture,* out_picture;
       BYTE *d_in,*d_out;

       cin >> fileName;
       fif = FreeImage_GetFileType(fileName.c_str());
       bitmap = FreeImage_Load(fif,fileName.c_str());
       unsigned int pitch = FreeImage_GetPitch(bitmap);
       unsigned int height = FreeImage_GetHeight(bitmap);
       unsigned int width = FreeImage_GetWidth(bitmap);
       bitmap = FreeImage_ConvertToGreyscale(bitmap);
       cudaHostAlloc(&in_picture,sizeof(BYTE)*pitch*height,cudaHostAllocMapped);
       cudaHostAlloc(&out_picture,sizeof(BYTE)*width*height,cudaHostAllocMapped | 
cudaHostAllocWriteCombined);

       FreeImage_ConvertToRawBits(in_picture,bitmap,pitch,8,0,0,0,TRUE);

       /********** GPU Part ***********/

       t = clock();

       /**************** Memory Allocation ****************/
       cudaHostGetDevicePointer(&d_in,in_picture,0);
       cudaHostGetDevicePointer(&d_out,out_picture,0);

       /****************** Kernel Launch Configuration *************/
       dim3 gridDim(width/32,height/32);
       dim3 blockDim(32,32);

       /********* Kernel Launch *************/
       sobel<<<gridDim,blockDim>>>(d_in,d_out,width,pitch);
       cudaThreadSynchronize();

       /********** Taking the result back from GPU **************/

       // Elapsed time calculation.
       cout << "Elapsed Time = " << ((float)clock()-t)/CLOCKS_PER_SEC << "s" << endl;

       /*********************************/

       /******** Producing the output *********/
       FIBITMAP *dst =
FreeImage_ConvertFromRawBits(out_picture,width,height,width,8,0,0,0,TRUE);
       string outputName = fileName + "_output.";
       outputName.append(FreeImage_GetFormatFromFIF(fif));
       FreeImage_Save(fif,dst,outputName.c_str(),0);

       // Free memory allocation.
       cudaFree(in_picture);
       cudaFree(d_in);
       cudaFree(d_out);
       cudaFree(out_picture);
       FreeImage_Unload(bitmap);
}
```

OpenMP Naïve implementation

```
#include <iostream>
#include <cmath>
#include <ctime>
#include <Windows.h>
#include <FreeImage.h>
#include <string>
#include <omp.h>

using namespace std;

unsigned int height,width,pitch;

void sobel(BYTE *image,BYTE *G){
       int thread_id = 0;
       int num_threads = omp_get_max_threads();
       omp_set_num_threads(num_threads);
       int block_size = (height - 1)/num_threads;
#pragma omp parallel for default(thread) private(thread_id)
       for(thread_id = 0; thread_id < num_threads; ++thread_id){
              for(int i = 1 ; i < width; i++){
                     int limit = thread_id * block_size + block_size;
                     for(int j = 1 + thread_id * block_size; j < limit ; j++){
                            int ur = (i+1) + (j-1)*pitch;
                            int dr = (i+1) + (j+1)*pitch;
                            int ul = (i-1) + (j-1)*pitch;
                            int dl = (i-1) + (j+1)*pitch;
                            int Gx = image[ur] + 2*image[(i+1) + (j)*pitch] + image[dr] -
image[ul] - 2*image[(i-1) + (j)*pitch] - image[dl];
                            int Gy = image[dl] + 2*image[(i) + (j+1)*pitch] + image[dr] -
image[ul] - 2*image[(i) + (j-1)*pitch] - image[ur];

                            G[i + j*width] = sqrtf(Gx*Gx + Gy*Gx);
                     }
              }
       }
}


int main(){
       string fileName;
       clock_t t;
       FREE_IMAGE_FORMAT fif;
       FIBITMAP *bitmap;

       BYTE * in_picture,* out_picture;


       cin >> fileName;
       fif = FreeImage_GetFileType(fileName.c_str());
       bitmap = FreeImage_Load(fif,fileName.c_str());
       pitch = FreeImage_GetPitch(bitmap);
       height = FreeImage_GetHeight(bitmap);
       width = FreeImage_GetWidth(bitmap);
       bitmap = FreeImage_ConvertToGreyscale(bitmap);
       in_picture =  (BYTE *) malloc(sizeof(BYTE)*pitch*height);
       out_picture =  (BYTE *) calloc(sizeof(BYTE),width*height);

       FreeImage_ConvertToRawBits(in_picture,bitmap,pitch,8,0,0,0,TRUE);

       /**************/
       t = clock();

       sobel(in_picture,out_picture);

       cout << "Elapsed Time = " << ((float)clock()-t)/CLOCKS_PER_SEC << "s" << endl;
       /**************/

       FIBITMAP *dst =
FreeImage_ConvertFromRawBits(out_picture,width,height,width,8,0,0,0,TRUE);
       string outputName = fileName + "_output.";
       outputName.append(FreeImage_GetFormatFromFIF(fif));
       FreeImage_Save(fif,dst,outputName.c_str(),0);

       // Free memory allocation.
       free(in_picture);
       free(out_picture);
       FreeImage_Unload(bitmap);

       return EXIT_SUCCESS;
}
```

References

nVidia’s OpenCL Programming Guide for the CUDA Architecture