Parallel Implementation of Computer Vision Algorithms

Sobel Transform, Hough Transform, Pyramidal Transform

Report for

Independent study (ECEN 5840-909) Under Prof. Sam Siewert

By

Surjith Bhagavath Singh

Team Information

Dr. Sam Siewert

Ramnarayan Krishnamurthy

Ryan Claus

Matthew Vis

Table of Contents

Introduction

The primary focus of this independent study is to parallelize the basic image transforms using OpenCL in Altera DE1 SoC and compare the results with CUDA implementation in Jetson TK1.

Analysis includes power profiling, how fast/slow is DE1-SoC compared to Jetson TK1. Main objective is to make the code power efficient to do the image transform. FPGA Architecture in general is fast at doing things parallel because of the implementation in low level logic gates. But for this case, having a parallel platform such as OpenCL and converting the C code into Verilog/VHDL and then to low level logic elements could make some difference. How different that is from doing things parallel using GPUs.

This report explains the three basic image transformation algorithms and their implementation in OpenCL. This report also covers how to set up the OpenCL environment in DE1-SoC.

The ultimate aim is to contribute some progress for Software Defined Photometry Research being done by Prof.Sam Siewert.

What is different from initial proposal?

Initially proposed to render a 2D image in 3D Space captured by normal camera using FPGA Hardware Acceleration (DE1-SoC). The main idea is to use hardware acceleration for these kind of data intensive fusion algorithms. But later I came to a conclusion to continue the work done by Chris Wagner on OpenCL (Hardware Acceleration) for basic transforms. I have completed Sobel, Hough and Pyramidal fusion algorithms in OpenCL (Parallel Version).

Hardware – Overview

The DE1-SoC Development Kit has a robust hardware design platform built around the Altera System-on-Chip (SoC) FPGA, which combines the latest dual-core Cortex-A9 embedded cores with industry-leading programmable logic (Cyclone V) for ultimate design flexibility. Altera's software support is in the primitive stage but it is the industry's leading software for this platform. Altera's SoC integrates ARM Processor with the FPGA Fabric using the high speed AMBA interconnect Bus. The DE1-SoC development board includes hardware such as high-speed DDR3 memory, video and audio capabilities, Ethernet networking, USB Capabilities.[1]

[1] http://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=836&FID=eac30a7aaacf5187a4ace0d613cd4676

The DE1-SOC Development Kit contains all components needed to use the board in conjunction with a computer that runs the Microsoft Windows XP or later (64-bit OS and Quartus II 64-bit are required to compile projects for DE1-SoC).

Disadvantage of the board is that it has lot of other fancy stuff (7-Segment displays and other peripherals which is not needed for our analysis). I cannot be able to turn them off when it is in HPS (Hard processor System) mode. Future work includes figuring out a way to disable the unused peripherals.

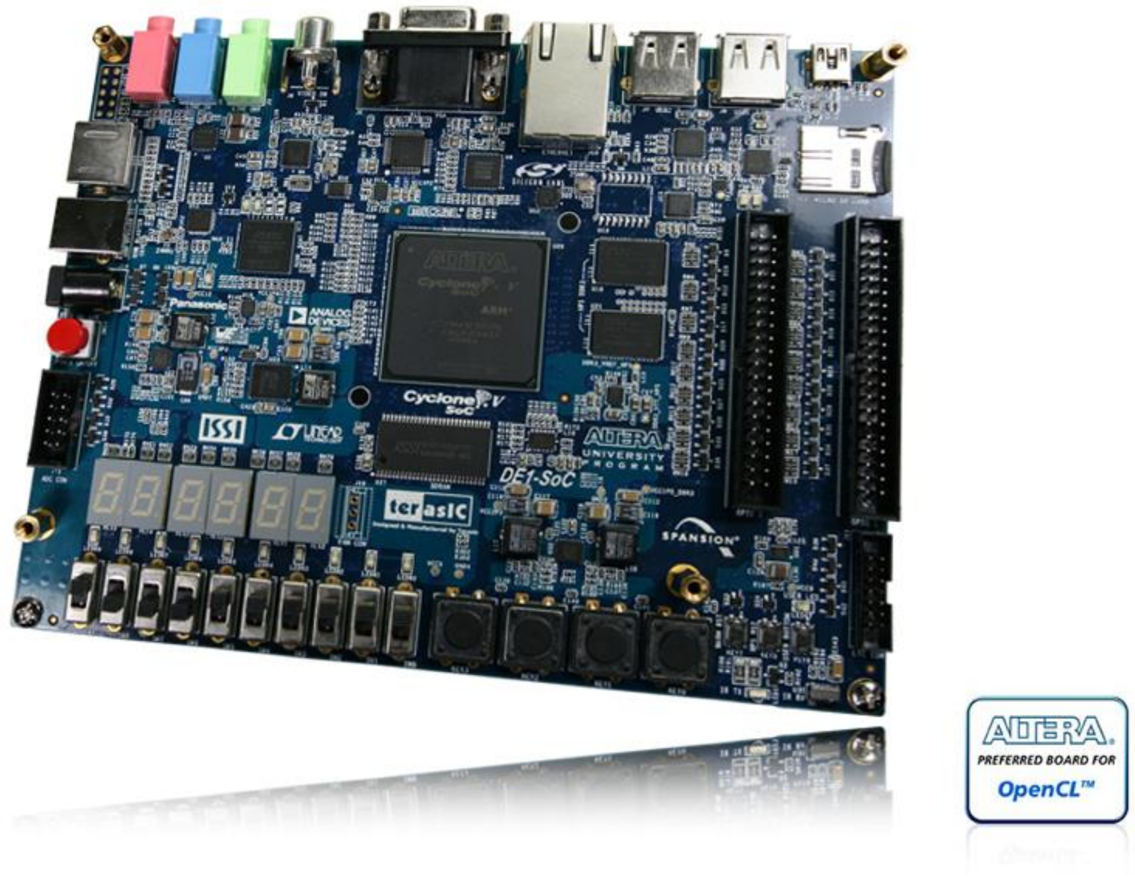


Figure 1: DE1-SoC

Parallel Computing

The Single core CPUs were replaced by multicore CPUs for performance improvement over a time. In order to do a massive, data intensive tasks like 3D image rendering using a single or multi core CPUs were a myth once upon a time. That was the time GPUs played an important role by parallelizing the process using more cores. Later it became a platform to do massive data intensive operations (General Purpose Graphics Processing Unit) Which is suitable for computer/machine vision applications. Now there is a potential in FPGA and DSP domains to do the parallel tasks at gate level logics which could accelerate the process.

OpenCL Overview

Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL specifies a programming language (based on C99) for programming these devices and application programming interfaces (APIs) to control the platform and execute programs on the compute devices. OpenCL provides a standard interface for parallel computing using task-based and data-based parallelism.[2]

OpenCL is fueled by open source community formed by big companies in FPGA, GPU, DSP domain. But the support for FPGA platform is in primitive stage. Altera has a best practices guide[3], which is helpful in writing an optimized code. In general, for OpenCL there is lot of support compared to FPGA Oriented support. CUDA is dominating the market because of its simplicity and efficient libraries. There is still lot of development going on in this parallel platform.

GPU vs FPGA

FPGAs Implement the algorithm in hardware, in general hardware is much faster than software. But code development time for FPGA was a myth before generalizing the platforms like OpenCL. But GPUs are fast at doing floating point operations and the software work much closer to the hardware helps it fast enough in par with FPGA. NVidia Kepler GPU in Jetson TK1 SoC can run 192 floating point operations in a single clock cycle. DE1 SoC have 384 DSP blocks which is capable of doing floating point operations, The idle power taken by DE1-SoC is much larger compared to Jetson TK1. The software setup for Jetson is much simpler and easy to use compared to Altera's software.

[2] https://en.wikipedia.org/wiki/OpenCL
[3] https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf

Sobel Transform

Algorithm

Sobel is a differential operator helps in edge detection. Differential matrix convoluted through the entire image gives the gradient. Differential Matrix (G_x) convolved with the original image gives the gradient of the pixel in x direction. Differential Matrix (G_y)Convolved with original image gives the gradient of the pixel in y direction

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

This gradient component in x direction and y direction gives the directional component. In order to get the actual gradient at the pixel, absolute value of both G_x and $G_y$has to be calculated.

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$

In order to make it optimized, we can take the individual absolute values and add them separately. It is not accurate but it will give a reasonable value. This level of accuracy is enough for the chosen application.

G = abs(G_x) + abs(G_y);

Parallel version

The algorithm shows that there is no data dependency between neighbor pixels and it can be easily parallelized.

Resolution of the image: $\boldsymbol{width * height}$

Number of pixel: $\boldsymbol{width * height}$

Number of convolutions: $\boldsymbol{2 * width * height}$ **[both x and y directions]**

This algorithm can be parallelized width*height times; the following code snippet is the parallelized code for each pixel. Since the image that has been stores is an 8-bit image. So the output has been limited for 255.

```
#pragma unroll
for(i=0;i<3;i++)
{
      #pragma unroll
      for(j=0;j<3;j++)
      {
            G_y += (Gy[i][j])*frame_in[(x+j-1)+(width*(y+i-1))];
            G_x += (Gx[i][j])*frame_in[(x+j-1)+(width*(y+i-1))];
      }
}

G = abs(G_x) + abs(G_y);

if(G>255)
      frame_out[index] = 255;
else
      frame_out[index] = G;
```

Code Snippet 1: Sobel Kernel parallelized version

According to the concept of OpenCL, Kernel size is defined by the user and it has to generate the hardware according to the kernel size. But Altera OpenCL compiler optimizes the code and it generates one kernel of hardware and it pipelines the data through it.

Sobel Results

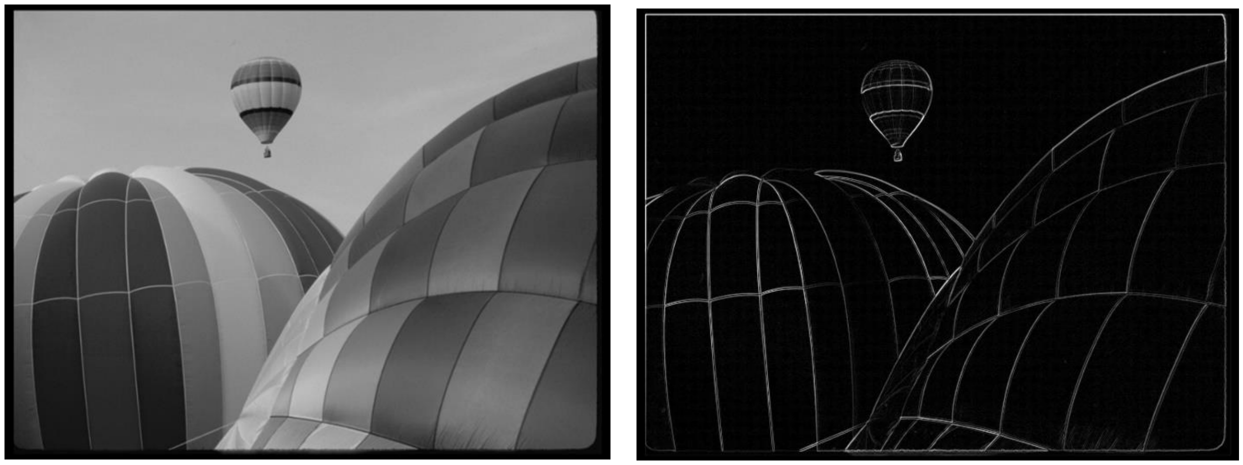


Figure 2:Sobel Transform
a.Input Image b.Output Image

Hough Transform

Algorithm

Hough Transform is a method used for feature extraction (Line detection, Circle detection or pattern recognition). It detects the pattern by converting the image from x and y plane to ro and theta plane. A line in x-y plane is a point in r-θ plane. The line has been detected by polling mechanism. Each line can be represented using a slope and an intercept. These two can be represented as a point in r-θ plane.

$$r = x\cos\theta + y\sin\theta$$

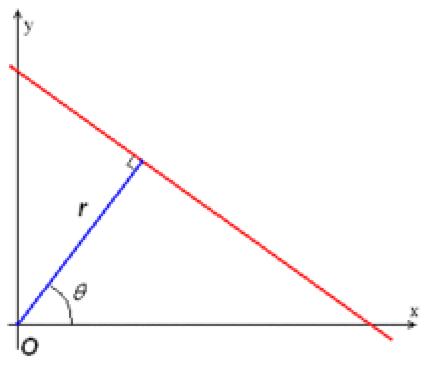


Figure 3: Line representation (XY - R_Theta)[4]

$$x = \frac{r - y\sin\theta}{\cos\theta}$$

$$y = \frac{r - x\cos\theta}{\sin\theta}$$

Hough can be executed after running an edge operator in the image which improves the accuracy. Edge operator gives a grayscale or binary image, for each pixel; above a certain threshold will be polling for its coordinates in r and theta plane. At the end of transformation, if there is a line in the image; it's r and theta point will have high polled number relative to other points in r and theta plane.[5]

Parallel Version

For each pixel it has to go through 180 iterations in order to poll all the possible theta of the particular pixel. Even in hough we don't have any data dependency, which is easy for parallelizing it.

Resolution of the image: $\boldsymbol{width * height}$

Number of pixel: $\boldsymbol{width * height}$

Number of iterations: $\boldsymbol{180 * width * height}$ **(Serial Sequence)**

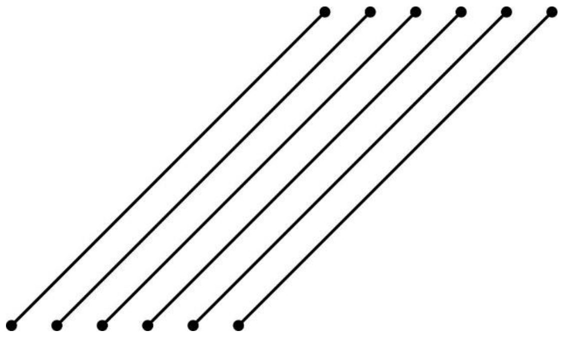
Here is the code snippet for each pixel and each pixel is parallelized. r value for each theta value ranging from 0 to 180 is computed and corresponding output pixel is incremented. This algorithm can be parallelized width*height times; the following code snippet is the parallelized code for each pixel.

[4] https://thecuriousastronomer.files.wordpress.com/2014/07/20140708-174639-63999679.jpg
[5] http://www.keymolen.com/2013/05/hough-transformation-c-implementation.html

```
if( frame_in[index] > 250 )
{
      for(int t=0;t<180;t++)
      {
            double r = ( ((double)x - center_x) * cos((double)t * DEG2RAD)) +
            (((double)y - center_y) * sin((double)t * DEG2RAD));

            frame_out[ (int)((round(r + hough_h) * 180.0)) + t]++;

      }

}
```

Code Snippet 2: Hough Parellel code

Hough Results



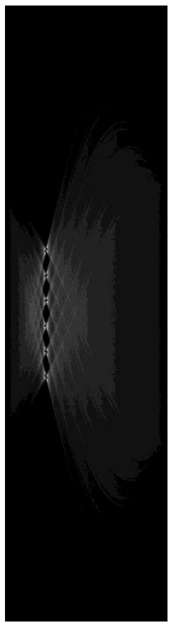


Figure 4: Hough Transform
a. Input Image b.Output Image

From the results it is visible that it has 6 bright spots in the image. In the input image there are 6 parallel lines which resulted the points in output image in a straight line (Slope is constant for all the images and intercept alone changes). The output image's width is constant because theta varies from 0 to 180. Height varies according to the resolution (diagonal) of the input image.

Pyramidal Transform

Algorithm

Pyramidal conversion is done by using two filters Laplacian and Gaussian. Gaussian kernel is used to scale it down and laplacian helps in scaling it up. It acts as a low pass filter and removes high frequency components in the image.

Pyramidal Up conversion

While doing pyramidal up conversion each pixel is taken and it is zero padded with filter applied on the output image. For each pixel in the original image, 4 pixels are created in the up transform.

Pyramidal Down Conversion

It subsamples the image and takes 1 pixel for every four pixels and it is the converse of Up conversion. After subsampling the image it applies the Gaussian filter.

Parallel Version

Down-sampling is easy to parallelize than Up-sampling. Down sampling is just taking a subset of the entire image. For Up-sampling it requires four times the number of threads to get the image. It can be better understood from the code and the code is self-explanatory.

Pyramidal Down Transform

Resolution of the image: $\boldsymbol{width * height}$

Number of pixel: $\boldsymbol{width * height}$

Number of iterations: **25***$\boldsymbol{width * height}$ **(Serial Sequence)**

25 iterations are the Gaussian matrix iteration for each pixel. This is for down Conversion.

Pyramidal Up Transform

Resolution of the image: $\boldsymbol{2 * width * 2 * height}$

Number of pixel: $\boldsymbol{2 * width * 2 * height}$

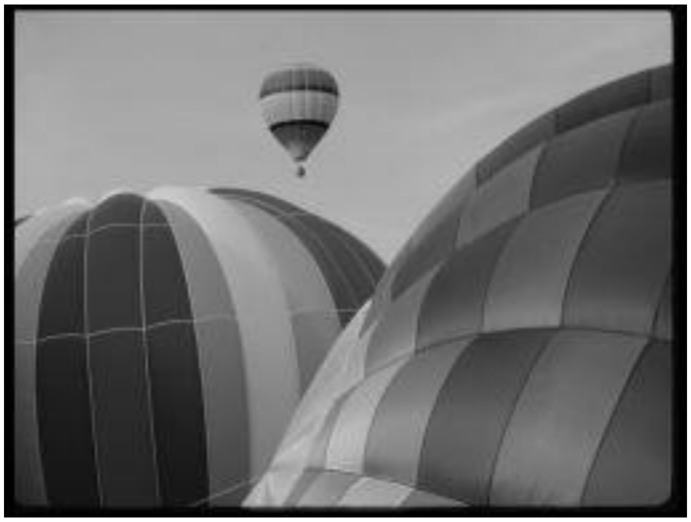
Number of iterations: $\boldsymbol{25 * 2 * width * 2 * height}$ **(Serial Sequence- Rough estimate)**

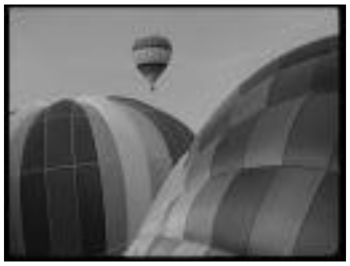
25 iterations are the Laplacian matrix iteration for each pixel. This is for Up Conversion.

For every pixel border condition has to be checked. And it checks for index with multiple of two to save the originality of the image. A pixel with multiple of two will not be altered and the Laplacian/Gaussian is applied for other pixels.

Since the up transform has four times the number of pixels than original image. It creates 4 times the threads to do the up conversion. FPGA Pipelines the kernel according to the hardware availability.

Pyramidal Results





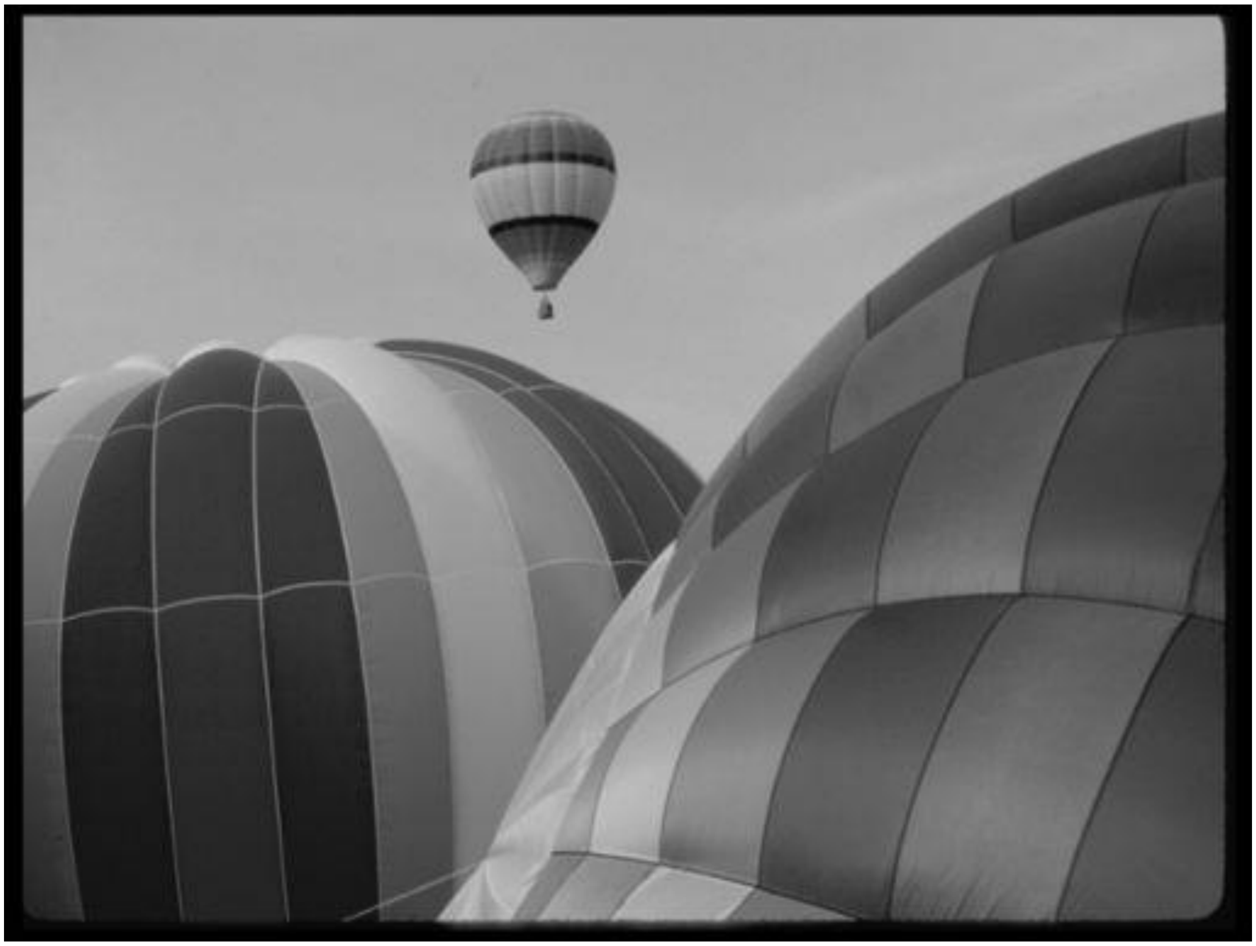


Figure 5: Pyramidal Output – All images are scaled down 10 times to fit inside the document
*a. Input Image(2560*1920) b.Pyramidal Down(1280*960) c. Pyramidal Up(5120*3840)*

Host Code (C++ and OpenCL)

The above mentioned codes are OpenCL Kernel codes. Host code is the code running in ARM Dual core at DE1 SoC. This code calls the OpenCL Kernel. This code acts a master code for the OpenCL Kernel. This calls the device code whenever it needs parallelization in FPGA Logic. Host code needs some parameters to call the OpenCL kernels.

- cl_device_id
- cl_platform_id
- cl_program
- cl_kernel
- cl_command_queue
- cl_context
- cl_mem

These parameters are best described in Altera OpenCL best practices guide. Refer khronos webpage for detailed functions. Number of threads can be mentioned when the kernel is being enqueued.

Steps involved in writing any host code for OpenCL

The following sequence has to be followed to initialize OpenCL from host code. Altera's example codes are taken as reference for this code development.

1. Use "findplatform" function with the parameter as the name of the platform. In this case "Altera" is the plat form. This function returns a value of parameter type cl_platform_id. Store the value in a variable of data type cl_platform_id.

   ```
   cl_platform_id platform = findplatform("Altera");
   ```

2. Use "clGetDeviceIDs" function with parameters as platform id, device type, address of the variable to store the device ID . This returns the error message status value in a variable of datatype cl_int.

   ```
   cl_int status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device,
   NULL);

   checkError (status, "Error: could not query devices");
   ```

3. Use status variable to check the error types. If it is non zero then it is an error message. Prompt out error using "checkError" function.

   ```
   checkError (status, "Error: Error Message");
   ```

4. Create a context using "clCreateContext" function and parameters as number of devices, device id, and address of status variable. This returns a datatype of cl_context. This return value has to

be used for command queuing.

```
context = clCreateContext(0, num_devices, &device, &oclContextCallback, NULL,
&status);
```

```
checkError(status, "Error: could not create OpenCL context");
```

5. Create a command queue on the context that has been created before using the function called “clCreateCommandQueue” and parameters as context, device id and return error status. This returns a variable of datatype queue. This queue variable has to be used for en-queuing the kernels.

```
queue = clCreateCommandQueue(context, device, 0, &status);
```

```
checkError(status, "Error: could not create command queue");
```

6. Compile the code using Altera OpenCL Compiler or cross compiler installed in windows machine. Follow the steps to compile under Compilation. The compiler gives an aocx file which is the binary file to program the hardware. There is a function called “getBoardBinaryFile” with the parameter of board file name and device id. This returns a string of the entire binary file which can be programmed into the device.

```
std::string binary_file = getBoardBinaryFile(CL_FUNCTION, device);
```

7. Program can be extracted from the binary file using the function called “createProgramFromBinary” with parameters of context, the binary file in string format, and device id.

```
program = createProgramFromBinary(context, binary_file.c_str(), &device, 1);
```

8. Build the program using the function called “clBuildProgram” with the parameters of program file from the previous step, number of devices, address of the device. This returns an error status and check error using the status variable. This flashes the hardware in FPGA.

```
status = clBuildProgram(program, num_devices, &device, "", NULL, NULL);
```

```
checkError(status, "Error: could not build program");
```

9. Now the kernel can be created using the function called “clCreateKernel” with the parameters of program from step 7 and the kernel name (This has to be same as the function name in .cl file). This returns a variable of type cl_kernel.

```
  pyramidalKernel = clCreateKernel(program, "pyrUp", &status);
```

```
  checkError(status, "Error: could not create pyrUp kernel");
```

10. To create a memory buffer in FPGAs DRAM (64 MB in DE1 SoC), use the function called "clCreateBuffer" with parameters as context, Read/Write access details, size of the buffer and error status. This returns a pointer of data type cl_mem.

```
in_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(unsigned int) *
height * width, NULL, &status);
checkError(status, "Error: could not create device buffer");
```

11. Pass the kernel arguments using the function called "clSetKernelArg" with parameter as the kernel name, argument address, size of the argument and the address of the argument need to be passed.

```
status  = clSetKernelArg(pyramidalKernel, 0, sizeof(cl_mem), &in_buffer);

status |= clSetKernelArg(pyramidalKernel, 1, sizeof(cl_mem), &out_buffer);

checkError(status, "Error: could not set pyramidal args");
```

12. Write the data into the buffer using "clEnqueueWriteBuffer" with the parameters from step 10, queue , size of the buffer and size of the datatype. After every enqueue wait for it to fininsh using the function called "clFinish" with the parameter of queue. So it make sure all the data is written into the buffer. This function writes the data from local memory of ARM processor to FPGA.

```
status = clEnqueueWriteBuffer(queue, in_buffer, CL_FALSE, 0, sizeof(unsigned
int) * height * width, input, 0, NULL, NULL);

checkError(status, "Error: could not copy data into device");

status = clFinish(queue);

checkError(status, "Error: could not finish successfully");
```

13. To enqueue the kernel use the function called "clEnqueueNDRangeKernel" with the parameters of local/global work group size (Refer Khronos webpage for clarification) and queue. Wait till it completes using "clFinish". This "clFinish" acts like a synchronization function makes it wait till the previous function completes execution.

```
status = clEnqueueNDRangeKernel(queue, pyramidalKernel, 2, NULL, workSize,
workSize, 0, NULL, NULL);

checkError(status, "Error: could not enqueue pyramidal kernel");

status  = clFinish(queue);

checkError(status, "Error: could not finish successfully");
```

14. To read back the buffer from DRAM of FPGA to local memory of ARM, use the function called "clEnqueueReadBuffer" with the parameters of queue, local buffer address and size of the buffer and cl memory address.

```
status = clEnqueueReadBuffer(queue, out_buffer, CL_FALSE, 0, sizeof(unsigned
int) * out_width * out_height, output, 0, NULL, NULL);

checkError(status, "Error: could not copy data from device");

status = clFinish(queue);

checkError(status, "Error: could not successfully finish copy");
```

15. Clean up the memory that has been used in the OpenCL using this function called "clReleaseMemObject" with the parameter as the buffer name.

```
clReleaseMemObject(in_buffer);
```

The host code reads the image file from the sd card and transfers them into input buffer(Local memory of ARM processor) then passing it to DRAM of FPGA using opencl functions mentioned above. It reads back the output buffer from DRAM of FPGA to ARM's Local memory. Then writing it back to sdcard. The "display.cpp" code is self-explanatory and properly commented.

For better understanding of the functions mentioned refer khronos guide for OpenCL[6]. Altera's Sample code has been taken as reference for this implementation[7].

Device Code

Device code is the parallel code(.cl) includes the kernels that are running parallel. It has few global and local parameters such as

- global size
- local size
- global work id
- local work id

Each thread in a parallel execution has two ids. One is global id and local id. It is clear from the picture below and this helps in making the task distributed. The entire task is parallelized for each pixel element. It has the global dimension of two and global work size of the resolution which is described below.

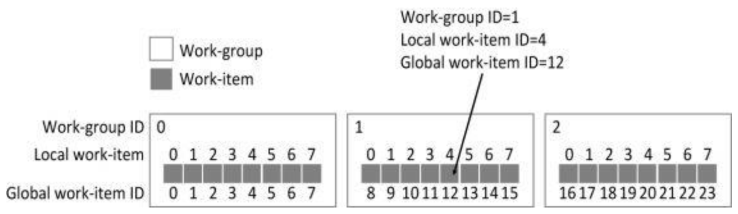


Figure 6: Global and Local ID Representation[8]

[6] https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf
[7] https://www.altera.com/support/support-resources/design-examples/design-software/opencl/sobel-filter.html
[8] http://www.fixstars.com/images/openclbook/dtp_462724_USER_CONTENT_0_html_3bb3f126.jpg

Steps involved in writing any device code

1. Get the global IDs according to the number of dimension, using the function called "get_global_id" with parameter as the selected dimension. This function returns an integer value.

   ```
   int x = get_global_id(0);

   int y = get_global_id(1);
   ```

2. Get the global size of the kernel execution using the function "get_global_size" with parameter as the dimension. This function returns an integer value. This corresponds to width and height of the image.

   ```
   int width = get_global_size(0);

   int height = get_global_size(1);
   ```

3. Calculate the index, this index is unique and represents the pixel number in the image.

   ```
   int index = x + y*width;
   ```

4. Whatever happens after this access the information according to the index, and each operation is running in parallel. Inside each kernel things are running sequentially.

Setting up the environment[9]

1. Download Quartus II and Cyclone V device support from Altera website and install into a desired location from https://dl.altera.com/?edition=web
2. Download Altera EDS from Altera website and install into a desired location from https://dl.altera.com/?edition=web. This software is required for cross-compiling the Host Program.
3. Download Altera SDK for OpenCL from http://dl.altera.com/opencl/?edition=subscription and install in Altera directory created from Quartus II installation (created in previous step). It needs a license file.
4. Download the Terasic board support package from Terasic website located here http://cd-de1-soc.terasic.com. Extract the BSP under into a folder named "de1soc" and copy to the Altera OpenCL board directory, altera/hdl/board/terasic/de1soc.

Compiling the OpenCL Kernel

1. Download and installed all the required software mentioned in the document "DE1SOC_OpenCL_v02.pdf"[10].
2. Make sure you have a valid Altera OpenCL license to use the Altera Offline Compiler (AOC).
3. Save the Altera OpenCL license file on your local hard drive and append the / path to the LM_LICENSE_FILE environment variable.

[9] Chris Wagner IS Report, http://ecee.colorado.edu/~siewerts/softdefined_photom/IS-Research/Chris_Wagner_IS_Report/Chris_Wagner_IS_Report.pdf
[10] http://www.terasic.com.tw/attachment/archive/836/DE1SOC_OpenCL_v02.pdf

4. Use the AOC tool to compile .aocx (Altera Offline Compiler Executable) file from OpenCl kernel source file pyramid.cl.
    a) Open the Altera EDS command shell.
    b) Open directory where the OpenCL (.cl) code is located (/device)
    c) Enter command:

    ```
    aoc -v --board de1soc_sharedonly hough.cl -o hough.aocx
    ```

    d) This will create a hough.aocx file in the /device folder. This is a FPGA hardware configuration file which is used to configure Cyclone V FPGA

Compiling the host code

1. The next step is to cross-compile the host code to target arm32 platform. To do this make sure you have the arm-linux-gnueabihf cross-compiler setup on your system.
2. To cross-compile the code, run the following command in the main directory.

    ```
    make
    ```

3. Move the following 3 files to the DE1SOC board by either copying them to your SD card/pen drive or by using SFTP (or copying them to the SD card manually).
    a. hough.aocx, FPGA Hardware configuration file
    b. hough, executable for host
    c. balloons.pgm, test image file

Running the OpenCL code on DE1-SoC

1. Load the Linux image that comes with the DE1-SoC BSP.
    a) Extract the Linux image file linux_sd_card_image.img from linux_sd_card_image.zip.
    b) Write the Linux image file linux_sd_card_image.img into the microSD card.
    c) Insert the SD card into the DE1-SoC development kit.
    d) Make sure the SW10 DIP switch (Figure 7) is set to MSEL[4:0] = 01010 [1]

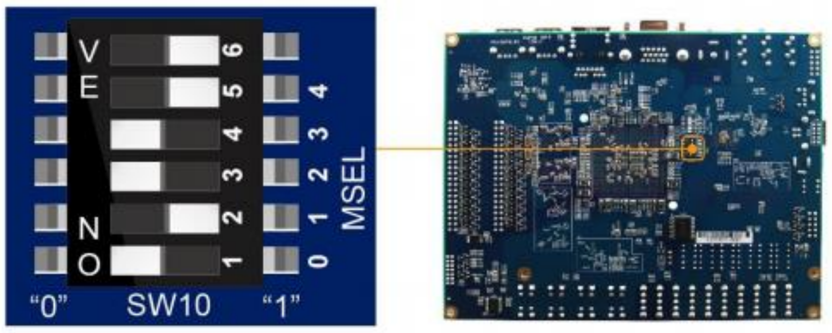


Figure 7: SW10 Switch Configuration

2. Using either putty or ssh log in as root to the terminal on the DE1-SoC running Linux.
3. Load initial environment variables using the following command:

    ```
    source ./init_opencl
    ```

4. Load the FPGA hardware configuration stored in hough.aocx by enter the following command

    ```
    aocl program /dev/acl0 ./hough.aocx
    ```

5. Make sure the host program is executable by running the following command

    ```
    chmod +x hough_exec
    ```

6. Run the host program by typing the following command

    ```
    ./hough_exec
    ```

Conclusion

The code for the transforms , make files, Power Analysis results were uploaded on github[11] repository. The link for the repository is given below. FPGA is in par with Jetson GPU and the OpenCL Platform is in primitive stage, it could become better at one point. Comparison would be better if both hardware platforms are running in OpenCL. Unfortunately, Jetson TK1 doesn't support OpenCL. If they support in the future, that could be a fair comparison.

[11] https://github.com/surjithbs17/OpenCL