

Lab Exercise

Optimizing the Hough Transform Kernel

Background Information

In this lab exercise, you will be optimizing a kernel for an FPGA to calculate the Hough Transform of the pixels within a picture. You will follow the flow presented in class to achieve this optimization – you will verify functionality using emulation, and you will use an HTML optimization report to decipher which optimizations might be beneficial.

The Hough transform is used in computer vision applications. After an image has been processed with an edge-detection algorithm such as a Sobel filter, you are left with a monochrome (black/white) image. It is useful for many further detection algorithms to consider the image as a set of lines. However, an image of black and white pixels is not a convenient or useful representation of these lines to algorithms such as object detection. The Hough transform is a transform from pixels to a set of “line votes.”

Before getting to the code, here is the theory behind the Hough Transform.

It is commonly known that a line can be represented in a slope-intercept form:

$$y = mx + b$$

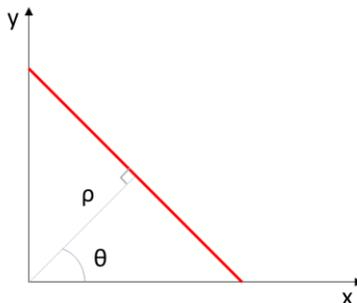
In this form, each line can be represented by two unique constants, the slope (m) and the y-intercept (b). So, every (m,b) pair represents a unique line. However, this form presents a couple of problems. First, since vertical lines have an undefined slope, it cannot represent vertical lines. Secondly, it is difficult to apply thresholding techniques to.

Therefore, for computational reasons in many detection algorithms, the Hesse normal form is used. This form has the equation below.

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

In this form, each unique line is represented by a pair (ρ, θ) (pronounced “rho” and “theta”). This form has no problem representing vertical lines, and you will learn how thresholding can easily be applied after the Hough Transform is applied.

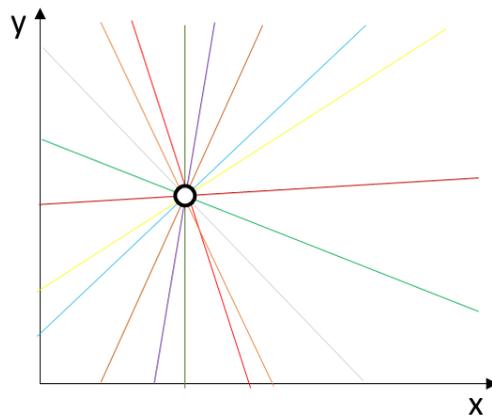
The following picture depicts what the ρ and θ values represent in the equation. For every line you want to represent (see the red line in the picture), there will be a unique line you can draw from the origin to it with the shortest distance (see the gray line in the picture). Another way of looking at this, is perpendicular line from the red line crossing the origin. ρ is the distance of the shortest line that can be drawn from the origin to the line you are wanting to represent. θ is the angle from the x-axis to that line.



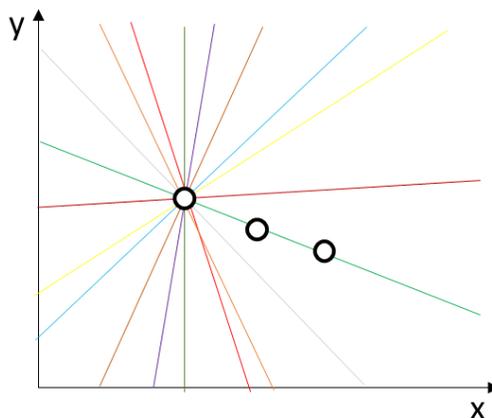
When working with an image, a corner of the image is traditionally considered to be the origin (the origin is not at the center), so the largest value ρ can be is the measure of the diagonal of the image. You can choose for the ρ values to all be positive, or to be allowed to be positive and negative. If you choose for all ρ values to be positive, then the range of θ is from 0 to 360 degrees. If you choose for ρ to be allowed to be positive or negative, the range of θ is from 0 to 180 degrees. These ranges are quantized in order to define a finite solution space.

Remember that the picture before being input into the Hough Transform has already gone through an edge detection algorithm and is therefore monochrome (each pixel is either black or white). The edges that have been detected are represented by the white pixels. The Hough Transform will transform the white pixels into an array of votes for lines.

Each white pixel in the image is potentially a point on a set of lines. The picture below represents the lines one pixel can potentially be a part of. (Note: not all potential lines are drawn, it is meant only for illustrative purposes.) A line with every potential slope passing through that pixel is potentially a line that appears in the image. So, 1 vote will be accumulated for each of these lines.



The code will loop through each pixel in the image, accumulating votes for lines as it goes. As a line accumulates more votes, the likelihood of that being a correct representation for a line in the picture goes up. So, as visualized below, the green line is going to accumulate 3 votes, which will make it a more likely candidate than the other lines. A threshold can easily be applied, therefore, by simply defining the amount of votes which is “enough” to define whether a line is present or not.



Now, let's take a look at the code to implement this transform. First, the complete algorithm will be shown, and then it will be explained piece by piece.

```
//A lookup table of sin and cos values at whole integer degree values
#include "sin_cos_values.h"

char pixel_array[IMAGE_HEIGHT*IMAGE_WIDTH];
short accumulators[THETAS*RHOS*2];

for (uint y=0; y<IMAGE_HEIGHT; y++) {
    for (uint x=0; x<IMAGE_WIDTH; x++){
        unsigned short int increment = 0;

        if (pixel_array[(WIDTH*y)+x] != 0) {
            increment = 1;
        } else {
            increment = 0;
        }

        for (int theta=0; theta<THETAS; theta++) {
            int rho = x*cos_table[theta] + y*sin_table[theta];
            accumulators[(THETAS*(rho+RHOS))+theta] += increment;
        }
    }
}
```

Let's first take a look at the array declarations at the top of the code.

```
char pixel_array[IMAGE_HEIGHT*IMAGE_WIDTH];
short accumulators[THETAS*RHOS*2];
```

The pixel array is the image itself, and each pixel occupies a place in the array.

The accumulators array will keep track of our line votes. Each place in the array represents a potential line in the image. Recall that a unique line is represented by a pair (ρ, θ) . Therefore, the number of all of the potential lines in our image is equal to all potential values of ρ times all potential values of θ . ρ is the distance from the origin, which is defined as a corner of the picture. The greatest value of ρ is the measure of the diagonal of the picture. We will also let ρ be either positive or negative so that θ is bounded between 0 and 180 degrees. We will quantize at integer values for ρ , and integer degrees for θ . Our number of accumulators, therefore, is the measure of the diagonal of the picture (RHOS in the code) times 2 times 180 degrees (THETAS in the code).

Now, let's examine at the code to implement the algorithm.

```

for (uint y=0; y<IMAGE_HEIGHT; y++) {
    for (uint x=0; x<IMAGE_WIDTH; x++){
        ...
    }
}

```

The outer loop will loop through each pixel in the image, accumulating votes for all the potential lines a pixel could be a part of as it goes.

```

unsigned short int increment = 0;

if (pixel_array[(WIDTH*y)+x] != 0) {
    increment = 1;
} else {
    increment = 0;
}

```

If the pixel is white (!=0), then we will add a 1 to all of the accumulators for potential lines defined by that pixel. Otherwise, we will not add anything to the accumulator. We do it in this manner so that the control logic inside the FPGA and the computation logic that we will duplicate later with pragmas is simpler and consumes less logic resources..

```

for (int theta=0; theta<THETAS; theta++) {
    int rho = x*cos_table[theta] + y*sin_table[theta];
    accumulators[(THETAS*(rho+RHOS))+theta] += increment;
}

```

For each pixel location, all of the lines that can have that pixel location as part of their values needs to receive a vote. Recall the formula we are using to represent a line:

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

The x and y values are constants for the duration of this inner loop. We will plug in every possible value of θ , along with the x and y, and solve for ρ given that θ . We will then cast a vote (add 1 to the accumulator location) for that (ρ, θ) pair. In this manner, we cast a vote for every line that is a possibility, traversing the possibilities in an arc from 0 degrees to 180 degrees.

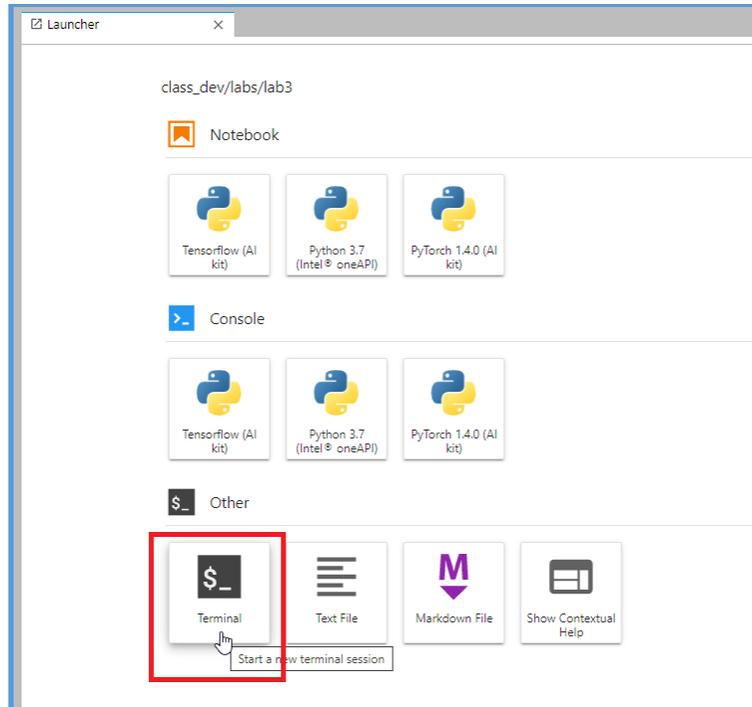
Now, let's begin the lab and see what optimizations we can do to improve the total execution time in the FPGA. Don't worry if you don't completely understand the algorithm, it is sufficient to think of it as a convenient piece of code to perform optimizations on. However, if you would like to learn more, the Wikipedia* entry for the Hough Transform is a great place to start:

https://en.wikipedia.org/wiki/Hough_transform

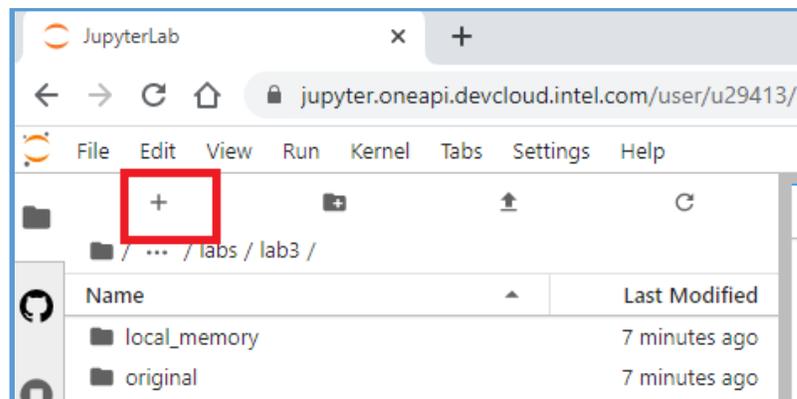
Part A. Setup

1. For this lab, you will be working directly in the terminal in Jupyter Lab, and also opening the source code files directly. To get started, open a terminal within the Jupyter Lab if you do not already have one open.

To open a terminal, double-click the Terminal button in the Launcher tab of Jupyter Lab, as shown below.



If you do not see the Launcher tab, click the "+" button at the top left of Jupyter Lab, and a Launcher tab will open. The "+" button is shown below.



- _____ 2. After the terminal is open, navigate to the lab directory, and run the setup script. (Note: The instructions for the lab will assume you unzipped your lab files to your home directory.)

```
$ cd ~/labs/lab3
$ source ./setup.sh
```

Note: It is important to **source** the setup script, and not simply run it, since it is modifying environment variables. It is modifying your path to point to the Beta 8 versions of the tools, since there is a report file rendering present in the current version we need to avoid.

- _____ 3. Here are some important notes about the lab that you should read before moving on:

Your results may vary somewhat from the screenshots, and the code line numbers may vary slightly from what is shown. This is because the screenshots were taken with a different version of the tool, and the code has also changed slightly.

This lab is designed to give you valuable insight and experience each step of the way. You will go through a series of optimizations to a kernel. **If you want to skip the code modifications and use the solutions, it is OK! You will still learn a lot. If you do not get to every part, it is OK! You will still learn a lot.**

Solutions for every coding step are available in the directory `~/labs/lab3/solutions`.

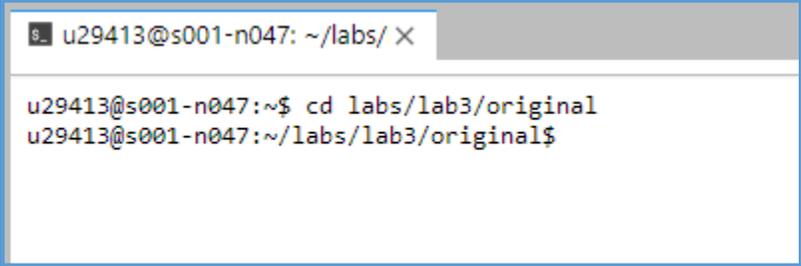
The file `~/labs/lab3/commands.txt` contains the commands you will be running so you can cut and paste them if you desire.

A clean script is contained in every subdirectory if you need it. Run it by typing `source ./clean.sh`

Part B. Examine the Code Structure

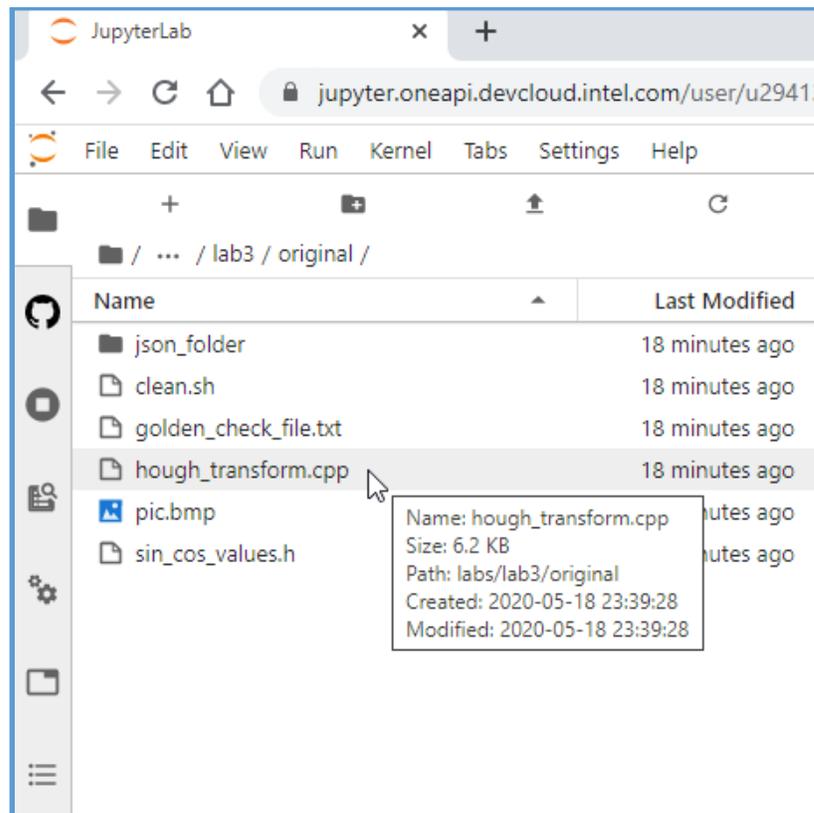
1. Once you have a terminal open within Jupyter Lab, at the terminal prompt browse to the directory named "original" within lab3. The remainder of the instructions in this lab will assume you unzipped the labs.zip file into your home directory (denoted by ~). If you unzipped the files into a different directory, replace the ~ in the commands with the directory you started from.

```
$ cd ~/labs/lab3/original
```

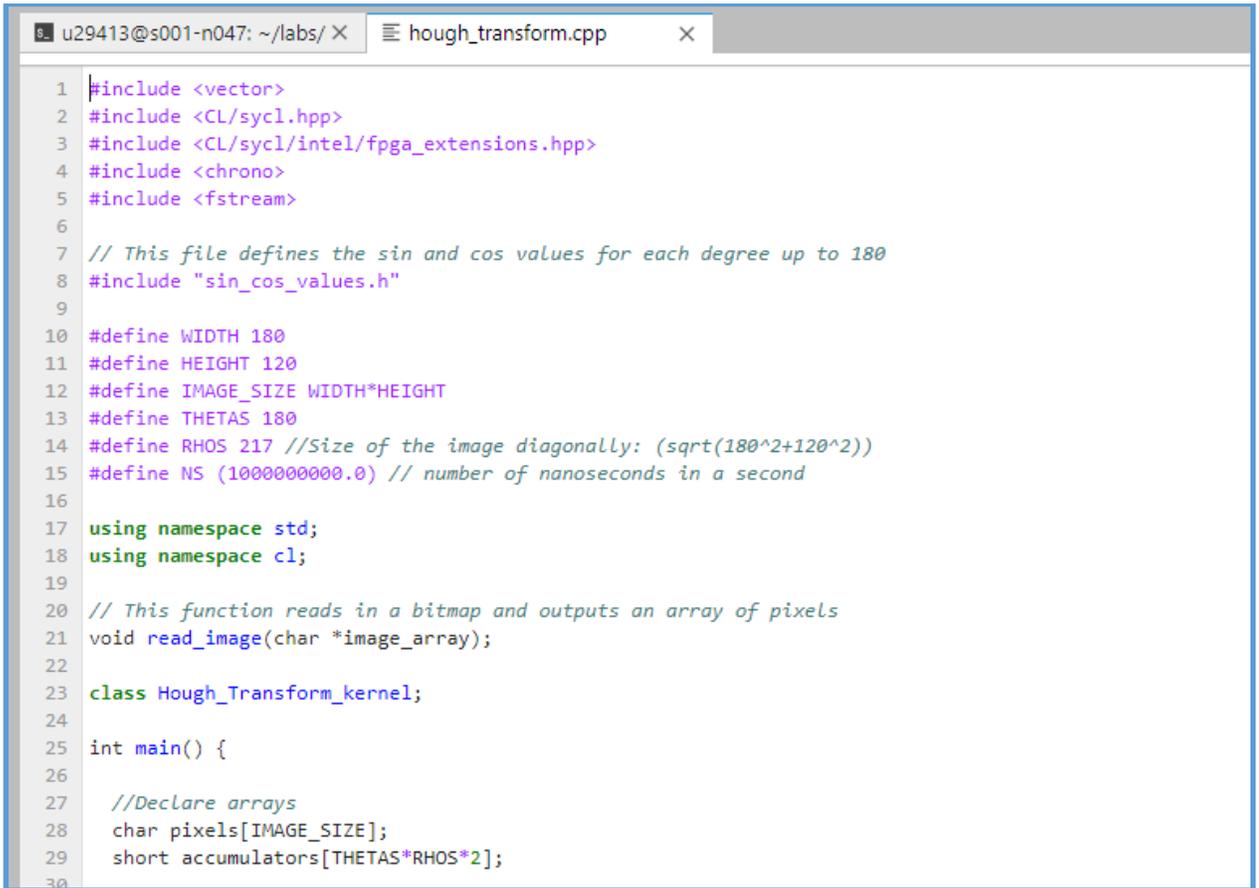


```
u29413@s001-n047: ~/labs/ X
u29413@s001-n047:~$ cd labs/lab3/original
u29413@s001-n047:~/labs/lab3/original$
```

2. Open the source code within Jupyter Lab by double-clicking on the file ~/labs/lab3/hough_transform.cpp in the file browser on the left side of the Jupyter Lab environment. This is shown below.



3. The code should now be open in a tab within Jupyter Lab as shown below.



```
1 #include <vector>
2 #include <CL/sycl.hpp>
3 #include <CL/sycl/intel/fpga_extensions.hpp>
4 #include <chrono>
5 #include <fstream>
6
7 // This file defines the sin and cos values for each degree up to 180
8 #include "sin_cos_values.h"
9
10 #define WIDTH 180
11 #define HEIGHT 120
12 #define IMAGE_SIZE WIDTH*HEIGHT
13 #define THETAS 180
14 #define RHOS 217 //Size of the image diagonally: (sqrt(180^2+120^2))
15 #define NS (1000000000.0) // number of nanoseconds in a second
16
17 using namespace std;
18 using namespace cl;
19
20 // This function reads in a bitmap and outputs an array of pixels
21 void read_image(char *image_array);
22
23 class Hough_Transform_kernel;
24
25 int main() {
26
27     //Declare arrays
28     char pixels[IMAGE_SIZE];
29     short accumulators[THETAS*RHOS*2];
30 }
```

4. In the text editor, search for “Block off this code” by using Ctrl-F to open a Find dialog. This will take you to the section of code with all of the SYCL code. Blocking off the code in this manner and restricting all of the SYCL constructs to the block ensures that when the block finishes execution, the SYCL objects are destructed. The destructor routines for the SYCL objects ensure that all work with them is complete before destruction. In this manner, the block (enclosed by {}) acts as a synchronization mechanism.

This is important because without this synchronization mechanism, the buffer that holds the accumulators will not be written back to the host before the host reads the memory allocated for the accumulators.

Blocking in this manner is the technique most SYCL samples use in order to ensure synchronization of data at the host.

Note, another way to synchronize this without a block would be to create an access to the buffer from the host. Since only one thing can interact with the buffer at a time, the device’s interactions would finish up before the host could use its accessor. If you would like to see an example of this alternate technique, consult the sample called “FPGA tutorial: Caching local memory to improve performance” that you can generate using the command `oneapi-cli`.

- ____ 5. The command scope in a DPC++ program is where actions are submitted to the queue. Search for “Device queue submit” within the source code. Here you will see where the command scope begins. The screenshot shown next illustrates the entire command scope.

```
83 //Device queue submit
84 queue_event = device_queue.submit([&](sycl::handler &cgh) {
85 //Uncomment if you need to output to the screen within your kernel
86 //sycl::stream os(1024,128,cgh);
87 //Example of how to output to the screen
88 //os<<"Hello world " <<8+5<<sycl::endl;
89
90 //Create accessors
91 auto _pixels = pixels_buf.get_access<sycl::access::mode::read>(cgh);
92 auto _sin_table = sin_table_buf.get_access<sycl::access::mode::read>(cgh);
93 auto _cos_table = cos_table_buf.get_access<sycl::access::mode::read>(cgh);
94 auto _accumulators = accumulators_buf.get_access<sycl::access::mode::read_write>(cgh);
95
96 //Call the kernel
97 cgh.single_task<class Hough_transform_kernel>([=]() {
98     for (uint y=0; y<HEIGHT; y++) {
99         for (uint x=0; x<WIDTH; x++){
100             unsigned short int increment = 0;
101             if (_pixels[(WIDTH*y)+x] != 0) {
102                 increment = 1;
103             } else {
104                 increment = 0;
105             }
106             for (int theta=0; theta<THETAS; theta++){
107                 int rho = x*_cos_table[theta] + y*_sin_table[theta];
108                 _accumulators[(THETAS*(rho+RHOS))+theta] += increment;
109             }
110         }
111     }
112 });
113 });
114
115 });
```

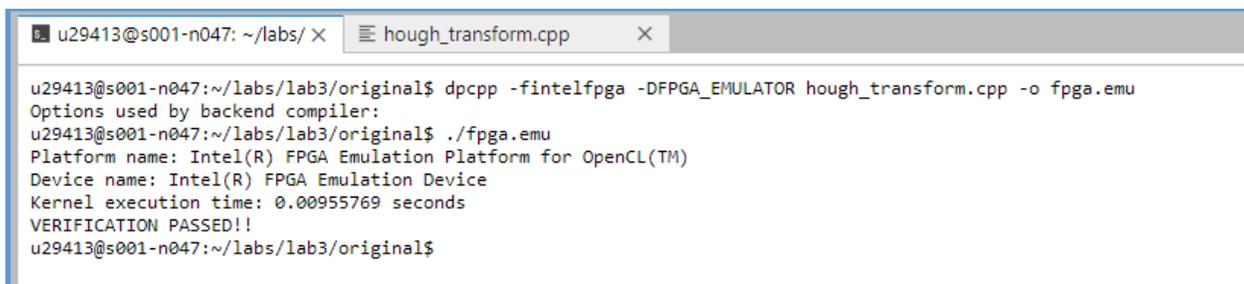
- ____ 6. Recall that in DPC++ the kernel scope is what encases the kernel code that will run on the device. Search within the code for “Call the kernel”. It is the code shown below which comprises the kernel code. We will be optimizing the code within the kernel scope during this lab.

```
96 //Call the kernel
97 cgh.single_task<class Hough_transform_kernel>([=]() {
98     for (uint y=0; y<HEIGHT; y++) {
99         for (uint x=0; x<WIDTH; x++){
100             unsigned short int increment = 0;
101             if (_pixels[(WIDTH*y)+x] != 0) {
102                 increment = 1;
103             } else {
104                 increment = 0;
105             }
106             for (int theta=0; theta<THETAS; theta++){
107                 int rho = x*_cos_table[theta] + y*_sin_table[theta];
108                 _accumulators[(THETAS*(rho+RHOS))+theta] += increment;
109             }
110         }
111     }
112 }
113 });
114
```

- ____ 7. Examine the remainder of the code to your desired level of understanding. If there is anything you do not understand, your instructor will be happy to answer your questions.

Part C. Emulate the Code and Examine the Optimization Report

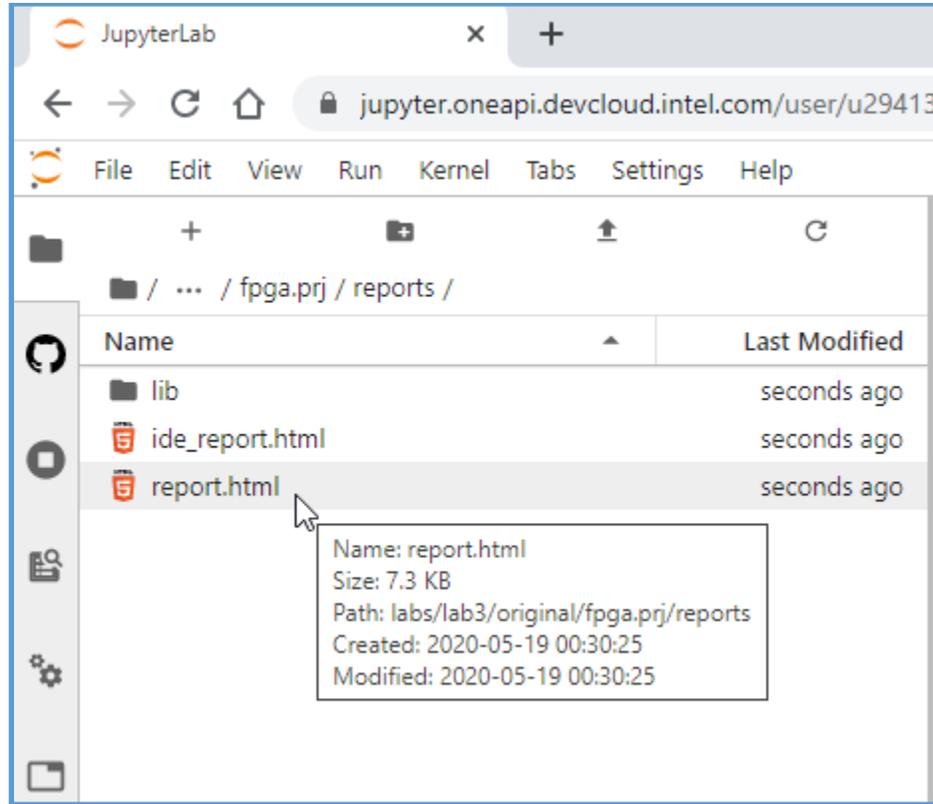
1. You should still be at the terminal prompt in the directory `~/labs/lab3/original`. Ensure that you are in this directory.
2. We will now compile the code for emulation. Recall from the presentation and the previous lab that emulation is used to ensure functionality of the code, including the code within the kernel scope. To compile for emulation, run the following command:
`dpcpp -fintel-fpga -DFPGA_EMULATOR hough_transform.cpp -o fpga.emu`
3. This command will produce a file called `fpga.emu` which is an executable which will run on the host that contains the host code as well as an emulated version of the kernel code. Run this code now by typing in the following command:
`./fpga.emu`
4. You should see the following output to your screen. This means the run was successful. If you were working with code of your own, it would be expected that you would go through many rounds of emulation to get the functionality of your code correct.



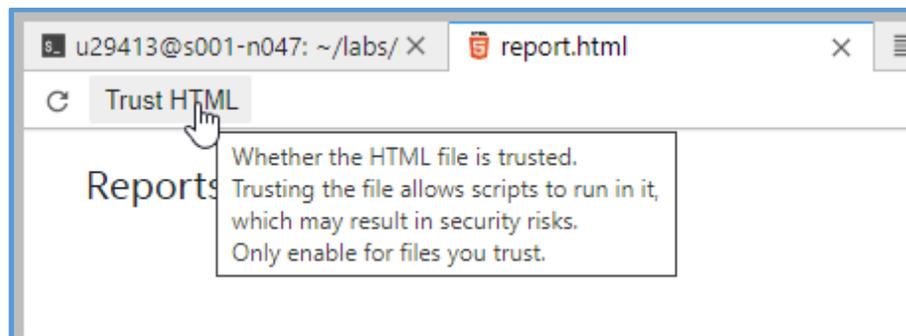
```
u29413@s001-n047: ~/labs/ × hough_transform.cpp ×
u29413@s001-n047:~/labs/lab3/original$ dpcpp -fintel-fpga -DFPGA_EMULATOR hough_transform.cpp -o fpga.emu
Options used by backend compiler:
u29413@s001-n047:~/labs/lab3/original$ ./fpga.emu
Platform name: Intel(R) FPGA Emulation Platform for OpenCL(TM)
Device name: Intel(R) FPGA Emulation Device
Kernel execution time: 0.00955769 seconds
VERIFICATION PASSED!!
u29413@s001-n047:~/labs/lab3/original$
```

5. After you achieve successful emulation of your kernel, the next step is to examine the optimization report to examine the estimate performance of the loops, the structure of the kernel's memory, and the amount of resources that the kernel consumes in the FPGA. We will do that now. In order to compile the code to an object file and generate a static HTML optimization report for the kernel, run the following 2 commands. The second of these commands will take a little longer to complete than the prior (15-30 seconds).
`dpcpp -fintel-fpga -c hough_transform.cpp -o fpga.o`
`dpcpp -fintel-fpga -fsycl-link -Xshardware fpga.o`

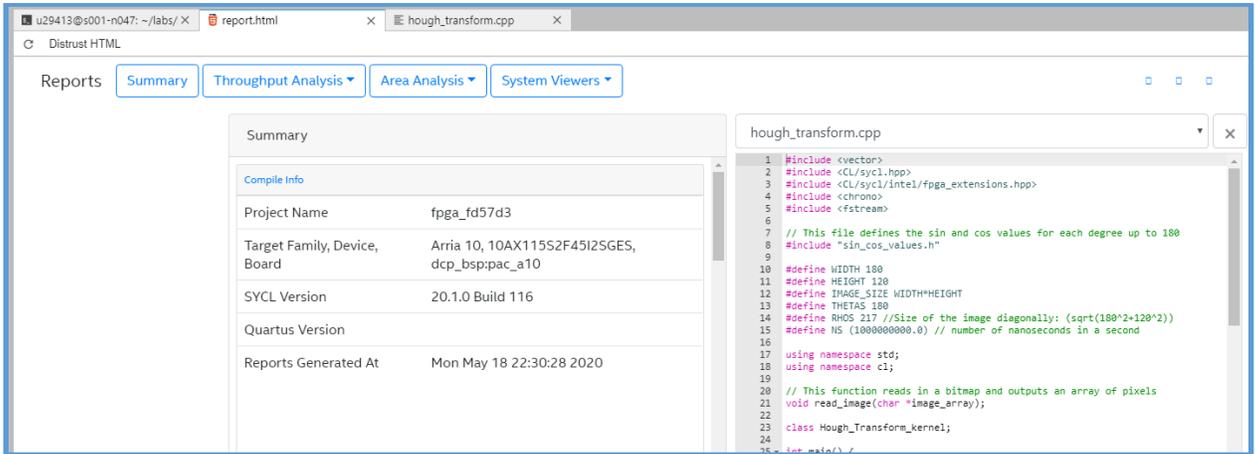
6. After the commands in the last step have completed, a subdirectory called fpga.prj/ will be created. Under that directory, there is a reports/ subdirectory, and the optimization report is there called report.html. Open that report now by browsing to it on the left side panel of Jupyter Lab and double-clicking it.



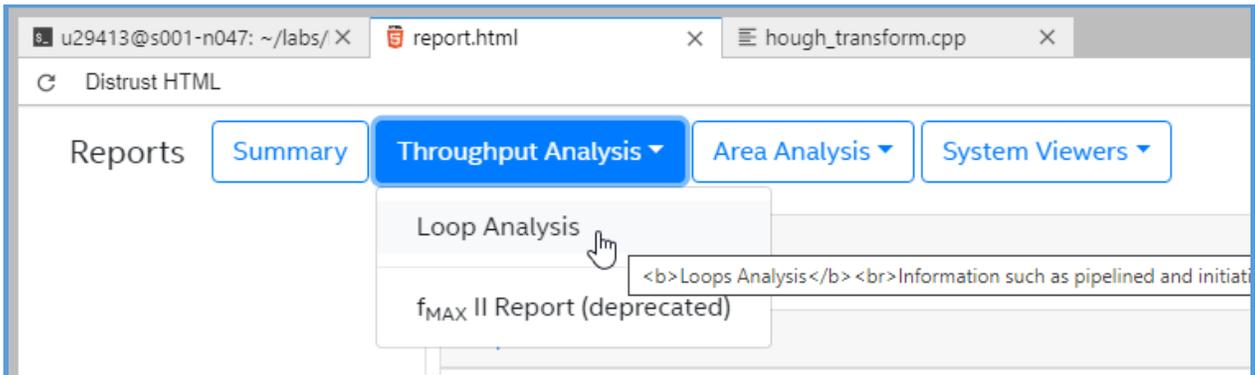
7. Click "Trust HTML" if needed to get the report file to open fully.



8. You will now see the static HTML optimization report in your browser. Look at the different report sections by clicking on the boxes outlined in blue at the top to get an overview of the type of information contained in them.



9. This first implementation of the Hough Transform was not a very good one. Go to the Loops Analysis section of the report by going to the Throughput Analysis box, clicking it, and then clicking Loops Analysis, as shown below.



10. This section of the report gives you an analysis of how well your loops will perform. For single work-item kernels (those launched with the single_taskSYCL API call), the Initiation Interval (II) of every loop is calculated and reported. Recall that the II is the number of clock cycles between new pieces of data being input into the processing pipeline. A high II means that many cycles are spent stalling, with the hardware not being used.

Examine the II values in this version of the kernel by clicking on the line that begins with Kernel: under the Loop List section of the report.

The screenshot shows a web browser window with the following content:

- Browser tabs: u29413@s001-n047: ~/class, report.html, commands.txt
- Page title: Distrust HTML
- Navigation buttons: Reports, Summary, Throughput Analysis, Area Analysis, System Viewers
- Left sidebar: Loop List
 - Kernel: Hough_Transform_kernel
 - Hough_Transform_kernel.B1
 - Hough_Transform_kernel.B3
 - Hough_Transform_kernel.B5
- Main content: Loop Analysis table

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency
Kernel: Hough_Transform_kernel	hough_transform.cpp:102				
Hough_Transform_kernel.B1	hough_transform.cpp:103	Yes	>=1	240.0	11
Hough_Transform_kernel.B3	hough_transform.cpp:104	Yes	>=1	240.0	203
Hough_Transform_kernel.B5	hough_transform.cpp:111	Yes	~233	240.0	481

The highest II is 233 clock cycles! This is very high, amounting to hundreds of wasted clock cycles between each loop iteration!

11. Click on the line in the report where the highest loop II is shown. Details about why this II is so long will be shown in the bottom pane of the report. The block of code where the bottleneck is inferred will also be highlighted.

Loop List

- Kernel: Hough_Transform_kernel (h)
 - Hough_Transform_kernel.B1 (h)
 - Hough_Transform_kernel.B3
 - Hough_Transform_kernel.B5

Loop Analysis

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency
Kernel: Hough_Transform_kernel	hough_transform.cpp:102				
Hough_Transform_kernel.B1	hough_transform.cpp:103	Yes	>=1	240.0	11
Hough_Transform_kernel.B3	hough_transform.cpp:104	Yes	>=1	240.0	203
Hough_Transform_kernel.B5	hough_transform.cpp:111	Yes	~233	240.0	481

Details

Hough_Transform_kernel.B5:

- Compiler failed to schedule this loop with smaller II due to memory dependency:
 - From: Load Operation ([hough_transform.cpp: 113](#))
 - To: Store Operation ([hough_transform.cpp: 113](#))
- Most critical loop feedback path during scheduling:
 - 192.00 clock cycles Load Operation ([hough_transform.cpp: 113](#))

You can also jump to the line of code where the loop was written by clicking on the link within the “Source Location” column.

Hough_Transform_kernel.B1	hough_transform.cpp:103	Yes	>=1	240.0	11
Hough_Transform_kernel.B3	hough_transform.cpp:104	Yes	>=1	240.0	203
Hough_Transform_kernel.B5	hough_transform.cpp:111	Yes	~233	240.0	481

```

107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
    increment = 1;
  } else {
    increment = 0;
  }
  for (int theta=0; theta<THETAS; theta++){
    int rho = x*cos_table[theta] + y*sin_table[theta];
    _accumulators[(THETAS*(rho+RHOS))-theta] += increment;
  }
  });
  });
  //Wait for the kernel to get finished before reporting the profiling

```

The bottlenecks occurring are “memory dependencies.” This means we are waiting on an operation from memory to complete before starting a new iteration of the pipeline. (As a side note, the other type of dependency that can cause a bottleneck is a data dependency, which means a calculation takes too long to complete.) They all occur on lines 112 and 113, where we are obtaining values from the sine and cosine lookup tables and looking up and incrementing accumulators.

- ____ 12. Examine the Details shown at the bottom for this loop. Notice memory dependencies are mentioned often. Memory dependencies are making the II of the loop very large. This means we are waiting on an operation from memory to complete before starting a new iteration of the pipeline.

Details

Hough_Transform_kernel.B5:

- Compiler failed to schedule this loop with smaller II due to memory dependency:
 - From: Load Operation ([hough_transform.cpp: 113](#))
 - To: Store Operation ([hough_transform.cpp: 113](#))
- Most critical loop feedback path during scheduling:
 - 192.00 clock cycles Load Operation ([hough_transform.cpp: 113](#))

- ____ 13. Scroll until you see the section entitled “Most critical loop feedback path during scheduling.”

Details

- Most critical loop feedback path during scheduling:
 - 192.00 clock cycles Load Operation ([hough_transform.cpp: 113](#))
 - 40.00 clock cycles Store Operation ([hough_transform.cpp: 113](#))
 - 0.85 clock cycles 16-bit Integer Add Operation ([hough_transform.cpp: 113](#))
 - 0.29 clock cycles 1-bit Or Operation ([hough_transform.cpp: 113](#))
- Hyper-Optimized loop structure: n/a
- It is an approximation due to the following stalleable instructions:

One of the lines of code having a large impact to the scheduling of the loop is happening at line 113. When the code is examined, it can be seen that this line accesses the accumulators with both a load and a store (to increment the value). We will optimize this bottleneck some in the next section of the lab.

```
110     }
111     for (int theta=0; theta<THETAS; theta++){
112         int rho = x*_cos_table[theta] + y*_sin_table[theta];
113         _accumulators[(THETAS*(rho+RHOS))+theta] += increment;
114     }
115 }
116 }
117 }
118 );
```

- ____ 14. Open the “Area Analysis of System” section of the report. Expand the Kernel System section. This shows the resources used by the kernel.

The screenshot shows the 'Area Analysis' report in Intel Quartus Prime. The 'Area Analysis of System' section is selected, and the 'Kernel System' is expanded in the left sidebar. The main table displays resource utilization for various components:

	ALUTs	FFs	RAMs	MLABs	DSPs
<input type="checkbox"/> Static Partition	179950 (21%)	358572 (21%)	492 (18%)	0 (0%)	123 (8%)
<input type="checkbox"/> Kernel System	13702 (2%)	22747 (1%)	215 (8%)	75 (0%)	2 (0%)
Global interconnect	1720	2581	61	0	0
System description ROM	0	67	2	0	0
<input type="checkbox"/> Hough_Transform_kernel	11982 (1%)	20099 (1%)	152 (6%)	75 (0%)	2 (0%)

- ____ 15. The next step would be to compile the kernel to a full executable for the FPGA and to run it on the FPGA itself (including the -Xsprofile switch if it is desired to see profiling information in the Intel® VTune™ Amplifier). Remember you can do all of that (including running it on an FPGA board!) on the Intel® DevCloud. Since that step takes hours, we won't do it here. To get started on the Intel DevCloud with Intel FPGAs, visit this site after the lab: <https://software.intel.com/en-us/articles/getting-started-with-intel-devcloud-for-oneapi-projects> and click on “FPGA Vector-Add Sample Walkthrough.”

If you did run the kernel on an FPGA in the DevCloud, you would find the execution time is **about 2.704 seconds**. We will compare that to other runs as we optimize the kernel.

```

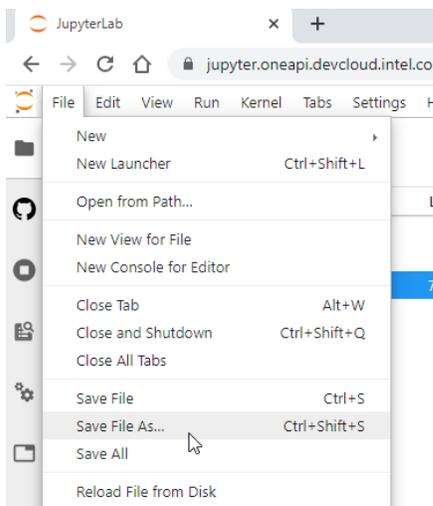
u29413@s001-n082:~/DevConFPGALab/original$ ./hough_original.out
Platform name: Intel(R) FPGA SDK for OpenCL(TM)
Device name: pac_a10 : Intel PAC Platform (pac_1fb00000)
Kernel execution time: 2.70481 seconds
VERIFICATION PASSED!!
u29413@s001-n082:~/DevConFPGALab/original$

```

Part D. Implement Local Memory for the Accumulators

1. Change the directory to `~/labs/lab3/local_memory` by typing the following command in the terminal prompt.

```
cd ../local_memory
```
2. Open the file `~/labs/lab3/hough_transform_CHANGE_ME.cpp` by browsing to it in the left panel in Jupyter Lab and clicking on it.
3. For this optimization, you will implement a local memory to hold the accumulator values since our II is still very high due to the access time required to load and store these values from/to global memory. Recall from the presentation that to implement a local memory in a single work-item kernel, you simply declare an array within the kernel scope. In the file `hough_transform_CHANGE_ME.cpp`, create a local memory for the accumulators by declaring an array called `accum_local` in the kernel scope of the code. `accum_local` should be the same size and type as the array called `accumulators`.
4. When you are finished modifying the code, save the file as `hough_transform.cpp`. Do this by using the File -> Save File As... dialog within the Jupyter Lab environment.



Remember that solutions are available if you need them or would like to get through the lab faster. The solution for this step of recoding is available in `~/labs/lab3/solutions/local_memory/hough_transform.cpp`.

5. Compile the code for emulation using the same command you have used in previous steps. If there are syntax errors, correct them and recompile.
6. Run the emulation executable by using the command `./fpga.emu`. If you do not see the message `VERIFICATION PASSED!`, then correct your code and try again.
7. Compile your code into an object file and generate a static optimization report using the 2-step method with the `dpcpp` commands used in previous steps.

____ 8. Open the static optimization report by browsing to it in Jupyter Lab and double clicking. It will be in the following location. You may have to click “Trust HTML.”

`~/labs/lab3/local_memory/fpga.prj/reports/report.html`

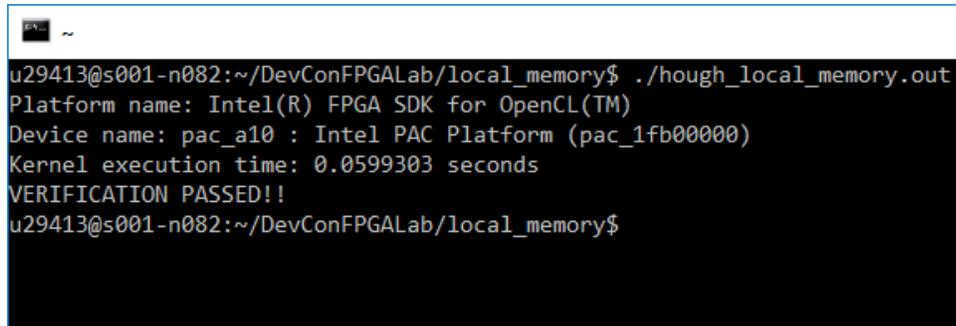
____ 9. Open the Loops Analysis section of the report and observe the improved II. Wow, what a difference! The largest II is now only ~2 clock cycles (it is approximate because it is an interaction with global memory, which has some nondeterminism).

The screenshot shows a Jupyter Lab interface with a browser window displaying a static optimization report. The browser tabs include 'u29413@s001-n047: ~/class', 'report.html', and 'commands.txt'. The report interface has a 'Reports' section with buttons for 'Summary', 'Throughput Analysis', 'Area Analysis', and 'System Viewers'. The 'Loop Analysis' section is active, showing a tree view on the left and a table on the right. The table lists the following data:

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency
Kernel: Hough_Transform_kernel	hough_transform.cpp:102				
Hough_Transform_kernel.B2	hough_transform.cpp:106	Yes	1	240.0	6
Hough_Transform_kernel.B4	hough_transform.cpp:110	Yes	>=1	240.0	11
Hough_Transform_kernel.B6	hough_transform.cpp:111	Yes	>=1	240.0	202
Hough_Transform_kernel.B8	hough_transform.cpp:119	Yes	~2	240.0	236
Hough_Transform_kernel.B9	hough_transform.cpp:126	Yes	~1	240.0	9

____ 10. Open the Area Analysis of System section of the report. Also open that section for the last compilation (from the constant_cache/subdirectory). Observe how the FPGA resource utilization has changed, especially that the onchip RAMs being used have gone up. This makes sense because we used them to hold local copies of the accumulators.

- ____ 11. We will not run on the FPGA in the interest of time, but if you did compile and run this code on the FPGA, you would find the runtime would be about **0.0599 seconds**, much reduced from our last 2 runtime numbers.

A terminal window with a black background and white text. The prompt is 'u29413@s001-n082:~/DevConFPGALab/local_memory\$'. The user has entered './hough_local_memory.out'. The output is: 'Platform name: Intel(R) FPGA SDK for OpenCL(TM)', 'Device name: pac_a10 : Intel PAC Platform (pac_1fb00000)', 'Kernel execution time: 0.0599303 seconds', and 'VERIFICATION PASSED!!'. The prompt is shown again at the end.

```
u29413@s001-n082:~/DevConFPGALab/local_memory$ ./hough_local_memory.out
Platform name: Intel(R) FPGA SDK for OpenCL(TM)
Device name: pac_a10 : Intel PAC Platform (pac_1fb00000)
Kernel execution time: 0.0599303 seconds
VERIFICATION PASSED!!
u29413@s001-n082:~/DevConFPGALab/local_memory$
```

Part F. Unroll the Inner Loop and Apply the ivdep attribute

- _____ 1. Change the directory to `~/labs/lab3/unroll_ivdep` by typing the following command in the terminal prompt.

```
cd ../unroll
```

- _____ 2. For this optimization, you will unroll the inner loop of the code in order to direct the compiler to create hardware so that more loop iterations can occur in parallel. You will also apply the pragma `ivdep` to the loop so that the compiler knows the memory operations within the loop are independent of memory operations that happen during other iterations of the loop (if they are considered dependent, the loop will not be unrolled).

In order to unroll the loop, a pragma presented during class needs to be applied to the loop by inserting the pragma before the loop in the code. Also, in order to apply the `ivdep` attribute to the code, the attribute needs to be inserted before the loop. The pragma and attribute should be applied to the loop that loops through every possible value of `theta`. **Unroll the loop 32 times, anything larger will result in long compile times (which would be ok if we weren't time constrained for the lab) for the optimization report stage.**

Make the change to the code and save it as `hough_transform.cpp` or copy in the solution in `~/labs/lab3/solutions/unroll/hough_transform.cpp`.

- _____ 3. Compile the code for emulation, and run the emulation executable (see past steps for the commands). Do this until you get the VERIFICATION PASSED! message.
- _____ 4. Compile the code into an object file and static optimization report using the `dpcpp` commands presented in past steps. Expect this step to take a couple of minutes.
- _____ 5. Open the static optimization report by browsing to it and clicking, as in previous steps.

6. Open the Loops Analysis section of the report. Observe that the loop has been unrolled 32 times and that the II is improved.

Name	Source Location	Pipelined	II	Scheduled fMAX	Latency
Kernel: Hough_Transform_kernel	hough_transform.cpp:102				
Hough_Transform_kernel.B2	hough_transform.cpp:106	Yes	1	240.0	6
Hough_Transform_kernel.B4	hough_transform.cpp:110	Yes	>=1	240.0	6
Hough_Transform_kernel.B6	hough_transform.cpp:111	Yes	>=1	240.0	199
32X Partially unrolled Hough_Transform_kernel.B7	hough_transform.cpp:121	Yes	~1	240.0	467
Hough_Transform_kernel.B8	hough_transform.cpp:128	Yes	~1	240.0	12

7. Now, open the “Kernel Memory Viewer” section of the report. It is under the Systems Viewers heading.

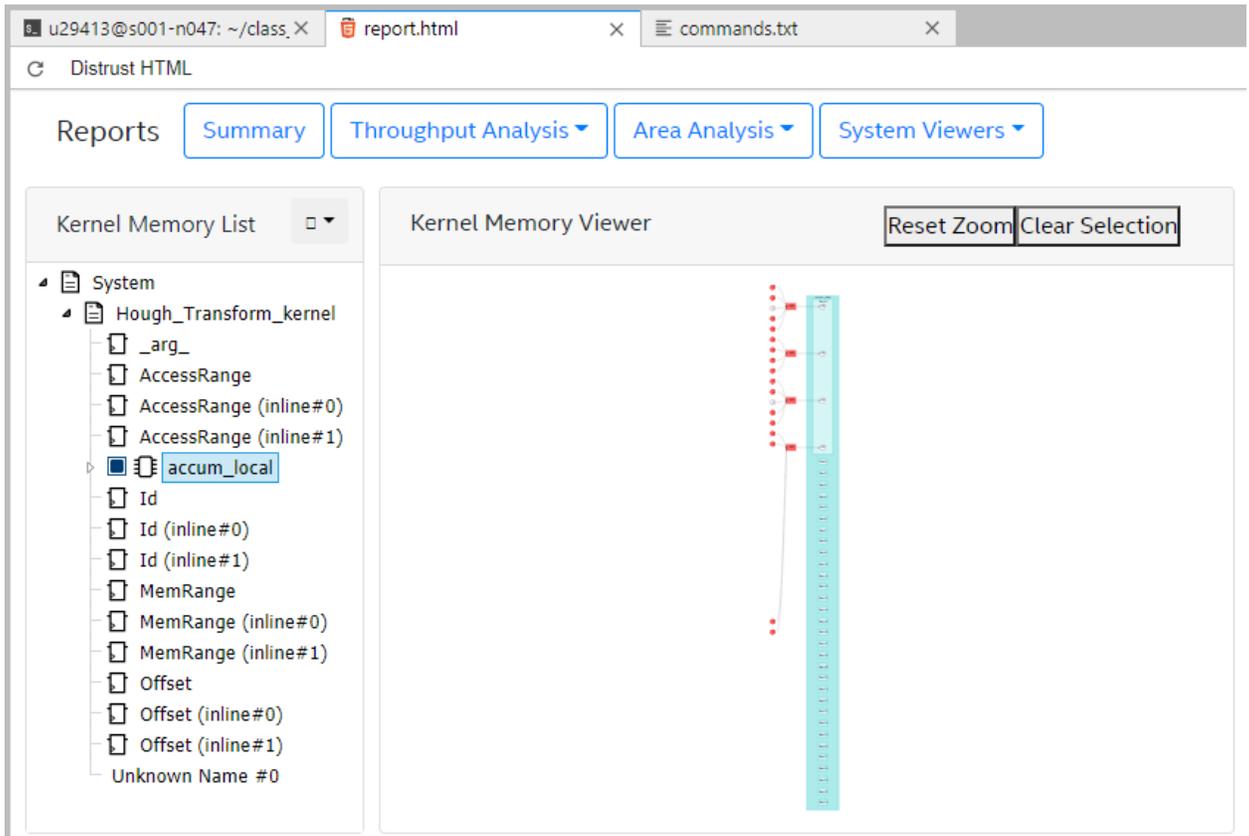
Kernel Memory Viewer
Shows memory system connections including those between loads and stores specific to logical ports on the memory banks. You can also view replicate nodes for each memory bank.

```

8 #include <CL/s>
9 #include <CL/s>
10 #include <chrn>
11 #include <fstr>
12
13 // This file d
14 #include "sin_
15
16 #define WIDTH :
17 #define HEIGHT
18 #define IMAGE_
19 #define THETAS
20 #define RHOS 2
21 #define NS (10
22
23 using namespac
24
25 // This functi
26 void read_imag
27
28 class Hough_Tr

```

8. Click on `accum_local` in the Memory List, as shown below. This section of the report is giving us a visual representation of the onchip memory structures built for our kernel scope code. Red in general is bad, it means that there is potential stalling on those load and store points. Since we unrolled the loop, 32 values from the `accum_local` memory structure are demanded by the unrolled loop structure at one time. This massive demand from the memory structure has caused the need for arbitration, and introduced potential stalling.



9. If you compiled to a full FPGA executable and ran this version of the code on an FPGA, you would see the runtime is about **0.0205 seconds**.

```
u29413@s001-n082:~/DevConFPGALab/unroll_ivdep$ ./hough_unroll_ivdep.out
Platform name: Intel(R) FPGA SDK for OpenCL(TM)
Device name: pac_a10 : Intel PAC Platform (pac_1fb00000)
Kernel execution time: 0.020524 seconds
VERIFICATION PASSED!!
u29413@s001-n082:~/DevConFPGALab/unroll_ivdep$
```

Part F. Bank the accum_local Memory Structure

- _____ 1. Change the directory to ~/labs/lab3/banking by typing the following command in the terminal prompt.

```
cd ../banking
```

- _____ 2. For this step, more extensive changes to the code were necessary. So, the re-coding has been done for you.

The next optimization will use the numbanks attribute. Recall that banks are structures that have independent ports from the rest of the memory structure, but that only contain a portion of the contents. For example, if we created 2 banks, 1 bank would contain half of the data and the other bank would contain the other half of the data, each half could be read from independently.

The banks will be created using the lower index, and the numbanks attribute must be set to a power of 2.

There are 2 changes that were made to the code for this step:

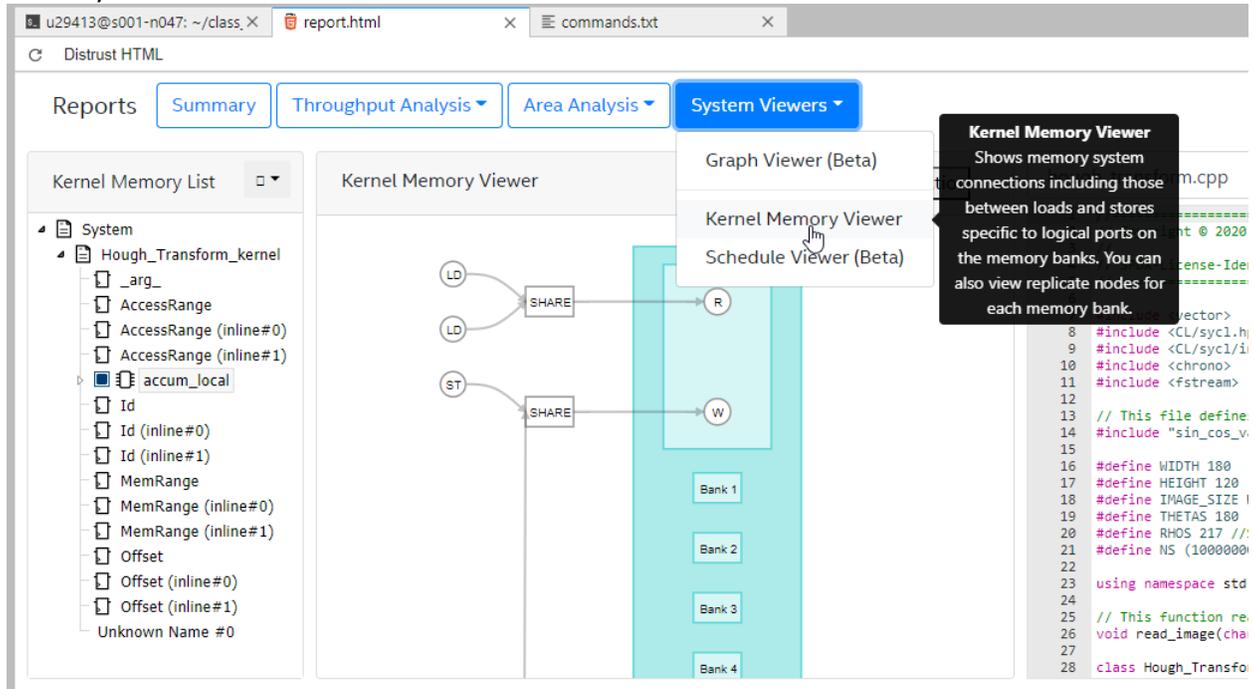
- Structure accum_local as a 2-dimensional array instead of a 1-dimensional array.
 - The lower dimension should be a power of 2 closest to 180, so 256
 - Everywhere you loop through the indices for accum_local will need a change
- Apply the numbanks attribute to the accum_local variable.

- ____ 3. Open the file `hough_transform.cpp` within the Jupyter Lab environment and search for “Call the kernel.” Observe where the banking has been declared using an attribute when `accum_local` is declared. Also observe that the memory needed to be made 2 dimensional in order to accomplish this, so it had implications for other code in the kernel.

```
96 //Call the kernel
97 cgh.single_task<class Hough_transform_kernel>([=]() {
98
99     [[intel FPGA::numbanks(256)]]
100     short accum_local[RHOS*2][256];
101
102     for (int i = 0; i < RHOS*2; i++) {
103         for (int j=0; j<THETAS; j++) {
104             accum_local[i][j] = 0;
105         }
106     }
107
108     for (uint y=0; y<HEIGHT; y++) {
109         for (uint x=0; x<WIDTH; x++){
110             unsigned short int increment = 0;
111             if (_pixels[(WIDTH*y)+x] != 0) {
112                 increment = 1;
113             } else {
114                 increment = 0;
115             }
116
117             #pragma unroll 32
118             [[intel FPGA::ivdep]]
119             for (int theta=0; theta<THETAS; theta++){
120                 int rho = x*_cos_table[theta] + y*_sin_table[theta];
121                 accum_local[rho+RHOS][theta] += increment;
122             }
123         }
124     }
125
126     for (int i = 0; i < RHOS*2; i++) {
127         for (int j=0; j<THETAS; j++) {
128             _accumulators[i*THETAS+j] = accum_local[i][j];
129         }
130     }
131
132 });
133
134 });
```

- ____ 4. Compile the code for emulation, and run the emulation executable until you achieve the VERIFICATION PASSED! statement.
- ____ 5. Compile the code to an object file and a static optimization report. This will take a few minutes.

- ___ 6. Open the optimization report.
- ___ 7. Navigate to the Memory Viewer section of the report. Select `accum_local`. Notice that the red is gone! This means there is no arbitration, and no more potential stalling when accessing that memory structure.



- ___ 8. The execution time of this final version of the kernel is about **0.00825 seconds** on an FPGA in the Intel DevCloud.

```

u29413@s001-n082:~/DevConFPGALab/banking$ ./hough_banking.out
Platform name: Intel(R) FPGA SDK for OpenCL(TM)
Device name: pac_a10 : Intel PAC Platform (pac_1fb00000)
Kernel execution time: 0.00825041 seconds
VERIFICATION PASSED!!
u29413@s001-n082:~/DevConFPGALab/banking$

```

- ___ 9. You have reached the end of the exercise, and the end of the class. Thank you so much for attending!

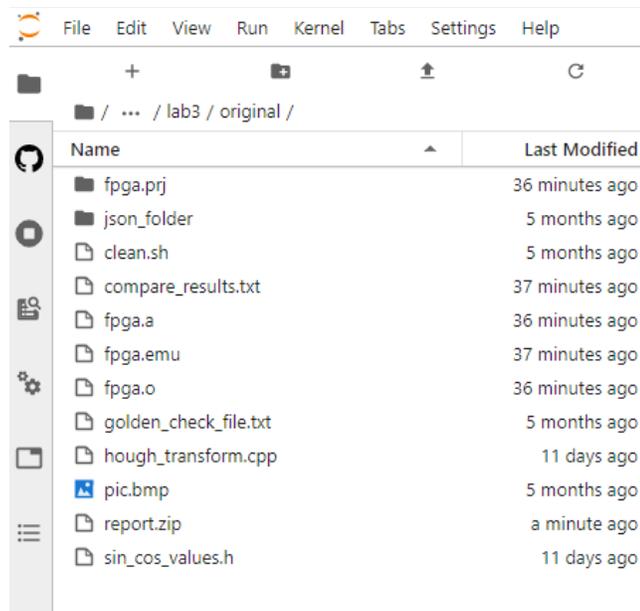
You have reached the end of the lab exercise. The next section is an appendix.

Appendix. How to Transfer Report Files to Your Local Computer

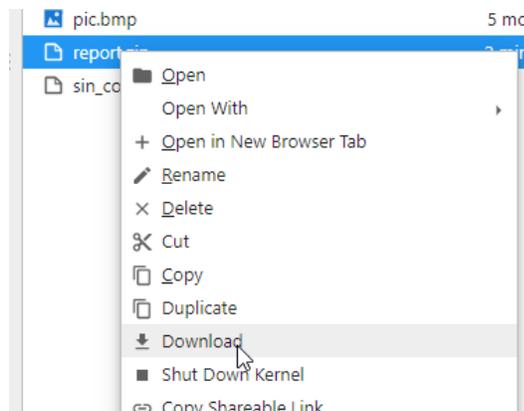
1. The original (Part C) subdirectory was used for these instructions. Replace with the subdirectory you are working from. In the terminal, navigate to the subdirectory you are working from. You will need to transfer the entire `fpga.prj/reports/subdirectory` to your local machine. The first step is to zip this directory into a zip file. Execute the following command (this command was done from `~/labs/lab3/original/`)

```
$ zip -r report.zip fpga.prj/reports
```

2. A file called `report.zip` will be created in the current directory. Navigate to the directory you are in inside of the terminal using the file browser on the left side of Jupyter, as shown in the screenshot below.



3. Right-click the `report.zip` file, and select Download.



4. Unzip the report.zip file on your local machine, navigate to fpga.prj/reports/ within the unzipped directory, and open the file report.html.

Report: fpga_bae426

File | C:/work/developer/oneAPI/class_instances/class_1013/labs_work/report/fpga.prj/reports/report.html#view1

Phoenix Buy CV G Family Phoenix Virtual Learning Animation Career Health SoC Griffin Learning Shutdown

Reports Summary Throughput Analysis Area Analysis System Viewers

Summary Content

- Compile Info
- Kernels Summary
- Clock Frequency Summary
- System Resource Utilization Summary
- Quartus Fitter Resource Utilization Summary
- Compile Estimated Kernel Resource Utilization S
- Warnings Summary

Summary

Compile Info

Project Name: fpga_bae426

Target Family, Device, Board: Arria 10, 10AX11552F45I25GES, intel_a10gx_pacpac_a10

SYCL Version: 20.3.0 Build 72

Quartus Version:

Reports Generated At: Tue Oct 13 07:12:27 2020

Kernels Summary

Name	Source Location	Kernel Type	Autorun	Workgroup Size	# Compute Units	Target Frequency (MHz)
Hough_Transform_kernel	:0	Single work-item	No	1,1,1	1	Not specified

Clock Frequency Summary

System Resource Utilization Summary

Quartus Fitter Resource Utilization Summary

hough_transfo

```

1 //=====
2 // Copyri
3 //
4 // SPDX-L
5 // =====
6
7 #include
8 #include
9 #include
10 #include
11 #include
12
13 // This f
14 #include
15
16 #define N
17 #define H
18 #define I
19 #define T
20 #define R
21 #define N
22
23 using nam
24
25 // This f
26 void read
27
28 class Hou
29
30 int main(
31
32 //Decla
33 char pi
34 short a
35
36 //Initi
37 std::fi
38

```

Details

Intel Corporation. All rights reserved.

Intel, the Intel logo, Altera, Arria, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

*Other names and brands may be claimed as the property of others.