ECE 3270

LAB 5 REPORT

June 13, 2022

Tim Driscoll
Clemson University
Department of Electrical and Computer Engineering
tdrisco@clemson.edu

Abstract

The main goal of this lab is further develop skills regarding how the FPGAs compute. This main goal also included a further introduction into OpenCL and different computing optimizations. This goal was achieved in the lab through the implementation of 1 dimensional convolution. The 1D convolution was first implemented through a basic C program. This C program was then used to model an OpenCL implementation which was then emulated to test the results.

1 Introduction

Lab five was broken up into two main parts that ultimately built on each other. The first part of the lab was to design and time the implementation of a C program to complete 1D convolution. The second part of the lab was to implement this process of convolution in an OpenCL program. The process of convolution involves iteratively taking the the dot products of two vectors that are at different offsets. This process can be completed by having one vector held stationary and then having the other vector which is typically referred to as the kernel shifted across the vector. The dot product is taken at each sequential shift. The process of convolution creates a resulting vector that is equal to the sum of the length of the two vectors minus 1. After understanding how the convolution could be completed it was coded in a basic C program. The basic C program was then used to create the kernel of the OpenCL program. After the kernel was created the remaining steps could be implemented to finish the OpenCL implementation. The specific design steps will be discussed in section 2 below.

2 Design

2.1 C Program Implementation

The first part of lab five was to create a C program to perform one dimensional convolution. I performed this part of the lab in a modular fashion in order to allow me to easily translate the modular functions into and OpenCL function. Two functions were designed in order to implement the convolution. One function was designed to perform the convolution process and produce a result vector. The second function was designed to read in input data from a command line specified binary file. These functions will be further discussed in sections 2.1.1 and 2.1.2 respectively. The overall flow of the C program involves declaring the kernel as specified in the lab description then opening the file to read in the input vector data. The binary function was structured to store the length of the vector as the first four bytes, then the vector data was represented by the remaining bits. The read binary function was then called in order to obtain the length of the input vector and the input vector data itself. This newly obtained data was then used to call the convolution function which calculated a result vector. The convolution function was timed in order to record data on how long it took to calculate the results. After the results of the convolution were obtained a formatted output was displayed to the terminal to display the results.

2.1.1 Convolution Function

To perform the actual convolution calculation a simple C function was written. This function used a kernel vector, input vector and their respective lengths as inputs. The lengths of the two inputs were then summed and subtracted by one in order to determine the length of the resulting convolution vector. A for loop was then designed to run from zero to the length of the result vector, allowing for a dot product calculation to be

performed for each successive index of the result vector. Within this for loop there was a nested for loop to run through the length of the kernel allowing for the dot product calculation to be performed. There is also a nested if statement within the nested for loop to check for the edge cases. It is important to check for these edge cases because there are times in the convolution calculation where only portions of the kernel overlap with the input vector. The if statement ensures that no multiplication occurs when there isn't an overlap between the two vectors. After the outer for loop finishes running the calculations for the convolution are completed.

2.1.2 Read Binary Function

In order to read in the input vector data allowing for an ease of testing a simple function to read from a binary file was created. This function used a file pointer to the previously opened binary file, an integer variable and the input vector float pointer as inputs. The integer variable input (length) was used to store the first four bytes from the binary file, which specifically represents the length of the input vector. After the length of the vector is obtained the memory for the input vector could be allocated. This allocation occurred using a call to malloc where the length obtained was multiplied by the size of a float. After the allocation occurred the data for the input vector could be read in from the binary file using fread. Once the fread is completed the necessary would be stored in the allocated vector to later be used in the convolution calculation.

2.1.3 C Program Testing

In order to test the C program a series of different binary file inputs were used. These binary file inputs contained data of different basic test input vectors. The calculations were then performed by the program and outputted to the terminal. Since the input vectors were relatively small and simple the results could be verified through hand calculations. The first test can be viewed by figure 1 which receives a simple input vector with a length of four. Thus it can be concluded the resulting convolution vector will have a length of six as is shown. After performing the calculations by hand it can be confirmed that the results are correct. The second test can be viewed by figure 2, which receives a larger input vector of length 12. Thus it can be concluded that the resulting convolution vector will have a length of 14 as is shown. After performing the calculations by hand it can be confirmed that the results are also correct. These results will also be used to further verify the results of the OpenCL implementation. It is also important to note that the timing data for the convolution was recorded. Although the data was recorded it has less significant relevance when being compared to the OpenCL implementation because the times will be very similar. The reason the OpenCL implementation doesn't show a large speed up is because it is merely an emulation and is not being run on the FPGA.

2.2 OpenCL Program Implementation

The next part of the lab was to use OpenCL to complete convolution. In order to start this section of the lab the original OpenCL file structure was copied from lab 2.This original file structure was then cleaned out and renamed to represent what was needed for the current project. All of the files that were used for libraries and VHDL could be removed from the folder. After these files were removed the remaining files were renamed to represent lab five and convolution. Next a new run.sh file was downloaded from canvas and added to the emulation folder replacing the previously existing run.sh file. Once the initial setup of the OpenCL file structure had been completed edits were made to the necessary files in order to complete the convolution implementation. Minor edits were made to the Makefile in the main aocl_convolution folder, the Makefile in the emulation folder and the run.sh file in the emulation folder. These simple edits included changing the target and project names to reflect the names used in lab 5 instead of the names from lab 2. It was also important to add the necessary commands to execute the OpenCL program in the run.sh file, this included adding the binary input file name as a command line argument. After these minor changes were accounted for the major changes could be made to the kernel and the host code convolution_cl.cl and main.cpp respectively. These files will be further discussed in sections 2.2.1 and 2.2.2 respectively. Once these two files were completely edited the OpenCL program could be made and emulated to verify the results.

2.2.1 OpenCL Kernel Implementation

The first step in building an OpenCL program is to write the kernel. After updating the OpenCL file structure and making some basic adjustments to three of the files the kernel could be written. To start writing the kernel the inputs and kernel header needed to be written. The inputs to the kernel would include two constant float pointers, two integers and a global float pointer. The two constant float pointers represented the the kernel vector and input vector inputs. Both of these vectors could be declared as constants because they only needed to be read from. Declaring them as constants would result in a slightly faster execution due to more aggressive caching. Next, the two integer inputs were used to simply represent the lengths of the kernel and input vector. Lastly the global float pointer was used to represent the result vector, which needed to be both read and written to thus representing the reason it was declared globally. Writing the body of the kernel was very simple and was completed by copying the convolution function described in section 2.1.1. A minor change was made to this original function which represented how the kernel would be implemented in a parallel fashion. To make this change the outer for loop was removed from the original function and the counter variable used in this outer for loop was set to the value returned by get_global_id(0). The outer for loop in the original C implementation is what ran through every point in the result vector in order to make the calculations. Thus by replacing this for loop and its counter variable with get_global_id(0) it allows for every point in the result vector to be calculated at the same time. When implemented on an FPGA this would result in a great speed up when

compared to the C program. Since the parallel OpenCL was implemented this way it could complete the entire convolution process in the time that the C program completed one loop iteration. Again it is important to note this speed up is not reflected by timing results reported in the test cases because the OpenCL program could only be run as an emulation. The speed up would become apparent when the the OpenCL program is run on the FPGA.

2.2.2 OpenCL Host Code Implementation

The last step to completing the OpenCL program was to write the host code represented by the file main.cpp. Writing the host code contains multiple sub-steps broken into four functions, some of the sub-steps were already completed and reused from the code created for lab two. The first step in writing the host code was implementing the init_problem() function. This function simply mimicked the read binary function implemented in the C program as described in section 2.1.2. This function was simply used to read in the input data from a binary file and allocate the memory needed for the result vector. The only other data needed was the kernel and its length which was globally defined. The second step in creating the host code was to write the platform layer. The platform layer is where some of the initial OpenCL setup occurs and is very close to being the same from one project to the next. The platform layer was represented the first portion of the init_opencl() function and set up the device. The next step in the design was to create and build the program which was also copied directly from lab two and only required a change to the name of the file being built. After building the program the kernel was created which was also copied directly from lab two but changed to represent the name of the newly created kernel (convolution_kernel). After this initial setup the steps became more unique to the lab 5 implementation. Still working in the init_opencl() the next step was to allocate and transfer buffers onto the device. This step involved using the clCreateBuffer function three times in order to create a buffer for the kernel, input and result vectors. After creating the buffers both the kernel and input vectors were transferred to the device using the clEnqueueWriteBuffer function. The third and final major step in creating the host code was implementing the run() function. This function was first used to set all kernel arguments in the proper order using the clSetKernelArg function. Next the global work size was set to represent the length of the convolution result vector and the kernel was launched using the clEnqueueNDRangeKernel function. It was important to make sure the kernel had an event wait list set to the transferring of the buffers to the queue. This was needed to make sure the needed data was in the queue prior to launching the kernel. After the kernel was launched the convolution would be performed and the result buffer needed to be transferred back from the device. This process was completed by using the clEnqueueReadBuffer function. It was again important to set the event wait list to the launching of the kernel event. Having this event wait list would ensure that the result buffer didn't get read before the kernel was launched. Lastly the formatted output could be printed to the terminal in the run function. This formatted output was designed to match that of the C program. In main.cpp the last step that was taken was implementing the cleanup() function which was used to free all the resources that were

allocated.

2.2.3 OpenCL Program Testing

In order to test the OpenCL program the same binary files used to test the C program were used. By using the same files the outputs of the previously verified C program could be used to verify the OpenCL program. The two test outputs can be viewed by figures 3 and 4. When comparing them to 1 and 2 respectively is can be observed that the the C program and OpenCL program produce the same results. These results were also verified by hand as noted in section 2.1.3 above. Through these tests it could be confirmed that the OpenCL program produces the correct results.

2.3 Helper C Program

In order to test both the C and OpenCL program binary files containing the input vector needed to be written. The most efficient way to create these binary input files was to write a very basic C program to write the binary files in the desired format. This C program had an overall flow of defining a structure the contained a single integer to represent the length of the vector. The structure then contained a number of floats matching the length and storing the data. A hard coded binary file was then opened to be written to. Lastly a single fwrite command was used to write the whole structure to the binary file. In order to create different binary file the data in the structure could be edited and the file that is open would be changed. This simple function allows for the creation of binary test files and with some minor changes many different files could be made.

2.4 Improving Kernel Performance

There are many different optimization techniques that can be used in order to improve the performance of the OpenCL kernel. One simple method that can be used if the maximum size of a work group is known would be specify and restrict this maximum size. When the maximum work group size is restricted it allows the compiler to perform more aggressive optimization when matching the kernel to the FPGA hardware. Specifically it well ensure when the kernel is matched to the hardware there will not be any excess in logic resources used. Another optimization that can be performed in order to improve the kernel performance would be to have the compiler generate multiple compute units if the FPGA hardware has enough resources. If multiple compute units are generated then each compute unit would be implemented as an individual pipe line. The compiler would then be able to distribute the work groups across the different compute units. This would result in a large performance increase if the FPGA had the resources. When running on an actual FPGA there are many different ways to increase the speed and performance of the kernel. Some of the other optimization methods include kernel vectorization and memory management. The ability to optimize a kernel's performance when running on

and actual FPGA will depend on the task being accomplished by the kernel and the resources available on the FPGA.

3 Conclusion

The main goal of this lab was to gain further knowledge of how FPGAs compute, gain exposure to OpenCL and different optimization techniques. This goal was achieved through the development of both a C program and an OpenCL program to perform one dimensional convolution. The C program was used to verify the results of a convolution algorithm that could be later used to model the OpenCL program. The creation of the OpenCL program provided a hands on experience to working with OpenCL and developing an entire program. In the implementation of the OpenCL it was important to understand how the FPGA computes especially in the design of the parallel OpenCL kernel. Although the speed of from the C program to the OpenCL program wasn't apparent due to limited testing abilities, it can be inferred when looking at how the FPGA would compute. Through the creation and exploration of the two programs all of the main goals were observed and achieved.

The lab provided a great learning experience and allowed for an application of many of the topics that were discussed during lecture. It was especially important to have a thorough understanding of how OpenCL works and all he different components and steps that went into creating a functional program. Some of the main lessons that were learned throughout this lab included how to implement a parallel kernel in OpenCL and the general steps that were needed to write a complete program in OpenCL. This lab continued to reinforce the benefits of creating a modular design. I was able to easily translate the modular C program into the code need for the OpenCL program. Throughout this lab the only errors that occurred were minimal syntax bugs that were easily fixed and a memory allocation error that was corrupting the data in the result vector. Overall the lab provided a great learning experience through the process of implementing an OpenCL program to perform convolution.

4 References

Altera SDK For OpenCL Best Practices Guide. San Jose: Altera.

M.Smith "Digital Computer Design" Clemson University Holcombe Department of Electrical and Computer Engineering, January 2020

M.Smith "ECE3270 Digital System Design Lab 5: OpenCL Convolution Design" Clemson University Holcombe Department of Electrical and Computer Engineering, January 2020

W. Daily, R. Curtis Harting and T. Aamodt, Digital Design Using VHDL a systems approach. Cambridge University Press

5 Appendix

```
tdrisco@riggs309lab02:~/Lab5_ECE3270$ gcc -Wall lab5_convolution.c
tdrisco@riggs309lab02:~/Lab5_ECE3270$ ./a.out bintest.bin

Convolution Output:

--------------------------------
Kernel:
0.333333  0.500000  0.166667
--------------------------------
Input Vector:
1.000000  2.000000  3.000000  4.000000
--------------------------------
Convolution Vector:
0.333333  1.166667  2.166667  3.166667  2.500000  0.666667
--------------------------------
Convolution Time: 0.000007

tdrisco@riggs309lab02:~/Lab5_ECE3270$
```

Figure 1: C Program Test 1

```
tdrisco@riggs309lab02:~/Lab5_ECE3270$ gcc -Wall lab5_convolution.c
tdrisco@riggs309lab02:~/Lab5_ECE3270$ ./a.out bintest2.bin

Convolution Output:

--------------------------------
Kernel:
0.333333  0.500000  0.166667
--------------------------------
Input Vector:
1.000000  2.000000  3.000000  4.000000  5.000000  6.000000  7.000000  8.000000  9.000000  10.000000  11.000000  12.000000
--------------------------------
Convolution Vector:
0.333333  1.166667  2.166667  3.166667  4.166667  5.166667  6.166667  7.166667  8.166667  9.166667  10.166667  11.166667  7.833333  2.000000
--------------------------------
Convolution Time: 0.000007

tdrisco@riggs309lab02:~/Lab5_ECE3270$
```

Figure 2: C Program Test 2

```
+ ./convolution_cl bintest.bin
Error: kernel argument info is not available

Convolution Output:

--------------------------------
Kernel:
0.333333  0.500000  0.166667
--------------------------------
Input Vector:
1.000000  2.000000  3.000000  4.000000
--------------------------------
Convolution Vector:
0.333333  1.166667  2.166667  3.166667  2.500000  0.666667
--------------------------------
Convolution Time: 0.000820
```

Figure 3: OpenCL Program Test 1

```
+ ./convolution_cl bintest2.bin
Error: kernel argument info is not available

Convolution Output:

--------------------------------
Kernel:
0.333333  0.500000  0.166667
--------------------------------
Input Vector:
1.000000  2.000000  3.000000  4.000000  5.000000  6.000000  7.000000  8.000000  9.000000  10.000000  11.000000  12.000000
--------------------------------
Convolution Vector:
0.333333  1.166667  2.166667  3.166667  4.166667  5.166667  6.166667  7.166667  8.166667  9.166667  10.166667  11.166667  7.833333  2.000000
--------------------------------
Convolution Time: 0.001107
```

Figure 4: OpenCL Program Test 2