**NCL lab 보고서**

**Maro Han**

**Week1:**

**Main objectives:**

**Learning about the basics of the FPGA using slides from MEC seminar:**

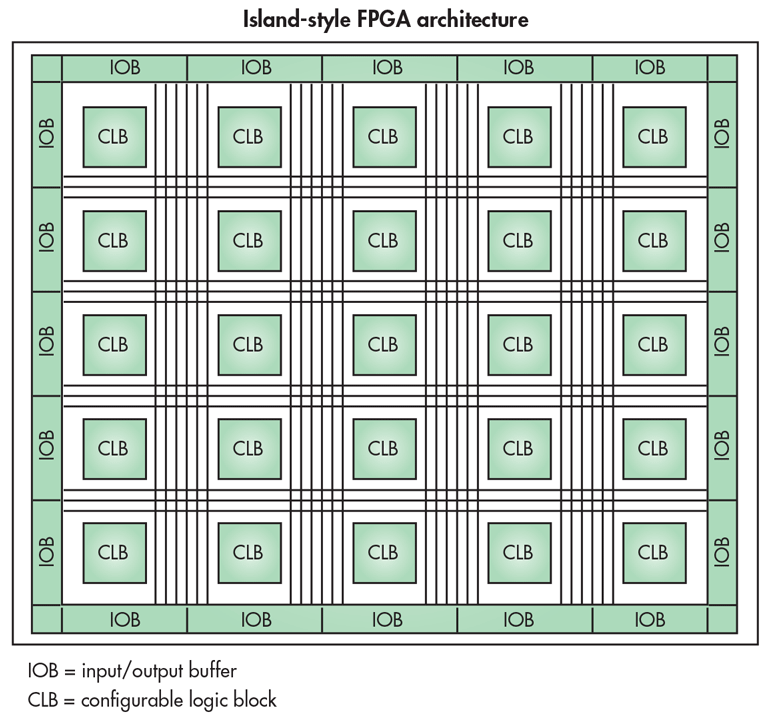
Learned about the TestBed environment

We have three FPGAs. Xilinx kcu1500 Development kit, Future Systems DSCo-ku060, Intel Arria10 Development kit. Xilinx FPGA supports the OpenCL environment and HDL development environment while the other two FPGAs are researching so they can support various DL related open source and user guide development environment. In our experiment we decided use the Xilinx FPGA



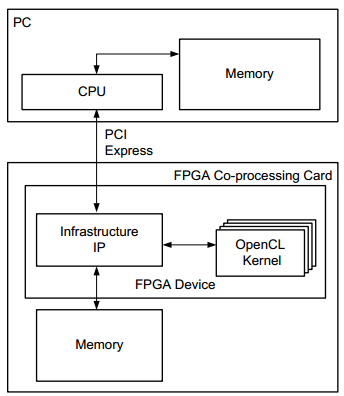
After that I learned about the general structure of FPGAs. Simply put, FPGAs consists of I/O blocks, configurable logic blocks (CLB) and on chip memory in the form of Block RAM. The CLBs on FPGA can be designed flexibly allowing FPGAs to be flexible and give high throughput.

A simplified diagram of a typical FPGA device:



Then I learned about the FPGA memory architecture.

The basic memory structure is similar to most other processors that work in conjunction with a CPU. It is connected with the CPU with a PCI express and CPU provides memory that is on the CPU chip. In the FPGA co-processing card, there is on-chip memory. There is off-chip memory unit connected to the FPGA, and because of the small BW between on-chip and off-chip memory, it is critical for us to maximize data-reuse in the FPGA.

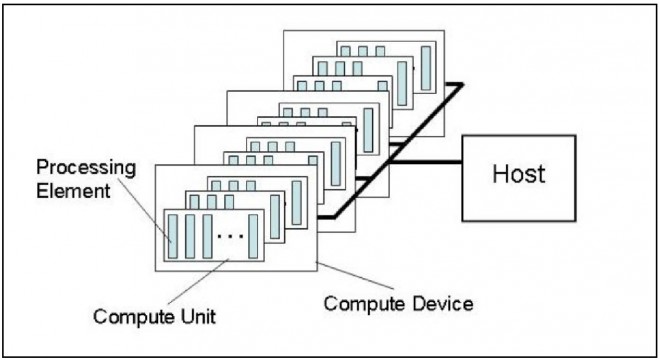


On-chip memory

Off-chip memory

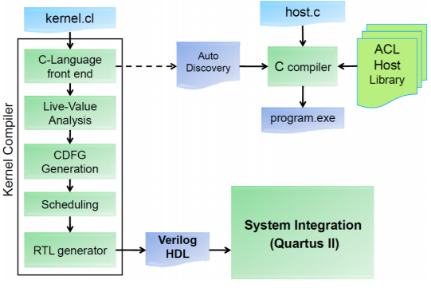
Then we moved on, and learned about how the OpenCL allows the usage of FPGA to program.

OpenCL is a sort of programming interface that allows the abstraction of hardware devices. It provides portability between different hardware. For example, OpenCL view CPU, GPU and FPGAs the same. A device with one host and a series of compute units that has multiple processing elements (processor).

OpenCL Logical view of hardware devices:

OpenCL platform model views devices in built in kernels and compute units, which are connected to the CPU and functions accordingly. The open CL memory model divides the memory section into four. Host memory, global/constant memory, local memory, private memory. Host memory and global/constant memory is in the CPU and is only accessible by the host. In the FPGA we are planning on using, Host memory is stored in DDR SDRAM. Local memory is shared by processing elements in one compute unit, and private memory is only accessible by one processing element. Local memory is stored on-chip memory and is stored in BRAM. Private memory is also stored on-chip and is stored in the form of registers.

OpenCL FPGA framework:



NDRange is an index space that serves to store work items. In our FPGA, scheduling is done in unit of a work group so we have to divide the work in multiple work groups. Hence, the number of work groups are defined by global size (the total number of work items) and local size (which is the number of work items in a group).

SDAceel takes care of compilation in our machine. During compilation, SDAceel exploits conceptual loops using pipelining and vectorising. For example, code written in kernel is surrounded by three nested loops to traverse the entire work-group size. To exploit the loop, the compiler only synthesizes one operator of the loop. The threads use this synthesized operator to efficiently run the loop.

**Learn about the OpenCL language and how to utilize it to do FPGA programming**

Reading the document Altera SDK for OpenCL guide provided by ALTERA. Through the guide I was able to obtain knowledge on the programming flow of FPGA programming. How kernel interacts with the host program. How to compile kernels, how to write efficient kernel and host programs that maximize memory and performance.

On top of that I learned command and macros specific to FPGA programming that instruct what, how many FPGAs to use how to increase memory efficiency and chip performance. Moreover, I learned FPGA specific Kernal pragmas and attributes such as max\_work\_group\_size or unroll, which we can use to specify how computation in our program is done again manually increasing memory efficiency and program performance.

In addition, I read the “The OpenCL Specification 1.2” which taught be about functions used to use in the host code. These include clCreateCommandQueue, clBuildProgramFromBinary and many others used to create context, platform, deviceid, commandqueue, and program.

**Week2:**

**Reading and analyzing the vector\_add kernel and host code provided by Intel.**

The kernel code is relatively simple. It adds the vector A to vector B and stores its results into vector C. It draws its value from global memory located at the CPU.

The host code sets up the hardware and software so the kernel code can be executed in the correct fashion. We can parse the host code into four parts each segmented as functions blocks. Initialization of parameters, initializing problem, running and cleaning up.

Before diving into these functions, basic variables are defined. These include platform, num\_devices, device and more. Then, using the option function the programmer synthetically analyzes the command line options to read if there are additional options input by the user.

Then, we initialize the parameters.

To Initialize the parameters, the following functions are executed (bold words are variables).

1. Set the working directory to the one that contains the executable.
2. Find the platform in which the search string “Intel(R) FPGA SDK for OpenCL(TM)” and store it in **platform**.
3. List all the devices we can use and store it in **device**.
4. Create context and store it in **context**.
5. Find path to aocx file and store it in **binary\_file**.
6. Using **binary\_file** variable create a program from the binary and store it in **program**.
7. Build the program using **program**.
8. Define per-device objects (queue, kernel, n\_per\_device, inputs and outputs)
9. Create per-device objects and store it in the aforementioned objects accordingly.
   1. For n\_per\_device (number of calculated objects per device), we spread out the remainder equally among devices after dividing number of instructions by number of devices.
10. Finally, we check if everything went according to plan and move on to the next step.

As we can see, this phase of the code prepares the running of our program. It is critical to appreciate that this step of the function is reusable in other codes that serves to run other kernel codes that and is not limited to only serve for vector\_add kernel code. With little modification we can use parts of this code to train a deep learning program (which is usually just a lot of simple calculations) and use it in my research later.

The next segment of the code initializes the problem. This part of the code has the following structure:

1. Initializes input randomly using the rand\_float() function.
2. Creates a **ref\_output** object so we can use it to compare the accuracy of our hardware.
3. If we are using SVM program, we need to map inputs into SVMMaps and unmap them when we are done with the computation of our reference output.

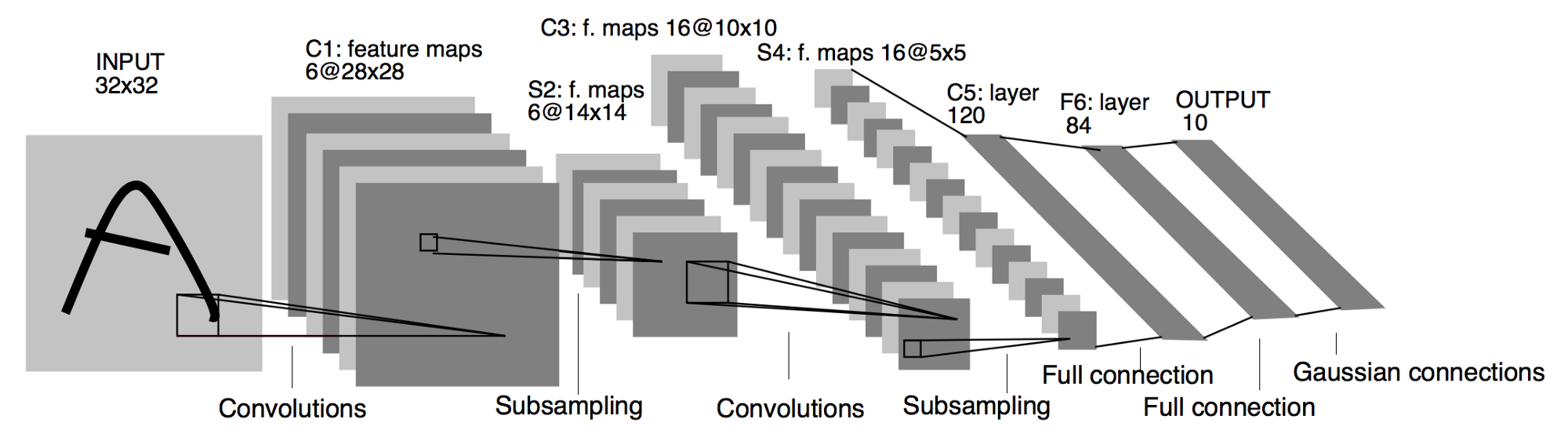
This segment of the code is also usable later in our deep learning example. However, we must note that producing predicted values for the DNN code is more complex than predicting the result of a vector addition. Hence, the prediction area must need extensive rebuilding in order to be used again.

**Week 3:**

**Porting opencl Lenet-5 CNN code to FPGA:**

We imported the Lenet-5 CNN code and modified it to fit our environment. The kernel code (.cl code) did not need to be changed in order to be adapted. However, the host code did have to get modified. The below figure is the standard Lenet-5 model. In our implementation, we lose the F6 layer and have a 6-stage DNN.

The Kernel code has three parts. Convolution, Pooling, Full connection and padding layer. With these four parts, the code emulates the following CNN layers:



Our host code starts by initializing variables used in the computation. Then it reads MNIST data and MNIST label, which will later tell us the accuracy of our network. The first four 32 bit integers of the MNIST dataset specifies the data type (magic number), number of images, number of rows and number of columns. The read MNIST function in the code parses the first four integers then stores the pixel data in an array and returns the array.

After this, the main function executes the init\_cl function. The init cl function carries out the following functions:

1. Identifies the platform using the **clGetPlatformID** function. The **clGetPlatformID** function takes cl\_platform\_id type variable as input and stores the platform-id used to the input cl\_platform\_id type variable. Platform id can be used to identify a specific OpenCL platform, which is a collection of devices managed by OpenCL framework, and is used to share resources and execute kernels on devices.
2. Then using the **clGetDeviceID** Function, I extracted the id of the device we were going to use. The Function GetDeviceID function takes the platform name, and specifies what type of device we are looking for. Then stores the device ID in a cl\_device\_id type variable. Here, the original code had the option CL\_DEVICE\_TYPE\_CPU, which was not suitable for our case. Hence, I changed the options to CL\_DEVICE\_TYPE\_ALL so it could find the FPGA device we are planning on using.
3. Using this device ID, we created the context for our program using the **clCreateContext** function. Context in OpenCL contains a set of devices, memory accessible to devices, memory accessible to devices and one or more command-queues. Context is used to schedule execution of kernels or operations on memory objects.
4. Using the context and **clCreateCommandQueue** function we created we created the command queue for our program. Command queue basically contains a list of commands that are executed during the program sequentially.
5. Creates board binary file using the aocx file we compiled before and **getBoardBinaryFile** function. **GetBoardBinaryFile** function takes device id and name of aocx file as arguments. Then outputs the binary file of the board.
6. Using the created board binary file, the context, device id and the function **createProgramFromBinary**, we create a program object. Program object inhibits four information. One, reference to an associated context. Two, program source or binary. Three, the latest successfully built program executable, list of devices for which the program executable is built on, the build options used, and a build log. Finally, the number of kernel objects that is attached. This section was actually done with **createProgramFromSource** in the original code. However, in our environment there were no compiler to compile the created program from source. Hence, I had to change this section of the code so the program was executed using the already compiled binary.
7. Using the **clBuildProgram** function we compiled and linked our program executable to the device. We used the program object created previously with **createProgramFromBinary** function.
8. Then using the built program, we create the kernel for convolution, polling and full connection layer using the **clCreateKernel** function. The kernel is created from the kernel code written in the .cl file and the built in program. Kernel is a function written in a program and executed by the OpenCL device.
9. After this we finally start preparing the neurons. Using **prepareCNeurons** function we prepared the neurons for the convolutional layer and the last two full connection layers. The **prepareCNeurons** function intakes the number of neurons, number of kernels, kernel width, output image size, input image size, weights and returns a CNeuron structure. The CNeuron structure contains the context, command queue, three kernels created before, kernele width, number of neurons, number of kernels, output image size, and input image size.
10. After establishing the neurons, we initialized the layers. Each layer is defined with a child class of the parent class Layer. For example, the initial layer is defined by the subclass ILayer. The parent class Layer contains a feature map, and the child class ILayer has a feature buffer intended to store result from the initial layer of our code. CLayer – which is a grandchild class of Layer – can store neurons that were created in the previous section of our code. PLayer – also a grandchild class of Layer – contains context, commandqueue, kernel for pooling, number of neurons, image size and feature buf. The pooling layer Enqueues its command by setting kernel arguments 0,1,2,3 as input map, featurebuf, input width, and image size. Then Equeues it into NDRange Kernel using the commandQueue.
11. This ends the preparation stage for our code. Now we start the computation.
12. Our program checks the accuracy of our code by having an integer variable flag which increases every time our result matched the expected result. The code computes all data though the six layers. Then we print out the time taken for the computation, time taken to identify each MNIST data.

**Week 4:**

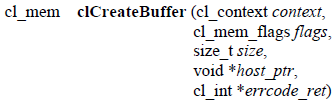
**Aligning memory so DMA can be used for memory transfer, ultimately achieving performance improvement**

Previously, we were not using DMA to transfer memory between host and local memory. If we can use DMA to enhance memory transfer, we can reduce the overhead cost that is paid every time memory is transferred in small chunks. In our system the code will automatically attempt to use DMA to transfer memory as long as the transferred chunks are 64 byte aligned. If not, it will show error messages such as this:



We can see that the host ptr is misaligned and is thus causing the problem. In order to align the buffer to 64 bytes we must tackle the buffer’s creation stage.

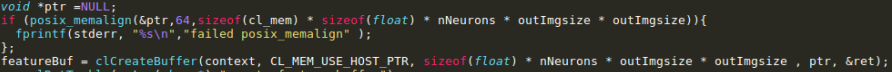
In our code, featureBuf buffer stores the input and output of convolution, pooling and Fconnect layer. This buffer is created using clCreateBuffer function:



Before optimization, our code looked like this:



If we code like this, featureBuf is not aligned to 64 bytes. So I changed to code to:

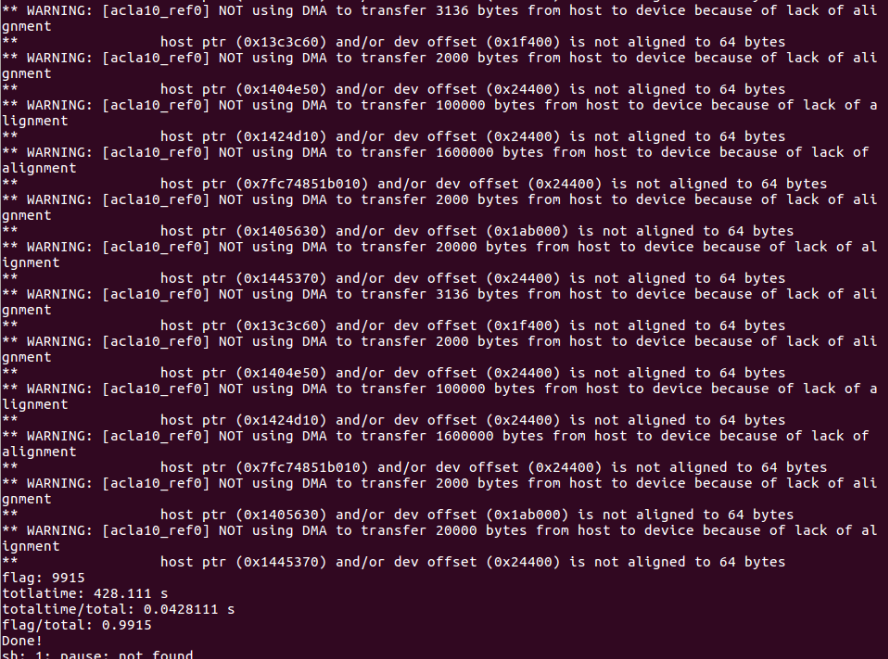


Using posix\_memalign we made the pointer ptr be aligned to 64bytes. Then I used the CL\_MEM\_USE\_HOST\_PTR flag in clCreateBuffer function. This flag forces featureBuf to use memory space used by the pointer ptr, which is aligned to 64 bytes.

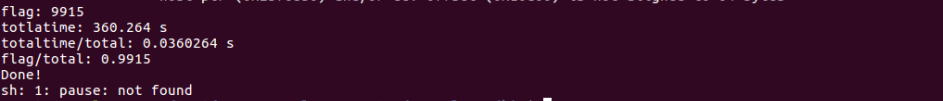
Using this design, we were able to reduce the amount of misalign errors.

**Results:**

Without DMA optimization:



With featureBuf DMA optimization:



|  |  |  |
| --- | --- | --- |
| **Lenet-5 CNN on intelFPGA\_pro board** | | |
|  | **Time taken per image** | **Accuracy** |
| **No DMA optimization** | 0.0428s | 99.15% |
| **DMA optimization for host buffer** | 0.0360s | 99.15% |

Using this method, we were able to reduce the time taken to process one image file to 0.0360s which is 18% faster compared to the time taken when DMA was not optimized.

**Discussion**:

Our CNN model resulted in satisfactory numbers. Unfortunately, due to the lack of resources we were not able to compare its FPGA’s functionality with other devices such as a GPU or a CPU. However, with the results we have obtained, we can conclude that the use of porting the FPGA for CNN was successful as it was able to reach a 99.15% accuracy and could process each image in 0.0360s.

There could be a lot of improvements that could be made to make the system more efficient in many ways. For example, there still are more buffers that should be aligned to 64bytes. This will allow the memory transfers to be more efficient, increasing the overall performance of this machine.

Moreover, the structures used in the computation could be packed more efficiently using the packed command. This could make the memory usage more efficient, helping us when the number of samples increases further. Additionally, we should test our code with different number of samples and different type of samples to see if it works correctly in different contexts.