ECE 1373 HLS: DIGITAL DESIGN OF SYSTEMS ON CHIP

Convolutional Neural Networks for Image Recognition on FPGA's

Members:

Arjun Gandhi

Mahmud KhalafAlla

Venkatesh.R.Mahadevan

Table of Contents

List of Figures

List of Figures (Cont.)

1. Introduction

Neural networks are information processing paradigms that are inspired by the way biological systems such as the brain process information. They are composed of a large number of highly interconnected elements (called neurons) that work in unison to solve specific problems. Since the brain is composed of millions of neurons, one would naturally conclude that the parallel processing power of the same would be enormous. However, if we were to compare the brain's neurons with individual cores in a parallel computing cluster, we would find that the speed at which neurons could compute would be orders of magnitude slower than their hardware based counterparts. The question then arises as to how the brain is able to achieve a high throughput in terms of processing information in the form of signals coming from various mediums, and how once could incorporate it to make computers even faster.

Artificial neural networks (or ANN's) are an example of how the behaviour of the brain can be modelled in software. Their function is similar to the adjustment of synaptic connections in the human brain. These adjustments happen millions of times a second, which is what bestows our brain with its immense processing power. In addition, extensive research done in the scientific community has shown that different parts of our brain perform different functions. Their response to stimuli is often dictated by feedback and error correction. This idea of response via feedback gives rise to an ANN paradigm known as Convolutional Neural Networks (or CNN's). CNN's are a type of feed-forward ANN's that comprise of individual neurons that are tiled so as to respond to overlapping regions in the visual field. These are variations of multi-layer perceptrons, which were thought to be the basic processing units for perception in the early days of neural networks. They exploit spatially-local correlation through the enforcement of local connectivity patterns between neurons of adjacent layers, Simply put, the outputs of neurons from previous layers become inputs to the neurons in the subsequent layers, The output is determined by the activation functions associated with each layer. A simplified diagram of the same is given below

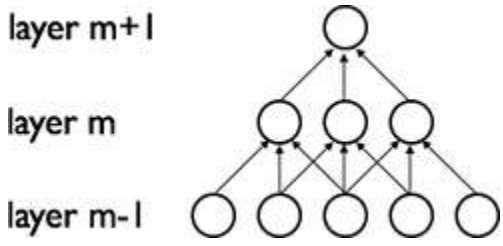

Fig 1.1: Simplified model of CNN

CNN's consist of various layers, each with its own unique processing abilities. The nature of layers implemented depends on the type of network used. The main goal of this project was to

design a CNN that could be used for image and speech recognition on FGPA's. This was designed based on the Caffe implementation [1], which consists of various layers, which can be interconnected in a manner suitable with the network being implemented. These layers, often referred to as vision layers, usually take images as input and produce other images as output. A typical "image" in the real-world may have one color channel (c=1), as in a grayscale image, or three color channels (c=3) as in an RGB (red, green, blue) image. But in this context, the distinguishing characteristic of an image is its spatial structure: usually an image has some non-trivial height h>1 and width w>1. This 2D geometry naturally lends itself to certain decisions about how to process the input. In particular, most of the vision layers work by applying a particular operation to some region of the input to produce a corresponding region of the output. In contrast, other layers (with few exceptions) ignore the spatial structure of the input, effectively treating it as "one big vector" with dimension *chw*.

1. Convolution: The Convolution layer convolves the input image with a set of learnable filters, each producing one feature map in the output image. This layer takes in the following parameters:

- **num_output (c_o):** the number of filters
- **kernel_size (or kernel_h and kernel_w)**: specifies height and width of each filter
- **weight_filler** [default type: 'constant' value: 0]

Optional parameters include:

- **bias_term** [default true]: specifies whether to learn and apply a set of additive biases to the filter outputs
- **pad (or pad_h and pad_w)** [default 0]: specifies the number of pixels to (implicitly) add to each side of the input
- **stride (or stride_h and stride_w)** [default 1]: specifies the intervals at which to apply the filters to the input
- **group (g)** [default 1]: If g > 1, we restrict the connectivity of each filter to a subset of the input. Specifically, the input and output channels are separated into g groups, and the ith output group channels will be only connected to the ith input group channels.

Thus if the input layer has dimensions n * c_i * h_i * w_i, the output would have dimensions as n * c_o * h_o * w_o, where h_o = (h_i + 2 * pad_h - kernel_h) / stride_h + 1 and w_o likewise.

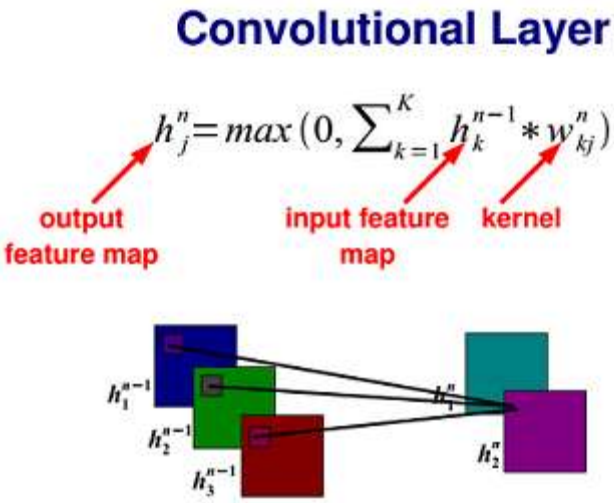

Fig 1.2 Convolutional layer

2. Pooling: The Pooling layer convolves the input image with a set of learnable filters and then computes the maximum or the average of the values in the region encompassed by the filters, each producing one feature map in the output image. This layer takes in the following parameters:

- **kernel_size (or kernel_h and kernel_w):** specifies height and width of each filter

Optional parameters include:

- **pool** [default MAX]: the pooling method. Currently MAX, AVE, or STOCHASTIC
- **pad (or pad_h and pad_w)** [default 0]: specifies the number of pixels to (implicitly) add to each side of the input
- **stride (or stride_h and stride_w)** [default 1]: specifies the intervals at which to apply the filters to the input

Thus if the input dimensions are n * c * h_i * w_i, the output would have dimensions n * c * h_o * w_o, where h_o and w_o are computed in the same way as convolution.

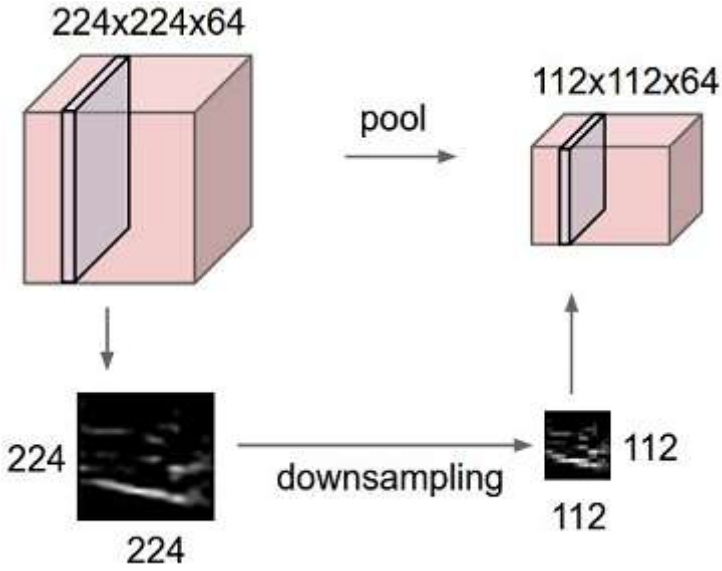

Fig 1.3 Pooling layer

3. Local Response Normalization (LRN): The local response normalization layer performs a kind of "*lateral inhibition*" by normalizing over local input regions. In *ACROSS_CHANNELS* mode, the local regions extend across nearby channels, but have no spatial extent (i.e., they have shape local_size x 1 x 1). In *WITHIN_CHANNEL* mode, the local regions extend spatially, but are in separate channels (i.e., they have shape 1 x local_size x local_size). Each input value is divided by $(1+(\alpha/n)\sum ix2i)\beta$, where n is the size of each local region, and the sum is taken over the region centered at that value (zero padding is added where necessary).

Optional parameters include:

- **local_size** [default 5]: the number of channels to sum over (for cross channel LRN) or the side length of the square region to sum over (for within channel LRN)
- **alpha** [default 1]: the scaling parameter (see below)
- **beta** [default 5]: the exponent (see below)
- **norm_region** [default *ACROSS_CHANNELS*]: whether to sum over adjacent channels (*ACROSS_CHANNELS*) or nearby spatial locations (*WITHIN_CHANNEL*)

Thus if the input dimensions are n * c * h_i * w_i, the output would have dimensions n * c * h_o * w_o, where h_o and w_o are computed in the same way as convolution.

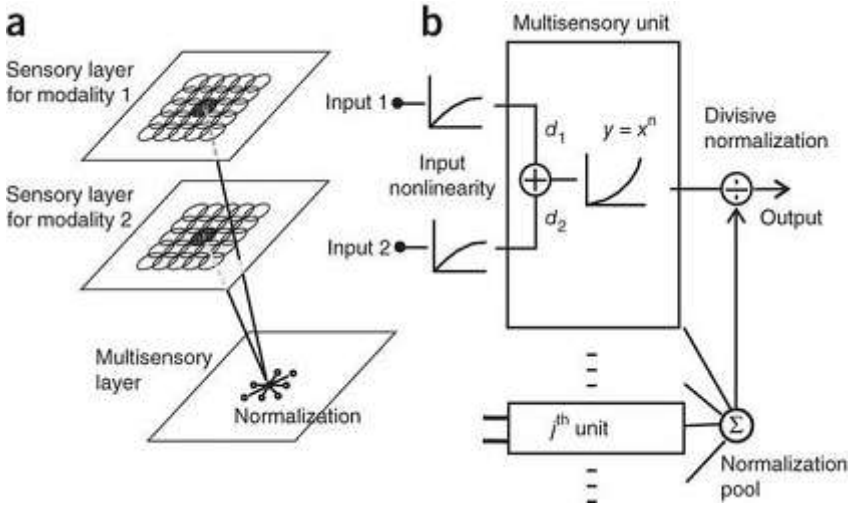

Fig 1.4 Local response normalization layer

4. **ReLU:** Given an input value x, the ReLU layer computes the output as x if x > 0 and negative_slope * x if x <= 0. When the negative slope parameter is not set, it is equivalent to the standard ReLU function of taking max(x, 0). It also supports in-place computation, meaning that the bottom and the top blob could be the same to preserve memory consumption.

Optional parameters include:

- **negative_slope** [default 0]: specifies whether to leak the negative part by multiplying it with the slope value rather than setting it to 0.

The input and output sizes match those of the preceding and subsequent layers.

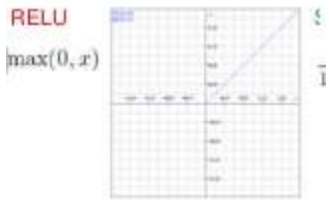

Fig 1.5 ReLU layer

5. InnerProduct: The InnerProduct layer (also usually referred to as the *fully connected* layer) treats the input as a simple vector and produces an output in the form of a single vector (with the blob's height and width set to 1). It takes in the following parameters:

- **num_output (c_o):** the number of filters

- **weight_filler** [default type: 'constant' value: 0]

Optional parameters include:

- **bias_filler** [default type: 'constant' value: 0]
- **bias_term** [default true]: specifies whether to learn and apply a set of additive biases to the filter outputs

Thus, if the input dimensions are n * c_i * h_i * w_i, the outputdimensions come out as n * c_o * 1 * 1.

Using the above layers, we try to replicate Alex Krizhevsky's network in Caffe. Fig 1.6 shows the traditional AlexNet, while Fig 1.7 shows the differences between the computational flow graphs of CaffeNet versus AlexNet [2] .

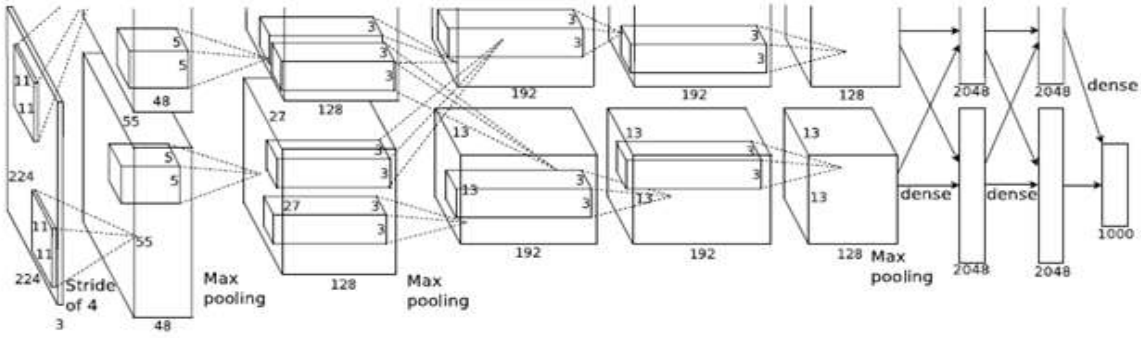

Fig 1.6 Alex Krizhevsky's network

Furthermore, we have implemented AlexNet using the SDAccelTM platform from Xilinx. SDAccel™ is a development environment for OpenCL applications targeting PCIe® based Virtex®-7, and Kintex®-7 FPGA accelerator cards. This environment enables concurrent programming of the system processor and the FPGA logic without the need for RTL design experience. The application is captured as a host program written in C/C++ and a set of computation kernels expressed in C, C++, or the OpenCL C language.

SDAccel allows the division of code into parts which can be executed separately on the host (CPU) and the accelerator (FPGA). The main features of the host code include:

- Connecting to the platform and compute device.
- Creating a compute context.
- Creating the compute kernel and allocating memory.
- Executing the kernel function.
- Reading back the results.

The main features of the kernel code are:

- Executing the kernel function (user-specified)
- Writing the output back to the host when requested.

The SDAccel version used was version 2014.3. Fig 1.8 shows the flow graph for the same.

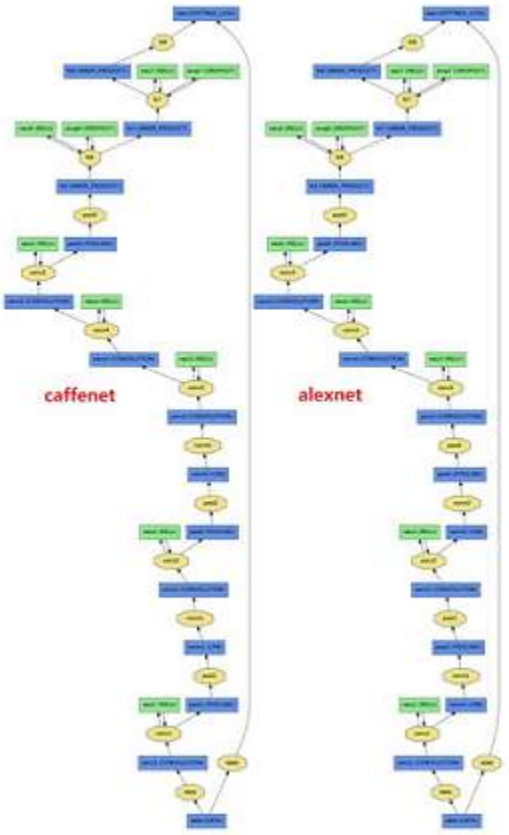

Fig 1.7 CaffeNet vs AlexNet

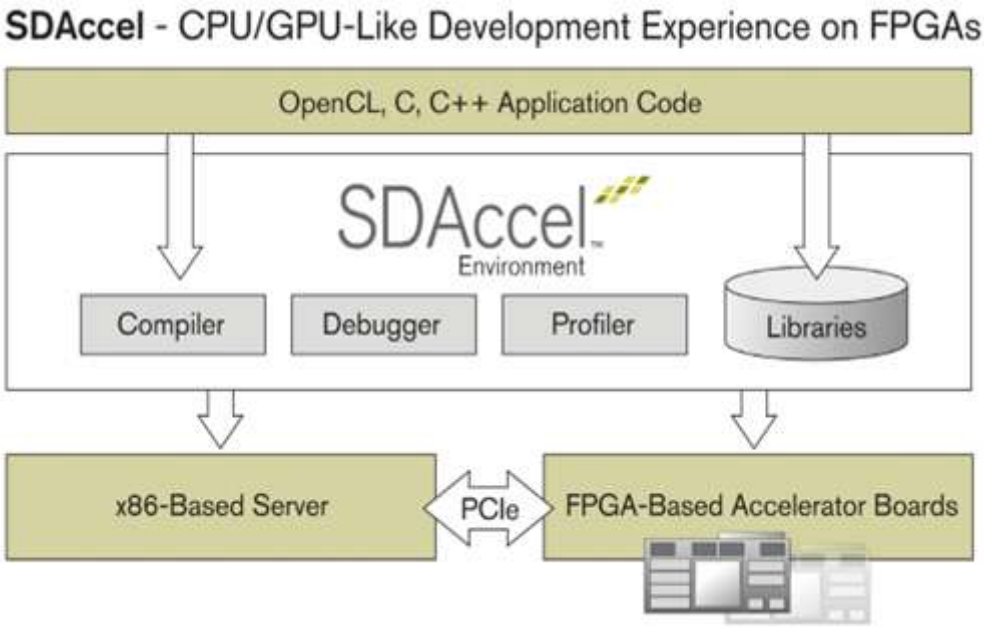

Fig 1.8 SDAccel development platform for Xilinx FPGA's

Thus, on a high-level, our project goal was three-fold:

1. Create the CONV, POOL, LRN and ReLU layers using OpenCL and test them using SDAccel.

2. Merge these layers into the existing Caffe implementation prepared by Roberto that was set up to run on FPGA's.

3. Benchmark and compare speed and accuracy of the results of the hardware-cum-software implementation of Caffe (which would use our IP's) against the software-only version of Caffe.

The next section describes the current status of the project.

2. Current Status

As of the end of our project (July 2015), we had worked on and completed the following components:

1. Creation of the CONV, POOL, LRN and ReLU layers of various sizes (corresponding to those needed for AlexNet) using OpenCL.

2. Testing the above layers using randomized inputs with an OpenCL testbench.

3. Running the host and kernel code on the Virtex7 FPGA using the SDAccel platform and measuring and monitoring area, performance and latency estimates.

4. Integration of these layers into the current Caffe code (modified to run on FPGA's) and benchmarking the performance of the former against the CPU-only version.

We also made certain functionality improvements over the software implementations of these layers to achieve greater performance on FPGA's. These included the following:

1. Loop parallelism via the application of pipeline directives

2. Using bit-shifts wherever possible.

3. Addition of pragmas to ensure compatibility with the AXI bus.

4. Rewriting loops wherever possible to achieve speedup.

The following functionality was not present in our project, but would be desirable:

1. Conversion of the above IP's from HLS-based to pure OpenCL. This would allow fine-grained thread-level parallelism at runtime.

2. Creation of other IP's (InnerProduct, FullyConnected) using OpenCL. Currently, our implementation in Caffe covers only 14 of the 20 possible IP's (70% hardware-based). Our goal is to make it 100% hardware-based.

3. Identify performance botllenecks in our designs, and implement fixes for the same so to achieve speedup equivalent to that of the CPU-only Caffe version.

4. Performing real-time video recognition using multiple instances of our IP's.

3. Initial Architectural Design

The initial architectural design was based on two sources:

1. Implementation of CNN's using the DaDianNao[3] architecture. This consisted of the CONV followed by the LRN followed by the POOL followed by the CLASS layers.

2. Input dimensionality reduction using the Principal Component Analysis layer, followed by compression and decompression of intermediate results using the compress/decompress layers. The compressed layers go into a special memory unit, and the results from the memory unit are decompressed whenever a fetch request is initiated.

Based on these initial assumptions, our block diagram for the system was as follows:

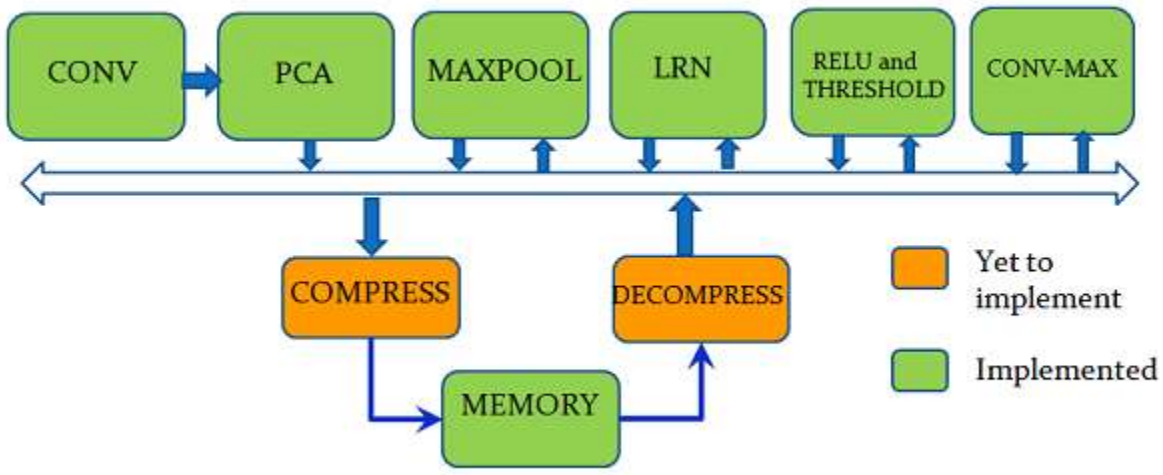

Fig 3.1Initial system architecture

The above block diagram shows that all the above layers are connected via the CDB (Common Data Bus) to the compression/decompression units that reside close to memory. The input enters from the left (CONV). After convolution is performed, the output of CONV is fed to the PCA unit, where the size is reduced to the fewest number of components required for image classification. All other layers (MAXPOOL, LRN, ReLU and THRESHOLD, CONVMAX) communicate directly with memory via Send/Receive requests. Upon a Receive request, the layer number and the number of bytes requested is sent via the CDB to the decompression layer, which forwards it to memory. The location in memory where the relevant data is stored can be found using the layer number. The bit next to the layer number is used to check for the validity of the data. If it is 0, it means the data is old, and if 1, the data is new. If 1, the memory unit forwards the compressed data to the decompress unit, which decompresses it before forwarding it to the requesting unit.

The exact opposite occurs when a Send request is sent. Upon the latter occurring, the compress unit receives the request, compresses the data, and sends the layer number and the compressed data to the memory unit for storage. The memory unit then stores the incoming data in row-major order (in consonance with the C language) and marks the valid bit as 1. The valid bit is separate for each layer and data for the latter, and is set to 0 whenever multiple requests are made for operation on the same block of data.

Fig 3.2 Memory sub-system architecture

The need for compression and decompression arose from the possibility of the input size being larger than the sizes of input supported by HLS. Even if one were to do PCA, the size of the output could still be larger than that supported by the HLS tools. We also needed to make sure that memory bandwidth requirements were minimum as well, since repeated Send/Receive requests could create memory bottlenecks. Various algorithms (which were is use at this point in the project) were explored, including but not limited to:

- FELICS
- Goulomb-Rice with Differential-Differential Pulse Code Modulation
- LOCO-I
- LZSS
- JPEG

- Arithmetic
- Huffman

Fig 3.3 briefly summarizes the performance of some of these compression algorithms.

Algorithm	DDPCM+GR	JPEG-LS	FELICS	Arithmetic Code
Technology	0.15µm	0.18µm	0.13µm	Vertex 4
Operating freq	170 MHz	40 MHz	273 MHz	123 MHz
Throughput	2720 MB/s	9.9 MB/s	546 MB/s	15,37 MB/s
Compression	5.26 b/s	-	3.40 b/s	4.55 b/s
Parallelism	16	1	2	1
Memory	Not used	2.25KB	1.9KB	7.7KB
Reference	[2]	[3]	[4]	[5]

Fig 3.3 Compression algorithms [4]

Based on the above numbers, we decided to choose DDPCM+GR as it seemed to have the best performance in terms of operating frequency, throughput , compression speed and the ability to be parallelized.

For the decompression phase, our assumptions were based on the fact that we chose DDPCM+GR for encoding and decoding, and were as follows:

- Prediction scheme would based on LOCO-I to reduce correlative redundancy between sequential pixels.
- The data packages after source coding would contain a continuous stream of prefix codes, in order to eliminate the header data imposed by more advanced packing schemes.
- Higher demand would be placed on the decoding side in terms of resource usage, because of the need for high parallelism when a new prefix code was counted and decoded each clock cycle
- Faster decompression could be achieved via parallel decoder modules, the downside being reduced quality.

Fig 3.4 below shows the top-level pipeline diagram for the compression/decompression (aka the encoder/decoder modules), while Fig 3.5 shows the system flow diagram for the same. Further information about these modules can be found in [4].

The PCA layer, as described previously, was used for reduce the input feature space so as to allow HLS to proceed successfully. It would do so by measuring the variance of two adjacent

input pixels with respect to each other, and then apply an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. Fig 3.6 shows an illustration applied to genomic classification.

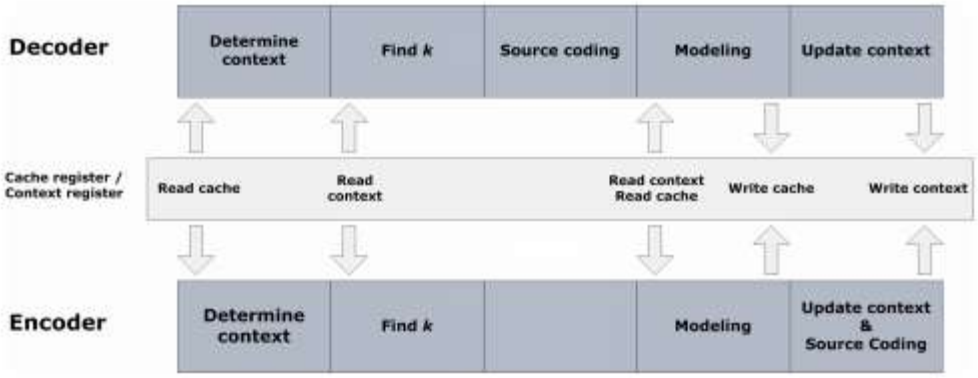

Fig 3.4 Top-level pipeline diagram

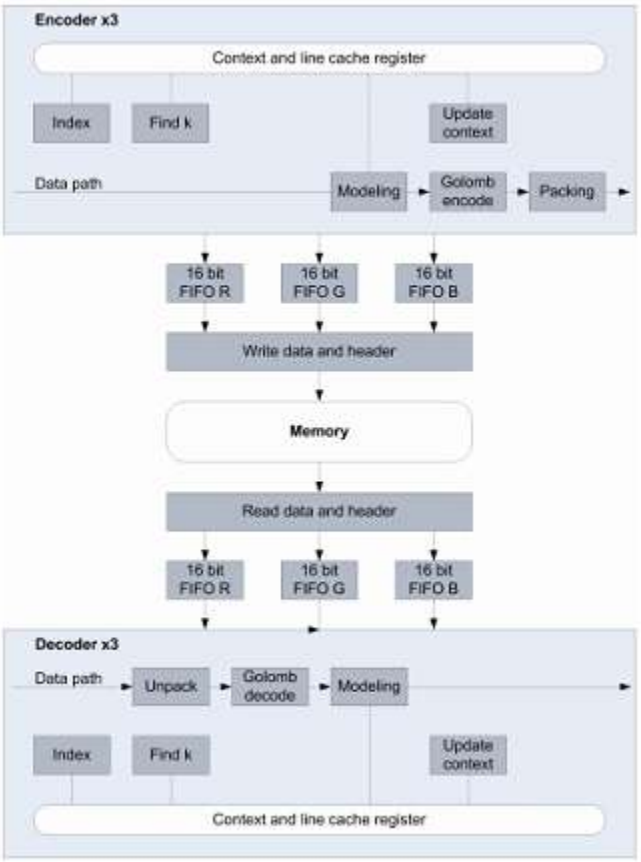

Fig 3.5 System-flow diagram

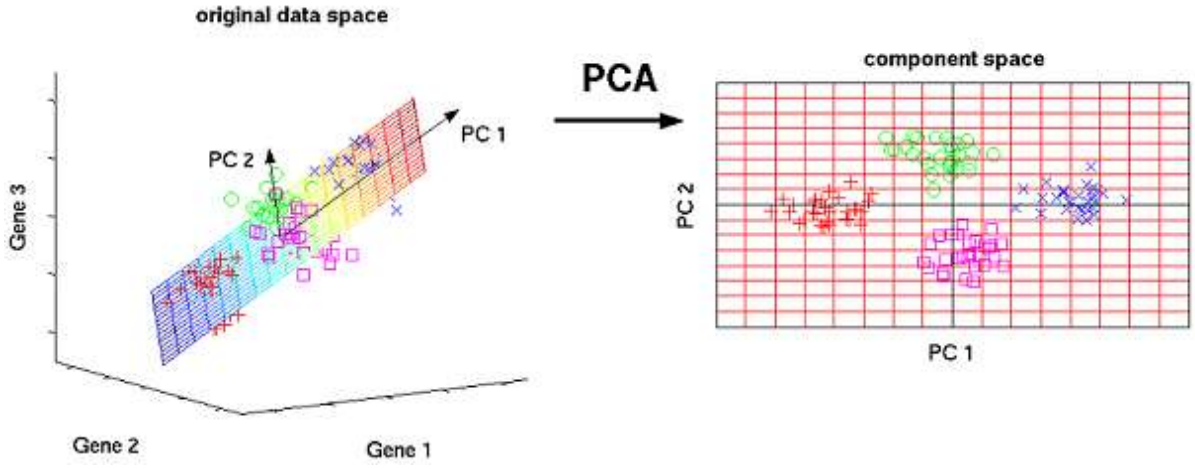

Fig 3.6 Principal Component Analysis [5]

The next section goes on to describe the architectural considerations and the final design that was chosen.

4. Architectural Evolution and Final Architectural Design

As described above, we had completed the design of the various layers using Vivado HLS, and had targeted it for the Zedboard. Later on, we decided to move our design methodology to the SDAccel platform. This was because of the following advantages offered by SDAccel:

- Provision of CPU-GPU like platform to increase performance.
- Use of OpenCL APIs to increase parallelism.
- Burden of context switching is on OpenCL.
- Facilitating integration with Caffe library

Fig 4.1 below shows the execution flow of a program using SDAccel.

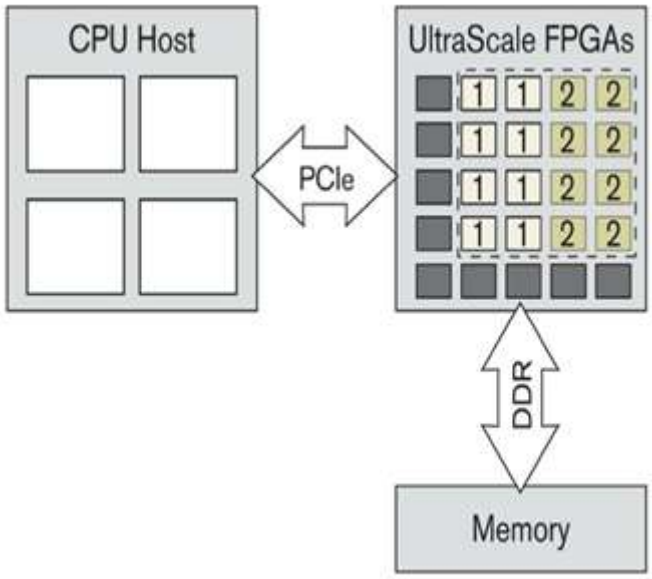

Fig 4.1 Program execution flow using SDAccel

Thus, we realized that we could save a lot of effort involved in re-creating the IP and fine-tuning the software to achieve performance gains. The block diagram of the final system is shown below:

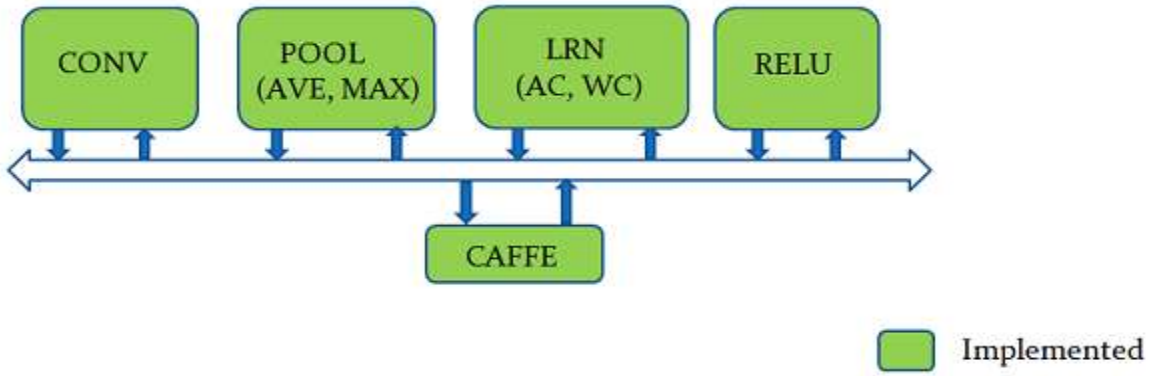

Fig 4.2 Final system architecture

In the above diagram, the CONV, POOL, LRN and RELU layers are connected directly to the Caffe program via software API calls. These kernels are created as HLS kernels, are invoked in Caffe using the *clEnqueueTask* API call. This allows us to apply HLS directives to our IP's, while still invoking them using the C standard. All of the IP's have been pipelined to achieve the maximum speedup possible while using them within the SDAccel standard.

Figs 4.3, 4.4, 4.5, 4.6, 4.7 and 4.8 show the pseudo-code listing for the final implementations of the CONV, POOL (AVE), POOL (MAX), LRN (WC), LRN (AC) and ReLU layers, respectively.

```
for row from 0 to NUM_OUT_ROWS in unit steps
  for col from 0 to NUM_OUT_COLS in unit steps
    set window_start_idx = (row * (NUM_DATA_COLS) * STRIDE) + (col * STRIDE)
    set  out_idx = (row * (NUM_OUT_COLS)) + col
    initialize bufc[out_idx] to 0
    for filter_row from 0 to NUM_MASK_ROWS  in unit steps
        for filter_col from 0 to NUM_MASK_COLS  in unit steps
        set   data_idx = window_start_idx + (NUM_DATA_COLS* filter_row) + filter_col
        set  filter_idx = (filter_row * NUM_MASK_COLS) + filter_col
        set  bufc[out_idx]  = bufc[out_idx]  + (bufa[data_idx] * bufb[filter_idx])
.
```

Fig 4.3 Pseudo-code for CONV layer

```
for row from 0 to IHEIGHT - NUM_MASK_ROWS + 1 in STRIDE steps
  for col from 0 to IWIDTH - NUM_MASK_COLS + 1  in STRIDE steps
    initialize m, idx
     for prow from 0 to NUM_MASK_ROWS  in unit steps with row+prow <IHEIGHT
        for pcol from 0 to NUM_MASK_COLS  in unit steps with col+pcol < IWIDTH
            if (inbuf[IDX2C(col + pcol, row + prow, IWIDTH)] > m)
                        set idx = IDX2C(col + pcol, row + prow, IWIDTH)
                        set  m = inbuf[idx]
    set obuf[count]=m
    increment count
 iterate
```

Fig 4.4 Pseudo-code for POOL (MAX) layer

```
for row from 0 to IHEIGHT - NUM_MASK_ROWS + 1 in STRIDE steps
  for col from 0 to IWIDTH - NUM_MASK_COLS + 1  in STRIDE steps
    initialize m, idx
     for prow from 0 to NUM_MASK_ROWS  in unit steps with row+prow <IHEIGHT
        for pcol from 0 to NUM_MASK_COLS  in unit steps with col+pcol < IWIDTH
                    IDX2C(col + pcol, row + prow, IWIDTH)] > m)
                        set idx = IDX2C(col + pcol, row + prow, IWIDTH)
                        set  m+ = inbuf[idx]
    set obuf[count]=m/(NUM_MASK_ROWS*NUM_MASK_COLS)
    set m to 0
    increment count
 iterate
```

Fig 4.5 Pseudo-code for POOL (AVE) layer

```
set values for alpha, beta and size
for  n from 0 to NUM_BOTTOM_BLOBS in unit steps
    for c from 0 to NUM_BOTTOM_CHANNELS in unit steps
     for  h from 0 to IHEIGHT in unit steps
       for w from 0 to IWIDTH in unit steps
          set c_start = c - (size - 1) / 2
          set c_end = min(c_start + size, NUM_BOTTOM_CHANNELS)
           set c_start = max(c_start, 0)
           set scale = 1
         for  i from c_start  to  c_end in unit steps
           set value = data in bottom blob at points (n, i, h, w)
           set scale += value * value * alpha / size
      set final value = data in bottom blob at (n, c, h, w) / pow(scale, beta)
iterate
```

Fig 4.6 Pseudo-code for LRN (AC) layer

```
set values for alpha, beta and size
for n from 0 to NUM_BOTTOM_BLOBS in unit steps
    for  c from 0 to  c NUM_BOTTOM_CHANNELS in unit steps
      for h from 0 to IHEIGHT in unit steps
        set h_start = h - (size - 1) / 2
        set h_end = min(h_start + size, IHEIGHT)
        set  h_start = max(h_start, 0)
        for w from 0 to IWIDTH in unit steps
          set scale = 1
          set w_start = w - (size - 1) / 2
          set w_end = min(w_start + size, IWIDTH)
          set w_start = max(w_start, 0)
          for nh from h_start to  h_end in unit steps
            for nw from w_start to  w_end in unit steps
              set value = data in bottom blob at location(n, c, nh, nw)
              set scale += value * value * alpha / (size * size)
        set final value =  data_in bottom blob at location( n, c, h, w) / pow(scale, beta)
iterate
```

Fig 4.7 Pseudo-code for LRN (WC) layer

```
for i from 0 to NO_OUT_NODES  in unit steps
                        initialize out_array[i] to 0
                        for  j from 0  to NO_IN_NODES in unit steps
                        set out_array[i] += in_array[j] * synaptic_weights[(i*NO_IN_NODES) +j]
        set temp = out_array[i]
         if (temp >= 0)
           set out_array[i] = temp
         else
            set out_array[i] = 0
 iterate
```

Fig 4.8 Pseudo-code for ReLU layer

This section has presented a detailed description of our project. To summarize, we had created the initial versions of our IP's using Vivado HLS, and had then ported them to SDAccel. We created a top-level testbench to test each IP, and then integrated the same into the existing implementation of Caffe. This was then run on an FPGA, and the results of the same were compared to the CPU version. These results are described in the forthcoming sections.

5.Methodology

Initially, when we were creating our project in Vivado HLS, our project had both hardware and software components. Different approaches were therefore adopted when dealing with them.

In the case of software, a rough version of the software was first written with all the major components in place, was iteratively updated later on to achieve required functionality. This promoted flexibility and helped us to manage complexity by taking advantage of the quick iteration times afforded by software development.

In the case of hardware, the design flow started by understanding of the Zedboard and the Zynq SOC. Having gained a basic understanding of our project architecture, we next moved on to components for which we were relatively confident of the implementation. This included creating the MicroBlaze soft processor and setting up of the compression/decompression and memory blocks. Working on these components allowed us to be productive while gaining further understanding of the hardware architecture.

Having set-up these peripheral components, an initial detailed block diagram of the hardware system was drawn up. We then started to implement hardware components which we predicted would be very simple to interface with the rest of the SOC. These included all the IP's listed previously. Each of our hardware blocks was developed independently but with the eventual integration requirements in mind. All individual blocks were implemented, simulated, and debugged.

Later on , since our project had moved from implementing the IP's on the Zedboard to the same using SDAccel, we had to do relatively little work in terms of porting, since our IP's were already written in C. The only additional work that needed to be done was creating test benches in OpenCL and thoroughly testing our IP's for correctness.

The next step was FPGA-level implementation during which we ran the entire program on the Virtex7 FPGA. We created a top-level OpenCL testbench and ran it on the FPGA. During this stage we fixed any integration oversights, and synthesis/PAR gave us an estimation of the overall project resource utilization.

The last stage was combining our IP's with Caffe, and creating a top-level OpenCL testbench that ran our program against a set of test images from the Caffe database. Fig 5.1 on the next page shows the project design flow.

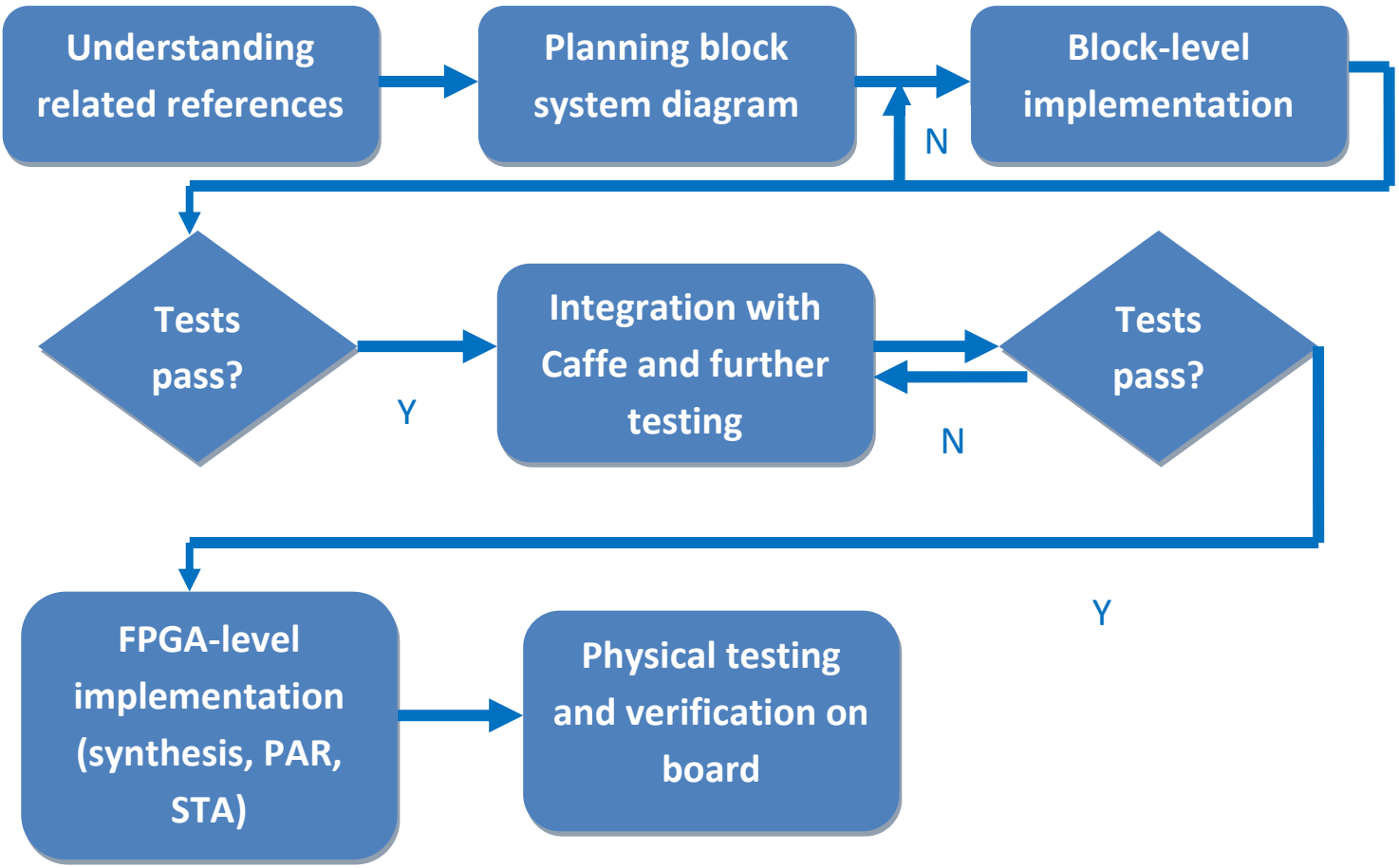

Fig 5.1 Project design flow

5.1. Design Environment

The FPGA board used in our project was the Alpha Data ADM-PCIE-7V3 from Xilinx, which consisted of one Xilinx Virtex-7 690T FPGA device and 16 GB of DDR3 memory. This project was carried out using computers running Windows or Linux, and Vivado and Vivado HLS versions 2014.3. All of the software for the both Vivado and SDAccel tools was written in C, with the test benches in the latter coded using OpenCL. At the outset of this project, our presentations, reports, and reference documents were shared using Google Docs. However, as development began we quickly moved to Github for the source code portion. Github enabled us to have a record of past revisions and to work on source code simultaneously (though working on the same file simultaneously was generally avoided due to the potential for introducing conflicts). In summary, the tools used for this project include:

- FPGA board: Xilinx Alpha Data ADM-PCIE-7V3

- Languages: C, OpenCL
- FPGA design: Vivado 2014.3 and Vivado HLS 2014.3 (earlier), SDAccel 2014.3 (later)
- Source code control: Github

5.2. Partitioning

This project was partitioned based on design environment. From the software perspective, all of the IP's were initially developed and tested using Vivado, Vivado HLS and SDAccel. Their functionality was exercised using test benches coded in OpenCL. On the hardware side, board-level verification helped us to find out any inefficiencies and allowed the addition of various HLS directives to the source code to improve performance and accuracy.

Apart from the partitioning above, learning and planning were also taken in to consideration as an essential part of this project. Learning how the SDAccel and Vivado tools worked and their various interfaces, planning out the communication protocols, figuring out how to arrange data memory and other administration or arrangement made besides coding were also crucial and regarded as priority.

5.3. Simulation, Verification and Testing

Generally speaking we have performed three types of simulation and testing after our design procedure. Each type of verification step is detailed below, and reflects the overall design methodology described at the beginning of Section 5.

Block-level functional simulation

This was the functionality testing to ensure our IP's behaved as intended. In this stage, OpenCL test benches were used to test our IP's for all corner cases that could occur during runtime (non-square matrix sizes, need for padding for ghost or halo cells, out of bounds access etc).

Top-level testing in Caffe

In this stage, the IP's were integrated with Caffe, and were synthesized and run on the FPGA.

Comparison with software implementation of Caffe

In this stage, the performance of the hardware-based run above was compared with the software-based run from Caffe. Tweaks were then applied to our hardware models to ensure reasonable performance.

6. Contributions

Arjun Gandhi:

- High-level architectural planning
- Did research on various types of CNN and DNN architectures.
- Created and tested the ReLU and THRESHOLD layers in Vivado, Vivado HLS and SDAccel
- Created and tested the CLASS layer in Vivado and Vivado HLS.
- Created and tested the MEMORY layer in Vivado and Vivado HLS.

Mahmoud KhalafAlla:

- High-level architectural planning
- Did research on various types of CNN and DNN architectures and gave illustrations about the basic components of such networks.
- Created and tested the CONV layers in Vivado, Vivado HLS and SDAccel, in a manner compliant with Caffe implementation.
- Created and tested the CONVMAX layers in Vivado and Vivado HLS.
- Proposed several ideas to improve the performance of CONV and POOL layers performance in SDAccel.

Venkatesh Mahadevan:

- High-level architectural planning
- Did research on various types of CNN and DNN architectures.
- Created and tested the POOL layers in Vivado, Vivado HLS and SDAccel for AVE , MAX and STOCHASTIC cases.
- Created and tested the LRN layer in Vivado, Vivado HLS and SDAccel for five different activation functions (STEP, LINEAR_COMBINE, CONTINUOUS_LOG_SIGMOID, CONTINUOUS_TAN_SIGMOID and SOFTMAX) for both the WITHIN_CHANNEL and ACROSS_CHANNEL cases.
- Created and tested the LRN layer in Vivado, Vivado HLS and SDAccel for both the WITHIN_CHANNEL and ACROSS_CHANNEL cases in a manner compliant with the Caffe implementations.
- Created and tested the PCA layer in Vivado and Vivado HLS.
- Created and tested the compression/decompression layers using DDPCM+GR encoding/decoding in Vivado and Vivado HLS.

7. Design Characteristics

This section describes the resource utilization and timing estimates for all our individual IP's, as well comparison of the performance between the software-only and hardware-cum-software based versions of the image recognition software using Caffe. In addition, the time spent by the team on each of the different project components is also specified.

7.1. Resource Utilization and Timing Estimates

Fig 7.1.1 below shows the resource utilization and timing estimates for the CONV IP.

Kernel Name	Target Freq	Estimated Freq	Start Interval	Best Case	Avg Case	Worst Case	FF	LUT	DSP	BRAM
Conv1	167 MHZ	144 MHZ	239837	239836	239836	239836	10652	10614	12	138
Conv2	167 MHZ	178 MHZ	11350	11349	11349	11349	4239	3588	11	4
Conv3	167 MHZ	178 MHZ	1329	1328	1328	1328	2978	2480	11	3
Conv4	167 MHZ	178 MHZ	1329	1328	1328	1328	2978	2480	11	3
Conv5	167 MHZ	178 MHZ	1329	1328	1328	1328	2978	2480	11	3

Fig 7.1.1 Resource utilization and timing estimate for CONV layer

Fig 7.1.2 shows the resource utiltization and timing estimates for the POOL IP for the MAX case.

Kernel Name	Target Freq	Estimated Freq	Start Interval	Best Case	Avg Case	Worst Case	FF	LUT	DSP	BRAM
pool1	167 MHZ	190.84 MHZ	9028	9027	9027	9027	9470	28521	2	10
pool2	167 MHZ	190.84 MHZ	1244	1243	1243	1243	4123	10736	2	4
pool5	167 MHZ	190.84 MHZ	132	131	131	131	1937	3460	0	2

Fig 7.1.2 Resource utilization and timing estimate for POOL (MAX) layer

Fig 7.1.3 shows the resource utiltization and timing estimates for the POOL IP for the AVE case.

Kernel Name	Target Freq	Estimated Freq	Start Interval	Best Case	Avg Case	Worst Case	FF	LUT	DSP	BRAM
pool1	167 MHZ	157.978 MHZ	9052	9051	9051	9051	5267	8583	8	10
pool2	167 MHZ	162.602 MHZ	1283	1282	1282	1282	4272	4573	8	3
pool5	167 MHZ	190.84 MHZ	180	179	179	179	3361	2953	6	2

Fig 7.1.3 Resource utilization and timing estimate for POOL (AVE) layer

Fig 7.1.4 on the next page shows the resource utiltization and timing estimates for the LRN IP for the AC case.

Kernel Name	Target Freq	Estimated Freq	Start Interval	Best Case	Avg Case	Worst Case	FF	LUT	DSP	BRAM
lrn1	167 MHZ	190.84 MHZ	635250 2~9982 502	6352501	9075001	99825 01	1024 4	10995	109	0
lrn2	167 MHZ	190.84 MHZ	111090 2~1745 702	1110901	1587001	17457 01	1022 0	10958	109	0

Fig 7.1.4 Resource utilization and timing estimate for LRN (AC) layer

Fig 7.1.5 shows the resource utiltization and timing estimates for the LRN IP for the WC case.

Kernel Name	Target Freq	Estimated Freq	Start Interval	Best Case	Avg Case	Worst Case	FF	LUT	DSP	BRAM
lrn1	167 MHZ	190.84 MHZ	599727 0~6062 047270	599726 9	146961 3269	606204 7269	10176	11061	107	0
lrn2	167 MHZ	190.84 MHZ	105071 0~1060 108710	105070 9	257002 069	106010 8709	10159	11037	107	0

Fig 7.1.5 Resource utilization and timing estimate for LRN (WC) layer

Fig 7.1.6 below shows the resource utilization and timing estimates for the ReLU case.

Kernel Name	Target Freq	Estimated Freq	Start Interval	Best Case	Avg Case	Worst Case	FF	LUT	DSP	BRAM
relu_1	167 MHZ	190.84 MHZ	15144	15143	15143	15143	1467	1611	5	0
relu_2	167 MHZ	190.84 MHZ	542	541	541	541	879	1055	0	0
relu_3	167 MHZ	190.84 MHZ	77	76	76	76	876	1054	0	0
relu_4	167 MHZ	190.84 MHZ	62	61	61	61	875	1041	0	0
relu_5	167 MHZ	190.84 MHZ	49	48	48	48	875	1041	0	0
relu_6	167 MHZ	190.84 MHZ	22	21	21	21	873	1034	0	0
relu_7	167 MHZ	190.84 MHZ	22	21	21	21	873	1034	0	0

Fig 7.1.6 Resource utilization and timing estimate for ReLU layer

Fig 7.1.7 below shows the programming and execution times of all layers on the FPGA, and the ratio of the same to the execution time on a CPU. Of particular interest is the fact that the CONV and POOL layers considerably slow down execution times (by almost 17x) due to their huge sizes. Efforts are underway at this time to optimize their execution so as match CPU performance. Appendix A shows an example of the top-level testbench coded in OpenCL. Appendix B shows an example of the kernel code in C. Appendix C shows an example of the .tcl file used for execution, while Appendix D shows an example of the run log.

	Time to Program FPGA	Total Layer Tim	FPGA Execution Time	CPU Execution Time	Ratio
conv1	1413.77	9510.22	8096.45	11.703	691.826882
relu1	1323.63	1372.32	48.69	0.267	182.359551
norm1	1648.54	2387.58	739.04	9.555	77.3458922
pool1	1331.28	1385.08	53.8	0.916	58.7336245
conv2	1332.1	25474.5	24142.4	24.035	1004.46848
relu2	1329.34	1362.1	32.76	0.172	190.465116
norm2	2600.67	2783.16	182.49	6.053	30.1486866
pool2	1328.9	1431.57	102.67	0.619	165.864297
conv3	1330.88	88693.5	87362.62	13.049	6694.96666
relu3	1313.78	1329.98	16.2	0.061	265.57377
conv4	1326.8	81278.3	79951.5	10.323	7744.98692
relu4	1326.18	1344.53	18.35	0.061	300.819672
conv5	1322.29	61532	60209.71	7.336	8207.43048
relu5	1325.02	1339.75	14.73	0.041	359.268293
pool3	1325.28	1506.23	180.95	0.169	1070.71006
relu6	1325.65	1328.82	3.17	0.005	634
relu7	1322.31	1325.43	3.12	0.005	624

Fig 7.1.7 Programming and execution times of all layers, and ratio w.r.t CPU time

7.2. Where the time went

Figure 7.2.1 on the next page summarizes the time consumed in each type of work for our project. First week of actual work started on February 6, instead of the beginning of the semester.

		Understanding the problem and making plans	Research and learning	Setting up, learning and debugging tools	Block-level coding and simulation	Integration, System-level verification and Testing	Total hours
05-Feb	week1	10	10				20
12-Feb	week2		10	10	10		30
19-Feb	week3	10	10	10			30
26-Feb	week4		20	20	15		55
05-Mar	week5	10	15				25
12-Mar	week6			10	10		20
19-Mar	week7	10	20		10	5	45
26-Mar	week8	10	20		10		40
02-Apr	week9		20		20		40
09-Apr	week10	10			10		20
16-Apr	week11				30		30
23-Apr	week12				25		25
30-Apr	week13	5			20	5	30
07-May	week14	10			10	5	25
14-May	week15	10			10	10	30
21-May	week16	10	10	10	10	10	50
28-May	week17	10	10	10	5		35
04-Jun	week18	20		5	10	10	45
11-Jun	week19	20			10	10	40
18-Jun	week20	20			10	5	35
25-Jun	week21	10			10	5	25
02-Jul	week22				10	5	15
09-Jul	week23				10	5	15
16-Jul	week24					15	15
23-Jul	week25					15	15
	Total Hours	175	145	75	255	105	755
	Percentage	23.18%	19.21%	9.93%	33.78%	13.91%	100%

Fig 7.2.1 Time consumption analysis (time unit in hours)

The project itself lasted for 26 weeks, with no break time being taken. Working hours are approximate and estimated either from our weekly reports or group emails. On average our team spent about 30 hours on the project each week. However, actual working time for each week fluctuated.

The first column "***understanding the problem and making plans***", includes time taken for designing the system, planning the code structure and also group meeting times. Its time consumption peaks right before starting the block level coding and system integration. And its total consumption is the smallest among all five tasks, which is reasonable.

"***Tools***" include all software tools we have used, including ones for both FPGA design, MicroBlaze processor application design, as well as ones for simulation and verification. Therefore the time consumption on "*Setting up, learning and debugging tools*" also reaches its maximum at the beginning of the term as well as at the beginning of the overall integration.

"***Research and learning***" consumed a heavy portion of total time throughout the whole project period, as this was the first time the implementation of an image recognition system using a novel framework that made use of high-level synthesis using OpenCL was being performed, and this required system migration and recoding as well.

"***Block level coding and system level verification***" both took a large portion of our total time. Even though we simulated each block individually, integrating them together enhanced our

understanding of the system, brought out new bugs, and necessitated code updates. Therefore most of time spent during the integration stage was for debugging and making sure system accuracy was at its maximum and latency at its minimum.

The next section goes on to describe the problems encountered during the project phase.

8. Problems

Need to specify the sizes of the input and output at function interfaces for those functions that lie in the synthesis path

Dynamic memory allocation is not supported in Vivado or SDAccel, which forces the user to specify the sizes of the input and output and hence makes the program inflexible for varying sizes.

Inability to synthesize library functions such as malloc(), pow(), and file I/O functions such as printf(), sscanf() etc

The Vivado suite of tools are not able to synthesize most library functions, with the result being that the user must pay attention to remove these out of the synthesis path.

Prevention of the use of DATAFLOW directive at lower levels

The use of the DATAFLOW directive would allow for function execution to start as soon as data was ready, without waiting for the full computation to complete. However, the limitation in Vivado HLS of having this applied only to the top-level prevented parallelism at lower-levels which were more compute-intensive, and at the same time data-parallel in many cases.

Inability to update global variables properly (All variables that are updated within a function need to be local to that function).

Both in Vivado and SDAccel, the use of global variables is seen to result in stuck-at values, which causes the co-simulation to hang. Reverting to local variables seems to resolve the issue, at the expense of more memory usage.

Input and output sizes must be limited. Board resource limits and clock timings are violated if they exceed a certain size, and it varies from application to application.

In Vivado, the tool does not give a warning in case the board's resources are exceeded, and allows the user to proceed with the remaining synthesis steps. In SDAccel, the tool does not do a check for the case in which resources are exceeded, and crashes arbitrarily with a core dump.

Need to specify loop bounds for loops in code that does not lie in the critical path for synthesis (for latency calculations).

In the case of functions that do not lie in the synthesis path, both tools seem to have the need for loop limit specification for loops, and failure to do so results in the inability in latency calculation on part of the tool.

Recursive functions are not supported.
Both Vivado and SDAccel seem to reject the handling of recursive functions as they cannot seem to infer recursion depth, and hence cannot estimate the size of hardware that needs to be created.

Classes and templates are not completely supported.
Both Vivado and SDAccel cannot synthesize constructs that result from object-based programming paradigms.

Need to use specialized hardware for floating point operations to obtain reasonable accuracy
Both Vivado and SDAccel require that the user use the appropriate hardware (LogiCore or Synthesized) to achieve the required accuracy for floating-point operations.

Inability of the tool to inform the user about the type of logic which a particular piece of code maps to. This often leads to cases wherein the co-simulation feature runs for hours and never finishes.
The Vivado suite of tools seems to run co-simulation for hours in the case the inputs/outputs are large or the operations are compute-intensive, and this seems to be due to the fact that the tool does not inform the user about the characteristics of the hardware to which it maps the compute operations, and any stuck-at faults not handled by that hardware result in simulation stalls and cannot be resolved by the user.

Segmentation faults if the improper interface type is specified during RTL export.
If the user tries to use any other interface besides '*axilite*', the Vivado suite of tools seem to crash during RTL export instead of exiting gracefully with an informative error message.

Generation of test vectors (in the cosim_tv.exe file) for every input combination in the design. While this may exercise the design thoroughly, any stuck-at faults or X propagation issues that are encountered in this process cannot be traced back to the high-level code, since these result from the logical behaviour of the RTL synthesized by the tool. The resultant effect is that co-simulation can hang sometimes and never finish.
The Vivado suite of tools does not seem to do intelligent vector generation to verify the design. Rather, it tries to perform the latter via an exhaustive process, resulting in millions of combinations of test vectors that result in very long simulation times.

Vector-less activity propagation is run during bitstream generation when the IP is integrated with the Zynq 7000 SOC. This results in a low confidence level when the design is tested on the board. The tool requires the user to input a SAIF file or a VCD file for dynamic power estimation using Primetime. Only then can the user achieve a high confidence level on their design.
In order to gain high-confidence about their design, the Vivado suite of tools seem to require that the user specify the power intent of the design, and cannot produce a satisfactory result for designs where power consumption cannot be estimated during the initial phases of design.

Lack of profiling/debugging features in SDAccel
Simulation or hardware integration often crashes in SDAccel. Due to lack of profiling/debugging features, it becomes impossible for the user to debug the design for issues.

Lack of timely support on Xilinx forums
Answers to questions on Xilinx forums often go unanswered for a long time, and it forces the user to either abandon the problem or fix it by themselves, and can prove to be devastating for projects that need to be delivered on time to end users.

The next section concludes the report by providing suggestions for future improvements.

9. Retrospective, Conclusion, Suggestions and Comments

In this project, we implemented an image recognition system using Caffe on a Virtex-7 FPGA using the SDAccel platform from Xilinx. During this process, many topics introduced in the course were exercised and various tools were utilized. The major take-away from this project course was the experience of designing large digital systems implemented on an FPGA. This project took two full academic terms of time and its completion was based on cooperation from each member. Before the integration stage, our regular meeting time used to be once a week, and during the interval between meetings we also had regular discussions and debates through e-mails and Skype. After each individual module's completion, we started to meet more frequently during the integration stage (two or three times each week). Even though our team was diversified and our members were engaged in either research, work or study, everyone made efforts in completing the project and the time management was fairly effective.

During this process different types of problems were encountered. Some were related to the understanding of the usage of a software tool for a specific project while others were related to the integration of different hardware block interfaces. Although many problems were addressed by acceptable responses, in retrospect, there are lessons and suggestions to learn for future projects. In addition to better time management and project scope definition, we believe that we would have been able to achieve a lot more had the tools been bug-free, or quick fixes provided by Xilinx. Thus, in addition to providing fixes for the above for future tool editions, we have outlined additional items which would enhance the scope of the project and also benefit anyone who would wish to work upon it in the future. We anticipate that future work in this project would focus mostly on achieving higher performance for the FPGA implementation of the Alex net model in comparison to the CPU implementation. Based on this assumptions, suggestions for scope improvement would include:

- Increasing throughput/reducing latency of individual layers which can be done by porting the HLS kernels into smaller kernels by breaking down the computational tasks involved.
- Programming multiple kernels simultaneously on the FPGA reducing the fixed time overhead associated with downloading the hardware on the Alpha data card.
- Streaming multiple images through the FPGA implementation to reduce the fixed overhead for programming the board per image.
- Port the entire Alex net model in HLS kernels which is currently only 70 %.
- Enhancing the parallelism in hardware by breaking down larger HLS kernels into smaller OpenCL kernels.
- Deploying multiple instances of the entire network (all layer instances in a network) over a network on FPGA boards and stream multiple images through the entire network. (Completely eliminates the programming overhead and provides maximum parallelism).

- Conversion of Caffe using C++ AMP/Thrust to allow the user to leverage the compute power of Amazon or Microsoft FPGA's on the cloud.

.

10. References

[1] http://caffe.berkeleyvision.org/tutorial/layers.html

[2] https://jblkacademic.wordpress.com/2015/07/15/alexnet-visualization/

[3] http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=7011421&punumber%3D7008946%26sortType%3Dasc_p_Sequence%26filter%3DAND(p_IS_Number%3A7011360)%26pageNumber%3D3

[4] http://www.diva-portal.org/smash/get/diva2:651974/FULLTEXT01.pdf

[5] http://www.nlpca.org/fig_pca_principal_component_analysis.png

Appendix A - Testbench example (portion calling the kernel)

```
  }

  // Write our data set into the input array in device memory
  //
  err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0, sizeof(float)*IWIDTH*IHEIGHT, img, 0, NULL, NULL);
  if (err != CL_SUCCESS)
  {
    printf("Error: Failed to write to source array img!\n");
    printf("Test failed\n");
    return EXIT_FAILURE;
  }


  // Set the arguments to our compute kernel
  //
  err = 0;
  err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
  err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
  if (err != CL_SUCCESS)
  {
    printf("Error: Failed to set kernel arguments! %d\n", err);
    printf("Test failed\n");
    return EXIT_FAILURE;
  }

  // Execute the kernel over the entire range of our 1d input data set
  // using the maximum number of work group items for this device
  //

#ifdef C_KERNEL
  err = clEnqueueTask(commands, kernel, 0, NULL, NULL);
#endif
```

Appendix B - Kernel file example

```
void pool1_max_float(float *in, float *out) {
  #pragma HLS INTERFACE m_axi port=in offset=slave bundle=pool_gmem
  #pragma HLS INTERFACE m_axi port=out offset=slave bundle=pool_gmem
  #pragma HLS INTERFACE s_axilite port=in bundle=pool_control
  #pragma HLS INTERFACE s_axilite port=out bundle=pool_control
  #pragma HLS INTERFACE s_axilite port=return bundle=pool_control

  float inbuf[IWIDTH*IHEIGHT], obuf[OWIDTH*OHEIGHT];

  //memcpy(inbuf, (float *)in, IWIDTH*IHEIGHT);

  int i;
  for (i=0; i<IWIDTH*IHEIGHT; i++)
      inbuf[i]=in[i];

  float m;
  int idx;
  int count = 0;
  int col, row, pcol, prow;

OUTER_MAX_DATA_LOOP:for (row = 0; row < IHEIGHT - NUM_MASK_ROWS + 1; row += STRIDE) {

    #pragma HLS PIPELINE
    #pragma HLS loop_tripcount min=26 max=26
    INNER_MAX_DATA_LOOP:for (col = 0; col < IWIDTH - NUM_MASK_COLS + 1 ; col += STRIDE) {
          #pragma HLS loop_tripcount min=26 max=26
                        m = -FLT_MAX;
                        idx = -1;
        OUTER_MAX_MASK_LOOP:for (prow = 0; (prow < NUM_MASK_ROWS && row + prow < IHEIGHT); ++prow) {
            #pragma HLS loop_tripcount min=3 max=3
            INNER_MAX_MASK_LOOP:for (pcol = 0; (pcol < NUM_MASK_COLS && col + pcol < IWIDTH); ++pcol) {
                #pragma HLS loop_tripcount min=3 max=3
                                //printf("INBUF: %f\n", inbuf[IDX2C(row+prow, col+pcol, IHEIGHT)]);
                              if (inbuf[IDX2C(col + pcol, row + prow, IWIDTH)] > m) {
                               idx = IDX2C(col + pcol, row + prow, IWIDTH);
                                m = inbuf[idx];
                                  }
                            }
                        }
         obuf[count] = m;
         count++;
    }
```

Appendix C - Run file (.tcl) example

```
# Define the project for SDAccel
create_project -name prj_ocl_pooling  -dir . -force
set_property platform vc690-admpcie7v3-1ddr-gen2 [current_project]

# Host Compiler Flags
set_property -name host_cflags -value "-g -Wall -D FPGA_DEVICE -D C_KERNEL" -objects [current_project]

# Host Source Files
add_files "main.c"
add_files "pool1_max_float.h"
set_property file_type "c header files" [get_files "pool1_max_float.h"]

# Kernel Definition
create_kernel pool1_max_float -type c
add_files -kernel [get_kernels pool1_max_float] "pool1_max_float.c"

# Define Binary Containers
create_opencl_binary -device [lindex [get_device "fpga0"] 0] pool1_max_float
set_property region "OCL_REGION_0" [get_opencl_binary pool1_max_float]
create_compute_unit -opencl_binary [get_opencl_binary pool1_max_float] -kernel [get_kernels pool1_max_float] -name ocl_pooling

# Compile the design for CPU based emulation
compile_emulation -flow cpu -opencl_binary [get_opencl_binary pool1_max_float]

# Run the compiled application in CPU based emulation mode
run_emulation -flow cpu -args "pool1_max_float.xclbin"

report_estimate

# Compile the application to run on the accelerator card
build_system
#
# Package the application binaries
package_system
```

Appendix D - Log file example

```
+-----------------------+------------+-----------------------+----------------+---------------+
| pool1_max_float       | c          | fpga0:OCL_REGION_0    | pool1_max_float | 1            |
+-----------------------+------------+-----------------------+----------------+---------------+
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
OpenCL Binary = pool1_max_float

Kernels mapped to = clc_region

Timing Information (MHz)
+---------------------+-------------------------+-----------------------+-------------------------+
| Compute Unit        | Kernel Name             | Target Frequency      | Estimated Frequency     |
+---------------------+-------------------------+-----------------------+-------------------------+
| ocl_pooling         | pool1_max_float         | 166.945               | 190.84                  |
+---------------------+-------------------------+-----------------------+-------------------------+

Latency Information (clock cycles)
+---------------------+-------------------------+----------------------+----------------------+----------------------+----------------------+
| Compute Unit        | Kernel Name             | Start Interval       | Best Case            | Avg Case             | Worst Case           |
+---------------------+-------------------------+----------------------+----------------------+----------------------+----------------------+
| ocl_pooling         | pool1_max_float         | 9028                 | 9027                 | 9027                 | 9027                 |
+---------------------+-------------------------+----------------------+----------------------+----------------------+----------------------+

Area Information
+---------------------+-------------------------+-----------+-----------+-----------+-----------+
| Compute Unit        | Kernel Name             | FF        | LUT       | DSP       | BRAM      |
+---------------------+-------------------------+-----------+-----------+-----------+-----------+
| ocl_pooling         | pool1_max_float         | 9470      | 28521     | 2         | 10        |
+---------------------+-------------------------+-----------+-----------+-----------+-----------+
-------------------------------------------------------------------------------
INFO: [SDAccel 60-238] Compiling host...
INFO: [SDAccel 60-239] Compiling host...COMPLETE
INFO: [SDAccel 60-120] Building system...
INFO: [SDAccel 60-251]   Hardware accelerator integration...
INFO: [SDAccel 60-249]   Generating OpenCL binary...
INFO: [SDAccel 60-264] Building system...COMPLETE
INFO: [SDAccel 60-351] Xilinx binary containers created for vc690-admpcie7v3-1ddr-gen2 can only be executed on ADM-PCIE-7V3. These binaries will not execute on any othe
r board.
build_system: Time (s): cpu = 09:10:40 ; elapsed = 03:14:38 . Memory (MB): peak = 371.109 ; gain = 0.000 ; free physical = 5991 ; free virtual = 46473
INFO: [SDAccel 60-267] Packaging for PCIe...
INFO: [SDAccel 60-268] Packaging for PCIe...COMPLETE
INFO: [Common 17-206] Exiting sdaccel at Fri Aug 14 23:58:42 2015...
```