Literature Survey & Hardware Acceleration for Convolution Neural Networks

Yao Sun
School of Electrical and
Computer Engineering
University of Toronto
Toronto, Ontario, Canada
Email: yaosun.ys@gmail.com

***Abstract*—This purpose of this paper is two-fold, I first survey conventional technologies related to the hardware acceleration of Convolution Neural Networks and Neural Computing in general on FPGAs, beginning by giving a brief but somewhat thorough view into the operations of a Neural Network and Convolution Neural Network, then focusing on implementation issues of said networks on Field-Programmable Gate Arrays (FPGAs). Afterwards, a simplified version of a baseline processor is constructed based on Convolution Neural Networks (shortened to convnets). Based on this model, a Field-Programmable Gate Array hardware accelerator is constructed.**

I. Introduction

Neural Networks form the basis of many technologies utilized today. Most conventional search engines are driven in some form by neural networks [1], and neural networks see utilization in many different topics such as face and location recognition, natural language processing, subject identification and classification. Neural Networks such as these belong to a design space called Deep Neural Networks, deep referring to the numerous non-linear hidden layers between program input and output. These non-linear relationships allow Neural Networks to learn complicated and non-trivial relationships between the I/O of the program, allowing them to solve complex and generic problems.

One popular example of this is the contrast between conventional face detection technology such as Viola-Jones versus Convolution Neural Networks [2]. Research of this topic has shown that the simple binary nature of Haar feature classifiers used in Viola-Jones does not permit for easy detection of more subtle and varied expressions or unexpected lighting. Additionally, Haar features are hand crafted, and require human assistance in designing, while the more generic nature of convnets effective allows the neural network to train (or design) its own set of features given a correct dataset as shown by Li et al. [2]. The ability to identify more complex relationships such as these make Deep-Learning networks enticing as they potentially solve a larger set of problems than previously possible.

The limitations of Deep-Learning networks such as convnets have always been computation bound. The current trend of applying Deep-Learning networks to conventional programming problems arises due from the decrease in computation cost to a point where commercial applicability is at an acceptable level. Extrapolating from this, as computing cost reduces further, neural networks will become to focal point in an increasing amount of systems. Thus the primary issue is solving or reducing the issue of computational cost to an acceptable level. This paper will offer the following contributions.

- Introduce the core concepts behind Convolution Neural Networks, including the biological inspirations, the make-up of a *neuron* and backpropagation. This serves as the backdrop for future discussions.
- Survey conventional FPGA based acceleration algorithms, their focuses, and the issues behind FPGA based Convolution Neural Network implementations.
- Present a novel FPGA based hardware acceleration design that improves upon some of the papers surveyed.

II. Background

Convolution Neural Networks (convnets) are biologically inspired variants of Multilayered Perceptrons, more conventionally known as Neural Networks. It was found that the visual cortex exploits the spatial properties of two dimensional inputs to reduce the amount of connectivity between neurons [3]. This discovery by Hubel and Wiesel coincided with research on the Perceptron a predecessor to Convolution Neural Networks. Originally constructed to solve problems related to 2-D data sets for problems related to images, speech and text, a convnets defining feature is the scarcity in which neurons are connective related to Neural Networks. Although generic NNs can also handle these fields of problems, LeCun et al. [4] point out that generic Neural Networks are ill equipped to deal with 2-D data sets. For example, image representations commonly contain hundreds if not thousands of data points, a 100*100 input refers to 10,000 connections. This leads to overfitting issues if training data is scarce and the memory requirement makes this approach unattractive [4].

Convolution Neural Networks also exploit the spatial correlation of the input topology, by forcing neurons to only view local (a small portion) of the input data. This allows convnets to first extract *local* features before being utilized to build spatial or temporal objects [4]. Altogether, the implied spatial invariance, sparse connectivity and focus on localized input make Convolution Neural Networks attractive for 2-D data sets with spatial correlation such as images, text, speech.

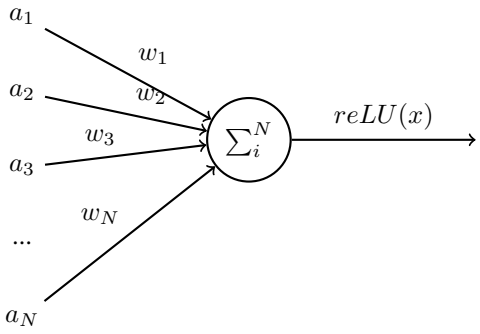

Fig. 1. Mathematical model of a neuron

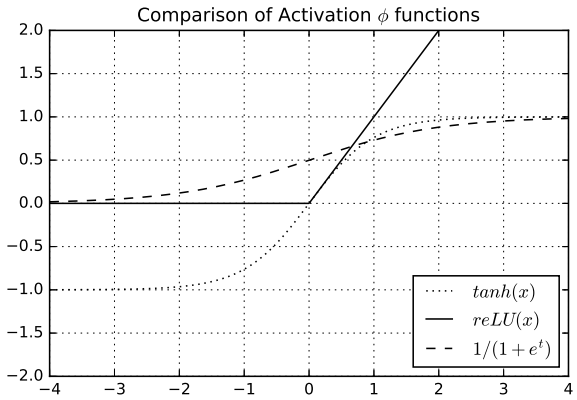

Fig. 2. Comparison of activation functions

III. Theoretical Model

Convolution Neural Networks are composed of building blocks called Neurons (depicted in figure 1), inspired by their organic variants, Neurons essentially compute a multiply-add function. Furthermore, neurons *activate* based on a non-linear function, that maps the n inputs to one output [4], [2], [5]. Defining I as the input function for each individual neuron, where w_i and a_i represent the ith coefficient and input value respectively, b represents the bias of the neuron, and N represents the total amount of inputs:

$$I(c, a) = b + \sum_{i=0}^{N} w_i * a_i \tag{1}$$

Likewise, the output function ϕ is simply a non-linear mapping of the summed input value to an output:

$$\phi(I) = f_{nonlinear}(I) \tag{2}$$

Activation functions are typically designed to mimic their organic counterparts, with the previously (and still widely used) choice being a sigmoid function, the most popular of which is $tanh(x)$. However, sigmoid functions are notoriously slow to train, as backpropagation algorithms require the derivative of said functions, leading to infinitely small deltas as the functions are positively or negatively saturated [5]. Instead, Alex et al. [5] propose the rectified linear unit function $reLU(x)$ of the form:

$$reLU(x) = max(0, x) \tag{3}$$

This function offers faster training as it yields a constant derivative, as opposed to sigmoid function (depictions in figure 2). With respect to hardware, $reLU$ functions are also significantly easier to implement, as opposed to a sigmoid function.

Typically in a generic Neural Network, each *layer* of the network would be composed of an array of neurons. Each neuron would be fully connected to the previous layer (or the input). Additionally, each connection or *synapse* would have a unique weight w_i attached to it. From this generic fully connected network model, one can derive a convolution neural network. Therefore given an input of N units, and a hidden layer of N neurons, there would be $N * N = N^2$ synapses between them and N^2 unique weights, as explained by Haykin et al. [6], and as discussed previously. In contrast to this, from the works of LeCun et al. one can see that Convolution Neural Networks are allowed to be sparsely connected through three mechanisms: locality, shared weights and subsampling [4].

The issue of connectivity is resolved by having each neuron only be connected to a local area, in the example depicted by LeCun et al. [4] this was limited to a $5 * 5$ field. Recall that that the nature of unique weights in a general Neural Network also led to scalability issues with respect to memory. In a convolution neural network, a plane of neurons instead shared a weight vector, this is called a *feature map* and it is common for a convolutional layer to consist of several feature maps. This in turn solves the issue of spatial invariance, as the calculations for a feature map can be considered a convolution of the feature onto the input. Thus an alternative name for a set of feature maps is a convolution layer. Logically, each convolution layer can be considered a 3-D construct.

Mathematically, one can depict a single convolution layer as a set of weights from each feature maps, and a set of neurons from each feature map. Let us denote:

$$\Omega = \{\Omega_1, \Omega_2 ... \Omega_M\} \tag{4}$$

Be the set of all feature maps containing a $l*l$ matrix of values and let $Omega_k$ be one particular feature map from the set.

$$\Omega_k = \begin{bmatrix} w_{11} & \dots & w_{1x} & \dots & w_{1l} \\ \dots & \dots & w_{yx} & \dots & \dots \\ w_{l1} & \dots & w_{lx} & \dots & w_{ll} \end{bmatrix} \tag{5}$$

Likewise, let us define:

$$\Gamma = \{\Gamma_1, \Gamma_2 ... \Gamma_M\} \tag{6}$$

As the set of neurons, one layer per feature map. If one assumes that *stride=1*, then there should be as many neurons as there is input units. Therefore assuming the input is of size $n * n$, so too is the size of Γ_k, then let us define $\gamma_{k,ij}$ one

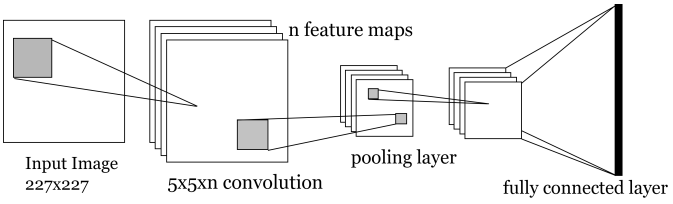

Fig. 3. Possible convnet implementation

individual neuron in Γ_k. Thus, the value of each neuron is then:

$$I_{l*l,ij} = 5 * 5 \; portion \; of \; input \; at \; coordinate \; ij \tag{7}$$

$$\gamma_{k,ij}(I) = f_{nonlinear}(sum(\Omega_k \circ I_{l*l,ij})) \tag{8}$$

Thus each neuron on the feature map is the non-linear mapping of the summation of the hadamard multiplication of the input and weight matrices for that particular feature map [4].

Pooling is also utilized by convnets to reduce the amount of data input to later layers of the network, as well as reduce sensitivity to distortion and translation [4]. The topic of subsampling is well researched in the areas of digital image processing, thus I avoid discussing it in detail. However the general model is a mapping from $l*l$ matrix to a smaller $m*m$ where $l\%m = 0$. Where each of the units on the smaller matrix is either the maximum or average over an area on the larger matrix.

Generally, a convnet consists of layers of convolution and pooling layers with a fully connected classifier layer at the end of the network (depicted in fig 3), the output of which is a vector of data useful to the user. For a classifier network, this could be the probability of an image belonging to a certain class [4].

The final layer of a neural network is a fully connected network, otherwise known as a *classifier* as referenced by Tianshi et al. [7], it may be a single or multiple layers (in which case it is identical to the *Multilayer Perceptron*) the role of which is a mapping between the output of the feature maps to useful info such as probability.

Within the context of Hardware Implementations, back propagation is usually considered to be an offline process, most implementations use pre-trained data with small amounts of fine tuning, usually with software networks. This is done before being uploaded onto hardware. Most research dedicated to acceleration lies in forward propagation as indicated by Tianshi et al. [7], for the sake of completeness, I briefly mention back propagation. Back-propagation, learning is the training of weight values within a neural network. Specifically, it refers to the training of feature maps in Convolution Neural Networks [8]. Omondi et al, the method used is gradient descent, in which the gradient of a vector is used to determine the correct *direction* of approach to the nearest minimum. presents the simplest and clearest explanation. Let Ω^{new} be the new weight value of the set Ω in a feature map and let Ω^{old} be the old or current value. Then:

$$\Omega^{new} = \Omega^{old} + \epsilon\delta x \tag{9}$$

$$\delta = (d - y)\Gamma'(I) \tag{10}$$

$$(x, d) = (x_i, d_i) \tag{11}$$

Where $y = \Gamma(I)$ is the set of all output vectors from a convolutional layer. (x, d) represent the training data, composed of desired inputs and outputs. ϵ is a constant with bounds $0 < \epsilon < 1$ represents the learning factor [8]. Thus the idea is for weight values to converge to a desired threshold. One can again reiterate the importance of $reLU(x)$ function, as a constant derivate lends well in allowing each weight to *progress* in learning, sigmoid functions tend to saturate and can lead to overfitting due to insufficient data or training time.

IV. Hardware & GPU Implementation Survey

I examine papers from Chen et al. [7], who focused on the implementation of convnets with an emphasis on constrained memory allocation and working space within a hardware context. I contrast this with a more traditional GPGPU implementation by Kriezhevsky et al. [5] whose paper was published earlier, and brought upon several algorithmic changes to neural network implementations. For a more historical perspective on the progression of hardware based accelerators (affectionately termed *Neurocomputers*) and some of the core exploits related to hardware accelerated Neural Networks, I explore the works of Omondi et al. [8]. Numerical precision is important as most computations done in neural networks involves floating point operations, reduction in precision is explored by Gupta et al. [9], to examine the effects of limited numerical precision on error rate and training time.

The design space for hardware implementation is large. Most implementations such as Kriezhevsky et al. [5] focus on GPU implementations, and several popular implementations such as Caffe [10] and Matconvnet [11] as well as followed suit. I hypothesis this to be a result of ease of implementation and the natural mapping between the parallel nature of GPUs to the parallel nature of feature maps and neurons. Additionally, convnets consist almost entirely of floating point calculations, of which FPGAs may be ill suited for. I start with how each implementation explores the inherent parallelism presented by convnets. From Omondi et al. [8], there are several points of parallelism that map well to FPGAs, of which I explore:

- *Architectural/Neuron parallelism* - Parallel nature of individual neurons, perhaps most important in increasing throughput if exploited. Difficult to map onto FPGA due to volume of neurons.
- *Bit-level parallelism* - Accelerations specific to implementation of computation units, multipliers etc.

Chen et al. [7] indicate the primary issue with FPGA based neural network implementation is related to the amount of neurons mappable onto a chip. With a $32x32$ computation layer costing $2.66m^2$ making simulating a large network inadmissible. Therefore accelerators in FPGAs mainly deal

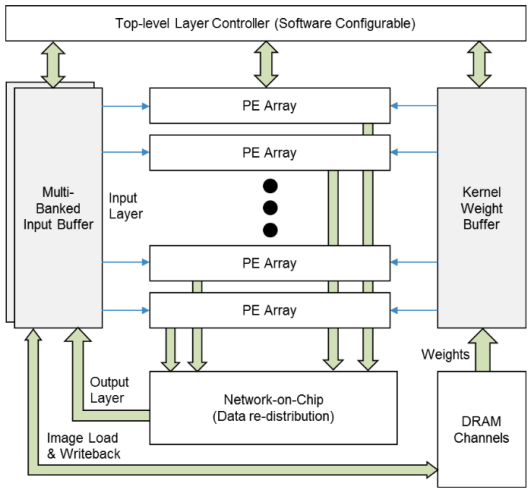

Fig. 4. Microsoft's extendable neural network accelerator [1]

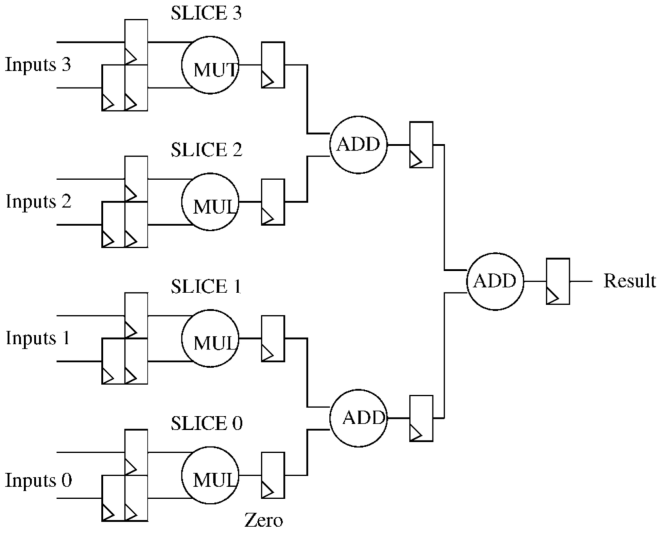

Fig. 5. Tree configuration allowing for parallel computation of neuron outputs, from [8]

with the design space of computational gain vs. memory. I first survey the techniques and hardware utilized by researchers to obtain results.

A. Methodology

With respect to hardware, focus has been placed upon accelerating the feed forward aspect of convnets. Training is usually done a priori on either a GPU implementation [5], or a SIMD CPU implementation [12], [13], [2]. The resulting weights are then loaded onto the hardware accelerator.

The works of Tianshi et al. [7] utilize an accelerator simulator in software (used for architecture exploration), and served as the basis of later verilog implementation. Such a workflow is closely related to my current workflow (discussed later). For measurements of critical path delay, area and energy, Synposys Design Compiler and ICC Compiler were utilized.

In the works of Ovtcharov et al. [1], the team worked atop of Microsoft's Catapult project (a network for FPGAs designed to handle high throughput systems such as Microsoft Bing). Microsoft realized their design on hardware, utilizing a *Catapult FPGA card*, which consist of a mid range Stratix V D5 FPGA, and 8GB of DDR3 RAM. The team targeted one of the most complex models: AlexNet. Training was done offline utilizing Project ADAM, and the team plans on updating to newer Arria 10 FPGAs in a future project.

Chen et al. [12], utilized the Vivado suite for the majority of their design flow. With focus on loop unrolling, the team utilized Vivado HLS to generate the verilog design. Furthermore, the chip was implemented onto a Xilinx FPGA chip, consisting of an AXI4Lite bus for control communication, and AXI4 bus for data transfer and a soft processor utilized for control signals. The board itself was a VC707, consisting of a Virtex7 FPGA chip. This was in comparison to an Intel Xeon 2.20 GHz SIMD processor.

B. Bit-level parallelism

The major advantage of a hardware based implementation approach is the ability to transform data efficiently through parallel bit-level mechanisms. One implementation suggested by Omondi et al. [8] suggests utilizing the DSP48 slices available in a Xilinx Virtex-4 FPGA device (at the time state-of-the-art). I recall that convnets forward propagation mainly consists of multiply, add and multiply-add computations that DSP48 slices are specialized for. One can also configure the slices as indicated in figure 5, for parallel processing. This approach is somewhat replicated by Chen et al. [7], as they propose a similarily designed

Prior research also suggests that one may reduce memory and space usage by reducing precision to 16-bits for weight and 8-bits for activation function [8], the findings of Chen et al. is consistent with this view [7]. This idea is shared by Chen et al., whose results indicate the sufficiency of a 16-bit fixed-point representation [7], when applied to the MNIST benchmark. In addition to this, the power and area characteristics of 16-bit fixed point far outweigh the tradeoff in precision: $1309.32\mu m^2$ vs $7997.76\mu m^3$, $576.90\mu W$ vs $4229.60\mu W$. Representing around a 400% decrease in area and 700% decrease in power usage. In an algorithmic context, Gupta et al. [9] explore the effects of rounding with limited numerical precision, finding that rounding plays an integral role in training error in back propagation algorithms. At a fixed word length $WL = 16$, having a float length of 8 and rounding using *Round to nearest* produced significantly increased test errors (1.5%) when compared to *Stochastic rounding*.

C. Architectural Parallelism

The parallel nature of multiply-add in calculating neuron values lend well for high throughput systems, most approaches consist of a high throughput parallel multiply-add-accumulate

hardware alongside block ram and a control unit. One novel approach is suggested by Girau & Nancy [8], coined *Field Programmable Neural Arrays* and *Field Programmable Neural Networks* which consist of configured FPNAs. Borrowing from graph theory, FPNAs are viewed as directed graphs $(N, \epsilon),\ N = nodes,\ \epsilon = edges$

$$Pred(n) = \{p \in N | (p, n) \in \epsilon\} \tag{12}$$

$$Succ(n) = \{s \in N | (n, s) \in \epsilon\} \tag{13}$$

Where $Pred(n)$ defines the set of predecessor nodes and $Succ(n)$ likewise defines the set of successor nodes. An additional set: N_i is defined as the set of input nodes. For each neural array, there is a set of operators $\alpha_(p, n)$ for each edge in ϵ and a set of activators with f_n being the activation function (same as $f_{nonlinear}$ previously defined). The FPNN is then responsible for defining the operator $\alpha_(p, n)$, in addition to a binary function $r_n(p)$, determining if a connection between $Pred(n)$ and n exists. Likewise a $S_n(s)$ applies the same methodology to successors. Essentially, tasks are passed as input values like messages, received by each node. Depending on the configuration and contents of the task different operations can be conducted through $\alpha_(p, n)$. This fits within the context of neural networks and is essentially a generically defined version of a neural network, as indicated by Girau & Nancy, it's evidence of a more generic parallel computing paradigm [8]. While implementation issues are not specified in detail, it is mentioned that fan-in of such a structure (especially considering the fully connected nature of FPNAs) become an issue. The paper does however introduce the transformation from the naive neuron based theoretical model (presented in this paper) to a different form a separation (in this case layers) and more importantly a rigorous graph model of neural networks.

The works of Kriezhevsky et al. [5] famously introduced the concept of rectified linear units (reLUs), that has since revolutionized neural network training. Through their research, it was found that on a four layer convnet reLUs were able to achieve a 25% training error rate in one sixth of the time when compared to a conventional sigmoid function ($tanh(x)$). The paper focused on what would later become known as AlexNet, a convnet designed for training and classification of a large throughput of data (approx. 1.2 million images). Kriezhevsky et al. explain that even with a large quantity of data being processed, the convnet was still prone to overfitting. The general procedure for dealing with this is simple augmentation of input images to create the amount of data available for the network [5]. These include translation, rotation, and stretching of the input image. The team also employed more complex augmentations such as finding the principle component of each image is summed with the principle components multiplied by a gaussian random variable and the eigenvalue corresponding to the eigenvector [5]. Another facet of improvement with respect to training accuracy is dropout, a recently introduced technique that sets the output of a hidden neuron to zero with

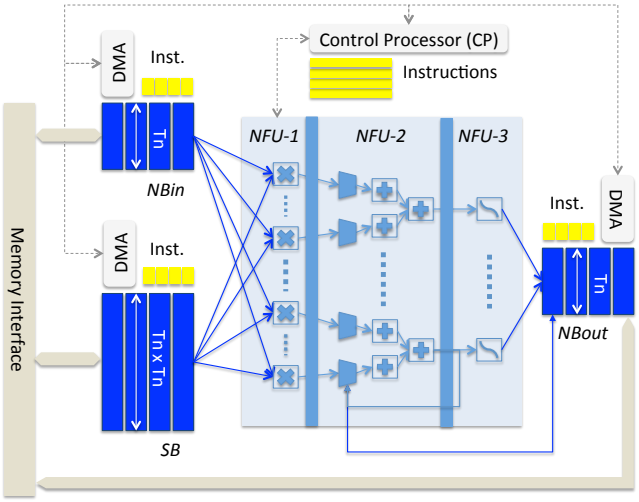

Fig. 6. Architectural layout of DianNao, a convnet implementation for FPGAs designed for high throughput and low footprint. From [7]

a 50% probability. This allows a single neural network to *emulate* different neural networks, and removes what Krizhevsky et al. calls [5] co-adaptation of neurons, in which neurons become dependent on other neurons. Generally, the focus is on algorithmic architectural changes, with the focus on increasing training speed and reducing overfitting for a large convnet (the largest layer containing 254,440 neurons). Krizhevsky et al. [5] were able to achieve a top-1 and top-5 error rate of 37.5% and 17.0% respectively, nearly 10% lower then that of the state-of-the-art.

In a paper dedicated more to hardware implementation, a Microsoft team of Ovtcharov et al. outlines an architecture (presented in figure 4) consisting of a set of programmable processing arrays. The input image is streamed on-chip from the DRAM and stored into the input buffer, the input is then streamed simultaneously into each PE (vague, but presumably each PE represents one feature map) which computes the 3-D convolution layer. Similar to the architecture presented by Chen et al. [7] it contains a software based control unit. The team notes comparable performance to cutting edge Caffe deployments on high-end GPGPUs noting the difference in power usage $235W\ vs\ 25W$ for GPUs and FPGAs respectively. The team goes on the mention that the power usage combined with cost make GPGPUs impractical for production compared to FPGAs. Both configurations focus on per-layer parallelism, meaning the relationships between layers can be kept in memory. The difference being the amount of hardware available to Microsoft allows it to simulate a full convnet layer, whereas Chen et al. focus on simulating parts of a layer in sequence [7].

The common theme between several implementations such as Chen et al. [7] and Microsoft's project is utilizing FPGA's parallelization capabilities to increase the throughput of the system. Therefore discussions revolving around prop-

erly pipelining and memory access patterns become issues. While convnets have been typically considered a computation heavy problem, memory bandwidth becomes an issue as large quantities of computation units are created. The works of [12] and [7] is particularly interesting as it offers us a view into the progression of FPGA based convnet design within the span of a few years, both utilizing some form of high level synthesis, this makes both papers extremely relevant to our accelerator. Tianshi et al. [7] implemented a design consisting of a Neural Functional Unit (**NFU**), an input buffer (**NBin**), an output buffer (**NBout**), and a synapse buffer (**SB**), see fig. 6, alongside a control processor. The bulk of the computation is done through the NFU, controlled via the Control Processor through a instruction set stored in memory. Recalling figure 1, the operations related to computation of a layer in a convnet can be decomposed into three sections, represented by Tianshi et al. as **NFU-1**, **NFU-2**, **NFU-3**. NFU-1 refers to computational units dedicated to the multiplication of synaptic weights (feature maps) to input neurons. NFU-2 contains dedicated adder trees for summation of a large vector, additionally, it can serve as a max pooling functional unit. Finally, NFU-3 contains a piecewise linear interpolation of the desired activation function $f_{nonlinear}()$ stored in a small ram. All possible computation scenarios related to convnets are then covered by a combination of NFUs, with computations being done on a per-layer basis. The team observed a total area of $3,023,077um^2$, and a total power consumption of $485mW$. The team was also able to achieve a $21.08x$ reduction in energy, when compared to a SIMD baseline, the hardware accelerator was also able to achieve, on average, a $117.87x$ gain over the SIMD core.

The works of Chen et al. [12] depict an alternative design. Here, the *computation engine* roughly maps to the NFU of Tianshi et al. [7] discussed previously. barring the separation of the unit into distinct sections (**NFU-n**). Instead, a larger design space is opened regarding Computation-to-Communication ratio, and the roofline model. To optimize for data sharing and reduce the complexity of hardware formed through loop unrolling, Chen et al. categorized iterations of a loop dimension into three categories:

- *Irrelevant* - If an iterator does not appear in any access function of an array. Therefore the dimension is irrelevant to the corresponding loop.
- *Independant* - The data is completely separable along the loop dimension, two computation units are able to access data in such a space without any dependance upon each other.
- *Dependant* - Data is not separable along loop dimension. Cannot be seperated.

The idea is then to unroll in such a way such that *dependant* data accesses are minimized, emphasizing irrelevant and independent unrolling behaviour, as dependent unrolls often generation complex connection topologies (see fig 7). The proposed hardware accelerator is listed in fig 8, overall, one can see some common themes developing from the works several works, consisting of a set of input and output buffers, alongside a high throughput computational hardware and a programmable controller either on or off chip, the input and output buffers are the connected to a larger memory either on or off chip.

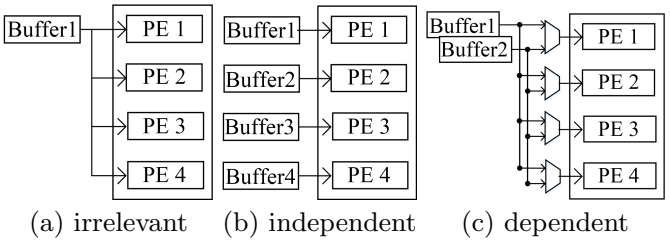

Fig. 7. Output hardware logic of unrolled loop, depending on memory access patterns, from [12]

Chen et al. [12], were able to achieve a $5.1x$ reduction in power consumption ($18.61W$ vs $95.00W$) when compared to an Intel Xeon 2.20GHz processor, and measured a $17.42x$ increase in overall throughput, ($61.62GFLOPS$ vs $3.54GFLOPS$). Although this result may be in contrast to that of Tianshi et al. [7], the differences in configurations, negligent details and other variables make comparisons between speeds difficult.

V. Implementation & Restrictions

Due to the nature of some restrictions imposed by the assignment, there were some boundaries on what was accomplishable within the context of the course. Compromises were made based on the restrictions imposed by the assignment and Verilator:

- **Lack of integrated DSPs -** Utilizing *Verilator* meant that IP Cores provided by Xilinx or any other commercial FPGA company were not applicable. As Neural Networks make heavy use of Floating Point adders and multipliers, DSPs are a popular choice within many FPGA boards, the generic nature of the assignment prevented the use of any such DSPs.
- **Lack of IP Cores -** Design lacks an integrated Xilinx interface (again owing to the lack of IP proprietary IP cores), with the lack of an onboard processor chip such as Microblaze I *emulate* the existence of off-board DRAM and through the testbench itself.
- **Yosys issues** While verilator was able to successfully provide a functional model to confirm the validity of the hardware accelerator (alongside Vivado & Vivado HLS), I was never able to successfully synthesize the circuit in Yosys. This forced me to utilize a Vivado workflow alternatively for RTL synthesis.

VI. Methodology

The methodology chosen follows closely to the assignment instructions, with a few deviations. For my implementation, I chose to implement a variant of LeNet, first presented by

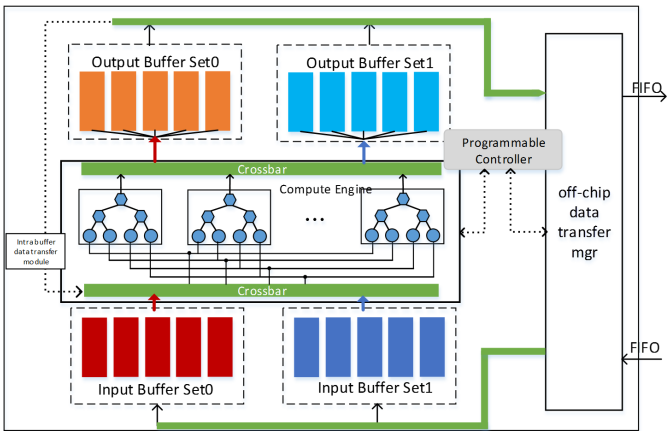

Fig. 8. Proposed accelerator hardware, note the similarities to the works of Tianshi et al., from [12]

	T_i	T_n	$stride$	$dim_{feature}$	dim_{input}
L_{conv1}	1	20	1	5*5	28*28
L_{pool}			2		
L_{conv3}	20	50	1	5*5	12*12
L_{pool}			2		
L_{conv5}	50	500	1	4*4	4*4
L_{reLU}					
L_{conv7}	500	10	1	1*1	1*1

Fig. 9. Description of layer configuration for proposed convnet

LeCun et al. [4] in 1995, the details of this neural net is listed in figure 9. I also utilize the very popular MNIST dataset to train our network. Training is done a priori on a MATLAB & CUDA implementation of neural networks called matconvnet [11]. I utilized two separate implementations of LeNet, the first of which which is coded in C, and can be compiled using gcc/g++ serving purely as a software baseline. The second implementation (cnn-synth), also contains C++ source files intended for Vivado HLS.

For the implementation, I targeted a Xilinx Kintex7 board. For functional verifications, I utilized both CSIM provided by Vivado HLS, as well as Verilator. Measurements of area, time and critical path were primarily done using Vivado, as Yosys repeatedly segmentfaulted. I suspect this to be deficiencies in the generated verilog files by Vivado HLS or due to my circuit being too large in design.

On the software side, the algorithmic implementation of the convnet was benchmarked on a single-threaded Intel Core **i7-3770k** without SIMD.

A. Project Setup & Utilization

Here I follow closely to the tutorial guidelines, commands such as **make csyn**, **make view**, all work as intended (of course under the assumption that **site.mk** is properly set). Under the **tb** folder, the testbench is designed to run through all test images associated with the MNIST data. A full run-through takes approximately 12-14 hours. Commands related to YOSYS are also available, however I haven't been able to move past the segment faults. Under `cnnalgo`, contains several header files and a `main.c`, this contains a floating point software implementation of the convnet eventually hardware accelerated. This folder contains a separate makefile, independent of the `common.mk` build system, running make will generate a `main.out` that reports computation time.

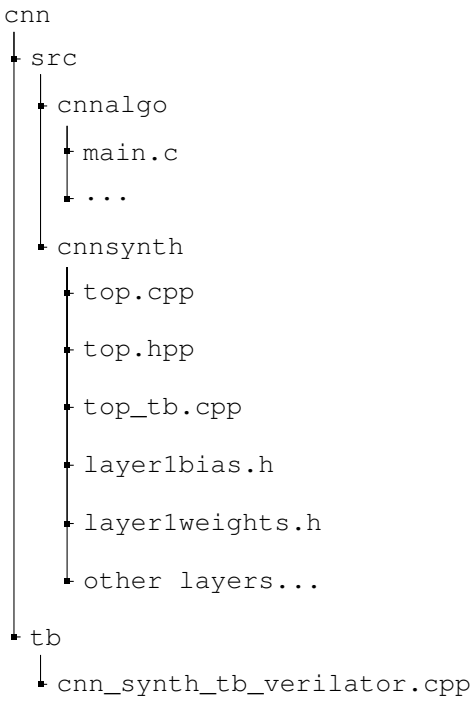

Fig. 10. Project file structure, with most important files listed

Within the `cnnsynth` folder, `top.cpp` holds the majority of the Vivado HLS hardware implementation, the various weight header files contain the weights used at each individual level. Finally a `images.h` contains the input files utilized by the Vivado testbench for **CSIM** and an identical version is stored in `/tb` for the Verilator testbench. Also included is `cnn-synth-verilog` which contains the verilog files generated by Vivado HLS. Verilator outputs are omitted here as the **.vcd** file becomes extremely large.

VII. Design

With respect to *bit-level parallelism*, we left Vivado HLS to implement the adder tree necessary for parallel computation of multipliers, specified from the unroll pragma (discussed later). Our design also incorporates elements of Gupta et al. [9], which confirmed the sufficiency of 8-bit fixed point values for input and synapse weight (feature map) representations. Our overall design is similar to that of Tianshi et al. [7] and Chen et al. [12], however my design lacks the generic nature of their designs, as I was not able to simulate a soft processor for purposes of control.

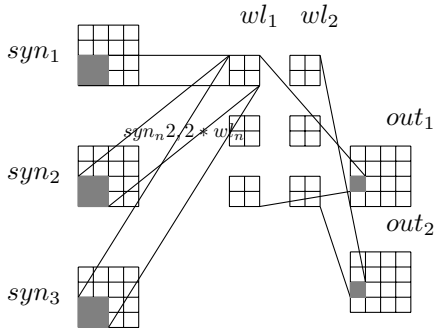

Fig. 11. Matrix representation of a convolution layer in a convnet, notice the parallels, such a configuration has $D = 3$, $D_f = 2, stride = 1$

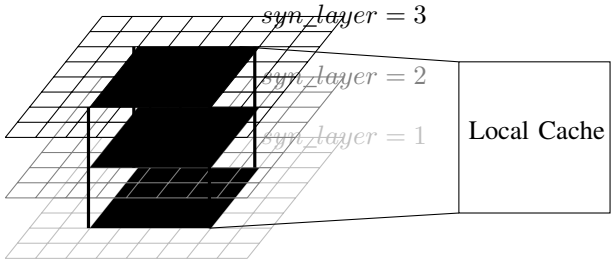

Fig. 12. Visualization of inputs to the local cache of each computational unit. The 3D matrix represents the inputs to the current layers (this is either the output image or the output of the previously layer).

The computation of a output synapse from a particular convolution layer is calculated according to figure 11 (refer to figure 14 for list of dimensions), where in our particular model $D = 3$ and $D_f = 2$, and consists of two feature maps of size $2 * 2$. From the figure, one can see that the major operations consist of several data-independant products, and a summation of all outputs, as I previously referenced using the works of Tianshi et al. [7]. Therefore, one would like to be able to keep computationally relevant data in RAM blocks closer to the computation. The idea is then to store all *layers* of the input relevant to the computation of $input * feature\ map$ into a local RAM block or cache, thereby reducing read/write quantities of the larger external RAM. With respect to figure 11, this would be collecting all inputs of $Local = syn_{n,i*j,2*2}$ for $n = 1, 2, 3$ and where i, j refer to the top-left most coordinate of the input. Figure 12 contains a visualization of data local cache stores. wl_1 and wl_2 are already stored onto ROMs close to the computation, and therefore do not have this requirement.

Within Vivado HLS, I propose a convolution functional unit `conv()`, responsible for all computations related to convolution. A pooling functional unit `pool()`, related to pooling functions, and a rectified linear unit functional unit `reLU()`. Our proposed implementation differs in that the proposed design lacks a complex control unit, therefore functions are called with a set order, and the implementation has only the ability to implement our desired convnet (specified in figure 9).

```
L3_Loop:
for (y0 = 0; y0 < N0; y0++) {
    for (x0 = 0; x0 < M0; x0++) {
        for (x = 0; x < Mf; x++)
#pragma HLS PIPELINE
            for (y = 0; y < Nf; y++)
                for (d = 0; d < D; d++)
                    L2_copy[d][y][x] =
                    L2[d][y + y0][x0 + x];

        for (d0 = 0; d0 < Df; d0++) {
#pragma HLS PIPELINE
            L3_conv(x0, y0, d0);
        }
    }
}
```

Fig. 13. Partial code of `conv()` implementation, computation of 3rd layer (convolution layer)

Figure 13 refers to our proposed convolution design, referring specifically to the 3rd layer functional unit. I first introduce the variables utilized to specify the dimensions of computation, refer to figure 14 for a list of dimension variables and their properties. From the proposed code, one can see that a local copy of the input data (either from the input image, or from the output of the previous layer) is stored in `L2_copy`, this process is pipelined for parallelization and automatic unrolling. One can see that the previous discussion of implementing a local RAM copy of the input is realized in such a fashion, therefore one can say that.

$$L2\_copy = syn_{n,i*j,5*5}\ \{n \mid n \in \mathbb{Z}, 0 <= n < 20\} \quad (14)$$

Here, $i * j$ sets the top-left coordinate of the block, n specifies the synapse layer output from the last layer, in this case L_{pool1} outputs 20 layers of outputs. And $5*5$ specifies the dimensions of the input extracted. Second, I unroll along the output dimension, creating $d_0 = D_f$ copies of `L3_conv()`. Conceptually, this would be equivalent to computing all feature maps in a particular two dimensional section of the input synapse (matrix) in parallel, specified by variables y_0 and x_0.

var	dim
D	Input feature dim
D_f	Output feature dim
N	Input feature x-dim
M	Input feature y-dim
N_0	Stride=1, pad=0, output dim
M_0	Stride=1, pad=0, output dim
M_f	Filter x-dim
N_f	Filter y-dim

Fig. 14. All dimensions and their representative variables used in design report and code

```
void L3_conv(int x0, int y0, int d0)
{
int x, y, d;
int tmp = 0;
#undef D
#undef Nf
#undef Mf

#define D    20
#define Nf    5
#define Mf    5

    for (x = 0; x < Mf; x++)
#pragma HLS PIPELINE
        for (y = 0; y < Nf; y++)
            for (d = 0; d < D; d++)
                tmp += ((L2_copy[d][y][x] *
                W3[d0][d][y][x]) >>
                FIX_POINT_SHIFT);

    L3[d0][y0][x0] = tmp;
}
```

Fig. 15. Inner computations of 3rd layer of `conv()`, by pipelining a perfect loop

```
L2_Loop:
for (int d0 = 0; d0 < D; d0++)
    for (int y0 = 0, y1 = 0; y0 < N;
    y0 += STRIDE, y1++)
        for (int x0 = 0, x1 = 0; x0 < M;
        x0 += STRIDE, x1++)
            for (int y = 0; y < STRIDE; y++)
#pragma HLS PIPELINE
                for (int x = 0;x < STRIDE;x++)
                    L2[d0][y1][x1] =
                    (L1[d0][y0 + y][x0 + x] >
                    L2[d0][y1][x1]) ?
                    L1[d0][y0 + y][x0 + x] :
                    L2[d0][y1][x1];
```

Fig. 16. Proposed implementation of a pooling layer (layer 2)

Within `L3_conv()` (see figure 15), I pipeline a perfect loop to allow Vivado HLS to parallelize and unroll the compute structure. From the works of Chen et al. [12], this allows Vivado HLS to generate a variant of the adder tree configuration previously discussed in the bit-level optimizations section. **Note:** In current versions of the code, `L3_conv()` has been merged into the main body of `conv()`, instead of being a separate function, I found reduced error rate after doing so, reasons are unclear but warrant further investigation. However I feel that a function lends well to a conceptual explanation of the hardware' so I've refrained from updating the report to match the project.

With respect to pooling layers, from figure 16, one can see that a 2D unroll is performed with respect to both the x and y dimensions. I utilize max pooling for simplistic hardware logic, in addition to not suffering further aliasing

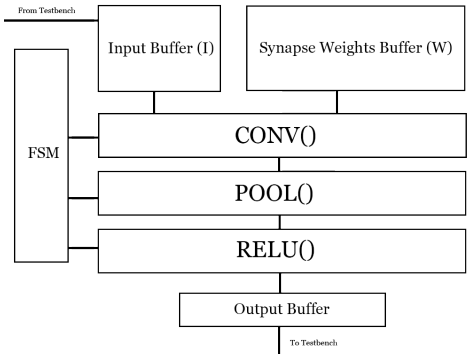

Fig. 17. Conceptual hardware layout of proposed accelerator

due to fixed point computation. The `relU()` functional unit is fairly trivial, consisting of a full unroll (500 units) with 1 to 1 mapping from D to D_f. As discussed previously in our theoretical concepts, I implemented a rectified linear unit as our activation function.

Lastly, `top()` holds the top level input and output signals as well as instantiated all lower level functions. Input to the proposed hardware accelerator is done via a 32bit signal that directly saves into a BRAM on the chip. Output is via a 32bit signal, outputting an integer that represents the **index** of the estimated output number.

Conceptually, our proposed hardware is modeled by figure 17, here the synapse weights or feature maps are stored in a local BRAM, the input ram receives data from the test bench and an on board Finite State Machine controls the operation flow of the hardware. As discussed previously, three separate computation hardwares are present `conv()`, `pool()`, `reLU()`, finally the output is stored onto an output buffer to be read by the test bench.

Metrics	**-g**	**-O3**	**HW**
$Time(s)$	99.16	15.278	$5.526 * 10^{-4}$
$Images$	10000	10000	1
Img/sec	100.846	654.547	1809.496
$Multiplier$	$1.0x$	$6.49x$	$17.49x$

Fig. 18. Results of hardware accelerator benchmarked against software implementation

VIII. Results

I examined the following aspects of the hardware accelerator: speed, utilization, latency, top-1 error rate benchmarked against a floating point software implementation.

I benchmarked the hardware accelerator against the software version of the convnet (`main.c`). For the software implementations, I compiled two versions of `main.c`, one with `-O3` optimizations enabled. On the hardware side, I calculated the cycles taken by verilator to complete one iteration of the cycle. From the results, I can conclude that the hardware accelerator design is $17.49x$ faster than an unoptimized software design,

and approximately $3x$ faster than an optimized software design, detailed information available in figure 18. For input data, I utilized the same MNIST dataset contained within `images.h`.

Utilizing Vivado, I was able to gather utilization statistics from the synthesized design. Overall, I found that utilization was within the resources given by the selected board. However, BRAM utilization was extremely high, indicating that I may have stored too many weight and input values on the board too liberally. For future works, this would definitely be a vector for improvement.

Resource	Utilization	Total	%
LUT	76822	203128	37.82
LUTRAM	1787	112800	1.58
FF	36457	406256	8.97
BRAM	530.50	540	98.24
DSP	1052	1700	61.88
IO	82	468	17.52

Fig. 19. Resource utilization on a Xilinx Kintex board

Convnets often benchmark error rate using top-1 and top-5 hit or miss rate. Interestingly enough, Vivado's CSIM and our software implementation all produced a miss rate of 98 out of 10000 input images, or a top-1 miss rate of $0.98\%s$. However, Verilator produced 154 misses, hence a miss rate of 1.54%. This indicates a 157% increase in misses utilizing the same data-set and input images. Possible reasons for this may be increased aliasing in the adder structure. This would also be an avenue for exploration in future works.

IX. Conclusions & Future Works

Through this paper, I examined the theoretical backgrounds concerning convnets, discussing theoretical models that make up the basic components of a convnet. I discussed both historical and relevant papers to the assignment, especially exciting was the conclusions of Kriezhevsky et al. [5], for advancing the state-of-the-art of convnets and the works of Chen et al. [12] for providing a workflow similar to my design. Next, I spent a great deal of time discussing both the design choices and the implementation methodology. Overall I felt the design section was a little messy, however I tried to keep the section concise and explain my major design choices. Finally, I was able to gather a set of results from Vivado HLS, CSIM, Verilator and Vivado.

Had I given myself more time I would have liked to run some design space explorations in order to obtain conclusions in depth. For future works, I would definitely like to explore optimizing the input bandwidth, utilizing AXI protocols and implementing a sophisticated control mechanism utilizing a soft processor (similar to Tianshi et al. [7]). Additionally, I would have liked to implement unroll factors and explore their effects on the pipeline.

Hardware design is very complex, and an accelerator of this scale is by far the largest undertaking attempted by myself. I knew from the onset of the project that High Level Synthesis would enable circuits of greater complexity, and allow me to focus on design detail rather than the more intricate components of the hardware. I was taught (again) by hardware design the amount of computation time involved in iterating a design. It took around an hour for Vivado HLS to generate the verilog, and around 5-6 hours for Verilator to generate test results for all input images.

Overall, I feel that the topic of convnets maps well to an FPGA structure. The flexibility in design has generated a large design space, with each report I read in the survey proposing an alternative design. I am satisfied with the implementation of the computation units for my design, however I do feel that memory locality and utilization of buffers can be improved in my design. I neglected to use some of the more complex Vivado HLS features such as FIFOs. This warrants future work, something I will definitely be looking into.

References

[1] J.-Y. K. Kalin Ovtcharov, Olatunji Ruwase, "Accelerating deep convolutional neural networks using specialized hardware," February 2015. [Online]. Available: https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/

[2] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua, "A convolutional neural network cascade for face detection," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 5325–5334.

[3] C. R. Sammer Hijazi, Rishi Kumar, "Using convolution neural networks for image recognition."

[4] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[6] S. Haykin and N. Network, "A comprehensive foundation," *Neural Networks*, vol. 2, no. 2004, 2004.

[7] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *SIGPLAN Not.*, vol. 49, no. 4, pp. 269–284, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2644865.2541967

[8] A. R. Omondi and J. C. Rajapakse, *FPGA implementations of neural networks*. Springer, 2006, vol. 365.

[9] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR, abs/1502.02551*, vol. 392, 2015.

[10] E. S. Yangqing Jia. Caffe — deep learning framework by the bvlc. [Online]. Available: http://caffe.berkeleyvision.org

[11] T. M. Team. (2014) Using gpu acceleration. [Online]. Available: http://www.vlfeat.org/matconvnet/gpu/

[12] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[13] Y. LeCun, K. Kavukcuoglu, C. Farabet *et al.*, "Convolutional networks and applications in vision." in *ISCAS*, 2010, pp. 253–256.