An architectural design of deep learning accelerator unit

K. Smily Pradeepthi  P. Aswani Kumar M.Tech, Assistant professor
1,2Eluru College of Engineering, Duggirala, Pedavegi, West Godavari, Andhra Pradesh, INDIA

Abstract: Machine learning plays an important role in the field of artificial intelligence. Deep learning shows excellent ability in solving complex learning problems than that of machine learning. But, the size of the neural networks, and demand of practical applications possess significant challenge to construct a high performance implementations of deep learning neural networks. Hence to improve the performance and maintain the low power cost, in this paper a scalable deep learning accelerator unit (DLAU), is designed for large-scale deep learning networks using field-programmable gate array (FPGA) as the hardware prototype. The proposed DLAU uses carry save adder in the computation process and as further process of the project to increase the computation speed parallel self timed adder is used in the design. Experimental results on the state-of-the-art Xilinx FPGA board demonstrate that the DLAU accelerator achieved more speed when compared to other devices like ASIC.

Index Terms—Deep learning, field-programmable gate array (FPGA), hardware accelerator, neural network.

I. INTRODUCTION

Deep learning is a specific subset of Machine Learning, which is a specific subset of Artificial Intelligence. Artificial Intelligence is the broad mandate of creating machines that can think intelligently Machine Learning is one way of doing that, by using algorithms to glean insights from data (see our gentle introduction here) Deep Learning is one way of doing that, using a specific algorithm called a Neural Network. Neural networks are inspired by the structure of the cerebral cortex. At the basic level is the perceptron, the mathematical representation of a biological neuron. Like in the cerebral cortex, there can be several layers of interconnected perceptrons. Input values, or in other words our underlying data, get passed through this “network” of hidden layers until they eventually converge to the output layer. This is explained in the next section.

Deep Learning is at the cutting edge of what machines can do, and developers and business leaders absolutely need to understand what it is and how it works. This unique type of algorithm has far surpassed any previous benchmarks for classification of images, text, and voice. It also powers some of the most interesting applications in the world, like autonomous vehicles and real-time translation. There was certainly a bunch of excitement around Google’s Deep Learning based Alpha Go beating the best Go player in the world, but the business applications for this technology are more immediate and potentially more impactful. In the 1980s, most neural networks were a single layer due to the cost of computation and availability of data. Using Deep Learning to classify and label images isn’t only better than any other traditional algorithms: it’s starting to be better than actual humans. Facebook has had great success with identifying faces in photographs by using Deep Learning. It’s not just a marginal improvement, but a game changer: Speech recognition is another area that has felt Deep Learning’s impact. Spoken languages are so vast and ambiguous. Baidu – one of the leading search engines of China – has developed a voice recognition system that is faster and more accurate than humans at producing text on a mobile phone; in both English and Mandarin. Google is now using deep learning to manage the energy at the company’s data centers. They’ve cut their energy needs for cooling by 40%. That translates to about a 15% improvement in power usage efficiency for the company and hundreds of millions of dollars in savings. Deep Learning is important because it finally makes these tasks accessible.

As a main means to accelerate deep learning algorithms, FPGA (Field-Programmable Gate Array) has high performance and low power consumption. It poses significant challenges to implement high performance deep learning networks with low power cost, especially for large-scale deep learning neural network models. So far, the state-of-the-art means for
accelerating deep learning algorithms are field-programmable gate array (FPGA), application specific integrated circuit (ASIC), and graphic processing unit (GPU). Compared with GPU acceleration, hardware accelerators like FPGA and ASIC can achieve at least moderate performance with lower power consumption.

To tackle these problems, a scalable deep learning accelerator unit named DLAU to speed up the kernel computational parts of deep learning algorithms is presented. In particular, we utilize the tile techniques, FIFO buffers, and pipelines to minimize memory transfer operations, and reuse the computing units to implement the large size neural networks. This approach distinguishes itself from previous literatures with following contributions. The DLAU accelerator is composed of three fully pipelined processing units, including tiled matrix multiplication unit (TMMU), part sum accumulation unit (PSAU), and activation function acceleration unit (AFAU). Different network topologies such as CNN, DNN, or even emerging neural networks can be composed from these basic modules. Consequently, the scalability of FPGA-based accelerator is higher than ASIC-based accelerator.

II. LITERATURE REVIEW

DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers.

As applications such as Apple Siri, Google Now, Microsoft Cortana, and Amazon Echo continue to gain traction, web service companies are adopting large deep neural networks (DNN) for machine learning challenges such as image processing, speech recognition, natural language processing, among others. A number of open questions arise as to the design of a server platform specialized for DNN and how modern warehouse scale computers (WSCs) should be outfitted to provide DNN as a service for these applications. In this paper, we present DjiNN, an open infrastructure for DNN as a service in WSCs, and Tonic Suite, a suite of 7 end to-end applications that span image, speech, and language processing. We use DjiNN to design a high throughput DNN system based on massive GPU server designs and provide insights as to the varying characteristics across applications. After studying the throughput, bandwidth, and power properties of DjiNN and Tonic Suite, we investigate several design points for future WSC architectures. We investigate the total cost of ownership implications of having a WSC with a disaggregated GPU pool versus a WSC composed of homogeneous integrated GPU servers. We improve DNN throughput by over 120× for all but one application (40× for Facial Recognition) on an NVIDIA K40 GPU. On a GPU server composed of 8 NVIDIA K40s, we achieve near-linear scaling (around 1000× throughput improvement) for 3 of the 7 applications. Through our analysis, we also find that GPU-enabled WSCs improve total cost of ownership over CPU-only designs by 4-20×, depending on the composition of the workload.

A high-performance FPGA architecture for restricted Boltzmann machines.

Despite the popularity and success of neural networks in research, the number of resulting commercial or industrial applications have been limited. A primary cause of this lack of adoption is due to the fact that neural networks are usually implemented as software running on general-purpose processors. Algorithms to implement a neural network in software are typically \( O(n^3) \) problems -- as a result, neural networks are unable to provide the performance and scalability required in non-academic settings. In this paper, we investigate how FPGAs can be used to take advantage of the inherent parallelism in neural networks to provide a better implementation in terms of scalability and performance. We will focus on the Restricted Boltzmann machine, a popular type of neural network, because its architecture is particularly well-suited to hardware designs. The proposed, multi-purpose hardware framework is designed to reduce the \( O(n^2) \) problem into an \( O(n) \) implementation while only requiring \( O(n) \) resources. The framework is tested on a Xilinx Virtex II-Pro XC2VP70 FPGA running at 100MHz. The resources support a Restricted Boltzmann machine of 128x128 nodes, which results in a computational speed of 1.02 billion connection-updates per-second and a speed-up of 35 fold over an optimized C program running on a 2.8GHz Intel processor.

III. INTRODUCTION TO NEURAL NETWORKS

The simplest definition of a neural network, more properly referred to as an 'artificial' neural network
(ANN), is provided by the inventor of one of the first neurocomputers, Dr. Robert Hecht-Nielsen. He defines a neural network as: "...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs. In "Neural Network Primer: Part I" by Maureen Caudill, AI Expert, Feb. 1989 ANNs are processing devices (algorithms or actual hardware) that are loosely modeled after the neuronal structure of the mammalian cerebral cortex but on much smaller scales. A large ANN might have hundreds or thousands of processor units, whereas a mammalian brain has billions of neurons with a corresponding increase in magnitude of their overall interaction and emergent behavior. Although ANN researchers are generally not concerned with whether their networks accurately resemble biological systems, some have. For example, researchers have accurately simulated the function of the retina and modeled the eye rather well. Although the mathematics involved with neural networking is not a trivial matter, a user can rather easily gain at least an operational understanding of their structure and function.

**Basics of Neural Networks**

Neural networks are typically organized in layers. Layers are made up of a number of interconnected 'nodes' which contain an 'activation function'. Patterns are presented to the network via the 'input layer', which communicates to one or more 'hidden layers' where the actual processing is done via a system of weighted 'connections'. The hidden layers then link to an 'output layer' where the answer is outputs as shown in the graphic below.

**IV.DLAU ARCHITECTURE AND EXECUTION MODEL**

Fig. 2 describes the DLAU system architecture which contains an embedded processor, a DDR3 memory controller, a DMA module, and the DLAU accelerator. The embedded processor is responsible for providing programming interface to the users and communicating with DLAU via JTAG-UART. In particular it transfers the input data and the weight matrix to internal BRAM blocks, activates the DLAU accelerator, and returns the results to the user after execution. The DLAU is integrated as a standalone unit which is flexible and adaptive to accommodate different applications with configurations. The DLAU consists of three processing units organized in a pipeline manner: 1) TMMU; 2) PSAU; and 3) AFAU. For execution, DLAU reads the tiled data from the memory by DMA, computes with all the three processing units in turn, and then writes the results back to the memory.

In particular, the DLAU accelerator architecture has the following key features

**FIFO Buffer:** Each processing unit in DLAU has an input buffer and an output buffer to receive or send the data in FIFO. These buffers are employed to prevent the data loss caused by the inconsistent throughput between each processing unit.

**Tiled Techniques:** Different machine learning applications may require specific neural network sizes. The tile technique is employed to divide the large volume of data into small tiles that can be cached on chip, therefore the accelerator can be adopted to different neural network size. Consequently, the FPGA-based accelerator is more scalable to accommodate different machine learning applications.

**Pipeline Accelerator:** We use stream-like data passing mechanism (e.g., AXI-Stream for demonstration) to transfer data between the adjacent processing units, therefore, TMMU, PSAU, and AFAU can compute in streaming-like manner. Of these three computational modules, TMMU is the primary computational unit, which reads the total weights and tiled nodes data through DMA, performs the calculations, and then transfers the intermediate part
sum results to PSAU. PSAU collects part sums and performs accumulation. When the accumulation is completed, results will be passed to AFAU. AFAU performs the activation function using piecewise linear interpolation methods. In the rest of this section, we will detail the implementation of these three processing units, respectively.

**Fig. 2. DLAU accelerator architecture**

**A. TMMU Architecture**

TMMU is in charge of multiplication and accumulation operations. TMMU is specially designed to exploit the data locality of the weights and is responsible for calculating the part sums. TMMU employs an input FIFO buffer which receives the data transferred from DMA and an output FIFO buffer to send part sums to PSAU. Fig. 3 illustrates the TMMU schematic diagram, in which we set tile size = 32 as an example. TMMU first reads the weight matrix data from input buffer into different BRAMs in 32 by the row number of the weight matrix \( n = i \% 32 \) where \( n \) refers to the number of BRAM, and \( i \) is the row number of weight matrix). Then, TMMU begins to buffer the tiled node data. In the first time, TMMU reads the tiled 32 values to registers Reg_a and starts execution. In parallel to the computation at every cycle, TMMU reads the next node from input buffer and saves to the registers Reg_b. Consequently, the registers Reg_a and Reg_b can be used alternately.

For the calculation, we use pipelined binary adder tree structure to optimize the performance. As depicted in Fig. 3, the weight data and the node data are saved in BRAMs and registers. The pipeline takes advantage of time-sharing the coarse-grained accelerators. As a consequence, this implementation enables the TMMU unit to produce a part sum result every clock cycle.

**Fig. 3 TMMU Architecture**

**B. PSAU Architecture**

PSAU is responsible for the accumulation operation. Fig. 3 presents the PSAU architecture, which accumulates the part sum produced by TMMU. If the part sum is the final result, PSAU will write the value to output buffer and send results to AFAU in a pipeline manner. PSAU can accumulate one part sum every clock cycle, therefore the throughput of PSAU accumulation matches the generation of the part sum in TMMU.

**Fig. 4 PSAU Architecture**

**C. AFAU Architecture**

Finally, AFAU implements the activation function using piecewise linear interpolation \( y = a_i x + b_i, x \in [x_i, x_{i+1}] \). This method has been widely applied to implement activation functions with negligible accuracy loss when the interval between \( x_i \) and \( x_{i+1} \) is
insignificant. In activation function implementation of sigmoid function is done.

V. EXTENSION

In this section, the architecture and theory behind PASTA is presented. The adder first accepts two input operands to perform half additions for each bit. Subsequently, it iterates using earlier generated carry and sums to perform half-additions repeatedly until all carry bits are consumed and settled at zero level.

A. Architecture of PASTA

The general architecture of the adder is shown in Fig. 1. The selection input for two-input multiplexers corresponds to the Req handshake signal and will be a single 0 to 1 transition denoted by SEL. It will initially select the actual operands during \( \text{SEL}=0 \) and will switch to feedback/carry paths for subsequent iterations using \( \text{SEL}=1 \). The feedback path from the HAs enable the multiple iterations to continue until the completion when all carry signals will assume zero values.

![Fig. 5. General block diagram of PASTA](image)

B. State Diagrams

In Fig. 6, two state diagrams are drawn for the initial phase and the iterative phase of the proposed architecture. Each state is represented by \((C_{i+1}, S_i)\) pair where \(C_{i+1}\), \(S_i\) represent carry out and sum values, respectively, from the \(i\)th bit adder block. During the initial phase, the circuit merely works as a combinational HA operating in fundamental mode. It is apparent that due to the use of HAs instead of FAs, state \((11)\) cannot appear.

During the iterative phase \((\text{SEL}=1)\), the feedback path through multiplexer block is activated. The carry transitions \((C_i)\) are allowed as many times as needed to complete the recursion. From the definition of fundamental mode circuits, the present design cannot be considered as a fundamental mode circuit as the input–outputs will go through several transitions before producing the final output. It is not a Muller circuit working outside the fundamental mode either as internally; several transitions will take place, as shown in the state diagram. This is analogous to cyclic sequential circuits where gate delays are utilized to separate individual states.

![Fig. 6. State diagrams for PASTA](image)

(a) Initial phase (b) Iterative phase

C. Recursive Formula for Binary Addition

Let \(S_{ji}\) and \(C_{ji+1}\) denote the sum and carry, respectively, for \(i\)th bit at the \(j\)th iteration. The initial condition \((j = 0)\) for addition is formulated as follows

\[ S_i^0 = a_i \oplus b_i \]

\[ C_{i+1}^0 = a_i b_i. \]

The \(j\)th iteration for the recursive addition is formulated by

\[ S_i^j = S_i^{j-1} \oplus C_i^{j-1}, \quad 0 \leq i < n \]

\[ C_{i+1}^j = S_i^{j-1} C_i^{j-1}, \quad 0 \leq i \leq n. \]

The recursion is terminated at \(k\)th iteration when the following condition is met:

\[ C_n^k + C_{n-1}^k + \cdots + C_1^k = 0, \quad 0 \leq k \leq n. \]

Now, the correctness of the recursive formulation is inductively proved as follows.

Theorem 1: The recursive formulation of \((1)-(4)\) will produce correct sum for any number of bits and will terminate within a finite time.

Proof: We prove the correctness of the algorithm by induction on the required number of iterations for completing the addition (meeting the terminating condition).
Basis: Consider the operand choices for which no carry propagation is required, i.e., \( C_{0_i} = 0 \) for \( \forall i, i \in [0..n] \). The proposed formulation will produce the correct result by a single-bit computation time and terminate instantaneously as (4) is met.

Induction: Assume that \( C_{k+1} \neq 0 \) for something bit at kth iteration. Let 1 be such a bit for which \( C_{1_{+1}} = 1 \). We show that it will be successfully transmitted to the next higher bit in the \((k+1)\)th iteration.

As shown in the state diagram, the kth iteration of 1th bit state \((C_{k+1},S_{k})\) and \((1+1)\)th bit state \((C_{k+2},S_{k+1})\) could be in any of \((0,0)\), \((0,1)\), or \((1,0)\) states. As \( C_{k+1} = 1 \), it implies that \( S_{k} = 0 \). Hence, from (3), \( C_{k+1} = 0 \) for any input condition between 0 to 1 bits.

We now consider the \((1+1)\)th bit state \((C_{k+2},S_{k+1})\) for kth iteration. It could also be in any of \((0,0)\), \((0,1)\), or \((1,0)\) states. In \((k+1)\)th iteration, the \((0,0)\) and \((1,0)\) states from the kth iteration will correctly produce output of \((0,1)\) following (2) and (3). For \((0,1)\) state, the carry successfully propagates through this bit level following (3).

Thus, all the single-bit adders will successfully kill or propagate the carries until all carries are zero fulfilling the terminating condition. The mathematical form presented above is valid under the condition that the iterations progress synchronously for all bit levels and the required input and outputs for a specific iteration will also be in synchrony with the progress of one iteration. In the next section, we present an implementation of the proposed architecture which is subsequently verified using simulations.

VI. RESULTS

The Verilog HDL Modules have successfully simulate, verified and synthesized using Xilinxise14.5.

PROPOSED RESULT:
SYNTHESIS RESULTS:

<table>
<thead>
<tr>
<th>Logic Utilization</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of slices</td>
<td>78</td>
<td>4656</td>
<td>1%</td>
</tr>
<tr>
<td>Number of slice Flip-flops</td>
<td>99</td>
<td>9312</td>
<td>1%</td>
</tr>
<tr>
<td>Number of 4 input LUTS</td>
<td>85</td>
<td>9312</td>
<td>1%</td>
</tr>
<tr>
<td>Number of bonded jobs</td>
<td>52</td>
<td>232</td>
<td>22%</td>
</tr>
</tbody>
</table>

TIMING SUMMARY:
EXTENSION RESULT:

SYNTHESIS RESULTS:

<table>
<thead>
<tr>
<th>Logic</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
</table>

RTL SCHEMATIC:

DESIGN SUMMARY:

<table>
<thead>
<tr>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of slices</td>
</tr>
<tr>
<td>Number of slice Flip-flops</td>
</tr>
<tr>
<td>Number of 4 input LUTS</td>
</tr>
<tr>
<td>Number of bonded iobs</td>
</tr>
</tbody>
</table>

TIMING SUMMARY:

COMPARISON TABLE.

<table>
<thead>
<tr>
<th>SYSTEM</th>
<th>AREA</th>
<th>DELAY(ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed (CSA)</td>
<td>78 slices</td>
<td>4.04ns</td>
</tr>
<tr>
<td>Extension(PASTA)</td>
<td>53 slices</td>
<td>4.040ns</td>
</tr>
</tbody>
</table>

VII. CONCLUSION

In this paper, we have presented DLAU, which is a scalable and flexible deep learning accelerator based on FPGA. The DLAU includes three pipelined processing units, which can be reused for large scale neural networks. The proposed DLAU uses carry save adder in the computation process and as further process of the project to increase the computation speed parallel self timed adder is used in the design. Experimental results on Xilinx FPGA prototype show that DLAU accelerator achieved more speed when compared to other devices like ASIC.
REFERENCES