# The Use of Field Programmable Gate Arrays to Accelerate Neural Network Computation

Michael Greer, Joao Foltran, Justin Gutter
Department of Electrical and Computer Engineering
University of Wisconsin-Madison

Submitted for EPD 397 Section 007
Bart Skarzynski
June 18, 2018

1. Introduction (Joao)

2 Background
- 2.1 Description of Neural Networks and their Use in Artificial Intelligence (Joao)
  - 2.1.1 Topology
  - 2.1.2 Deep and Recurrent Neural Networks
  - 2.1.3 Parallel Design and Training
- 2.2 Description of Graphical Processing Units (Joao)
- 2.3 Description of Field Programmable Gate Arrays (Michael)
- 2.4 Neural Networks implemented on FPGA (Michael)

3. Analysis
- 3.1. Performance Evaluation for Recurrent Neural Network (Justin)
- 3.2. Performance Evaluation for Deep Neural Network (Justin)

4. Conclusion
- 4.1 Recommendation (Justin)
- 4.2 Ethical Consideration (Michael)

5. References

## Executive Summary

Artificial Intelligence (AI) is used to increase productivity and improve security of a wide range of industries. For instance, AI is used by the banking industry to construct secure forms of profile authentication, the health industry to reduce the time taken to process medical reports and the financial industry to provide a faster alternative to typing. As the popularity and complexity of AI applications increase, there exists a need to improve the implementation efficiency of AI technology. At their core, AI systems consist of artificial neural networks (ANNs), which are mathematical models that allow machines to learn desired functions. The most common hardware currently used to implement ANNs are Graphical Processing Units (GPUs). However, a promising type of hardware that can potentially outperform GPUs in implementing ANNs, due to their reconfigurability, are Field Programmable Gate Arrays (FPGAs). This paper provides the necessary technical background on ANNs, GPUs and FPGAs, analyzes the differences in performance of GPUs and FPGAs in implementing ANNs and provides a recommendation for the most efficient hardware to implement ANNs.

ANNs are parallel structures of interconnected nodes used to model relationships between sets of inputs and outputs. To model this relationship, each node calculates a partial output based on subsets of inputs by using activation functions, which are mathematical functions that vary depending on the application. The ANN finally computes the overall output for each set of inputs by combining the output of each of its nodes. The most common types of hardware used to implement ANNs are GPUs and FPGAs due to their inherent parallel structure. While GPUs and FPGAs are parallel structures of processing units capable of performing a large number of simultaneous operations, GPUs are not reconfigurable, while FPGAs are reconfigurable. This means that FPGAs can be programmed differently for each ANN, thus improving the network's implementation efficiency.

We compare FPGAs and GPUs by analyzing the results of performance evaluations for a recurrent neural network (RNN) and a deep neural network (DNN). We used execution performance, efficiency, and peak performance utilization as metrics to compare both technologies because these metrics together paint a holistic picture of their performance. Despite our analysis being conducted on two types of neural networks, we observed similar trends for both technologies. One of the most important was that batching inputs together, batches of 10 in the instance of the results we used, improves execution performance, efficiency, and utilization for both FPGA and GPU. Comparing FPGA to GPU, they had similar execution performance, but FPGA had 10 to 100 times better efficiency than GPU as a result of its superior utilization. GPU's software overhead limits the amount of processing units that can be utilized during execution while FPGA can be configured with a design that keeps a majority of its processing units utilized during execution.

From our analysis, we conclude that FPGAs are the best implementation hardware for the acceleration of neural network computation. Despite being faced with scaling issues, FPGAs have superior peak performance utilization that enable them to have the best balance of performance and efficiency.

# 1. Introduction

Artificial Intelligence (AI) systems have demonstrated potential to benefit society by improving productivity, security and efficiency of a wide range of industries. For instance, AI systems have been known to benefit the banking industry by enabling more secure forms of profile authentication, the health industry by reducing the time taken to process medical reports (Koivikko, 2008) and the financial industry by providing a faster alternative to typing through speech recognition systems (Ruan et al., 2017). Although research in AI systems continues to reveal new contexts that can benefit from AI technology, there exist limitations regarding its implementation.

The limitations associated with AI are due to the computational complexity of training artificial neural networks (ANNs), the foundational tool used to build AI systems. As a result of the innate parallelism exposed by the structure of ANNs (as explained in section 2.1.2), there are two widely used types of computer hardware to train ANNs: Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Although both types of hardware are well-equipped in performing parallel computations, FPGAs may be reconfigured to an ANN's unique design, while GPUs do not possess such a capability (as explained in sections 2.2-2.3). Because of this reconfigurability and recent overall improvements to hardware specifications, FPGAs are a promising type of implementation hardware to improve the efficiency in training ANNs.

# 2. Background

## 2.1 Description of Neural Networks and their Use in Artificial Intelligence

### 2.1.1 Topology

The fundamental tool for the implementation of AI systems is the use of ANNs. ANNs are used to model relationships between sets of inputs and outputs. A network consists of an interconnected system of nodes, where sets of nodes are organized into parallel layers. In a given layer, each node takes the outputs of nodes from previous layer as inputs, where each of these inputs have a weighted edge associated to it. In each node, an activation function is used to generate a single output based on the weighted input values entering the node.

The activation function is simply a multivariable mathematical function that takes the inputs to a given node as its inputs and produces a real number between 1 and 0 as output. For nodes within the network, this output can be interpreted as the partial output of the ANN for a given set of inputs. In order to account for the fact that some inputs are more relevant at a given node, the activation function assigns bigger weights to inputs that have a higher value of weighted edge. Although the defining formula for an

activation function varies with application, a widely used example of activation function is shown in figure 2.1.1(a).

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \; for \; j = 1, \ldots, k$$

**Figure 2.1.1(a)**: *Example of an activation function* . The activation function shown takes k inputs from a node in the ANN and computes a real number normalized to the range [0,1]. This normalization is analogous to the implementation of squashing functions, as will be explained in section 2.3. Figure by Data Science.

This design repeats iteratively for all layers in the network, where the only differences occur in the first layer nodes, which take the overall input of the system as inputs and the output node, which computes the system's overall output. This design procedure is depicted below in figure 2.1.1(b), where red arrows represent the weighted edges and the green spheres represent the nodes.

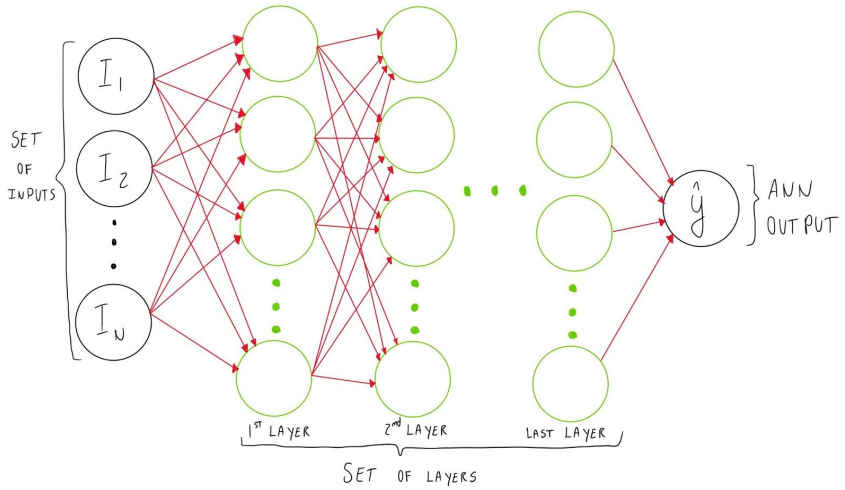


**Figure 2.1.1(b)**: *Example of ANN with N inputs and a single output*. In the ANN shown, each node takes N inputs originated from the original inputs or outputs from nodes of previous layers and calculates an output using activation functions. Figure by authors.

## 2.1.2 Deep and Recurrent Neural Networks

Among the various possible ways of constructing ANNs, there are some useful designs central to AI implementations. For instance, a common type of ANNs used in AI is called Deep Neural Network, where the network is characterized by the presence of multiple hidden layers of nodes, where hidden

layers refer to the additional layers of nodes between the input and output nodes. This design allows the network to model complicated relationships between inputs and outputs by iteratively computing partial outputs based on subsets of the inputs, as explained in section 2.1.1. Whereas the use of a single layer of nodes does not allow for this segmented approach and is thus limited to modeling linear relationships between inputs and outputs (Hoang and Guerraoui, 2018).

Another useful type of ANNs is called Recurrent Neural Network (RNN), which are characterized by a sequential relation between their nodes. Along with the use of internal memory units in the network, this allows RNNs to model time sequences and time-varying data (Mikolov, 2011). By constructing a neural network that presents not only the characteristics of RNNs, but also multiple hidden layers of nodes, it is possible to build RNNs that are pivotal in AI applications of computer vision, speech-recognition and text-recognition (Nurvitadhi, 2016).

### 2.1.3 Parallel design and Training

In this section, we analyze the parallel architecture of ANNs and this design's implications in training the network. As explained in section 2.1.1, the end goal of ANNs is to model relationships between sets of inputs and outputs. In order to model this relationship efficiently, the ANN computes successive partial relations between subsets of inputs by using layers of nodes and weighted edges, shown as red arrows in figure 2.1.1. The parallel structure not only allows for any node to be connected to any original input or node output from previous layers, but it also allows for improved computational efficiency in the training process. The training process consists of feeding training data into the network and using mathematical algorithms to optimize the values of the weighted edges and thus minimize the error between the output produced by the ANN and the desired output. In this context, training data are sets of inputs along with their expected outputs used to train the network for accurately predicting outcome values for unexperienced data (Gustavo et al., 2004).

The most widely used mathematical algorithm for ANN training is the Batch Gradient Descent (BGD), which is a probabilistic model based on dividing the training dataset into batches of equal size and proceeding to update the weighted edges values after processing each batch of data (Ruder, 2016). We present an example model to illustrate the algorithmic process. We assume the batch size for our model is M, the number of inputs our model ANN takes is N, and the training set size is P. The processing portion of BGD starts by inputting a batch of M data points and calculating M output values using the current weighted edge values. The M calculated output values are then compared to the M expected values by using a cost function. The cost function is simply a 2M-variable mathematical function that outputs how far the calculated outputs are from the desired output and varies according to application. At this stage, the algorithm analyzes which values of weighted edges minimize the cost function by using a gradient descent approach (Ruder, 2016).

The gradient descent approach consists of computing the gradient, or in other words, the multi-dimensional derivative of the cost function and analyzing in which direction the cost function achieves a minima. The algorithm proceeds to calculate the weighted edge values that cause the cost

function to reach its minima and updates these values for the next iteration of the algorithm. This process continues iteratively until all data points from the training set are used. One iteration of this algorithm is shown in figure 2.1.3 below.

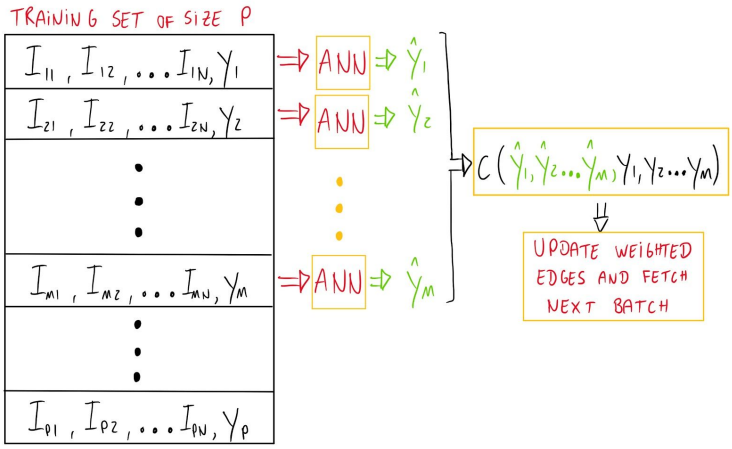


**Figure 2.1.3**: *Implementation of the first iteration of BGD with batch size M*. In the schematic, a training set of size P and a cost function C are used. In the first iteration, the inputs corresponding to the first M rows of the training set are individually input into the ANN and the first M outputs are calculated. The calculated outputs are compared with the expected ones by using the cost function and the values for the weighted edges are then updated. Figure by authors.

As explained above, each iteration of the BGD takes M data points, where each data point consists of N inputs and one output. To achieve this functionality, BGD performs operations on N inputs at the same time. In fact, due to the inherent dependency of an artificial network's output on many input variables, any training algorithm must perform N simultaneous operations P successive times to operate properly. In typical AI applications, the value of N is large in order to accurately model the output's dependency on many variables and the value of P is large in order to accurately optimize the weighted edge value. Thus, in order to implement a training algorithm, it is necessary to use a hardware efficient in performing parallel computations, which in this context are equivalent to simultaneous computations. The most common types of hardware used are GPUs and FPGAs due to their inherent parallel structure, as explained below in sections 2.2 and 2.3. It is also important to point out that the efficiency of training algorithms can be optimized by choosing appropriate values for the algorithm's batch size (Johnson and Zhang, 2015), as will be explained in section 3.

## 2.2 Description of Graphical Processing Units

GPUs are parallel structures of processing cores capable of performing a large number of simultaneous computations. In a typical GPU, hundreds or thousands of simple processing cores, built mainly of transistors, are capable of efficiently performing computation on integer and floating-point numbers, the predominant data type found in neural networks (Cook and Keromytis, 2006). The

Floating-point representation enables a larger range of numbers to be representing than traditional binary representation at the expense of precision where the radix point of a number can move or "float" when representing a number. Because GPUs are equipped with many processing cores, a set of floating point operations can be distributed across available cores to compute results in parallel, thus reducing the amount of time it would take to compute the same results on a Central Processing Unit (CPU) found in most desktops and laptops. Furthermore, each GPU core is equipped with a small, fast cache for data storage which further accelerates its performance compared to CPU which has a relatively larger, slower caches at its disposal. As described in section 2.1.3, training ANNs requires the simultaneous computation of a large number of operations, and as a result, the number of floating-point operations per second (FLOPS) a processing unit can perform is an essential metric to the overall performance of a neural network.

When compared to CPUs, which consist of relatively few cores that specialize in a wider breadth of complex operations, GPUs typically are able to perform better in barebones computation benchmarks. A study by Intel Corporation points out that, even after applying optimizations to the CPU and GPU used in their benchmark, the number of FLOPS performed by the GPU was on average 2.5 times higher than the CPU. The processors used in the study had similar release dates; the GPU used being Nvidia GTX280 and the CPU used being Intel Core i7-960 (Lee et al., 2010). For ANNs, where the number of FLOPs required to train a network is of the magnitude of tens of billions, this difference in performance could mean a reduction of multiple days in required training time (Data Science, 2018).

The high FLOP performance of GPUs when compared to CPUs has contributed for their popularity in ANN applications. However, recent studies and ANN implementations suggest that FPGAs achieve and maintain high FLOP performance which can train ANNs at a faster rate than GPUs (Lee et al., 2010). The higher efficiency of FPGAs is due to their adaptability and optimization to different network structures, as will be discussed in section 2.3.

## 2.3 Description of Field Programmable Gate Arrays

FPGAs are user-programmable integrated circuits consisting of a set of configurable logic blocks that can be uniquely routed together. By having programmable components, FPGAs have a powerful advantage over other hardware alternatives by providing instant manufacturing turnaround and negligible prototype costs (Brown, 2012). Because FPGAs have recently enjoyed improvements to their overall component specifications, more research has turned to the feasibility utilizing this unique hardware to optimize performance for applications which were long thought to be slow.

FPGAs consist of an array of uncommitted elements that can be interconnected by the user in any way supported by its routing resources. The structure of FPGAs consists of a two dimensional array of logic blocks, known as Configurable Logic Blocks (CLBs), interconnected by a routing architecture, with which have programmable switches that will connect a logic block to a wire. Logic circuits are realized by utilizing a set of logic blocks and connecting them together using switches. Because memory blocks

are only capable of reading/writing a single value at any one time, FPGAs have a number of on-chip Block RAM (BRAM) to distribute data and create the illusion of the ability to read/write multiple data from memory at one time. Finally, modern FPGAs have dedicated digital signal processor (DSP) blocks for performing expensive logic operations such as multiplication and division to greatly alleviate the strain of CLB resources to equivalently implement such operations.

The structure of the CLBs vary from one FPGA to another, with simple FPGAs using a single universal gate, which can implement any boolean function (e.g. AND, NOT, OR) without the use of any other gate type, and complex FPGAs using multiplexers or lookup tables, which loads all desired logic values and selects one based on some condition. Beyond basic combinational logic gate, modern logic circuits require memory elements to realize sequential logic, where an output depends on present input values as well as past inputs. Logic blocks are equipped with flip-flops as single bit storage units that may be concatenated as registers for multi-bit storage. Although these combination and sequential logic elements enable high versatility of FPGA logic blocks, the real power of these logic blocks are harnessed when many of them are connected together to realized high-level applications.

The routing architecture of a FPGA has a direct impact its the overall performance. If there exists too few wires to connect CLBs, the board will be unable to utilize all of its CLBs and thus unable to realize sophisticated logic circuits; on the other end, the excess of wires inevitably means that some of those wires will go unused and thus wastes area on the board. Another consequential routing architecture characteristic is the length of the wire segments. If all wire segments are long, there would be a high area and delay costs when implementing local interconnections; conversely, if all wires are shorts, long interconnections need a series of wires connected via programmable switches which results in significant delay costs. The programmable switches which connect the wire segments are the last component of the routing architecture which affects the performance of an FPGA. Although there are a number of technologies used to implement switches such as Static RAM (SRAM), antifuses, and eraseable programmable read-only memory (EPROM), the consequence of these switches are that they occupy larger area and exhibit a significant amount of parasitic resistance and capacitance, which increases circuit area and delays when an unnecessary number of switches are used in a design.

## 2.4 Neural Networks implemented on FPGA

In the context of neural networks, CLBs, BRAMs and DSPs are essential structures to implement ANNs. The networks are modeled using a graph with nodes and interconnects between nodes. Specifically a bipartite graph structure is used for these nodes and interconnects. Bipartite is a structure of layers and interconnects can only go between layers that are adjacent to each other. For each node certain computations and storage needs to be implemented. Each node must somehow store the weights of the interconnects going into that node. This is achieved by using a BRAM block for that node for this storage.

Each node must also perform a multiply and sum operation by multiplying each of the inputs going to that node via a interconnect, with the weights attributed to those interconnects and then summing a bias value to that. The number of nodes in the previous layer is the amount of multiply and sum operations that the current node needs to perform because each node can have an interconnect from every node in the previous layer to the current node. DSP units perform these operations effectively and fast. We want all these operations to happen in parallel so we use a number of DSP units for each node that is equal to the number of nodes in the previous layer.

After this large, parallel computation has been performed the result from this computation is sent through a squashing function that saturates this value to be between [-1, 1]. This squashing function varies according to the neural network implemented and can be complex in its mathematical nature.Due to this complexity, squashing functions are saved as a lookup table, as x and y values, and saved in BRAM. When sending a value though this squashing function we pass it the x value and then we get the y value as an output. This greatly speeds up this stage but there is a loss of accuracy as you are limited to the size of the BRAM memory and the accuracy that it can provide. (Gomperts, 2011)

In this context, a back propagation algorithm is used to readjust the weights on the interconnections of nodes in the graphical representation of the neural network. Once the output has be retrieved from the neural network it is compared with an expected training value and we compare the two and receive an error differential. An error is calculated for each layer as the error propagates layer by layer from finish to start and each layers error is depended upon its successors error. Each layer has its own algorithm to adjust its weights and this algorithm is implemented as a control ‘unit’ called a state machine. One state machine per layer is configured in hardware, and are created by the use of CLBs. A shared hardware backpropagation arithmetic logic unit is used to perform the multiplication and additions needed to calculate the weight corrections and the errors to pass back to the layer before it. This backpropagation happens layer by layer sequentially which is why a single piece of hardware for calculation is used (Gomperts, 2011).

Finally, the entire operation of the neural network is handled by a control unit, known as unit a state machine. This state machine will ultimately start the operation of the neural network. After an initiated start, control signals will be stimulated for correct operation of the Neural Networks input and output. Then when the operation is complete, this state machine will notify the user that the hardware has completed its computations. This state machine is unique to each implementation of neural network and is implemented using CLBs (Gomperts, 2011).

In summary, each node is implemented and configured in a physical location on the FPGA. Each node is independent of one another except for their interconnections together described by the graph flow of a bipartite graph. Each node has BRAM holding its weights, and has multiple DPS units to perform multiplication and addition operations in parallel. These calculations trickle through this configured hardware layer by layer, sequentially, until an output is received. A back propagation training algorithm is performed for each layer, each having its own hardware state machine to adjust the weights associated with interconnects. Operation is controlled through another ‘unit’ of hardware called the state machine. This state machine controls the entire Neural Networks operation.

# 3. Analysis

## 3.1 Performance Evaluation for Recurrent Neural Network

We present the results of a performance evaluation of FPGAs and GPUs for a Gated Recurrent Unit (GRU), which is a Recurrent Neural Network (RNN) variant. GRUs extend on the architecture of general RNNs by dynamically adjusting weights based on current inputs as well as past inputs. Because RNNs primarily consist of dense matrix vector multiplications in order to process data, they are exposed to large amounts of parallelism which GPUs and FPGAs are well-equipped to exploit. For the evaluation report referenced for the performance comparison, the GRU algorithm used included a memorization optimization, which removed a class of matrices operations that improved performance by 46% on average. A consequence of the optimization used in the evaluation report is that the dense matrix vector multiplications dominate the runtime of the GRU algorithm which maximizes the amount of parallelizable operations.

In Table 3.1.1, we present the specifications of FPGA accelerators used in the performance evaluation. Using Altera Stratix V and Arria 10 FPGAs, they both have approximately 4 MBs of on-chip RAMs but Stratix V has 256 DSPs while Arria 10 has 1024 DSPs. These DSPs are used to perform floating point multiplication and accumulate operations so the FPGAs used different designs given their available resources. In Table 3.1.2, we present the specifications of the Nvidia GTX Titan GPU.

**Table 3.1.1:** *Specifications of FPGAs used in RNN evaluation.* Table by Nurvitadhi, E.

| Accelerator Designs | FPGA | ALMs | DSPs | M20Ks |
|---|---|---|---|---|
| 8 clusters, 256 FMAs | Stratix V | 296K | 256 | 4MB |
| 32 clusters, 1024 FMAs | Arria 10 | 224K | 1024 | 4MB |

**Table 3.1.2:** *Specifications of Nvidia GTX Titan GPU used in RNN evaluation.*Table by Nvidia

| GTX TITAN GPU Engine Specs: | |
|---|---|
| CUDA Cores | 2688 |
| Base Clock (MHz) | 837 |
| Boost Clock (MHz) | 876 |
| Texture Fill Rate (billion/sec) | 187.5 |
| **GTX TITAN Memory Specs:** | |
| Memory Clock | 6.0 Gbps |
| Standard Memory Config | 6144 MB |
| Memory Interface | GDDR5 |
| Memory Interface Width | 384-bit GDDR5 |
| Memory Bandwidth (GB/sec) | 288.4 |

The results shown in Figures 3.1.3, 3.1.4, and 3.1.5, below are the comparison of execution performance, efficiency, and peak performance utilization of GPU and FPGA with varying matrix dimensions. The authors of these reports provided results for a NoBatch mode, where inputs are fed in one at a time, and Batch10 mode, where a group of 10 inputs are fed in at a time. In this context, the batch sizes for an implementation are analogous to the implementation explained in section 2.1.3 in analyzing the BGD algorithm.

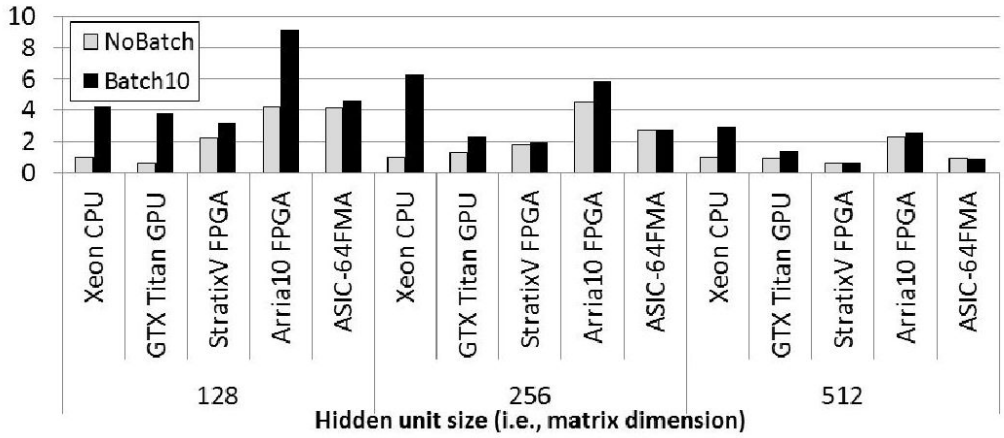


**Figure 3.1.3:** RNN *Performance relative to baseline software on CPU.* In this figure, the performances depicted are relative to the baseline performance of a CPU, depicted as the left-most value in the graph. Figure by Nurvitadhi, E.

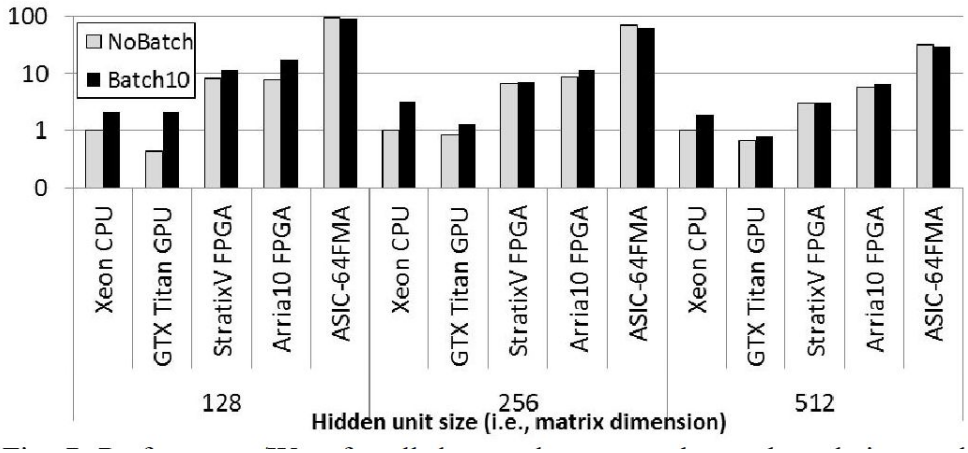


**Figure 3.1.4:** *RNN performance/watt (i.e. Efficiency) relative to baseline software on CPU with no batching.* In this figure, the performances are again shown relative to the left-most CPU performance value, and in this case, the vertical axis is in logarithmic scale. Figure by Nurvitadhi, E.

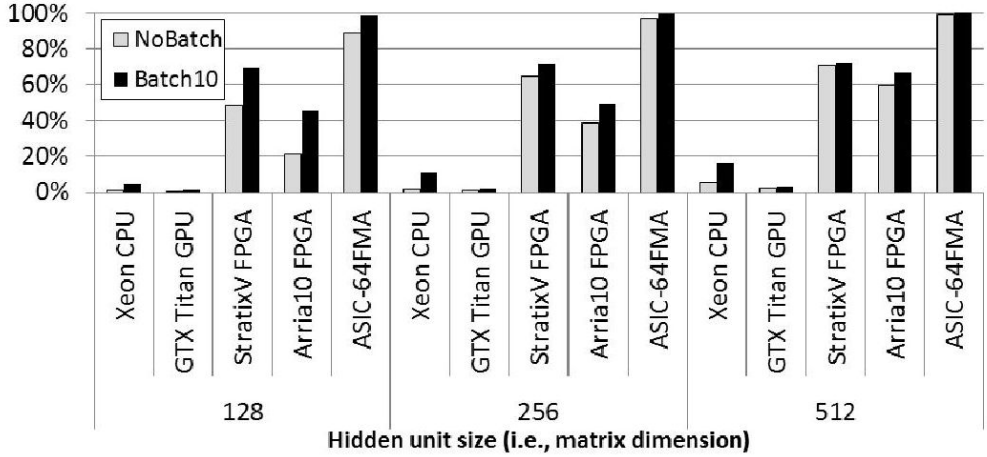


**Figure 3.1.5:** *RNN peak performance utilization.* This shows the performance of the different hardware types relative to their peak theoretical performance. Figure by Nurvitadhi, E.

From these reports, FPGA performed better than GPU in NoBatch mode with the exception of Stratix V for 512 by 512 matrices. In addition, FPGA had far better utilization of its peak performances which resulted in significantly higher efficiency compared to GPU. GPU's underutilization is as a result of a characteristic of software dependent hardware unable to extract fine-grained parallelism from matrices. FPGAs, on the other hand, can realize a custom hardware design which distributes matrix data into many on-chip RAMs that can be processed by multiple available DSPs. From the execution performance, it can be observed that as the matrix dimension size increases, performance among the FPGA designs decreases significantly. Although the FPGAs have fairly good utilization despite this decreasing trend in execution performance, its utilization, along with its execution performance, would suffer drastically if the neural network increased too much. In this context, we would see this decrease in

performance if the neural networks increased to a point in which a single matrix's data exceeded that on-chip RAM available on the FPGAs. Off-chip memory penalties associated with scaling FPGA designs for larger neural network implementations are crucial arguments against their widespread use.

## 3.2 Performance Evaluation for Deep Neural Network

Analogously to section 3.1, we now present the results of a performance evaluation of FPGAs and GPUs for a Binarized Neural Network (BNN), a Deep Neural Network variant. As discussed in section 2.1.2, deep neural networks are widely used for their high accuracies when performing classification tasks used in computer vision, text-recognition and speech-recognition. In order to achieve these high accuracies associated with DNNs, larger networks with more parameters are required to the detriment of efficient execution. BNNs have been used to circumvent some of these execution challenges found in normal DNNs by exhibiting a compact binary representation of network parameters. This representation allows networks to fit in on-chip RAM while offering comparable accuracy to their normal representation counterparts.

To illustrate this, the specifications of Nvidia Titan X GPU are shown below in Table 3.2.1. In Table 3.2.2 shown below, we present the specifications of the FPGA accelerators used in the performance evaluation. Implemented on Altera Arria 10, the two accelerator design were synthesized varying at which the number of processing engines (PEs), an alias for DSPs, utilized.

**Table 3.2.1:** *Specifications of Nvidia Titan X GPU used in BNN performance evaluation.* Table by Nvidia.

| GPU Engine Specs: | |
|---|---|
| NVIDIA CUDA® Cores | 3584 |
| Base Clock (MHz) | 1417 |
| Boost Clock (MHz) | 1531 |
| **Memory Specs:** | |
| Memory Speed | 10 Gbps |
| Standard Memory Config | 12 GB GDDR5X |
| Memory Interface Width | 384-bit |
| Memory Bandwidth (GB/sec) | 480 |

**Table 3.2.2:** *Specifications of FPGAs used in BNN performance evaluation.* Table by Nurvitadhi, E.

| Design name | FPGA64 | FPGA1024 |
|---|---|---|
| Frequency | 220MHz | 150MHz |
| Implementation | Aria 10 | Aria 10 |
| Number of PEs | 64 | 1024 |
| On-chip RAMs | ~4MB | ~4MB |
| Peak throughput (TOP/sec) | 0.9 | 9.8 |

The performance evaluation used a set of layer configurations with their characteristics shown in Table 3.2.3. The evaluation results comparing execution performance, efficiency, and peak performance utilization are shown below in Figures 3.2.4, 3.2.5, and 3.2.6. The authors of the report also used a normal no batch mode and a batch mode with a size of 10 inputs.

**Table 3.2.3:** *Parameters of the BNN layer configurations used in performance evaluation.* Table by Nurvitadhi, E.

| Name | Outputs | Inputs | Binarized model size (MB) |
|---|---|---|---|
| Alex/VGG 7 | 4096 | 4096 | 2.00 |
| Alex/VGG 8 | 1000 | 4096 | 0.49 |
| NT-We | 600 | 4096 | 0.29 |
| NT-Wd | 8791 | 600 | 0.63 |
| NTLSTM | 2400 | 1201 | 0.34 |

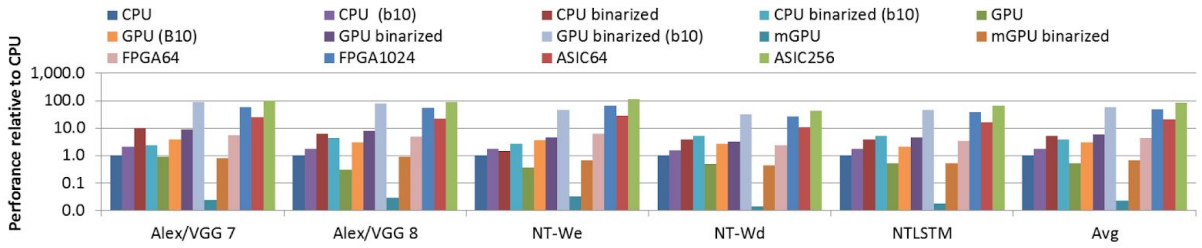


**Figure 3.2.4:** *BNN Performance (indicated by binarized for GPU) relative to baseline software on CPU.* Figure by Nurvitadhi, E.

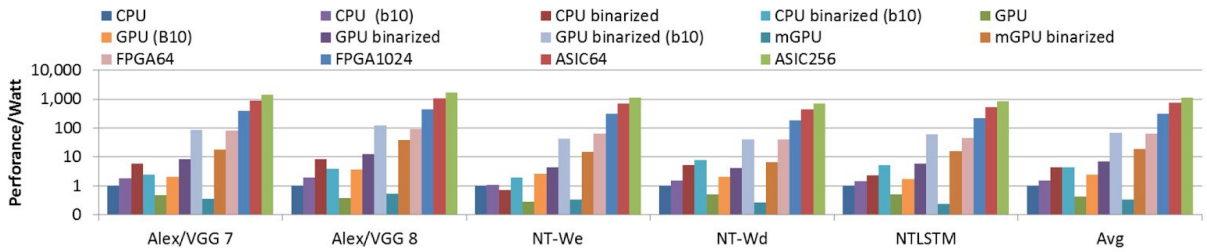


**Figure 3.2.5:** *BNN Performance/Watt (i.e. Efficiency).* This shows the performance relative to baseline software on CPU, indicated by binarized for the GPU. Figure by Nurvitadhi, E.

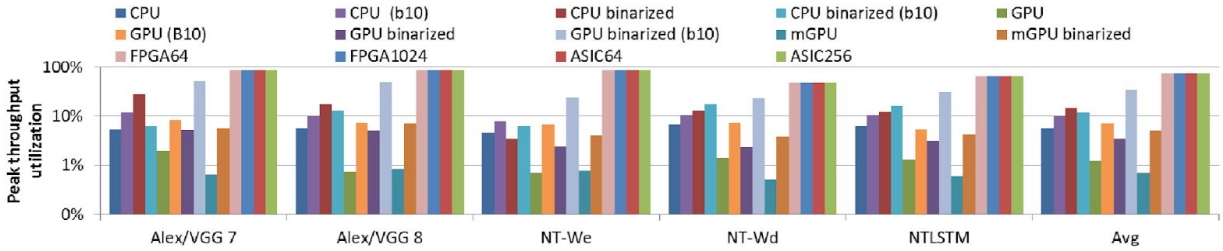


**Figure 3.2.6:** *BNN peak performance utilization (indicated by binarized for GPU).* Figure by Nurvitadhi, E.

From the results of the evaluation report, FPGA64 had slightly higher execution performance compared to non batched GPU, while FPGA1024 significantly outperformed both non-batched and batched GPU. Even though FPGA accelerators have a lower overall peak performance when compared to GPUs, their higher peak performance utilization enabled them to have higher overall execution performance. Furthermore, FPGA64 had nearly 10 times better efficiency over batched GPU, while FPGA1024 had nearly 100 times better efficiency over batched GPU. GPU performance suffers due to the insufficient amounts of fine-grain parallelism that the software overhead of GPU can extract. While some of these performance challenges are minimized by batching, GPU performance will suffer due to poor utilization. FPGAs can synthesize a custom hardware design for a BNN, which enables them to distribute network parameters across available PEs at a high bandwidth that keep their PEs well utilized.

# 4. Conclusion

## 4.1 Recommendation

From our analysis on two types of neural networks, we conclude that FPGAs are the ideal hardware implementation for the acceleration of neural network computations. When compared to GPUs, FPGAs have the best balance of performance and efficiency. This is mainly because FPGAs may be configured with a custom design which optimizes the resources available on an FPGA to perform computations for a specific network architecture. As a result of being configured with such a custom

design, FPGAs may support high peak performance utilization by keeping their DSPs working throughout execution. Although state-of-the-art FPGAs are well-equipped to handle many computations challenges found in the implementation of neural networks, they still encounter scaling issues for larger networks. This happens because in large networks, the necessary parameters do not fit in on-chip memory that invoke crippling memory penalties. As the semiconductor industry starts investing in FPGA as the standard for AI applications, FPGAs can expect to enjoy improvements to their overall specifications which may enable them to overcome this limitation.

## 4.2 Ethical Considerations

Even though there are no ethical considerations directly related to the implementation of Neural Networks using FPGAs, there are some ethical considerations regarding the implementation of AI systems using ANNs. Neural Networks are currently limited by their implementation speed, as explained in section 2.1.3. Thus, once this limitation is overcome, AI systems constructed with ANNs will potentially become even more powerful tools. Because of the contexts in which AI technology is used, this could lead to ethical and safety issues.

For example, improvements in the implementation of ANNs has led to the use of AI technology in constructing autonomous vehicles. Despite more than $80 billion dollar have been invested by companies such as Tesla, Uber and Google, to optimize the efficiency and safety of self-driving vehicles (Jalopnik, 2017), fatal accidents still occurred when testing these vehicles. This demonstrates that, although AI has potential to build innovative and helpful technologies, these technologies can also present a risk to people's lives.

Further, advanced implementations of AI technology have also been used by the medical industry to optimize medical procedures. AI is used to develop surgical robots and Virtual Reality tools that enable various medical procedures to be executed with higher efficiency. Similarly to the case of autonomous vehicles, AI technology also presents a risk to people's lives due to the inherent health risks associated with these medical procedures.

In conclusion, because of the inherent safety risks associated with some AI applications, future applications of AI technology must be accurately designed and carefully tested to ensure that they will not pose unnecessary risk to their users.

## 5. References


Brown, S. D., Francis, R. J., Rose, J., & Vranesic, Z. G. (2012). Field-programmable gate arrays.

Ciregan, D., Meier, U., & Schmidhuber, J. (2012, June). Multi-column deep neural networks for image classification. In Computer vision and pattern recognition (CVPR), 2012 IEEE conference on (pp. 3642-3649). IEEE.

Cook, D. L., & Keromytis, A. D. (2006). Graphical Processing Units. *CryptoGraphics: Exploiting Graphics Cards for Security*, 9-24.

Felton, R. (2017, October 16). A Whopping $80 Billion Has Been Invested So Far In The Self-Driving Car Race . Retrieved June 18, 2018, from https://jalopnik.com/a-whopping-80-billion-has-been-invested-so-far-in-the-1819504421

Gomperts, A., Ukil, A., & Zurfluh, F. (2011). Development and implementation of parameterized FPGA-based general purpose neural networks for online applications. IEEE Transactions on Industrial Informatics, 7(1), 78-89.

Gustavo E. A. P. A. Batista, Prati, R. C., & Monard, M. C. (2004). A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations Newsletter,6*(1), 20. doi:10.1145/1007730.1007735

Hoang, L., & Guerraoui, R. (2018). Deep Learning Works in Practice. But Does it Work in Theory? Retrieved June 2, 2018.

Johnson, R., & Zhang, T. (2015). Accelerating Stochastic Gradient Descent using Predictive Variance Reduction. *Rutgers University Journal*. Retrieved June 18, 2018.

Koivikko, M. P., Kauppinen, T., & Ahovuo, J. (2008). Improvement of Report Workflow and Productivity Using Speech Recognition— A Follow-up Study. *Journal of Digital Imaging, 21*(4), 378-382. doi:10.1007/s10278-008-9121-4

Lee, V. W., Hammarlund, P., Singhal, R., Dubey, P., Kim, C., Chhugani, J., . . . Chennupaty, S. (2010). Debunking the 100X GPU vs. CPU myth [Abstract]. *ACM SIGARCH Computer Architecture News, 38*(3), 451. doi:10.1145/1816038.1816021

Maguire, L. P., McGinnity, T. M., Glackin, B., Ghani, A., Belatreche, A., & Harkin, J. (2007). Challenges for large-scale implementations of spiking neural networks on FPGAs. Neurocomputing, 71(1-3), 13-29.

Mikolov, T., Kombrink, S., & Burget, L. (2011). Extensions of Recurrent Neural Network Language Model. *Institute of Electrical and Electronic Engineers*. Retrieved June 4, 2018.

Neural Networks Training. (2018, January). Retrieved June 17, 2018, from https://www.superdatascience.com/

Nurvitadhi, E., Sim, J., Sheffield, D., Mishra, A., Krishnan, S., & Marr, D. (2016, August). Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In Field Programmable Logic and Applications (FPL), 2016 26th International Conference on (pp. 1-4). IEEE.

Omondi, A. R., & Rajapakse, J. C. (Eds.). (2006). FPGA implementations of neural networks.

Ruan, S., Wobbrock, J. O., Liou, K., Ng, A., & Landay, J. A. (2018). Comparing Speech and Keyboard Text Entry for Short Messages in Two Languages on Touchscreen Phones. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 1*(4), 1-23. doi:10.1145/3161187

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *Numerical Algorithms,42*(1), 63-73. doi:10.1007/s11075-006-9023-9

Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., & Cong, J. (2015, February). Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 161-170). ACM.

Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Phanishayee, A., Schroeder, B., & Pekhimenko, G. (2018). TBD: Benchmarking and Analyzing Deep Neural Network Training. arXiv preprint arXiv:1803.06905.

Zhu, J., & Sutton, P. (2003, September). FPGA implementations of neural networks–a survey of a decade of progress. In International Conference on Field Programmable Logic and Applications (pp. 1062-1066). Springer, Berlin, Heidelberg.