EECE 5640 High Performance Computing

Final Project Literature Review: High Performance of Neural Networks on GPUs

Stav Rones

rones.s@northeastern.edu

4.20.2022

Abstract

Deep Neural Networks have shown to deliver state of the art performance for ML applications such as image classification, natural language processing, self-driving cars and autonomous robotics. Since high performing networks can often take days or weeks to train, software and hardware optimizations aim to accelerate network training while maintaining model accuracy. GPUs have been the leading hardware accelerator of both network training and inference because of their massively parallel SIMT architecture well suited for the matrix multiplication computations that dominate the training process. This literature review covers why GPUs are the hardware accelerator of choice for popular neural network variants such as Convolutional, Recurrent and Low Precision. This review shows various ways to further optimize DNN performance, usability and scalability on GPUs through GPU dynamic memory, arbitrary precision, distributed and heterogenous GPU systems and auto-optimizing GPU algorithms. Additionally, the energy usage and future directions using FPGAs are discussed.

I. SUMMARIES

1. GPU Implementation of Neural Networks (2004)

This paper was one of the first to present the implementation of a multilayer perceptron neural network classification workload on a GPU. The classification of input features is performed by a series of vector inner products of the input feature vector and the weight vector for each network layer. This inner product is then summed with the bias vector and filtered with a sigmoid function for each layer. This process can be optimized by accumulating input and weight vectors and performing matrix multiplication instead of vector inner products.

A single input vector performs faster on GPU because of its pipelined vector processing; however, the GPU implementation is fully optimized when the workload involves many readily available input layers to be calculated in parallel. The workload used in this paper is text-based retrieval of image data, where every MxM pixel window within a large base image needs to be classified with the NN. This allows multiple input vectors, namely the pixel windows, to be combined into a matrix and multiple output classifications can be computed simultaneously. In this paper, an 11x11 window was used with a network with 30 nodes per network layer. When using an ATI RADEON 9700 PRO board to perform the workload, a 20x speedup was achieved with the same result as CPU only implementation.

The paper also shows how the vector and matrix multiplications and non-linear thresholding are performed on the GPU by converting the input and weight vectors into textures and using the vertex shader and pixel shader over a number of rendering phases. This method was necessary to utilize computation on the GPU, as it was before the invention of libraries for general use GPUs (GPGPUs) such as NVIDIAs CUDA framework, which allowed programmers with no graphics processing experience to utilize GPU computations. This paper does not cover the GPU implementation or optimization of training a NN, only classification of an already trained model.

2. Performance and Scalability of GPU-based CNNs (2010)

This paper implements a flexible high-performance C++ and CUDA library for the acceleration of both the training and classification of an arbitrary Convolutional Neural Network (CNN). A CNN is a type of Multi-Layer Perceptron (MLP) network that is optimized for 2D pattern recognition problems. Typically, images are fed through a feature extractor before being inputted to a neural network for training and classification. The problem with this implementation is that the feature extractor is static, and therefore cannot adapt to the network topology nor to the parameters generated by the training of the NN. In a CNN, the feature extractor is the beginning part of the trainable NN in the form of convolutional and subsampling layers before the standards MLP layers. These layers are responsible for feature extraction of edges, corners, end points or non-visual features in other signals. By using subsampling layers after the convolution layers, the shift and distortion sensibility of the detected features are reduced.

The main difference in hardware utilization for the convolutional layer is that the since convolutional and subsampling layers are not fully connected to adjacent layers, the memory access pattern is irregular. To fully utilize the GPU, the memory needs to be accessed in a coalesced fashion, meaning that every GPU thread accesses the memory location with its thread-id as offset. To exploit this optimization, a technique called “unfolding” is used to make the access patterns more linear for the convolutional layers. This allows the forward- and backpropagation to be implemented as a matrix product.

To evaluate the performance of their CUDA CNN library, the performance was compared to a trivial CPU version (CPU_{triv}) and an optimized CPU version (CPU_{opt}). The trivial version was a single threaded non vectorized CPU implementation, while the optimized CPU version utilized Intel’s Integrated Performance Library (IPP) and Math Kernel Library (MKL) which take advantage of SIMD Streaming Extensions (SSE). The GPU implementation used the CUBLAS library. Firstly, the time breakdown of each training part for a single training iteration, namely forward propagation, backwards propagation, weight update and GPU-CPU memory copy were compared for each implementation. The results showed that around 60% is spend in the backwards phase and around 40% in the forward phase while the weight update is negligible.

Next, the scalability of each implementation was compared by scaling the training patterns fed to the network, which in turn increased the number of neurons, weights, and biases.

The results showed that while the optimized CPU version outperformed the trivial implementation, the CUDA version was not only the fastest but also scaled the best with input size.

This paper points out that optimizing the training speed of a network is not only convenient, but necessary to find the optimal classification accuracy of a network for a given workload. This is because the ideal parameters of a neural network (network structure, initial value range, learning method, learning rate, etc.) can only be determined through trial and error, so shortening the training time often leads to more accurate results.

This paper also points out some bottlenecks of GPU utilization – namely CPU to GPU memory transfers, and double precision GPU computations. Neither of these issues are relevant to the NN workload because the same classification accuracy can be achieved with single prevision compared to double, and computation to memory transfer ration is high enough that it is not a bottleneck for the NN workload. This makes NNs a very suitable workload for GPGPUs.

3. Large Scale Recurrent Neural Network on GPU (2014)

This paper shows how a Recurrent Neural Network (RNN) can be parallelized and trained efficiently on a GPU. An RNN is a modified version of a NN that has recurrent connections that allows the network to remember past processed information. This makes an RNN suitable for non-linear time sequence processing tasks, such as a natural language model. RNNs are very computationally expensive to train because they often have output nodes equal to a vocabulary, on the order of 10^3. Additionally, the recurrent connections require many training EPOCHs before they reach convergence. Because of this, traditional RNNs divide the output layer into classes which decreases training time but at the cost of inference accuracy. This paper shows an optimized way to train RNNs on a GPU, as well as demonstrates the GPUs ability to train a RNN with no class output layer and achieve state of the art accuracy on the Microsoft Research Sentence Completion Challenge (MRSC).

The training algorithm used for an RNN is backpropagation through time (BPTT), which expands the infinite recursive nature of RNN training into a finite feed forward network. This network can then be trained like a regular DNN but is optimized differently. RNN training only involves activation and monitoring of 1 neuron on the input and output layer, since the RNN

calculates the probability that a word comes after any previous word where the input and output nodes represent the vocabulary. The first optimization that exploits this is to store the weight parameters and perceptron states in the GPU global memory while keeping the input data in the CPU main memory, in contrast to the technique for image targeted DNNs which store the input data in the GPU memory. Only a small group of index data are demanded to control the training process, and therefore there is little benefit to storing these data on the GPU even though CPU-GPU memory transfer is time consuming. Secondly, since only one input neuron is activated while the rest are 0, the matrix-vector multiplication for each layer can be simplified into an operation of copying the of the corresponding columns to the destination vector.

The proposed two step pipelined method for optimizing the RNN training involves combining kernels. This is not possible with the baseline CUBLAS, so the weight matrices and bias vectors of each kernel are combined instead of the computations. CUBLAS stores matrices in column-major order however, so the combined matrices require an extra step to be differentiated before computation. Two methods are proposed to solve this problem, Leading Dimension Array (LDA) and TRANS. Although both these methods could cause bad performance because of discontinuous memory access, results show that the LDA method still shows significant speed up from the baseline CUBLAS method.

The RNN training is also optimized to avoid conditional branches in the GPU calculations that arise when combining kernels for a pipelined execution. This is done by fitting the model to the warp size by adding dummy nodes to the hidden and output layers so they are integral multiples of the warp size – namely 32.

The experiments for the GPU RNN implementation involved a 50,000 word dataset, and a network with input and output nodes equal to a vocabulary size of 10,000 with varying training depths of 2, 5, and 10. Results for the GPU CUBLAS implementation show significant speedup from the CPU implementation using INTEL Math Kernel Library (MKL), and speedup that increases with hidden layer size and BPTT depth. The pipelined implementation only worked for larger networks, since for small hidden layers the dummy nodes wasted space. The pipelined, LDA, and warp formatted implementation performed the best showing an additional 2-38% speedup compared to the GPU CUBLAS.

4. **SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks (2018)**

Going wider, deeper and non-linear with NN models increases the accuracy of image recognition tasks, but such memory intensive models are limited by GPU memory. SuperNeurons is a dynamic GPU memory scheduling runtime that enables network training beyond the GPU DRAM capacity – reducing the network wide peak memory usage down to the maximal memory usage among layers. This is achieved by combining the memory optimizations of Liveness Analysis, Unified Tensor Pool, and Cost-Aware Recomputation. Exiting memory optimizations exist in frameworks such as Caffe and MXNet, but are static and are not well tuned to address the data dependency variations in non-linear networks. Non-linear networks use fan or join connections for non-neighboring layers which create a complicated runtime resource-management pattern for both forwards and back propagation. Additionally, existing static memory optimizations for DNN training reduce the memory usage at the expense of speed. SuperNeurons provisions necessary memory spaces for the training while maximizing the speed by seeking convolution workspaces within the constraint of native GPU memory size.

Liveness Analysis enables different tensors to reuse the same physical memory at different time partitions, in contrast to the baseline method which allocates an independent tensor for each memory request. A tensor is a 4D structure that cuDNN uses to store the inputs and outputs of a single layer and is used as a basic memory scheduling unit of a network. For every layer, the input and output tensors are tracked, and tensors that are not needed in subsequent layers are eliminated. For example, in back propagation only tensors computed before the current gradient calculation are needed, so Liveness Analysis saves up to 50% of memory. However, using native memory operations such as cudaMalloc and cudaFree incur a non-trivial overhead of the frequent memory operations needed to dynamically manage the memory for each layer. To combat this, a fast heap-based GPU memory pool utility is implemented by preallocating a large chunk of GPU memory for a shared memory pool. This pool uses and updates a list of allocated and empty nodes in a ID-to-Node hash table for every request.

Unified Tensor Pool is a consolidated memory pool abstraction that further alleviates the GPU DRAM shortage by asynchronously transferring tensors in and out of CPU DRAM memory. Only computationally intensive layers such as the convolutional layers can utilize this

optimization, since the layer computations need to be large enough to hide the communication incurred by prefetching and offloading. To further optimize this method, a Tensor Cache is implemented which reduces communication time by exploiting temporal locality of the tensors. This cache implements at LRU policy to match the head-to-tail and tail-to-head computation and dependency pattern of the tensors. Tensors in the cache are locked if they have a later dependency.

Cost-Aware Recomputation exploits the fact that POOL, ACT, LRN and BN layers use together about 50% of memory but only account for 10% of the computation time. This memory can be saved by recomputing these forward dependencies in the backpropagation, which saves all the memory while only incurring a fraction of the performance loss. Before the training iteration begins, the runtime finds all the layers whose backwards propagation memory usage plus the forward computation memory usage for all its dependencies is greater than the max layer forward memory dependency. These forward propagation dependency tensors for these layers will be freed immediately and recomputed during back propagation.

As a result of these memory optimizations, SuperNeurons allows training linear and non-linear networks that are 3x wider (via increased batch size) and 3-12% deeper (via more layers) than popular CNN frameworks can handle. Not only that, but performance speed is greater as well, mainly as a result of dynamically allocating convolutional workspaces per layer.

5. **APNN-TC: Accelerating Arbitrary Precision NNs on Ampere GPU Tensor Cores (2021)**

Accelerating NN training through quantization lowers memory consumption and leads to better runtime performance while making minimal modifications to the original model architecture and only slightly degrading accuracy. Efforts to implement diverse precision networks are restricted by limited precision support on GPUs. Nvidia GPU architectures have specialized cores to support low-precision dense matrix multiplication such as FP16 Tensor Cores on the Volta architecture and int1, int4 and bitwise operations on the Turing architecture, however, quantized DNNs usually require arbitrary precision for example 1-bit weights and 2-bit activations. This paper introduces 3 techniques for supporting arbitrary precision NNs, namely

an arbitrary precision neural network tensor core (APNN-TC) that supports arbitrary bit computations, an arbitrary precision layer design (AP-Layer) including matrix-matrix (APMM) and convolution (APConv), and an arbitrary precision neural network framework (APNN) to minimize data movement across layers.

APNN-TC efficiently maps arbitrary precision NN layers to Tensor Cores with specialized compute primitives and memory architectures by exploiting the 1-bit Tensor Core compute primitive on the Nvidia Ampere GPU architecture. The AP-Bit emulation design decomposes each arbitrary-bit scalar digit into a sequence of 1-bit scalar digits, so that arbitrary computation can be conducted with the only 1-bit and shift operations. This is performed through bit decomposition, batch-based Tensor Core computation followed by bit combination, a process which has negligible overhead compared to the computation complexity.

The Arbitrary-Prevision Matrix Multiplication (APMM) kernel is used for fully connected layers, and does not use existing Binary Matrix-Matrix Multiplication (BMMA) kernels which leads to inefficiencies that ignore data reuse opportunities for decomposed weight matrices and add overhead from BMMA kernel to kernel communication. To facilitate data reuse, the APMM kernel uses batch-based double caching, which tiles matrices and caches them in shared memory and fragment located in registers. Tuning the block size plays a significant role in the caching, so a heuristic algorithm is introduced that selects the best block size for n and m based on the matrix size, weight bit p and feature bit q from a set of 16, 32, 64 and 128 bits. Bit combination of the BMMA result matrices into APMM outputs has low complexity, but suffers from memory bottlenecks. To combat this, two novel designs are included to mitigate memory overhead, namely semantic aware workload shared memory allocation and inter-thread communication.

The APConv kernel implements two strategies to combat the issues with existing low precision convolutional kernels, namely uncoalesced memory access and data padding with 0 that leads to erroneous results. First, channel major data organization transforms uncoalesced and unaligned memory access to a coalesced and aligned memory by splitting each P-bit feature matrix into P 1-bit feature matrices and consecutively storing all channels of elements with the

same spatial location. Secondly, input aware padding design adaptively adjust padding values according to input values.

The APNN design aims to improve performance at the framework level by introducing minimal-traffic dataflow and incorporating a semantic-aware kernel fusion to minimize the memory access across layers. To achieve minimal-traffic dataflow, all the weights in the hidden layer are quantized before the model inference computation. Additionally, data is packed bit by bit for both weight and feature map to effectively maintain and transfer arbitrary-bit data. Semantic-aware Kernel fusion fuses APMM/APConv with its following quantization, BN, pooling and ReLU kernels into a single kernel to minimize the global memory access. Additionally, the shared memory access if further reduced by directly reusing values in registers.

When comparing APMM with Nvidia low-bit GEMM operations, a 2.5x speedup a 3x speedup are found for int4 and in8 respectively. This shows that using emulated 1bit compute primitives are actually better than the int4 or int8 native primitives. APConv operations are compared to Nvidia implementation of low bit convolution TCs, which similarly achieve a 3.78x and 3.07x speedup for in4 and int8 convolutions respectively. APNN performance was evaluated with 3 mainstream NN models and the ImageNet dataset, and in general found that APNN design showed significant speedup over ResNet and VGG.

6. **A History-Based Auto-Tuning Framework for Fast and High Performance DNN Design on GPU (2020)**

Attempting to optimize DNN application on GPUs is a very time consuming and low level process for ML engineers as a result of the high level of diversity in DNN optimizations. This paper presents a novel auto-tuning framework to optimize DNN design on a GPU by leveraging the tuning history efficiently in different scenarios. The auto-tuning process finds the optimal mapping from an operation to the low-level GPU code known as the schedule. Different schedule templates are provided with tunable factors such as tiling and unrolling, and the algorithm chooses values the optimize performance. Existing auto-tuners are very time consuming to because of the many tunable parameters, long parameter optimizing convergence times and large number of EPOCHs per parameter test.

The proposed auto-tuning process reduces the time by efficiently exploring the schedule space through taking advantage of previous tuning history in different scenarios. This occurs in two phases which are TL (Transfer Learning) based auto-tuning with filtering and GGA (Guided Genetic Algorithm) based auto tuning. The input network is first decomposed into layers since the hardware optimization is conducted in a layer by layer manner, and each layer is assigned a match score which represents it similarity to the history records. If the match score is low, the TL-based tuning flow is used, which filters the history of different layers in the database to find the optimal hardware executing schedule. If the match score is high, the GGA-based auto-tuning flow is used so that the optimal schedule for the hardware execution can be found during the guided crossover and mutation process.

The TL flow first uses a filter to avoid selecting records with irrelevant or contradictory input settings that can poison the model and compromise the converging speed and performance. This filter is based on the importance and similarity of the features, which is determined by parsing the history into feature vectors of the tunable parameters and comparing the similarity of the feature vectors. Vector similarity is obtained by first finding the importance of each features by applying a moving average to the statistics and the numbers of times that a feature splits the decision tree. The importance of each features is then scaled and use to determine the similarity of features for filtering.

The GGA based auto-tuning is similar to the TL filter, but additionally includes the history performance, utilization ratio, and parallelism in addition to the similarity of the input layer and the history record. The algorithm uses the knobs as the “genes” of the algorithm and generate mutated offspring of varying performance. These “elite genes are stored in a separate dictionary and guide the later mutations.

On an Nvidia 1080Ti using the first convolutional layer of ResNet-18, an average of 8.96x and 4.58x acceleration is observed using the proposed TL and GCC filtering over the XGBoost and GA tuner respectively.

7. **Profiling DNN Workloads on a Volta Based DGX-1 System (2018)**

This paper analyzes both the weak and strong scalability properties of DNN training on a distributed GPU system when varying parameters such as the network model, mini-batch size, number of GPUs, GPU connection hardware and GPU connection software. Distributed GPU DNN training is parallelized by using either the model parallel or data parallel implementations. Model parallel distributes a single model across nodes, while the data parallel replicates the model across nodes which each receive a different mini-batch to process. The model parallel approach is better suited for networks with more fully connected layers than convolutional layers, while the data parallel approach is better for the opposite case. The latter scenario is often the case, and therefore the data parallel approach is more common and supported by all frameworks, and thus is the implementation chosen for this study.

Distributed data parallel training involves first using the CPU to distribute the model as well as the first set of mini-batches to all GPUs. Each GPU accumulates the gradients for each input of the mini-batch in parallel. When all the gradients are computed, they are sent to the Server GPU to average the gradients, compute the updated parameters and send them to all of the GPUs to start the next training EPOCH. This can either be done synchronously or asynchronously, each have their pros and cons. With synchronous, GPUs will idle while waiting for the updated parameters from the server GPU, especially if there are asymmetrical GPU-GPU connections. With asynchronous, the GPUs will not have to idle, but since it does not receive updated parameters as often the accuracy for each EPOCH is decreased and therefore so is the total convergence time. This study is performed with the MXNet framework, which utilized the synchronized approach. The inter GPU communication can either be performed with direct memory access via NVLink and P2P calls, or with NCCL API. P2P allows data transfer to peers without needing to stop the GPU kernel, and the granularity is typically smaller than a cache line. NCCL uses its own kernel with pipelined collective operations such as AllReduce for gradient averaging and Broadcast for parameters distribution, but it incurs a greater overhead.

To analyze the effect of varying certain parameters, the DGX-1 system was used which contains 2 CPUs directly connected to 4 GPUs each via PCIe, and GPUs connected to each other via NVLink direct access. The direct connections between GPUs were asymmetrical, meaning some GPU-GPU transfers took more time than others, and some GPUs needed a GPU or CPU intermediary for communication. MXNet was used to train 5 popular networks that had diversity

in parameter size and layer computational intensity, and results were obtained with the nvprof profiler.

The first parameter that was varied to determine its effect of distributed training was the mini-batch size. When increasing the mini-batch size, more computation can be performed before communication is needed, as well as the total number of communications for total training is reduced. The study assumes that increasing the mini-batch size does not effect the training accuracy. Results showed that increasing the minibatch size improved training speed for every model across every GPU count and connection software configuration. The limiting factor for further increasing the mini-batch size is the GPU memory, which scales higher than linearly since more feature-maps need to be stored before they are sent to the server GPU. When comparing P2P vs NCCL, P2P was found to perform better for smaller networks and a lower GPU count, as a result of its faster transfer times and lack of overhead. However, NCCL and its pipelined collective operations amortize the overhead when a GPU count of 4 or more are used and show faster results than P2P for larger networks. Lastly, the study finds that the model plays a large role in the distributed training scaling. Networks that have higher computation, i.e more parameters and higher mini-batch sizes, can overlap computation with communication to achieve latency hiding.

In general, distributed training achieves the best scaling speedup with large a large mini-batch size, computationally intensive network, and NCLL collective communication operations for GPU count over 4.

8. **Heterogeneous System Implementation of Deep Learning NN for Object Detection in OpenCL Framework (2018)**

Many accelerated NN implementations only run only on Nvidia hardware through the CUDA framework, which ignores many important high performance NN applications such mobile, embedded and low power. OpenCL code can be applied to heterogenous systems including CPU, all vendor GPUs, DSP and FPGA. This paper presents an OpenCL based DNN object recognition framework that can be implemented on general GPUs, an embedded system, and an FPGA system.

The OpenCL API resembles CUDA, where both the host and kernel is programmed and memory is copied between them via a global, constant, device and shared buffers. Different than CUDA, however, is the fact that arithmetic operations of array pointers of the buffer are opaque to the host and must be executed on an OpenCL device. The convolution layers of the implemented CNN are computed using GEMM (General Matrix Multiplication), which accounts for 92% of the GPU execution time. The GEMM is blocked in both dimensions and distributed to each work item the OpenCL device.

OpenCL was successfully used to implement an DNN inference algorithm on both an embedded system and an FPGA system. The embedded system consisted of an ARM Cortex-A15 dual core with a Mali T628 MP6 GPU run with Ubuntu. The Mali T628 consists of 2 asymmetrical GPUs, whose workload ratio could be successfully balanced using strategic blocking and OpenCL barrier synchronization call to eliminate idling. The FPGA system consisted of a Xeon E5-2620 V4 CPU and an Arria 10 FPGA run with CenterOS 7. The FPGA suffers from a lack of cache memory, which greatly increases the speed of GEMM on devices on a GPU for example with an L2 cache memory. However, using OpenCL pipes or channels, GEMM kernels could be directly connected to eliminate the need for costly global memory and increase the data transfer efficiency.

9. Evaluating the Energy Efficiency of Deep CNNs on CPUs and GPUs (2016)

This paper evaluates the performance and energy consumption of various CNN frameworks on both CPU and GPU to determine the training energy efficiency. Additionally, a methodology is introduced that quantifies the energy consumption of each layer, identifying which layers are the most power hungry. CNNs contain 3 types of layers – convolutional, pooling, and fully connected. While fully connected contain many more parameters than convolutional, convolutional are computationally more intensive.

This study evaluated four successfully proven CNNs, namely AlexNet v2, OverFeat, VGG_A and GoogleNet and were implemented using frameworks such as Caffe, Torch 7, TensorFlow, MXNet, and CaffeConTroll. The hardware used was a 16-Core Intel Xeon E5 - 2650 v2 with Hyper-Threading enabled and 32GB of DDR3 memory, an Nvidia Tesla K20m with 5GB of memory and an Nvidia Titan X with 12GB of memory all run with CentOS v7. A

modified Convnet profiler tool was used to collect power and performance data for each phase, and the CPU and DRAM power data were collected with Intel's RAPL interface and Nvidia System Management Interface.

Factors that were modified to test their impact on power efficiency were CPU and GPU libraries, network topologies, batch size and hardware parameters such as hyper-threading, Error Correction Code (ECC) and Dynamic Voltage Frequency Scaling (DVFS).

The first factor that impacted energy efficiency was the use of cuDNN, an DNN library for Nvidia GPUs that includes optimized operations for convolution and pooling. When using cuDNN over native kernel code, the power consumption increased by 16% due to full utilization of the hardware, while the energy consumption reduced by 42% which comes from reduced training time. When analyzing the network topology in relation to power consumption, it was found that the convolutional layers consume 87% of the total energy while the fully connected layer consume 10% and the pooling / activation layers consume less than 5%. Increasing the training batch size increases the power consumption and therefore reduces the energy efficiency until memory saturation is reached, at which point further increasing the batch size does improve energy consumption

When comparing hardware, the Titan X 54% more energy efficient than the K20m when using cuDNN despite its much higher power consumption. Interestingly, the CPU accounted for 22 - 40% of energy consumption even though it is mostly in an idle state when training with GPUs. When using CPU alone, the energy consumption is much higher than the GPU implementations even on an optimized CPU implementation such as Caffe-OpenMP.

Other factors effecting energy consumption were hyper threading, ECC and DVFS. Hyper threading was found to negatively impact performance as well as energy efficiency, since this application saturates hardware with computation and does not experience benefits from memory latency hiding. ECC consumes memory, so removing it can free up some space for larger batch sizes, and since CNN training typically do not require bit-level accuracy it can tolerate random errors. DVFS allows the GPU frequency to be dynamically modulated to optimize energy consumption. The study found that energy consumption was relatively high at both the lowest and highest frequencies while it was lowest at intermediate frequencies. Low frequencies caused longer execution time, while increasing the frequency beyond a certain level increases the power consumption with limited return on performance. Training is an energy-

aware fashion requires operating at frequencies with a good trade-off between execution time and power consumption.

In summary, to optimize the energy efficiency of NN training, the hardware should be optimally utilized to minimize training duration. Additionally, a CNN should only be used when necessary as it required much more computation that fully connected layers.

10. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep NNs? (2017)

Recently, energy efficient and extremely customizable Intel FPGAs have been achieving raw floating point performance up to 9.2 TFLOP/s, which is within striking distance of state of the art GPUs. GPUs are the device of choice for FP16 or FP32 GEMM operations involved in classic DNN implementations, but they fail to perform well for emerging DNNs that exploit network sparsity and compact data types which would be much better suited on the extremely customizable FPGA. This study evaluates a selection of DNN algorithms on Intel Arria 10 and Stratix 10 FPGAs and compares it to performance and energy efficiency of a Titan X Pascal GPU. Additionally, the paper develops a customizable DNN hardware accelerator template for FPGA that can support various next generation DNNs.

The customizable hardware architecture template for DNNs consists of memory and on-chip data management units, GEMM unit to compute convolutional and fully connected layers, and miscellaneous layers unit to compute the other DNN layer types such as ReLU, Batch Norm and Pool. Weights and inputs are first loaded into on-chip buffers in ODM from memory. Then, the GEMM units perform convolution and FC layers with blocked matrix operations and outputs the result to the MLU which performs the other layer computations (ReLu/BatchNorm/Pool). Theses results are stored in the on-chip buffer in ODM and the process is repeated for the next layer. The GEMM unit is customizable to use systolic-based or broadcast-based architectures across the PEs. The FPGA DNN architectures is mean to support arbitrary precision data types, sparse pruned DNNs, Binarized DNNs and Ternarized DNNs.

Results show that GEMM operations for pruned, Int6 and binarized DNNs are 10%, 50%, and 5.4x faster on the Stratix 10 FPGA than on the Titan X Pascal GPU. Additionally, the Ternary 2-bit ResNet model on the Stratix 10 can deliver 60% better performance over the Titan

X Pascal GPU using the FP32 ResNet while being 2.3x better in performance/watt and within 1% of accuracy. However these are only estimated theoretical results and GPUs still beat out FPGAs in classical FP16 or FP32 dense GEMM.

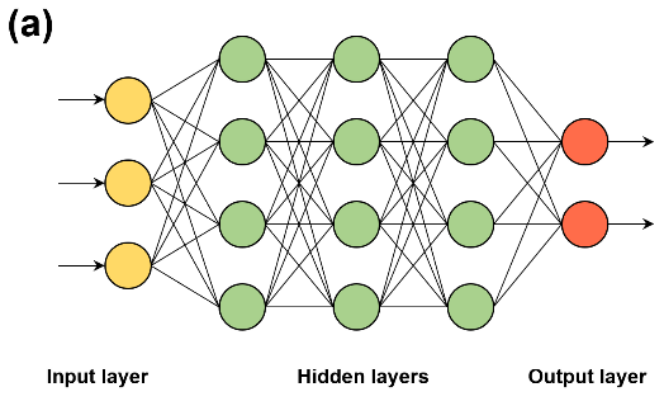
II. DISCUSSION

DNN Overview

Model

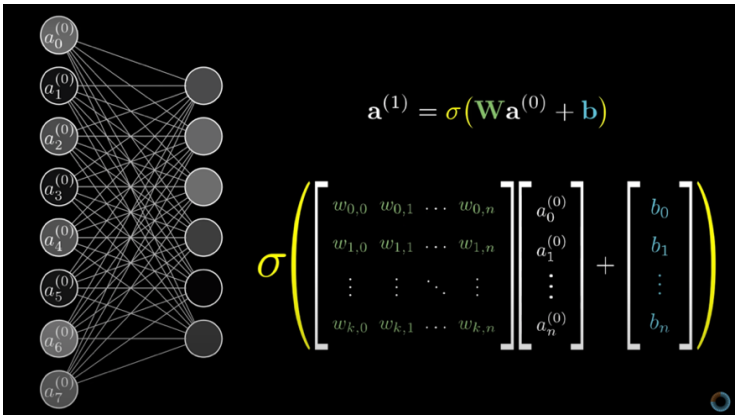
A neural network (NN) is a Machine Learning (ML) model based on the neurons and synapses of the human brain. All neural networks consist of a feed forward graph structure where the nodes of each layer are connected via weighted edges to nodes of adjacent layers. The model is used to transform input information, known as the feature vector, into an output classification where each node in the output layer represents the probability that the input belongs to that classification. A deep neural network (DNN) refers to one which has more than 1 hidden layer, which is the layer between the input and output. A diagram of a 3-layer DNN is shown in Figure 1, which has an input vector of length 3 and an output layer of length 2. This means that 3 inputs are used to determine a binary classification – for example you input your rating of the artists the Rolling Stones, Led Zeppelin and Justin Bieber and the network determines the probabilities that you are or are not a fan of rock and roll.

Figure 1. Simple DNN visualization



Each output neuron and hidden layer neuron is calculated by performing an inner product between the previous layer and its weights (edges) to each node in the previous layer. The computed neuron value is then summed with its scalar bias value, and then mapped using an activation function, of which various equations can be used such as sigmoid, ReLu, or Softmax. The activation function for the output layer typically scales each neuron to be between 0 and 1 to represent probability. The inner product computation can be parallelized by computing an entire layer at once through combining all the weights present between two layers into a matrix and multiplying by the previous layer, as visualized in figure 2. This process is repeated for each proceeding layer.

Figure 2. Visualization of Layer to Layer Neuron Computation



This computation can further be parallelized by combining multiple independent input observation vectors into a matrix and performing matrix-matrix multiplication to simultaneously compute multiple output vectors. GEneral Matrix-Matrix multiplication, also known as GEMM, is the main computation involved in both inference and training of a neural network.

Training

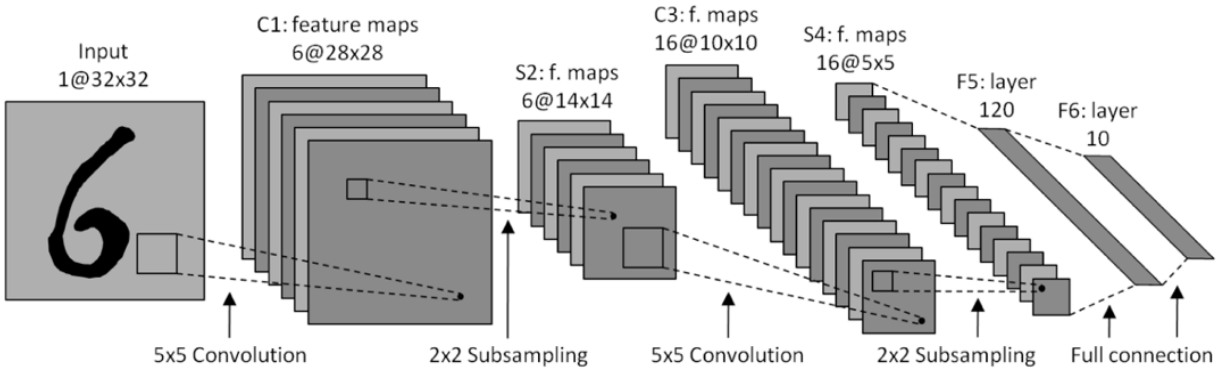
While inference refers to the classification of inputs using the network, training refers to the algorithmic selection of the values for the weights and biases of every neuron. This is done through a 3-phase process of forwards propagation, backwards propagation, and weight update. All the parameters of the network are initially randomized and labeled inputs from a training set are fed through the network in the forward phase to compute outputs the same way as inference is computed. The difference between the labeled output (one output layer neuron is 1 and the rest

are 0) is backpropagated through the network so that the error gradient of each neuron can be calculated to reflect if it should be increased or decreased. This is known as backwards propagation, and also consists of matrix multiplications. Finally, the values of all the weights and biases are updated to reflect the calculated gradient for each parameter, and this process of gradient descent is repeated until the weights and biases reach convergence. Multiple independent inputs can be trained simultaneously and before their accumulated gradients are used to update network parameters, the number of which is known as the mini-batch size. This plays an important role on the performance of the training.

CNNs

Simple DNNs only contain fully connected (FC) and activation layers, but there are many network derivatives that contain specialized types of layer connections that are used for different classification tasks. The most popular is a convolutional network (CNN) which delivers state of the art performance for optical character recognition [16] and differs from a DNN in its use of convolutional (Conv) and subsampling layers. To perform image recognition with a standard DNN, the images typically need to be fed through a features extractor before being classified with the FC layers, which is a static approach that cannot adapt to the network topology. In a CNN, The Conv layers act as feature extractor for edges corners and endpoints, while the following subsampling layers reduce shift and distortion sensibility of the features. A Conv layer differs from a FC layer in that each neuron is connected to a small subset of neurons in the previous layer, and the number of outgoing connections is not equal for all neurons in a Conv layer. A visualization of a standard CNN is shown in Figure 3.

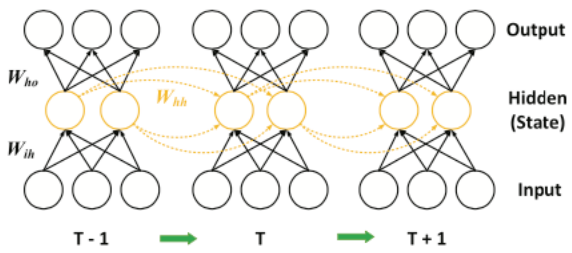
Figure 3. Architecture of the LeNet5 CNN



RNNs

Another derivative of the classical Multi-Layer Perceptron (MLP) model is the Recurrent Neural Network (RNN) specialized for non-linear time sequence processing tasks such as a natural language model and speech recognition. Recurrent refers to hidden layers that depend not only on input features but hidden layer outputs from previous inputs in the time domain. The depth of the recurrence is depends on the model topology, and is usually chosen to reflect the application it is being trained for. RNNs are trained with back propagation through time (BPTT), which expands the recursive nature into a finite feed forward network so it can be trained like a regular DNN. An RNN is typically used to determine the probability that a given word comes after any other given words, so the input and output layers have 1 node per word in the vocabulary. This results in specialized training in the sense that inputs only activate 1 node while the rest are 0 and the output truth is a single activated node as well. An example of an RNN topology is shown in Figure 4,

Figure 4. RNN Architecture



DNN variants spanning topology, size, and numerical precision are a very important part of research and advancement for the improvement of ML accuracy, and therefore specialized DNN accelerating hardware needs to be flexible enough to handle current and future models.

Use of GPUs for NNs

DNNs have proven to be very high performing techniques in applications such as computer vision [11], natural language processing [12], self-driving cars [13] and autonomous robotics [14]. However, modern networks are extremely large and the number of parameters in state-of-the-art networks has doubled roughly every 2.4 years [15]. Not only does this lead to training time of days or even week, but the millions of parameters push the memory capabilities of modem computing systems to the limit. For example, Inception v4 is a model with 515 basic layer that consume 44.2 GB of memory in training, and the deep learning community widely acknowledges that making networks deeper and wider improves the quality of image recognition tasks [4]. Because of the impressive accuracy of these models, the need for accelerating performance in both the time and memory domain has become the focus of both industry and academia.

Why GPUs

Parallel and vector processing are shown to drastically accelerate the time of both training and inference, leading to breakthroughs in machine intelligence. This is because the main computation of DNN training and inference is dense GEMM, whose parallelizable computations are primed to take advantage of computational architectures that exploit data reuse, data locality and data density. While multi-core CPUs and Streaming-SIMD Extensions (SSE) such as AVX vectorization are a valid way of achieving GEMM acceleration, they do not hold a candle to the throughput possible with modern GPUs. NNs in particular are very well suited for offloading to GPU because it does not suffer from a bottleneck that often occurs with other GPU workloads – namely CPU-GPU memory transfer rate. The amount of data that has to be copied between the host and device, namely the network's inputs and outputs, is relatively small in comparison to the large matrices that must be handled inside the network. Additionally, GPUs only recently have adopted more hardware for double precision, and still struggle to match

double performance to that of single floating point. Because NNs shows no difference between double and single precision implementations [2], this bottleneck again does not impact DNN performance.

Historic vs Modern Usage

The first publication of using a GPU for a NN was in 2004 where an ATI RADEON 9700 PRO board was used to perform fast NN inference for image text detection in a convolved image. Using each convolution as an independent input allows the GPU to be fully exploited with GEMM instead of single input vector-matrix multiplication. At the time, GPUs were for graphics processing only, not like the general purpose GPUs (GPGPUs) available today. Using the GPU for general purpose calculations required a deep understanding of the hardware architecture, where problems had to be implemented using a graphics API and algorithms needed to be transformed into a graphics pipeline friendly format. Since then, GPUs are now programmable through a C like API via vendor specific platforms such as Nvidia's CUDA and AMD's HIP, as well as through cross-platform frameworks such as OpenCL. The GPU market is now largely driven by optimizing NN workload, as can be seen by NVidias Tensor Cores (TC) introduced in the Volta architecture. TCs are specialized computational units for computing layer wise input and output Tensors, which are 4D memory units. Additionally, NVidia specific GPU libraries and APIs such as cuDNN, cutlassCONV, NCCL and tensorRT are libraries to optimize DNN computations on GPUs. All modern DNN frameworks such as Caffe, MXNet, TensorFlow and Torch not only support GPU computation, but distributed GPU computation as well.

MLP Derivatives on GPUs

While the FC layers of the classical NN model perfectly match the GEMM operations of the GPU, variant networks such as CNNs and RNNs that have non-linear and non-uniform topologies that do not directly map to GPU hardware which call for specialized implementations. For CNNs with varying number of input and output connections for convolutional layers, the memory pattern is irregular and ever layer needs to know the type of its subsequent layer to know the number of connections. One method used to simplify the backpropagation section of training is to "push" the gradient from the upper layer instead of pulling it from the lower layer

as described in detail in [16]. This gives the layers a regular structure, so that knowledge about neighboring layers is not required and the memory access can be optimized using coalescing and data locality. Additionally, the "unfolding" technique described in [17] is used to map convolutional kernels into uniform matrices so that forwards and back propagation can be implemented as a matrix product and data locality memory optimizations and coalescing can again be exploited. [2] shows that the GeForce GTX 275 GPU implementation of the CNN is faster than the trivial and optimized Intel Core i7 CPU implementations, and scales better with increasing network size.

For RNNs, the BPTT algorithm described in [18] can be used for training that expands the infinite recursive nature of the RNN training into a finite feed forward network which can be trained like a regular DNN. The nature of the RNNs application wherein only one input and output node are activated per training iteration can be exploited to optimize the RNN for GPU. Firstly, since the inputs and outputs are so sparse, the weight parameters and perceptron states can be stored in the device memory while keeping the input data in the CPU main memory. This contrasts with the technique for image targeted DNNs which store the input data in GPU memory. Only a small group of index data control the training process, so there is little benefit to storing the data on the GPU despite the costly memory transfer. Secondly, the matrix-vector multiplications for each layer can be simplified into an operation of copying the corresponding columns to the destination vector since most of the inputs are 0. [3] demonstrates the GPUs ability to train a RNN with no class output layer and achieve state of the art accuracy on the Microsoft Research Sentence Completion Challenge (MRSC).

DNN Precision on GPU

Arbitrary Precision (AP) NNs is an emerging area of NN optimizations that use very low bit weights and edges, as low as 1 and 2 bits in Binary (BNN) and Ternary (TNN) models. Lowering precision through quantization lowers memory consumption allowing larger batch sizes and network sizes, and surprisingly can achieve better runtime performance with minimal degradation in accuracy [19]. However, computation is only improved with hardware support, and most GPU architectures do not even include int1 and int4 datatypes. Nvidia's Ampere architecture introduced a 1-bit TC and logical operations, which can be exploited to emulate

higher bit precision networks. [5] shows a method for AP emulation, AP GEMM (APMM), AP Convolution (APConv) and even an APNN framework for the Ampere architecture to allow experimenting with different types of APDNNs. This is achieved by decomposing an arbitrary-bit scalar digit into a sequence of 1-bit scalar digits so that computation can be conducted with only 1-bit and shift operations. This method was able to achieve significant speedup over CUTLASS kernels and various NN models such as ResNet and VGG for APNNs. Without the 1-bit TC on the Ampere architecture, the more flexible FPGA poses a platform well suited for AP NNs.

DNN Memory on GPU

GPU memory can be a limiting factor for NN implementations, since the network needs to fit on the device memory more memory memory allows for a larger mini-batch size. Larger batch sizes are shown to improve the model training speed without impacting the model accuracy [7], so maximum utilization of memory is desired. While onboard GPU memory increases with every new architecture, experimenting with very large models is still limited to a model parallel distributed approach which is complex to implement and has undesirable overhead. Existing frameworks such as Caffe and MXnet have static memory optimizations that work well for linear and FC networks, but do not perform well for dependency-complex non-linear networks that utilize fans and joins to connect neurons in non-consecutive layers. Impressively, a dynamic GPU memory scheduling runtime called SuperNeurons presented in [4] enable training far beyond the GPU DRAM capacity. By combining memory optimizations such as Liveness Analysis, Unified Tensor Pool and Cost Aware Recomputation, the peak network memory usage is reduced to the maximal memory usage among layers.

Liveness Analysis enables different tensors to reuse the same physical memory at different time partitions, by recycling tensors that are no longer depended on. For examples, during back propagation this frees all the tensors in memory from layers after the current backpropagation layer. Unified Tensor Pool further alleviates GPU DRAM by asynchronously offloading and prefetching tensors to and from CPU memory. This optimization is very useful for convolutional layers, whose computation is intensive enough to hide the communication latency, allowing additional space for convolutional workspaces. Cost aware recomputation finds

tensors whose recalculation time in the back propagation phase is small enough that recomputing them to save memory space yields better performance than the memory costly alternative. By combining these GPU memory optimizations, SuperNeurons allows training linear and non-linear networks that are 3x wider and 3-12% deeper than popular CNN frameworks could typically handle.

DNN Optimization Techniques for GPU

There are many ways to further optimize DNN training on GPUs that target hardware utilization in a single worker setting, such as reducing memory footprints to increase batch size, reducing precision, fusing kernels and layers, and improving low-level kernel implementation. The diversity of DNN GPU optimizations makes it difficult and time consuming to know which optimizations will work best, if at all, for the given network topology and system configuration. One option is to simulate these optimizations by building a dependency graph from low-level kernel traces, and performing graph transformations to simulate different optimizations. The effectiveness of this method is described in [20], but it requires the programmer to have a deep understanding of the optimization at a hardware level and requires the programmer to manually input the optimization. Another option is to use auto-optimization, which frameworks use to test various optimizing parameters. The downside to auto-optimization is that it is expensive in nature to test many optimizations, and often has a very long convergence time. An auto-tuning framework that utilizes a history database of GPU DNN optimizations for various network topologies is presented in [6] that shows reduced search times of 8.96x and 4.58x compared to other auto-optimization tuners such as XGBoost and the Genetic Algorithm tuner in TVM Nvidia 1080Ti using the first convolutional layer of ResNet-18.

DNN training with Distributed GPUs

Distributed training is an extremely popular way of optimizing DNN training which involves a system of interconnected GPUs and CPUs. Host to Device connections are typically implemented with PCIe, while GPUs can communicate directly using NVidias NVLink which allows direct memory access without having to halt the kernel, or with NCCL that offers collective communication operations. DNN training can either be distributed using model

parallelism or Data Parallelism. Model parallelism splits a single model across multiple nodes, and is better suited for networks with more FC layers than Conv layers. Data parallelism replicates the model on every node, and each uses a different mini-batch to determine multiple gradients in parallel. Data parallelism is better suited for models with more convolutional layers, and paired with the fact that it is less complex to implement makes it the distributed approach of choice.

The gradients on each node are aggregated and averaged on either the host or a server GPU node, and new parameters based on this aggregation are then sent to all the nodes for new training iterations known as the Parameter Server (PS) model. This can be done using direct memory access or NCCL pipelined collective communication such as AllReduce for the gradient aggregation and Broadcast for parameter update. When comparing these methods, [7] found that on the Volta DGX-1 system that contained 8 interconnected GPUs and 2 CPUs, NVlink worked better for smaller networks with lower GPU counts, while the overhead of NCCL was amortized and proved to perform better with larger networks and more GPU nodes.

Nodes int the parameter server model can either wait for the new parameters to begin the next mini-batch iteration or begin right away, known as synchronous vs asynchronous stochastic gradient descent (ASGD). While A-SGD does not suffer from idle devices, it has the issue of decreased convergence time since the accuracy for each EPOCH is not improved as with synchronous. [21] finds that with A-SGD, distributed speedups can still be achieved, but it is better to use a lower number of GPUs for “Cold Start” training. This is because the different directions that parameters can take early in training average out to produce slow convergance rates.

With distributed systems, it is also important to note that device interconnects are often asymmetrical because of physical limitations such as distance and device ports. For example, in the DGX-1 system, certain GPU nodes have double the communication speed with neighboring GPUs than other nodes, and sometimes requires an intermediate GPU or CPU to facilitate communication. This plays a large role in device utilization and collective communication operations, and should be carefully considered when constructing workflows that could potentially perform latency hiding by overlapping communication with computation.

Additionally, this brings up usage of heterogenous system utilization such as utilizing the CPU when the GPUs are computing, a method proposed in [22].

DNN with Cross Platform GPUs

The vast majority of DNN GPU implementations are built exclusively for Nvidia devices using CUDA code and libraries such as cuDNN, or frameworks that utilize these interfaces under the hood. This misses the opportunity to operate NNs on heterogenous, mobile, embedded and low power systems that cover many important high performance NN applications. OpenCL is a framework that allows parallel programming for CPUs, all vendor GPUs, Digital Signal Processors (DSP), and FPGA. OpenCL resembles CUDA, and a NN implementation can be implemented using OpenCL GEMM operations. [8] shows the successful implementation of a CNN on both a mobile GPU device and an FPGA. Using OpenCL, the kernels in the FPGA can directly pass input and output layers through OpenCL piping / channeling, overcoming the lack of FPGA on board memory. [23] presents a PipeCNN, a specialized OpenCL framework for CNNs specifically for FPGAs, however an OpenCL framework for general NN implementation has yet to be introduced.

Energy Usage of DNN training on GPUs

One area of consideration when discussing high performance GPU computing is the energy efficiency of such computations. While performance, speed, and accuracy of models seems to be the driving factor behind NN training, this often occurs at the cost of high energy consumption. [9] evaluates the energy efficient of deep CNNs on CPUs and GPUs to determine overall energy usage as well as per-layer power consumption. Findings show that when comparing Conv, Pool and FC layers, Conv are much more computationally intensive even though the contain less parameters. However, this does not mention if convolutional optimizations described in [2] were used. Four CNNs were implemented using 5 different DNN frameworks on a 16-core Intel Xeon E5 2650 v2, and Nvidia Tesla K20m with 5GB memory and an Nvidia Titan X with 12 Gb of memory. Factors such as optimization libraries, network topologies, hyper-threading, Error Correcting Code and Dynamic Voltage Frequency Scaling (DVFS) were modified to determine their impact on energy usage. When using cuDNN over

native kernel code, the power consumption increased by 16% while the energy consumption reduced by 42%. ECC was found to not be necessary in NN training because of the networks tolerance for errors, so removing it leads to more memory availability and therefore greater performance. In general, anything that increases hardware utilization and power consumption leads to lower energy consumption, because it reduces the amount of total training time. DVFS was found to have high energy consumption in the low and high ends while the middle range had optimal energy efficiency. To minimize the energy consumption of DNN training, models must be built that prioritize with both software and hardware utilization over fine grained accuracy improvements.

Future of DNNs - GPU vs FPGA

While GPUs are the hardware of choice for raw NN performance, they require specific hardware support for experimenting with different network variations, specifically sparce (pruned) networks and compact data type networks such as BNNs, TNNs and APNNs. An FPGA is not only energy efficient compared to GPU implementations, but recently Intel GPUs such as the Stratix 10 have theoretical raw FP performance up to 9.2 TFLOP/s [10]. A customizable hardware architecture template for FPGAs is presented in [10] that allows for GEMM computation for Conv, FC, ReLu, BatchNorm and Pool). Results show that GEMM operations for pruned, Int6 and BNNs are 10%, 50%, and 5.4x faster on the Stratix 10 FPGA than on the Titan X Pascal GPU. Additionally, the Ternary 2-bit ResNet model on the Stratix 10 can deliver 60% better performance over the Titan X Pascal GPU using the FP32 ResNet while being 2.3x better in performance/watt and within 1% of accuracy. Because of their energy efficient and highly customizable design, FPGAs may me the hardware of choice in the future to not only experiment with different NNs but implement them as well. For now, GPUs remain the dominating accelerator, and NN support from vendors makes them the device of choice.

III. Acknowledgements

I would like to thank Dr. Kaeli for teaching relevant and interesting topics in High Performance Computing, and having high expectations for standards for all of his students. His expertise in the field and importance to cutting edge HPC research is apparent and inspiring.

IV. REFERENCES

[1] Oh, Kyoung-Su, and Keechul Jung. "GPU implementation of neural networks." *Pattern Recognition* 37.6 (2004): 1311-1314

[2] Strigl, Daniel, Klaus Kofler, and Stefan Podlipnig. "Performance and scalability of GPU-based convolutional neural networks." *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2010.

[3] Li, Boxun, et al. "Large scale recurrent neural network on GPU." *2014 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2014.

[4] Wang, Linnan, et al. "Superneurons: Dynamic GPU memory management for training deep neural networks." *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 2018.

[5] Feng, Boyuan, et al. "Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021.

[6] Mu, Jiandong, et al. "A history-based auto-tuning framework for fast and high-performance DNN design on GPU." *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020.

[7] Mojumder, Saiful A., et al. "Profiling dnn workloads on a volta-based dgx-1 system." *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018.

[8] Li, Shuai, et al. "Heterogeneous system implementation of deep learning neural network for object detection in OpenCL framework." *2018 International Conference on Electronics, Information, and Communication (ICEIC)*. IEEE, 2018.

[9] Li, Da, et al. "Evaluating the energy efficiency of deep convolutional neural networks on CPUs and GPUs." *2016 IEEE international conferences on big data and cloud computing (BDCloud), social computing and networking (SocialCom), sustainable computing and communications (SustainCom)(BDCloud-SocialCom-SustainCom)*. IEEE, 2016.

[10] Nurvitadhi, Eriko, et al. "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?." *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*. 2017.

[11] G.Campanella et al., “Clinical-grade computational pathology using weakly supervised deep learning on whole slide images,” Nat. Med. 25, 1301–1309 (2019).

[12] M.Gardner et al., “AllenNLP: A deep semantic natural language processing platform,” in Proc. Workshop NLP Open Source Software, 56th Annual Meeting of the Association for Computational Linguistics, 2018, pp. 1-6.

[13] Z.Chen and X. Huang, “End-to-end learning for lane keeping of selfdriving cars,” in Proc. IEEE Intelligent Vehicle Symposium, 2017, pp. 1856-1860.
[14] J.Mahler et al., “Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics,” in Proc. Robotics: Science and Systems Conference, 2017.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. MIT Press, 2016. http://www.deeplearningbook.org.

[16] P. Y. Simard, D. Steinkraus, and J. C. Platt, “Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis,” in *Proc. of the 7th Int. Conference on Document Analysis and Recognition*, 2003, pp. 958–962.

[17] K. Chellapilla, S. Puri, and P. Y. Simard, “High Performance Convolutional Neural Networks for Document Processing,” in *Proc. of the 10th Int. Workshop on Frontiers in Handwriting Recognition*, 2006.

[18] M. Bode´n, “A guide to recurrent neural networks and backpropagation,” *The Dallas project, SICS technical report*, 2002.

[19] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. [n.d.]. Quantized Convolutional Neural Networks for Mobile Devices. In CVPR’16.

[20] Zhu, Hongyu, Amar Phanishayee, and Gennady Pekhimenko. "Daydream: Accurately Estimating the Efficacy of Optimizations for {DNN} Training." *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020.

[21] Paine, Thomas, et al. "Gpu asynchronous stochastic gradient descent to speed up neural network training." *arXiv preprint arXiv:1312.6186* (2013).

[22] Jiang, Yimin, et al. "A Unified Architecture for Accelerating Distributed {DNN} Training in Heterogeneous {GPU/CPU} Clusters." *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020.

[23] Wang, Dong, Jianjing An, and Ke Xu. "PipeCNN: An OpenCL-based FPGA accelerator for large-scale convolution neuron networks." *arXiv preprint arXiv:1611.02450* (2016).