Navigating the Design Space of Reconfigurable Neural Networks Accelerators

Mario Paulo Drumond
PARSA, I&C, EPFL

***Abstract*—Neural Networks are an important class of algorithms used in many machine learning tasks, such as image classification and speech recognition. These algorithms are compute-intensive and its users often need heterogeneous acceleration to achieve satisfactory performance. We survey the landscape of heterogeneous acceleration for Neural Networks, comparing three classes of accelerators, GPUs; ASICs; and FPGAs, according to three factors: flexibility, energy-efficiency, and scalability. We show that GPUs are flexible but suffer from poor energy-efficiency, ASICs are energy efficient but inflexible, and FPGAs can achieve the flexibility of GPUs and near ASIC energy-efficiency.**

***Index Terms*—FPGA, convolutional neural networks, accelerators**

I. Introduction

Machine learning algorithms are widely used, either by the scientific community or Internet services. These algorithms find applications in various fields, from web search to stock market forecasting, often operating over datasets measured in terabytes.

Neural networks are one example of a machine mearning emerging workload. They have been shown to perform well in many tasks like speech recognition and image classification. Moreover, they have been used to implement approximate computations [1], even approximating some algorithms in the PARSEC benchmark [2].

The complexity of a neural network can be roughly estimated by its number of synaptic weights. This number can range from a few hundreds to billions of weights [3], and even the biggest artificial networks can still be considered small when compared to the human brain, where just the visual cortex can have hundreds of trillions of synapses.

These algorithms are highly parallel and compute intensive, and users have often turned to heterogeneous acceleration, in the form of GPUs, looking for satisfactory performance in tasks like image classification.

Thus, to efficiently train and process data with neural networks, heterogeneous accelerators are becoming the norm, be it in scientific high-performance computing (HPC) or datacenter settings. We identify three main requirements for these heterogeneous accelerators:

- **Energy-Efficiency:** because processing bigger networks can impose prohibitive energy requirements;
- **Flexibility:** because researchers need to take advantage of acceleration in novel neural network algorithms;
- **Multi-node Scalability:** because neural networks' sizes should not be limited by chip die area constraints.

In this work we survey the neural networks accelerator landscape, and study the trade-offs involved in building efficient accelerators over three different computing substrates: GPUs, ASICs and FPGAs. We also present a research proposal that applies the knowledge from other accelerators to build a flexible and efficient FPGA accelerating framework.

The rest of this work is organized as follows: Section II introduces Neural Network algorithms and its computational requirements, Sections III, IV and V study GPU, ASIC and FPGA accelerators for Neural Networks and Sections VI and VII studies how can we use lessons learned in other accelerators to build an energy-efficient, scalable FPGA accelerators.

II. Introduction to Neural networks

Neural networks are a class of algorithms that mimic the brain's networks of neurons. They are generally composed by an input layer, one or more hidden layers and an output layers, and layers are connected by synapses.

Neural networks must be trained, by using known pairs of inputs/outputs to calculate its synaptic weights. Afterwards, networks can be queried with previously unseen inputs, in a process is called inference.

Proposal submitted to committee: May 27th, 2015; Candidacy exam date: June 3rd, 2015; Candidacy exam committee: Prof. James Larus, Prof. Babak Falsafi, Prof. Paolo Ienne.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(B. Falsafi) (signature)

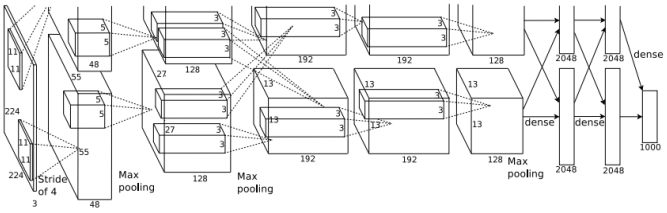

Fig. 1. Example of Convolutional Neural Network

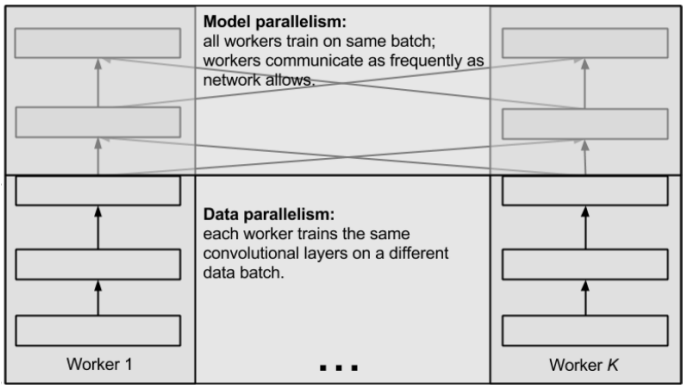

Fig. 2. Types of parallelism in neural networks

There are many types of neural networks, and in this work we focus on Convolutional Neural Networks (CNNs) and Deep Neural Networks (DNNs).

CNNs are networks where each neuron is the result of a convolution of a kernel with the previous layers' neurons. A layer consists of many sub-layers, called feature maps and a kernel is composed by a number of synaptic weights that are learned during the training phase. All neurons of an output CNN layer share the same weights (i.e., the same kernel is convoluted with the input layers), but neurons from different output feature maps do not share kernels.

DNNs are similar to CNNs, but kernel weights are not shared by output neurons, thus, they have a higher number of weight values.

In figure 1, we see an example of network that won the Imagenet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 [4]. It is composed by five convolutional layers followed by fully connected layers. Each convolutional layer is followed by a pooling layer. For instance, the first convolutional layer of the network has kernel size 11x11x3, with an input layer of size 224x224x3 and an output layer of size 55x55x48.

This network has three types of layers: convolutional, pooling and fully connected. There are indeed many types of layers, and in this work we are going to focus on convolutional layers, as they are representative of the challenges involved in accelerating neural networks computations.

The two main algorithms involving neural networks are training and inference, and they often must be executed in multiple computing nodes to achieve satisfactory throughput. CNNs and DNNs use similar training and inference algorithms, and the parallelization techniques applied to them are also similar but face different constraints, as we will see.

A. Inference and Training

Inference is done by propagating inputs trough the network layers.

Training is done by initializing weights with some lightweight computational method, and then forward propagating training data through the network. The error is then calculated against the training labels and backpropagated through the layers. Finally, the weights are updated based on the calculated errors. This process is repeated for the entire training dataset.

Therefore, the training algorithm comprises three phases of similar complexity: forward propagation (inference), backpropagation and weight update. To increase performance, training is done in mini-batches, where training inputs are batched together, back propagated errors are accumulated for each batch, and the third phase of the algorithm, weight update, occurs only once after each mini-batch, using the accumulated errors.

Unsupervised training uses a cost function to calculate the reconstruction error of the output layers, as opposed to calculating the error based on labels. The backpropagation and weight update phases are similar to the supervised training algorithm. It is worth noting that calculating the reconstruction error is often computationally expensive, and can dominate the computing time in training.

B. Distributed neural network acceleration

Neural Network training often involves millions of training inputs. Computational requirements can be high: Google used 3 days of a 1000 16-core machines to train an unsupervised network with 1.8 billion synaptic weights [3].

On the other hand, both the training and inference algorithms are highly parallel, and one can exploit of this parallelism by using a single accelerator or a distributed accelerator. With networks expected to grow by orders of magnitude it is necessary to achieve scalable, distributed acceleration, both for training and inference algorithms.

Distributed neural network processing can be achieved by exploiting two types of parallelism: model and data parallelism [5]. Figure 2 is a visual representation of these models. Model parallelism divides the network between workers with weights remaining in the same node throughout computations, and network inputs and intermediate values being moved between nodes. In this model, the amount of inter-node communication depends on connectivity layers. Data parallelism processes batches in parallel. Inputs and intermediate values remain in the same close to nodes. For training, weights have to be transferred between nodes during the weight update phase.

III. GPU Acceleration

Neural networks algorithm are modeled as matrix multiplications in GPUs. In this approach, the weights are organized in one matrix and the neuron values on other, and matrix multiplication calculates the output neurons. Users can use this approach to take advantage of heavily optimized matrix

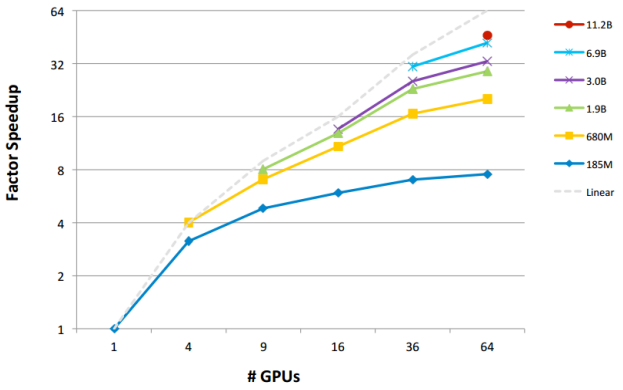

Fig. 3. Factor speedup obtained for varying sizes of network and number of GPUs, normalized for the size of the network.

multiplication libraries. In inference algorithms, some implementations can achieve near peak floating point performance in high-end GPUs, but some steps of the training algorithm are memory bound.

The mapping of DNNs to matrix multiplication often results in sparse weight matrices, but the sparsity patterns are known at development time. In these networks the programmer often faces the challenge of choosing between traversing the sparse matrices or performing additional operations to avoid traversing the sparse regions of the weight matrix. To mitigate this problem, Coates et al. [6] propose a novel approach, where filters are grouped in blocks, improving the matrix sparsity pattern. This change allows them to multiply the weight and neuron matrices in succession of dense sub-matrices multiplications.

A. Scaling and Distributed Computing

Scaling to multiple GPU nodes is often challenging. Due to inter-node communication, traditional ethernet networks quickly become a performance bottleneck. Furthermore, GPUs need big datasets to achieve high performance: they can only achieve high functional unit utilization with high data parallelism. With smaller datasets, GPUs fail to hide memory and control latencies, and cannot sustain high utilization. Thus, for both for model and data parallel approaches, processing smaller data-subsets results in reduced performance.

Coates et al. [6] proposed several techniques to improve GPU scalability, the main one being the use of off the shelf Infiniband high-throughput interconnects. They take advantage of DNNs' low connectivity pattern to exploit model parallelism with low communication overhead. In their implementation GPUs share a small number of neuron values, located on the borders of the regions mapped to two GPUs. Communication is not only minimized, but it is also local: GPUs will share values with other GPUs that hold adjacent morel regions. With DNNs, communication overheads scale sublinearly with number of nodes.

B. Evaluation

1) Energy efficiency: GPU accelerators are able to achieve orders of magnitude more throughput than general purpose CPUs, due to the high parallelism of the algorithm that can be properly exploited by GPU architectures.

GPUs are also more energy-efficient than CPUs, because they minimize von Neumann overheads.

However, there are some limitations to GPU efficiency. First, they need to maintain high functional unit utilization to be energy-efficient because the overhead of maintaining many GPU cores active dominates energy [7]. Furthermore, the extreme multithreaded model makes it hard for threads to share values. Values can be shared either directly through specialized scratch pad memories or indirectly through the cache hierarchy. Thus, some opportunities for data reuse are wasted; for example, two consecutive kernels cannot share on-chip buffers. The resulting higher number of high energy main memory accesses limits energy-efficiency, as shown in [8].

2) Scalability and its Limits: Coates et al. [6] showed that it is possible to achieve some level of scalability for neural network training, as depicted in 3. The authors are able to train a network that is similar to the one used in [3] using around 330x less machines. They achieve further scalability using bigger networks. They note that future neural network topologies can be designed in a way that improves scalability.

Scaling to multiple nodes allows researchers to train and test bigger networks, however, as shown in [9] there are limits to scalability. Some of them are inherent limitations in neural network distributed processing, but there are additional limitations in GPU scalability. The lack of an exclusive optimized accelerator network and the energy efficiency reduction are exclusive artifacts of GPU architectures, and other accelerators can be designed to avoid these problems, as shown by DaDianNao [10].

3) Flexibility: Coates et al. [6] build an scalable accelerator using only off-the-shelf components, and their code can be abstracted in a software library. Their system is flexible and easily portable.

Library based approaches have been embraced by the neural network community. The three most prominent Neural Networks frameworks (Caffe [11], Torch7 [12] and Theano [13]), have backends that integrate GPU libraries. Although distributed computation support for Neural Networks is still in its infancy, it is fair to assume that future distributed accelerators will have to consider the library-based approach used until now.

C. Summary of GPU acceleration

GPUs are flexible high-throughput accelerators that achieve higher energy-efficiency than CPUs. These benefits make them the most common computational platform used by Neural Network researchers. However, we showed that GPUs face limited scalability and some inefficiency. In the next sections, we study a couple of alternatives for Neural Network acceleration.

IV. ASIC Acceleration - DaDianNao

Chen et al. [8], [10] identified several bottlenecks for energy efficiency in GPUs and CPUs. In DianNao [8] they showed that proper loop tiling reduces memory accesses by orders of magnitude, minimizing energy overheads. They also showed

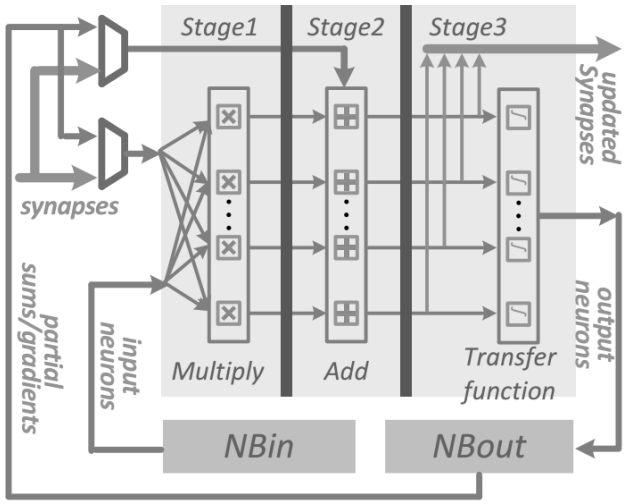

Fig. 4. A neural functional unit in DaDianNao

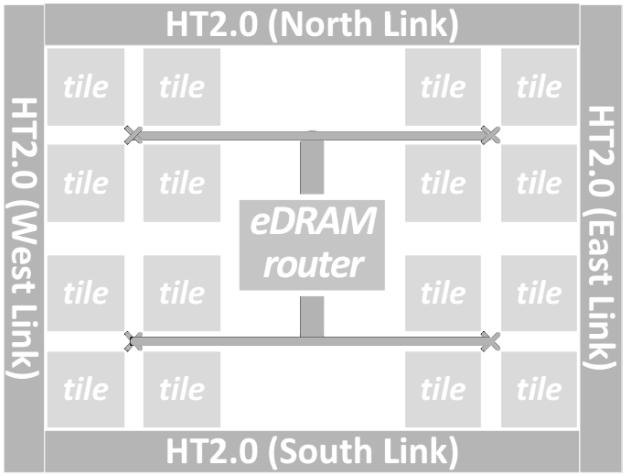

Fig. 5. Tile distribution within a DaDianNao computing node

that neural network inference can be achieved by operating over 16-bit fixed point number representation, as opposed to using expensive 32-bit floating point operations. They built an accelerator based on those lessons, carefully using on-chip buffers to exploit data reuse. Their results were promising, but the need for high energy main memory accesses limited their accelerators' energy efficiency.

In DaDianNao [10] they addressed these issues by proposing an scalable accelerator architecture, using multiple nodes to process networks that did not fit into a single chip memory, eliminating the need for main memory accesses.

This section will study the main insights of their design and the benefits of using ASIC accelerators in CNNs and DNNs.

A. Accelerator Architecture

Figure 4 presents the architecture a neural functional unit (NFU), the basic processing element of the accelerator. The NFU consists of a three stage staggered pipeline, where the first stage performs multiplications between input neuron values and synaptic weights, the second stage performs the sum of weighted values and the third stage applies the neurons sigmoid functions (transfer function) to the outputs. This architecture allows the NFU to exploit parallelism in the algorithm, by pipelining operations and processing many inputs in parallel.

The authors show that for inference, 16-bit fixed-point operators result in low output error, but for training, 16-bit operators prevent training convergence. Convergence is affected because the backpropagation step often operates over numbers that cannot be represented by 16-bit fixed point operators. Thus, the NFU can operate over 16-bit operators for inference and 32-bit for training. Other work [14] has confirmed that 32-bit floating point operators are not necessary for neural networks algorithms. Inference and training can be performed using operators with lower accuracy, and the authors of DaDianNao defer this investigation for future work.

A single chip, or processing node, comprises 16 tiles with one NFU and some buffer space each, as shown in Figure 5. Tiles are connected by a fat tree, and intermediate outputs are stored on tile buffers. At the end of a neuron computation, output values are routed through the fat tree to the central eDRAM router and then to the next NFU. An HT2.0 low latency interconnect allows inter node communication, with a bandwidth of 6.4GB/s in each of the four directions. Neural networks are distributed through the chip in a model parallel way: synaptic weights are kept fixed near the NFUs while neuron values are transferred between computing nodes. This design aims to achieve high scalability and reduce data movement for networks with a high number of synapses.

A NFU is fed from either other NFUs or its local buffers. On-chip buffers are built using embedded DRAM (eDRAM). This technology — an on-chip memory technology that is denser than SRAM — allowed the authors to put 36MB of on-chip buffers in a $66mm^2$ chip. This design choice is made to address the efficiency bottlenecks of DianNao [8], effectively eliminating the need for main memory accesses.

To eliminate the need for main memory, the accelerator needs to scale horizontally. For example, to implement the convolutional layer of the network proposed in [3], it needs 64 nodes. It is important to note that scaling horizontally incur additional inter node communication overhead.

The accelerator is programmed with a simple ISA. Instructions are used to configure NFUs, distributing the network across nodes and across NFUs within nodes. The ISA also contain buffer instruction to configure the NFU buffers and to set up data reuse. There are also instructions to configure NFUs to process different layers. For example, some layers do not require a sigmoid operation after the weight-neuron reduction, and thus, the last stage of the NFU pipeline needs to be turned off.

B. Evaluation

1) Energy efficiency: Chen et al. [10] evaluate the DaDianNao accelerator against a K20M GPU baseline, making performance and energy comparisons. Their GPU baseline run

cuda-convnet [15], a state-of-the-art neural network library for GPUs.

Their accelerator die area is $66mm^2$ of which 48% is eDRAM. This area is roughly 12% of a K20M die area. The estimated power, assuming 100% toggle rate is 15.97W, roughly 10% of a state-of-the-art GPU power consumption. Most of their accelerator power is spent in the HT interconnect and eDRAM blocks, with respectively 50.14% and 38.30% of the power.

The authors evaluate several distributed configurations of DaDianNao, ranging from single-node to 64-node accelerator configurations, comparing them against the GPU baseline. The average inference speedup range from 21.38x for a single-node system to 450x for a 64-node system, with convolutional layers dominating execution time. The average training speedup range from 12.62x for a 1-node system to 300x for a 64-node system. Here, scalability is better due to the higher computation per communication ratio.

Energy reduction trends are different: for inference, average energy gains range from 300x for single-node to 150x for 64-nodes, for training, the reduction ranges from 173x in single-node systems to 66x in 64-node systems. As the number of computing nodes increase, communication overheads become dominant over performance and energy.

Architecturally, DaDianNao does a thorough design space exploration and eliminates most of the inefficiencies from general purpose processors, and they are able to achieve better scalability than GPU accelerators.

2) Limitations of ASIC: The authors admit there is a risk associated to freezing algorithms in hardware, but this risk is minimized thanks to fast evolving hardware.

However, there are additional costs of using ASIC accelerators; its total cost of ownership can be high, and ASIC development costs can overshadow potential energy savings.

Moreover, high-performance computing or data-center clusters — where such accelerators are most useful — often execute complex algorithms, where only single stage of the computation is done by neural networks, severely limiting the usefulness of specific purpose accelerators.

The best option would be a compromise between energy-efficiency and flexibility, with a multipurpose accelerator that could potentially follow algorithmic evolution and also accelerate other workloads. Thus, FPGAs are a potential candidate for this task: their reconfigurability allows the acceleration of future neural network algorithms or even other workloads, and we show they are more energy efficient than GPUs.

V. FPGA Acceleration

The literature has many neural networks accelerators implemented in FPGA [16], [17], [18] but most of them fail to present a thorough design space exploration and fully exploit FPGA buffer space and computational capacity. The most prominent exception is [19], where the authors present a methodology to automatically explore the design space, find the optimal reconfigurable accelerator for a given layer and match the accelerator computational capacity with its memory bandwidth requirements, leveraging on data reuse to maximize

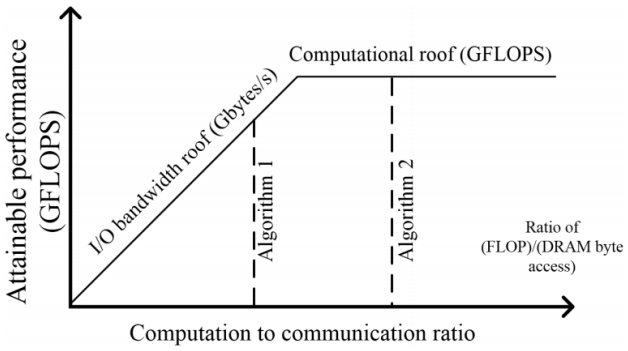

Fig. 6. Basis of the roofline model

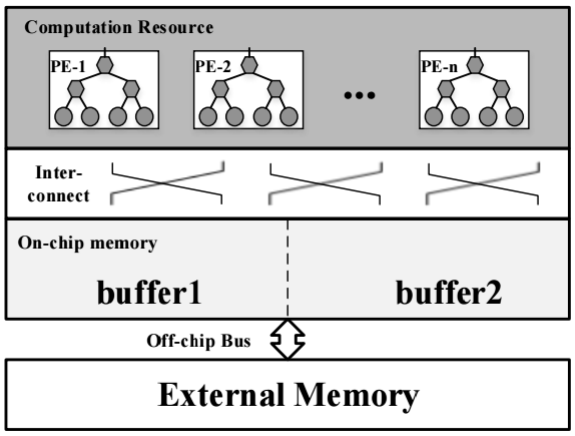

Fig. 7. FPGA accelerator architecture

```
for(row=0; row<R; row+=Tr) {
 for(col=0; col<C; col+=Tc) {
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
    //load output feature maps
    //load weights
    //load input feature maps

    for(trr=row;trr<min(row+Tr,R);trr++){
     for(tcc=col;tcc<min(col+Tc,C);tcc++){
      for(too=to;too<min(to+Tm,M);too++){
       for(tii=ti;tii<min(ti+Tn,N);tii++){
        for(i=0; i<K; i++) {
         for(j=0; j<K; j++) {
         L: output_fm[too][trr][tcc] +=
              weights[too][tii][i][j]*
              input_fm[tii][S*trr+i][S*tcc+j];
} } } } } }
    //store output feature maps
} } } }
```

Fig. 8. Pseudo code of a tiled convolutional layer

throughput. We closely study this work and compare it with ASICs and GPUs in the design space.

A. Roofline Model

Zhang et al. [19] identify computation and communication as the main constraints regarding system throughput. In order to balance these constraints, they propose a roofline performance model, as shown in figure 6. There, the "y" axis represents the maximum attainable performance, in floating point operations per seconds (GFLOPS), and the "x" represents the *computation to communication ratio* (CTC). The two roofs are shown in the picture: the bandwidth and computational roofs, and, for a given CTC, we can identify the maximum throughput and the minimum between the two roofs. For instance, in the figure, algorithm 1 would be I/O bound, and algorithm 2 would outperform it, fully utilizing the hardware computational potential.

However, actual accelerator implementations sometimes have to choose between achieving high throughput and exploiting reuse opportunities, and thus, a good methodology is needed to exploit the design space for FPGA accelerators. The authors develop a methodology to perform this analysis by exploring different optimizations of HLS code. They use polyhedral analysis to extract parallelism from the algorithm (to increase the accelerator throughput) and explore reuse opportunities (to increase the CTC ratio).

B. Accelerator Architecture

Their baseline computing subtract is shown in figure 7. It is composed by several processing elements (PEs), on-chip buffers, external memory, and on-/off-chip interconnect. Their on-chip buffers use double buffering, to allow overlap between communication and computation.

C. Computational and Memory Access optimizations

There are three main optimizations that the architecture can exploit: loop unrolling, loop pipelining and loop tiling. Figure 8 presents the pseudo-code of a tiled convolutional layer, divided in two sections: the outer loops that have impact on external data transfers and the inner loops that have impact over the on-chip computations. In this code, the inner loops will be targeted by loop unrolling and pipelining, while loop tiling will affect the structure of the outer loops, setting up data reuse in the PEs.

Loop unrolling allows one or more iterations of the loop to be executed by different processing elements. However, data dependencies across loop iterations need to be observed, since they will influence the interconnect between buffers and processing elements. Complicated dependency patterns incur unfeasible interconnects.

Loop pipelining allows loop iterations to be pipelined within a single processing element, enabling further parallelism, since loop iterations can overlap in the pipeline.

Finally, tiling is the most important optimization applied. Proper tilling dramatically reduces the memory bandwidth of an algorithm, since it allows the PEs to reuse the tile working set, as opposed to streaming the entire dataset from memory.

1) Design Space Exploration: Polyhedral-based optimization is used to identify all legal loop transformations, and pre-synthesis results are used to estimate resource utilization and performance. They choose the design with highest performance, and, in case two designs have the same performance, they choose the design with the lower memory bandwidth requirements.

The last challenge is to find a design that performs well in all network layers. Different layers have different optimal unroll factors, and thus, it is necessary to find a sub-optimal design that performs well for all layers. For their case study, they are able to identify one set of parameters with a worst case performance degradation of 5%, when compared to each layer optimal configuration.

D. Evaluation

Zhang et al. [19] evaluate their methodology with a case study that uses a board equipped with a Xilinx FPGA chip Virtex7 485t. They compare their design against an Intel Xeon CPU E5-2530.

They obtain the best performance when compared with previous reconfigurable neural network accelerators, attaining a throughput of 66 GFLOP/s. They are also able to extract the best performance density from the FPGA, even though they are the only ones to employ floating-point numeric representations in their computations. Their accelerator obtains 4.79x performance gains over 16-threaded CNN implementation on the baseline CPU, 5.1x less power consumption and 24.6x energy reductions.

1) Reconfigurable accelerators performance roof: Although the maximum performance obtained in the FPGA board seems disappointing when compared to GPU performance (hundreds of GFLOP/s), there are several factors to consider. First, the authors focus the methodology and in fast FPGA development using HLS, instead of focusing in presenting a single optimized implementations. One important design decision that reflects this focus was to include a soft-processor in the architecture, with IP interconnects. The alternative, namely using a dedicated block to handle configuration and control, would free up more resources for the accelerator.

For example, Microsoft [18] was able to obtain about 3x better performance on a FPGA accelerator, with slightly higher power (their power budget was 25W vs. 18W). Unfortunately there are not enough details on that design for a thorough comparison.

Furthermore, floating-point representations also limit performance; since floating-point operations are known to be the bottleneck of FPGA designs clock frequency. Their accelerator achieves slightly higher energy efficiency than GPUs (roughly 15% of the throughput with less than 10% of the power), but different design choices could trade programmability for better energy-efficiency. They also do not study scalable accelerators, and in fact, to the best of our knowledge no scalable FPGA neural network accelerators exist on the literature. In the next section, we propose a different set of trade-offs, sacrificing programmability to increase energy efficiency and achieve scalability.

Fig. 9. Energy efficiency table/graph

VI. Research proposal

TABLE I
PERFORMANCE COMPARISON BETWEEN GPUS, ASICS AND FPGAS ON DISTRIBUTED DNN TRAINING

Device	# Nodes	Normalized Performance $[Ops/(s*mm^2)]$	Energy Efficiency $[Ops/W]$
FPGA	1	1.97G	29.5G
	9	1.92G	28.8G
	64	1.92G	25.1G
GPU	1	1.75G	5.33G
	9	874M	2.90G
	64	436M	1.66G
ASIC	64	15.7G	148G

Zhang et al. [19] maximized computational throughput using polyhedral-based optimizations to explore the design space of convolutional networks accelerators. However, their exploration is limited in many dimensions, such as: multi-layers optimizations, training acceleration and distributed FPGA accelerators. These three additional design space dimensions, if properly explored, have the potential to change the CNN acceleration state-of-the-art from GPUs to FPGAs, further contributing to make FPGAs main-stream accelerators in both HPC and data-center settings.

Several challenges hinder the transition of FPGAs from niche to mainstream accelerators. First we need better system integration, and there are proposed solutions in the literature, like Catapult [20] and Intel/Altera HARP. Second, the exploration of such a rich design space requires better ways to express algorithms. C-like imperative representations limit the design space by imposing unnecessary ordering constraints. In order to expose all the options, we need a functional representation of neural network algorithms. These representations would allow us to properly analyze the space of possible operation schedules, or even traverse this space in a stochastic, non-deterministic way.

Finally we need show that, for neural networks, FPGAs can achieve the same or better performance than GPUs. Machine learning researchers have demonstrated openness to the idea of using heterogeneous systems in their studies by embracing GPU computing. If we provide them with a competitive alternative, we will take an important step towards turning FPGAs into mainstream accelerators.

We have addressed the last issue, by modeling the performance and energy-efficiency of GPUs, FPGAs and DaDianNao, in single node and 64-node distributed settings. Table I shows the results obtained during training of a mini-batch of the network used in [3]. The FPGA modeled was the same used in [18], with adjusted performance for 32-bit fixed point operators. The GPU is modeled with the same configuration as the one used in [6].

Table I shows that FPGAs can be an order of magnitude more efficient than GPUs, while achieving almost the same area normalized performance. They are still less efficient than ASIC accelerators, but they provide flexibility that is unmatched by ASIC accelerators.

VII. Conclusion

In this work we surveyed the landscape of neural network heterogeneous accelerators. We identified three classes of accelerators: GPUs, ASICs and FPGAs, comparing them according to three factors: flexibility, energy-efficiency and scalability. We showed that GPUs are flexible and scalable, however, they are not the best option as energy-efficient accelerators; ASICs on the other hand provide energy-efficiency and scalability, but are inflexible. Finally, we showed that FPGAs can be more energy-efficient and scalable than GPUs, while maintaining some level of flexibility.

Based on these findings, we propose the development of a scalable FPGA accelerator, in order to make FPGAs the de facto state-of-the-art in Neural Network acceleration, and to take an important step moving FPGAs from a niche accelerator to a mainstream general purpose accelerator.

References

[1] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pp. 449–460, IEEE Computer Society.

[2] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, "BenchNN: On the broad potential application scope of hardware neural network accelerators," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 36–45.

[3] Q. V. Le, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng, "Building high-level features using large scale unsupervised learning," in *In International Conference on Machine Learning, 2012. 103*.

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

[5] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," in *arXiv:1404.5997 [cs]*.

[6] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *Proceedings of The 30th International Conference on Machine Learning*, pp. 1337–1345.

[7] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 280–289, ACM.

[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pp. 269–284, ACM.

[9] H. F. Frank Seide, "On parallelizability of stochastic gradient descent for speech DNNS," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 235–239.

[10] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 609–622.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the ACM International Conference on Multimedia*, MM '14, pp. 675–678, ACM.

[12] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*.

[13] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," in *arXiv preprint arXiv:1211.5590*.

[14] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *arXiv:1502.02551 [cs, stat]*.

[15] A. Krizhevsky, "https://code.google.com/p/cuda-convnet."

[16] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 247–257, ACM.

[17] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pp. 109–116, IEEE.

[18] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," in *Microsoft Research Whitepaper*, vol. 2.

[19] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pp. 161–170, ACM.

[20] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pp. 13–24, IEEE Press.