DeepRecon: Dynamically Reconfigurable Architecture for Accelerating Deep Neural Networks

Tayyar Rzayev[1], Saber Moradi[2], David H. Albonesi[1], and Rajit Manohar[2]

[1]Computer Systems Laboratory, Cornell University, Ithaca, NY, USA , {tr265, dha7}@cornell.edu

[2]Computer Systems Laboratory, Yale University, New Haven, CT, USA , {saber.moradi, rajit.manohar}@yale.edu

ABSTRACT

Deep learning models are computationally expensive and their performance depends strongly on the underlying hardware platform. General purpose compute platforms such as GPUs have been widely used for implementing deep learning techniques. However, with the advent of emerging application domains such as internet of things, developments of custom integrated circuits capable of efficiently implementing deep learning models with low power and form factor are in high demand. In this paper we analyze both the computation and communication costs of common deep networks. We propose a reconfigurable architecture that efficiently utilizes computational and storage resources for accelerating deep learning techniques without loss of algorithmic accuracy.

1. INTRODUCTION

Mobile devices are becoming increasingly involved in our daily lives as diverse and active participants in our decision making. Using speech recognition and understanding at the level of intelligent response and language translation, visual recognition and scene understanding to take signals from the environment, and smart search of the vast knowledge database on the cloud, our devices are becoming sophisticated partners in our interactions. The majority of data analysis and processing that goes into supporting these systems resides in warehouse-scale computers and is communicated via the cloud infrastructure. Moving some of the processing from the cloud to the device would make our devices more intelligent, responsive and would save a lot of energy. That would require improving the efficiency of both the algorithms and the underlying mobile platform that performs the processing.

Over the past few years there has been a significant improvement in the algorithmic prediction accuracy on a number of visual recognition benchmarks. This result has come from the shift in performing the processing using deep neural networks as opposed to traditional hand-crafted computer vision methods. These networks have become the new state of the art for many of the applications in the vision pipeline such as object detection, classification, localization and labeling [1, 2, 3].

Additionally, there has been a number of applications where deep learning has been shown to perform relatively well end-to-end, where instead of performing one of the tasks in the pipeline, it consumes raw (or minimally processed) inputs and performs the entire task. One such example is the end-to-end speech engine [4], where the deep neural network replaces a number of models that normally work together. Another example is an end-to-end self-driving car implemented using a convolutional neural network (CNN) [5]. Here, the neural network takes camera inputs and produces steering wheel commands. While it is unclear whether deep learning models will be able to replace more traditional methods for such applications commercially, they have certainly earned their place in the design space.

Algorithmic prediction accuracy isn't the only important metric of interest, however. A significant challenge in implementing deep learning models is achieving high algorithmic prediction accuracy while meeting latency, throughput, and energy requirements. Throughput and energy consumption are crucial because they determine the performance and cost of a given solution as well as its viability given the constraints of the system. For instance, energy is a cost factor for warehouse-scale computing, while in mobile devices it is a resource constrained by battery life. Additionally, latency can be a critical design consideration for applications that operate under strict deadlines such as self-driving cars and soft real-time applications such as visual recognition in hand-held mobile devices.

Given the wide applicability of deep neural networks and their computational complexity, fast and efficient processing is very important. Inference (performing the predictions) usually has strict performance and power/energy constraints, while training (backpropagation for CNNs) is very long and costly. Hence, both processes benefit significantly from optimization. Two avenues of improvement have been pursued. On the one hand, more efficient algorithmic methods [1, 6, 7] have been proposed to converge to the same result either faster or cheaper on some of the metrics. The algorithmic methods include: moving towards simpler nonlinear functions, using more convolutional and pooling and less fully connected layers, and optimizing network size and data representation. Some of these are applicable only to inference, while others improve training as well.

On the other hand, more efficient hardware methods of computing have been proposed that are either fine-tuned to computing the algorithm efficiently or take advantage of specific algorithmic improvements. Thus, acceleration of deep neural networks on various co-processors has been another natural next step in achieving these goals. GPGPUs have been commonly used to obtain high speedups in both inference and training due to the prevalence of single-instruction multiple data (SIMD) parallelism. More recently, FPGAs have been shown to perform very well on inference with potential to accelerate training as well [8].

While GPUs and FPGAs provide significant speedups for deep learning computing, they are power hungry, an issue that can be addressed by special-purpose accelerators. There have been a number of DSP-like ASIC accelerators in the recent literature [9, 10, 11] that have been shown to deliver higher energy-efficiency than the programmable accelerators such as GPUs and FPGAs. However these accelerators have been designed with a fixed hardware configuration that is tailored to specific network features (for example, only convolution layers, or fixed kernel sizes, or fixed bit widths that work on a specific benchmark). These architectures suffer from being overfitted to the benchmark at hand and don't generalize well to a variety of networks that require different representations, or have different kernel sizes. Also many of these accelerators are specifically improved for inference and don't generalize well to training.

In this paper, we argue that in order to address those two shortcomings of the ASIC accelerators, the hardware must support some degree of reconfigurability. Given the appropriate design choices, the hardware could switch between a set of configurations that are more optimal on a per-benchmark basis with a low overhead. This would allow for dynamic reconfiguration that would tailor to benchmarks more generally than a fixed design. We also propose a hardware architecture that would be able to support training with minimal overheads, as efficient acceleration of training is an important problem to solve.

The main technical contributions of this paper are as follows:

1. **Quantization Tradeoffs in Hardware:** We analyze how quantization to various bit widths of fixed- and floating-point representations affects hardware performance and implementation costs as well as prediction accuracy.

2. **Reconfigurable Architecture:** We present a dynamically reconfigurable architecture that is capable of delivering more performance and efficiency to each benchmark as compared to a fixed design without loss of accuracy.

3. **Memory System effects:** We demonstrate the effects this reconfigurable design would have on the memory system, and show comparable improvements in memory bandwidth, caching, and energy.

4. **Training:** We address additional hardware requirements needed to include the support for training to full prediction accuracy.

In the next section we introduce some of the background of deep neural networks and discuss related work. In Section 3, we discuss the effects of quantization on the algorithm performance and compare fixed-point and floating-point data representations. Section 4 presents our proposed design, with evaluation results and discussion in Section 5. We conclude the paper and outline future work in Section 6.

2. BACKGROUND

In this section we discuss deep convolutional neural network models, including the algorithms for training and inference as well as our benchmarks. We also cover some of the related work in accelerating deep learning using application-specific integrated circuits (ASICs). When discussing algorithm and hardware implementations, we use the term "accuracy" to mean the prediction accuracy of a given algorithm, and "precision" to refer to the hardware bit-precision of the computation that implements the algorithm.

2.1 Deep CNNs

Current state-of-the-art deep convolutional neural networks consist of a number of layers processed sequentially from input to output. The layers can consist of the following common types:

- Convolutional layer: This layer is used for feature extraction. It has a number of filters of various sizes that are convolved with a given input channel (here input can refer to input to the network or output of the previous layer).

- Pooling layer: This layer does not have any stored filters, but rather performs a simple function (for example: min, max, or mean) across multiple inputs. Typically, this layer is applied after a convolutional or nonlinear layer.

- Nonlinear layer: This layer does not have any stored filters, but rather perfroms a nonlinear operation on each input. Currently, the most commonly used is the rectified linear unit (ReLU), which maps negative values to 0. Typically, this layer is applied after a convolutional or pooling layer.

- Fully connected layer: This layer is used for feature extraction, and is represented by a dense matrix that has an entry for every input-output pair. To compute the output the weight matrix is multiplied by the input vector; thus, each output receives a weighted sum of the inputs.

Equation 1 (Figure 1) shows that to compute the forward path of a fully connected layer, we need to multiply the

$$z^{l+1} = f((W^{(l+1)}) * a^{(l)}) \quad (1)$$

$$\nabla_{W^{(l)}} J = ((W^{(l+1)})^T \delta^{(l+2)}) .* f'(z^{(l+1)})(a^{(l)})^T \quad (2)$$

$$\theta = \mu\theta - \alpha\nabla_\theta J(\theta, x^{(i)}, y^{(i)}) \quad (3)$$

Figure 1: Equations for computing inference and training. Equation 1 represents computing forward path for one layer of a deep neural network. Z is the activation of this layer, a is the input to this layer, W is the set of weights. * is multiplication for a fully connected layer and convolution for the convolutional layer. *f()* is a pooling function, or a nonlinearity or both. Equation 2 shows gradient computation with respect to weights W of layer l for training. Here .* represents element-wise product. Equation 3 represents weight update via gradients during training.

weights with inputs from the previous layer (for a convolution layer, we need to convolve a sliding weight kernel across the inputs), and then apply a function. This function can be a pooling function, a non-linear function or both. The bulk of computation here is either a matrix-vector multiply, or a dot product.

Equations 2 and 3 show gradient compute and weight update with stochastic gradient descent that comprise the back-propagation algorithm. Here also the bulk of the computation is either a matrix-vector multiply or a dot product. Usually this is performed over a mini-batch of inputs, which leads to larger values being accumulated and the computation transformed into a matrix-matrix multiply. Also, the weight update in Equation 3 represents accumulating small values into large values. This is partially accomplished via parameters α and μ that represent learning rate and momentum of the stochastic gradient descent respectively [12].

Table 1 shows the set of benchmarks we used for the evaluation in this paper: mnist, cifar, and imagenet. For each benchmark we trained a deep network on a corresponding dataset as indicated in the table. For example, when we refer to mnist benchmark, that implies LeNet [13] network that was trained on MNIST dataset. Cifar10net is the CIFAR-10 full model developed by Caffe [14] for the corresponding dataset. Caffenet is the Caffe version of AlexNet [1].

2.2 Deep Learning Acceleration

As mentioned previously, there have been a number of fixed-point inference accelerators in the literature over the past few years [9, 10, 11, 15]. DaDianNao [9] is a supercomputer for machine learning that uses 32 bit fixed point representation for training on the MNIST dataset, and 16 bit fixed point for inference. They focus more on memory system design and routing rather than detailed analysis of the compute engine.

Eyeriss [11] is an architecture for acceleration of inference that uses row-stationary data locality and data sparcity

Benchmark	mnist	cifar	imagenet
Dataset	MNIST	CIFAR-10	ImageNet15
Network	LeNet	Cifar10net	CaffeNet
Conv Layers	2	3	5
Pooling Layers	2	3	3
FC Layers	2	2	3

Table 1: Benchmarks used in the analysis.

to claim an impressive 34 frames per second on the convolutional layers of AlexNet. The data bitwidth is assumed to be 16 bit fixed point for inference. Training is not supported.

Origami [10] is an architecture that focused on reducing memory bandwidth in order to enable the performance to scale up. According to Cavigelli et al., previous work isn't conclusive on the required precision for convolutional neural networks and that 16 and 18 bit fixed point units are most commonly used [10]. The authors performed a simple analysis of 12-16 bitwidths for layer inputs and outputs based on a very limited set of images (~700 images total for training and test). This architecture is based on 12 bit fixed point math and relies on specific filter bank sizes to achieve its performance and energy-efficiency.

NnX [15] is a coprocessor for accelerating deep neural networks and is optimized for multiple streams. This architecture is implemented in an FPGA rather than ASIC technology, but achieves impressive energy-efficiency improvement over multiple commercial CPU and GPU options. The authors implement an array of functional units consisting of one convolution, one pooling and one non-linear programmable function, which are all implemented in 16 bit Q8.8 format (further explained in Section 2). While this may save on a variable shifter unit as compared to using a variable Qm.n format, it limits the architecture to a fixed dynamic range and bitwidth.

All of these accelerators show improvements in energy-efficiency over GPUs and report either comparable or better performance results. However, they all assume a static fixed-point architecture using either 12 or 16 bits for inference. Our work explores the impact of bit precision by presenting a detailed analysis of the effects data representation and precision have on accuracy as well as on tradeoffs in performance, energy-, and area-efficiency. Additionally, given the wide range of applications of convolutional neural networks we demonstrate the benefits of a reconfigurable precision architecture.

3. ALGORITHM ANALYSIS

In this section we present an analysis of quantization of network parameters, activations and inputs and its effect on prediction accuracy for the chosen benchmarks. For each benchmark we trained a deep network on a corresponding dataset as per Table 1. For example, when we refer to the mnist benchmark, this means a LeNet network that was trained on the MNIST dataset, as shown in Table 1. We explore both fixed point and floating point representations for the quantization. Both representations show promising results to be

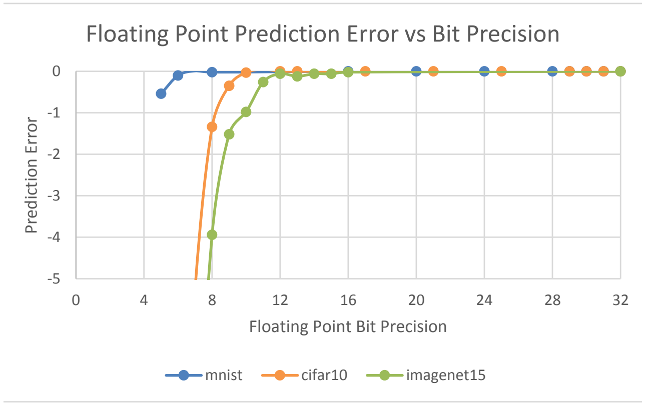

Figure 2: Floating point bit precision versus algorithm prediction accuracy for mnist, cifar, and imagenet benchmarks. Values on the y axis represent the difference of accuracy at a given precision on the x axis and accuracy at baseline precision (32 bit floating-point).

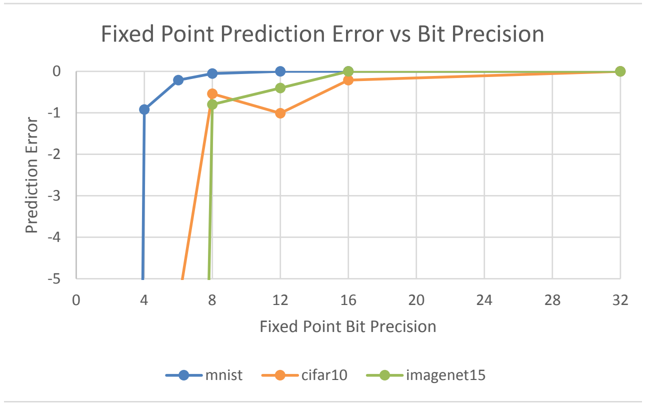

Figure 3: Fixed Point bit precision versus algorithm prediction accuracy for the mnist, cifar, and imagenet benchmarks. Values on the y axis represent the difference of accuracy at a corresponding precision on the x axis and accuracy at baseline precision (32 bit floating point).

exploited by special-purpose compute and memory architectures.

3.1 Quantization

The quantization is achieved using the following method, based on [16]. For floating point representations we pick the lowest exponent that can represent the whole data set assuming no limitation on the mantissa; then we search for the minimum mantissa bit width that doesn't degrade prediction accuracy. As can be seen in Figures 2 and 3, at bit widths close to the baseline (32 bit floating point representation), the prediction accuracy is constant. As we lower the bit width beyond 16 bits, we observe a decay in prediction accuracy, with the onset much earlier in fixed point representation in Figure 3. The specific accuracy decay is dependent on the network and the application. Arguably, additional gains can be achieved by improving the representation even further, but that is not the focus of our paper. For each fixed point quantization level we use different scaling factors for weights, and for activations of each of the layers that we find by sweeping.

3.2 Floating point

For the floating point representation each bit width is represented by several parameters, such as exponent, mantissa, and bias, usually represented as $1.exp.man. - bias$. To pick the best representation we compute the exponent based on the weight and activation distributions and search for the mantissa that preserves the network prediction accuracy. Optimal bias value(s) could also be adjusted, but this is out of the scope of this work. Instead we use symmetric bias, where $bias = 2^{(e-1)} - 1$. In Figure 2, values on the y axis represent the difference between accuracy at a given precision on the x axis and accuracy at baseline precision (single-precision floating-point). We observe that 6-11 bits is sufficient to achieve less than 0.5% accuracy loss as compared to the baseline for the selected three benchmarks. This number of bits depends on the benchmark-network pair used for the application, and might further increase beyond 11 bits for more complex applications.

3.3 Fixed Point

For the fixed point representation each bit width is represented by the Qm.n format, where m represents the number of effective integer bits, and n represents the number of effective fractional bits in the representation. Each layer's input, output and weights choose a separate m and n. This is helpful because weights have been observed to have much smaller values than layer activations, so weights would have a smaller m and larger n. In fact, m or n could even be negative which would shift the decimal point to either to the left-hand-side or to the right-hand-side of the fixed point number, respectively, which expands the range of representable numbers in a given number of fixed-point bits. If we try to optimize for smaller bit widths we get similar results as the floating point experiments as can be seen in Figure 3. Here it appears that 6-12 bits should be sufficient to achieve accuracies close to the 32 bit baseline.

3.4 Discussion

We compare the fixed and floating point representations with respect to the range of bit widths that is sufficient to perform inference with baseline accuracy. A more narrow bit representation may be cheaper to build in hardware; however, scalability is also important, i.e. the narrower the range of appropriate bit width across the benchmarks, the easier the design of the reconfigurable hardware.

We perform our comparison based on the analysis of Fig-

Benchmark	Minimum bit width	
	Fixed point	Floating point
mnist	6 bits	6 bits
cifar	8 bits	9 bits
imagemet	12 bits	11 bits
all	6 - 12 bits	6 - 11 bits

Table 2: Comparison of efficiency of fixed and floating point representations for inference on mnist, cifar, and imagenet benchmarks.

ures 2 and 3. We pick an error of 0.5% or less compared to the baseline as a cutoff for no loss of prediction accuracy. The comparison results are presented in Table 2. We can see that across the set of benchmarks the ranges of minimum bit widths are comparable with floating point representation doing slightly better.

Another aspect of comparing these representations is the additional hardware their respective architectures would need in order to support backpropagation training. For a floating-point architecture to support backpropagation (and be able to train to full baseline accuracy), we would need to add support for accumulation to full 32-bit. So the hardware requirement is a 32 bit floating point accumulator. The idea would be to perform most of the computation in the efficient narrower format - as in Equation 3 - and then extend to full precision when the learning rate would be applied as in Equation 2.

A fixed point based architecture would need to support a similar type of operation, which would entail a fixed point to floating point conversion hardware in addition to a 32 bit floating point accumulator, which is reflected in Table 3.

This analysis motivates a discussion on the accelerator architecture that is reconfigurable on a per benchmark (network + dataset) basis. This would achieve higher performance and energy-efficiency as compared to a full-precision baseline, yet maintain full prediction accuracy, unlike a fixed narrow bit width architecture. That is because the wider functional unit can be broken down into several narrower units and work in parallel when narrower bit width suffices, or recombine into a wider multiplier when the application demands a wider dynamic range to represent the values in flight. A range of appropriate floating-point bit widths (that depend on network-dataset pairs) dictates that the architecture support a wide range of bit widths in its compute pipeline as well as be able to pack them efficiently into the memory. Thus, the shorter the representation that suffices for a certain benchmark - the more computations we can perform in parallel per unit area and energy, since we would be reusing the same hardware with some small overheads.

4. HARDWARE ARCHITECTURE

In this section we discuss arithmetic units that can be dynamically fractured into various precisions based on benchmark demands. Based on our analysis from the previous section, this permits computing us to compute the network's output faster with less energy without loss of prediction accuracy. We evaluate the architecture for inference (forward path) and discuss extensions to efficiently support backpropagation training.

Representation	Energy-efficiency	Area-efficiency	Training requirements
16 bit fixed point	1	1	Converter FL32⊕
8 bit fixed point	5.92	2.10	
16 bit floating point	2.30	2.23	FL32⊕
8 bit floating point	41.3	12.7	

Table 3: Comparison of efficiency of fixed and floating point hardware for inference. Energy-efficiency is computed in (Gops/s/W) a.k.a (Gops/J) and area-efficiency in (Gops/s/mm2). Then all results are normalized to 16 bit fixed point hardware to simplify comparison. FL32⊕ should be read as 32 bit floating-point adder.

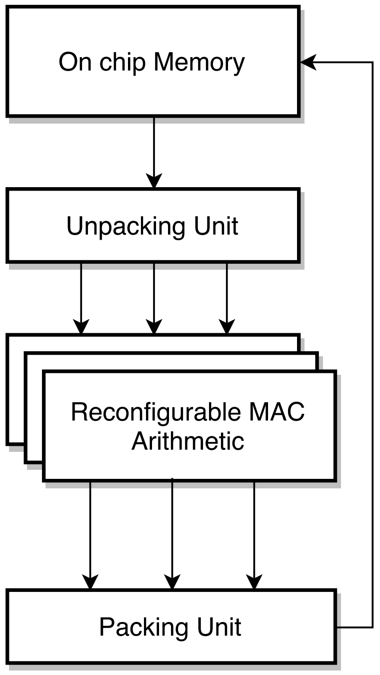

Figure 4: Reconfigurable accelerator architecture. Data from memory is unpacked and distributed among the appropriate lanes of the arithmetic unit. After the compute is done, the final results are packed again and stored back to memory.

4.1 Hardware Costs

Contrary to conventional wisdom, where floating-point hardware is considered more costly than fixed-point, we use a simplified non-IEEE representation of floating-point, which ends up producing cheaper multipliers than fixed-point. The reason is that this representation has much simpler normalization logic and for a fixed number of input bits, it uses a narrower multiplier than a fixed-point representation. For instance, a 16 bit floating point multiplier that is represented by 5 bits of exponent and 10 bits of mantissa would only

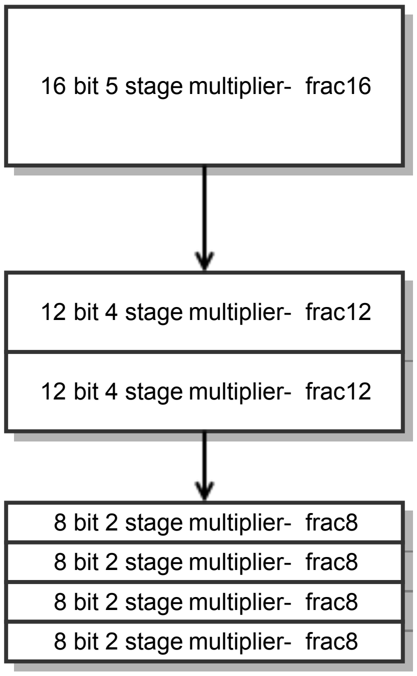
16 bit 5 stage multiplier- frac16
12 bit 4 stage multiplier- frac12
12 bit 4 stage multiplier- frac12
8 bit 2 stage multiplier- frac8
8 bit 2 stage multiplier- frac8
8 bit 2 stage multiplier- frac8
8 bit 2 stage multiplier- frac8

Figure 5: Reconfigurable multiplier architecture. Due to resource sharing the same fully-pipelined multiplier can either perform one 16 bit multiplication in 5 cycles, or two 12 bit multiplications in 4 cycles, or four 8 bit multiplications in 2 cycles.

need a 10-bit multiplier and a five bit adder, while a 16 bit fixed-point multiplier would need a wider (16 bit) multiplier, which is slower and/or more expensive in area and power. We list the comparative costs in Table 3. A 16 bit floating point multiplier is 2.3X more energy-efficient, while being 2.2X more area-efficient than its fixed-point counterpart. As we lower the number of bits, that gap increases as seen in the table. This observation, coupled with the efficiency of representation presented in the previous section, motivates the design of our reconfigurable floating point multiplier.

4.2 Arithmetic Pipeline

We propose the reconfigurable arithmetic pipeline seen in Figure 5, which is based on a 5-stage fully-pipelined 16-bit floating-point multiplier. We have implemented resource sharing across partial multipliers and adders to enable full range of floating point precisions from 6 bits to 16 bits. Currently the exponent is fixed at 5 bits, but we plan to support variable exponents in the future. As can be seen in Figure 5, the architecture supports a 16 bit base configuration and can fracture on demand into 2x12 bit or 4x8 bit floating point multipliers. We picked those design points as they naturally lend themselves to efficient resource sharing. Additionally, the architecture is capable of supporting all of the intermediate bit widths by using a higher bit width configuration and tying the low-order mantissa bits low. Currently this yields the same energy-efficiency as a higher bit width configuration; in the future we plan to support power gating to make the reconfigurable hardware more energy-efficient in those settings.

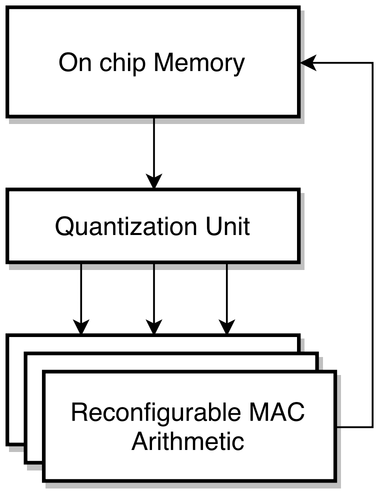

Figure 6: Reconfigurable accelerator architecture for training. Data from memory gets quantized and distributed among the appropriate lanes of the arithmetic unit. After the gradient math in low precision is done, the final results will be accumulated to full precision and stored back to memory.

4.3 Memory

In order to support variable precisions efficiently, the computed results are packed according to their bit widths into on-chip memory. Conversely, when computing the next layer, as those values are loaded back from memory, they are unpacked and distributed to the appropriate reconfigurable multiply-accumulate lanes as shown in Figure 4.

4.4 Extensions for Training

This architecture can be extended to support backpropagation training. Due to the fact that during training very small gradient updates need to be accumulated to modify the filter bank weights, the weight accumulation needs to occur at high precision. This update is indicated in Equation 3. Thus, on-chip memory needs to store high precision weights. As these weights are loaded from memory, they are quantized to the appropriate bit width, and then forwarded to the respective lanes of the arithmetic unit. The bulk of matrix-matrix multiply math still uses low precision, thus completing faster and using less energy. This phase is shown in Equation 2. Thus, the gradient computation can still be mapped to the reconfigurable arithmetic unit as can be seen in Figure 6. Once computed, the gradients are accumulated into the weight update in full precision and sent back to memory.

As previously shown in Sections 3.4, the architecture can be extended to support training using backpropagation by adding a quantization block and a full 32 bit accumulator. We would still receive significant gains due to efficient narrow multiplies, albeit with ammortized overheads of accu-

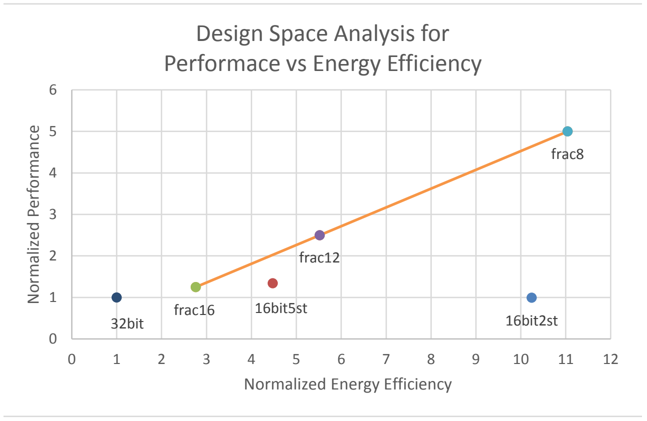

Figure 7: Floating point multiplier performance computed in $Gops/s$ versus energy-efficiency in $Gops/J$ normalized to the 32 bit design. Straight line connecting several design points represents the reconfigurable design. The other design points are baseline multipliers with various degrees of pipelining.

mulating into the weight update as given by Equation 3. We leave detailed evaluation of the gains the proposed architecture would deliver in the context of backpropagation training to future work.

5. DISCUSSION AND RESULTS

In this section we present evaluation results of the proposed reconfigurable accelerator design. First we discuss RTL-based evaluation of the arithmetic pipeline to determine the design that would deliver the most performance at the highest energy- and area-efficiency. We compare our proposed reconfigurable design to three baselines: one of which is the true baseline for the system (32bit), and the other two - narrower bit width more aggressive designs - with higher performance, energy-efficiency and area-efficiency. All of the design points used in our evaluation are described in Table 4. We present the analysis of the advantages of the reconfigurable architecture with respect to the memory subsystem, as we evaluate the effects on effective bandwidth and relative gains in memory access energy.

5.1 Arithmetic

We evaluate the reconfigurable arithmetic design and compare it to several baseline designs in terms of performance, energy-efficiency, and area-efficiency. Energy-efficiency and area-efficiency here represent how densely or cheaply we can perform the computation per unit cost (in other words, how many operations we can perform with one joule of energy or in one mm^2 per unit of time). The results are shown in Figures 7 and 8. On the x axis we show cost efficiency and on the y axis we show performance. The ideal design would be in the top right corner where both performance and cost-efficiency are maximized.

We built all of the designs in Verilog and synthesized them using a 28nm technology to evaluate their energy, delay, and

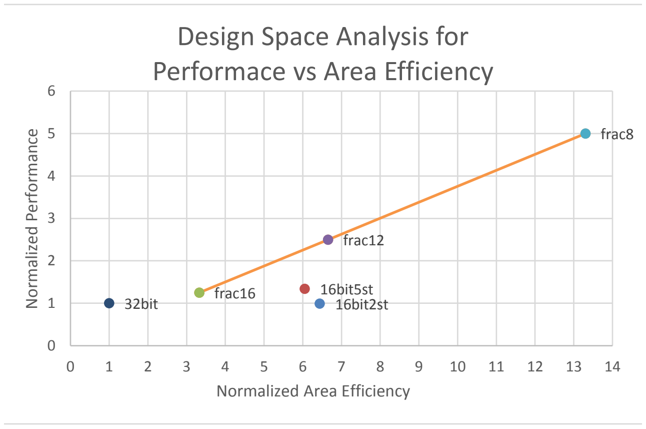

Figure 8: Floating point multiplier performance computed in $Gops/s$ versus area-efficiency in $Gops/s/mm^2$ normalized to the 32 bit design. Straight line connecting several design points represents the reconfigurable design. The other design points are baseline multipliers with various degrees of pipelining.

Design name	Bit width	Stages	
32bit	32	9	baseline
16bit5st	16	5	HP baseline
16bit2st	16	2	LP baseline
frac16	16	5	reconfig
frac12	12	4	reconfig
frac8	8	2	reconfig

Table 4: Design points presented in the evaluation with abbreviated names and corresponding details, such as bit width and number of stages. HP stands for high performance, LP stands for low power.

area. The main system baseline (32bit) a 32 bit (non-IEEE) floating point multiplier (just as in the algorithm studies). As can be seen in Figures 7 and 8, all of our configurations outperform the baseline by significant margins, so we developed two additional baselines. The first is a 2-stage 16 bit floating-point multiplier (16bit2st). The other is a more aggressively pipelined 5-stage 16 bit floating point multiplier (16bit5st). Comparing them to each other in Figure 7, we see that the 2 stage design is more energy-efficient, while the 5-stage design delivers more performance. Additionally, as we can see in Figure 8, they are similar in terms of area-efficiency. Thus, these two designs trade off performance for energy-efficiency for the same area-efficiency.

As mentioned previously, our reconfigurable multiplier is based on the more aggressively pipelined 16bit5st multiplier as it is geared towards more performance and resource sharing. As we can see in Figures 7 and 8, adding reconfigurability where highest configuration is 16 bits has 38% energy-efficiency, 45% area-efficiency, and 6.8% performance degradation when compared against the 16 bit fixed multiplier (16bit5st). However, that enables us to fracture the 16 bit multiplier into two 12 bit multipliers or four 8 bit multipli-

Benchmarks	Configuration	Accuracy loss	Memory gains			Arithmetic gains	
			Caching	Ops/S	$Energy/Access$	$Energy/Op$	$Perf/mm^2$
mnist	frac8(6bit)	0.10%	5.3	5.3	5.3	11.04	13.3
cifar	frac12(9bit)	0.35%	3.6	3.6	3.6	5.52	6.65
imagenet	frac12(11bit)	0.26%	2.9	2.9	2.9	5.52	6.65

Table 5: Full system analysis showing gains obtained from memory system and reconfigurable arithmetic normalized to 32 bit floating point baseline inference system (similar to a GPU).

ers. These designs both show improvements in performance and energy-efficiency over the baselines and in some cases are sufficient to deliver the required accuracy. However, it is still important to have capability of 16 bit floating-point math for more complex applications.

The 12-bit configuration point (frac12), delivers about twice the performance of the 16 bit baseline designs, while being slightly more area-efficient than either of them. It surpasses the energy-efficiency of 16bit5st, but is only about half as energy-efficient as 16bit2st multiplier as can be seen in Figure 7.

Finally, the most efficient is the 8 bit configuration (frac8), as it is the closest to the top right corner of both Figures 7 and 8. It delivers 3.7X and 5X the performance of the 16bit5st and 16bit2st baseline designs, respectively, while being about 2X more area-efficient than either of them. And while it is only 7% more energy-efficient than 16bit2st, it consumes 2.5X less energy per op as compared to 16bit5st.

To summarize, as seen in Figures 7 and 8, the 16 bit configuration is a bit less efficient than the 16 bit baselines (16bit2st and 16bit5st), which can be explained by the overheads due to the fracturing capability of the design. The 12 bit configuration (frac12) surpasses both baselines on all metrics, except it slightly lags behind the 16bit2st on energy-efficiency. Finally, the 8 bit configuration (frac8) is even further into the ideal corner, thus it surpasses all of the designs on all metrics. The lower bit width configurations of this architecture deliver more performance and energy-efficiency, yet it is crucial to support the whole range of bit widths because in some applications the lower bit width configuration is unacceptable due to accuracy losses from quantization. With added power gating capabilities, there will be more intermediate points between the three configurations that will save energy from the unused portions of the multiplier.

5.2 Memory

To evaluate the memory subsystem, we analyze the benchmarks - mnist, cifar, and imagenet. Their baseline memory footprints can be seen in Table 6 in Mebi-bits. For each benchmark, we picked the bit width from Section 3.4 with which we could perform inference with no loss of accuracy. Table 6 presents the respective quantized and packed memory footprints for each benchmark. This packing results in 2.9X, 3.6X, and 5.3X reduction in footprint. This footprint reduction has several effects.

First, the effective memory bandwidth (BW) increases 3-5X. This is because given the same Gbps BW interface, we will receive 3-5X more filter weights in the same amount of time. This is very desirable as we would require an increase in effective BW to fully utilize our lower precision configurations of the reconfigurable arithmetic unit. This footprint compression permits not only more efficient communication from memory to the functional units, but also effective local caching, since we can store 3-5X more values in local caches. Given an appropriate access pattern, this could allow for a quadratic increase in utilization (3-5X more values cached and 3-5X more values fetched means up to 9-25X more functional unit utilization). This should be sufficient to fully utilize the fractured configuration.

Benchmark	Baseline (Mib)	Bit width	Quantized (Mib)	% of baseline
mnist	13.5	6	2.53	19
cifar	2.68	9	0.75	28
imagenet	222	11	76.4	34

Table 6: Memory footprint analysis for baseline and quantized/packed deep neural network weights.

Secondly, packing also reduces the memory access energy per loaded value 3-5 fold. Given that the total memory access energy is obtained from

$$Energy_{memoryRef} = numberOfReferences * (Energy/Ref)$$

it follows that the total energy would be decreased by 3-5X as well, since our method does not affect the total number of references or the access pattern.

5.3 Full System

We can see full system evaluation of the reconfigurable architecture in Table 5. We observe a varying per benchmark 5.5-11X improvement in energy-efficiency at the same time as a 6.6-13X improvement in area efficiency. This means that using the same area as the baseline design, we could increase the performance by a factor of 6.6-13X and only have to pay 18-20% extra power. This would result in 5.5-11X energy saved on computation per benchmark. On the memory system side, we get 3-5X energy savings as well as 3-5X caching and BW improvement. These two collectively provide for up to 9-25X extra computation parallelism, assuming a favorable access pattern, which would be sufficient work for the 6.6-13X performance boost.

6. CONCLUSIONS AND FUTURE WORK

In this paper we present an analysis and design of reconfigurable architectures to accelerate deep learning applications. The presented architecture can take advantage of varying precisions required to achieve state-of-the-art prediction accuracy across various deep learning networks and datasets. Across the selected benchmarks our architecture delivers 6.6-13X improvement in performance density per unit area and 5.5-11X energy-efficiency improvement for the compute architecture. It also enables the memory system to improve energy use, bandwidth, and caching each by a factor of 3-5X. This leads to enough additional parallelism to feed the extra performance boost.

For future work, we propose to demonstrate the performance and efficiency gains of our reconfigurable architecture on training deep neural networks. We also propose to implement power gating across unused elements in order to achieve even more energy-efficiency and provide a more fine-grained control over input bit width configurations as well as support variable exponent sizes.

Acknowledgments

This research is supported by National Science Foundation grant CCF-1065307 and Samsung Corporation.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[2] Justin Johnson, Andrej Karpathy, and Li Fei-Fei. Densecap: Fully convolutional localization networks for dense captioning. *arXiv preprint arXiv:1511.07571*, 2015.

[3] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1915–1929, 2013.

[4] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[5] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars. *CoRR*, abs/1604.07316, 2016.

[6] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR*, abs/1602.07360, 2016.

[7] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR*, abs/1510.00149, 2015.

[8] E. Nurvitadhi, Jaewoong Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016.

[9] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society.

[10] L. Cavigelli and L. Benini. A 803 GOp/s/W Convolutional Network Accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, PP(99):1–1, 2016.

[11] Y. H. Chen, T. Krishna, J. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, Jan 2016.

[12] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer Texts in Statistics. Springer New York, 2014.

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[15] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 682–687, 2014.

[16] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented Approximation of Convolutional Neural Networks. *arXiv preprint arXiv:1604.03168*, 2016.