Design and Analysis of a Hardware CNN Accelerator

Kevin Kiningham
Stanford
kkiningh@stanford.edu

Michael Graczyk
Stanford
graczyk@stanford.edu

Athul Ramkumar
Stanford SCPD
athul@stanford.edu

Abstract

In recent years, Convolutional Neural Networks (CNNs) have revolutionized computer vision tasks. However, inference in current CNN designs is extremely computationally intensive. This has lead to an explosion of new accelerator architectures designed to reduce power consumption and latency [20].

In this paper, we design and implement a systolic array based architecture we call `ConvAU` to efficiently accelerate dense matrix multiplication operations in CNNs. We also train an 8-bit quantized version of Squeezenet[14] and evaluate our accelerator's power consumption and throughput. Finally, we compare our results to the reported results for the K80 GPU and Google's TPU. We find that `ConvAU` gives a 200x improvement in TOPs/W when compared to a NVIDIA K80 GPU and a 1.9x improvement when compared to the TPU.

1. Introduction

Since the remarkable success of AlexNet[17] on the 2012 ImageNet competition[24], CNNs have become the architecture of choice for many computer vision tasks. Inference on a trained CNN can be highly computationally expensive. A typical model might require billions of multiply-accumulate operations (MACs), load millions of weights, and draw dozens of watts during inference (see table 1). This computational cost can provide significant barriers to deployment; in [16], Google projected in 2013 that three minutes of voice search per user (implemented using deep neural networks) would require Google to double its datacenter size at the time.

In order to decrease execution time and power consumption, researchers and tech companies have investigated and built special purpose hardware for CNN inference. We present and analyze our own CNN accelerator `ConvAU`. Our architecture's main feature is a 256×256 systolic array of multiply-accumulate cells, allowing fast dense matrix-matrix and matrix-vector operations. We benchmark our system's latency, power, and performance using Squeezenet[14] trained on ImageNet[24] with Keras [4], quantized to 8 bits and executed using Tensorflow™ [1]. Finally, we compare our accelerator's throughput and power consumption to a GPU and Google's TPU.

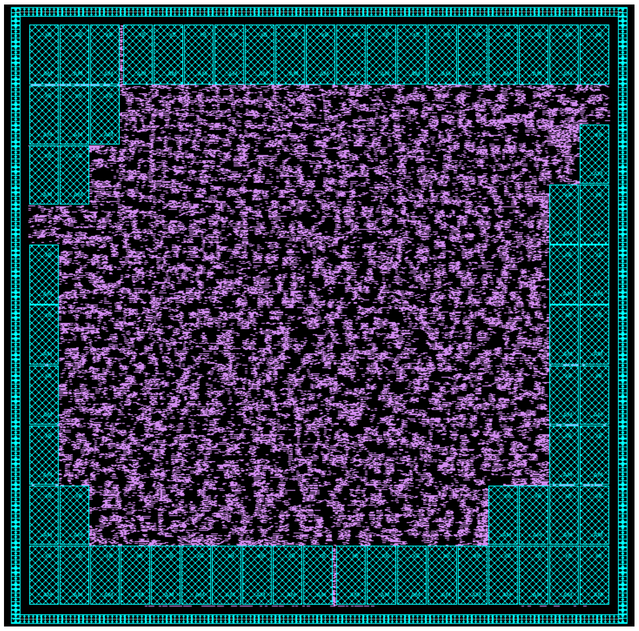
Figure 1: Final layout achieved after place-and-route

Table 1: Influential CNNs Benchmarked on a GTX 1080

Model	Total Weights	FPS	Avg Watts
InceptionV3	23,817,352	101.35	98
VGG16	138,357,544	169.65	120
ResNet50	25,583,592	207.05	100
SqueezeNet	1,235,496	388.76	80

2. Related Work

In recent years neural network accelerator design has been a topic of enormous interest to the computer architecture community. CNNs were traditionally executed on CPUs, GPUs, and even FPGAs [22], which can easily adapt to new network architectures. However, this flexibility comes at the price of efficiency. As a result, application specific accelerators have been developed to maximize efficiency.

Current approaches can mostly be classified into a combination of four categories, described below.

Dense Linear Algebra Accelerator The most common approach for CNN accelerators is to accelerate dense matrix-matrix and matrix-vector operations. Examples include DianNao[3] and Google's TPU[16]. Since dense linear algebra operations dominate the computational cost of CNN inference, these approaches have dramatically outperformed traditional CPUs and GPUs. However, one drawback is that many activations during inference are sparse, which means that some computation is wasted.

Quantization Another approach is to quantize all operations to a small number of bits[8, 28, 2]. Quantization can be used on CPUs and GPUs, but is most powerful in combination with hardware acceleration. Since small integer operations can be significantly more efficient to implement in hardware than floating point, this can dramatically increase the overall efficiency of the accelerator. However, this increased efficiency comes at the cost of decreased accuracy, although recent research has shown that by retraining the network this decrease can be made acceptably small[27, 6].

Sparse Linear Algebra Accelerators Some more recent approaches take advantage of sparsity. The EIE[10] and ESE[9] authors design an architecture that is able to operate directly on a sparse form, which allows them to avoid computation where the graph is zero. In addition, they encode the weight matrix in an efficient compressed format, which allows them to avoid loading zero weights from memory. In order to maintain accuracy, they perform pruning during training by removing weights below a threshold and then retraining, which allows the network to recover and only nominally affects accuracy[11, 12].

Analog Accelerators Finally, another approach that has gathered significant interest is to use mixed signal or analog computation. In [23], the authors show that mixed signal MAC operations can be performed efficiently using a switched capacitor design. In [19], the authors use a similar design to perform full matrix-matrix operations. Other approaches have included using resistive RAM elements [25] and flash transistors [7].

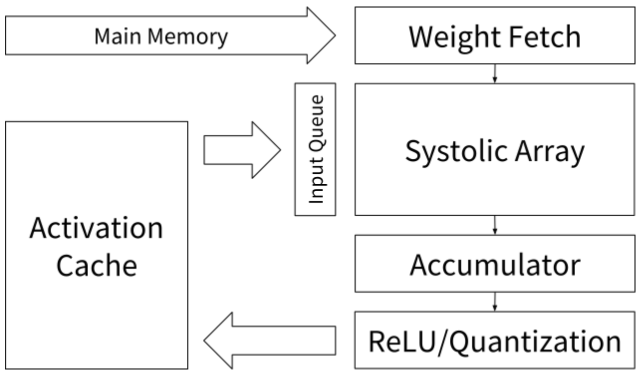

Figure 2: Overview of our entire design

One drawback is that analog and mixed signal accelerators require potentially expensive conversions between digital and analog representations. Additionally, analog designs tends to be larger, more error prone, more difficult to implement, and more costly to manufacture than the equivalent digital designs.

3. Accelerator Design

At it's core, our accelerator is a dense matrix multiplication unit (MMU) that can perform 256×256 8-bit integer multiply and 32-bit integer accumulate per cycle. Dense matrix multiplication acceleration has been researched extensively as it has been used in GPUs and DSP algorithms, with most common implementation methods being systolic arrays, FFTs, or the Winograd algorithm.

`ConvAU` uses a systolic array loosely based on Google's TPU[16]. A systolic array is a homogeneous grid of processing elements (PEs), each with a small amount of with each element connected only to it's neighbors. During execution, each PE can optionally read from it's neighbors, compute a simple function, and store the result in it's local memory.

We decided to use simple pipelining concepts to increase the throughput of the design. We have three main pipe stages: input/weight loading, matrix multiplication and activation (ReLu), and load results to output queue. Since the three operations are independent, we can schedule our operations to perform a matrix multiplication every operation cycle.

We chose to use a systolic array since it's easy to design and allows for an extremely efficient implementation. It also gives us a large amount of flexibility since it can accelerate any network architecture that uses dense matrix multiplication. For CNNs in particular, convolutions, batch normalization, and fully connected layers can all be efficiently implemented using dense matrix multiplications (see section 4). Together, these represent the vast majority of the

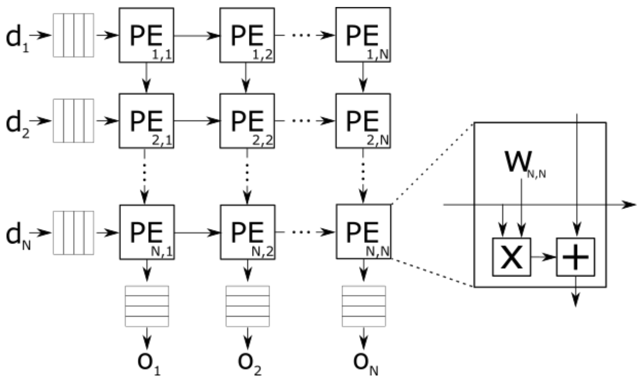

Figure 3: Our systolic array design. During execution, data flows in from the left and is multiplied by preloaded weights. Each column computes a partial product $o_i = \sum_{j=1}^{N} w_{ij} \cdot d_{ij}$ after $N \cdot i$ cycles.

computation required during inference.

3.1. Systolic Array Design

In `ConvAU`, each PE has a multiplier and an accumulator, and can compute a single MAC operation per cycle. Weights are stored locally before execution begins. For an $N \times N$ matrix multiplication, it takes $2N - 1$ clock cycles for the output wave to propagate and complete the multiplication. We use a 256×256 8-bit SRAM FIFO (64kB) to feed the data into the systolic array[5]. Instead of padding the data to synchronize data loading with the systolic array multiplier, we use a valid bit on each row to synchronize the data. This saves nearly 64kB of memory on the input and output.

3.2. Weight Loading

One important aspect of our design is that the weights must be preloaded into the array before multiplication can take place. To do this, we load the weights similarly to how we load the data, using a queue at the array edge and having each PE pass it's current weight to it's neighbor. This take 256 cycles to complete, although it can be pipelined with a matrix multiplication.

An alternative approach would be to instead flow the weights vertically down the array and compute the partial products at each node. However, this has the disadvantage of stopping execution at the completion of a 256×256 multiplication. Additionally, we would have to reread the weights from memory even if we wanted to reuse the same weights as the previous operation. Since weights are frequently shared (e.g. for filters in convolutional layers, or across batches of input) our approach should result in fewer memory accesses and pipeline stalls than the alternative.

3.3. Quantization

Artificial neural networks (ANNs) have been known for decades to contain the expressive power to approximate arbitrary real valued functions [13]. To run neural networks on real hardware, we must represent real network weights and activations in a finite way, amenable to efficient computation and storage. ANNs are typically implemented using floating point. Floating point directly corresponds to real numbers and handles large dynamic ranges without conceptual complexity. However, simplicity and elegance come at a cost. Large floating point matrix multiplies are more complex and less efficient in hardware than analogous integer operations. Companies using ANNs at scale have been exploring ways to use quantization to improve inference efficiency since at least 2011 [26].

Experiments have shown that the dynamic range and precision afforded by floating point are unnecessary for many deep learning applications. In particular, ANN inference can be performed using fixed-point matrix multiplies without significant loss of accuracy [6]. Other experiments have found seemingly contradictory results [18] when considering GPU computation. Still, since ANN inference can be executed in fixed point with little loss of precision and since fixed point hardware is simpler to design than floating point, we decided to only support quantized networks.

We chose our quantization strategy by deciding on implementation requirements, then finding a technique which met them and was simple to design. First, we decided against implementing floating point hardware for the reasons mentioned above. *Any floating point computation must be off-chip*. This guided us toward a design which involved precomputing quantization metadata at train time, then loading that metadata onto the chip along with the weights.

Second, we considered how best to avoid losses of accuracy during quantization. Uniform quantization can waste precision because weights are typically not distributed uniformly throughout their dynamic range. Although research into non-uniform quantization shows promise [11], we found that MAC operations on uniformly quantized values could be implemented using a systolic design. MACs on non-uniformly quantized values would require special operations and would have severely complicated the hardware. *Use uniform quantization with static representations.*

Finally, we decided to represent quantization metadata for each matrix as a 32-bit scale and 16-bit signed offset. We take our representation from the Google library gemmlop [15] in order to keep our hardware compatible with the quantization used in Tensorflow™ (which uses gemmlowp for quantized operations) and because of several important facts about CNNs identified in the gemmlowp documentation. First, CNNs often require zero-padding, so *the real value zero must be exactly representable in any quantized* *activations.* Second, our hardware supports ReLU activations, often used by CNNs. ReLUs can be implemented simply and efficiently in hardware using the gemmlowp quantization strategy. *ReLU must be efficient to compute.*

In the next few sections, we describe our quantization strategy in detail. We will use a simple two-layer feed forward ReLU network as an example. We assume this network has already been trained to arbitrarily high floating point precision. Specifically, let $X \in \mathbb{R}^N$ be the real valued neural net input, W_1, b_1, W_2, b_2 the weights and biases of the first and second layers. The outputs y are

$$\begin{aligned} h &= \max(0, xW_1 + b_1) \\ y &= hW_2 + b2 \end{aligned} \tag{1}$$

3.3.1 Quantizing a Network

Activations and weights must be quantized to use 8-bit arithmetic. Each real value ξ in the network is represented by 8-bit value ξ_q through the formula

$$\xi = \xi_s(\xi_q - \xi_z), \tag{2}$$

where ξ_s is the scale parameter and ξ_z the zero point for ξ.

This formula applies to both weights and activations. The values for z_s and z_z are specific to each weight or activation matrix. In our example, we choose values

$$\begin{gathered} (x_s, x_z), (h_s, h_z), (y_s, y_z) \\ (W_{1q}, W_{1s}, W_{1z}), (b_{1s}, b_{1z}) \\ \text{and} \\ (W_{2q}, W_{2s}, W_{2z}), (b_{2s}, b_{2z}). \end{gathered}$$

We constrain the bias scale to equal the products of the weight and activation matrix scales, $b_{ns} = x_{ns}W_{ns}$ for all layers n.

For our experiments, we used the Tensorflow™ tool `quantize_graph` to choose these quantization parameters. This tool uses execution traces to determine parameters that determine the dynamic of activations and chooses parameters to quantize parameters into those ranges. More sophisticated methods which globally optimize for accuracy, as in [21] could possibly yield better results, but these simple quantized results were close enough to baseline in our tests.

3.3.2 Quantized Computation

`ConvAU` uses 8-bit multiplies and uses 32-bit accumulators for all intermediate values. The host device loads the quantized weights and all quantization metadata listed above to `ConvAU` for processing. The scale parameters are not sent directly. Instead, a derived quantity $\frac{h_{ns}w_s}{h_{(n+1)s}}$ is computed for the boundary between each layer and represented by a 32-bit multiply followed by a right shift. The floating point inputs x must also be quantized to $x_q = x/x_s + x_z$ before sending. On the device, we need to compute h_q. Letting w denote the i^{th} column of W_1 and $\tilde{h} := xW_1$,

$$\begin{aligned} \tilde{h}^i &= x \cdot w = \sum (x_s(x_q - x_z))(w_s(w_q - w_z)) \\ &= x_s w_s \sum (x_q - x_z)(w_q - w_z) \\ &= x_s w_s \Big(\sum x_q w_q - \sum x_z w_q - \sum x_q w_z + \sum x_z w_z\Big) \\ &= x_s w_s \Big(\sum x_q w_q - x_z \sum w_q - w_z \sum x_q + N x_z w_z\Big), \end{aligned}$$

so letting $K_i = -x_z \sum w_q + N x_z w_z - b_z$

$$\begin{aligned} h_q^i &= h_z + \frac{1}{h_s}\max(0, b + x \cdot w), \\ &= h_z + \frac{x_s w_s}{h_s}\Big(b_q + \max\Big(-b_q, \sum x_q w_q - w_z \sum x_q + K_i\Big)\Big). \end{aligned}$$

The device treats the sum $\sum x_q$ as a special systolic column. We precompute Ki off-device and represent $\frac{x_s w_s}{h_s}$ as 32-bit multiply followed by a shift. Note that the simple form of the result is due to our restriction on the scales b_s. This form allows the hardware to compute the entire ”layer” operation using 8-bit multiplies with 32-bit accumulators.

3.3.3 Dequantized Results

`ConvAU` computes the operation described in the previous section for each layer in the network. The 8-bit outputs of the last layer must be downloaded from the device and converted on the host to the final floating point values. The work done by the host is minimal and can be skipped entirely for tasks where only the relative values of outputs are required, for example when performing classification with logits.

4. Mapping CNN Layers to Matrix Multiply

In order to actually perform inference on `ConvAU`, each operation in the network must be mapped to a dense matrix multiplication or activation operation. In this section we briefly explain how common network operations are mapped.

4.1. Fully Connected Layers

Fully connected layers are the easiest layers to implement as dense matrix multiplication, since they correspond almost directly. Each activation of a fully connected layer is the result of a dot product of the input values and the weights plus a bias. Thus, the entire layer can be implemented as directly as a matrix multiplication between the input and the weights, plus a bias term. In order to implement the addition of the bias term, we reduce the threshold of the activation function by the bias, and add the bias to the offset value during quantization (see section 3.3.2 for details on this transformation).

4.2. Convolutional Layers

A convolutional layer is made up of a convolution between the input and filter weights, plus a bias term. The convolution operation can be reduced to a matrix multiplication between input "patches" (where patch i is the portion of the image that would be multiplied by the filter at step i of the convolution) and the filter weights. This transformation from input to patches is commonly known as "im2col". We handle the bias term as we did in the fully connected layers by incorporating it into the activation and quantization steps (see section 3.3.2).

4.3. Batch Normalization

A batch normalization layer simply shifts and scales the input by two learned values (both constant at inference time), μ and σ. Since batch norm typically succeeds a convolution or fully connected layer, we can combine both layers together by shifting and scaling the weights and biases of the previous layer, and then treating the previous layer as normal.

Let h be the output of a layer with input x that feeds into a batchnorm layer. Then the output of the batchnorm layer is given by y as follows.

$$h = Wx + b$$
$$y = \frac{h - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

We can then transform the learned weights, W, and bias, b, to include the batch norm terms by computing W' and b' as follows.

$$W' = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} W$$
$$b' = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} (b + \mu)$$

After this transform, we can compute the output after both layers as follows.

$$y = W'x + b'$$

Note that this transformation must be done before inference (i.e. during or after training); we do not support batch normalization where the statistics change during inference.

5. Evaluation

To evaluate our design, we implemented `ConvAU` as an RTL model in SystemVerilog. The RTL model was then simulated executing inference on the 8-bit quantized Squeezenet model described in section 3.3 using Synopsys VCS M-2017.03. After executing, the simulation produced a Value Change Dump (VCD) file that recorded information about the state of the RTL model at each time step.

The dataset used during this simulation was the ILSVRC2016 CLS-LOC validation dataset [24] provided by the ImageNet project. Unless otherwise noted, we randomly sampled JPEG files from the 50000 validation set then resized them the 224×224 pixels using bilinear interpolation. No additional preprocessing or feature extraction was applied.

We then synthesized the RTL model using the Synopsys Design Compiler M-2016.12-SP2, targeting the 32/28nm Synopsys educational libraries, and performed place and route with Synopsys ICC L-2016.03-SP5-2 (final layout shown in figure 1). Our area numbers are as reported by Synopsys ICC after place and route. Our reported clock frequency is the maximum estimated clock frequency reported by Synopsys ICC after place and route.

To estimate our power consumption, we randomly sampled a representative set of matrix operations from each layer of inference during the simulation described above. We then converted the VCD file produced by our simulation into a Switching Activity Interface Format (SAIF) file that records the statistical properties of each register in the design during each of the sampled operations. These switching activities were then propagated to the final placed and routed design using using PrimeTime F-2011.06-SP3-4. Using PrimeTime, we then estimated our total power consumption by source and the power consumption of each operation.

5.1. Results

An overview of our final results compared to other existing architectures is listed in table 2. We found that `ConvAU` is able to achieve a peak throughput of 93.6 TOPs while only consuming 21W, for a ratio of 4.4 TOPs/W.

This significantly out performs the NVIDIA K80 GPU, which achieves only 2.8 TOPs at 98W for a ratio of 0.022 TOPs/W. Note that because the K80 does not natively support 8-bit integer operations, we compare it's floating point performance to `ConvAU`'s integer performance; since floating point operations are more complex (and require more energy to compute) a version of the K80 that allows integer operations may be more competitive.

The TPU achieves a similar throughput to `ConvAU` at 92.0 TOPs at a higher power consumption of 40W for a ratio of 2.3 TOPs/W. One possible reason that the TPU's

Table 2: Overview comparison between existing architectures and ConvAU

				Measured W		TOPs		TOPs/W	
Model	µm^2	nm	MHz	Idle	Busy	8b Int	Float	Total	Ratio
Haswell[1,2]	662	22	2300	41	145	2.6	1.3	0.018	244x
K80/die[2]	561	28	560	25	98	-	2.8	0.022	200x
TPU/die[2]	$\leq$331[4]	28	700	28	40	92.0	-	2.3	1.9x
ConvAU[3]	77	32	714	-	21	93.6	-	4.4	1x

[1] Haswell E5-2699 v3
[2] As reported in [16], see paper for more experimental details.
[3] Power from inference on 8-bit quantized Squeezenet.
[4] Area reported as ”less than half the Haswell die size”.

power consumption is higher is that the TPU has significantly more on chip memory than `ConvAU`; the TPU has 28 MiB of on chip memory compared to `ConvAU` which has 4 MiB. We chose to use only 4MiB since the network we were evaluating with (SqueezeNet) did not require a large activation cache.

To investigate how our performance would scale if we increased the amount of memory on chip, we analyzed the power consumption down by cell type (listed in table 3). Here we see that memory accounts for only about 16% of the total power consumption, with the majority going to registers and combinational logic (35% and 44% respectively).

If we assume that power is linear with the amount of memory, we can calculate that for a memory size similar to the TPU (28 MiB) `ConvAU` would consume a total of 42.8W, with a TOPs/W ratio of 2.2. Thus, while our design outperforms the TPU on an absolute scale, the two designs are competitive when adjusted for memory size.

Table 3: Breakdown of power consumption by cell type.

Cell Type	W	%
Memory	3.2	15.7
Clock	1.1	5.4
Register	7.3	35.3
Combinational	9.1	44.0
Total	20.7	

6. Future Work

We would like to investigate several possible improvements as future work. The first is to allow `ConvAU` to take advantage of sparsity in both the weight matrix and the dynamic activations. Recent work by S. Han [10] shows that exploiting these sparse structures can significantly reduce the total number of operations, allowing for more efficient inference.

Second, we would like to extend `ConvAU` to allow for training. Since training typically requires more precision than inference, this would require us to increase the number of bits each MAC can operate on, as well as possibly support floating point operations. In addition, training requires updating the weights each cycle, which breaks our design assumption the weights will be frequently reused. As a result, supporting training may require changing how weights are loaded into the array.

Third, in this paper we have assumed a large enough off-chip memory bandwidth to feed the array each clock cycle. However, in practice memory bandwidth can constrain the throughput of inference depending on the computation needed by the particular model. As a result, we would like to investigate how different memory bandwidth constraints effect our results, and increase both the total memory bandwidth and the effective memory bandwidth by reducing the size of the weights and activations.

7. Conclusion

In this paper we introduce the design of `ConvAU`, a CNN inference accelerator. The core of `ConvAU`’s design is a 256x256 systolic array structure that can efficiently execute the dense matrix multiplies found in CNNs. We then train a 8-bit quantized version of Squeezenet, and evaluate `ConvAU`’s throughput, latency, and power consumption during inference.

We find that `ConvAU`gives a 200x improvement in TOPs/W when compared to a NVIDIA K80 GPU and a 1.9x improvement when compared to the TPU. When power consumption is adjusted to memory size, we find that the TPU and `ConvAU`have similar TOPs/W at a slightly higher clock rate. Thus, `ConvAU`is able to accelerate typical CNN workloads with comparable efficiency and performance to existing architectures.

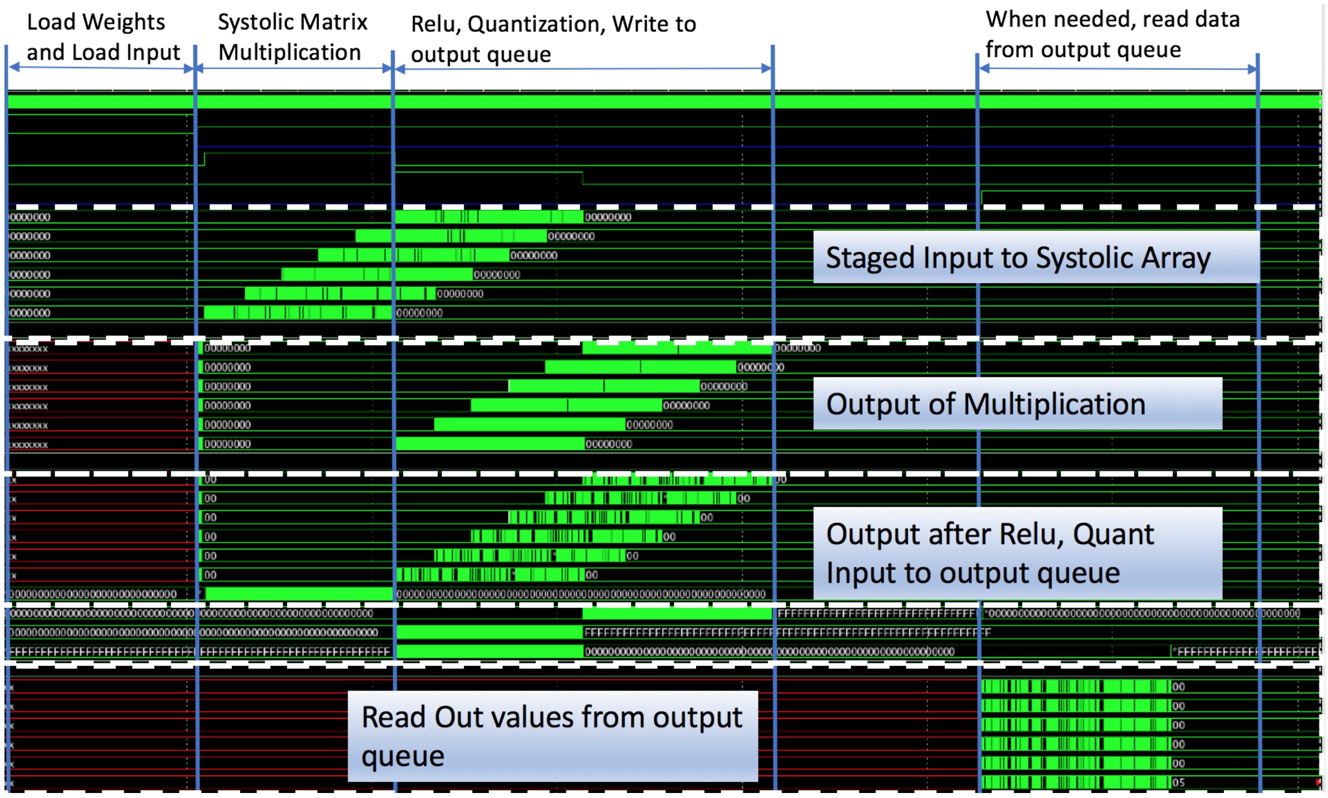
Load Weights and Load Input
Systolic Matrix Multiplication
Relu, Quantization, Write to output queue
When needed, read data from output queue
Staged Input to Systolic Array
Output of Multiplication
Output after Relu, Quant Input to output queue
Read Out values from output queue

Figure 4: Waveform of full operation of our entire design

A. Appendix - Example Operation

An example waveform of `ConvAU` performing inference on a single layer is given in figure 4. First weights and inputs are loaded into the system. Then the actual matrix multiplication is executed, taking 512 cycles. Next activations are executed and the output is fed back into the input queue. Optionally, the activations may be read out to off chip memory. The total process takes 1024 cycles, or 1.4 µs at the maximum clock frequency of 714 MHz.

Note that matrices being multiplied by the same weight matrix can be pipelined together, meaning that N successive matrix multiplications only have latency $N \cdot 512 + 512$ instead of $N \cdot 1024$ cycles.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. Yodann: An architecture for ultra-low power binary-weight cnn acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2017.

[3] 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 *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[4] F. Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[5] A. Claros. Asynchronous fifo.

[6] M. Courbariaux, Y. Bengio, and J.-P. David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[7] L. F. et al. Floating-gate transistor array for performing weighted sum computation, August 2014. US Patent Application US14459577.

[8] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *ICML*, pages 1737–1746, 2015.

[9] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. *ArXiv e-prints*, Dec. 2016.

[10] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528, 2016.

[11] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[12] S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran, et al. Dsd: Dense-sparse-dense training for deep neural networks. 2016.

[13] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks Volume 4, Issue 2*, page 251257, 1991.

[14] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[15] G. Inc.

[16] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*, 2017.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[18] L. Lai, N. Suda, and V. Chandra. Deep convolutional neural network inference with floating-point weights and fixed-point activations. *arXiv preprint arXiv:1703.03073*, 2017.

[19] E. H. Lee and S. S. Wong. 24.2 a 2.5 ghz 7.7 tops/w switched-capacitor matrix multiplier with co-designed local memory in 40nm. In *Solid-State Circuits Conference (ISSCC), 2016 IEEE International*, pages 418–419. IEEE, 2016.

[20] C.-Y. Lin. E6895 advanced big data analytics lecture 10:, 2015.

[21] D. Lin, S. Talathi, and V. Annapureddy. Fixed point quantization of deep convolutional networks, arxiv. org cs. *LG, nov*, 2015.

[22] Y. Ma, N. Suda, Y. Cao, J.-s. Seo, and S. Vrudhula. Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–8. IEEE, 2016.

[23] B. Murmann, D. Bankman, E. Chai, D. Miyashita, and L. Yang. Mixed-signal circuits for embedded machine-learning applications. In *Signals, Systems and Computers, 2015 49th Asilomar Conference on*, pages 1341–1345. IEEE, 2015.

[24] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[25] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 14–26. IEEE Press, 2016.

[26] A. S. V. Vanhoucke and M. Z. Mao. Improving the speed of neural networks on cpus. *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

[27] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

[28] C. Zhu, S. Han, H. Mao, and W. J. Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.