DNN Accelerator Using Systolic Arrays

JAE WON CHOI
(20173597)
JINNAPAT INDRAPIROMKUL
(20121097)
ANDREAS MEULEMAN
(20176429)
JEONG MIN PARK
(20150334)

1. INTRODUCTION

Over the past few years, deep neural networks have vastly improved the accuracy of image classification and regression tasks. More and more applications are using and developing their own deep neural networks. While these neural networks can improve the accuracy significantly, they come with high computational costs. Vanilla neural network (with at most one hidden single layer) can be run on a CPU with not much latency, but majority of recently developed networks, such as ResNet or GoogleNet, rely on GPUs to do the heavy lifting. To this day, GPU is the best platform to run and train networks but it is very energy inefficient. To tackle this problem, we propose a domain-specific design aimed at accelerating neural networks using systolic arrays.

We propose using systolic array architecture for our deep neural network accelerator. Systolic array architecture is an aggregate of Processing Elements (PEs) which is simply a MAC unit that processes the data and sends it to the next PE [Kung 1982]. The true power of systolic array architecture comes from its topological design which depends on what it is trying to compute. Deep neural network is essentially a collection of matrix multiplications, so we use two-dimensional systolic array that takes in batched data (batch-size $\times$ feature-size). As opposed to traditional matrix calculation using nested loops, systolic array lets data flow through only once to produce a result. There are a couple of things we wish to explore.

—The optimal order in which data should be streamed from memory to systolic arrays to maximize parallelism parallelism
—Scalability of systolic array architecture (whether it can support deeper networks without losing too much of its original improvement in latency)

2. BACKGROUND

2.1 Systolic Array

A systolic architecture aims at mapping a high-level computation onto a hardware structure, such that data batch flows through an array as computation occurs [Kung 1982]. When a single input element is involved in more than one output, a general purpose processor tend to require repetitive memory access or extra cache to store such element throughout the computation. With systolic array, processing elements (PEs) and their data path can be designed so that inputs are passed through while results are accumulated, hence eliminates redundant memory access. These PEs are essentially an ALU with store and forwarding logic. They provide parallelism at much smaller cost than GPUs or multi-core CPUs. There are various designs for a convolution computation as presented in [Kung 1982] where each differs in how input, weight, and output flows through an array. Typically, elements can systolically move through an array in one direction, stay in a PE, or get boardcasted to multiple PEs in one cycle. Depending on data size, these designs result in different array size, data path, and input/output loading mechanism.

2.2 Neural Network: Feed Forward

Throughout this paper, we will be using the following notations:

— N is the batch size
— L: the number of layers
— $d^n, n \in [\![1, L]\!]$: output size of layer n
— $x^n_{pj}, (p, j) \in ([\![1, N]\!], [\![1, d^n]\!])$: the j^{th} linear output of layer n for data p.
— $y^{n-1}_{pi}, (p, i) \in ([\![1, N]\!], [\![1, d^{n-1}]\!])$ is the i^{th} output of layer $n-1$ for data p. This is used as input for layer n.

2.2.1 *Neuron Activation.* Activation functions take outputs of the matrix multiplication shown in 5.1 and adds performs non-linear transformation to them. Adding non-linearity is the most important part of neural network - without it, the network is simply a linear function. An array of activation functions takes x^n_{pj} as input and produces the non-linear output y^n_{pj}. This output will then be sent to the next layer as inputs. While there are variety of activation functions, our architecture uses the sigmoid function:

$$y^n_{pj} = f(x^n_{pj}) = \frac{1}{1 + e^{-x^n_{pj}}} \tag{1}$$

2.3 Neural Network: Backpropagation

Backpropagation is how neural network learns from data. After a forward pass, the network will compute its loss and do a gradient descent on all the weights in the network. It does this by using a chain rule on all the weights in the network. This whole process is called backpropagation. Here are some notations for backpropagation:

— $t_{pj}, (p, j) \in ([\![1, N]\!], [\![1, d^L]\!])$ are the targeted results
— $\varepsilon = \sum_{i=1}^{d^L} \frac{1}{d^L}(t_{pj} - y^L_{pj})^2$ is the total error.
— η is a coefficient that determine how much the weights should be updated

2.3.1 *Single Layer.* The goal of backpropagation is to update the weights via gradient descent. Weights are updated using the following equation:

$$w_{ji}^L(t+1) = w_{ji}^L(t) + \eta \frac{\partial \varepsilon}{\partial w_{ji}^L}$$

We can further decompose the partial derivative as follows:

$$\frac{\partial \varepsilon}{\partial w_{ji}^L} = \frac{\partial \varepsilon}{\partial y_{pj}^L} \times \frac{\partial y_{pj}^L}{\partial x_{pj}^L} \times \frac{\partial x_{pj}^L}{\partial w_{ji}^L}$$

Since we know that:

— $\frac{\partial \varepsilon}{\partial y_{pj}^L} = \beta_{pj}^L = \frac{2}{d^L}(y_{pj}^L - t_{pj})$ that are computed right after the prediction is known

— $\frac{\partial y_{pj}^L}{\partial x_{pj}^L} = y_{pj}^L(1 - y_{pj}^L)$ with our activation function (1)

— $\frac{\partial x_{pj}^L}{\partial w_{ji}^L} = y_{pi}^{L-1}$ as $x_{pj}^L = \sum_{i=1}^{d^{L-1}} y_{pi}^{L-1} \times w_{ji}^L$

—We write $\delta_{pj}^L := \frac{\partial \varepsilon}{\partial x_{pj}^L} = g(\beta_{pj}^L, y_{pj}^L) = \beta_{pj}^L y_{pj}^L (1 - y_{pj}^L)$

We now know what to add the the weight:

$$\frac{\partial \varepsilon}{\partial w_{ji}^L} = \delta_{pj}^L y_{pi}^{L-1}$$

2.3.2 *Multiple Layers.* To extend the previous section to multiple layers, the first step is to notice that:

$$\beta_{pj}^n := \frac{\partial \varepsilon}{\partial y_{pj}^n} = \sum_{i=1}^{d^{n+1}} \frac{\partial \varepsilon}{\partial x_{pi}^{n+1}} \times \frac{\partial x_{pi}^{n+1}}{\partial y_{pj}^n} \; \forall n \in [\![2, L-1]\!]$$

And:

$$\sum_{i=1}^{d^{n+1}} \frac{\partial \varepsilon}{\partial x_{pi}^{n+1}} \times \frac{\partial x_{pi}^{n+1}}{\partial y_{pj}^n} = \sum_{i=1}^{d^{n+1}} \delta_{pi}^{n+1} w_{ij}^{n+1} \qquad (2)$$

The equation 2 corresponds to a matrix multiplication that should be done in the layer $n+1$. This could be handled in the same way as we handle matrix multiplication in the forward pass (5.1). We chose to duplicate resources in PEs to be able to start the backpropagation before the forward pass is completed.

We can now compute δ_{pj}^n and then $\frac{\partial \varepsilon}{\partial w_{ji}^L}$ the same way as we did for the last layer.

3. METHOD

We plan on using python to write a simulator for deep neural networks with fully connected layers only. Our simulator will be able to simulate deep neural network with any given depth but for our purposes, we plan on using a single layer network as our benchmark and later expand it to couple more layers to explore the relationship between the depth of the network and the rate of change in latency. We wish to design an architecture that can do both training and testing - both forward pass and backward pass (or back propagation).

Given that a typical PCIe Gen3x16 bus provides 12.5GB/s of effective bandwidth, we will use that as our memory bandwidth as well. We will also assume that all training data and labels are stored in the memory.

Metrics we will be using to evaluate different architectural designs are:

—Throughput: how much data can be processed in a given amount of time. This is especially important for training since it requires a large amount of data to the DNN and does not require a short response time.

—Latency: the time needed to process one batch of data. This metric is critical in the testing phase as users tend value response time [Patterson 2005].

—PE utilization: This metric will show how efficient our systolic array is. At a given cycle, we want to see how many PEs are computing and how many are idle.

—Power efficiency: the real advantage of a domain specific hardware is its power efficiency and we want to know the total performance/Watt. This unfortunately will be difficult for us to measure but we emphasize the importance regardless.

4. PRELIMINARY ANALYSIS/RESULTS

Our goal is to mimic the implementation of Googles TPU (tensor processing unit) which is the most widely known DNN accelerator that uses systolic array architecture. TPU uses several techniques to accelerate DNN, which is very slow on modern general purpose processors. TPU uses a technique named quantization on a two-dimensional systolic array to reduce latency. With these techniques, TPU outperformed contemporary CPU (Haswell) by 17 to 34 times better on total-performance/Watt and outperformed GPU (K80) by 14 to 16 times better on total-performance/Watt [Jouppi et al. 2017]. Our goal is to implement and find a way to maximize concurrency in our architecture and create a light-weight simulator of a TPU.

5. ARCHITECTURE DESIGN

5.1 Matrix Multiplication: Computing Linear Output of One Layer

Our first approach is to stream input data sequentially in row-major/column-major order. Given a matrix Y^{n-1} and weight matrix W, we wish to perform a matrix multiplication $X^n = Y^{n-1} \times W^n$:

$$Y^{n-1} = \begin{pmatrix} y_{11}^{n-1} & \cdots & y_{1d^{n-1}}^{n-1} \\ \vdots & \ddots & \vdots \\ y_{N1}^{n-1} & \cdots & y_{Nd^{n-1}}^{n-1} \end{pmatrix}$$

$$W^n = \begin{pmatrix} w_{11}^n & \cdots & w_{1d^n}^n \\ \vdots & \ddots & \vdots \\ w_{d^{n-1}1}^n & \cdots & w_{d^{n-1}d^n}^n \end{pmatrix}$$

Our naive implementation will stream the first row, $(y_{11}^{n-1} \ldots y_{1d^{n-1}}^{n-1})$, then $(y_{21}^{n-1} \ldots y_{2d^{n-1}}^{n-1})$ and so on until $(y_{N1}^{n-1} \ldots y_{Nd^{n-1}}^{n-1})$. Once the matrix multiplication is completed, there will be a backward pass that will compute the gradients for each weight and update the weights by adding the gradients. To complete the matrix multiplication, we will need accumulators that will collect all the partial sums from the $d^{n-1} \times d^n$ PEs and stream them back to memory.

5.2 Weight Loading

In our design, data batch flows through the systolic array, while weights remain in each PE. A strict constrain is that weights cannot

$$\begin{tikzcd}
 & 0 \arrow[d, Rightarrow] & 0 \arrow[d, Rightarrow] & \cdots & 0 \arrow[d, Rightarrow] \\
y_{p1}^{n-1} \arrow[r, Rightarrow] & \boxed{w_{11}} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \boxed{w_{12}} \arrow[d, Rightarrow] & \cdots & \boxed{w_{1d^n}} \arrow[d, Rightarrow] \\
y_{p2}^{n-1} \arrow[r, Rightarrow] & \boxed{w_{21}} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \boxed{w_{22}} \arrow[d, Rightarrow] & \cdots & \boxed{w_{2d^n}} \arrow[d, Rightarrow] \\
\vdots & \vdots \arrow[d, Rightarrow] & \vdots \arrow[d, Rightarrow] & \ddots & \vdots \arrow[d, Rightarrow] \\
y_{pd^{n-1}}^{n-1} \arrow[r, Rightarrow] & \boxed{w_{d^{n-1}1}} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \boxed{w_{d^{n-1}2}} \arrow[d, Rightarrow] & \cdots & \boxed{w_{d^{n-1}d^n}} \arrow[d, Rightarrow] \\
 & x_{p1}^n & x_{p2}^n & \cdots & x_{pd^n}^n
\end{tikzcd}$$

Fig. 1. Systolic Array for Matrix Multiplication

$$\begin{tikzcd}
 & x_{pji-1}^n \arrow[d, Rightarrow] & \\
y_{pi}^{n-1} \arrow[r, Rightarrow] & \boxed{w_{ij}^n} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & y_{pi}^{n-1} \\
 & x_{pji}^n &
\end{tikzcd}$$

$$x_{pji}^n = y_{pi}^{n-1} w_{ij}^n + x_{pji-1}^n$$

Fig. 2. PE for Matrix Multiplication

move while a multiplication is in place. It is therefore intuitive to preload weights into the array column by column before any computation. With limited bandwidth, the number of cycles taken to preload an array is given by the following equation.

$$\left\lceil d \times \frac{d}{BW/2} + 1 \right\rceil \tag{3}$$

where BW is memroy bandwidth in bytes and each data elements is of size 2 bytes, and d is a feature size.

5.3 Accumulator

Our goal is to compute

$$x_{pj}^n = \sum_{k=1}^{d^{n-1}} y_{pk}^{n-1} w_{kj} \; \forall (j,p) \in [\![1, d^n]\!] \times [\![1, N]\!]$$

To do that, we start by computing $x_{pj1}^n = y_{p1}^{n-1} w_{1j}^n$ in the row of PEs, then we send all x_{pj1}^n to the next PEs (of weight w_{2j}^n) that will compute $x_{pji}^n = y_{p2}^{n-1} w_{2j}^n + x_{pj1}^n$ and so on. y_{pi}^{n-1} is simply passed to the PE of weight w_{1j+1}^n. Figure 2 shows the architecture of a simple PE. Once a PE's value has been sent to another PE, the first PE can be used again for another data batch.

To sum up, at the end of each cycle, each PE produces a partial sum like the following:

$$x_{pji}^n = y_{pi}^{n-1} w_{ij}^n + x_{pji-1}^n$$

And pass these values to the next rows. Until we obtain:

$$x_{pj}^n = x_{pjd^n}^n = y_{pd^n}^{n-1} w_{d^n j}^n + x_{pjd^n-1}^n$$

Figure 1 shows how the data are passed inside a systolic array for matrix multiplication.

$$\begin{tikzcd}
 & x_{pji-1}^n \quad \delta_{pj}^n \arrow[d, Rightarrow, shift right=3] \arrow[d, shift left=3] & \\
\begin{array}{c} y_{pi}^{n-1} \\ (\beta_{pij-1}^{n-1}, y_{pi}^{n-1}) \end{array} \arrow[r, Rightarrow, shift left=2] \arrow[r, shift right=2] & \boxed{\begin{array}{c} w_{ij}^n \\ \Delta w_{ij}^n \end{array}} \arrow[r, Rightarrow, shift left=2] \arrow[r, shift right=2] \arrow[d, Rightarrow, shift right=3] \arrow[d, shift left=3] & \begin{array}{c} y_{pi}^{n-1} \\ (\beta_{pij}^{n-1}, y_{pi}^{n-1}) \end{array} \\
 & x_{pji}^n \quad \delta_{pj}^n &
\end{tikzcd}$$

$$x_{pji}^n = y_{pi}^{n-1} w_{ij}^n + x_{pji-1}^n$$

$$\beta_{pij}^{n-1} = \delta_{pj}^n \times w_{ij}^n + \beta_{pij-1}^{n-1}$$

$$\Delta w_{ij}^n = \eta \delta_{pj}^n y_{pi}^{n-1}$$

Fig. 3. Complete PE

5.4 Multiple Layers

To support multiple layers, we may think of two intuitive approaches. One is having a systolic array in total, which will be used for every layer, and the other is having a systolic array for each layer. We chose to have a systolic array for each layer, since we can run multiple layers in parallel by having additional resources.

Weight preloading can be done for each layer in parallel, simply by adding ports to each systolic array, so that we don't have to wait for next layer's weight preloading when previous layer's matrix multiplication is done. We also added port to forward the output generated by previous layer to the next layer as input data. As soon as previous layer's matrix multiplication for one element is done, it will be fed to the next layer's input data, and next layer starts running. This scheme naturally feeds data to the next layer in diagonal manner, as we've done for one layer, and hides the latency of forwarding the output of previous layer to the next layer.

5.5 Backpropagation

In order to support backpropagation, our PE cells need to process 4 inputs and produce 4 outputs as shown in Figure 3. Our PE is very similar to PE introduced in [Chung et al. 1991]; we only added the input y_{pi}^{n-1} to compute Δw_{ji}^n. In addition to increased inputs and outputs, the following features are added to our PE design:

—Store y_{pi}^{n-1} and y_{pj}^n in two FIFOs to pass them to PEs when they need theses values

—Add d^n PEs to this layer to compute δ_{pj}^n using g. These correspond to PEs g on Figure 4

—Enable PEs with weights to pass δ_{pj}^n vertically and y_{pi}^{n-1} along with the partial β_{pi}^{n-1} horizontally

—Add a ALU to all PEs with weights to update the weights by computing $\eta \delta_{pj}^n y_{pi}^{n-1}$

—Add a ALU to all PEs with weights to prepare $\beta_{pid^n}^{n-1} = \beta_{pi}^{n-1}$ for the previous layer backpropagation by computing $\beta_{pij}^{n-1} = \delta_{pj}^n \times w_{ji}^n + \beta_{pij-1}^{n-1}$

5.6 Reusing Systolic Arrays

Our current architecture has an unlimited amount of PEs that follows the number of layers, input size and feature size. While this gives us insight on the best-case-scenario performances of systolic arrays for deep neural networks, it is not realistic for real systems. Indeed, most DNN applications features several layers with more

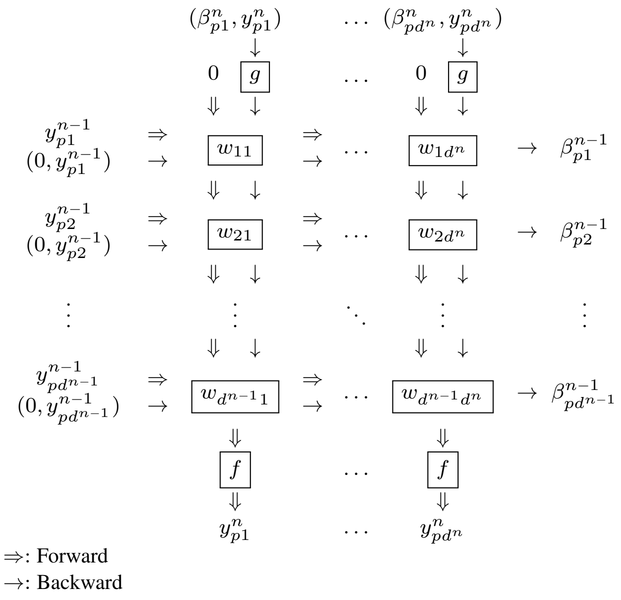

Fig. 4. Systolic Array for One Layer

than 1000 neurons each. That means millions of PEs, which would increase the hardware complexity dramatically. Besides, PEs can be inactive most of the time if the batch size is small compared the the number of neurons or the depth (Figure 6). Another concern about having many PEs is the spatial and therefore energy efficiency of our architecture.

To address this, we chose to use one fixed-size array. For our tests, we are only working on the matrix multiplication problem (Section 5.1) as backpropagation problem can be cast as a matrix multiplication (Section 2.3) and as the cycle count would scale linearly with the number of layers. Besides, as we are going to reload the weights multiple times, the backpropagation stage would not be able to take advantage of the already loaded weights as our previous architecture.

To describe our method, we will assume that:

—The systolic array has K rows and columns

— $d^{n-1} = Kd'^{n-1}$

— $d^n = Kd'^n$

We define block matrices as follows:

$$\begin{pmatrix} W^n_{11} & \cdots & W^n_{1d'^n} \\ \vdots & \ddots & \vdots \\ W^n_{d'^{n-1}1} & \cdots & W^n_{d'^{n-1}d'^n} \end{pmatrix} = W^n$$

$$W^n_{ij} = \begin{pmatrix} w^n_{1+K(i-1),1+K(j-1)} & \cdots & w^n_{1+K(i-1),Kj} \\ \vdots & \ddots & \vdots \\ w^n_{Ki,1+K(j-1)} & \cdots & w^n_{Ki,Kj} \end{pmatrix}$$

$$\begin{pmatrix} Y^{n-1}_1 & \dots & Y^{n-1}_{d'^{n-1}} \end{pmatrix} = Y^{n-1}$$

$$Y^{n-1}_i = \begin{pmatrix} y^{n-1}_{1,1+K(i-1)} & \cdots & y^{n-1}_{1,Ki} \\ \vdots & \ddots & \vdots \\ y^{n-1}_{N,1+K(i-1)} & \cdots & y^{n-1}_{N,Ki} \end{pmatrix}$$

$$\begin{pmatrix} X^n_1 & \dots & X^n_{d'^n} \end{pmatrix} = X^n$$

$$X^n_i = \begin{pmatrix} x^n_{1,1+K(i-1)} & \cdots & x^n_{1,Ki} \\ \vdots & \ddots & \vdots \\ x^n_{N,1+K(i-1)} & \cdots & x^n_{N,Ki} \end{pmatrix}$$

$$\begin{pmatrix} X^n_1 & \dots & X^n_{d'^n} \end{pmatrix} = \begin{pmatrix} Y^{n-1}_1 & \dots & Y^{n-1}_{d'^{n-1}} \end{pmatrix} \times \begin{pmatrix} W^n_{11} & \cdots & W^n_{1d'^n} \\ \vdots & \ddots & \vdots \\ W^n_{d'^{n-1},1} & \cdots & W^n_{d'^{n-1},d'^n} \end{pmatrix}$$

$$X^n_i = \sum_{i=1}^{d'^{n-1}} Y_k \times W^n_{ij}$$

6. RESULTS & DISCUSSION

To test our architecture, we ran our simulator on various settings. We first explored how batch size affects both the PE utilization and latency. We also explored both the feature size and outputs size. The resulting graph is shown in Figure 6. Cycle counts show linear relationship to batch size, feature size, and output size. For PE utilization, increasing feature size and output size decreases the overall PE utilization. However, increasing batch size increases the PE utilization linearly.

With naive implementation where we assume unlimited memory bandwidth, d cycles are taken to preload a weight matrix into the array, where d is feature size. While data takes $2 \times d$ cycles to stream through and complete multiplication. For a more realistic implementation, limited memory bandwidth BW is defined. In this case, weight preloading takes extra cycles as seen in 3.

7. FUTURE WORK

Preloading weight has a significant overhead since it needs to stream all the weights in first. The deeper the network, the longer it takes for the weights to be loaded to every PE. To alleviate this problem, both data and weights should be streamed simultaneously. By doing so, we can completely eliminate the latency from preloading weights to the network. The order in which weights are streamed must be done in a way so that PE utilization is maximized. This can be achieved by adding two registers in PEs to store two weights and separated ports for weight migration. When weights of the first layer is preloaded, computation starts. Meanwhile, second layer weights can begin to flow in the array and remain in the second register. The following layers can repeat this patters, overwriting the weights of a layer whose multiplication has been completed.

REFERENCES

Jai-Hoon Chung, Hyunsoo Yoon, and Seung Ryoul Maeng. 1991. Exploiting the inherent parallelisms of back-propagation neural net-

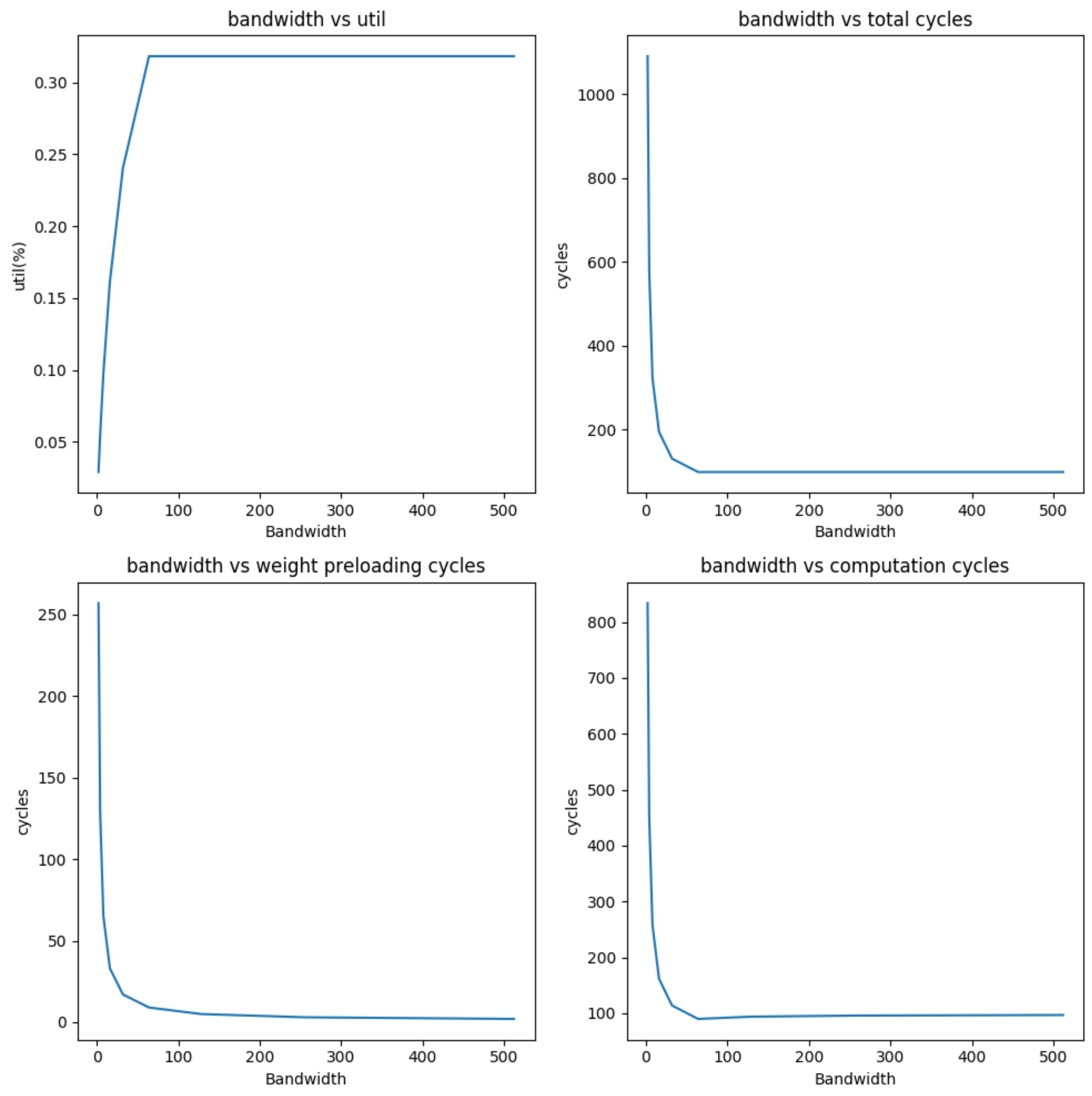

Fig. 5. Utility and cycles against bandwidth: With higher bandwidth, utility increases and cycle count decreases. At a certain point, both converge. This shows even with unlimited bandwidth, the architecture's efficiency plateaus. This suggests that the architecture is not quite scalable.

works to design a systolic array. In *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*. 2204–2209 vol.3. DOI:http://dx.doi.org/10.1109/IJCNN.1991.170715

N. P. Jouppi et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 1–12. DOI:http://dx.doi.org/10.1145/3140659.3080246

H. T. Kung. 1982. Why systolic architectures? *Computer* 15, 1 (Jan 1982), 37–46. DOI:http://dx.doi.org/10.1109/MC.1982.1653825

D. Patterson. 2005. Latency lags bandwidth. In *2005 International Conference on Computer Design*. DOI:http://dx.doi.org/10.1109/ICCD.2005.67

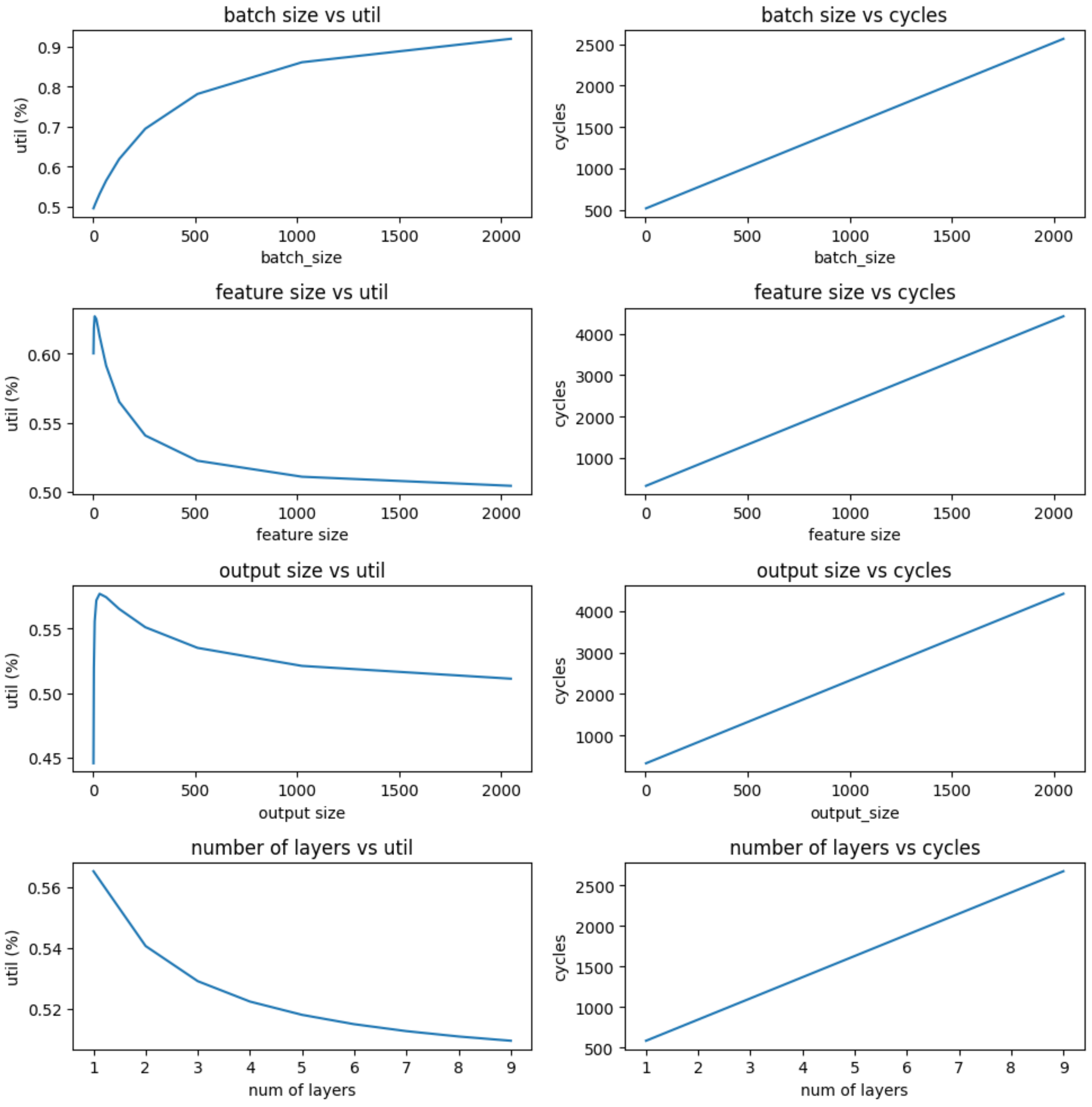

Fig. 6. Result of our architecture: the first row plots the utilization and cycle counts with different batch sizes, second row with different feature size and third row with different otput size. For all three cases, the cycle count increases linearly. For PE utilization, increasing feature and output size decreases PE utilization since our systolic arrays will grow quadratically.