Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2019-4745698

M.Sc. Thesis

SASCNN : A Systolic Array Simulator for CNN

Shashanka Marigi Rajanarayana

Abstract

Convolution Neural Networks (CNN) are used in many applications ranging from real-time object detection to robot-motion planning. CNNs are implemented on high-performance systems like multi-core CPU and GPU, these are of high power in nature and thus cannot be deployed in edge devices due to their limited battery power. The edge device has to provide real-time performance along with being low power, this prompts for an exploration of novel architectures catered towards the processing of CNNs. The recent works towards this goal have been the development of CNN accelerators using systolic array spatial architectures. The row-column stationary data-flow approach maximizes the reuse of weights, input feature maps and output feature maps across the array. Different applications require different performance, area and energy needs, and this makes it imperative to quickly prototype the architectural ideas and perform design space exploration. The challenging part is the non-trivial interactions between different architectural design parameters, as they play an important part in the complex design decisions. Hence, a hardware simulator to accelerate CNN is designed in this work. It is based on systolic array and uses row-column stationary data-flow with a near memory computing approach. The simulator supports different numerical precision such as 16-bit and 8-bit floating-point along with numerous design parameters such as the *size of the systolic array*, *latency of MAC operation*, *PE local memory size*, *PE local memory latency* and *external memory latency*. The functionality of the proposed design is verified on AlexNet. The *Destiny* memory modelling tool, along with energy and area estimation model, is used to perform a system study to investigate the trade-offs between different architectural design parameters.

SASCNN : A Systolic Array Simulator for CNN

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Shashanka Marigi Rajanarayana
born in Bangalore, India

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

DELFT UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF
MICROELECTRONICS & COMPUTER ENGINEERING

The undersigned hereby certify that they have read the undermentioned thesis, and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance of the thesis entitled **"SASCNN : A Systolic Array Simulator for CNN"** by **Shashanka Marigi Rajanarayana** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 5th November, 2019.

Chairman:

prof.dr.ir. Alle-Jan van der Veen

Advisor:

dr.ir. René van Leuken

Committee Members:

dr.ir. Arjan van Genderen

dr.ir. Sumeet Kumar

dr. Amir Zjajo

Abstract

Convolution Neural Networks (CNN) are used in many applications ranging from real-time object detection to robot-motion planning. CNNs are implemented on high-performance systems like multi-core CPU and GPU, these are of high power in nature and thus cannot be deployed in edge devices due to their limited battery power. The edge device has to provide real-time performance along with being low power, this prompts for an exploration of novel architectures catered towards the processing of CNNs. The recent works towards this goal have been the development of CNN accelerators using systolic array spatial architectures. The row-column stationary data-flow approach maximizes the reuse of weights, input feature maps and output feature maps across the array. Different applications require different performance, area and energy needs, and this makes it imperative to quickly prototype the architectural ideas and perform design space exploration. The challenging part is the non-trivial interactions between different architectural design parameters, as they play an important part in the complex design decisions. Hence, a hardware simulator to accelerate CNN is designed in this work. It is based on systolic array and uses row-column stationary data-flow with a near memory computing approach. The simulator supports different numerical precision such as 16-bit and 8-bit floating-point along with numerous design parameters such as the *size of the systolic array*, *latency of MAC operation*, *PE local memory size*, *PE local memory latency* and *external memory latency*. The functionality of the proposed design is verified on AlexNet. The *Destiny* memory modelling tool, along with energy and area estimation model, is used to perform a system study to investigate the trade-offs between different architectural design parameters.

Acknowledgments

> "You are what your deep, driving desire is. As your desire is, so is your will. As your will is, so is your deed. As your deed is, so is your karma."
>
> — Upanishad

Finally, after two years, my master study has come to a successful completion with this thesis work. I polished my skills and gained knowledge ranging from subject knowledge, presentation skills to writing skills. I made some good friends along the way. In retrospect, these two years have been a roller coaster ride with ups and downs, well, mostly ups. I thank TU Delft admission and administration committee for providing me an opportunity to study and gain knowledge at this university.

I would like to thank my advisor dr.ir. René van Leuken for his assistance during the entire course of the thesis work. You were there to help me analyse critical sections of the thesis, your feedback helped me profoundly. Thank you.

Sumeet, you were there to help me when I was stuck with some problems, and guided me in the successful completion of my thesis. You are very approachable, and easy person to talk to, you replied to my messages even when I messaged you at odd timings. Our discussions reminded me of how little knowledge I had, and how much I had to learn. You helped me analyze problems critically. You are a good mentor. Thank you.

Amir, you provided constructive and critical feedback during the presentations and for this report. Sanders, your materials on SystemC helped me with this work, and you helped me with other queries. David, you helped me by providing feedback on papers and this thesis. Antoon, you responded to emails as soon as possible, and solved any issues related to the PCs, software and network. Irma and Esther, you were there to help me with the administrative issues. Thank you.

Nikolas, Davide, Amitabh, and Shivanand, for the time we shared in these two years. Sindhura, Sherine, Bishwadeep, Nidarshan and Sushant, for the great time we had together. Bart and Ninad, for your presence in the room, without you guys the room would have felt more empty. And to all my friends in and outside India who were there to share my happiness with, and to cheer me up when I was in low spirits. The time I spent in the Netherlands would have not been as enjoyable as it was without your friendship and support. Thank you.

This goes without saying, I would like to thank my father, Rajanarayana, my mother, Jyothi, and my sister, Priyanka for their never-ending support. Even though you were quarter the circumference of the earth away, your love, encouragement, and memories sustained me during these two years without which my life would have been very difficult. I thank you from the bottom of my heart.

Shashanka Marigi Rajanarayana
Delft, The Netherlands
5th November, 2019.

Contents

List of Figures

List of Tables

1 Introduction

> "The most beautiful experience we can have is the mysterious. It is the fundamental emotion that stands at the cradle of true art and true science."
>
> — Albert Einstein

This chapter is outlined as follows: we firstly explain the relation between deep learning and compute systems. Then we further delve into the motivations and goals of this thesis, subsequently cover the approach taken for this work and the contribution of this thesis, and end with outline of this report.

Deep learning belongs to a class of machine learning algorithms which uses multiple layers to progressively extract meaningful higher-level features from the given data. Modern deep learning models use artificial neural networks, specifically Convolution Neural Networks (CNN) to perform machine learning tasks. Deep Neural Networks (DCNNs) and CNNs are extensively used in the discipline of computer vision like object detection, scene segmentation and action recognition in videos [1]. The accuracy of DNNs has increased over the years at a fast pace; the winner of the ImageNet (ILSVRC) challenge in 2012 was AlexNet architecture which had an accuracy of 84.7% [2] and the winner of 2015 was ResNet-152 which had an accuracy of 96.5% [3]. This has opened up new avenues for deployment of DNNs in drones [4], smartphone [5], autonomous vehicles [6] and many other edge devices.

The higher accuracy models have higher computational complexity due to the presence of a higher number of layers in the network. Let us consider AlexNet [2], which requires 0.8 GOPS to process a single image, while VGG-16 [7] takes 15 GOPS, which is in orders of magnitude higher when compared to AlexNet. Assuming that an autonomous vehicle has 5 cameras each runs at 30 frames per second (fps) and has an image resolution of 1080 pixels; running the AlexNet would require *4.1 TOP/sec* and running the VGGNet-16 would require *73.4 TOP/sec* compute system.

These DNNs are implemented on the cloud for some applications like autonomous vehicles, as it has high performance compute systems with a high power budget. In some off-line applications like autonomous drone navigation, the trained model is loaded on to the drone compute system and the inference is done using this edge compute system. The high-performance system in the cloud has different performance, memory and energy requirements when compared to edge devices which are of low power budget and small form factor. Even in the edge devices, the performance requirements vary from one application to another. The edge compute system should provide near real-time performance with a limited budget and this prompts for a need for hardware accelerators for accelerating inference in DNNs.

1.1 Motivation

There exists a difference in performance, energy and area requirements for high performance compute systems and edge compute systems, and this calls for a total revamp in how hardware accelerator architectures are perceived and designed, specifically for DNN applications as they are memory and computation intensive. This provides the designers with various challenges as explained below:

1. The high performance compute systems in the cloud are based on temporal architecture and they cannot be used in the edge devices due to energy and area budget.

2. The CNNs have structural and functional similarity, and in-order to exploit this to reduce the number of memory accesses, new data-reuse and data-flow methodology has to be used.

3. Different applications need different numerical precision as all applications do not require high precision. The numerical precision has an impact on the energy consumption and area of the accelerator chip.

4. To cater to the different application needs and to cut the total design time, there is a need to shorten the time spent on design space exploration of the accelerator architecture and obtain a first-order performance metrics for further analysis.

1.2 Thesis Goals

This thesis studies the acceleration of CNN on hardware accelerators. The CNNs have parallelism built-in and these can be exploited to build efficient hardware designs. The convolution layer is broken down into many parallel and independent sets and these are executed in parallel on a spatial compute architecture. The systolic array is an ideal candidate for accelerating the CNN layers as it reduces the number of off-chip memory accesses. A near-memory computing approach should be used to further reduce the number of off-chip memory accesses. The performance of the design can be increased by exploring alternate numerical precision for CNNs without compromising the accuracy. The primary objective of this thesis is to build a systolic array simulator for accelerating CNN layers. The simulator should include near-memory computing methodology and reuse different data types as much as possible to eliminate off-chip memory access. The simulator should be able to take in numerous design parameters and produce performance metrics like throughput, memory bandwidth, execution time, power consumption and area estimation among others. The aim of the system study is to investigate the impact of the systolic array size, memory size and numerical precision on these performance metrics of the proposed architecture. These performance metrics will help the designers to scope out the design space for optimal architecture to accelerate a particular application.

1.3 Approach

This thesis begins with a brief introduction of neural networks and deep learning. Next it goes on to explore the individual layers of CNN models to identify the possible parallelisms. Subsequently, the impact of CNN layers on memory, energy and computation requirements is explored by profiling a standard CNN model. An introduction of systolic array spatial architecture is presented, and subsequently, the state of the art hardware accelerators are presented, its drawbacks are analyzed and possible optimizations are explored. Different data-flow approaches are compared, and alternate numerical precision and near-memory processing are explored.

A SystemC based simulator is built which incorporates systolic array near-memory processing architectures with *row-column stationary* data-flow approach. The design is functionally verified against a golden reference data generated using Matlab. The AlexNet layers are simulated with different design parameters and the results are analyzed for design space exploration.

1.4 Thesis Contributions

The main contributions of this thesis are as follows:

- Explore and propose a systolic array spatial architecture for accelerating CNN layers.
- Use *Row-Column Stationary* data-flow methodology for maximum reuse of filter weights, and to exploit the inherent parallelism present in 3-D convolution.
- Use Near-Memory computing approach to solve the memory requirement issues in accelerators.
- Explore and utilize the different numerical precision for improving the performance of the proposed design.
- A SystemC implementation of a simulator, which takes in numerous design parameters and provides performance metrics for design space exploration of accelerators.
- A systematic study of the proposed architecture on convolution layers of the standard AlexNet model.

1.5 Outline

The further chapters of the thesis is organized as follows:

- **Chapter 2** provides introduction about DNNs, its operation and architecture, and profiles a standard CNN.
- **Chapter 3** provides state of the art compute architectures used in hardware acceleration of DNNs. Subsequently, alternate approaches to the existing method are proposed.

- **Chapter 4** describes the implemented systolic array simulator. The hardware architecture, characteristics, functionality and input parameters are explained.
- **Chapter 5** details the results and analysis of the design using the AlexNet model.
- **Chapter 6** concludes the work, and presents the future research directions.

This research was supported in part by the European Union and the Dutch government, as part of the ECSEL JU program under PRYSTINE project.

Figure 1.1: Programmable Systems for Intelligence in Automobiles.

2 Deep Neural Network

> "Nothing in life is to be feared, it is only to be understood. Now is the time to understand more, so that we may fear less."
>
> — Marie Curie

This chapter is outlined as follows: we first introduce deep learning, Convolution Neural Network and its operation. Then we further delve into Neural Network (NN) architecture, profile standard CNN architecture to get an overall understanding of its operational complexity from an algorithm perspective.

Deep learning is a subset of machine learning which uses Deep Neural Network (DNN) to solve machine learning tasks as shown in the Figure 2.1. Even though deep learning was proposed in 1960, it became a success from the year 2010 due to three main factors; first, the availability of large data, thanks to the Internet, to train the networks. Second, the compute capacity of the systems increased due to new multi-core architectures and lower technology nodes. Third, the success of DNN algorithms, which were brain-inspired, span a large number of different algorithms for training efficiently. The Figure 2.2 shows the performance of DNN in ImageNet challenge over the years [8].

DNN History

- 1960s - First proposal of DNN.
- 1989 - Digit recognition using Neural Nets (LeNet [9]).
- 2011 - Microsoft proposed the first speech recognition using DNN.
- 2012 - AlexNet used for classification in vision data.
- 2014+ - Research aiming at accelerating the DNN (DianNao [10], Neuflow [11]).

Deep learning uses two ways to train the system: *supervised learning* and *unsupervised learning*. A learning methodology is called *supervised learning* when all its training data is labeled, and it is called *unsupervised learning* when the training data is not labeled. After the training, the model is deployed for testing, which is also called *inference*. A Neural Network can be made of one or more layers of neurons and in case of multiple layers, there exist connections between them. A single neuron in each layer accepts inputs from preceding neurons and generates a single output. The neuron output is a dot product of inputs and weights, this output is fed to the neuron's activation function, which can be linear or non-linear. Multiple neurons form a layer and several layers form a network. A layer with no

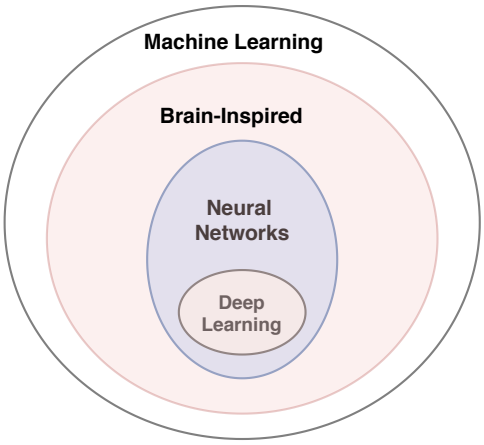


Figure 2.1: Deep learning in the frame of reference of Machine learning.

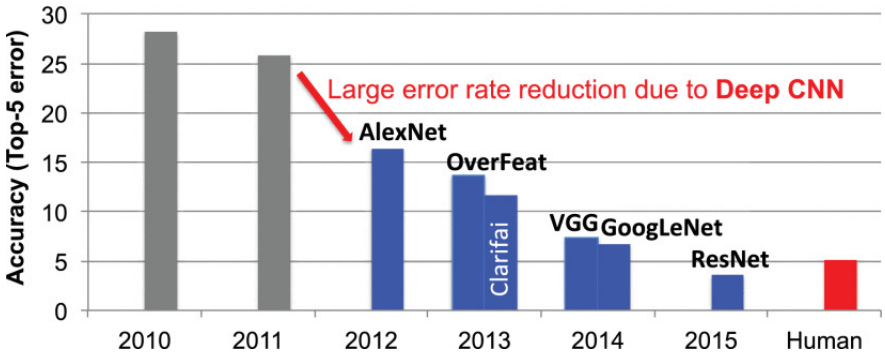


Figure 2.2: ImageNet challenge results.

preceding layer is termed an input layer and a layer with no successor is termed an output layer. The layers in-between the input layer and output layer are called hidden layers as shown in the Figure 2.5. A network is termed as *Deep* if there are large number of hidden layers, contemporary deep networks have more than hundreds of such hidden layers [3].

The neurons are connected through *inter-connects* [12]. The data is transferred through these *inter-connects* from the i^{th} neuron of one layer to the j^{th} neuron of the succeeding layer. Each *inter-connect* in each layer has a weight θ_{ij} associated with it. This weight is multiplied with the input value or activation which may decrease or increase the overall value. These weights are updated during the *training* phase, in which, different optimization techniques can be used, e.g., gradient descent. Gradient descent is a first-order optimization technique. The gradient of the function is calculated over the variable of interest and then the variable value is moved in the negative direction of the gradient. The size of the move (step) is proportional to the magnitude of the gradient descent and the changes in the weight are made corresponding to the learning rate. The *back-propagation* algorithm is used to calculate the gradient. In the *back-propagation* phase the data is fed to the network and the output values are collected. This forward movement of data is called *feed-forward*, also called *inference*. The result of *feed-forward* pass can be right or wrong, which is measured quantitatively using a loss function. The gradient of this loss function is calculated over the inputs of each layer

and each weight. The calculation of gradient is done iteratively starting from the output layer and ending at the input layer. Then the weights are updated according to the gradient descent rule:

$$\theta_{i,j}^{t+1} = \theta_{i,j}^{t} - \alpha \frac{\partial L}{\partial \theta_{i,j}} \tag{2.1}$$

Where L is the loss function, t is the time instance and α is the step size. One training iteration is a combination of *feed-forward/inference*, *back-propagation* and weight update. Usually, the number of iterations is in the magnitude of 100,000 to obtain sufficient accuracy on the test set.

There is a steady increase in accuracy of DNNs used in ImageNet challenge as shown in the Figure 2.2. The first large reduction in error occurred with AlextNet (84.7% accuracy) and the recent ResNet has an accuracy of 97%. This has resulted in many applications of DNNs ranging from medical to multimedia domain. Few of the applications are explained as follows:

- *Video processing*
  Video contributes the most to the big data as its share is more than 70% of today's traffic on the Internet. To extract intelligent and meaningful information from these videos computer vision is necessary. As the accuracy of DNNs have increased, they are extensively used in image classification [2], object detection and localization [13], and scene segmentation [14].

- *Speech processing*
  The DNNs usage in speech recognition has increased as it gives better results [15]. It is also used in audio generation and natural language processing [16].

- *Medical data processing*
  In the field of genetics, for detection of diseases such as autism and cancer, the DNNs have been proved useful [17]. They are also used to detect brain cancer and skin cancer using medical images [18, 19].

- *Robotics*
  In robotics, the DNNs are used to grasp things using robotic arms [20]. They are also used in visual navigation [21] and motion planning for robots [22], control of quadcopters and in autonomous navigation [23, 24].

The two stages of the DNN (i.e., training and inference) needs different computational capacity. The training of DNN needs large data-set and high performance computational system. The training may take anywhere between several hours to multiple days, thus training is done on the cloud since it has enough resources. Edge computing devices such as the Internet of Things (IoT) and robots can perform inference. Applications like mobile camera, autonomous quad-copter, autonomous vehicle, and many others require inference to be done near the sensor or require the compute engine to be located near the sensor for fast inference. It is desirable to extract the useful data from the video right at the sensor unit, as it reduces communication cost, latency and provides real-time inference capability.

2.1 Neural Network Architecture

A Neural Network(NN) has a number of layers. Each layer may perform a different operation or same operation with different size of inputs. The Neural Network architecture and the functionality of each layer is explained in this section.

2.1.1 Single Neuron

This neuron performs a dot product between two 1-D tensors and adds a bias to the result. This result is fed to a non-linear activation function. One of the 1-D tensors are the weights which is analogous to the synapses and the output and input tensors are equivalent to the axons of the neuron in the brain. The Equation 2.2 and Figure 2.3 shows the functionality and visualization of a single neuron, respectively. The *w* and *x* are the weight vector and input vector respectively. The function *f* performs a dot product between the weights and inputs, and adds a bias to the resultant to obtain the output *y* of the neuron.

$$\mathbf{y_i} = \mathbf{f}(\sum_{k=1}^{n} \mathbf{w_i}.\mathbf{x_i} + b) \tag{2.2}$$

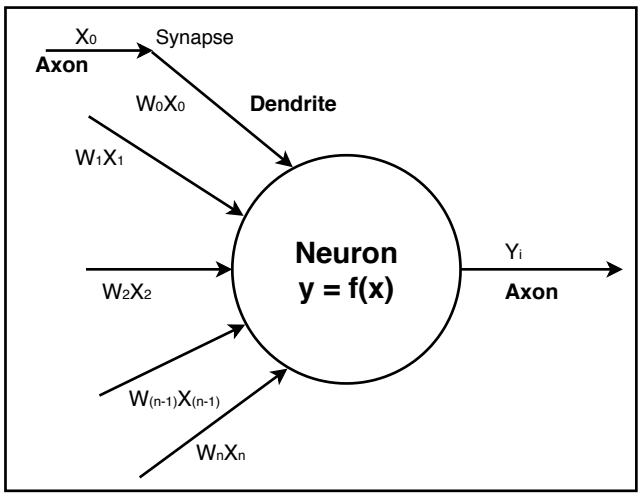


Figure 2.3: A single Neuron.

2.1.2 Single-Layer Perceptron (SLP)

It is the simplest form of NN and it does not contain any hidden layer. The neuron includes a weight vector in its memory, the weights from the weight vector are multiplied with the corresponding inputs from the input vector and a sum is obtained over the products. This sum is fed to an activation function to calculate the final output. It is called a single layer because it contains only the output layer as shown in Figure 2.4.

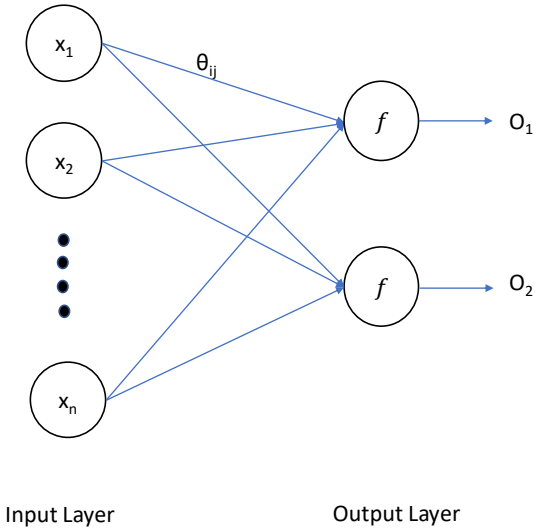


Figure 2.4: Single layer Perceptron (SLP).

2.1.3 Multi-Layer Perceptron (MLP)

This network is an extension of single layer perceptron network. This consists of multiple hidden layers, which are connected in a feed-forward manner. In MLP, each neuron of $layer_i$ is connected to $layer_{i+1}$ as shown in the Figure 2.5.

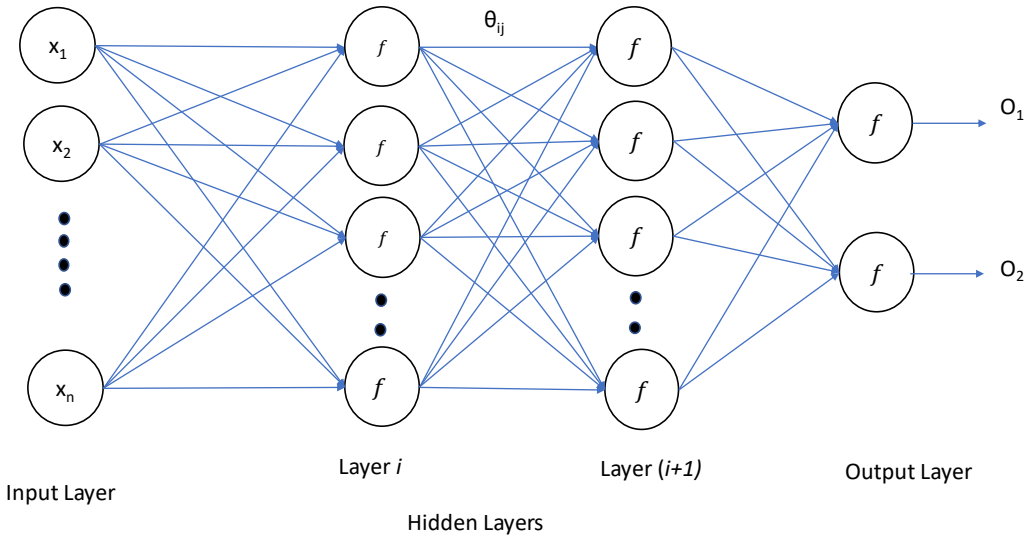


Figure 2.5: Multi-layer Perceptron (MLP).

2.1.4 Convolution Neural Network

The Convolution Neural Networks (CNNs) are a special type of DNN which are used in object detection and scene segmentation using images. CNNs are variations of MLP and the network has a specific mathematical operation called convolution. One of the key differences between MLP and convolution layers is weight re-use. These neural networks use convolution in at least one of their layers. CNN consists of a number of hidden layers along with the output and input layer. Each hidden layer may be a convolution layer, fully connected layer, pooling layer or activation function layer as shown in the Figure 2.6. Inference or feed-forward propagation of one input image (Tensor) is considered in this section for computational analysis of different layers in the CNN [25]. The layer parameters can be modified and used as tensor data as shown in the Table 2.1.

I	Input feature maps	1	CH	H	W
O	Output feature maps	1	N	P	Q
W	Learned weights (Filters)	N	CH	X	Y
B	Learned biases	N	-	-	-

CH/H/W	Depth / Height / Width of Input Feature maps
N/P/Q	Depth/ Height/Width of Output Feature maps
X/Y	Height/ Width of the Filters(Weights)

Table 2.1: Data format with dimension for a given layer *l* of a CNN.

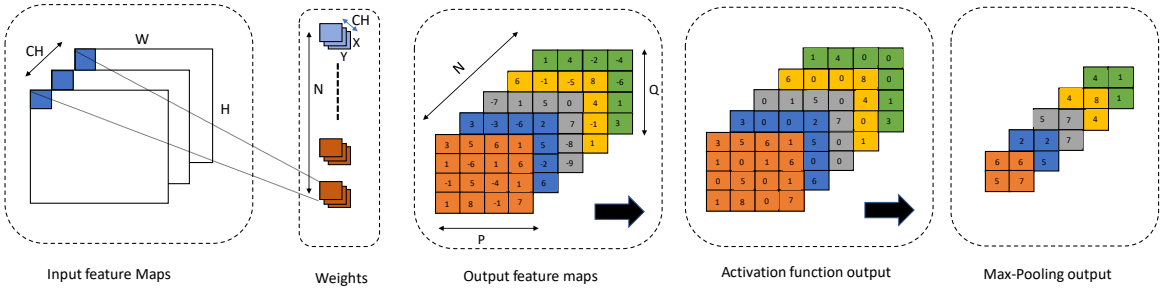


Figure 2.6: Layers of CNN.

2.1.4.1 Convolution Layer

A convolution layer (*conv*) layer performs spatial 3-Dimensional (3-D) convolution operation between input feature maps (ifmap) $\mathbf{I}^{\text{conv}}$ and 3-Dimensional convolution filters $\mathbf{W}^{\text{conv}}$, and adds the biases $\mathbf{B}^{\text{conv}}$ to them. Each layer takes in the output of the preceding layer (direct input if it is the fist layer). The input volume, of depth *CH* and and a 3-D filter is fed to the *conv* layer to obtain a 2-Dimensional (2-D) feature map (FM) and each layer produces a set of *N* output feature maps corresponding to *N* number of 3-D filters. A bias *B* is added to the 3-D convolved result. The equation for the convolution layer is described in the Equation 2.3.

$$\forall\{n, p, q\} \in [1, N] \times [1, P] \times [1, Q]$$

$$\mathbf{O}^{\text{conv}}[n,p,q] = \mathbf{B}^{\text{conv}}[n] + \sum_{ch=1}^{CH}\sum_{j=1}^{J}\sum_{k=1}^{K}\mathbf{I}^{\text{conv}}[ch,p+j,q+k].\mathbf{W}^{\text{conv}}[n,ch,j,k] \tag{2.3}$$

2.1.4.2 Activation Layer

The convolution layer is followed by an activation layer that applies non-linear function to the input data as shown in the Equation 2.4. The activation function can be Rectified linear Unit (RELU), TanH or Sigmoid.

$$\forall\{n,p,q\} \in [1,N] \times [1,P] \times [1,Q]$$

$$\mathbf{O}^{\text{act}}[n,h,w] = act(\mathbf{I}^{\text{act}}[n,h,w]) \tag{2.4}$$

2.1.4.3 Pooling Layer

The pooling layers is inserted between successive convolution layers to reduce the dimension of the feature space. Two prominent pooling functions are average pooling and Max-pooling (maximum of a given neighborhood K) as shown in the Equation 2.5.

$$\forall\{n,p,q\} \in [1,N] \times [1,P] \times [1,Q]$$

$$\mathbf{O}^{\text{pool}}[n,p,q] = max(\mathbf{I}^{\text{pool}}[n,p+r,q+s];(r,s \in [1:K])) \tag{2.5}$$

2.1.4.4 Fully connected Layer

In Fully Connected (FC) layer, a neuron in layer L_{i+1} takes in the outputs of all the neurons of layer L_i. In principle, this layer is the same as MLP as shown in the Figure 2.5. The FC layer operations can be broken down to matrix multiplication as shown in the Equation 2.6.

$$\forall\{n\} \in [1,N]$$

$$\mathbf{O}^{\text{fc}}[n] = \mathbf{B}^{\text{fc}}[n] + \sum_{ch=1}^{CH}\sum_{h=1}^{H}\sum_{w=1}^{W}\mathbf{I}^{\text{fc}}[ch,h,w].\mathbf{W}^{\text{fc}}[n,ch,h,w] \tag{2.6}$$

2.1.5 AlexNet Architecture

AlexNet was designed by Alex Krizhevsky [2]. It contains 8 layers, of which 5 are convolution layers and last 3 layers are fully connected as shown in the Figure 2.7. The convolution layers contain the pooling and activation functions embedded in it. The network uses ReLU as a non-linear activation function, max-pooling as a pooling function and soft-max function for classification. The soft-max function is a probability distribution function, which takes in the outputs of fully connected layer as inputs. The details of the filters and shown in the Table 2.2.

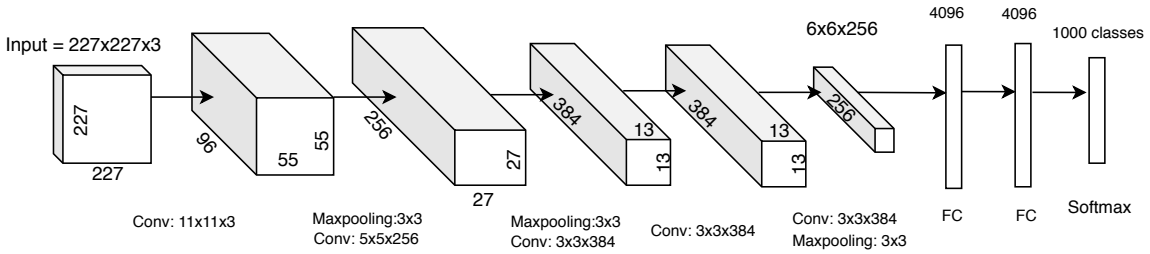


Figure 2.7: AlexNet architecture.

2.2 CNN Profiling

Two standard CNN models, AlexNet and VGGNet are profiled against ImageNet test-set. As seen from the Table 2.2, the accuracy of the model increases as the number of convolution layer increases [26]. For further analysis, only *conv* and *fc* layers are considered as they take up large execution time and memory accesses.

Metrics	**AlexNet** [2]	**VGG-16** [7]
Top-5 accuracy	80.3%	88.7%
Input	227x227	224x224
CONV Layers	5	16
Filter Sizes	3,5,11	3
Channels	3-256	3-512
Filters	96-384	64-512
Stride	1,4	1
Weights	2.3M	14.7M
MACs	666M	15.3G
FC layers	3	3
Weights	58.6M	124M
MACs	58.6M	124M
Total Weights	61M	138M
Total MACs	724M	15.5G

Table 2.2: Metrics and Network.

The computational cost during inference phase is the result of intensive arithmetic operation such as Multiply and Accumulate (MAC) operation. The computational complexity increases as the number of layers and number of filters increases. As the *conv* layers perform transformed matrix multiplication, they can be represented in terms of MAC operations. As seen from the Table 2.2, the *conv* layer consumes a very high number of MAC operations. Typically, 90% of the execution time is spent in *conv* operations during inference [27].

In this chapter, the operation of DNNs and its application was explained. Subsequently, the individual layers of CNN were explored from a mathematical perspective and we profiled the AlexNet and VGG-16 to get an understanding of the computational and memory complexity from an algorithmic perspective. The Chapter 3 looks at the state of the hardware architectures for CNN acceleration.

Compute Systems

3

"If I have seen further it is by standing on the shoulders of Giants."

— Isaac Newton

This chapter is outlined as follows: we first analyze the memory, energy and computational implications of CNN algorithms form a hardware perspective. Subsequently, state of art hardware accelerators and Near-Memory Computing is explored.

The focus of this work is low power edge computing devices which can run the DNN models. So, the scope of the design space exploration is limited to Application-Specific Integrated Circuit (ASIC) for *inference*, rather than high power consumption platforms like Central Processing Unit (CPU), Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA).

In Chapter 2, we analyzed the standard CNNs like AlexNet and VGGNet. Considering the AlexNet for further analysis, we see that the *conv* layers are computationally intensive as it takes 666 x 10^6 Multiply and Accumulate (MAC) operations and *fc* layers take 58.6x10^6 MAC operations. The size of weights in the *conv* and *fc* layers is around 10 MB and 240 MB respectively. In the *fc* layer, the weights are more when compared to *conv* layer weights. This results in an imbalanced total computation to total memory ratio, hence the CNN accelerators use different implementation strategies for *conv* and *fc* sections of the network during inference.

The number of memory accesses and the total energy consumption is related. Small sections of CNN models can be loaded fully onto the on-chip memory, but a complete DNN can not be loaded. There is a requirement of Dynamic Random Access Memory (DRAM) access, but accessing it is expensive in terms of energy and time. We can see from the Table 2.2, as the number of layers increases, so does the memory requirement and total operations count. An increase in cache size or DRAM is not beneficial in-terms of number of cycles and power consumption. On analyzing the Figure 3.1 in the context of a system, we see that accessing an element from DRAM is 200 times more expensive when compared to accessing the same from a Register File (RF) [28].

The memory bandwidth is the bottleneck in the *fc* layers whereas its computation and memory are the bottlenecks in *conv* layer [29]. This is due to the presence of high number of weights in the *fc* layers and higher number of computations in the *conv* layers. A higher number of weights leads to a high number of memory reads to the DRAM. In *conv* layer, each MAC operation incurs at-least 2 memory reads and 1 memory write. All these are directed towards external memory access (DRAM). The throughput, latency, and energy consumption are highly impacted due to the access of external memory (DRAM). The Figure 3.2 shows

the distribution of energy consumption between different layers of AlexNet [30] for inference of one image. The energy consumption is estimated using an online tool [30] based on consumption and data movement of the three data types, weights, output feature maps, and input feature maps for every layer. The unit of energy is measured in terms of energy required for one MAC operation. The percentage of energy consumption for movement of data outweighs the energy required for computation. The initial layers have a significant number of computations as seen in the Figure 3.2 [30]. This high energy consumption is due to external memory access for data. The DRAM accesses can be reduced by implementing different levels of memory hierarchy, as shown in the Figure 3.1, this reduces the latency and energy consumption. Hardware accelerators like Eyeriss [28] and ShiDianNao [5] employ two levels of memory. The first level of memory is the small on-chip local memory (usually Register Files) and the second level is a global buffer (SRAM) which is of larger in size. This implementation provides lower latency and lower energy consumption per memory access which is in the orders of magnitude lower than the external memory access [28].

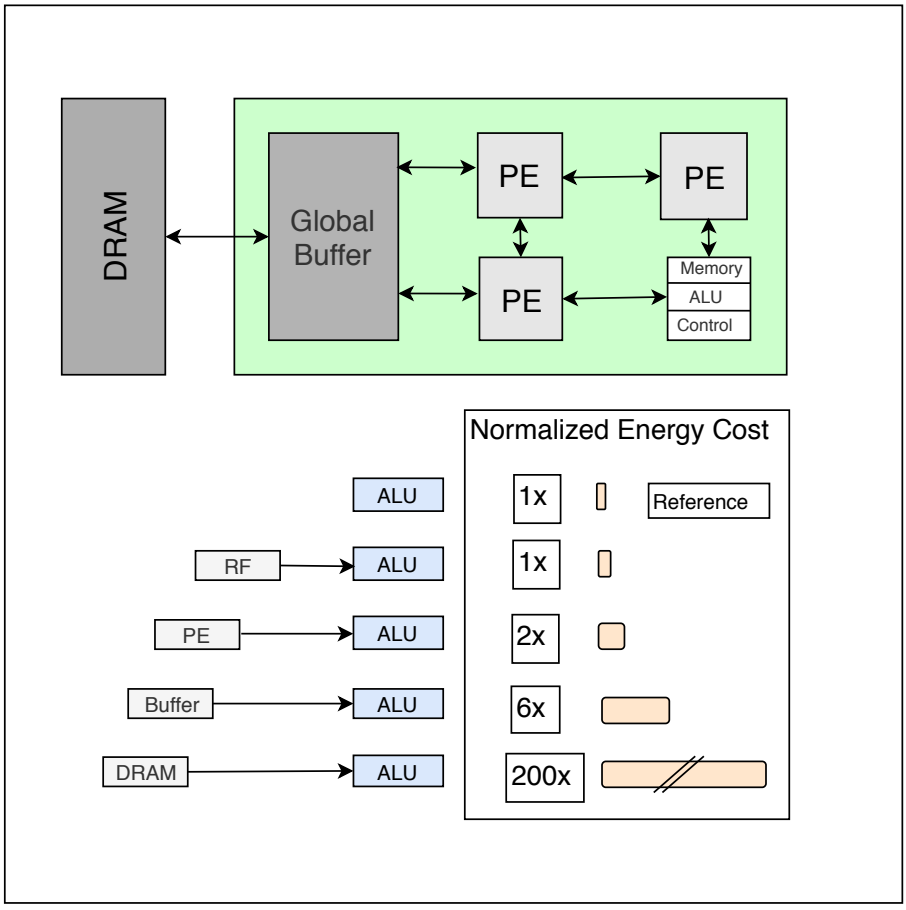


Figure 3.1: Memory and energy relation.

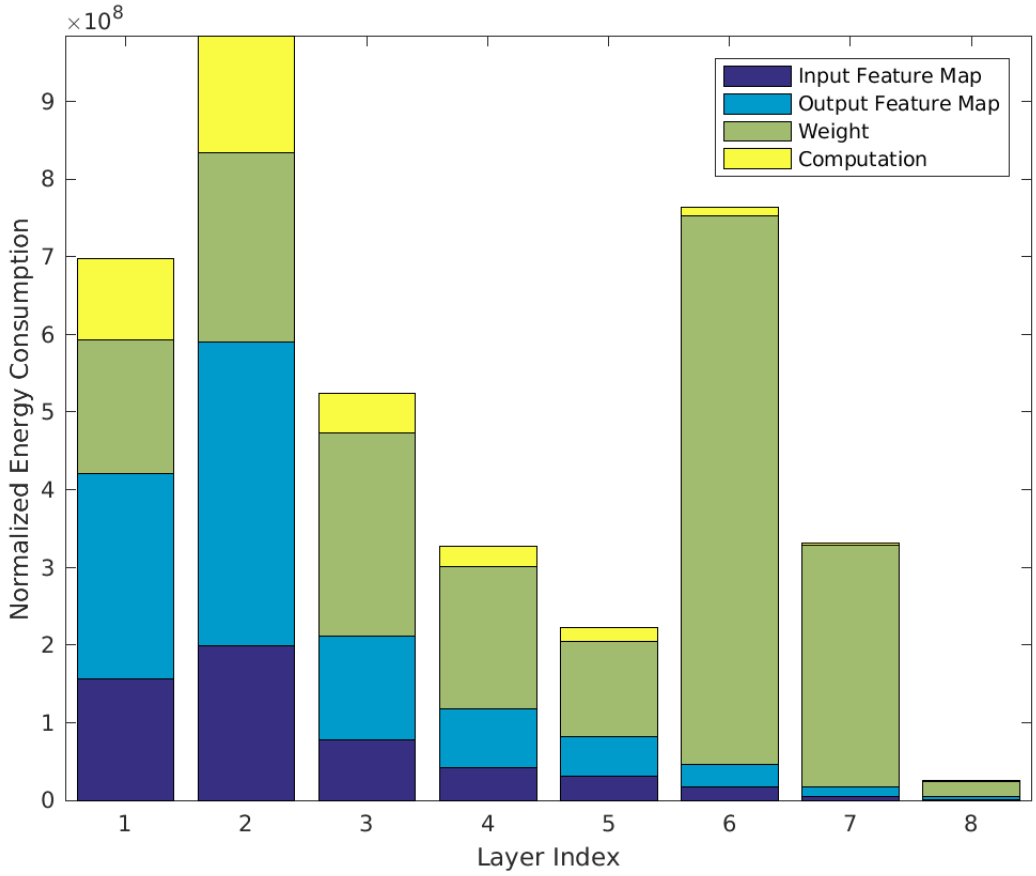


Figure 3.2: Energy visualization.

3.1 Hardware Architecture

Traditionally the inference is implemented on *Temporal compute systems* which use Single Instruction Multiple Thread (SIMT)/Single Instruction Multiple Data (SIMD) architectures. These are typically used in the CPU and GPU designs, but they do not provide real-time inference with low power. The *Temporal architecture* based compute systems typically contain a large number of Arithmetic Logic Units (ALUs) which fetch data from the memory. There is a memory unit and a control unit as shown in the Figure 3.3 [8], there is no communication between the ALUs. Usage of *Temporal architectures* results in repeated accesses to main memory for data even with enough parallelism in the algorithm. In the recent years *Data-flow/Spatial Architectures* are used to accelerate compute and memory-intensive tasks, here the Processing Engines (PEs) form a chain of processes and they communicate by passing data between themselves as shown in the Figure 3.3. This eliminates the need for external memory accesses. Processing Engines are similar to ALUs, they contain a MAC unit, local control unit and significantly large internal memory. *Spatial architectures* are typically employed in ASIC-based and FPGA-based designs.

On analyzing the DNNs, we see that there is a regularity in access pattern of the data and this can be leveraged in data flow machines to increase the data reuse from the local

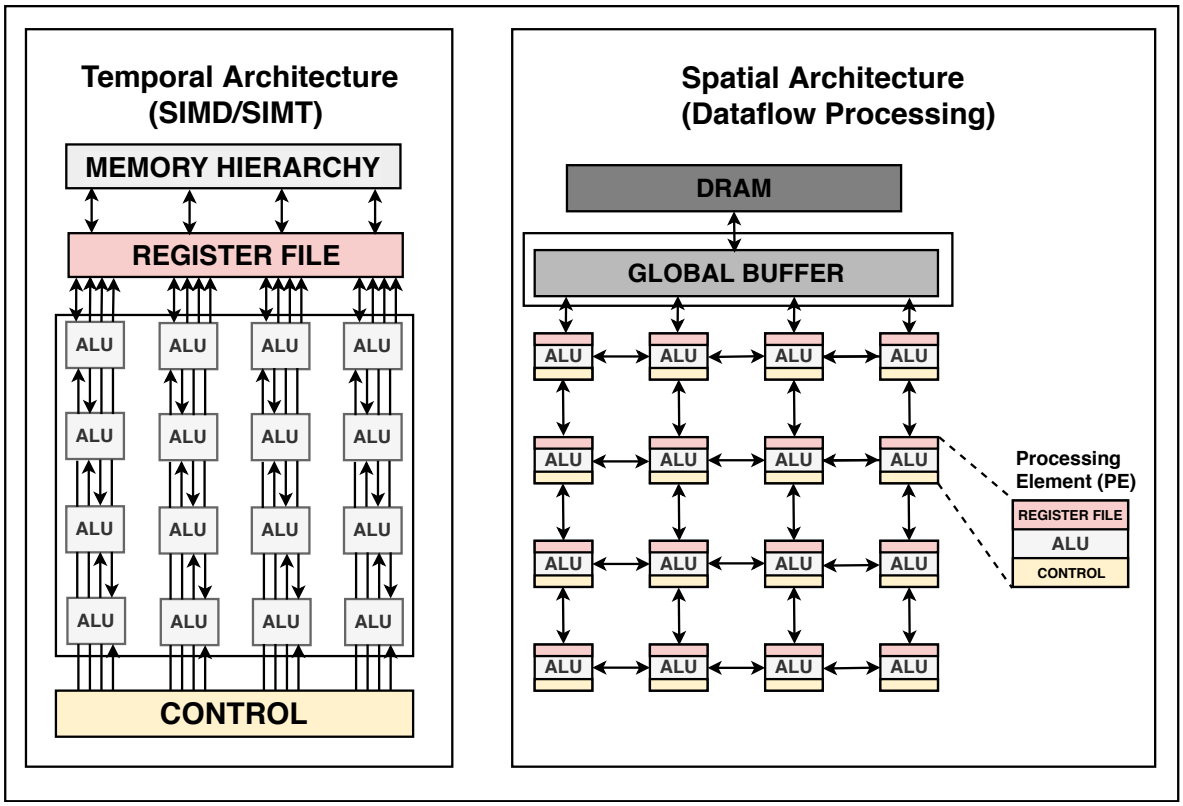


Figure 3.3: Compute architectures.

memory of PE to reduce the energy consumption. The memory hierarchy as shown in the Figure 3.1 can be employed, as we move closer to the compute logic, the energy per access reduces. The DRAM access is the most expensive one as we need to access the data from external memory, while data access from the RF is the lowest. Multiple levels of memory help in improving energy efficiency by providing lost cost access to lower memories like RF and shared memory (global buffer).

3.1.1 CNN Regularities

The CNNs have a functional and structural regularity which makes it an ideal candidate for implementation on the data-flow machines with local memory which can adapt to various shapes and sizes, thus providing energy efficiency. The data access from the DRAM is more expensive than accessing the same from the local memory, but the local memory is of limited capacity and expensive. For this reason, there is a need to reuse the data as much as possible and this can be done by optimizing data-flow through the exploitation of parallelisms which are inherently present in the CNNs. In the CNNs, the input feature maps, the activations, and filters can be reused. The reuse mechanism categories are shown in the Figure 3.4 [8]:

1. *Convolution Reuse*
   The filter weights and corresponding activations from the result of convolution with

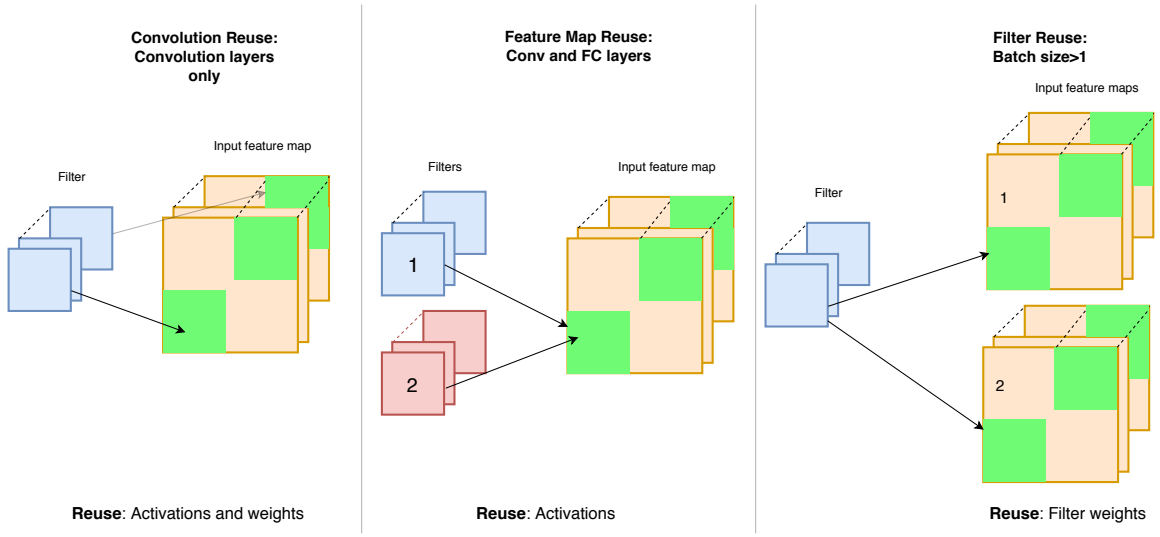


Figure 3.4: Data reuses in CNNs.

input feature map are reused. The data from one convolution output is reused across different channels and the results are accumulated to obtain the final sum. Here the activations and filter weights are ideal data to be stored in the local memory.

2. *Feature-map Reuse*
   The input feature maps are reused to the maximum by applying multiple filters over the same input feature map multiple times to obtain the desired result. The input feature maps are stored in the local memory.

3. *Filter Reuse*
   The filter weights are processed across multiple different input feature maps in parallel (also known as batch). The local memory stores the filter weights as they are reused.

3.1.2 Spatial Architectures

In *data-flow architectures*, the execution of instructions is ready as soon as the input operands for its operation is made available. Here, the data or operands for the new instructions are made available by directing the results of previously executed instructions. This type of processing forms a data-flow, indicating instructions that are to be executed. This results in a simultaneous execution of multiple instructions, leading to a higher throughput via concurrent computation. A typical data-flow computer consists of five units as shown in the Figure 3.5:

1. The specialised processing unit.

2. The memory unit, which holds the instructions and operands.

3. The arbitration unit, which delivers the data to the processing units.

4. The distribution unit, which transfers the results to the memory from the processing units.

5. The control unit, which oversees all other units.

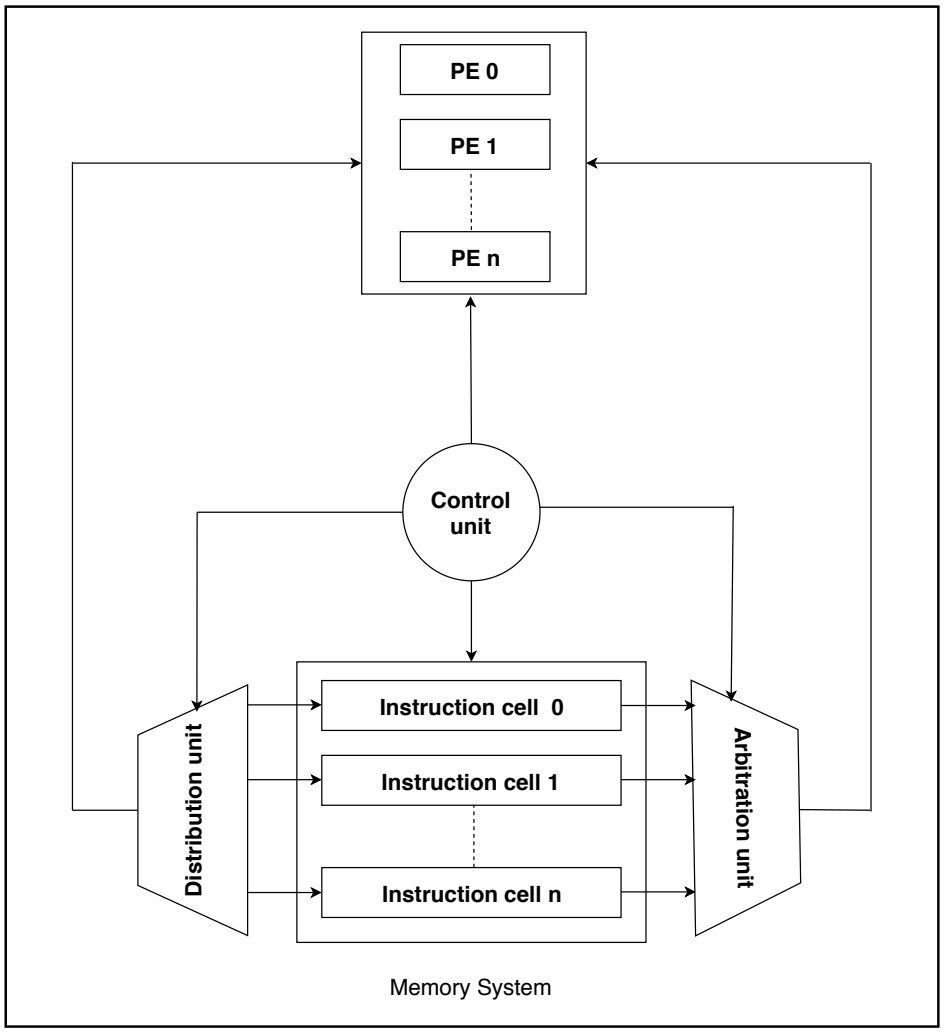


Figure 3.5: Data-flow machine.

The instruction unit operation packet contains operation code (opcode), operands and a destination PE address. The arbitration network is responsible for the proper distribution of the operands and opcode to the corresponding processing element of interest. Once the dispatched instruction is successfully executed, the results are sent back to the destination memory, which is overseen by the distribution network. The return packet contains the result and the destination memory address.

The data-flow based processors employ a 1-D array of Processing Engines and this is extended to a 2-D array to get a 2-D *systolic array* architecture. The systolic system consists of several interconnected Processing Engines, each capable of performing simple tasks like multiplication and addition. The control logic depends on the type of function the array needs

to perform. The information in this system flows in a rhythmic fashion between the PEs and communication with external entity occurs at the boundary of the array. The data flow here can be in multiple directions, the operands and partial products flow between the Processing Engines, whereas only the final results flow in the temporal architectures. The systolic array is easier to implement due to its regularity, easy reconfiguration and modularity.

The computational tasks can be categorized into two families- compute bound and I/O bound problems. In problems which are bound by compute, the total number of computations outnumbers the total number of input and output elements, otherwise it is I/O bound [31]. The systolic array is most suitable for compute intensive tasks such as linear equations, inversion and matrix multiplication. The convolution layer of DNNs which has repeated convolution operations can become I/O bound even if the convolution operation is compute bound. The PEs in the array can have a memory component which stores the data if needed and there is data transfer between the PEs. These two advantages result in lesser communication to the main memory, thus reducing the memory bandwidth and energy while increasing the throughput.

Based on data stream types, the systolic array can be of different shapes and types. The shape of the array is usually rectangle or square because it provides maximum utilization of array [31]. The different array types, its advantages and disadvantages are explained, assuming the following common terminologies:

- Each PE performs a simple MAC operation, where operands are stored in registers.

$$R_{\mathrm{c}} = R_{\mathrm{c}} + R_{\mathrm{a}} * R_{\mathrm{b}}.$$

- The total number of required PEs in the array is P. And the total time units required to obtain all the results of the operation is T.

- Let n be the size of the input matrices A and B. And C be the resultant of the matrix multiplication operation $C = A * B$. The following designs perform this operation.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} C = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

1. *Pipelined Arrays*
   This systolic array is rectangular in design. The constituent elements of the matrices *B* and *A* are inserted from upper and lower sides in a pipeline fashion [32], one skewed row and skewed column at a time as shown in the Figure 3.6. In general, for a two nxn matrices, we have:

$$P = n(2n - 1) \qquad T = 4n - 2.$$

   The array is simple in design, but on an average 50% of the PEs are idle during operation and the input elements are not pipelined every unit time into each PE.

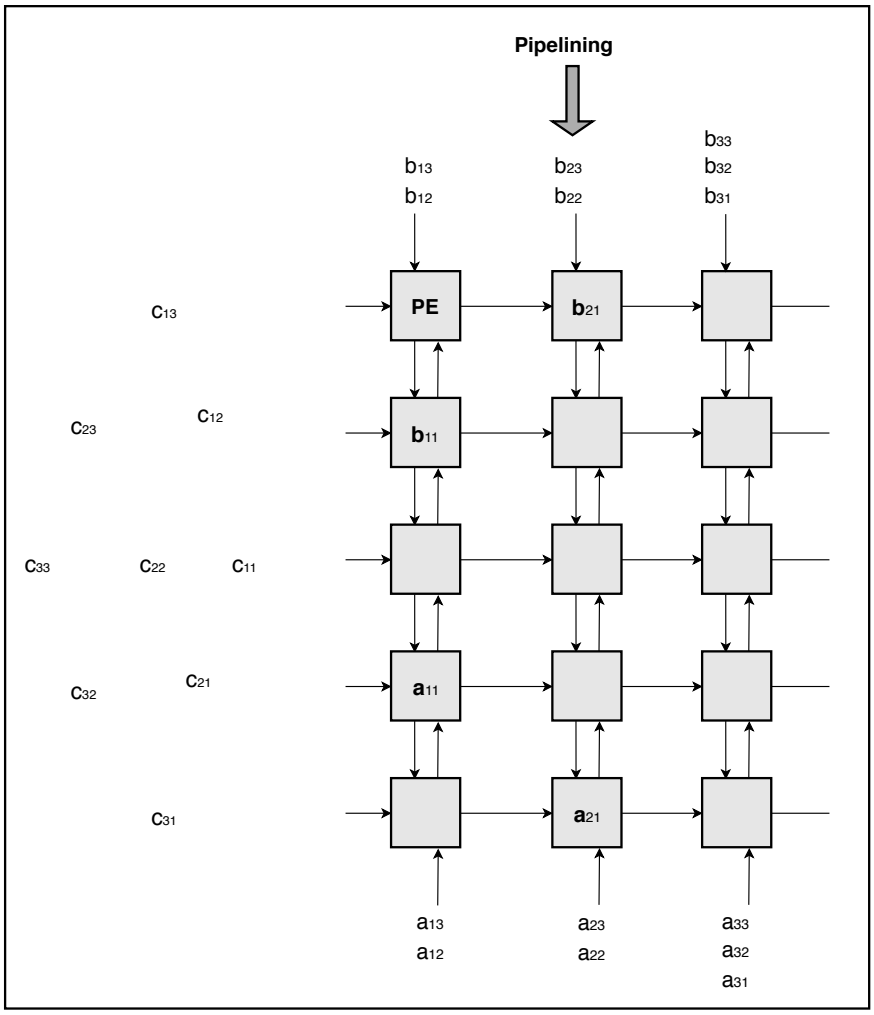


Figure 3.6: Pipelined array block diagram.

2. *Semi-broadcast Arrays*
   The semi-broadcast indicates one dimensional broadcasting [32]. From the left of the array, the matrix *A* is broadcasted while the matrix *B* is pipelined from the top as shown in the Figure 3.7 . In general, for a two nxn matrices, we have:

$$P = n^2 \qquad T = 3n.$$

   This design uses less turn about time as well as a lesser number of PEs when compared to *Pipelined Array*. Here the data movement time due to broadcast may be longer and the complexity of the control is higher than *Pipelined Array*. There is a need to conduct a trade off between the cost and time.

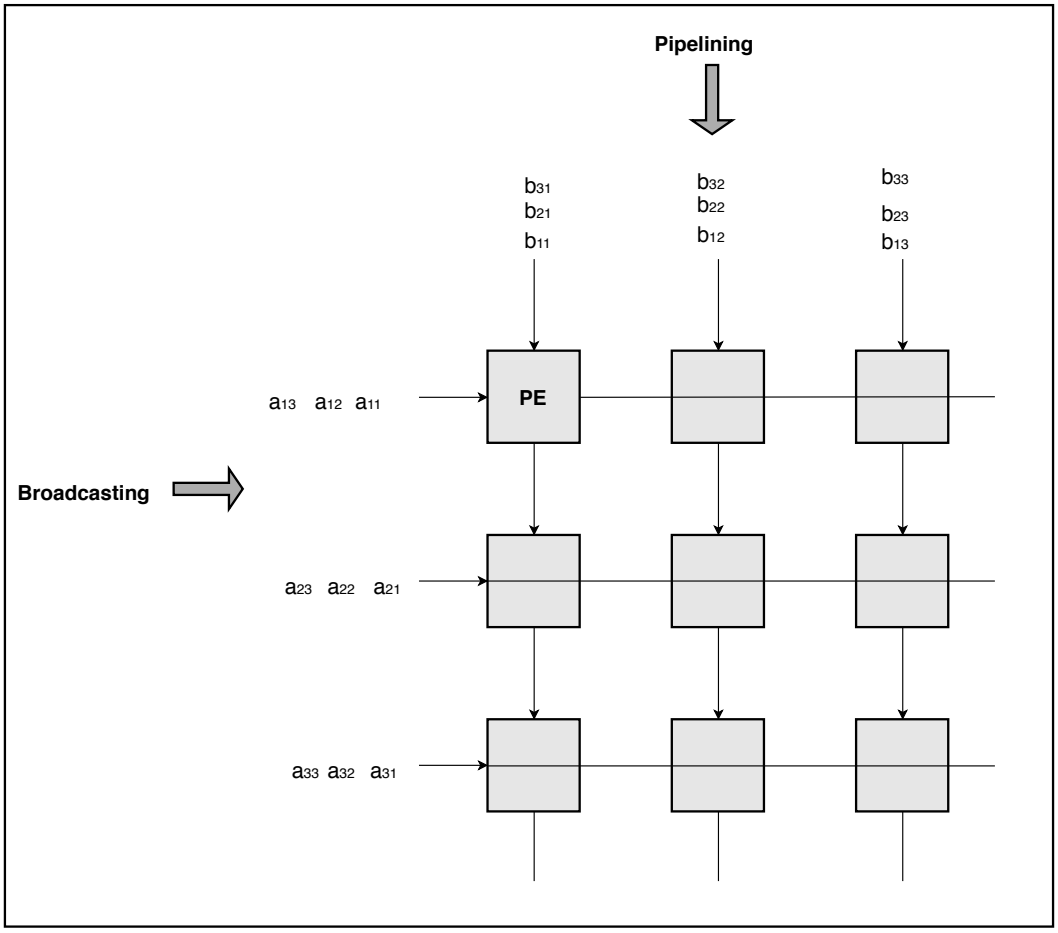


Figure 3.7: Semi-broadcast array block diagram.

3. *Broadcast Arrays*
This approach is a 2-D broadcasting scheme [32]. Here the data is broadcasted from two sides, by row, and by column, across the array as shown in the Figure 3.8. In general, for a two nxn matrices, we have:

$$P = n^2 \qquad T = 2n.$$

This approach has the same turn-about time as the *Semi-broadcast Arrays*. The utilization of the PEs is higher when compared to the other approaches. The degree of complexity for the control is highest because both the matrices are broadcasted.

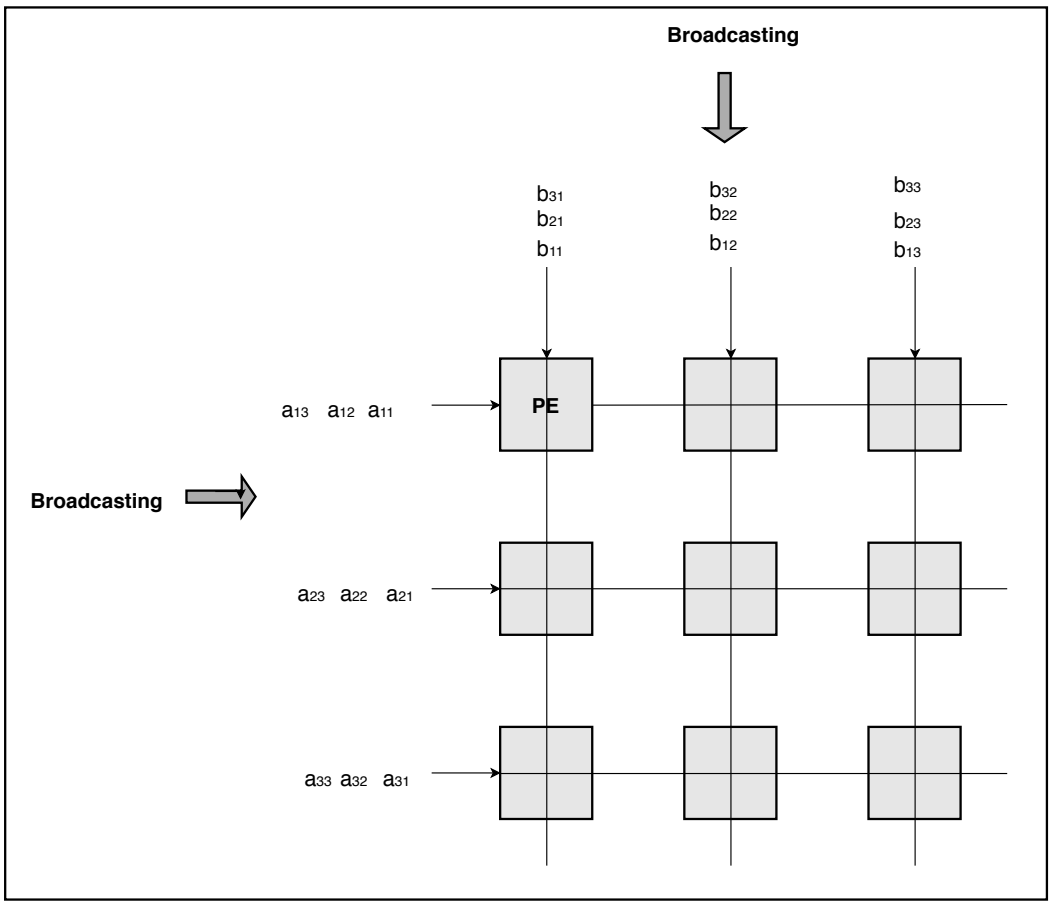


Figure 3.8: Broadcast array block diagram.

3.1.3 Data-Flow Representation

Based on the data movement, the DNN data-flow can be classified into four categories:

1. *Output Stationary (OS) data-flow*
   Definition:
   This processing is a method which aims at minimizing the energy consumption of writing and reading of the partial products. Here, each output feature map pixel is kept stationary in a particular PE. This approach aims at minimizing the accumulation cost of partial products and sums.
   Operation:
   The PE array processes the convolution in a 4-Dimensional (4-D) fashion, the partial sum of one convolution window from one channel is stored in the RF and added to the corresponding partial sum from the next channel. The partial products and sums are temporally accumulated in the RF of the PEs. The input activations are streamed and weights are broadcast across the PEs as shown in the Figure 3.9 [8].
   Examples:
   ShinDianNao [5] uses the output stationary approach, here each output activation is handled by a particular Processing Engine. The inputs for the corresponding PE is fetched from the neighboring PEs. A vertical and horizontal network is formed to communicate data between the PEs. The Processing Engine contains control logic to store the data for the required time before passing it to the next PE. The partial products are accumulated and stored in each PE of the array and then they are streamed out to the global memory (buffer). Other examples of this approach can be seen in works [33, 34].

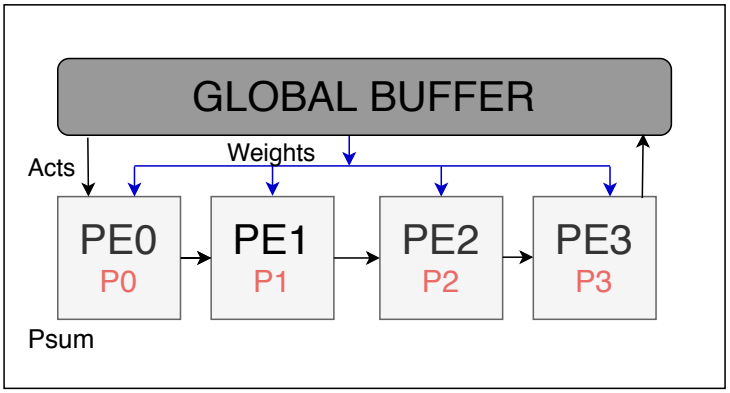


Figure 3.9: Output Stationary (OS) data-flow.

2. *Weight Stationary (WS) data-flow*
Definition:
This data-flow aims at minimizing the total energy incurred due to reading of filter weights by utilizing the data stored at Register File in Processing Engine maximum number of times. Once the weights are fetched from the global/external memory the convolution is performed P^2 number of times to generate all the output pixels which use the corresponding weights.
Operation:
Weights are read one by one from the DRAM into the Register File of each Processing Engine and are held stationary till it is not needed anymore. The processing runs as many MAC operation as possible on this data while it is stationary in the Register File, this maximizes the convolution and filter reuse. The input and output feature maps have to move through the PE array and global memory as shown in the Figure 3.10 [8]. The feature maps(activations/input feature maps) are broadcasted to all the Processing Engines and subsequently, the partial products are accumulated spatially across the Processing Engines. This method is most suitable for implementation of the convolution layer since there are filter and convolution reuse.
Examples:
The co-processor [35] employs eight 2-D convolution engines for processing. There are 100 Processing Engines per engine with each having its own local memory for storage of weights. Other examples of this approach can be seen in works [36, 37, 38].

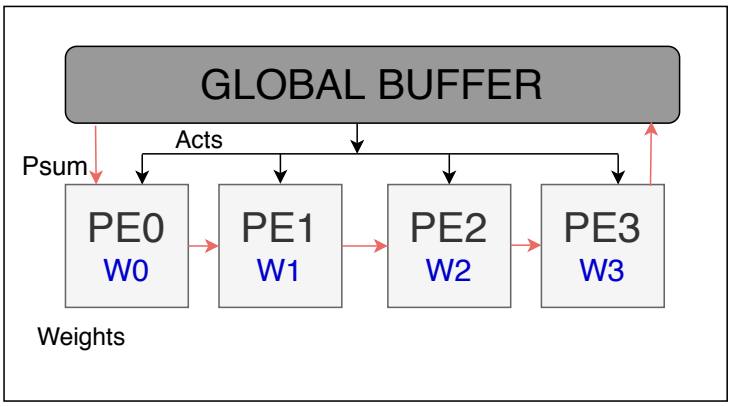


Figure 3.10: Weight Stationary (WS) data-flow.

3. *No Local Reuse (NLR)*
Definition:
As the name suggests no filter weights, input feature maps or activations are stored in the Processing Engine. The PE contains no local memory for data reuse. This approach is uneconomical in terms of energy per bit and economical in terms of area per bit.
Operation:
The partial products, activations, and weights are directly fed from the global buffer (memory), which is large. There is increased traffic to the global buffer as all the data types are stored in the global buffer. The input activations are multi-cast, weights are single-cast and partial products are spatially accumulated across Processing Engines

[8].

Examples:

The work from UCLA [39] uses this approach, and here the activations and filters are read from the global buffer and the PEs process the data with custom adder designed using tree approach. It can complete a single MAC operation in one cycle. Few other examples of this approach are seen in works [10, 11].

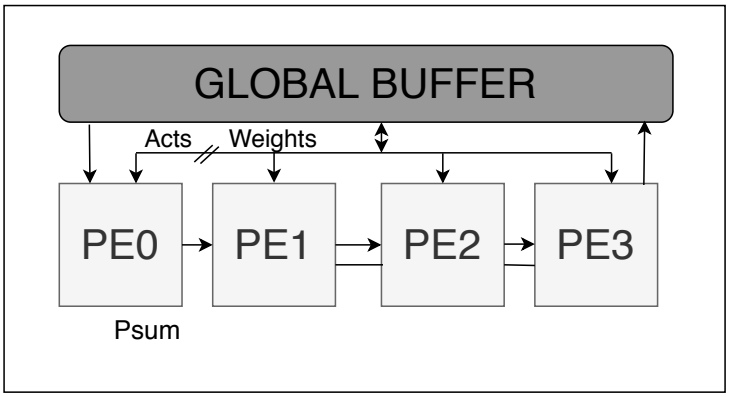


Figure 3.11: No Local Reuse (NLR) data-flow.

3.1.4 Numerical Representation

The numerical representation of the operands and parameters play an important role in the energy consumption and latency of the implementation. The works by [40, 41] have shown that the precision of weights and activations can vary in the range of 4 - 9 bits for AlexNet model across different layers and this would not change the accuracy of the implementation by more than 1%. These fine-grained variations in the model can be exploited to design a low precision Processing Engines which can support lower precision floating point (flp) or/and fixed-point (fxp) operands. In recent works like Efficient Inference Engine [42, 33, 43, 44] supports these different types of numerical precision.

The standard floating-point operation is of 32-bit in nature and is represented by $(-1)^s \times m \times 2^{(e-127)}$, where s is the sign bit and m is the 23-bit mantissa and e is the 8-bit exponent. The N-bit fixed-point number is represented by $(-1)^s \times m \times 2^{-f}$, where s is the sign bit and f determines the location of the decimal point and performs scaling action, the m is the (N-1)-bit mantissa. The Figure 3.12(a) shows the comparison between 32-bit floating-point and 8-bit dynamic fixed-point number representation. The *dynamic fixed point* format allows f to change value with respect to the expected dynamic range as described in the Figure 3.12(b). This dynamic range is advantageous in DNNs because the range of feature maps, activations and weights values may be different. Along with this, the dynamic range of the data types can also vary across layer types and between layers (example: *convolution layer and fully connected layer*). The dynamic fixed-point allows the bit-width to be varied to 10-bits for activations and 8-bits for weights [44].

Table 3.1 shows the energy and area comparison of 8-bit adder and multiplier unit with 32-bit floating and fixed-point designs[45]. Furthermore, the area and energy of fixed-

point adder unit scale proportionally with the bit-width. The area and energy of fixed-point multiplier unit scale quadratically with the bit-width [8, 43].

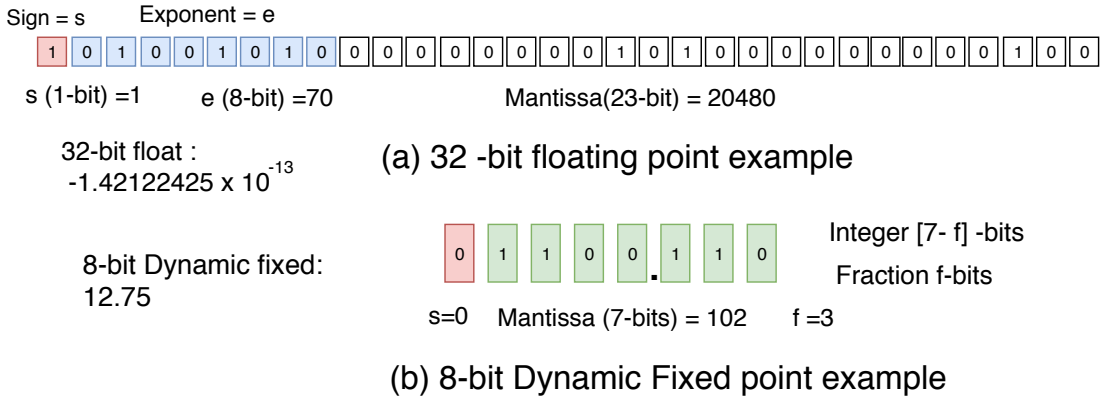


Figure 3.12: Dynamic fixed-point representation.

	Add Unit		**Multiply Unit**	
Factor	32-bit fix. pt.	32-bit fl. pt.	32-bit fix. pt.	32-bit fl. pt.
Area	3.8x	116x	3.8x	116x
Energy	3.3x	30x	3.3x	30x

Table 3.1: 8-bit fixed-point comparison.

By reducing the numerical precision, the energy and cost of storing the data in the local/global memory also reduces. This is because data transfer and memory access dominate the total energy consumption. The area of memory and total energy consumption scales linearly with the bit-width. The bit-width of the operands have to be reduced when changing from floating-point to fixed-point because there are no decrements in the area of the memory or the energy consumption if the precision is not reduced.

3.1.5 Near-Memory Processing

The preceding section analyzed DNN layers from a hardware perspective. The energy required to move the data from different levels in memory hierarchy dominates the energy required for computation. The spatial architecture provides a way to reduce the data movement by storing the loaded data in its local memory, which is closer to the computation unit (in the PEs). Significant efforts have been made to drive the off-chip memory or different memory types nearer to the computation. This section explores a few different memory technologies.

1. *DRAM*
   This memory is of high density and can reduce the total access energy. To avert the high energy cost of switching off-chip memory capacitance, a variant of DRAM called *embedded DRAM* (eDRAM) can be used as on-chip memory [46]. When compared with SRAM the density of eDRAM is 2.5x greater, and when compared with DRAM it is 321x more energy-efficient [10]. The bandwidth of eDRAM is more than DRAM,

and the former provides lower latency. This type of memory is more suitable for DNN processing, as it can store tens of acivations and weights in the on-chip memory, thus avoiding off-chip memory accesses [10]. The drawback of eDRAM is the lower density when compared to that of off-chip DRAM and this can increase the overall price of a chip.
Another variant of eDRAM would be the 3-D Memory, where the memory is stacked above the chip via Trough Silicon Vias (TSV). This memory is commercialized via High Bandwidth Memory (HBM) [47] and Hybrid Memory Cube (HMC) [48]. The 3D-eDRAM provides higher bandwidth and reduces energy per access by a factor of 5x when compared to the 2-D eDRAM. Few DNN works have been based on HMC, Neurocube [49] integrates the processors into the HMC die to bring the computation and memory near each other. The work Tetris [50] uses HMC with the work done in Eyeriss [51]. It achieved an energy reduction by 1.5x and an increase in throughput by 4.1x over a baseline chip containing 2-D DRAM.

2. *SRAM*
The standard on-chip local memory is based on SRAM technology. It is a non-volatile memory, it does not need to be refreshed periodically. SRAM is relatively more expensive than DRAM. Typically SRAM used 6T transistor model and is bulkier when compared to DRAM, but SRAM has lower latency than DRAM. The works [51, 42] are based on smaller local memory for storing the weights and activations, which is less than 512 Bytes.

3.1.5.1 DNN ASIC Accelerators

The previous sections explained systolic array, data-flow representation, reduced numerical precision and near-memory processing, which can be included in the design of edge devices to make them more efficient. Few of these improvements have been included in the works by [51, 42]. The state of the art designs are shown in the Table 3.2. The Eyeriss [51] uses small on-chip memory when compared to Efficient Inference Engine (EIE) [42]. They use different technology nodes and operating frequencies. EIE uses pruned network [42] to achieve high throughput and low operand precision. Eyeriss, on the other hand, used 16-bit operands and has small memory per PE. Here the architecture is fixed even though they have high performance. Other such architectures have been proposed in [52, 53, 11, 38]. All these works target a particular use case like Multimedia processing, speech processing, etc with fixed architectures. As new use cases are identified in near-to-far future which may require a wide range of performance and efficiency demands. There is a need to quickly prototype such designs. However, there are many parameters involved in the design, such as the number of Processing Engines required, memory size needed, operand precision and many more. Not to mention the different topologies of DNNs, which have a significant impact on the optimal hardware architecture. All the works that have been proposed target a particular application and cannot be used in other applications efficiently.

As there is a need for quick design space exploration of DNN accelerators, this work identifies a few key design parameters and analyzes the scalability of performance metrics when the design parameters are varied. Having this goal, a Systolic Array Simulator for CNN

(SASCNN) is proposed, which is further explained in the next chapter.

Metric	**Eyeriss** [28]	**EIE** [42]
Technology	65nm	45nm
Supply(V)	1	1
Area(mm^2)	12.2	40.8
On-chip Memory(*kB*)	192	10368
Precision(bits)	16	4
Number of PEs	64	168
Power(mW)	278	579
Frame rate(fps)	8	20
Throughput(GMACs/sec)	33.6	51.2
Frequency (MHz)	200	800

Table 3.2: State of the art DNN ASICs.

In this chapter we covered the advantages and disadvantages of different spatial systolic array architecture, the reuse mechanisms in CNNs, and analyzed the different data-flow representations like WS, OS, and NLR. Subsequently, the impact of numerical precision on energy and area of the adder and multiplier was explored. The different memory technologies suitable for Near-Memory processing was discussed. In Chapter 4, we will integrate the above explored design options in SASCNN.

4 System Design

> It had long since come to my attention that people of accomplishment rarely sat back and let things happen to them. They went out and happened to things."
>
> — Leonardo da Vinci

This chapter is outlined as follows: We start with the explanation of the simulator, then provide a brief introduction to the system design language used. Then we explain the memory modeling tool used, and subsequently, cover the design of the individual components of the system and its working.

For the training of the networks, the model is run on the cloud which may have GPUs or multi-core systems catered for high performance. These systems consume immense power and the same architectural parameters of high performance systems cannot be used for inference on edge computing devices like robots, autonomous vehicles, and smartphones. These edge devices need to be very low power and low latency with a reasonable high performance. Depending on the application and data format, the performance needs from the architecture varies. Since the design turn-around time is high and the the design parameters are fixed before-hand, there is a need for a tool that takes in plethora of architectural design parameters of interest and provides an estimate of performance metrics like latency, throughput, area and energy consumption. This work delivers a simulator, *Systolic Array Simulator for CNN* (SASCNN), which uses a systolic array to accelerate the layers in DNN. The Figure 4.1 shows the schematic of the simulator. The system takes in many design parameters and CNN layer hyper-parameters and generates the performance metrics.

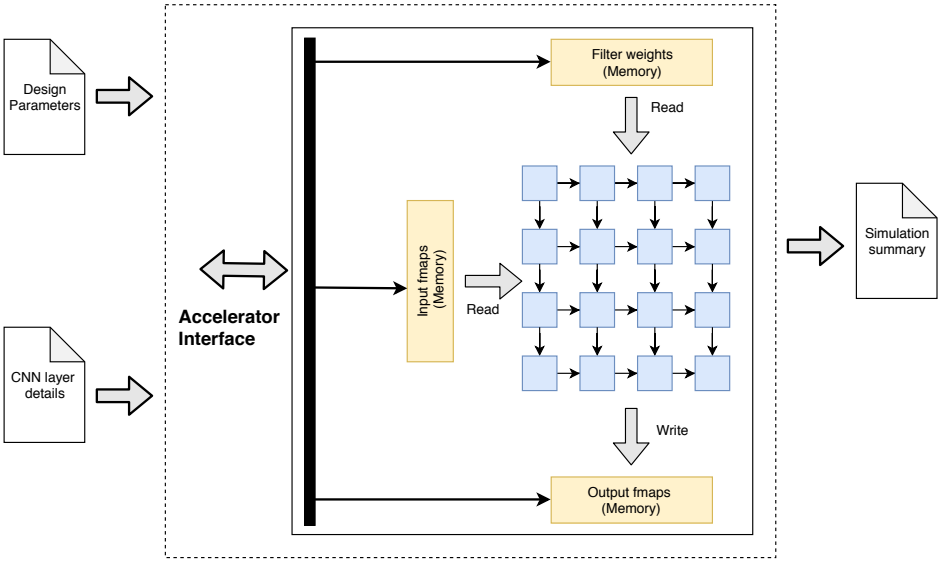


Figure 4.1: Schematic of SASCNN.

4.1 SystemC

SystemC is a library of C++ classes and it provides an event-driven simulation interface. It can be applied to architecture and performance modeling, system-level modeling and high-level synthesis among others. It provides a higher level of abstraction, which is independent of any detailed software and hardware implementation. It provides a complete language for modeling tasks pertinent for system design which are missing in complete or in part from the rest of the other languages as shown in the Figure 4.2. Transaction Level Modelling (TLM) provides early development of design and better hardware verification test bench. Partitioning and implementation of HW/SW portions become easier. It provides inbuilt data type like fixed point which is useful for this work. The predefined primitive channels help in the faster design of the system and software for testing, thus reducing the overall development cycle time. The Figure 4.3 shows the constituent elements of SystemC libraries [54]. The SASCNN work uses SystemC for modeling the systolic array architecture.

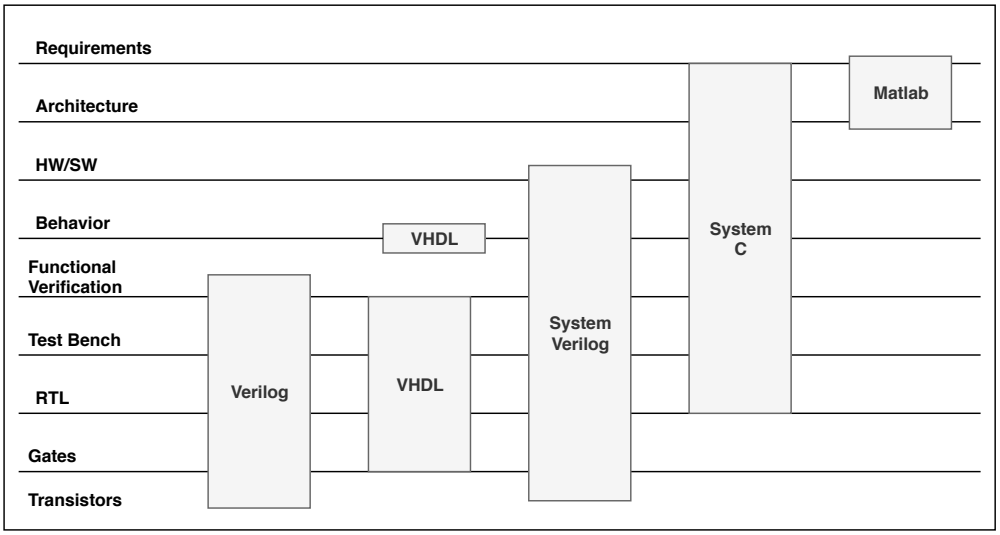


Figure 4.2: Comparison between SystemC and other design languages.

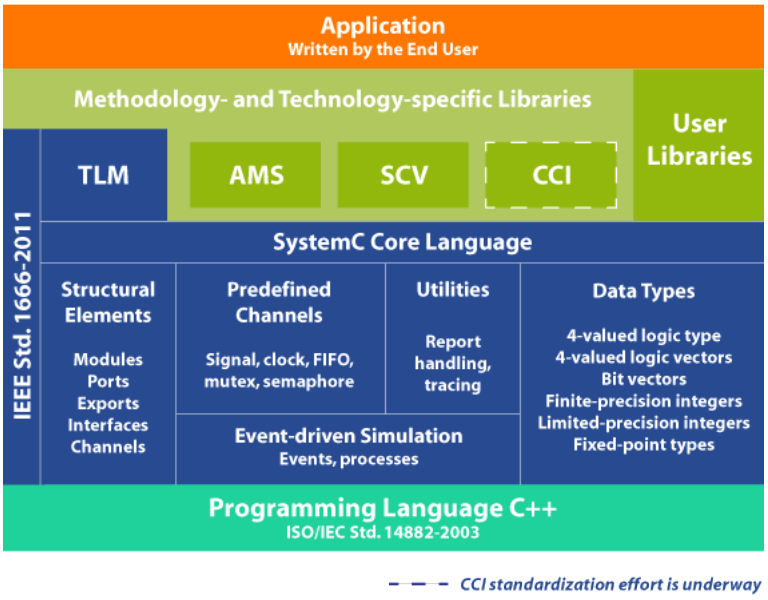


Figure 4.3: Architecture of SystemC language.

4.2 Memory Modeling Tool

The simulator is compatible with possible future memory technologies. For the analysis of the design, the simulator uses *DESTINY* memory modeling tool. It is a comprehensive tool with 3D and Multi-Level cell memory modeling capability [55]. It is used to model both 2D and 3D caches designed with five prominent memory technologies: Resistive RAM (ReRAM), eDRAM, SRAM, Spin Transfer Torque RAM (STT-RAM) and Phase Change RAM (PCM), thus covering both conventional and emerging memory technologies. The tool supports the technology nodes ranging from 22nm to 180nm.

4.3 Concurrency in CNNs

In the most widely used DNNs like AlexNet and VGGNet, the *conv* layers take up 90% of the total operations and contribute a large portion to the total data movement. This makes it a significant contributor to the energy efficiency and throughput of the networks. The work by Han and team [42], shows that the *fc* layers weights can be largely compressed to 1-5% of their original size, thus reducing the impact of fully connected layers. Nonetheless, both types of layers contribute the most to the computation time and data movement. Thus, analyzing these layers to extract parallelism is important. The shapes and parameter of the convolution and fully connected layers are explained in the Table 4.1.

Parameter shape	Explanation
N	No. of 3-D filter
CH	No. of filter channels
H(=W)	Input feature map height/width
X(=Y)	Filter weight/width (= I in FC)
P(=Q)	Output feature map height/width (= 1 in FC)

Table 4.1: The *conv/fc* layer parameters.

Lets us consider the Equation 4.2 for analyzing the parallelisms that convolution layers inherently exhibit. The *Conv2D* is the 2-D convolution operation function.

$$\forall\{n\} \in [1, N] \tag{4.1}$$

$$\mathbf{O}^{\text{conv}}[n] = \mathbf{B}^{\text{conv}}[n] + \sum_{c=1}^{CH} \mathbf{Conv2D}(\mathbf{I}^{\text{conv}}[c], \mathbf{W}^{\text{conv}}[n, c]) \tag{4.2}$$

- *Inter-Output Feature Map Parallelism*
  Since each channel of filters is convolved with the corresponding channel of the input feature maps to get an output channel, the output channels are independent of each other. Each output channel can be processed separately from the others. This implies that P_N ($P_N < N$) elements of $\mathbf{O}^{\text{conv}}$ (Equation 4.2) can be computed in parallel.

- *Intra-Output Feature Map Parallelism*
  The pixels in a single channel of output feature maps can be processed independently and concurrently. The O_P x O_Q values of $\mathbf{O}^{\text{conv}}$[n] can be processed concurrently ($0 <$ O_P x $O_Q <$ P x Q).

- *Parallelism in Inter-Convolution*
  The 3-D convolution in any convolution layer can be expressed as a sum of multiple 2-D convolution as shown in the Equation 4.2. The 2-D convolutions can be computed in parallel by computing P_C elements of Equation 4.2 concurrently ($P_C < CH$).

- *Parallelism in Intra-Convolution*
  The individual 2-D convolutions between the weights and input channel can be implemented in a pipeline fashion. Here all the individual multiplication with the filter kernel

is implemented in parallel [51], P_X x P_Y multiplications can be executed concurrently ($0 < P_X$ x $P_Y <$ X x Y).

4.4 Simulator Architecture

The simulator is based on the systolic array and row-column stationary data-flow approach. The Figure 4.4 shows the core elements of the systolic array architecture. Each PE is connected to its 4 neighboring PEs, except the PEs in the border which is only connected to 3 PEs. The data flows from top to bottom and left to right when in operation. The weight scheduler sends the weights of the filter to designated PEs and the input scheduler sends the input feature maps to PEs in the left-most column in the array. The accumulator collects the results of the convolution. Each PE contains a memory unit, a MAC unit, a control unit, and a memory to store the weights. The detailed description is explained later in this section.

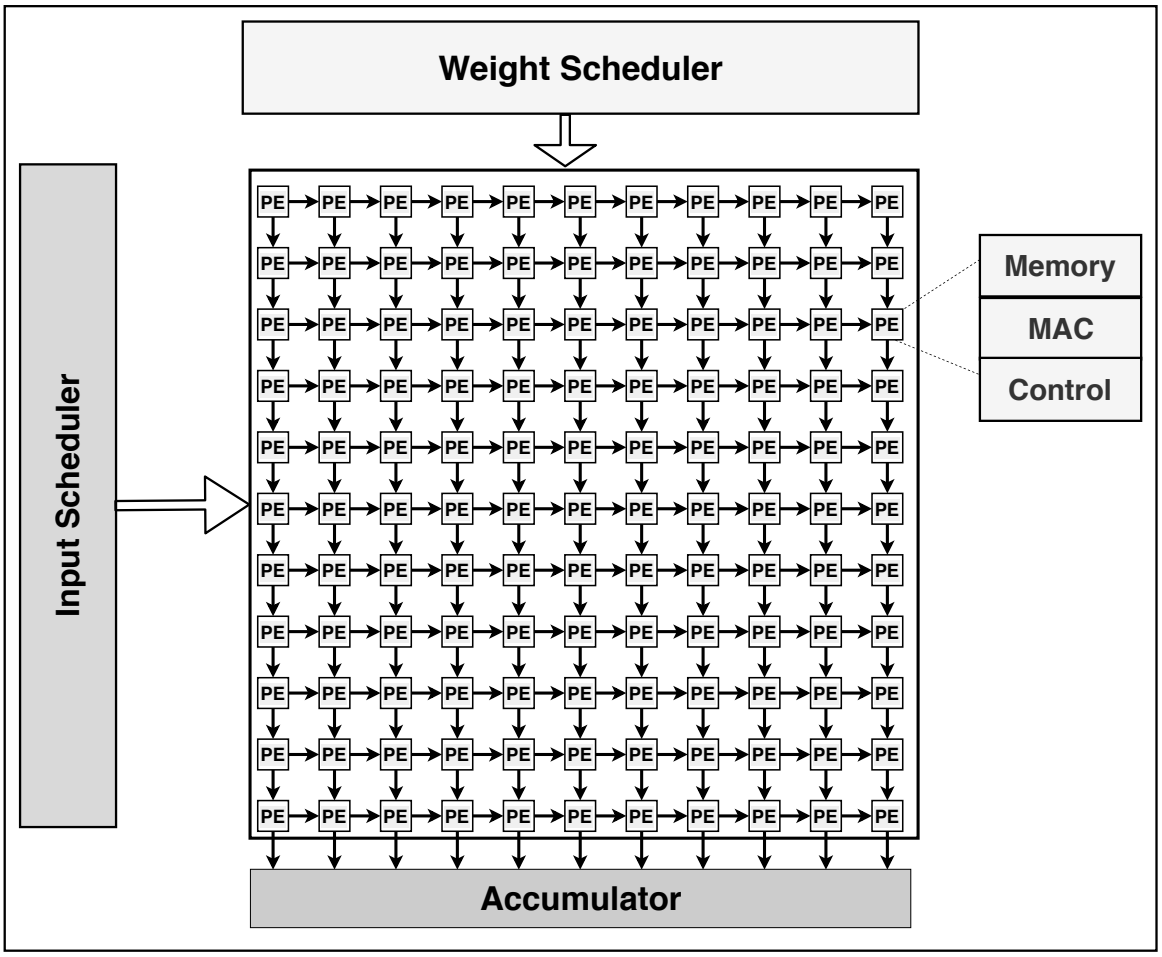


Figure 4.4: Block overview of the proposed architecture.

4.4.1 Modelling Setup

The SystemC model of the proposed architecture is shown in the Figure 4.5. The test bench reads the filter weights, bias, and input feature maps and stores them in an internal array. The test bench mimics the operation of the off-chip processor, it loads the required data from the internal array to the local memory of the PEs. The PE is modeled as a MAC unit and the local memory in all the PEs is combined and modeled as a multidimensional array memory. Each input channel is stored in the *ifmap memory* which is scheduled by the *input scheduler*, and weights and bias are stored in the *weights memory* which is scheduled by the *weight scheduler* as shown in the Figure 4.5. The *output scheduler* collects the results of convolution and stored it in the *ofmap memory* and the *controller* controls the loading, scheduling operations.

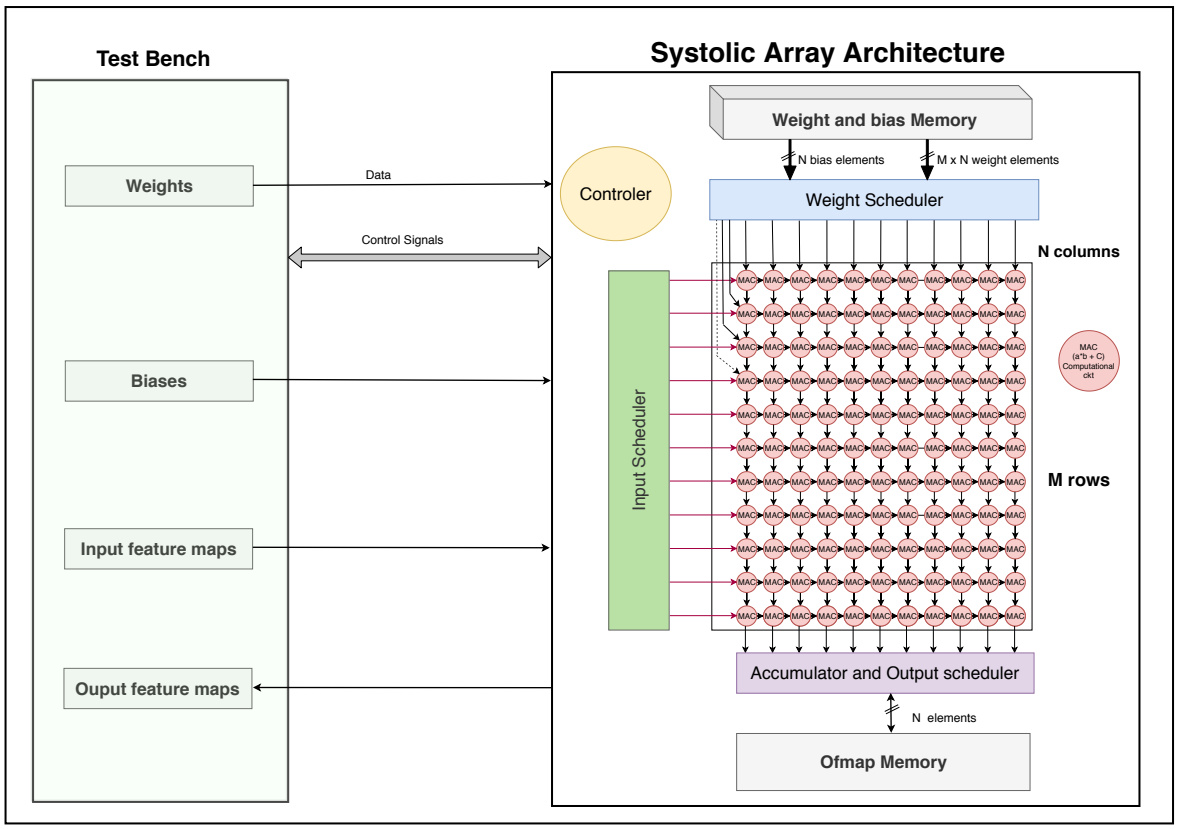


Figure 4.5: SystemC model of the architecture.

4.4.2 Simulator Parameters

The simulator takes in *design parameters* and *CNN parameters* as input from the user and generates a summary file consisting of performance metrics which is explained the next chapter.

4.4.2.1 Design parameters

The design space should be versatile enough to accommodate future technologies and cover different application needs. The Figure 4.6 describes the PE architecture, and the individual components are explained along with architecture parameters below:

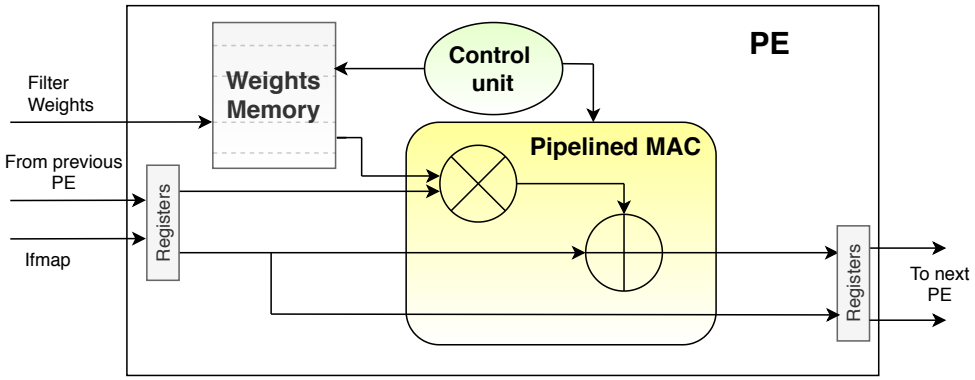


Figure 4.6: Processing Engine architecture.

1. *Size of the array*
   The size of the array gives the total number of PE needed. The size is defined by the *Array Height* (M - number of row) and *Array Width* (N - number of columns), therefore there are totally (M x N) Processing Engines as shown in the Figure 4.5.

2. *Latency in MAC operation*
   The PE contains a pipelined floating-point Multiply and Accumulate unit. The number of stages in the pipeline is variable and it is modeled in the form of latency in the simulator, as shown in the Figure 4.6.

3. *PE local memory size*
   The PE contains a local memory that is closer to the compute. The PE local memory should be able to store many filter weights and reuse them when necessary. The simulator models all the individual local memory of the PEs as a combined local memory. The size of this local memory in each PE is variable and this size is used for all the PEs in the array.

4. *PE local memory latency*
   This simulator supports different kinds of memory technologies like SRAM, eDRAM, ReRAM or any other possible future memory technology. The read and write latency of such memories is modeled in the form of variable.

5. *External memory latency*
   The accelerator is connected to the processor through a bus and the data is loaded from the external memory. The simulator also models the latency of the external memory in the form of a variable. This allows the simulator to be compatible with different external memory technologies.

4.4.2.2 CNN parameters

The simulator supports *conv* layers with different parameters, which are described in the Table 4.2. The simulator takes in one *conv* layer at a time and generates the results.

CNN Parameters	Description
Input Feature Map Height (H)	Vertical dimension of input channels.
Input Feature Map Width (W)	Horizontal dimension of input channels.
Number of Filters (N)	Number of filter kernels.
Filter Kernel Depth (CH)	Number of input channels/filter channels.
Filter Kernel Height (X)	Vertical dimension of input filter weights.
Filter Kernel Width (Y)	Horizontal dimension of input filter weights.
Padding (P)	Zero padding for the input feature maps
Stride (S)	Stride count during convolution operation

Table 4.2: CNN topology parameters.

4.4.3 Row-Column Stationary Data-Flow Mapping Methodology

The Chapter 3 explained various data-flow methodologies like *WS, OS, IS* and *NLR*. These approaches maximize the reuse of weights, input feature maps, output feature maps respectively. The corresponding mapping schematic is shown in the Figure 4.7 and Figure 4.8.

This simulator uses *Row-Column Stationary (RCS)* [51] approach which reuses the weights, inputs, and activations the most. The primary advantage of this data-flow is its ability to adapt to various conditions and optimize energy consumption for all three types of data movement. This *RCS* is explained in the following sections:

4.4.3.1 1-D Convolution Basic Block

This approach uses the idea of the strip mining method, which is often used in spatial architecture [56]. It breaks the 2-D convolution into smaller 1-D convolution basic blocks that can be executed in parallel. Each basic block operates on one row of the input filters and one row of the input feature map pixels in consideration for convolution and generates one row of partial sums. These partial sums are accumulated over time and space to get the output feature map pixels. The corresponding input filter weights for the convolution is stored in the local memory or the global buffer.
Each basic block is spatially and temporally mapped to one PE for execution. This approach keeps the filters in the PE stationary, thus utilizing the *inter-convolution parallelism* of CNN, which was explained in Section 4.3. The example of the 1-D Convolution basic block sliding window is shown in the Figure 4.9, here the PE performs the MAC operation for each sliding window of input pixels over time and stores the partial products in the registers. The sliding window goes over all the input pixels in a row to complete one 1-D convolution and also maximizes the weight data reuse. However, the convolution layers contain more than hundreds of thousands of such basic blocks, and since the mapping across PEs is non-trivial, the mapping impacts energy consumption and latency.

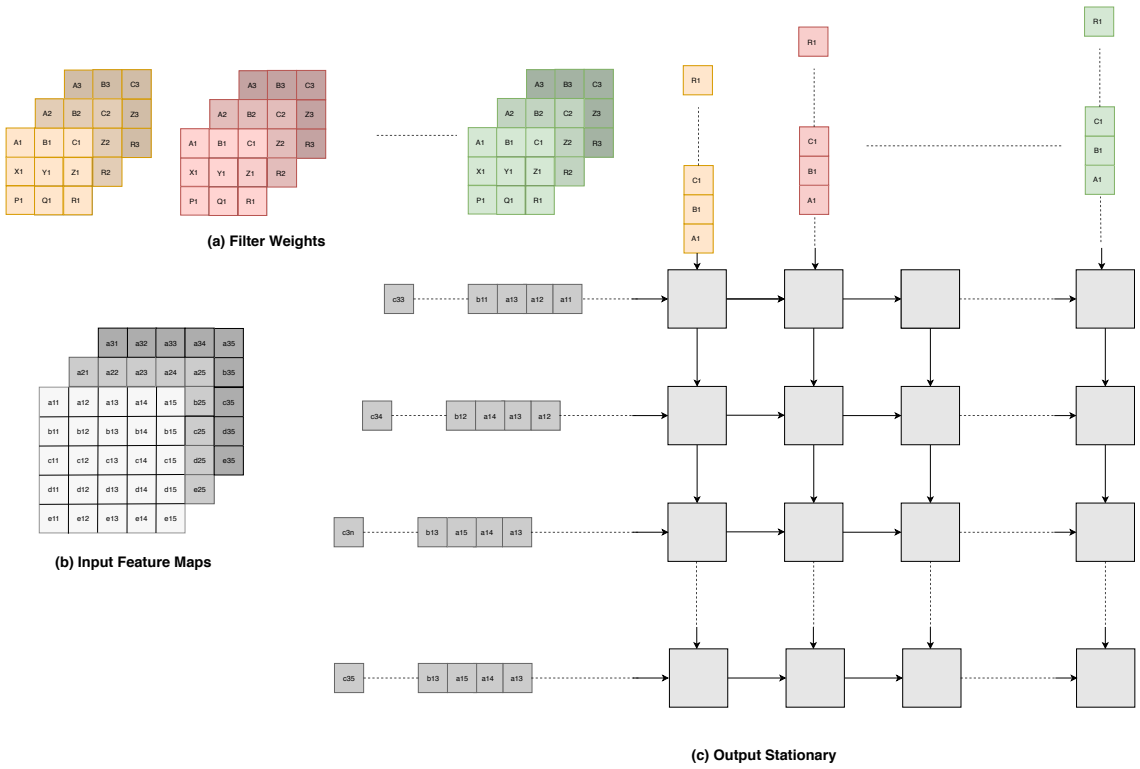


Figure 4.7: (a)Input feature maps, (b)Filter weights and (c)Output Stationary mapping.

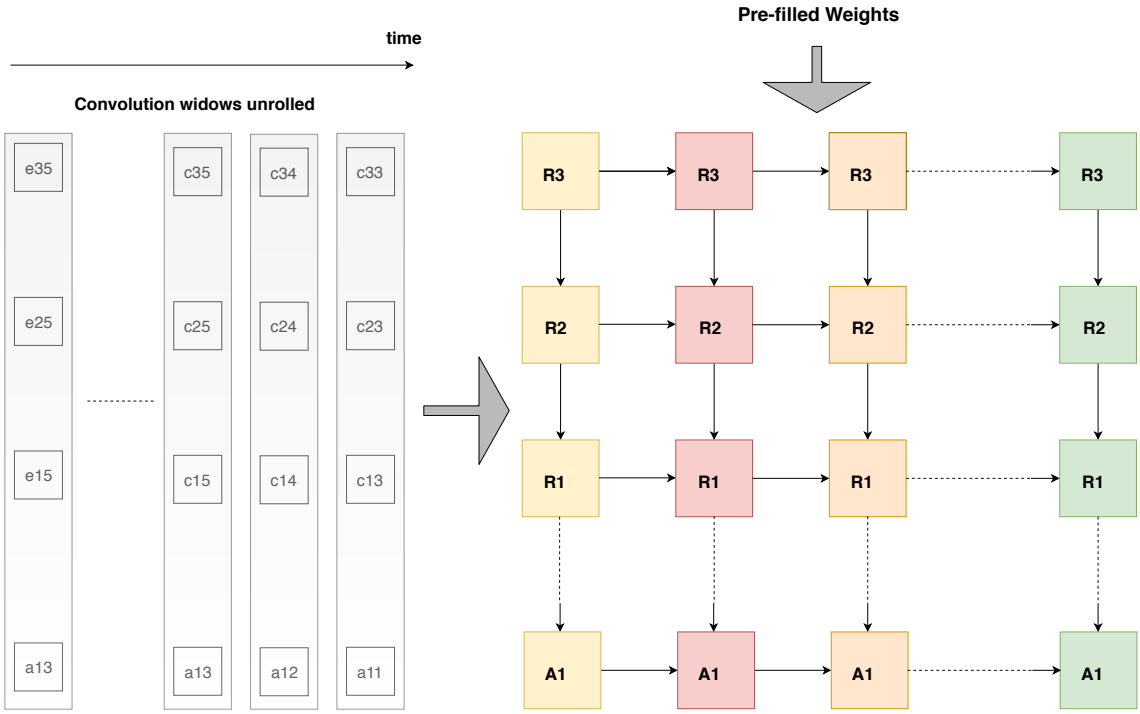


Figure 4.8: Weight Stationary mapping.

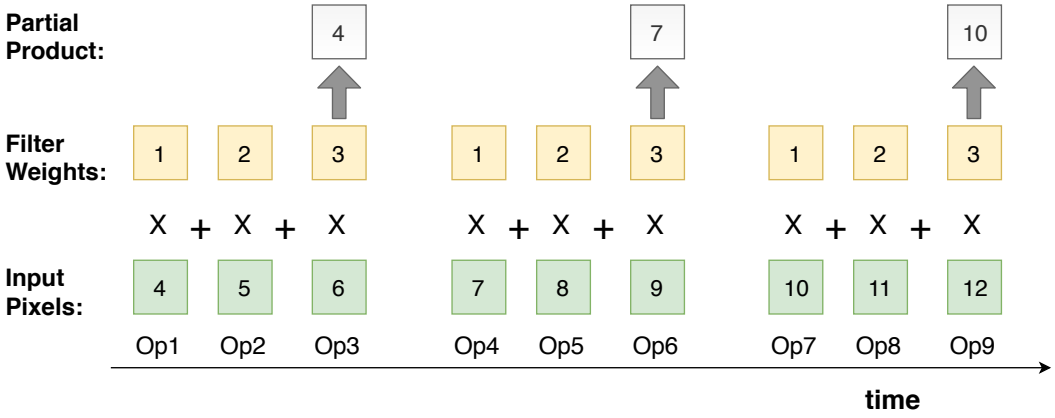


Figure 4.9: 1-D convolution basic block with X = 3 and H = 5.

4.4.3.2 2-D Convolution Mapping

The 1-D convolution can be extended across multiple rows to get a 2-D convolution while the results are accumulated vertically down the column shown in the Figure 4.10. One column of PEs store one set of filter weights and the input pixels are streamed to these PEs and the results are accumulated by the bottom-most PE to generate one output pixel. This 2-D mapping allows us to utilize inter-output parallelism, which was explained in Section 4.3.

This 2-D mapping allows us to map multiple filters across columns of PEs and obtain

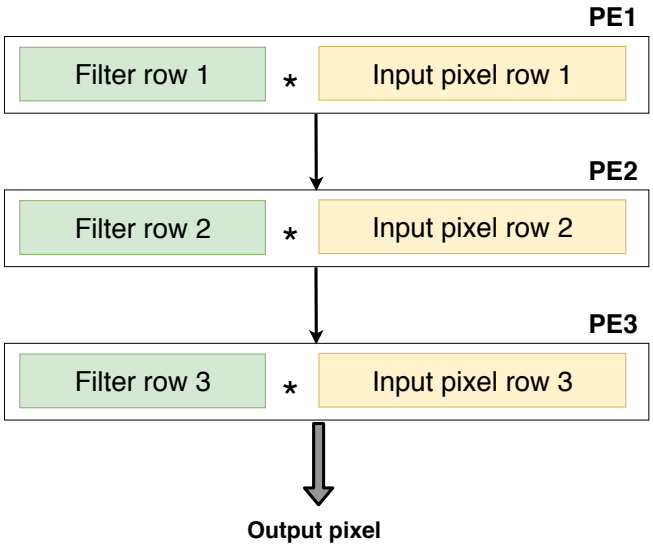


Figure 4.10: 2-D convolution mapping across one column.

multiple output feature map channels in parallel. Consider the Figure 4.11, here channel 1 (out of CH) of filter 1 (out of N) is mapped to the first column, channel 1 of filter 2 is mapped to the second column, channel 1 of filter 3 to the third column and so on. This mapping allows us to obtain partial results of multiple output channels in parallel. Since convolution

is performed with only the channel 1 of the input feature map and channel 1 of all the filters, the input pixels are maximally reused. The number of filters elements in each column is calculated by the size of the local memory in each PE. Since there are hundreds of thousands of 1-D convolution basic blocks, all of them can not be physically mapped to the PE systolic array in hardware due to constraints such as limited area and memory. Hence, the mapping is done in a two-step process:

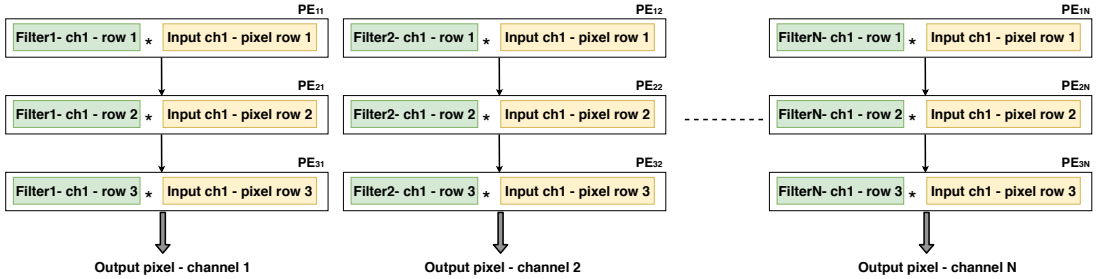


Figure 4.11: 2-D convolution mapping across systolic array.

1. *Logical Mapping*
   All the 1-D convolution basic blocks are deployed on a logical 2-D systolic array of PEs of the same size. Here each basic block is mapped to one logical PE in the array. A collection of several PEs in one column corresponds to one *conv-set*, this is replicated across columns to match the total number of 1-D convolution basic blocks. The height and width of each *conv-set* are determined by the filter size (X). Since there are *N* filter and *CH* filter channels, the total number of *conv-sets* required for complete processing on one *conv* layer is *N* x *CH*. The Figure 4.12 shows the logical convolution mapping, the weights which are stored in the local memory of each PE is fed to the MAC operation, the PEs shift the input pixels to the next PE, the partial products are collected down the column as explained in the Algorithm 1.

2. *Physical Mapping*
   This is a process of mapping different and multiple logical *conv-sets* onto the same physical PE and running it. By mapping multiple *conv-sets*, there is maximum reuse of input feature maps, and the output feature maps are accumulated across every column. The number of logical *conv-sets* to be physically mapped depends on the local memory size, the number of the filters (N) and the size of the PE array, which are user input values, as explained in Section 4.4.2.1. When all the filter channels and input feature map channels corresponding to the mapped *conv-sets* are executed, a new set of *con-sets* are mapped. This process is called *process-pass* and it iterated till all the filter and input feature map channels are convolved. The Figure 4.11 shows the physical mapping of *conv-sets* onto the PE array.

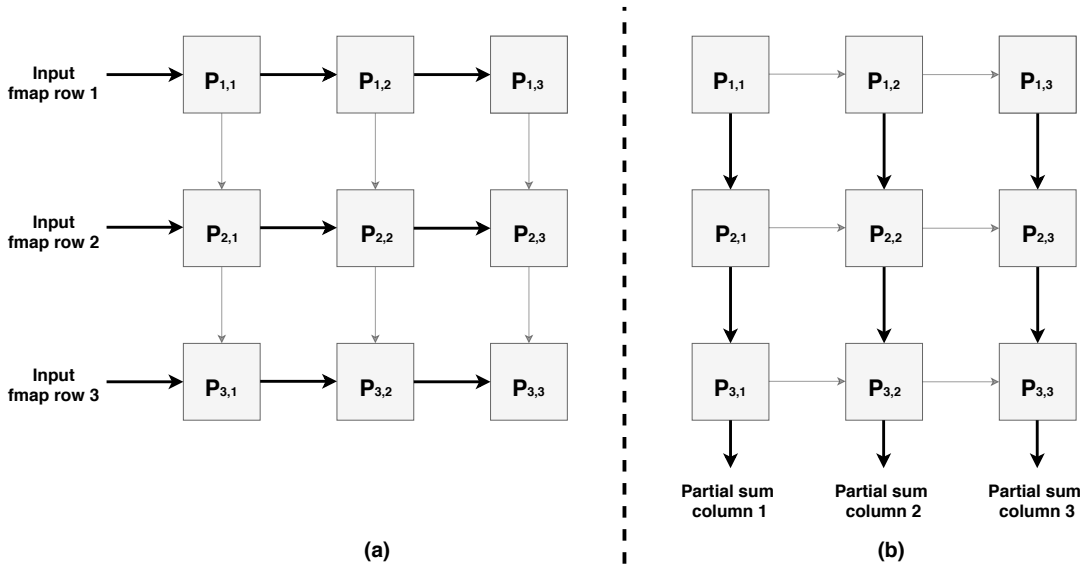


Figure 4.12: Data-flow in multiple *conv-set* to process multiple 2-D convolutions. (a) inputs are reused across columns, (b) partial sums are accumulated across columns for different output channels.

Algorithm 1: Logical Mapping

```
Initialization;
CH = depth of the filter weights;
ncs = N x CH;
cols = number of columns in the array;
i = 0;
while i < (ncs/cols)) do
    j = 0;
    while j < cols do
        map i^th conv_set to column j ;
        j++ ;
    end
    i++;
end
```

4.4.4 Algorithms

After the logical mapping of *conv-sets* on to the physical PEs, the weights are loaded on to the local memory of each PEs in the array. Then the input feature maps and weights from the local memory are scheduled concurrently into the MAC engine. When a *conv-set* is executed, the output feature map pixels are collected and saved.

1. *Weights loading algorithm*
   The filter weights are read from a user input *csv* file into an array in the test bench

of the design. During the operation of the design, several *conv-set* called *conv-set-block* is loaded into the weight memory of the PEs in the array. The *con-set-block* size is determined by the size of the array and the PE local memory. After loading the *conv-set-block*, the input data is streamed and weights are scheduled till all the loaded weights are reused to generate the output data. Then a new *conv-set-block* is loaded and the cycle is repeated till the given *conv* layer is executed as as shown in the Algorithm 2.

Algorithm 2: Weight loading algorithm

```
Initialization;
generate the physical mapping of all conv_set_blocks based on the given size of
  memory and array size;
n = number of conv_set_blocks;
i = 0;
while i < n do
    load conv_set_block[i];
    wait till conv_block_done_signal==1;
    i++;
end
```

2. *Weights scheduling algorithm*
Once the weights in the *conv-set-block* are loaded into the memory, the *conv-sets* from this memory, corresponding to the size of the array is selected and time unrolled. This unrolled weights of the *conv-sets* are loaded one weight at a time in parallel to the MAC unit in the PEs. When all the weights are scheduled and reused to the maximum extent, a new *conv-set-block* is loaded and scheduled, and the cycle is repeated till the given *conv* layer is executed as shown in the Algorithm 3.

Algorithm 3: Weight scheduling algorithm

```
Initialization;
ncs = number of conv_sets in the loaded conv_set_block;
M = number of rows in the array;
N = number of columns in the array;
i=0;
while i < (ncs/N) do
    j = 0;
    while j < N do
        time_unroll and schedule conv_set[i*N + j];
        j++ ;
    end
    wait till conv_block_done_signal == 1;
    i++ ;
end
```

3. *Input scheduling algorithm*
The input feature maps are read from a user input *csv* file into an array in the test bench of the design. The input feature maps can be divided into many windows

which are used in *conv-sets*. These input windows are calculated with padding and stride parameters, and these windows are time unrolled and fed to the PEs of the leftmost row of the array. This is continued till all the windows of the input feature map channel and the corresponding channel of all the weights are scheduled. If the input channel is fully used, a new channel is scheduled similarly, and the cycle is repeated till all the input channels are scheduled as shown in the Algorithm 4.

Algorithm 4: Input scheduling algorithm

```
Initialization;
C = number of input feature map channels;
N = number of windows in the input feature map channel adjusted to stride and
  padding;
i = 0;
while i < C) do
    j = 0;
    while j < N do
        time_unroll and schedule ifmap_window[i*C + j];
        j++ ;
    end
    wait till conv_channel_done_signal == 1;
    i++;
end
```

Algorithm 5: Output scheduling algorithm

```
Initialization;
ncs = number of conv-sets in the loaded conv-set-block;
C = depth of the filter weights;
N = number of windows in the input feature map channel adjusted to stride and
  padding;
i = 0;
while i < (ncs/N)) do
    j = 0;
    while j < N do
        Collect and store ofmap_window[i*N + j];
        j++ ;
    end
    wait till conv_channel_done_signal == 1;
    i++;
end
```

4. *Output scheduling algorithm*
After all the *conv-sets* corresponding to a channel and the size of the array is executed, the output pixels are stored in a memory outside the PE array. Then a new set of output pixels are loaded from the memory into the accumulator, the next *conv-sets* are executed, and the results are added to these output pixels. These new output pixels

are then stored in the output feature map (ofmap) memory and subsequently a new set of output pixels are loaded into the accumulator. This cycle is repeated till all the input channels are convolved with the corresponding weight channels as shown in the Algorithm 5.

4.4.5 Operational Flow

The simulation state diagram is shown in the Figure 4.13. In the initial stage of the simulation,

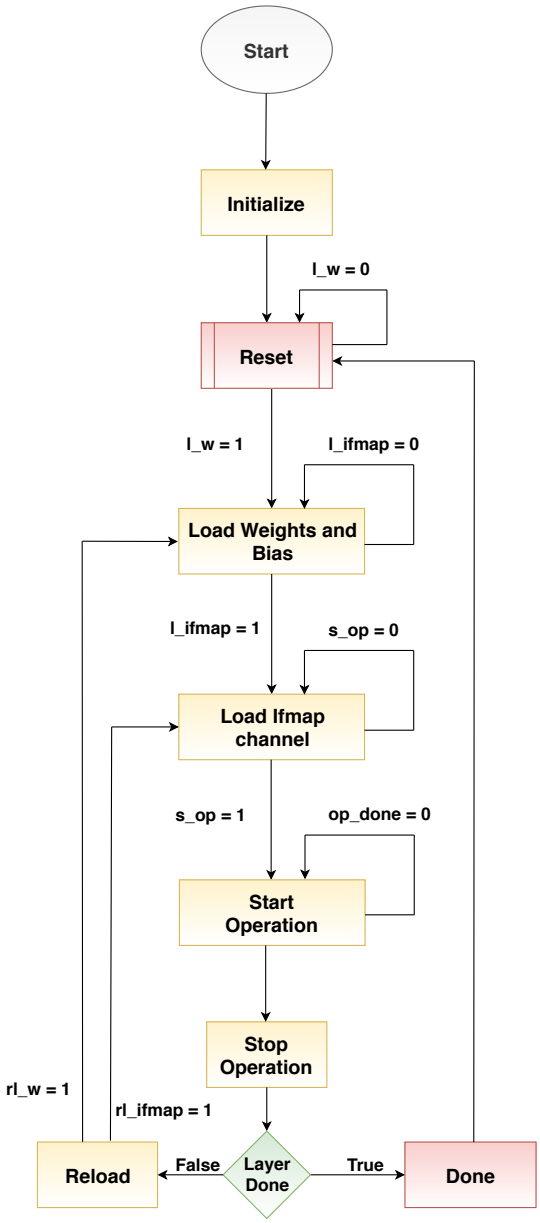


Figure 4.13: State diagram of the simulator.

the design parameters, CNN topology parameters, and input data are read from the files. Then

logical and physicals mapping of the *conv-sets* is done. Subsequently, the weights are loaded into the memory, input feature maps are streamed and convolution operations are performed till the relevant conditions are met. The weights and input feature maps are reloaded if needed. This cycle is repeated till the given convolution layer is completed. The results of the simulation are stored in the output file.
The input data and golden results are generated using Matlab for verification. The design is tested against the AlexNet parameters and as well as random weights and biases.

In this chapter we covered the explanation of the Systolic Array Simulator for CNN (SASCNN) and its SystemC modeling setup. Subsequently, we explained the memory modeling tool used. Then, concurrency in CNNs was explored which are utilized in the design of the architecture. The input parameters for the simulators were explained. The proposed row-column stationary data-flow mapping methodology was explained in depth which included logical and physical mapping, and loading and scheduling algorithms. Then we covered the operation flow and verification methods used. In Chapter 5, the detailed analysis of the results is covered.

5 Results and Analysis

> "If you fail, never give up because F.A.I.L. means "First Attempt In Learning". End is not the end, in fact E.N.D. means "Effort Never Dies." If you get No as an answer, remember N.O. means "Next Opportunity", So let's be positive."
>
> — Abdul Kalam

This chapter discusses the experiments done on standard CNN model and analyze the results obtained from the simulator. The performance metrics for 32-bit floating point design is discussed, and subsequently, 16-bit and 8-bit floating-point is covered. The interaction between different design parameters and the performance of the system is discussed for convolution layers of the CNN model.

The AlextNet model is used for analysis in this chapter. The individual layer details of AlexNet are shown in the Table 5.1 [2]. The simulator takes in only convolution layers, hence this chapter covers results and analysis of these. The performance of a design based on the given design parameters are analyzed for different numerical precisions of the system. Table 5.2 shows the energy and area comparison between different floating-point representation, adjusted to clock frequency [57, 58]. Table 5.3 lists the energy per 32-bit data access for different memory types used in the design [55, 51]. The control logic energy is calculated based on the number of load and store instructions issued in the simulation and each load or store energy is calculated using Flip-Flop switching energy [59].

Layer	Ifmap size	Padd	Stride	Number of filters	Filter size	Number of parameters	Ofmap size
CONV1	227*227*3	0	4	96	11*11*3	34944	55*55*96
CONV2	27*27*96	2	1	256	5*5*96	614656	27*27*256
CONV3	13*13*256	1	1	384	3*3*256	885120	13*13*384
CONV4	13*13*384	1	1	384	3*3*384	1327488	13*13*384
CONV5	13*13*384	1	1	256	3*3*384	884992	13*13*256
FC6	9216	-	-	-	-	37748736	4096
FC7	4096	-	-	-	-	16777216	4096
FC8	4096	-	-	-	-	4096000	1000

Table 5.1: AlexNet layers.

Precision (bit width)	Area (um^2)	Energy per operation 10^{-12} Joules
32-bit	4793.10	4.6
16-bit	1841.9	1.5
8-bit	336.672	0.23

Table 5.2: Floating-point MAC unit comparison.

Memory types	Read energy per 32 -bit access (10^{-9} Joules)	Write energy per 32-bit access (10^{-9} Joules)
Internal Memory	0.004	0.002
External Memory	0.4	0.2

Table 5.3: Memory energy.

5.1 Figures of Merit

The simulator takes in several parameters for simulating the given architecture. Covering the whole design space is nearly impossible. So the simulation is run for a few important parameters while keeping others constant. The Figures of Merit (FoM) for the various designs are evaluated using the following:

1. **Runtime Cycles**:
   The simulator outputs the total cycles required to execute the layer in consideration.

2. **Total Area**:
   The total area of the accelerator is the sum of the memory unit, control logic unit and the compute array.

3. **Total Energy**:
   The energy required to execute the given layer is the sum of the energy required to access memory, energy required for the computation and the control logic energy.

4. **Throughput**:
   This gives the compute performance of the given design, it indicates the effective operations per cycle.

$$\textit{Throughput} = \frac{\textit{Total number of operations}}{\textit{Total execution cycles}} \tag{5.1}$$

5. **Memory Utilization**:
   This is the external and internal memory bandwidth utilization for the design.

$$\textit{Memory Throughput} = \frac{\textit{Total data read/written}}{\textit{Total cycles required for reading/writing}} \tag{5.2}$$

6. **MAC Utilization**:
   This indicates the amount of time the MAC units of PE are active.

$$MAC\ Utilization = \frac{Total\ FLOP\ cycles}{Total\ execution\ cycles} \tag{5.3}$$

7. **Bisection Bandwidth**:
   The nodes in the network are divided into two sections, and the bandwidth available between these two partitions is called the Bisection Bandwidth. This metric gives an indication of the traffic between the partitions. Bisection is done in a way that the bandwidth is minimum between these two partitions.

8. **Energy-Delay Product (EDP)**:
   It is the product of total energy consumption and the runtime for given workload and design parameters. EDP takes into account the trade-off of increased delay for lower energy per operation.

5.2 Performance Analysis

Few parameters of the design as shown in the Table 5.4 are kept constant through all the simulations.

Parameters	Value
Operating frequency	1 GHz
External memory read latency	6 cycles
Internal memory read latency	1 cycles
External memory access width	32 bits
Internal Ofmap memory access width	32 bits
Floating point MAC unit latency	4 cycles
Flip-Flop switching energy	0.08 x 10^{-15} Joules/cycle

Table 5.4: Fixed simulation parameters.

5.2.1 Runtime Cycles

This section covers the variation of execution cycles with respect to different design parameters.

5.2.1.1 Runtime cycles vs Number of PEs

The size of the array is varied and corresponding runtime cycles are obtained. The Figure 5.1 shows the variation of total execution cycles for different number of PEs in the array while the memory size of each PE is fixed to 1KB and the system is set to 32-bits numerical precision.

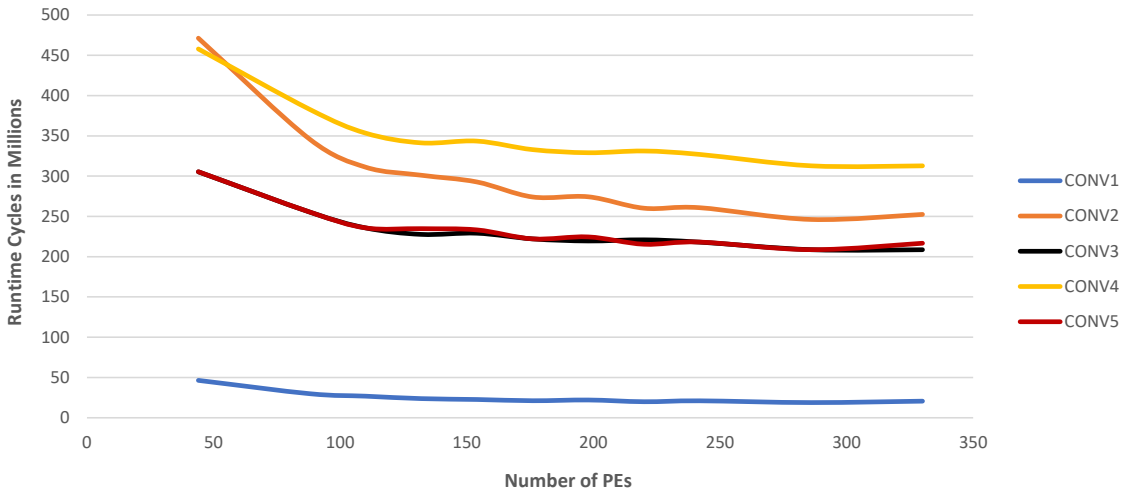


Figure 5.1: Variation of runtime with the number of PEs for different layers.

Significant reduction in runtime can be seen in *conv2* to *conv5* as there are more weights in these layers when compared to *conv1*, and there is more filter and ifmap reuse when there is an increase in the size of the array. There is a slight increment in runtime for certain number of PEs. This is due to the nature of systolic array, the whole array is modeled as a single unit without control over each column of the array. When the number of filters is not a multiple of the number of columns in the array, there is wastage of cycles performing shift operations in the array. This contributes to an increase in the runtime as seen in the Figure 5.1 and Figure 5.2.

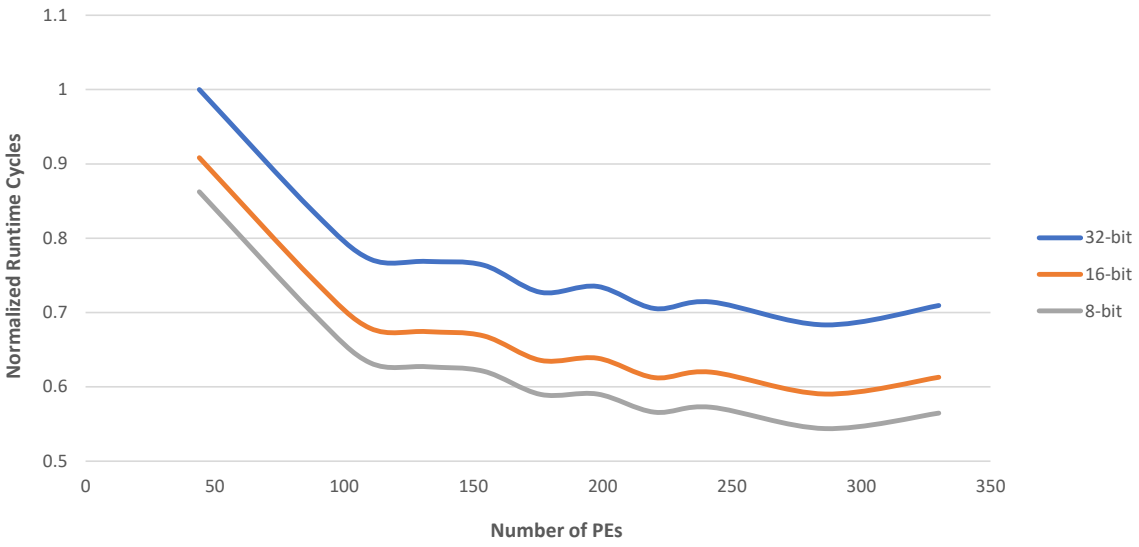


Figure 5.2: Variation of normalized runtime with number of PEs for *conv5* layer.

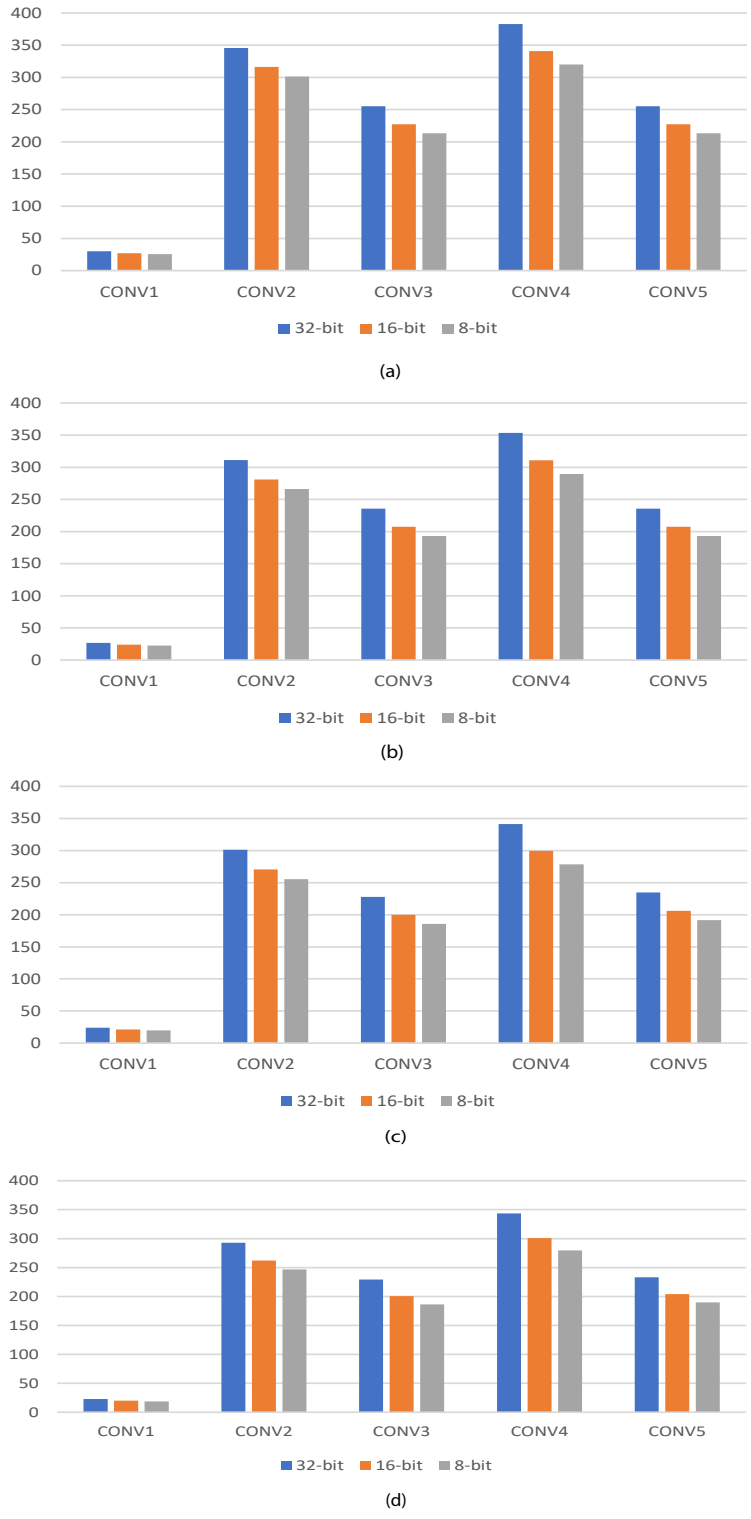


Figure 5.3: Runtime in million cycles for all 5 layers of AlexNet for different SA sizes, (a)11x8 = 88 PEs, (b)11x10 = 110 PEs, (c)11x12 = 132 PEs, (d)11x14 = 154 PEs. The Y-axis units are in million cycles and X-axis represents various workloads.

The Figure 5.2 shows the influence of number of PEs on the normalized runtime cycles. There is a more significant reduction in the runtime when moving from 32-bit to 16-bit numerical precision compared to moving from 16-bit to 8-bit. This reduction is attributed to the decrease in loading and scheduling cycles when moving from 32-bit to 16-bit or 8-bit designs. There is not much savings in scheduling cycles when moving from 16-bit to 8-bit design when compared to the savings when moving from 32-bit to 16-bit designs.
The Figure 5.3 shows the comparison of runtime for all layers for different Systolic Array (SA) sizes, and for different numerical precision. The 8-bit implementation has the least runtime among all the precision formats. The Figures 5.3(b) and 5.3(c) gives the highest drop in runtime time cycles and then there is small incremental reduction in runtime for 154 PEs and higher as shown in the Figures 5.3(d) and 5.2.

5.2.1.2 Runtime cycles vs PE memory size

The size of the internal memory in each PE is varied and corresponding execution cycles are obtained. The Figure 5.4 shows the variation of total execution cycles with respect to the memory size of PE in the array while the size of the array is fixed to 11x12 = 132 PEs.

The graph plateaus at around 1KB to 2KB of PE memory as it reaches the maximum

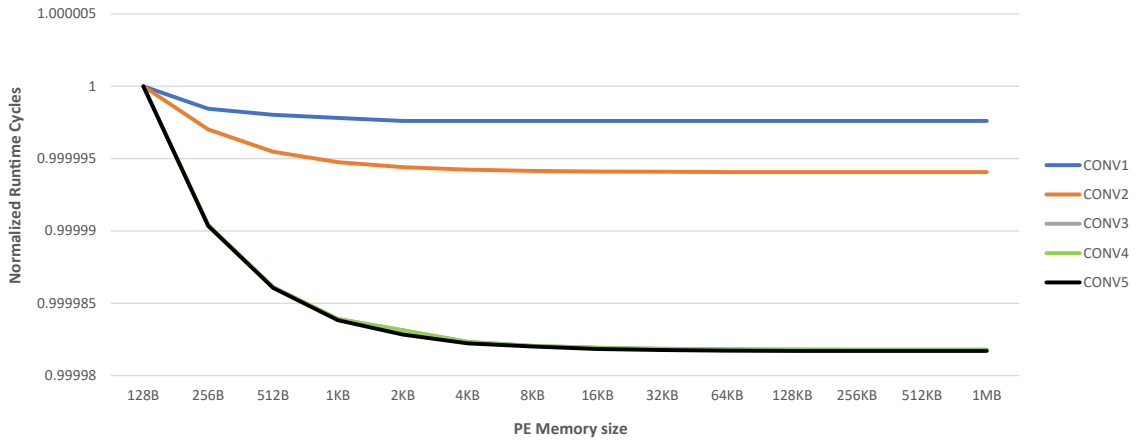


Figure 5.4: Normalized runtime cycles for different PE memory sizes.

reuse of stored filters and ifmaps for the given 132 PEs in the array. The reduction in runtime is more for *conv3*, *conv4* and *conv5* when compared to others as these layers have more weights. The Figure 5.5 shows the runtime of different layers of AlexNet when implemented using different numerical precision.

The highest drop in runtime cycles can be seen in the *conv4* layer when compared to other layers. The data is loaded from external memory and stored in the internal memory of PEs and then this data is reused to the maximum extent. For 32-bit implementation, a significant portion of the runtime is spent on loading the data to internal memory and it takes more cycles to load from external memory. When the numerical precision is reduced, so does the loading latency and this reduces the total runtime cycles. Hence we see a significant reduction in runtime when moving from 32-bit to 16-bit or 8-bit numerical precision.

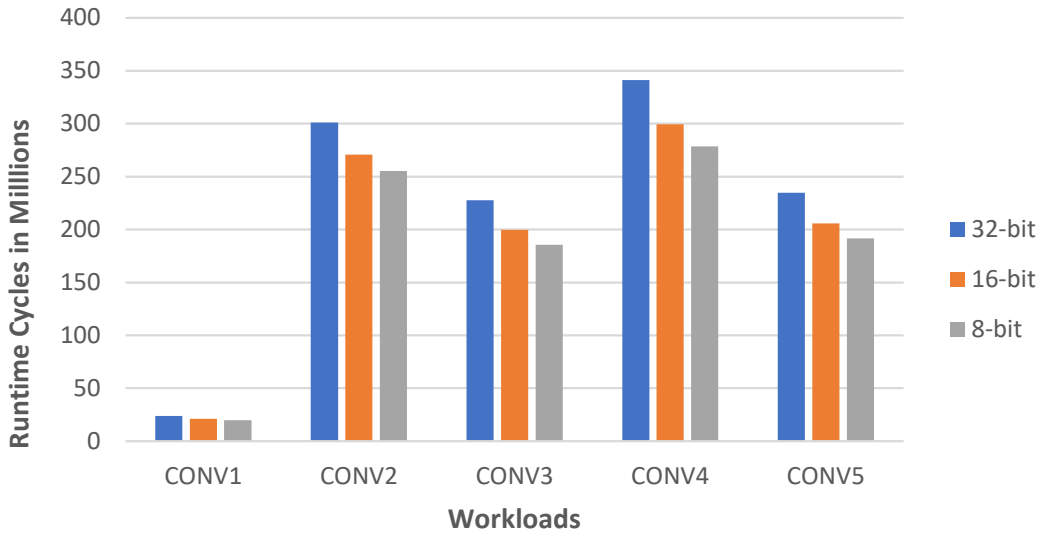


Figure 5.5: Runtime in million cycles for 11x12 = 132 PEs and PE memory of 1KB.

5.2.2 Total Energy

The energy consumption of a workload is an important performance metric. It gives the efficiency of implementation for the given workload on the given design. The Figure 5.6 shows the runtime energy for all the workloads run on 132 PE and 1KB internal memory configuration. The *CONV2* layer has the highest runtime energy due to the high number of ifmap accesses form external memory and the effective number of ifmap windows are higher for *CONV2* when compared to other layers. The *CONV1* layer has a stride of 4, hence this causes the effective number of ifmap accesses to be lesser than *CONV2*. There is a significant decrease in energy consumption across workloads when moved from 32-bit numerical precision to 16-bit or 8-bit numerical precision.

The energy per convolution layer can be divided into individual constituents of the design and these give insight into the energy distribution among various segments of the design. The total energy of the simulation run can be divided into:

1. *External memory access energy*:
   The *conv* layer loads the weights from external memory to internal memory and streams the ifmaps into the array. Total external memory access energy is obtained by multiplying the total external accesses with the energy per access. The external memory is modeled as *test bench* in this simulation and the energy per access is detailed in Table 5.3.

2. *Internal memory access energy*:
   The data stored in the internal memory is reused until it is not needed for further calculation. This data is accessed numerous times. Total internal memory access energy is obtained by multiplying the total internal memory accesses with the internal energy

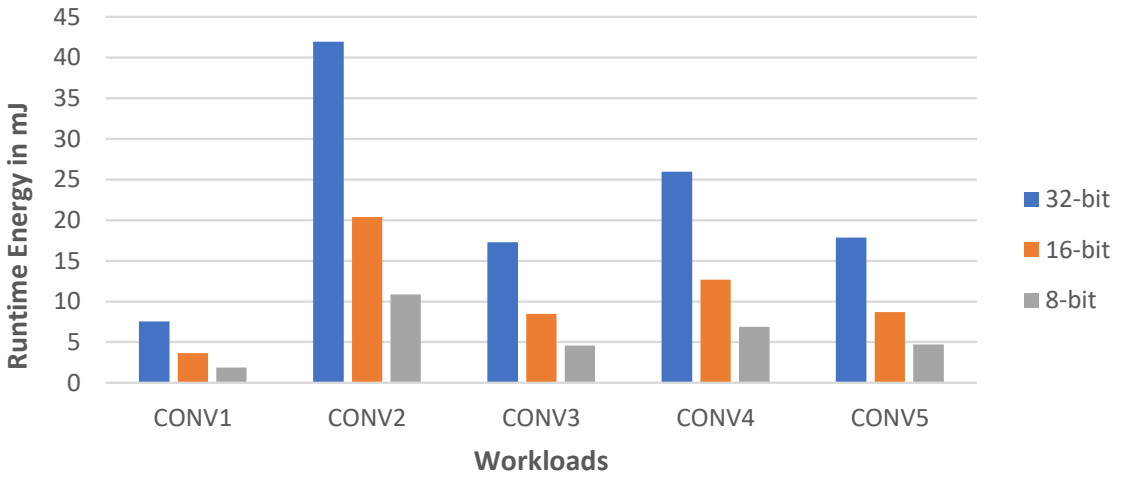


Figure 5.6: Energy consumption for different workloads and numerical precision.

per access. The size of the internal memory is a user input as explained in the Section 4.4.2.1 and the energy values are detailed in Table 5.3.

3. *MAC operational energy*:
   The energy for one MAC operation is multiplied with total floating-point operations to get the total MAC operational energy. The details of energy per operation are detailed in the Table 5.2.

4. *Control energy*:
   Each load and store operation incurs control energy as the control logic issues these operations. The Flip-Flop switching energy per cycle is listed in the Table 5.4. The total control energy is a function of total *for-loops* executed and total instruction issued.

5.2.2.1 Total Energy vs Number of PEs

The simulation is run for various SA size configurations by varying the number of PEs while keeping the internal memory size constant to 1KB. The normalized total energy variation with respect to the number of PEs for different workloads is shown in the Figure 5.7.

For a smaller number of PEs, the energy consumption is higher as the total energy is dominated by the *External Memory Energy* due to less weight and ifmap reuse across PEs. There is a need for frequent data loading from the external memory. As we increase the number of PEs there is a drop in energy consumption because the ifmaps and weights are reused more. The increase in *MAC Operational Energy* is less than the decrease in the *External Memory Energy*, hence we see a minimum in the total energy consumption as seen in the Figure 5.7. For number of PEs greater than 150, the total energy increases, because there is an increase in the total number of floating-point operations, this causes an increase in *MAC Operational Energy*, thus increasing the total energy. The decrease in the *External*

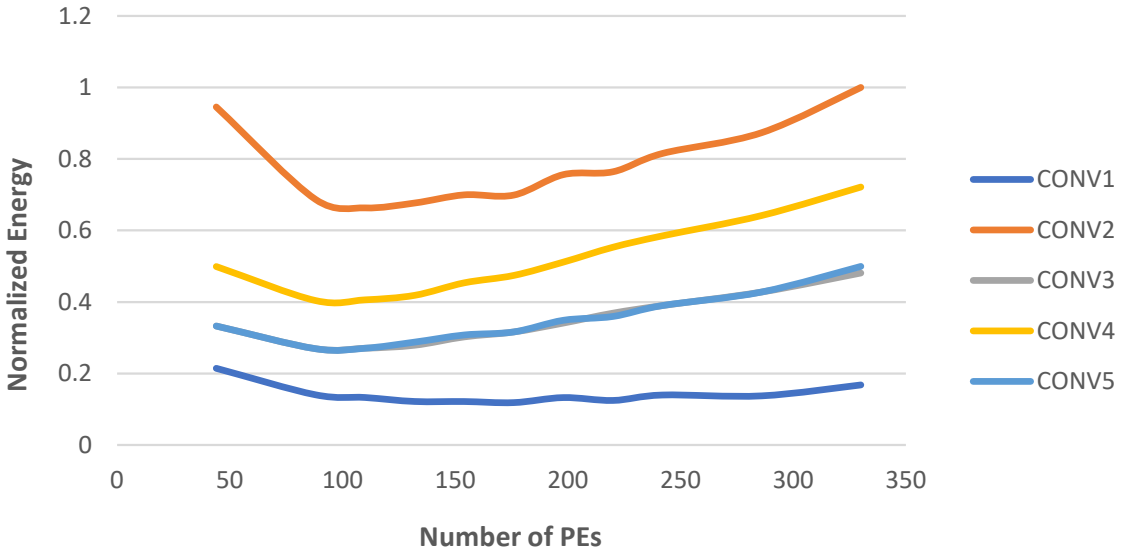


Figure 5.7: Variation of the normalized total energy with the number of PEs.

Memory Energy is less than the increase in the *MAC Operational Energy* and *Internal Memory Energy*. The local minimum is greatly visible for all layers except *conv1* as the values are normalized by the maximum energy value, which is from *conv2* layer.

The Figure 5.8 shows the distribution of total energy among its components for different number of PEs. On comparison of 5.8(c) and 5.8(d), we can observe an increase in the *MAC Operational Energy* and decrease in the *External Memory Energy*. The 110 PEs configuration has the lowest energy consumption when compared to others. This can be observed across different workloads. The *Control Energy* is very low when compared to other component energies.

The increase in the *MAC Operational Energy* is because of the inherent operational functionality of the SA. All the PEs in the SA operate in synchronization, thus it also performs zero operation while the data is shifted down the column. The simulator does not control every PE in the SA, it models the whole SA as a single module without finer grain controls over individual PE. Hence, until the data from the last PE of the last column is shifted out, all the other PEs perform zero operation. There is an increase in the total MAC flops if there is a very high number of PEs in the SA. The Figure 5.9 shows the variation of different component energy with respect to the number of PEs for *conv5* layer, and the decrease of *External Memory Energy* and increase of *MAC Operational Energy* and *Internal Memory Energy* can be observed.

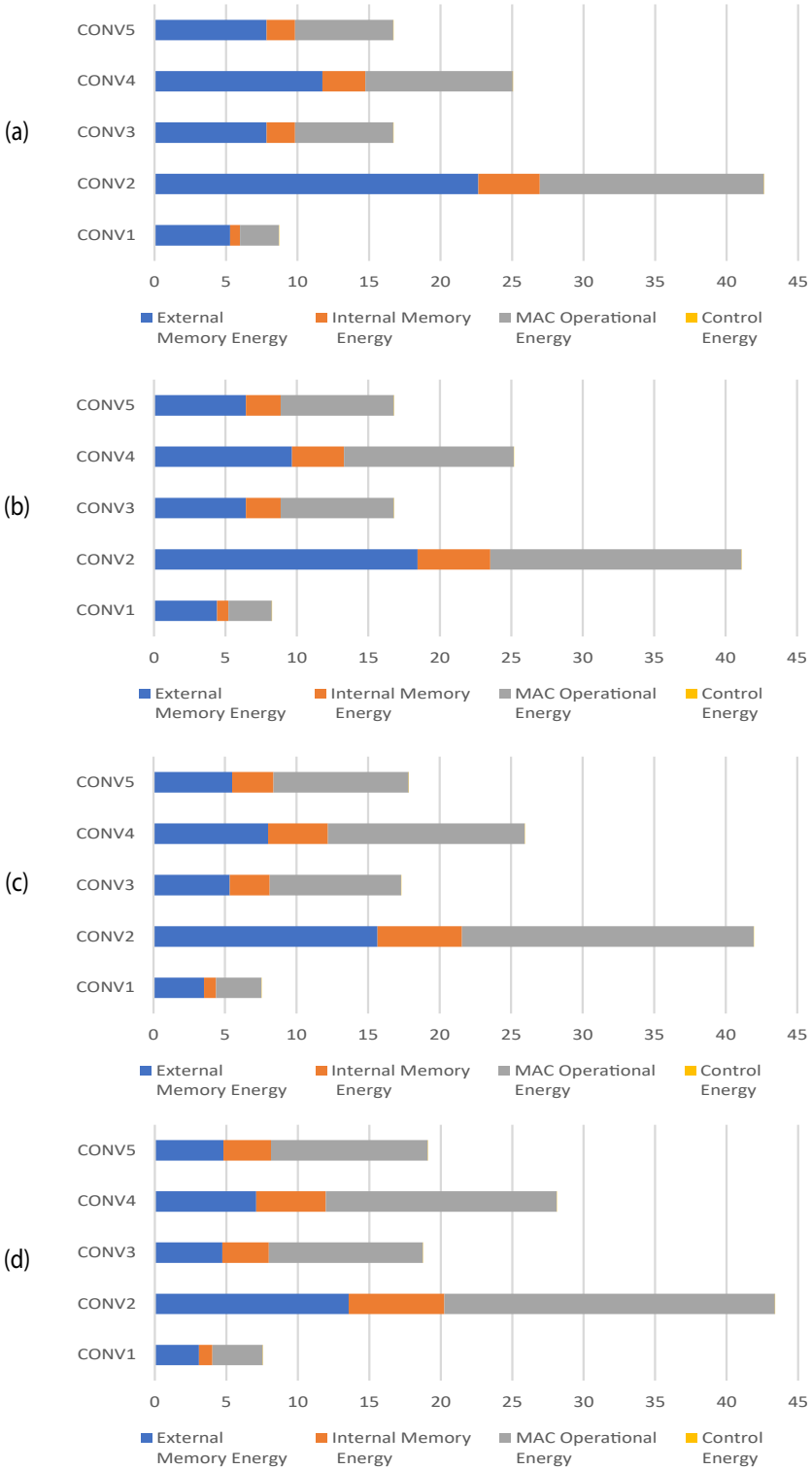


Figure 5.8: Energy distribution for different PE configurations, (a)11x8 = 88 PEs, (b)11x10 = 110 PEs, (c)11x12 = 132 PEs, (d)11x14 = 154 PEs. The X-axis units are in milli Joules cycles and Y-axis represents various workloads.

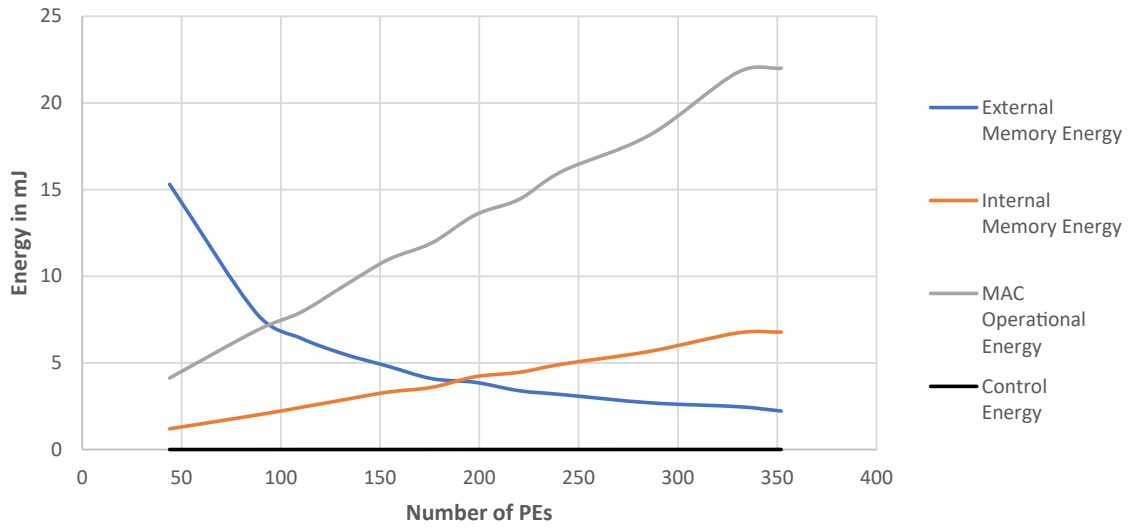


Figure 5.9: Variation of *CONV5* layer energy distribution with the number of PEs.

5.2.2.2 Energy vs Runtime

Understanding energy consumption helps to figure out the optimal design for the given workloads. On comparing the runtime with the corresponding energy consumption, one can see the point of inflection, where, on further decrements in the latency will not result in a great improvement in energy efficiency. The Figure 5.10 plots the points which correspond to total energy and latency combination for different convolution layer workloads and different numerical precision. The simulation is run for 1KB internal memory and 132 PEs configuration. We can observe a gradual decrease in the energy as the runtime decreases, but there is an inflection point from where the energy increases again, this point corresponds to the situation where the increment in *MAC Operational Energy* and *Internal Memory Energy* is greater than the decrement in *External Memory Energy*. All the workloads show similar pattern for different numerical precision as seen in the Figure 5.10(a) to Figure 5.10(e).

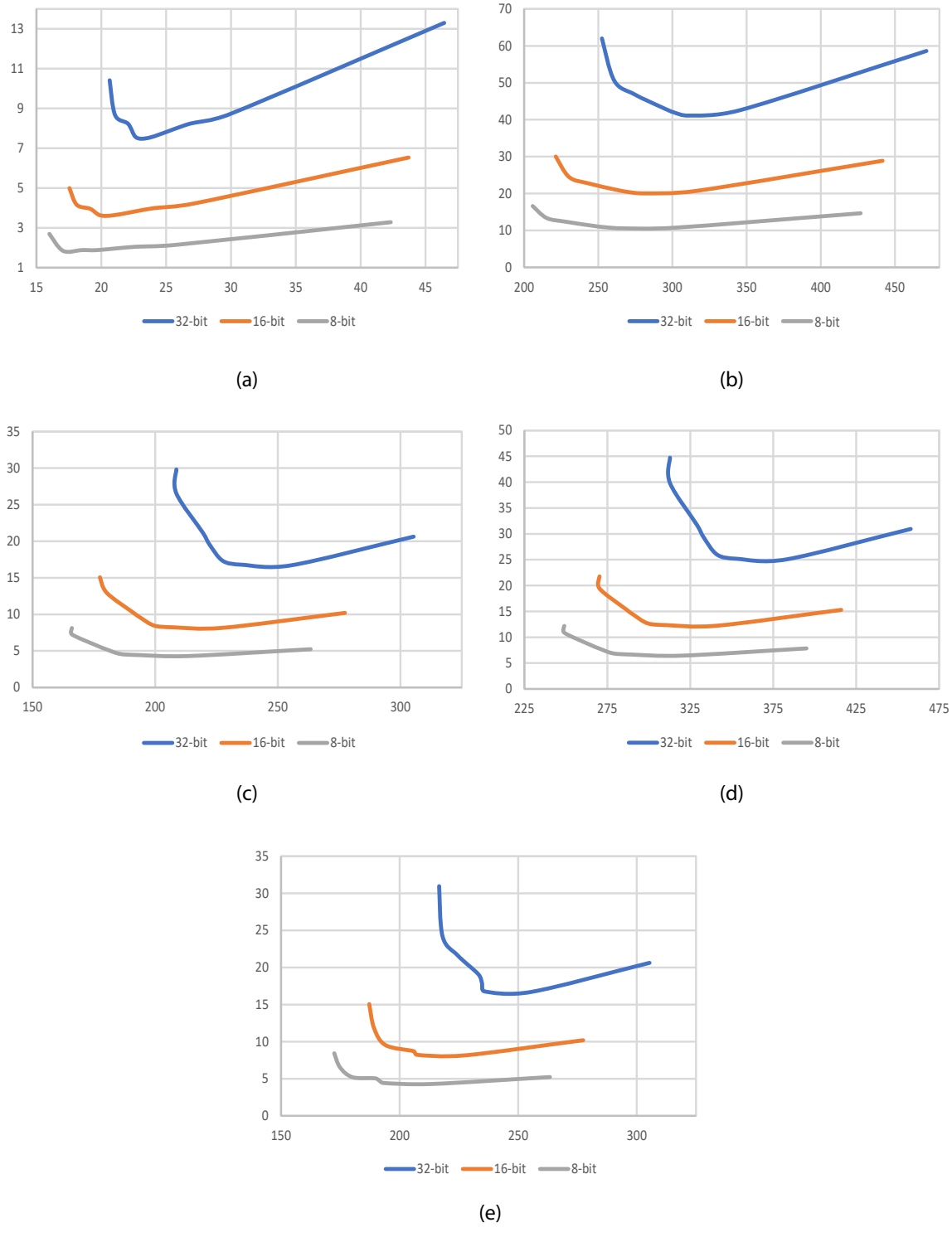


Figure 5.10: Variation of the total energy for different numerical precision and workloads, (a)*CONV1*, (b)*CONV2*, (c)*CONV3*, (d)*CONV4*, (e)*CONV5*. The X-axis units are in Million cycles and Y-axis units are in milli Joules.

5.2.2.3 Energy vs PE Memory Size

As the PE memory size increases, so does the total internal memory access energy. As the memory size increases the leakage power also increases. The leakage power of each memory configuration is obtained from *DESTINY* Tool[55]. This value is multiplied with the runtime to obtain the effective leakage energy during the runtime and then added to the dynamic energy during runtime to obtain total energy. The Figure 5.11 shows the variation of total energy with the PE memory size. There is a gradual increase in the total energy till 32KB and then there is a drop in the energy. This can be attributed to the low leakage power for 64KB, 128KB, and 256KB. The *DESTINY* tool optimizes the solutions for *Energy Delay Product* (EDP) and this results in less leakage power for 64KB, 128KB and 256KB memory solution. The tool searches for optimal EPD solution, hence there is an irregularity in the graph.

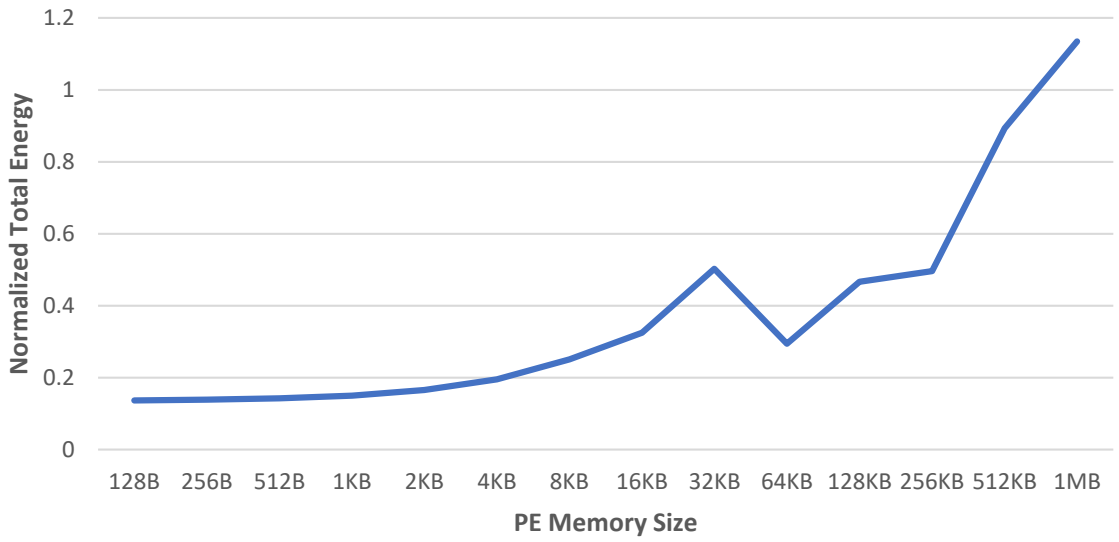


Figure 5.11: Variation of the total area of the SA with PE internal memory size.

5.2.3 Total Area

The total area is the sum of the number of PEs used in the design. The Figure 5.12 shows the variation of the area, the total area scales linearly with the number of PEs. For a higher number of PEs, there is a significant area savings when moved form 32-bit numerical precision to 16-bit or 8-bit precision. For a lower number of PEs, there are fewer area savings when moving from 32-bit to 16-bit or 8-bit implementation.

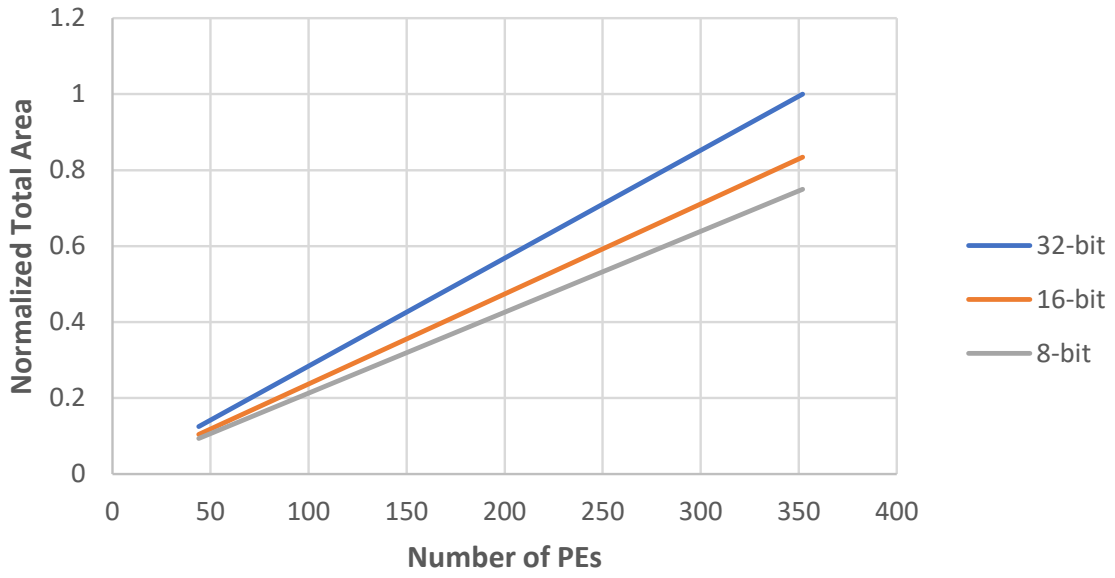


Figure 5.12: Variation of the total area of the SA with the number of PEs.

5.2.4 MAC Utilization

A high MAC utilization indicates that a large amount of time is spent on MAC operations and the rest is spent on data movement, and low MAC utilization indicates otherwise. The MAC utilization is given in the Table 5.5. The high MAC utilization is due to the nature of SA, all the PEs are active until the required data is shifted out from the last column of the array. There is a large number of zero operations since the whole SA is modeled as a single module and no finer low-level column control is modeled. The minimum and maximum utilization correspond to the *conv1* layer as it has the least number of parameters and ifmaps, and hence less time is spent on moving data from external memory, hence high utilization. For a higher number of PEs more time is spent on zero loadings into the PE memory, and hence less MAC utilization. This behavior is similar in other layers. The average utilization is 75% across layers.

MAC Utilization	Value (%)
Minimum	67.33
Average	75.54
Maximum	88.17

Table 5.5: MAC Utilization.

5.2.5 Bisection Bandwidth

The Bisection Bandwidth is dependent on the latency of PE and as well as the number of PEs in the SA. The Table 5.6 lists the bisection bandwidth for different layers. These calculations are based on the 132 PE and 1KB PE memory configuration. The *conv1* layer has a filter size of 11x11, hence the bandwidth of a single link after bisection is 1GB/s. This is multiplied with all the bisected links to obtain 11GB/s. The *conv5* layer has a filter size of 5x5, as only the first 5 rows of the SA is enabled during operation and the rest 6 is disabled, thus the effective bisection bandwidth is 5GB/s. The procedure is repeated for the rest of the layers which have a filter size of 3x3 to obtain 3GB/s bisection bandwidth.

Layer	**Bisection Bandwidth (GB/sec)**
CONV1	11
CONV2	5
CONV3	3
CONV4	3
CONV5	3

Table 5.6: Bisection bandwidth for different layers.

5.2.6 Throughput

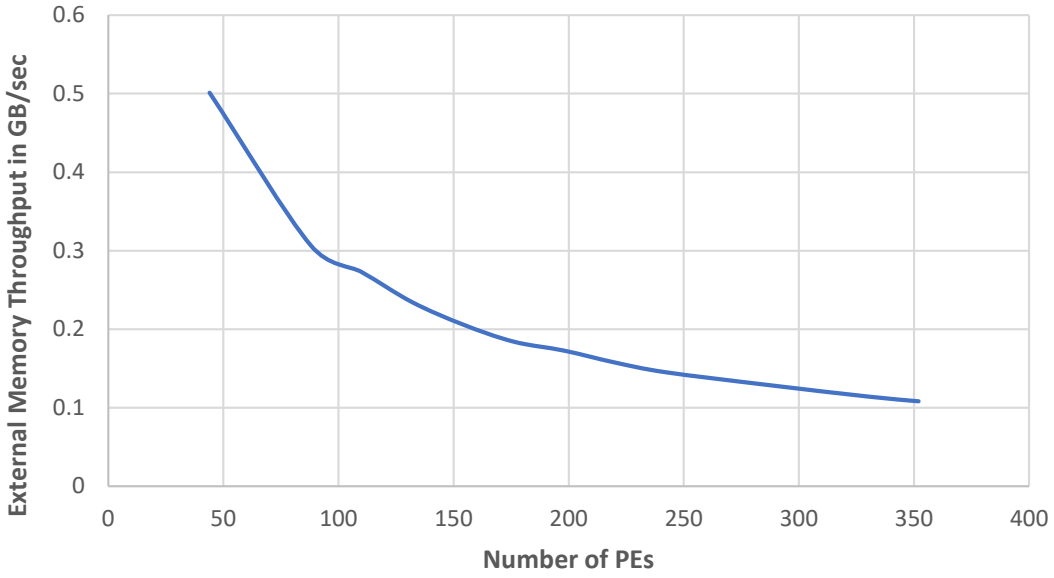
The total MAC operations can be divided into actual convolution operation (non-zero operations) and zero operations. The zero operations are inherent in the SA as the SA is modeled as a single module. While the data from the last column of the SA is shifted out, the rest of the columns perform MAC operation with zero values. Both of these types along with the effective throughput is listed in the Table 5.7. The throughput is listed for 3 PE configurations: 11x10, 11x12 and 11x14, as these show the lowest energy consumption and these correspond to the inflection points as discussed in the Section 5.2.2.2.

Number of PEs	Throughput (**GFLOPS/sec**)	
Size of SA	*Non zero ops*	*Zero ops*
110	4.97	31.02
132	5.77	37.71
154	6.16	44.48

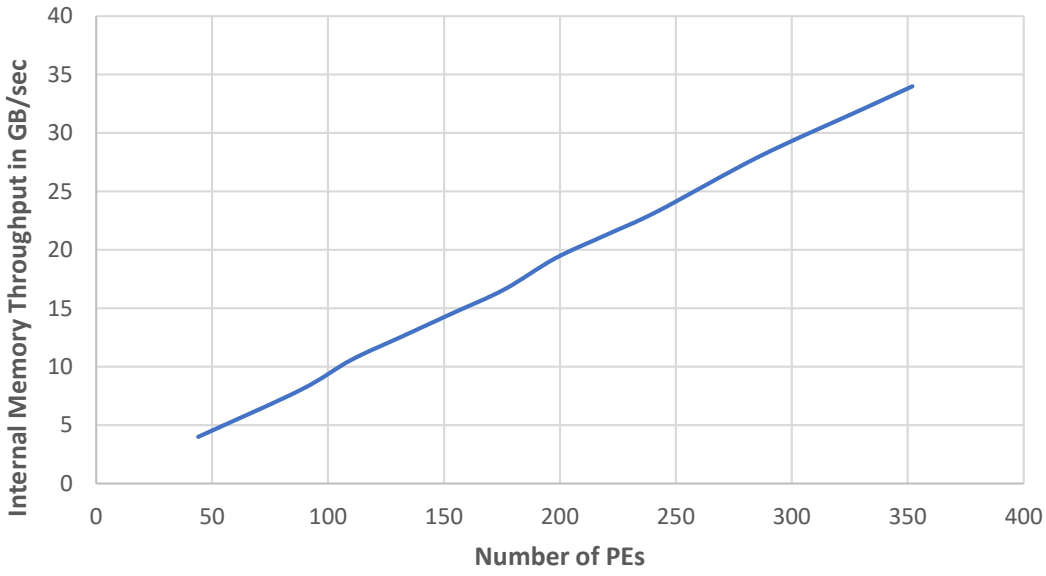
Table 5.7: Throughput for different PE configurations.

5.2.7 Memory Throughput

The memory throughput gives information about the utilization of the memory interconnect. The external and internal memory bus width is fixed to 32-bit.



(a)



(b)

Figure 5.13: Variation of the memory throughput with the number of PEs.

As the number of PEs increases, the total number of external memory accesses decreases due to the reuse of the loaded data. There is a near exponential decrease in the external memory throughput as the number of PEs approaches 125, then there is a plateau due to the law of diminishing returns, as seen in Figure 5.13(a). Whereas, when the number of PEs increases, so does the total internal memory accesses. The total internal memory accesses increment is due to an increase in data reuse which is stored in the local memory. This increment in throughput as the number of PEs increases can be seen in the Figure 5.13(b).

5.2.8 Energy Delay Product

To obtain the EDP for different design configurations, the PE memory size was fixed to 1KB and the number of PEs was varied to obtain the graph as shown in the Figure 5.14. We see a high EDP for a smaller number of PEs as there is significant energy spent on loading data from external memory. As the number of PEs increases the EDP drops to the lowest, this point corresponds to the point of inflection as explained in the Section 5.2.2.2. As we further increase the number of PEs the EDP gradually increases due to the increase in the *MAC Operational Energy* section of total energy. The increase in total energy offsets the advantages provided by the decrease in the runtime. This behavior is observed for other workloads as seen in the Figure 5.14.

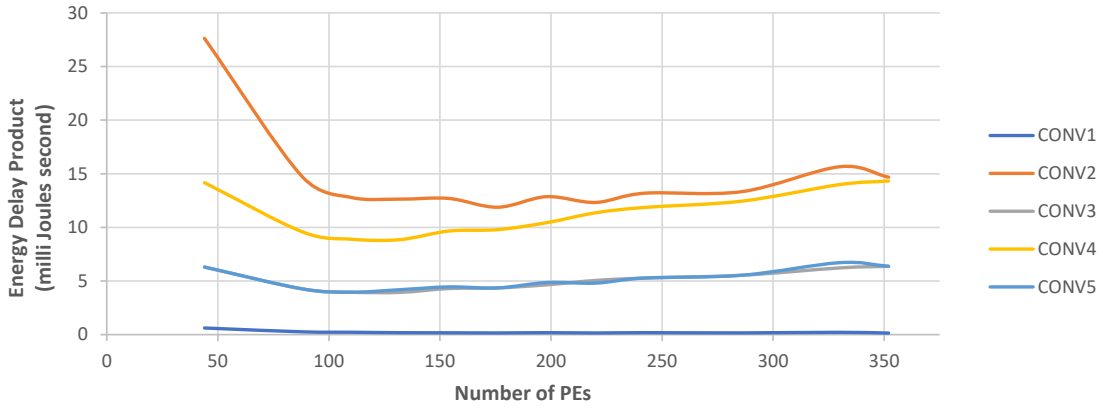


Figure 5.14: Variation of EDP with the number of PEs.

In this chapter the AlexNet workloads, FoM for performance evaluation of the design points were covered. Subsequently, total runtime, area, and energy evaluation for different numerical precision and various design points were explained. The implemented SA advantages and drawbacks were detailed, and then an optimal architecture was proposed based on the design space exploration. In Chapter 6, the conclusion and future work is covered.

6 Conclusion

"Somewhere, something incredible is waiting to be known."

— Carl Sagan

This chapter discusses the conclusion of the work done in this thesis, and presents the ideas and improvements that could be further explored to build on the current work. It ends with the publication relating to this thesis.

6.1 Summary

Deep Neural Networks (DNN) have been used in many applications, ranging from real-time image classification, object detection and speech processing to robot motion planning. Among the various DNN approaches, Convolution Neural Networks (CNN) is the most commonly used NN. Usually, the DNN implementations are done on a high-performance temporal architectural system like multi-core CPU and GPU, these are of high power in nature and cannot be deployed in edge devices. The inference is performed on the edge to provide low power and real-time performance. Hence, the edge devices require a novel architectures towards processing CNNs. The recent works towards this goal have been the development of DNN accelerators using data-flow spatial architectures. The systolic array is the most common spatial architecture used, and it is composed of 1-D or 2-D array of Processing Engines (PE). A typical PE has a MAC unit and a small local memory. The data in the systolic array flows rhythmically between PEs and the communication with the external memory happens at the boundary of the array. Few data-flow approaches like weight stationary and output stationary have been proposed for reusing weights and partial products respectively. The row-column stationary data-flow approach maximizes the reuse of weights, input and output feature maps across the array. Depending on the application, the numerical precision of the DNN can be varied to decrease the energy consumption and area of the design. Different applications require different performance, area and energy needs, and this makes it imperative to quickly prototype the architectural ideas, and perform design space exploration. The challenging part is the non-trivial interactions between different architectural design parameters, as they play an important part in the complex design decisions. Hence, a *Systolic Array Simulator for CNN* (SASCNN) has been proposed in this thesis. The systolic array uses row-column stationary data-flow with a near memory computing approach to accelerate individual layers of CNN. The simulator supports different numerical precision such as 16-bit and 8-bit floating-point. The simulator can take in numerous design parameters such as the *size of the systolic array*, *latency of MAC operation*, *PE local memory size*, *PE local memory*

latency and *external memory latency*. The simulator can support future memory technologies by varying the latency of PE internal memory, and supports a multi-stage MAC engine by varying the latency of floating-point operation. Since the external memory latency is not fixed, the simulator supports different external memory latency by taking it as an input parameter. The simulator supports *conv* layer for different filter sizes and, strides and padding for input feature maps. In this thesis, we consider eDRAM memory for PE internal memory, and it is modeled by using the *DESTINY* tool. The functionality of the systolic array convolution is verified on the AlexNet model. The Figures of Merit is evaluated for array size and internal memory size design parameters. For design space exploration, at first, the PE internal memory size is fixed and the size of the array is varied to obtain the performance metrics. Next, the size of the array is fixed and the PE internal memory size is varied to obtain the performance metrics. An energy and area estimation model is proposed and few other performance metrics like throughput, bisection bandwidth, energy-delay product, memory utilization, and MAC utilization are calculated for each design point. This procedure is repeated for 16-bit and 8-bit operand precision, and the results are analyzed. The variance of the PE internal memory has less impact on the runtime cycles of all workloads. The size of the array has a significant impact on performance and energy metrics. While the size of the PE internal memory is fixed to 1KB, the number of PEs is varied from 88 to 352. When the size of the array is small, the runtime is high and it exponentially drops when the size is gradually increased. The performance plateaus at 154 PEs, as there is no significant gain in runtime reduction on a further increase of array size. On analysis of energy consumption, the energy gradually decreases as the size of the array is increased and reaches a minimum point at 132 PEs. This is the inflection point, and on further increase in the size of the array, we see an increase in energy consumption. After this inflection point, the increase in MAC operational energy is greater than the decrement in the external memory energy. The energy-delay product is also a minimum for this point. The average MAC utilization is 75.54% and the bisection bandwidth scales proportionally with the size of the filters. The throughput of the optimal design point is 37.71 GFLOP/s. From the result, we can see that the performance and energy of the system depend critically on the size of the array and not much on the size of the PE internal memory for the *conv* workloads. This work provides a fast design exploration tool for designers to test different workloads on the proposed architecture.

6.2 Future Work

Numerous ideas can be integrated with this implementation to enhance the work done in this thesis. Due to time constraints some of the ideas could not be implemented and some other ideas had complexity and scope which was beyond the author's current knowledge. Few ideas which could be explored to augment this work are:

- The systolic array can be made run-time re-configurable to perform parallel convolutions in any column. This will utilize the PEs which are disabled when the filter size is smaller than the number of rows of PE.
- The *fully connected* layer and *max-pooling* layer of the CNN could be added to this simulator to get an overview of the complete CNN model. The results can be compared

with state of the art ASIC implementations.

- Include a zero skipping method to avoid zero loadings into the internal memory. And implement column control mechanism to skip zero MAC operations in-order to increase power efficiency.
- Implement an FPGA implementation of this design as a proof of concept and finally develop an ASIC.

6.3 Publication

The contributions of this thesis work are described in:

- Shashanka Marigi Rajanarayana, Sumeet Kumar, Amir Zjajo, René van Leuken, "**Towards Computationally-Efficient Cognitive Sensor Systems for Autonomous Vehicles**", IEEE International Conference on Cognitive Informatics and Cognitive Computing, 2019, in press.

Towards Computationally-Efficient Cognitive Sensor Systems for Autonomous Vehicles

Shashanka Marigi Rajanarayana, Sumeet Kumar, Amir Zjajo, René van Leuken
Circuits and Systems Group
Delft University of Technology, Delft, Netherlands
{S.MarigiRajanarayana,s.s.kumar, a.zjajo, t.g.r.m.vanleuken}@tudelft.nl

***Abstract*—Advanced driving assistance systems (ADAS) prepave regulators, consumers and corporations for the medium-term reality of autonomous driving with adaptive cruise control, collision avoidance and lane departure warning system. Various sensors like camera, RADAR and LIDAR, integrated into the vehicle assist driving. In addition, deep learning approaches are utilized in a wide range of applications ranging from object detection and scene segmentation to engine fault diagnosis and emission management to detect vehicle network intrusion. In this paper, we scope out the state of the art sensors subsystems in terms of its functionality, characteristics, specifications and communication protocol, and we describe cognitive deep learning based algorithms required for environment perception through these sensors. Subsequently, we analyze the cognitive algorithm by profiling the standard deep learning models, explore different compute platforms and possible algorithm and hardware optimization scenarios.**

***Index Terms*—Vision, radar, lidar, deep learning, convolution neural networks, spatial architectures, ASIC, GPU.**

I. Introduction

The autonomous vehicles decide the next step to take by performing inference of the input data on a trained machine/deep learning models. These algorithms require local processing since the latency and security risks of relying on the cloud are too high. Many of the embedded platforms that perform inference have stringent energy consumption, compute and memory cost limitations; efficient deep learning algorithms have thus become of prime importance under these constraints. In autonomous vehicles, the cognitive system involves a pipeline of processes consisting of semantic segmentation and object detection. In traditional machine learning approach, firstly feature extraction and selection is performed. This feature extraction includes manual labor, it is computationally expensive and prone to error, and it is done for every environmental condition and driving scenario. This process does not provide human level precision in object classification [1], localization, and segmentation tasks. Although, this process does not require large amount of training data as the features are hand engineered, it does not generalize to unseen scenarios.

In deep learning concept, the network extracts the features automatically, and classifies the data accordingly. Even though the network takes a significant amount of data to train, it can adapt to the various environmental condition like snow, rain, fog, etc with large data set of each environmental conditions. The data set size also depends on the network size and optimization methods used. The design is simple, as manual feature extraction is not required; and it is faster since, to classify an object only forward pass is required. The sensors, which perceive the environment feeds data to deep learning algorithms. Typically, they include vision, radar, and lidar, totaling up to a minimum of 12 sensors [2]. This results in effectively large data input to the compute systems and high load on the communication bus in the vehicle. The work by [3]–[5] provides an overview of the sensor technologies, localization, planning, control and coordination for autonomous vehicles.

The compute system is crucial for vehicle control, reaction time for real-time performance and safety. The systems requires high performance (with an order of trillions of operations per second), as it is a centralized system in the vehicle. In addition, the system should be low power, to extend the battery life and to increase the driving range of the electric vehicle. The deep learning algorithms like object detection and scene segmentation should be real-time, i.e. at 30 fps, which in-turn requires large communication bandwidth and low latency in the compute system. In addition, the system should be fault tolerant, robust to sensor failure, output failures, communication failure, etc, as the autonomous navigation is safety critical application. Moreover, the system should provide the same performance and safety in the event of failure of critical subsystems. The compute system should be able to support a large number of sensors, as numerous sensors are required to perceive the environment, which in turn provides a high data volume to the system. Similarly, the compute system should operate in various environmental conditions like snow, rain, high temperature, etc to provide safety and real-time performance in every environmental condition.

In this paper,

- We present a comprehensive study of cognitive sensor requirements and characteristics, and offer a comparison of deep learning algorithm for different sensor data.
- We profile a standard convolution neural network (CNN) to obtain the compute, memory and energy requirements.
- We analyze the system architecture for processing deep learning algorithms and propose computationally efficient optimizations in algorithm and hardware space.

The paper is organized as follows: Section II describes

This research was supported in part by the European Union and the Dutch government, as part of the ECSEL JU program under PRYSTINE project.

the latest sensors used in autonomous vehicle, its relevant properties, constraints, and industry vendors. Subsequently it describes the state of the art deep learning application for each sensor data and its analysis. Section III covers the benchmarks metrics of standard CNN models and describes the performance, power and memory requirements for inference systems. Section IV covers state of the art implementations, possible optimization methods of algorithms and compute hardware for inference. Finally, Section V provides a summary and main conclusions.

II. Sensor and Deep learning

Numerous types of sensors i.e infrared, ultrasonic, stereo vision, thermal vision, can be used in autonomous systems. The scope of this section is limited to radar, vision and lidar as they include some overlapping features of the above mentioned sensors. Environment perception by applying cognitive algorithms on sensor data augment object detection and scene understanding. More specifically, CNN [6] has achieved a high level of accuracy in object detection and semantic segmentation, which are fundamentally important in autonomous navigation.

A. Visual perception systems

In comparison to analog cameras, digital cameras offer higher signal to noise ratio, increased resolution, and small form factor. The 2D and 3D cameras require high dynamic range (>135dB) to provide a clear image even under extreme conditions like extreme sunlight, rain, smog, low sunlight, etc.

The Table I lists the specifications and range of the vision system; the resolution needs to be greater than 2 Mpixels so that the objects in the image can be recognized at a minimum distance of 100 m in an urban driving scenario. The frame rate should be greater than 30 fps for real-time performance. A 24-bit RGB channel system with 30 fps and 8 Mpixels resolution, has data rate of **685.8 MB/s**. This data should be processed at line rate at all temperatures and environmental conditions. The communication between the digital CMOS image sensor and the controller is serial in nature in raw capture mode.

Object detection: In the region based CNN (R-CNN) [7] approach thousands of region proposals (region of interest) per image are generated, which are applied to CNN in addition to bounding box regression to obtain the location of the object. The fast R-CNN [8] provides the region proposal by an external system, which uses selective search, while the classifier and bounding box regression is trained simultaneously. In [9] (faster R-CNN), bounding box proposal method for localization is augmented with a region proposal network. In [10], a new method is proposed to detect lanes and objects on road. It builds upon the Overfeat architecture [11] and uses the mask detector in [12] to improve the design. This method uses a sliding window detector to obtain the object mask and performs bounding box regression. In [13], a new approach for object detection using a single neural network in a single evaluation based on GoogLeNet [15] is proposed. The YOLO architecture, unlike the previous methods, which use sliding window or region of interest (ROI) pooling for localization, utilizes the fully connected layer to locate the object. Deformable Parts model (DPM) has distinct pipelines to extract features, classify regions and predict bounding boxes [14]. The Table II [13], shows the mean average precision (mAP) and frame rate of the video. In [16], a method is presented, which detects small objects by combining multilayer feature maps and uses a sub-pixel sliding window. The method is a derivative of anchor generation from faster R-CNN.

TABLE I
Vision sensor specifications

Specifications	Range
Resolution	2Mp-8Mp
High Dynamic Range	>135dB
Frame Rate	>30fps
Field of View	100°
No. of channels	3(RGB)
Pedestrian recognition	>100m

TABLE II
Performance on visual object classes

Model	Train data	mean AP	Frames Per Second (fps)
100Hz DPM [14]	2007	16	100
30Hz DPM [14]	2007	26.1	30
Fast YOLO	2007+2012	52.7	155
YOLO	2007+2012	63.4	45
Fast R-CNN [8]	2007+2012	70	0.5
Faster R-CNN VGG-16 [9]	2007+2012	73.3	7
YOLO VGG-16	2007+2012	62.1	18
Faster R-CNN ZF [9]	2007+2012	66.4	21

Semantic segmentation: Few methods have been proposed like incorporating the spatial information of the patch into CNN in order to enable learning of spatial priors [17]. In [18], a fully convolution network (FCN) is proposed, which can take arbitrary size input images to produce segmentation result. The method in [19] - [20] is a method, which integrates CNN with deep de-convolution neural network. In [19] a multipath approach is taken to parse a scene. The method in [20] introduces a new mapping between classes and filters at the deconvolution side of the design, in addition to filters at the expansion side of the network.

B. Radar perception systems

Radar system gives information about the distance, velocity, and angle of the moving object, not the shape and size. The speed of the object is calculated based on the Doppler effect. Radar is highly insensitive to environmental conditions like rain, wind, darkness, and fog. Multi-sensor networks composed of 4 or more short, mid, and long-range sensors are currently used in ADAS [21]. The short-range radars (SRR) are employed in blind spot detection and lane change and park assistance, while medium range radars (MRR) and long-range radars (LRR) are used in emergency braking and adaptive cruise control. The field of view (FOV) is the highest for SRR to detect close by objects and lowest for LRR. Radar

data has less data density when compared to optical sensors since the data is collected only if there is a moving object in the surroundings. The radar receiver lanes collect the reflected signals and send them to the processing unit serially. This raw data is processed to obtain the Doppler velocity, radial distance of the object, radar cross section (RCS), the azimuth angle and coordinates of the object with respect to the vehicle. This data is used to create an occupancy grid of the surrounding objects if needed for further processing. The Table III lists several of the technical specifications of the existing radar systems.
The AWR1243 Single-Chip 77-GHz and 79-GHz FMCW Transceiver from Texas Instruments has 4 receiver lanes, where each lane has a scalable data rate from 150 Mbps to 900 Mbps. It has a maximum sampling rate of **37.5 MS/s** with SPI (serial peripheral interface), CSI2 and MIPI (Mobile Industry Processor Interface). The BGT24MTR11 with XMC4500 by Infinion allows a maximum sampling frequency of **2.85 MS/s** with 8-bit resolution.

Static object recognition: The method in [22] uses a neural network for classification of static (stationary) objects like parked vehicles in radar grids. The radar data was accumulated by using the occupancy grid method [23]. If there is a 75% match between the area of the extracted object and area of the label, the label is assigned. If multiple objects are present, the largest possible object label is kept [22]. The method in [24] uses two independent classifiers: a random forest classifier, which takes extracted domain specific features as input, and a CNN based classifier. The random forest consists of decision trees, which are formed during training and has a limited number of parameters when compared to other classification methods. Subsequently, these two classifiers are combined to obtain increased classification accuracy. Feature-based classifiers need less training data when compared to CNN. The radar data is collected and accumulated into two grids i.e *occupancy grid* [22] and *amplitude max grid*. The features are extracted from *occupancy grid* and *amplitude grid* for random forest classifier. The CNN is based on GoogLeNet [15]. The ensemble classifier performs better in detecting cars and provides better results when compared to that of each of the individual classifier. The random forest classifier in the ensemble classifier offers a higher accuracy due to its hand engineered features.

Semantic segmentation: The method in [25] assigns a class label to every point cloud. The order of points in the point cloud is irrelevant and for each reflection two spatial coordinates, one consisting of radial distance r and azimuth angle ϕ and the other consisting of the ego-motion compensated Doppler velocity v_r and Radar Cross Section (RCS) σ. The PointNet++ [26] is used as a basis for the segmentation algorithm, and it is modified to handle two spatial and two feature dimensions. The model has a Multi-Scale Grouping (MSG) module and Feature Propagation (FP). The MSG module consists of 3 stages: sampling, grouping and feature generation stages. The data-set is sampled and grouped around the selected point, and a feature vector is generated for each sampled point. The output of an MSG module has a lower number of points, hence, providing abstract features. The FP modules propagate the features of sparsely populated point cloud to the next layer. The computation complexity is relatively high when compared to the static object classification. The dimension of the input data is high and the network is deep, resulting in large training and testing time. The higher feature dimensions help augment the segmentation, dynamic object classification and offer high accuracy.

TABLE III
RADAR SENSOR SPECIFICATIONS

Specifications	**LRR**	**MRR**	**SRR**
Transmit Power	55 dBm	-9 dBm/MHz	-9 dBm/MHz
Frequency Band	76-77 GHz	77-81 GHz	77-81 GHz
Bandwidth	600 MHz	600 MHz	4 GHz
Distance range	10-250m	1-100m	0.15-30m
FOV	$\pm 10°$	$\pm 42°$	$\pm 80°$
RCS	0.1 sqm	0.1-10 sqm	10-50 sqm

C. Lidar perception systems

Lidar system uses light pulses to determine the distance and shape of the object. The system is similar to a radar system, although lidar technology utilizes time of flight of the light signal reflected from the object to calculate the distance of the object. Continuous pulses are transmitted and upon reception of the reflected pulse, with the reflection time, drift in frequency and signal strength, a 3D visualization of the surroundings is generated. With this data, the location and velocity of the object are calculated, which are used in the field of collision mitigation, object detection, and driving assistance. The performance in terms of resolution, range, speed, and the azimuth angle is strongly influenced by rain, fog dirt, the lighting, and other environmental conditions. The Table IV lists the typical specifications of the lidar types.

The Velodyne HDL-64E has a user selectable frame rate between 5-15 Hz, and has an output point rate of 1.3 million points per second. Each point is 3 dimensional at minimum for calculation purposes; we consider only the 3-dimensional position data from the Lidar sensor system, and each point is represented by 128 bits in total (32 bit per dimension), subsequently, the data rate becomes **20.8 MB/s**.

The Lidar sensor outputs a sparse 3D points reflected from the objects, where each point represents a point on the object's surface with respect to the position of the sensor. Three main representations for the points are commonly used: point clouds, features, and grids [3]. Points clouds approach use raw sensor data as it provides finer and direct representation of the environment. Feature and grid representation requires pre-processing of the collected data. To extract information from the 3D point cloud, the point cloud is segmented and then the objects are classified. Here, segmentation is clustering of similar points in the point cloud among homogeneous groups, and the classification is performed on these clustered points. The traditional approach transformed the point cloud to 3D voxel grids, however this creates a large data size and requires huge computational resources. The deep learning

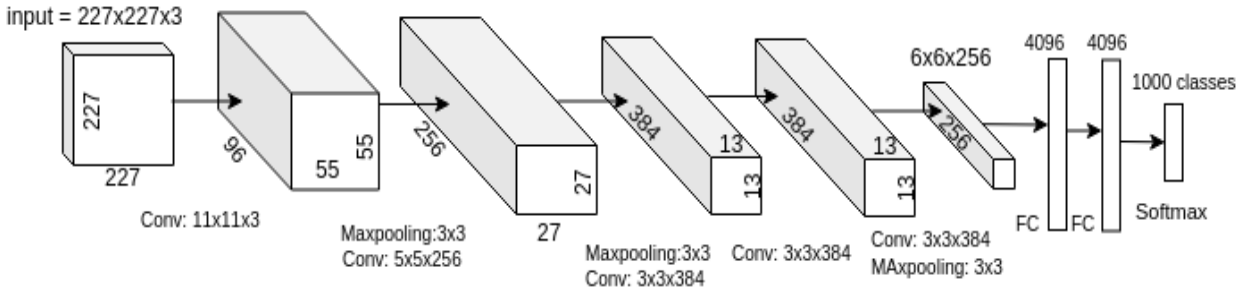


Fig. 1. AlexNet CNN architecture

TABLE IV
LIDAR SENSOR SPECIFICATIONS

Specifications	Solid State	Mechanical
FOV-Horizontal	120°	360°
FOV-Vertical	30°	40°
Scan rate	5-30Hz	5-20Hz
Range (maximum)	100-150 m	100-200 m
Output points	1.2M	1.3M
Power	<15W	>40W
Accuracy	>0.5	>0.4

based PointNet [27] approach takes in the point cloud as it is, in addition of being invariant to permutations of the points. It is a unified method, which does object classification and part segmentation, it has relatively low computation complexity (440 MOPS).

III. COMPUTE SYSTEMS

Two demanding constraints in deep learning system are memory and computation speed to provide real-time accurate performance with a frame rate of 20-30 fps. This section profiles the AlexNet and VGGNet benchmarks metrics (convolution layer and fully connected Layer) for the compute and memory requirements during inference, which are based on the model size, weight and bias. The layers/benchmarks in a standard CNN are shown in Table V and Fig. 1.

Compute complexity: The computation complexity is directly proportional to the number of convolution layers and filter size in the CNN model. The convolution layers perform matrix multiplications, the total operations can be measured in terms of the number of multiply and accumulate (MAC) operations. In the Table VI, it can be seen that the total MACs drastically increases as networks get deeper. The ALU utilization is highest for local response normalization (LRN) Analysis of data in [28] illustrates the ALU and DRAM utilization (Fig. 2). of CNN. LRN involves normalization of one input with data from multiple adjacent kernels [28]. The ALU is significantly higher for fully connected (FC) and convolution (Conv) layers as it involves intense matrix multiplications and additions. We analyzed the computation distribution among different layers of AlexNet model (Fig. 3); the deeper layers have small input feature map, thus resulting in less computation when compared to the initial layers. In the Fig. 3, the layers 1-6 are Conv

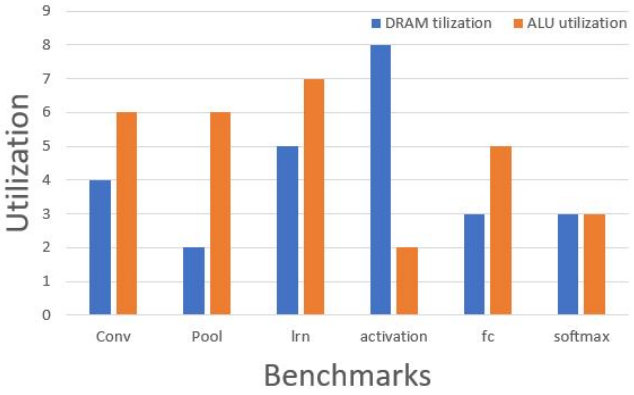


Fig. 2. GPU utilization

TABLE V
CNN BENCHMARKS

Layer	Operation
Convolution(Conv)	Convolution between the feature maps and weights
Max Pooling	A sample-based discretization process
Local response normalization (lrn)	Normalization of one input with adjacent kernels
Rectified Linear Unit(ReLU)	Non-linear activation function
Fully connected (FC)	Multilayer perceptron
Softmax	Function with an output range 0-1

layers and layers 7-8 are FC layers. The multiplication when compared to addition, requires a larger number of cycles and higher energy consumption.

Memory: Small models can be fully loaded onto the on-chip memory, but if the model is large e.g. deep neural networks (DNN), it cannot be stored and requires DRAM access. The total feature map memory requirement for VGGNet [30] with input image of 224x224 pixels is approximately 93 MB and total parameters (weights) memory is 552 MB. From the analysis of Table VI, we see an abrupt increase in the model size and required memory as the network grows deeper. An increase in cache size or DRAM is not beneficial in terms of cycles usage and power consumption. Single DRAM access is 200 times more expensive in terms of energy as shown in Fig.

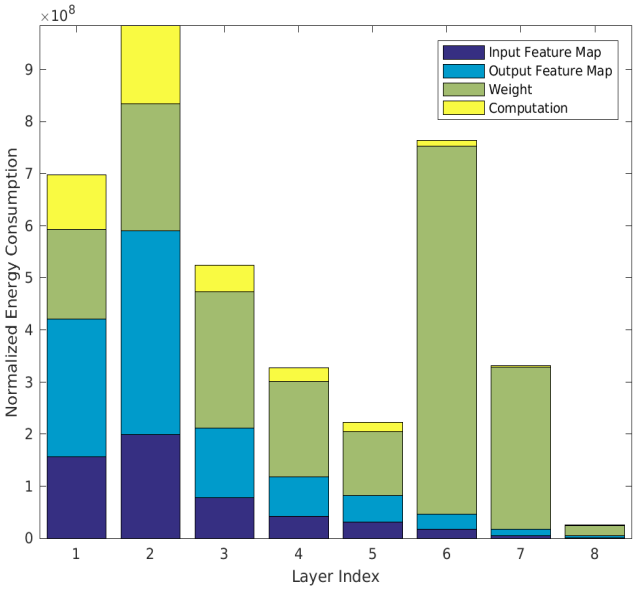


Fig. 3. AlexNet energy profile

TABLE VI
METRICS AND NETWORKS

Metrics	AlexNet	VGG-16
Input size	227x227	224x224
No. of CONV Layers	5	16
No. of weights	2.3M	14.7M
No. of MACs	666M	15.3G
No. of FC layers	3	3
No. of weights	58.6M	124M
No. of MACs	58.6M	124M
Total Weights	61M	138M
Total MACs	724M	15.5G

4. From GPU utilization analysis, the load/store and memory utilization increases as the model size increases. This results in less than real-time performance, and is expensive in terms of cycles and energy consumption. The utilization of DRAM is higher for ”normalization” and ”activation” as these use data from multiple concurrently executing convolutions processes as shown in the Fig. 2.

Energy: Energy consumption is estimated based on computation and the data movement of the three data types (weights, input feature maps, and output feature maps) for every layer [31]. The unit of energy is measured in terms of energy required for one MAC operation. The total energy is the sum of each individual layer for one image. We analyzed the breakup of energy consumption in terms of layers, three data types and computation for unpruned AlexNet model [31]. As illustrated in Fig. 3, the initial layers of the network have higher energy consumption due to data movement when compared to computation, and the deeper layers use significant energy in movement of weights.

Case study: For a pedestrian to be detected at a minimum distance of *50 meters* from the moving vehicle, the camera

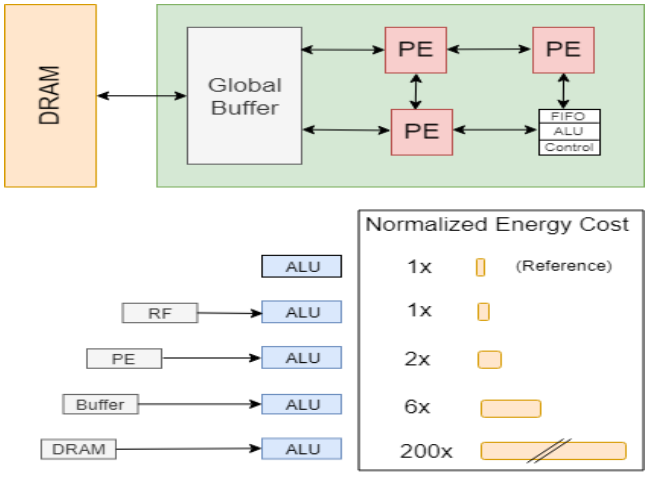


Fig. 4. Energy and memory hierarchy [29]

should have a minimum resolution of *1.6 Mp*. We explored standard deep learning models, *AlexNet* and *VGGNet-16* for an input image resolution of *1280x1280*, pixels while retaining the other variables of the models. The *parameters* are weights of the filters in 32-bit floating point format, and the MAC operations are in 32-bit floating point representation. The memory utilization, assumed to be 50%, is divided among SRAM and DRAM. The Table VII lists the important requirements for two standard models. The values obtained for inference of one image with an inference time of 200 ms. The energy required for memory access is calculated based on 28 nm technology[1],[2]. The number of operations for inference of a single frame, the number of weights, and subsequently total energy consumption increases linearly with numbers of layers. Similarly, as the input resolution increases, the power and total operation increases.

IV. OPTIMIZATION

Without loss of generality, we define optimization space as illustrated in Fig 5. The compute system optimization for inference is dependent on the algorithms used for convolution, and hardware architecture used to reduce the latency and increase throughput.

TABLE VII
COMPUTE REQUIREMENT

Metrics	AlexNet	VGGNet-16
Total Operations	25.5 G	505 G
Parameters(weights)	1.62 G	3.39 G
Memory access	77 G	1515 G
SRAM access energy	11.55 mJ	227.25 mJ
DRAM access energy	862.4 mJ	16968 mJ
Total Energy	873.95 mJ	17195 mJ
Power	4.36 W	85.9 W

[1]http://www.pee.ufrj.br/index.php/pt/producao-academica/teses-de-doutorado/2017-1/2016033211-study-and-development-of-low-power-consumption-srams-on-28-nm-fd-soi-cmos-process/file

[2]https://pubweb.eng.utah.edu/ cs7810/pres/14-7810-02.pdf

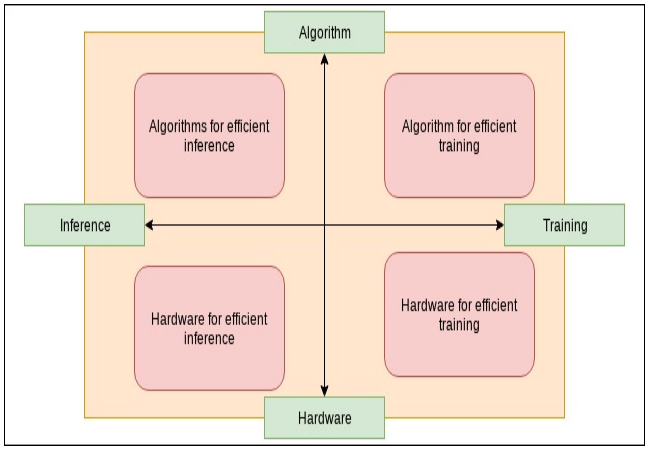


Fig. 5. Optimization space

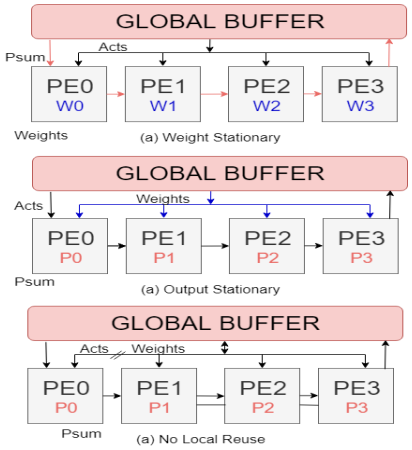


Fig. 6. Data-flow/Spatial architectures [29]

A. *Throughput and Memory Optimization:*

The throughput can be increased by adopting alternate convolution methods, by using data-flow architectures, changing number representation and exploiting the sparsity of data. By changing the number representation, the model can be fit in the on-chip memory.

Convolution algorithms: The convolution in CNN is matrix multiplication of the filter over the input image. The Conv and FC layers of a CNN are the computation kernels, which could be sped up by applying computational transformations [32]. The inputs need to be pre-processed before being applied to the network.

a) Fast Fourier Transform(FFT): FFT has convolution property; the FFT of the filter and FFT of input feature map is multiplied in the frequency domain, and subsequently an inverse FFT is applied to the resulting product to obtain the output feature map in the spatial domain. The FFT increases the storage capacity and required bandwidth, while decreasing the computations. However, FFT makes it difficult to apply optimization such as those that exploit weight sparsity.

b) Winograd Algorithm: Although the algorithm reduces the number of multiplications by rearranging the computations, this approach suffers from reduced numerical stability, increased storage, and requires a specialized processing unit, which is dependent on the size of the filter.

Several advantages and disadvantages exists for each of the above-mentioned methods. The mapping and scheduling of inputs and weights becomes complicated, the weights and feature maps data is replicated in the memory and it has complicated data access pattern. The speedup depends on the filter size and feature maps in the layers, the FFT is preferred for filters greater than 5x5 and Winograd for filters 3x3 and below. The complexity of memory control logic is high for data access.

Spatial architecture: The compute architectures is subdivided into temporal and spatial architectures. Temporal architectures include CPUs and GPUs, as a centralized control is used for a large number of ALUs. Spatial architectures are used in accelerators like ASIC and FPGA based designs. The data-flow processing in spatial architectures increases the data reuse from local memory to reduce energy consumption, and the ALUs form a processing chain, which allows them to communicate data among each other directly as shown in Fig. 6. The ALU have their own control logic and local memory and is referred to as Processing Engine (PE). The memory access provides a bottleneck for DNN processing. The cost per memory access increases as we move away from the ALU as shown in Figure 4. The DRAM access being the most expensive as the data needs to be accessed from off-chip, and data access from Register File (RF) is the lowest. Multiple levels of memory help to improve energy efficiency by providing low-cost access to lower memories like RF, shared memory or buffer memory. The drawback of low-cost access memories is low storage capacity. In CNN the data from filters, input feature maps (activations) and partial products are reused. This property can be exploited to reduce larger memory access, like DRAM access, by storing them in local memory hierarchy and accessing them multiple times. The Fig. 6 shows different data-flow architectures. The Data-flow architecture encapsulates a global buffer and numerous PEs. The DRAM, global buffer and PE communicate with each other through input and output FIFO. The global buffer is used to exploit the input data reuse and hide DRAM access latency, or for storing intermediate results. The PE consists of an ALU data-path, which can perform MAC and addition operation and a register file. The *Act* is the input feature-maps activations, *Weight* is the weight of the filter and *Psum* is the partial sums obtained after the MAC operation. The Table VIII compares the state of the design in spatial architecture.

1. Weight stationary (WS): The filter weights are stored in lower memory hierarchy after loading them from the DRAM. These are stationary for a longer time in the RF of PE (Figure6(a)). The processing uses these weights as many times as required for MAC operation while the weights are present in, RF, thus facilitating convolution and filter reuse of weights. The input feature maps (inputfmaps/activations) are broadcast

to all the PEs and partial products are accumulated across the PEs. Few other examples are found in [29], [33], [34].

2. Output Stationary (OS): For the same output activation, the accumulated partial sums are stored in the local RF. The input activations are streamed and weights are broadcast across the PEs as shown in the Figure 6(b). Energy consumption is reduced due to the storage of partial sums. Few examples of *OS* can be found in [35]–[37].

3. No local reuse (NLR): The PEs in this method does not contain local memory; as result products and weights are not stored in local memory (RF), there is no local data reuse. The data is stored in the global buffer as shown in the Figure 6(c), and as a result, the total access to the global buffer increases. This provides efficiency in terms of area (μm2/bit) but not in terms of energy (pJ/bit). The input activations should be multi-cast and the weights are single-cast, while partial sums are accumulated across PEs; few examples of *NLR* can be found in [38]–[40].

4. Row Stationary (RS): The row stationary aims at maximum reuse of every data type (input activations, weights and partial sums) at the local memory (RF) of the PEs unlike the *WS*, which stores only the weights and *OS*, and OS, which stores the partial sums. This reduces overall energy consumption. The paper [29] proposes the RS method and it provides a 1.4x to 2.5x energy efficiency when compared to other data flow methods [29].

B. Number Representation

The CNN that is run on CPU and GPU typically use 32bit floating point representation. The weight and activation values do not require this precision for inference and thus, 8-bit fixed point and 16-bit fixed point representations are chosen. The dynamic range of activations and weights vary from layer to layer and type of layer. In [41] an 8-bit filter weights and 10-bits activations are used achieving less than 1% degradation in accuracy. By reducing the precision, the energy and cost of storing the data also reduce. The Fig. 7 shows the 32-bit floating point number and 8-bit dynamic fixed point examples. The 32-bit floating point is represented by $(-1)^s \times m \times 2^{(e-127)}$, where s is the sign bit and m is the 23-bit mantissa and e is the 8-bit exponent. The N-bit fixed point number is represented by $(-1)^s \times m \times 2^{-f}$, and

TABLE VIII
STATE OF THE ART DIGITAL DESIGN

Metric	EEPSCP [42]	Eyeriss [29]	EIE [43]
Technology	40nm LP	65nm	45nm
Supply(V)	1.1	1	1
Area(mm^2)	2.4	12.2	40.8
On-chip Memory(kB)	148	192	10368
Power(mW)	76	278	579
Frame rate(fps)	47	8	20
Throughput(GMACs/sec)	102	33.6	51.2
Frequency (MHz)	204	200	800

performs scaling action, the m is the (N-1)-bit mantissa. The

TABLE IX
8-BIT REPRESENTATION

	ADD		MULTIPLY	
Factor	32-bit fix. pt.	32-bit fl. pt.	32-bit fix. pt.	32-bit fl. pt.
Area	3.8x	116x	3.8x	116x
Energy	3.3x	30x	3.3x	30x

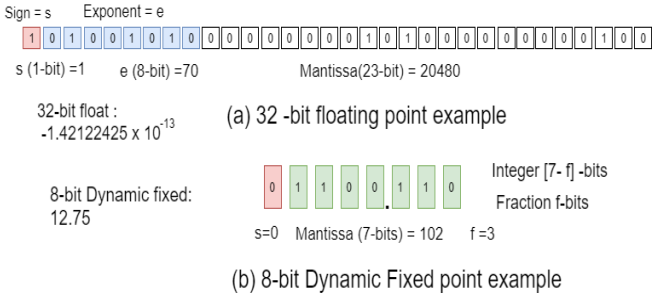


Fig. 7. Number representation [32]

advantages of 8-bit fixed point representation [44], is shown in Table IX. The 8-bit representation is efficient in terms of area and energy.

V. CONCLUSION

In this paper, a number of the sensor subsystems with its principle of operation, functionality, characteristics, specification, and communication protocol are documented. The cognitive algorithms corresponding to each sensor data type are analyzed and the possible advantages and disadvantages of each approach are described. The profiling section describes the energy, memory and computation distribution in a standard CNN. The type of compute architectures are compared and optimization methods in terms of convolution algorithms, datapath, algorithm and number representation are explored.

REFERENCES

[1] P. Navarro, C. Fernández, R. Borraz, D. Alonso, P. J. Navarro, C. Fernández, R. Borraz, and D. Alonso, "A Machine Learning Approach to Pedestrian Detection for Autonomous Vehicles Using High-Definition 3D Range Data," *Sensors*, vol. 17, p. 18, 12 2016.

[2] "Three Sensor Types Drive Autonomous Vehicles — Sensors Magazine."

[3] S. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. Eng, D. Rus, and M. Ang, "Perception, Planning, Control, and Coordination for Autonomous Vehicles," *Machines*, vol. 5, p. 6, 2 2017.

[4] J. Van Brummelen, M. OBrien, D. Gruyer, and H. Najjaran, "Autonomous vehicle perception: The technology of today and tomorrow," *Transportation Research Part C: Emerging Technologies*, vol. 89, pp. 384–406, 4 2018.

[5] R. Vivacqua, R. Vassallo, and F. Martins, "A low cost sensors approach for accurate vehicle localization and autonomous driving application," *Sensors*, vol. 17, no. 10, 2017.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[7] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Region-based convolutional networks for accurate object detection and segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, pp. 142–158, Jan 2016.

[8] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015.

[9] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015.

[10] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. Mujica, A. Coates, and A. Y. Ng, "An empirical evaluation of deep learning on highway driving," *CoRR*, vol. abs/1504.01716, 2015.

[11] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *CoRR*, vol. abs/1312.6229, 2013.

[12] C. Szegedy, A. Toshev, and D. Erhan, "Deep neural networks for object detection," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, (USA), pp. 2553–2561, Curran Associates Inc., 2013.

[13] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.

[14] M. A. Sadeghi and D. Forsyth, "30Hz Object Detection with DPM V5," in *Computer Vision – ECCV 2014* (D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, eds.), (Cham), pp. 65–79, Springer International Publishing, 2014.

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[16] Y. Tian, J. Gelernter, X. Wang, W. Chen, J. Gao, Y. Zhang, and X. Li, "Lane marking detection via deep convolutional neural network," *Neurocomputing*, vol. 280, pp. 46–55, 3 2018.

[17] C. Brust, S. Sickert, M. Simon, E. Rodner, and J. Denzler, "Convolutional patch networks with spatial prior for road detection and urban scene understanding," *CoRR*, vol. abs/1502.06344, 2015.

[18] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," *CoRR*, vol. abs/1411.4038, 2014.

[19] R. Mohan, "Deep deconvolutional networks for scene parsing," *CoRR*, vol. abs/1411.4101, 2014.

[20] G. L. Oliveira, W. Burgard, and T. Brox, "Efficient deep models for monocular road segmentation," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4885–4891, Oct 2016.

[21] "Lidar, Radar Digital Cameras: the Eyes of Autonomous Vehicles."

[22] J. Lombacher, M. Hahn, J. Dickmann, and C. Whler, "Potential of radar for static object classification using deep learning methods," in *2016 IEEE MTT-S International Conference on Microwaves for Intelligent Mobility (ICMIM)*, pp. 1–4, May 2016.

[23] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *Computer*, vol. 22, pp. 46–57, June 1989.

[24] J. Lombacher, M. Hahn, J. Dickmann, and C. Wohler, "Object classification in radar using ensemble methods," in *2017 IEEE MTT-S International Conference on Microwaves for Intelligent Mobility (ICMIM)*, pp. 87–90, IEEE, 3 2017.

[25] O. Schumann, M. Hahn, J. Dickmann, and C. Whler, "Semantic segmentation on radar point clouds," in *2018 21st International Conference on Information Fusion (FUSION)*, pp. 2179–2186, July 2018.

[26] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *CoRR*, vol. abs/1706.02413, 2017.

[27] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," *CoRR*, vol. abs/1612.00593, 2016.

[28] S. Dong and D. Kaeli, "Dnnmark: A deep neural network benchmark suite for gpus," in *Proceedings of the General Purpose GPUs*, GPGPU-10, (New York, NY, USA), pp. 63–72, ACM, 2017.

[29] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss," *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 367–379, 6 2016.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[31] T. Yang, Y. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," *CoRR*, vol. abs/1611.05128, 2016.

[32] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, pp. 2295–2329, Dec 2017.

[33] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," *CoRR*, vol. abs/1512.04295, 2015.

[34] S. Park, J. Park, K. Bong, D. Shin, J. Lee, S. Choi, and H. Yoo, "An energy-efficient and scalable deep learning/inference processor with tetra-parallel mimd architecture for big data applications," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 9, pp. 838–848, Dec 2015.

[35] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015.

[36] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 92–104, June 2015.

[37] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 13–19, Oct 2013.

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

[39] 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," *SIGARCH Comput. Archit. News*, vol. 42, pp. 269–284, Feb. 2014.

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

[41] Y. Ma, N. Suda, Y. Cao, J. sun Seo, and S. Vrudhula, "Scalable and modularized rtl compilation of convolutional neural networks onto fpga," in *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, (United States), Institute of Electrical and Electronics Engineers Inc., 9 2016.

[42] B. Moons and M. Verhelst, "An energy-efficient precision-scalable convnet processor in 40-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 52, pp. 903–914, April 2017.

[43] 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*, vol. abs/1602.01528, 2016.

[44] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, Feb 2014.

List Of Acronyms

CNN . . . . . Convolution Neural Network

CSV . . . . . Comma-Separated Values

DNN . . . . . Deep Neural Network

DRAM . . . . . Dynamic Random-Access Memory

EDP . . . . . Energy Delay Product

FoM . . . . . Figures of Merit

HBM . . . . . High Bandwidth Memory

HMC . . . . . Hybrid Memory Cube

ILSVRC . . . . . ImageNet Large Scale Visual Recognition Competition

IFmap . . . . . Input Feature Map

MAC . . . . . Multiply and Accumulate

MLP . . . . . Muli-layer Perceptron

NN . . . . . Neural Network

OFmap . . . . . Output Feature Map

PE . . . . . Processing Engine

PCM-RAM . . . . Phase Change Random Access Memory

ReRAM . . . . . Resistive Random Access Memory

RF . . . . . Register File

SRAM . . . . . Static Random-Access Memory

SIMD . . . . . Single Instruction Multiple Data

SIMT . . . . . Single Instruction Multiple Thread

SLP . . . . . Single-layer Perceptron

STT-RAM . . . . . Spin Transfer Torque Random Access Memory

SA . . . . . Systolic Array

TLM . . . . . Transaction Level Modelling

VGGNet . . . . . Visual Geometry Group Network

Bibliography

[1] C. Szegedy, A. Toshev, and D. Erhan, "Deep neural networks for object detection," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 2553–2561. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999792.2999897

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," Tech. Rep. [Online]. Available: http://image-net.org/challenges/LSVRC/2015/

[4] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "Ultra low power deep-learning-powered autonomous nano drones," *CoRR*, vol. abs/1805.01831, 2018. [Online]. Available: http://arxiv.org/abs/1805.01831

[5] Z. Du *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 92–104.

[6] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving," Tech. Rep. [Online]. Available: http://deepdriving.cs.princeton.edu

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[8] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[9] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[10] Y. Chen *et al.*, "DaDianNao: A Machine-Learning Supercomputer," 2014. [Online]. Available: http://delivery.acm.org/10.1145/2750000/2742217/p609-chen.pdf?ip=130.161.36.248&id=2742217&acc=ACTIVE%20SERVICE&key=0C390721DC3021FF%2E512956D6C5F075DE%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1542831928_4b117ed61a9b9488268f3aae5d14935b

[11] T. Chen, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning." [Online]. Available: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2541940.2541967

[12] "Efficient methods and hardware for deep learning — Stanford Digital Repository." [Online]. Available: https://purl.stanford.edu/qf934gh3708

[13] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," Tech. Rep. [Online]. Available: http://pjreddie.com/yolo9000/

[14] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Region-based convolutional networks for accurate object detection and segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 1, pp. 142–158, Jan 2016.

[15] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[16] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, "Natural language processing (almost) from scratch," *CoRR*, vol. abs/1103.0398, 2011. [Online]. Available: http://arxiv.org/abs/1103.0398

[17] B. Alipanahi, A. Delong, M. T. Weirauch, and B. J. Frey, "Predicting the sequence specificities of dna- and rna-binding proteins by deep learning," *Nature Biotechnology*, vol. 33, pp. 831–838, 2015.

[18] A. Esteva *et al.*, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, pp. 115–, Jan. 2017. [Online]. Available: http://dx.doi.org/10.1038/nature21056

[19] M. Jermyn *et al.*, "Neural networks improve brain cancer detection with raman spectroscopy in the presence of operating room light artifacts," *Journal of Biomedical Optics*, vol. 21, p. 094002, 09 2016.

[20] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *CoRR*, vol. abs/1504.00702, 2015. [Online]. Available: http://arxiv.org/abs/1504.00702

[21] S. Gupta *et al.*, "Cognitive mapping and planning for visual navigation," *CoRR*, vol. abs/1702.03920, 2017. [Online]. Available: http://arxiv.org/abs/1702.03920

[22] M. Pfeiffer *et al.*, "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots," *CoRR*, vol. abs/1609.07910, 2016. [Online]. Available: http://arxiv.org/abs/1609.07910

[23] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," *CoRR*, vol. abs/1610.03295, 2016. [Online]. Available: http://arxiv.org/abs/1610.03295

[24] T. Zhang *et al.*, "Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search," *CoRR*, vol. abs/1509.06791, 2015. [Online]. Available: http://arxiv.org/abs/1509.06791

[25] K. Abdelouahab *et al.*, "Accelerating CNN inference on fpgas: A survey," *CoRR*, vol. abs/1806.01683, 2018. [Online]. Available: http://arxiv.org/abs/1806.01683

[26] "Tutorial on Hardware Architectures for Deep Neural Networks." [Online]. Available: http://eyeriss.mit.edu/tutorial.html

[27] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *ICANN*, 2014.

[28] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 367–379, 6 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3007787.3001177

[29] S. Dong and D. Kaeli, "Dnnmark: A deep neural network benchmark suite for gpus," in *Proceedings of the General Purpose GPUs*, ser. GPGPU-10. New York, NY, USA: ACM, 2017, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/3038228.3038239

[30] "Deep Neural Network Energy Estimation Tool — Tool for Designing Energy-Efficient Deep Neural Networks." [Online]. Available: https://energyestimation.mit.edu/

[31] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982. [Online]. Available: https://doi.org/10.1109/MC.1982.1653825

[32] "Data Flow and Systolic Array Architectures." [Online]. Available: https://pdfs.semanticscholar.org/47f9/395124d2a1845d6430cd122f0c9a74af2dd8.pdf

[33] S. Gupta *et al.*, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015. [Online]. Available: http://arxiv.org/abs/1502.02551

[34] M. Peemen, A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," 10 2013, pp. 13–19.

[35] V. Gokhale *et al.*, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks," Tech. Rep. [Online]. Available: https://www.cv-foundation.org//openaccess/content_cvpr_workshops_2014/W17/papers/Gokhale_A_240_G-opss_2014_CVPR_paper.pdf

[36] M. Sankaradas *et al.*, "A massively parallel coprocessor for convolutional neural networks," in *Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, ser. ASAP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 53–60. [Online]. Available: http://dx.doi.org/10.1109/ASAP.2009.25

[37] S. Chakradhar *et al.*, "A dynamically configurable coprocessor for convolutional neural networks," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1815993

[38] L. Cavigelli *et al.*, "Origami: A convolutional network accelerator," *CoRR*, vol. abs/1512.04295, 2015. [Online]. Available: http://arxiv.org/abs/1512.04295

[39] C. Zhang *et al.*, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks." [Online]. Available: http://dx.doi.org/10.1145/2684746.2689060.

[40] J. Albericio *et al.*, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 382–394. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123982

[41] B. Moons and M. Verhelst, "A 0.3-2.6 TOPS/W precision-scalable processor for real-time large-scale convnets," *CoRR*, vol. abs/1606.05094, 2016. [Online]. Available: http://arxiv.org/abs/1606.05094

[42] S. Han *et al.*, "EIE: efficient inference engine on compressed deep neural network," *CoRR*, vol. abs/1602.01528, 2016. [Online]. Available: http://arxiv.org/abs/1602.01528

[43] M. Horowitz, "1.1 Computing's energy problem (and what we can do about it)," in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, 2014.

[44] Y. Ma *et al.*, "Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA," in *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, 2016.

[45] M. R. Shashanka, S. Kumar, A. Zjajo, and R. v. Leuken, "Towards Computationally-Efficient Cognitive Sensor Systems for Autonomous Vehicles," 2019, in press.

[46] D. Keitel-Schulz and N. Wehn, "Embedded dram development: Technology, physical design, and application issues," *IEEE Des. Test*, vol. 18, no. 3, pp. 7–15, May 2001. [Online]. Available: https://doi.org/10.1109/54.922799

[47] H. Jun *et al.*, "Hbm (high bandwidth memory) dram technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, May 2017, pp. 1–4.

[48] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," *2012 Symposium on VLSI Technology (VLSIT)*, pp. 87–88, 2012.

[49] D. Kim *et al.*, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 380–392, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/3007787.3001178

[50] M. Gao *et al.*, "Tetris: Scalable and efficient neural network acceleration with 3d memory," *SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 751–764, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3093337.3037702

[51] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 367–379, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/3007787.3001177

[52] D. Wu *et al.*, "A Novel Low-Communication Energy-Efficient Reconfigurable CNN Acceleration Architecture," Tech. Rep. [Online]. Available: http://kalman.mee.tcd.ie/fpl2018/content/pdfs/FPL2018-43iDzVTplcpussvbfIaaHz/3hlrKaEMsTw5knRjckQN3X/1wY9VRDv9YdvKXPScFOkwU.pdf

[53] G. Venkatesh, E. Nurvitadhi, and D. Marr, "ACCELERATING DEEP CONVOLUTIONAL NETWORKS USING LOW-PRECISION AND SPARSITY," Tech. Rep. [Online]. Available: https://arxiv.org/pdf/1610.00324.pdf

[54] "About SystemC - The Language for System-Level Modeling, Design and Verification." [Online]. Available: https://www.accellera.org/community/systemc/about-systemc

[55] M. Poremba *et al.*, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 1543–1546. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2757168

[56] J. J. Tithi, N. C. Crago, and J. S. Emer, "Exploiting spatial architectures for edit distance algorithms," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 23–34.

[57] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, Feb 2014, pp. 10–14.

[58] N. Brunie, "Modified fused multiply and add for exact low precision product accumulation," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, July 2017, pp. 106–113.

[59] F. Stas and D. Bol, "A 0.4v 0.08fj/cycle retentive true-single-phase-clock 18t flip-flop in 28nm fdsoi cmos," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2017, pp. 1–4.