HAROKOPIO UNIVERSITY

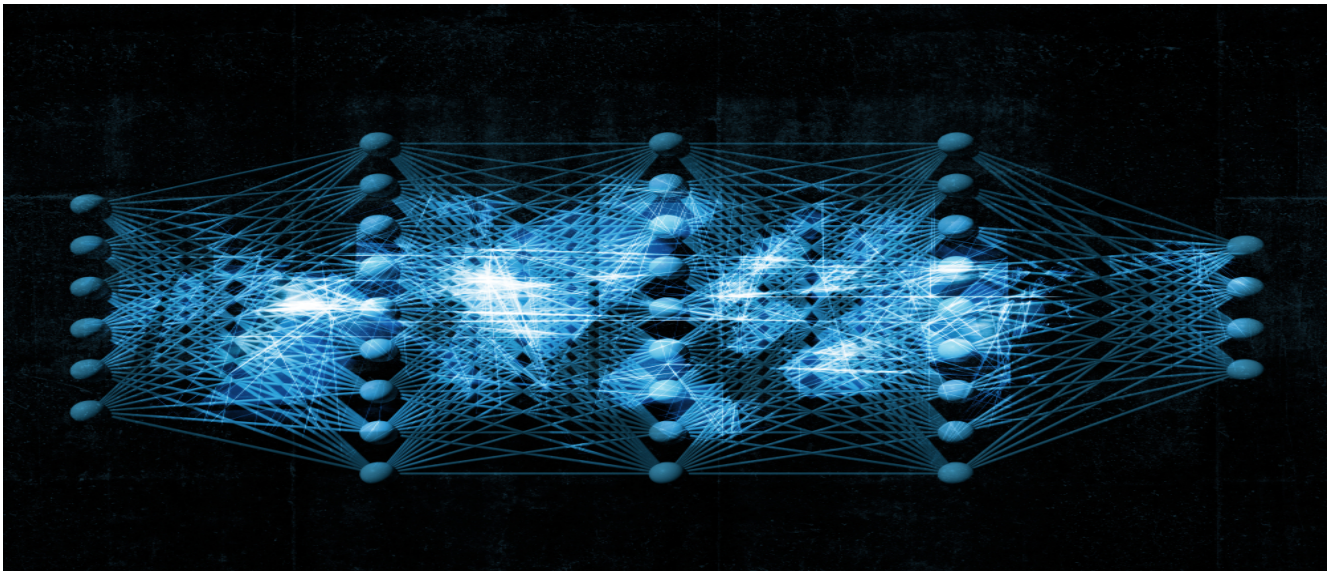
School of Digital Technology
Informatics and Telematics
Postgraduate Program Informatics and Telematics
Telecommunication Networks and Telematic Services

Energy Efficient DNN Inference Through Approximate Near-Threshold Voltage Computing

Master's Thesis

by

Nikolaos Iatridis



Athens, 2022

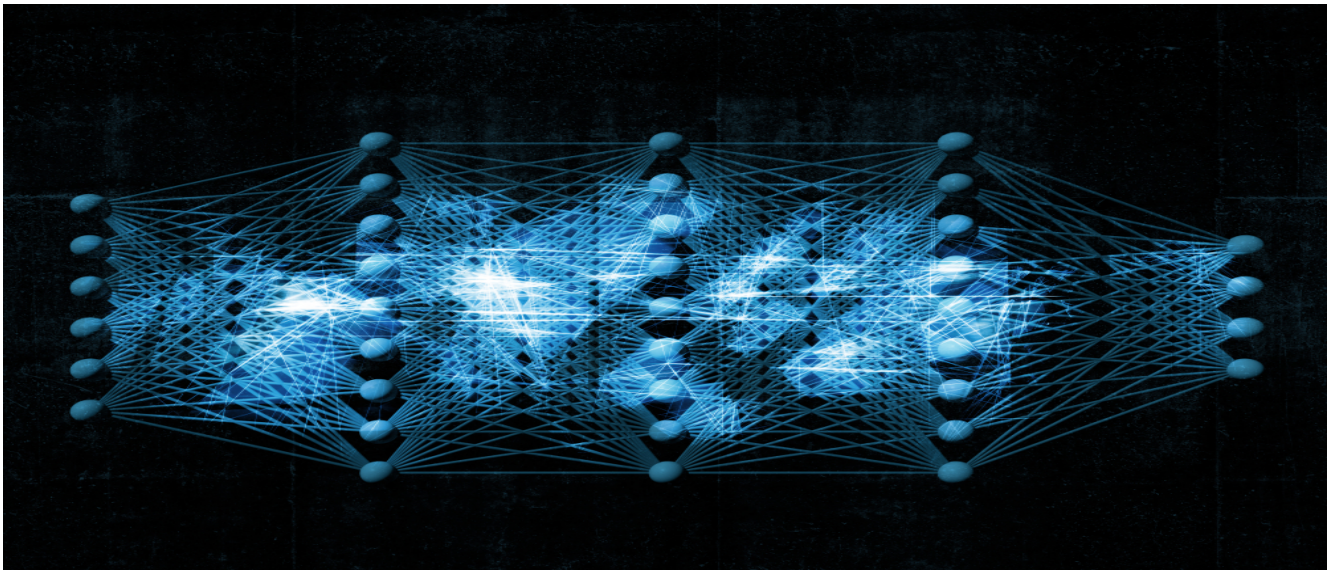
ΧΑΡΟΚΟΠΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

Σχολή Ψηφιακής Τεχνολογίας
Τμήμα Πληροφορικής και Τηλεματικής
Πρόγραμμα Μεταπτυχιακών Σπουδών Πληροφορική και Τηλεματική
Τηλεπικοινωνιακά Δίκτυα και Υπηρεσίες Τηλεματικής

Ενεργειακά αποδοτικά DNN με συνδυασμό
Approximate και Near-Threshold Voltage Computing

Μεταπτυχιακή εργασία

Νικόλαος Ιατρίδης



Αθήνα, 2022

HAROKOPIO UNIVERSITY

School of Digital Technology
Informatics and Telematics
Postgraduate Program Informatics and Telematics
Telecommunication Networks and Telematic Services

Committee

Supervisor: Dr. Sotirios Xydis,
Assistant Professor, Department of Informatics and Telematics,
Harokopio University

Dr. Christos Diou,
Assistant Professor, Department of Informatics and Telematics,
Harokopio University

Dr. Thomas Kamalakis,
Professor and Dean, Department of Informatics and Telematics,
Harokopio University

Ethics and Copyright Statement (Required)

I, Nikolaos Iatridis, hereby declare that:

1) I am the owner of the intellectual rights of this original work and to the best of my knowledge, my work does not insult persons, nor does it offend the intellectual rights of third parties.

2) I accept that Library and Information Centre of Harokopio University may, without changing the content of my work, make it available in electronic form through its Digital Library, copy it in any medium and / or any format and hold more than one copy for maintenance and safety purposes.

3) I have obtained, where necessary, permission from the copyright owners to use any third-party copyright material reproduced in the master thesis while the corresponding material is visible in the submitted work.

Dedicated to my Fantastic Three:

Paraskevi, Antonia, Maria

QUOTES

"The microprocessor is a miracle."

Bill Gates

"Everything that civilization has to offer is a product of human intelligence; we cannot predict what we might achieve when this intelligence is magnified by the tools that AI may provide, but the eradication of war, disease, and poverty would be high on anyone's list. Success in creating AI would be the biggest event in human history. Unfortunately, it might also be the last."

Stephen Hawking

AKNOWLEDGEMENTS

First and foremost, I would like to express a deep appreciation to my supervisor Dr. Sotirios Xydis for his continuous support, guidance, and patience during the fulfillment of this work.

I would like to offer special thanks to the other two members of my committee, Dr. Christos Diou and Dr. Thomas Kamalakis for their endless encouragement throughout my study at Harokopio University.

I am also grateful to all staff members and people I met there.

Finally, I would like to thank my wife who is the true hero behind this effort.
I wish my research journey would keep going.....

TABLE OF CONTENTS

LIST OF FIGURES 10
LIST OF TABLES 12
ACRONYMS 14
ABSTRACT 17
ΠΕΡΙΛΗΨΗ 18
1 INTRODUCTION 19
1.1 The Deep Neural Network Challenge 19
1.2 Motivation 20
1.3 Benefits of Approximate Near-Threshold Voltage Computing 20
1.4 The scope of the thesis 22
2 BACKGROUND ON DEEP NEURAL NETWORKS 23
2.1 What is Deep Learning? 23
2.2 Training and Inference 25
2.3 Types of Layers 27
2.3.1 Conv Layer 27
2.3.2 FC Layer 28
2.3.3 Nonlinearity 29
2.3.4 Pooling and Unpooling 30
2.3.5 Normalization 31
2.3.6 Compound Layers 32
2.4 The Convolutional Neural Networks 33
3 RELATED WORK ON ENERGY EFFICIENT DNN 37
3.1 Architectures for DNN Workloads 37
3.1.1 CPUs 38
3.1.2 GPUs 40
3.1.3 FPGAs 41
3.1.4 ASICs 42

3.2 Approximate Computing in DNNs 44
3.3 Near-Threshold Computing in DNNs 48
4 HARWARE ARCHITECTURES FOR DNN PROCESSING 51
4.1 Basic Key Metrics 51
4.2 The case of DNN Accelerators 53
4.3 Examining Data Reuse 54
4.3.1 Temporal Reuse 54
4.3.2 Spatial reuse 55
4.4 Why Dataflows are important 56
5 THE PROPOSED NTV-DNN FRAMEWORK 61
5.1 Summary 61
5.2 The NTV-DNN architectural model 61
5.3 Voltage allocation and scaling for NTV-DNN 64
5.4 NTV-DNN error model 68
5.5 NTV-DNN assessment tool flow 72
5.5.1 Pytorch : An open source machine learning framework 72
5.5.2 MAESTRO cost model 73
5.5.3 PytorchFI runtime fault injector 75
5.5.4 CIFAR-10 dataset 76
5.6 Experimental Setup 77
5.7 Experimental Results 83
5.7.1 Exploring voltage island granularity 83
5.7.1.1 Intra-layer 83
5.7.1.2 Cross-layer policies 87
5.7.2 Iso-resource NTV vs STC DNNs 90
5.7.3 Iso-performance NTV vs STC DNNs 91
5.7.4 NTV-DNN under relaxed error 92
5.7.5 NTV-DNN under different dataflows 97

5.7.6 The effect of Vth variability 100
5.8 Conclusions 102
5.9 Future Work 102
6 REFERENCES 103
7 APPENDIX A. 108

LIST OF FIGURES

Figure 1: An example of DNN used to image recognition 19
Figure 2: Power and energy relation 20
Figure 3: Energy efficiency of NTV operation 21
Figure 4: Trends of major sources of power dissipation in nano-CMOS transistor 22
Figure 5: Deep Learning as a subset of AI 23
Figure 6: A single neuron in a DNN and its main structure 24
Figure 7: Gradient descent (a) and backpropagation (b) 26
Figure 8: Illustration of a Conv operation 28
Figure 9: A Fully Connected Network 29
Figure 10: Various nonlinear activation functions 30
Figure 11: Max Pooling and Average Pooling Methods 31
Figure 12: Zero-insertion and Nearest-neighbors Unpooling Methods 32
Figure 13: Batch Normalization method 33
Figure 14: The architecture of a modern deep CNN 34
Figure 15: The architecture of AlexNet DNN model 34
Figure 16: Inception module from GoogLeNet 35
Figure 17: Residual learning: a building block 36
Figure 18: (a) Residual learning: a building block (b) ResNet block 36
Figure 19: Fire module: SqueezeNet's building block 37
Figure 20: High parallel architectural paradigms 38
Figure 21: Block diagram of a uniprocessor-CPU 39
Figure 22: (a) A SIMD unit (b) An FMA unit 40
Figure 23: A high-end GPU-based accelerator (NVIDIA Fermi GPU) 41
Figure 24: Basic structure of an FPGA 42
Figure 25: Block diagram of a TPU 43
Figure 26: The Printed Circuit Board of a TPU 43
Figure 27: The Systolic dataflow inside the Matrix Multiply Unit 45
Figure 28: Overview of Sculptor 46
Figure 29: The Scalpel framework 47
Figure 30: Overview of TTQ 48
Figure 31: FPEC number format and processing element design 48
Figure 32: Block diagram of GreenTPU 49

Figure 33: Overview of SAS 51
Figure 34: The energy consumption for various arithmetic operations and memory accesses in a 45nm process 53
Figure 35: DNN accelerator architecture 55
Figure 36: Overview of data reuse in DNN accelerators 56
Figure 37: The actions of read/write in a MAC 57
Figure 38: Levels of local storage hierarchy with different energy costs 58
Figure 39: Forms of input data reuse 59
Figure 40: Types of dataflows 59
Figure 41: Variants of Output Stationary Dataflow 60
Figure 42: An overview of Row Stationary Dataflow 61
Figure 43: An overview of the proposed NTV-DNN framework 63
Figure 44: The NTC Analysis tool 64
Figure 45: The NTV-DNN TPU based systolic array accelerator with a VI formation 65
Figure 46: The Power breakdown of an STC-16core and an NTC-128core architecture with and without DIBL effect 66
Figure 47: The NTV-DNN TPU based systolic array architecture with a cluster of VIs working in NTC 68
Figure 48: An overview of the NTV-DNN Error Model 73
Figure 49: PyTorchFI output summary of the AlexNet error model 74
Figure 50: The kcp_ws NVDLA-like dataflow for the 1st conv2d layer of AlexNet 75
Figure 51: The FI procedure in the 1st conv2d layer of AlexNet 76
Figure 52: Training a CNN Model with PyTorch 78
Figure 53: (a) An overview of mapping CONV2D to an accelerator (b) High-level Tool flow of MAESTRO 79
Figure 54: An overview of the supported hardware in MAESTRO 79
Figure 55: An overview of PyTorchFI 80
Figure 56: The mapping analysis convention 83
Figure 57: Power consumption per layer in NTC for different $V_{dd,\,NTC}$ and $cluster\_size_{NTC}$ (AlexNet) 89
Figure 58: Power consumption per layer in STC for AlexNet 90
Figure 59: Energy consumption per layer in STC for AlexNet 91
Figure 60: Energy consumption per layer in NTC for different $V_{dd,\,NTC}$ and $cluster\_size_{NTC}$ (AlexNet) 91
Figure 61: Performance per layer in STC for AlexNet 92
Figure 62: Performance per layer in NTC for different $V_{dd,\,NTC}$ and $cluster\_size_{NTC}$ (AlexNet) 92
Figure 63: Power consumption of AlexNet for $cluster\_size_{NTC} = 256$ 94

Figure 64: Energy consumption of AlexNet for cluster_size$_{NTC}$ = 256 94
Figure 65: Execution time of AlexNet for cluster_size$_{NTC}$ = 256 94
Figure 66: Total power NTC vs. STC for different DNN Models and cluster_size$_{NTC}$ 96
Figure 67: Total energy NTC vs. STC for different DNN Models and cluster_size$_{NTC}$ 97
Figure 68: Total performance NTC vs. STC for different DNN Models and cluster_size$_{NTC}$ 97
Figure 69: Power consumption for different DNN Models in NTC vs. STC 98
Figure 70: Energy consumption for different DNN Models in NTC vs. STC 98
Figure 71: Performance for different DNN Models in NTC vs. STC 98
Figure 72: The measured accuracy of different DNN Models in STC 99
Figure 73: The measured accuracy of different DNN Models for cluster_size$_{NTC}$ = 32 100
Figure 74: The measured accuracy of different DNN Models for cluster_size$_{NTC}$ = 64 100
Figure 75: The measured accuracy of different DNN Models for cluster_size$_{NTC}$ = 128 101
Figure 76: The measured accuracy of different DNN Models for cluster_size$_{NTC}$ = 256 101
Figure 77: The new performance of AlexNet in NTC for different F_{NTC} 102
Figure 78: The power gain of different DNN Models for various dataflow strategies 104
Figure 79: The energy gain of different DNN Models for various dataflow strategies 104
Figure 80: The performance of different DNN Models for various dataflow strategies 104
Figure 81: Power gain of different DNN Models for various V_{th} 106
Figure 82: Energy gain of different DNN Models for various V_{th} 106
Figure 83: Performance of different DNN Models for various V_{th} 107

LIST OF TABLES

Table 1: Shape parameters of a Conv Layer 29
Table 2: Different CNN models and their cost 34
Table 3: List of functions in NTC Analysis tool 63
Table 4: List of functions in NTV-DNN Error Model 73
Table 5: The mapping of data on the 1st cluster for the kcp_ws dataflow 75
Table 6: FI locations for the 1st Conv2d layer of AlexNet 77
Table 7: List of parameters input to MAESTRO 82
Table 8: List of parameters of accelerator_1.m file 82
Table 9: List of values altered in MAESTRO source code 83
Table 10: List of MAESTRO DNN Models 83

Table 11: The dimensions of the last layer for each MAESTRO DNN Model 83
Table 12: List of MAESTRO Mapping files for each DNN Model..... 84
Table 13: Mappings used to create each MAESTRO Mapping file 84
Table 14: List of NTC parameters 86
Table 15: List of FI parameters..... 87
Table 16: List of parameters for inference and training..... 87
Table 17: List of parameters for the voltage island granularity analysis 88
Table 18: Best Power/Energy efficiency gains and worst Performance loss of layers for AlexNet 93
Table 19: Power/Energy efficiency gains and Performance loss of 1st layer per DNN Model 93
Table 20: Best results for Power/Energy consumption and the related performance in NTC for all layers of different DNN Models 95
Table 21: Difference (%) between NTC and STC regime for the total of DNN Models..... 95
Table 22: The measured accuracy of different DNN Models during bit-flip for various cluster_size$_{NTC}$ and F_{NTC}..... 99
Table 23: The experimental results for AlexNet in NTC with/without Fault Injection 102
Table 24: The scheme of F_{NTC}, $V_{dd, NTC}$, num_PEs and cluster_size used for our research 104
Table 25: The experimental results NTV-DNN under different dataflows..... 105
Table 26: The scheme used when exploring the effect of V_{th} variability..... 106
Table 27: The experimental results exploring the effect of V_{th} variability in NTC ($V_{dd, NTC}$ = 0.65V)..... 107

ACRONYMS

ABS	Absolute
AC	Approximate Computing
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
ANN	Artificial Neural Network
ASIC	Application-Specific Integrated Circuit
BCR	Boost Control Register
BCU	Boost Control Unit
BN	Batch Normalization
BRAM	Block Random Access Memory
CIFAR-10	Canadian Institute For Advanced Research-10
CLB	Configurable Logic Block
CNN	Convolutional Neural Networks
Conv	Convolution
CPI	Cycles Per Instruction
CPU	Central Processing Unit
CU	Control Unit
DC	Densely Connected
DIBL	Drain-Induced Barrier Lowering effect
DNN	Deep Neural Network
DP	Dynamic Power
DSE	Design Space Exploration
DSP	Digital Signal Processing
ECU	Error Sensing Unit
EDP	Energy-Delay-Product
ELT	Error Log Table
FC	Fully Connected
FI	Fault Injection
FMA	Fused Multiply-Add
Fmap	Feature map
FPEC	Fixed Point with Error Compensation
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit

GBM	Global Buffer Memory
GPU	Graphics Processing Unit
GPGPU	General Purpose GPU
IFM	Input Feature Map
IS	Input Stationary
LP	Leakage Power
LUT	Look-Up Table
MAC	Multiply-Accumulate
MAESTRO	Modeling Accelerator Efficiency via Spatio-Temporal Resource Occupancy
MLD	Multi-cycle Latency Datapath
MLP	Multi-Layer Perceptron
NOC	Network On Chip
NTC	Near-Threshold Computing
NTV-DNN	Near-Threshold Voltage DNN framework
OFM	Output Feature Map
OS	Output Stationary
PE	Processing Element
PSUMs	Partial Sums
PVs	Process Variations
RELU	Rectified Linear Unit
RF	Register File
RL	Reinforcement learning
RNN	Recurrent Neural Networks
RS	Row Stationary
SAS	Self-Adaptive Sprint
SeMU	Sequence Monitor Unit
SIMD	Single Instruction, Multiple Data
SL	Supervised learning
SMT	Simultaneous Multithreading
SMs	Streaming Multiprocessors
SQRT	Square root
TECU	Timing Error Control Unit
TOPS	TeraOps per Second
TPU	Tensor Processing Unit
TTQ	Trained Ternary Quantization

UB	Unified Buffer
UL	Unsupervised learning
VI	Voltage Island
VR	Voltage Regulator
WS	Weight stationary

ABSTRACT

Artificial Intelligence (AI) evolution is accelerating, and Deep Neural Network (DNN) inference is at the forefront of computing architectures that are evolving to support the immense throughput required for AI computation. However, much more energy efficient design paradigms are inevitable to realize the complete potential of AI evolution and curtail energy consumption. The coordination of Approximate Computing (AC) together with Near-Threshold Computing (NTC) design paradigm can serve as the best candidate for providing the required energy efficiency. The scope of this diploma thesis is to explore and analyze the impacts of AC and NTC principles in modern multi-/many-core architectures eventually proposing NTV-DNN, a fine-tuned microarchitecture paradigm for energy efficient DNN inference.

Keywords: Artificial Intelligence (AI), Deep Neural Network (DNN), Near-Threshold Computing (NTC), Approximate Computing (AC), Tensor Processing Unit (TPU), Energy efficiency, Microarchitecture, Inference

ΠΕΡΙΛΗΨΗ

Η εξέλιξη της Τεχνητής Νοημοσύνης (AI) επιταχύνεται και η κανονική λειτουργία των Βαθιών Νευρωνικών Δικτύων (DNN), βρίσκεται στην πρώτη γραμμή των υπολογιστικών αρχιτεκτονικών που εξελίσσονται, για να υποστηρίξουν τον τεράστιο ρυθμό διαμεταγωγής (throughput) που απαιτείται για τους υπολογισμούς AI. Ωστόσο, είναι αναπόφευκτη η απαίτηση εύρεσης πολύ πιο ενεργειακά αποδοτικών προτύπων σχεδιασμού, για την αξιοποίηση του πλήρους δυναμικού της εξέλιξης της τεχνητής νοημοσύνης και τον περιορισμό της κατανάλωσης ενέργειας. Ο συντονισμός του Approximate Computing (AC) μαζί με το μοντέλο σχεδίασης Near-Threshold Computing (NTC), μπορεί να χρησιμεύσει ως ο καλύτερος υποψήφιος για την παροχή της απαιτούμενης ενεργειακής απόδοσης. Το αντικείμενο αυτής της διπλωματικής εργασίας είναι να διερευνήσει και να αναλύσει τον αντίκτυπο των αρχών AC και NTC σε σύγχρονες αρχιτεκτονικές πολλαπλών/πολλών πυρήνων προτείνοντας τελικά το NTV-DNN, ένα βελτιωμένο παράδειγμα μικροαρχιτεκτονικής για ενεργειακά αποδοτική κανονική λειτουργία DNN.

Λέξεις Κλειδιά: Τεχνητή Νοημοσύνη (AI), Βαθιά Νευρωνικά Δίκτυα (DNN), Ενεργειακή απόδοση, Μικροαρχιτεκτονική, Κανονική λειτουργία DNN (inference), Τάση κατωφλίου

1 INTRODUCTION

1.1 The Deep Neural Network Challenge

Deep neural networks (DNNs) currently form the basis for many modern artificial intelligence (AI) applications and have become extraordinarily popular. Since the breakthrough of DNNs in speech and image recognition (see Figure 1), the number of applications using DNNs has exploded. These DNNs are used in a wide variety of applications, from self-driving cars to cancer detection to playing complex games. In many of these areas, DNNs are now able to outperform human accuracy. The superior accuracy of DNNs stems from their ability to extract high-level features from raw sensory data by applying statistical learning to a large amount of data to obtain an effective representation of an input space. This differs from previous approaches that use hand-crafted features or rules developed by experts.

However, DNNs superiority in accuracy comes at the cost of high computational complexity. This leads to the design of more specialized hardware which gives rise to the need of improving compute performance and energy efficiency [1]. Until today, there has been tremendous interest in enabling efficient processing of DNNs. Some of the challenges we face for DNN acceleration are the following:

- The achievement of high performance and efficiency (e.g., energy).
- To provide sufficient flexibility to cater to a wide and rapidly changing range of workloads.
- To integrate well into existing software frameworks.

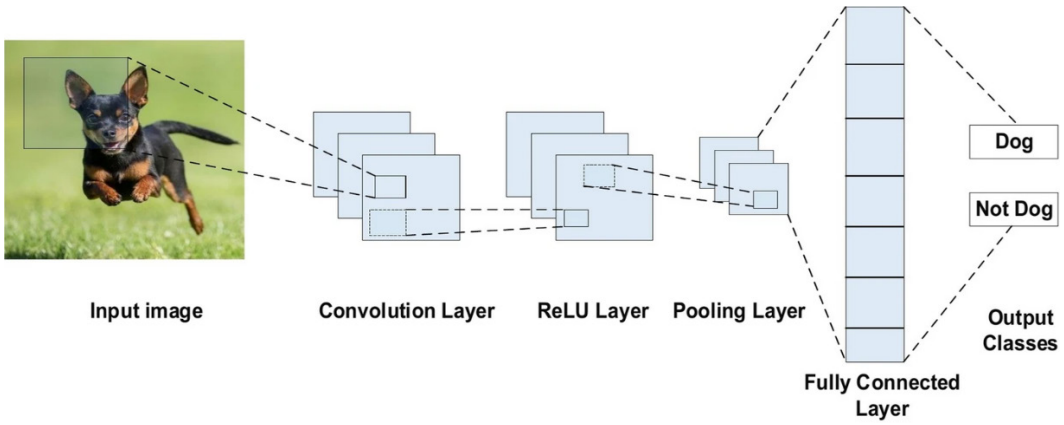


Figure 1: An example of DNN used to image recognition [2]

1.2 Motivation

While AI evolution is accelerating, computing architectures are also evolving to support the immense throughput required for AI computation. However, DNN training and inference requires more and more energy consumption which leads to the need of much more energy efficient design paradigms to realize the complete potential of AI evolution. Till today, vast research has been made in the fields of Approximate Computing (AC) and Near-Threshold Computing (NTC) design paradigms that could serve as the best candidates for providing the required energy efficiency. Nevertheless, the is not enough exploration in the field of combining AC and NTC to gain the best of power and energy relation (see Figure 2).

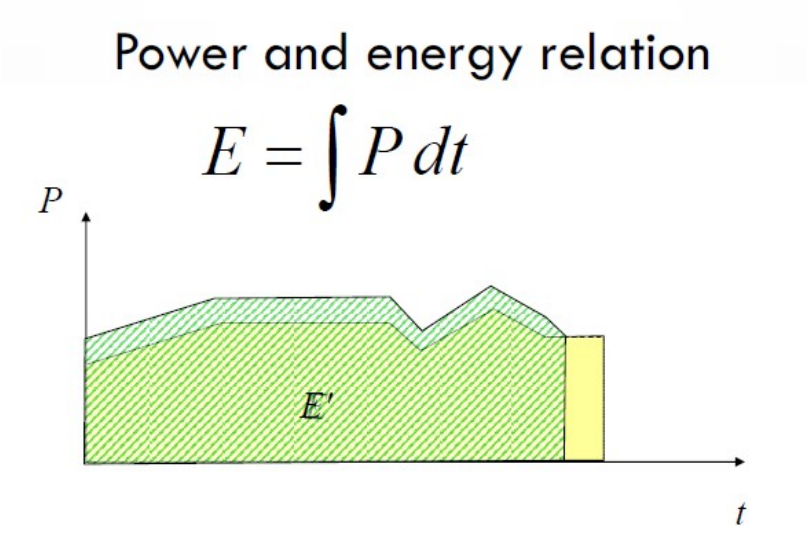


Figure 2: Power and energy relation [3]

1.3 Benefits of Approximate Near-Threshold Voltage Computing

Employing approximations at the hardware level is a very good design paradigm for achieving high gains in terms of area, power, energy, and performance efficiency [4]. AC has emerged as a new technique which serves to reduce the resources (e.g., design area and power) required to realize digital systems at the expense of a negligible or small amount of reduction in quality-of-results or accuracy. This trade-off between resources and accuracy is especially relevant for a large class of data-rich applications such as machine learning and multimedia processing that offer inherent error resiliency.

On the other hand, NTC operation has potential to improve energy efficiency by an order of magnitude [5]. NTC takes advantage of the quadratic relation between the supply voltage (V_{dd}) and the dynamic power, by lowering the supply voltage of chips to a value only slightly higher than the threshold voltage. At nominal operating voltage, the frequency of operation reduces almost linearly with reduction in the supply voltage, reducing performance linearly, and reducing active energy per operation

quadratically. Leakage power too reduces exponentially, and therefore reducing supply voltage should not only reduce power but also improve energy efficiency. This effect is expected to continue through subthreshold region, providing extreme energy efficiency. However, it peaks near the threshold voltage of the transistor (see Figure 3) and then starts reducing in the subthreshold region. This unexpected reduction in the subthreshold region is explained by the following argument. In the subthreshold region leakage power dominates, and it reduces with voltage but the reduction in frequency is larger than reduction in the leakage power, reducing energy efficiency. Therefore, it is desirable to operate close to the threshold voltage of the transistor for maximum energy efficiency, providing an order of magnitude increased energy efficiency compared to operating at the nominal supply voltage. Subthreshold operation does yield even lower power consumption, but at the expense of reduced energy efficiency, which may be desired in some applications.

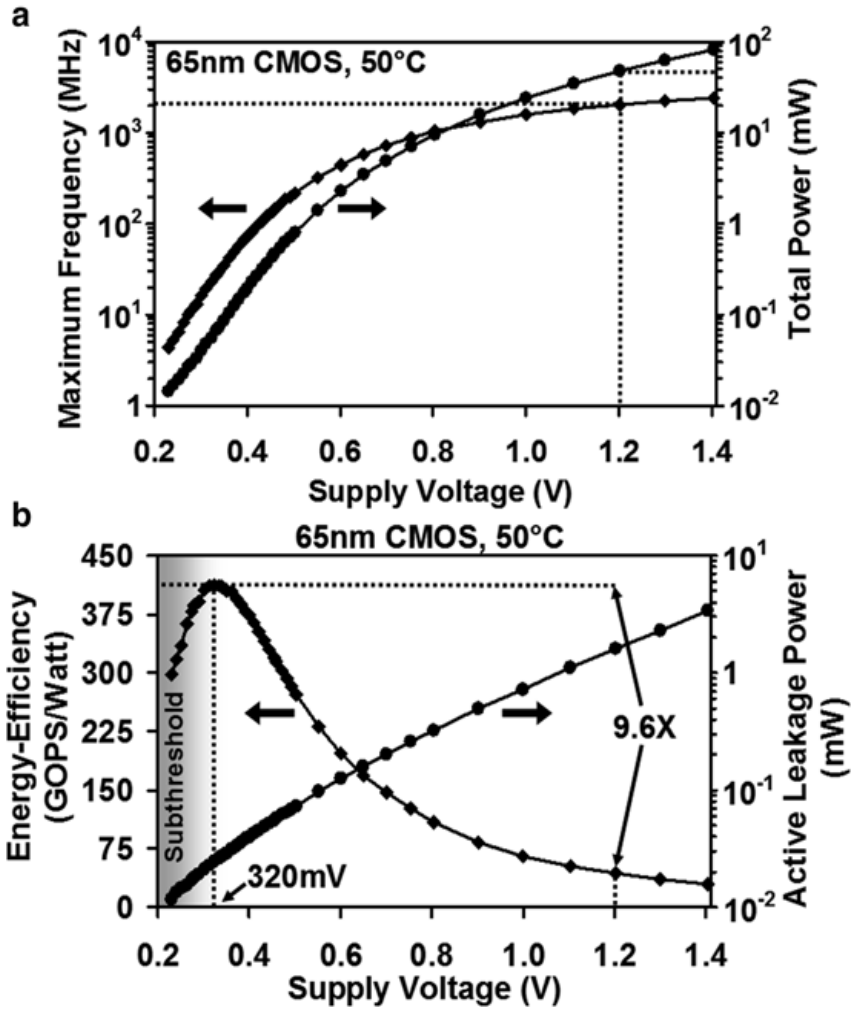


Figure 3: Energy efficiency of NTV operation [5]

1.4 The scope of the thesis

As Moore's law continues to provide designers with more transistors on a chip, power budgets are beginning to limit the applicability of these additional transistors in conventional CMOS design. Furthermore, as technology node shrinks towards 45 nm and below, gate leakage (i.e., leakage current due to direct tunneling) increases owing to the increased electric field (see Figure 4). The scope of this thesis is to study, explore and analyze the impacts of AC and NTC principles in modern multi / many-core architectures and to propose a fine-tuned micro-architecture paradigm for energy efficient DNN inference.

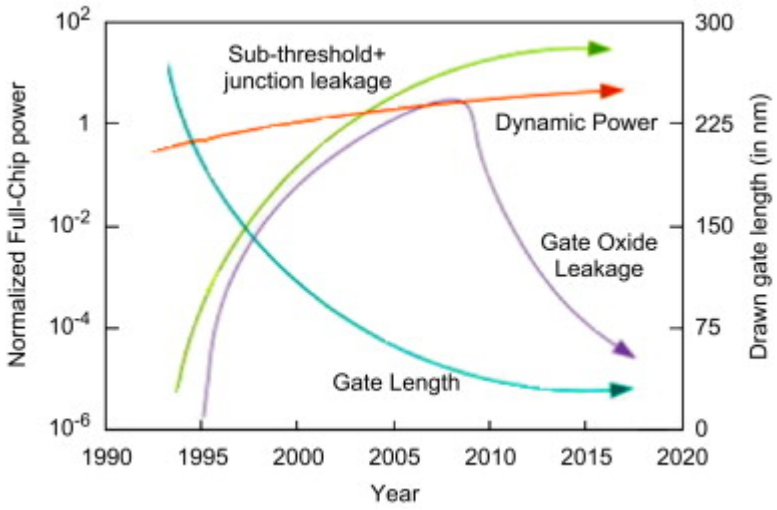


Figure 4: Trends of major sources of power dissipation in nano-CMOS transistor [6]

During our research, as a first step we had to decide which microarchitecture are we going to use during our study. Among CPUs, GPUs and ASICs (e.g. Deep Learning Accelerators-DLAs), we decided to focus on DLAs and specifically on the Google's Tensor Processor Unit (TPU), which is widely used on training and inference of DNNs.

As a second step, we had to choose among different DLA simulators, which is the best to use in our research. We understood that the efficiency (performance and energy efficiency) of a DNN accelerator depends on three factors: 1) the workload (DNN layers), 2) the amount and type of available hardware resources (hardware), and 3) the mapping strategy of a DNN layer on the target hardware (mapping) [57]. This led as to choose MAESTRO, an analytical cost model which receives DNN model description and hardware resources information as a list, and mapping described in a data-centric representation, and generates more than 20 statistics including total latency, energy, throughput, etc., as outputs.

Then, as a third step we focused on finding, through a thorough study, the appropriate techniques for AC and NTC so as to implement our Near-Threshold Voltage DNN framework (NTV-DNN) tool for early assessment of energy at various voltage variation levels. For the part of NTC, we adopted a

promising technique proposed from I. Stamelakos, S. Xydis, G. Palermo et al. [60] based on the on the formation of voltage islands (VIs) for the minimization of the impact of within-die variations, which are more evident at NTC, in both performance and power. As for AC, we decided to work with a runtime fault injector (PyTorchFI [59]) with the scope to test the resilience of our examined DNN Models in errors and how this affects accuracy and performance in an NTC regime. Finding the correct FI tool was not easy, as most of them were deprecated (e.g., Ares Fault Injection Framework). Finally, we had to choose among a wide range of DNN Models, the ones that will be examined during our research, according to different workloads and a dataset (CIFAR-10) for their training and inference.

Our experiments depicted significant reductions in power and energy consumption of about 80% and 50% respectively, for a range of $V_{dd,NTC}$ supply voltage between 0.6V to 0.65V for the total of our simulated 16 x 16 TPU-based accelerator, but with a cost of about 90% reduction in execution time and 3% reduction in accuracy. In addition, we have proven that for a TPU-based accelerator with a total of 256 PEs that works in STC, there is an equivalent accelerator with a double size of PEs and with similar performance that works in NTC which shows gains of about 50% and 52% in power and energy consumption respectively. We also concluded that, in terms of energy efficiency and performance, choosing the right dataflow strategy plays a crucial role for the designing of energy efficient DLAs.

2 BACKGROUND ON DEEP NEURAL NETWORKS

In this chapter, we present how DNNs are positioned in the context of artificial intelligence (AI) and how training and inference works. We also describe the different types of layers of a DNN and finally we focus on Convolutional Neural Networks (CNNs) which are placed at the frontend of Deep Learning.

2.1 What is Deep Learning?

Deep learning, which is also referred as DNN, is a subset of AI. AI is the science and engineering of building intelligent machines that could achieve goals like humans do, according to John McCarthy, who is considered the father of AI [7]. The relationship of deep learning to the whole of AI can be seen in Figure 5.

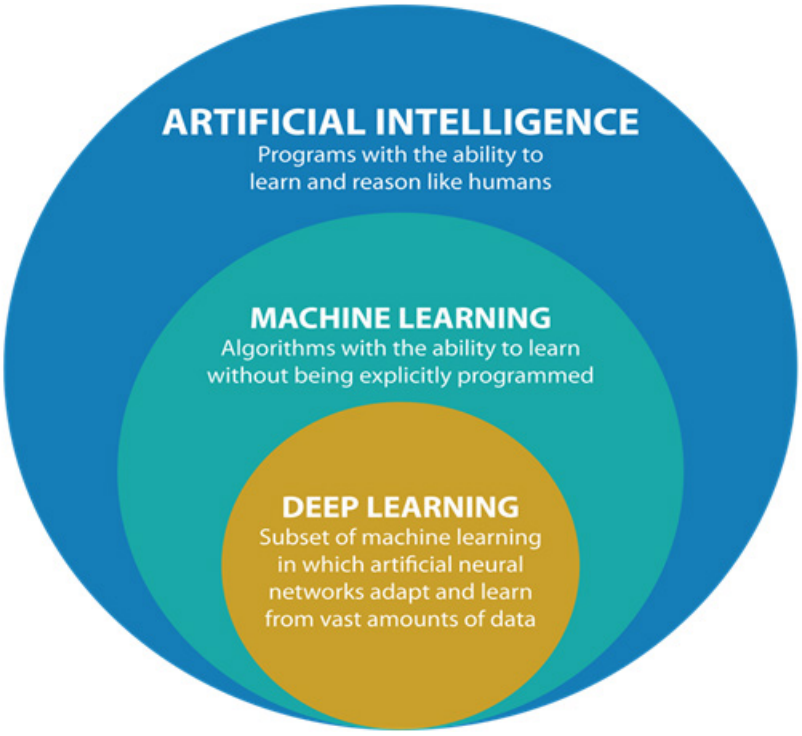


Figure 5: Deep Learning as a subset of AI [8]

An Artificial Neural Network (ANN) is made of inputs and outputs, which are organized into layers. These layers can be distinguished into three types: an input layer, a hidden layer(s), and an output layer [9]. Each of these layers is made up of smaller units called neurons, which are the fundamental computational units of the network for performing a specific task. A deep neural network or deep learning is essentially an ANN with many hidden layers, where the term “deep” refers to extra

layers [10]. Thus, a DNN is a network composed of multiple-computational layers. This involves numerous simple computations on a weighted sum of the input values.

Based on the internal structure of the network, there are several architectures for DNNs, including Multi-layer Perceptron (MLP) and Convolutional Neural Networks (CNNs). These two types are considered as the basis of the DNNs and the most used types. They have also received most of the attention, both in research and industry. Consequently, MLP and CNN are currently the backbones of deep learning. A DNN should have inputs, neurons, layers, activation functions, multiply-sum operations, loss functions, parameters, and a specific topology for being a network [9]. Indeed, DNNs are models created with linear algebra at their cores, and then later optimized with calculus (i.e., learning process). As a result, DNNs are fundamentally a chain of matrix operations applied to input data and a set of parameters required to map the output to the input.

As we understand, the basic operation in DNNs (e.g., CNN and MLP) is a series of matrix multiplications. Specifically, as seen in Figure 6, each neuron receives some inputs and performs a convolutional operation between the input and its weights (i.e., multiply-and-sum). Then, it adds the biases to the intermediate output to obtain the activation (i.e., g) before passing it to an activation function (i.e., F(g)) for non-linearity, which eventually gives the final output of this neuron. The mathematical equations that describe neuron's function are:

$$g = \sum_{j=1}^{n} \left(w_j x_j + b \right) \quad , \quad Y = F(g) \qquad (1)$$

where w is the weight, x is the input and b is the bias. Therefore, a compulsory operation in any DNN is a multiply-accumulate (MAC) operation, suitable for all kinds of matrix operations. Although it needs a large amount of data to be performed, MAC is the primary and most important operation in DNNs [9].

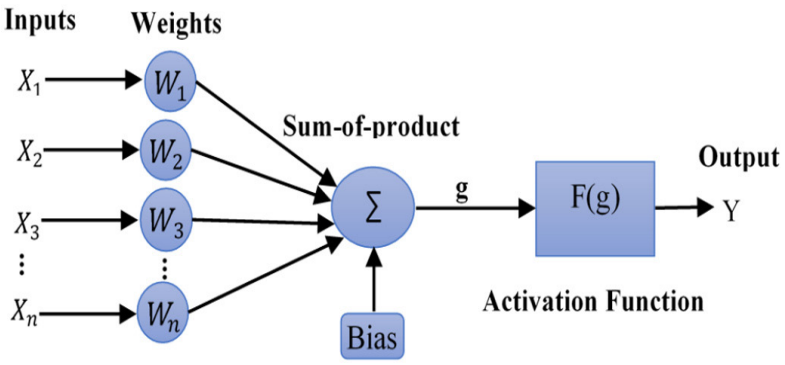
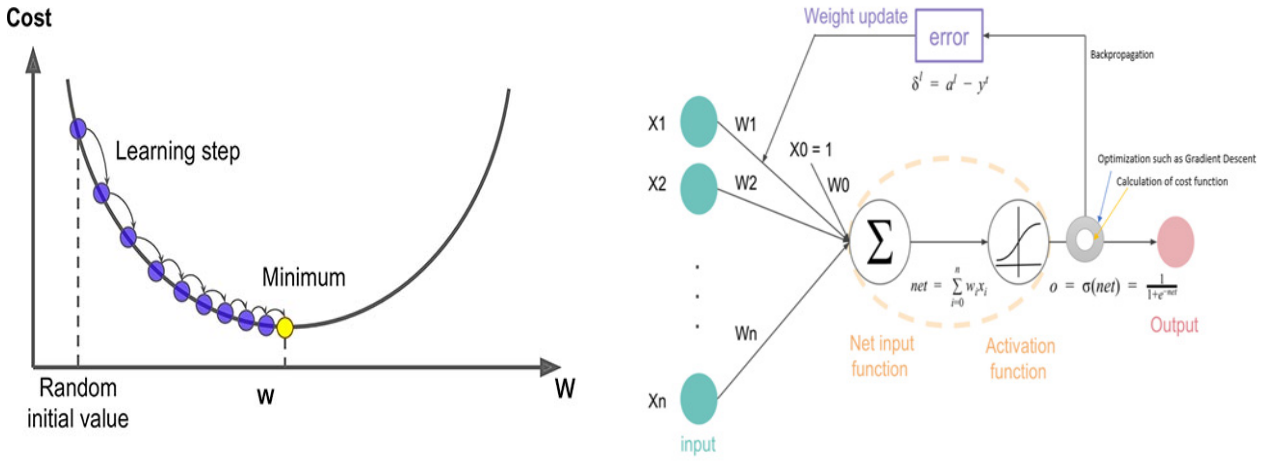


Figure 6: A single neuron in a DNN and its main structure [9]

2.2 Training and Inference

Since DNNs are an instance of machine learning algorithms, the basic program does not change as it learns to perform its given tasks. In the specific case of DNNs, this learning involves determining the value of the weights (and biases) in the network and is referred to as *training* the network. Thus, the goal of training DNNs is to find a set of weights to minimize the average loss over a large training set. Once trained, the program can perform its task by computing the output of the network using the weights determined during the training process. Running the program with these weights is referred to as *inference*.

When a neuron is activated, it calculates a function of all data it has, and compares the value of this function with a threshold value (activation function) which is characteristic of this neuron. If the value of the function is greater than the threshold value, then the neuron calculates the output, which forwards as input to the next (or next) neuron (s). During training the only thing that changes is the values of weights connections of neurons. When training a network, the weights $\left(w_{ij} \right)$ are usually updated using a hill-climbing (hill-descending) optimization process called gradient descent. In gradient descent, a weight is updated by a scaled version of the partial derivative of the loss with respect to the weight. Note that this gradient indicates how the weights should change to reduce the loss. The process is repeated iteratively to reduce the overall loss. An efficient way to compute the partial derivatives of the gradient is through a process called backpropagation. Backpropagation (see Figure 7(b)) operates by passing values backward through the network to compute how the loss is affected by each weight.



(a) (b)

Figure 7: Gradient descent (a) and backpropagation (b) [11][12]

Changes in weight values are not always made with the same way, but it depends a lot on the method we use. The three basic methods of training are described below:

- *Supervised learning (SL)*: SL occurs when we start with random values for weights, and we know the values of the inputs and the targets that the network must learn. During the training process, the network changes the values of the weights correcting them depending on the error we get (difference from the target). The learning process stops when the algorithm achieves an acceptable level of performance.

- *Unsupervised learning (UL)*: In problems in this category, training data are vectors that do not have corresponding labels. Therefore, the goal of UL is to find patterns when there are no "correct answers", or when they are impossible to calculate. A large subcategory of unsupervised tasks is the problem of clustering. Grouping refers to grouping observations in such a way that the members of a common group are similar to each other and differ significantly by members of other groups. Another very interesting category of unsupervised tasks are genital models. These models mimic the process of creating training data. A good genital model should be able to create new data which, although artificial, looks like the original. This way of learning is unsupervised because the process by which data is created ("born") is not immediately observable - only the data itself is observable.

- *Reinforcement learning (RL)*: The essence is learning through interaction. An RL agent interacts with its environment and, upon observing the consequences of its actions, can learn to alter its own behavior in response to rewards received. In the RL set-up, an autonomous agent, controlled by a machine learning algorithm, observes a *state* s_t from its *environment* at timestep *t*. The agent interacts with the environment by taking an action a_t in state s_t. When the agent takes an action, the environment, and the agent transition to a new state s_{t+1} based on the current state and the chosen action. The state is a sufficient statistic of the environment and thereby comprises all the necessary information for the agent to take the best action, which can include parts of the agent, such as the position of its actuators and sensors [13].

After training is completed, the networks are deployed into the field for inference (e.g., classifying data to “infer” a result). With inference you’ll get almost the same accuracy of the prediction, but simplified, compressed, and optimized for runtime performance. What that means is we all use inference all the time. Your smartphone’s voice-activated assistant uses inference, as does Google’s speech recognition, image search and spam filtering applications. Baidu also uses inference for speech recognition, malware detection and spam filtering. Facebook’s image recognition and Amazon’s and Netflix’s recommendation engines all rely on inference [14].

2.3 Types of Layers

In this section, we present the various popular layers from which DNNs are formed. We begin by describing the Convolution (Conv) and Fully Connected (FC) layers whose main computation is a weighted sum, since that tends to dominate the computation cost in terms of both energy consumption and throughput. We will speak for various layers that can optionally be included in a DNN and do not use weighted sums such as nonlinearity, pooling, and normalization.

2.3.1 Conv Layer

The CONV layer works by sliding many small filters across an image to extract meaningful features. Figure 8 displays the data structure of a CONV operation. The inputs to CONV layer are N feature maps (fmaps). Every fmap is convolved by a shifting window with a $R \times S$ kernel, which produces one pixel in one output fmap. The shifting window has a stride of S which is generally smaller than R. The N output fmaps are taken as the input fmaps for the next CONV layer [15].

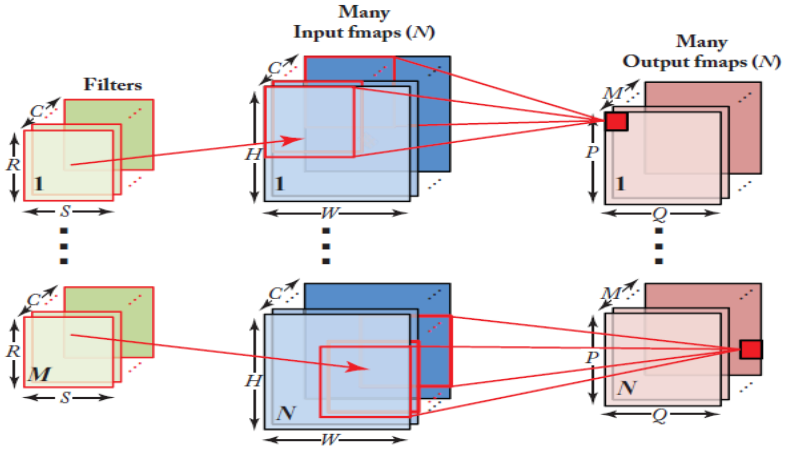


Figure 8: Illustration of a Conv operation [1]

Table 1 displays the shape parameters of a Conv Layer, the computation of which is defined as:

$$o[n][m][p][q] = \left(\sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} i[n][c]\left[U_p + r\right]\left[U_q + s\right] \times f[m][c][r][s] \right) + b[m],$$

$$0 \le n \le N,\ 0 \le m \le M,\ 0 \le p \le P,\ 0 \le q \le Q,\ P = \frac{(H - R + Q)}{U},\ Q = \frac{(W - S + U)}{U}. \quad (2)$$

where o, i, f, b are the tensors of the output fmaps, input fmaps, filters and biases. U is a given stride size.

Table 1: Shape parameters of a Conv Layer [1]

Shape parameter	Description
N	Batch size
M	Number of channels of output fmaps (output channels)
C	Number of channels of filter / input fmaps (input channels)
H/W	Input fmap spatial height/width
R/S	Filter spatial height/width
P/Q	Output fmap spatial height/width

2.3.2 FC Layer

Fully Connected (FC) Layers contain neurons that apply a linear transformation to the input vector through a weight's matrix. Every value in the output fmap is the result of a weighted sum of every input value in the fmap. FC layers do not support weight sharing thus making each calculation memory bounded. A nonlinear transformation is then applied to the result as shown below:

$$g = f\left(\sum_{t=0}^{n} w_i x_i + b\right)$$

where x is the input vector, w is the weight's vector, b is the bias and f is the non-linear activation function. Figure 9 displays an example of a fully connected network in which the green neurons represent the input, the blue neurons belong to the hidden layer and the red neurons represent the output.

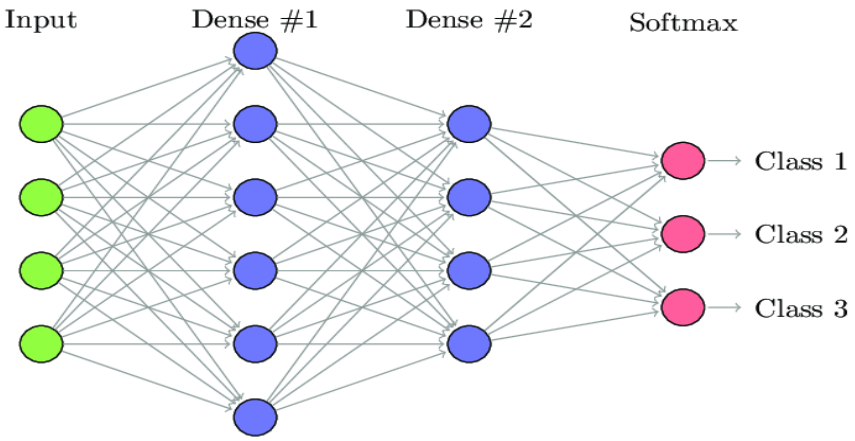


Figure 9: A Fully Connected Network [16]

In the above Figure we can understand why we call these kinds of layers Fully Connected or sometimes *Densely Connected* (DC). All possible connections layer to layer are present, meaning every input of the input vector influences every output of the output vector. An FC network is very useful for the work of *Classification* in a DNN. As we see in Figure 9 the hidden FC layer serves to classify the input data into various classes.

2.3.3 Nonlinearity

After a Conv or FC Layer, a nonlinear activation function is applied. There are various nonlinear activation functions, some of which are displayed in Figure 10. Among them the Rectified Linear Unit (RELU) is considered the most popular due to its simplicity and its ability to enable fast training, while achieving comparable accuracy [1]. The other variants of ReLU include Leaky ReLU, ELU, SiLU, etc., which are used for better performance in some tasks and show improved accuracy. The ReLU activation function is differentiable at all points except at 0. For values greater than 0, we just consider the max of the function [17]. This can be described as below:

```
if input > 0:
    return input;
else:
    return 0;
```

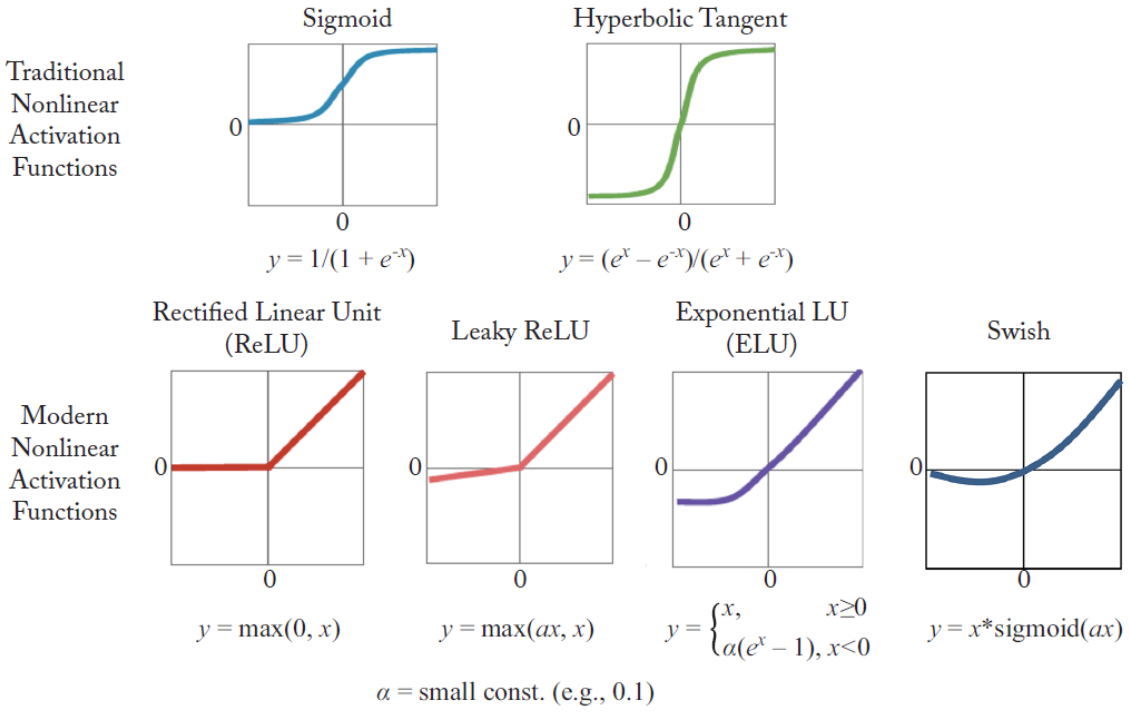


Figure 10: Various nonlinear activation functions [1]

Maxout is also a very promising nonlinearity which takes the maximum value of two intersecting linear functions and has shown to be very effective in speech recognition tasks [18]. The following function implements maxout:

$$h(x) = max\left(Z_1, Z_2, \ldots, Z_n\right) = max\left(W_1 \cdot x + b_1, W_2 \cdot x + b_2, \ldots, W_n \cdot x + b_n\right) \quad (3)$$

2.3.4 Pooling and Unpooling

Pooling is a method that focuses in getting a DNN to focus on higher-level features. In a convolutional neural network, pooling is usually applied on the fmap produced by a preceding convolutional layer and a non-linear activation function. During pooling a filter is selected which slides over the output fmap of the preceding convolutional layer. The most used filter size is 2×2 and it is slid over the input using a stride of 2. Based on the type of pooling operation you've selected, the pooling filter calculates an output on the receptive field (the part of the feature map under the filter) [19]. The most commonly used approaches are the following:

- *Max Pooling*: In this approach the filter simply selects the maximum pixel value in the receptive field. For example, as in Figure 11, if you have 4 pixels in the field with values 5, 3, 9 and 28, you select 28.

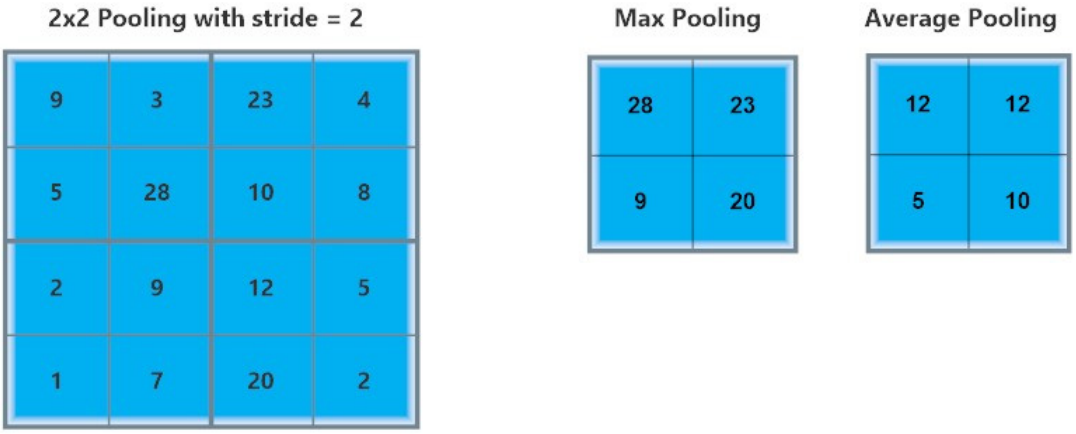


Figure 11: Max Pooling and Average Pooling Methods

- *Average Pooling*: Average pooling works by calculating the average value of the pixel values in the receptive field. Given 4 pixels with the values 2, 9, 1 and 7 the average pooling layer would produce an output of 4.75. As seen in Figure 11, rounding to full numbers gives us 5.

Unpooling or more generally *upsampling* is the method during which the spatial resolution of a fmap is increased. A commonly used form of unpooling method is to insert zeros between the activations as shown in Figure 12. Another method, also displayed in Figure 12, is interpolation with the use of nearest neighbors. Upsampling introduces structured sparsity in the input fmap and is generally used before a Conv or FC layer. It is an important tool that can improve energy efficiency and throughput [1].

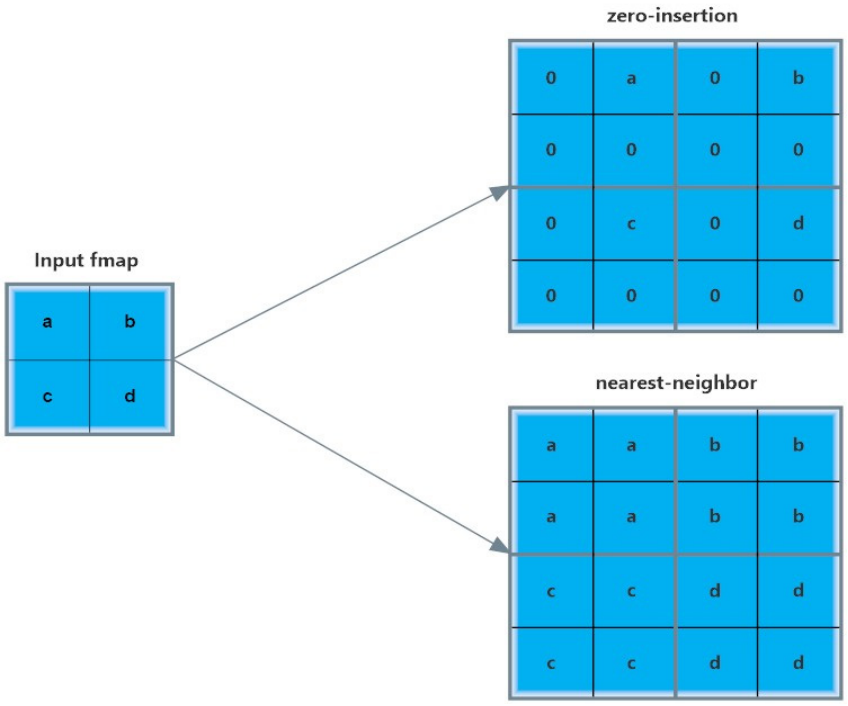


Figure 12: Zero-insertion and Nearest-neighbors Unpooling Methods

2.3.5 Normalization

When talking about *normalization*, we refer to the operation during which, given a set of data $\mathbb{D}=\{x^{(i)}\}_{i=1}^{N}$, a normalization function $\Phi: x \mapsto \hat{x}$ ensures that the transformed data $\widehat{\mathbb{D}}=\{\hat{x}^{(i)}\}_{i=1}^{N}$ has certain statistical properties [20]. By this method we can help to significantly speed up training and improve accuracy. In general terms normalization is a pre-processing technique used to standardize data. In other words, having different sources of data inside the same range [21]. If we do not normalize the data before training that can cause problems in our network, making it drastically harder to train and decrease its learning speed. We can distinguish two methods to normalize our data:

- Scale the data set to a range from 0 to 1:

$$x_{normalized}=\frac{x-m}{x_{max}-x_{min}} \qquad (4)$$

  where x is a data point, m is the mean of the data set, x_{max} is the highest value and x_{min} is the lowest value. This technique is generally used in the inputs of the data.

- Forcing the data points to have a mean of 0 and a standard deviation of 1:

$$x_{normalized}=\frac{x-m}{s} \qquad (5)$$

  where x is a data point, m is the mean of the data set and s is the standard deviation of the data set.

Batch Normalization (BN) is a wide adopted method in the design of CNNs and is usually performed between the Conv or FC layer and the nonlinear function. The normalization formula is the following:

$$z^N = \left(\frac{z - m_z}{s_z} \right) \qquad (6)$$

where z^N is the BN output, z is the neuron's output (before normalization), m_z is the mean of the neuron's output and s_z is the standard deviation of the output of the neurons. Figure 13 displays how normalization works. As we see, while the DNN model learns the parameters w and b, BN is added just before the activation function $f(z^N)$.

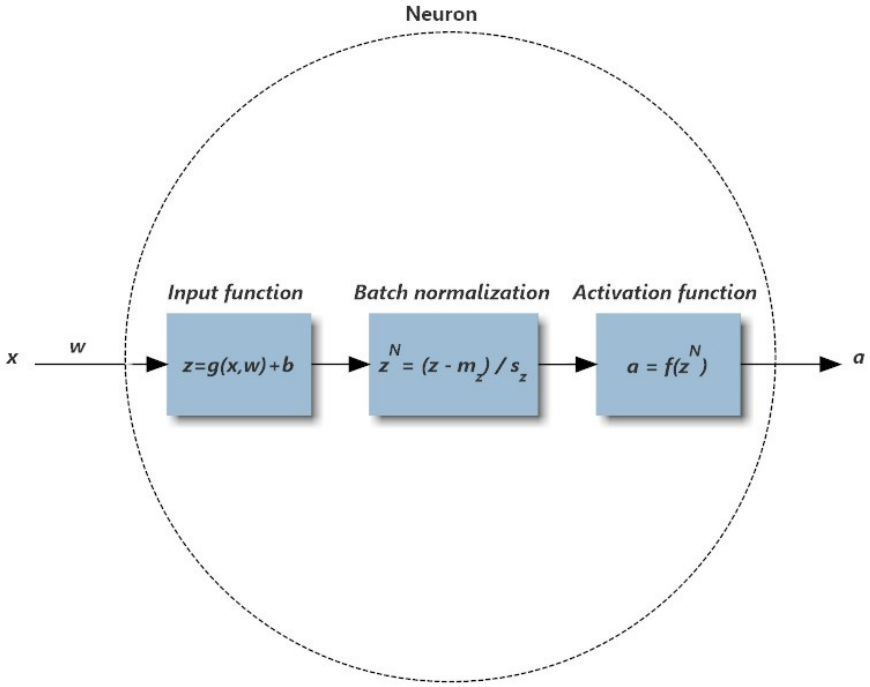


Figure 13: Batch Normalization method

2.3.6 Compound Layers

Compound layers are the result of the combination of all the primitive layers described above. An example of a compound layer is the *up-convolution* layer that combines upsampling (before applying convolution) and transposed convolution [22]. *Attention layer* is also another form of compound layer that is composed of matrix multiplications and feed-forward, fully connected layers [23]. This compound layer is commonly used in a type of DNNs called Transformers and can be useful in processing a wide range of data such as language and images.

2.4 The Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are composed of multiple Conv layers. The function of a CNN is based in the generation of successively higher-level abstracted input data (called feature maps) through each layer as shown in Figure 14 [1]. Each feature map preserves essential yet unique information. During time, there have been developed many CNN models, some of which are displayed in Table 2.

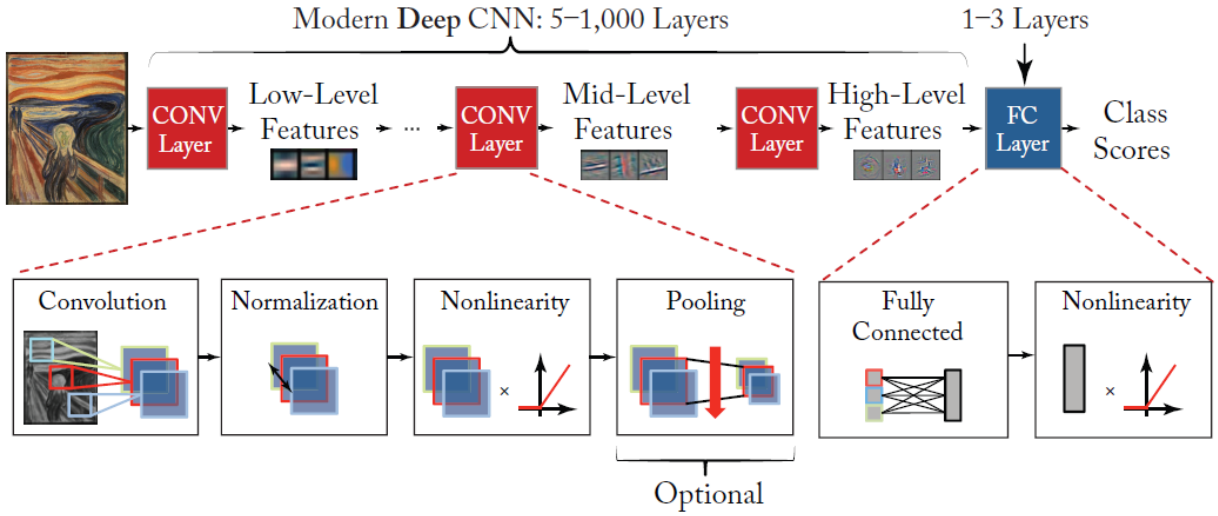


Figure 14: The architecture of a modern deep CNN [1]

Table 2: Different CNN models and their cost [30]

Model	Layer	Parameter [M]	Network size [MB]
AlexNet	8	61	227
GoogLeNet	22	7	27
ResNet-18	18	25.6	96
SqueezeNet	18	1.24	5

AlexNet [24] is considered to be the first CNN that won the ImageNet Challenge in 2012. Its architecture consists of five convolutional layers with a combination of max pooling followed by three FC layers. The ReLU nonlinear activation function is used in each of these layers. The max pooling operation is applied to the outputs of the first, second and fifth Conv layers as displayed in Figure 15. The network has 62.3 million parameters and needs 1.1 billion computation units in a forward pass.

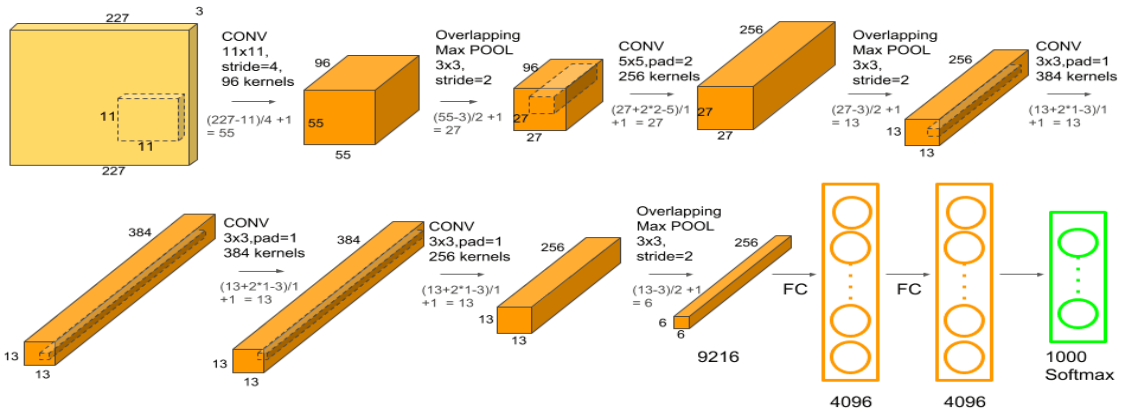


Figure 15: The architecture of AlexNet DNN model [25]

GoogLeNet [26] is a deeper CNN which has 22 layers. The 22 layers consist of three CONV layers, followed by nine inceptions modules (each of which are two CONV layers deep), and one FC layer. An inception module, as displayed in Figure 16, has an input which is distributed through multiple feed-forward connections to several parallel layers. GoogLeNet uses a stack of a total of 9 inception modules and global average pooling to generate its estimates. Max pooling between inception modules reduces the dimensionality.

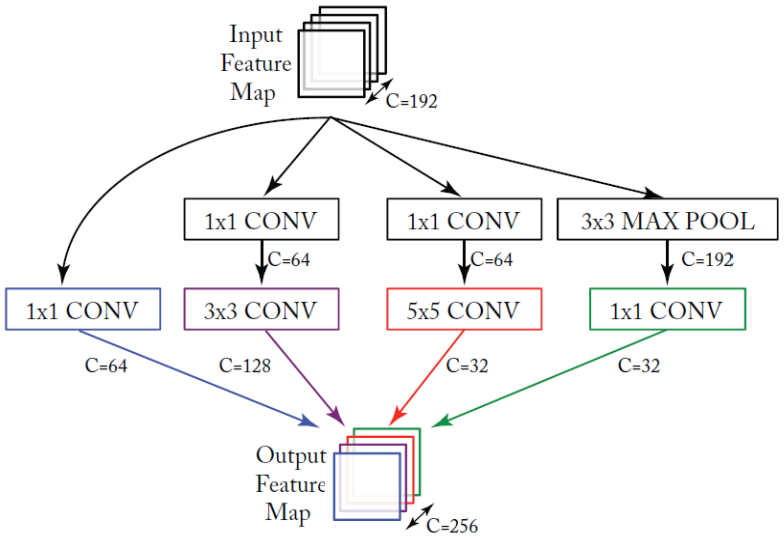


Figure 16: Inception module from GoogLeNet [1]

ResNet [27] is the idea that every additional layer should more easily contain the identity function (see Figure 17) as one of its elements. These considerations are rather profound, but they led to a surprisingly simple solution, a *residual block*. With it, ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015. The design had a profound influence on how to build deep neural

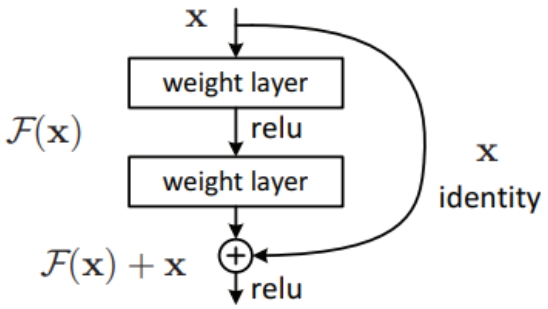


Figure 17: Residual learning: a building block [27]

networks. Figure 18 illustrates the residual block of ResNet, where the solid line carrying the layer input to the addition operator is called a residual connection (or shortcut connection). With residual blocks, inputs can forward propagate faster through the residual connections across layers. ResNet combines 4 convolutional layers in each block together with the first 7×7 convolutional layer and the final FC layer, 18 layers in total. Therefore, this model is commonly known as ResNet-18 [28].

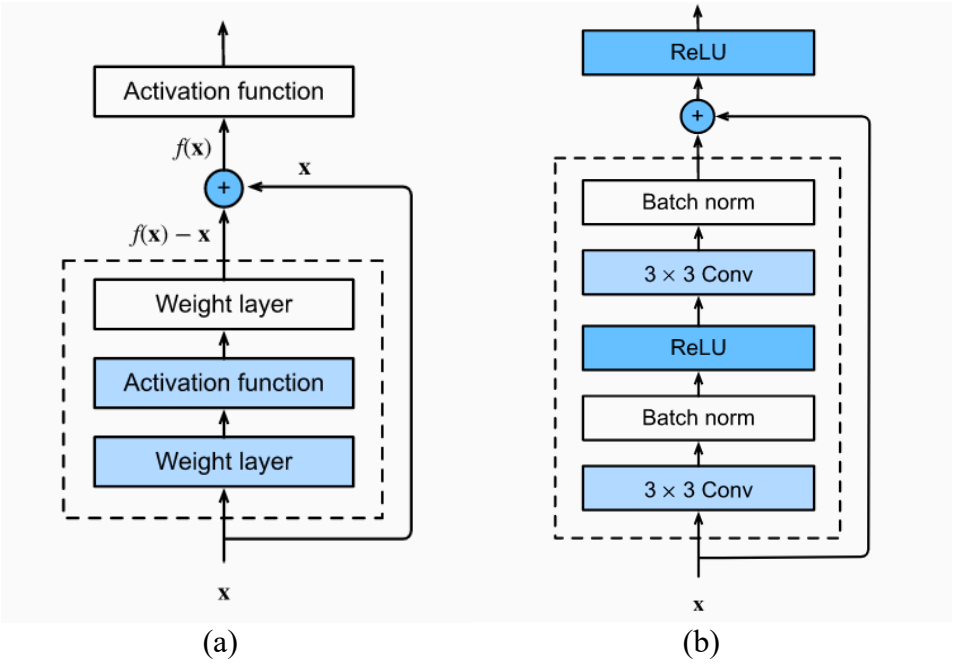


Figure 18: (a) Residual learning: a building block (b) ResNet block [28]

SqueezeNet [29] begins with a standalone Conv layer, followed by 8 *fire modules*, ending with a final Conv layer. We gradually increase the number of filters per fire module from the beginning to the end of the network. The fire module (illustrated in Figure 19) is comprised of: a *squeeze* Conv layer (which has only 1x1 filters), feeding into an expand layer that has a mix of 1x1 and 3x3 convolution filters. ReLU is applied to activations from squeeze and expand layers. With SqueezeNet, we achieve a 50 × reduction in model size and obtain better accuracy results compared to AlexNet model.

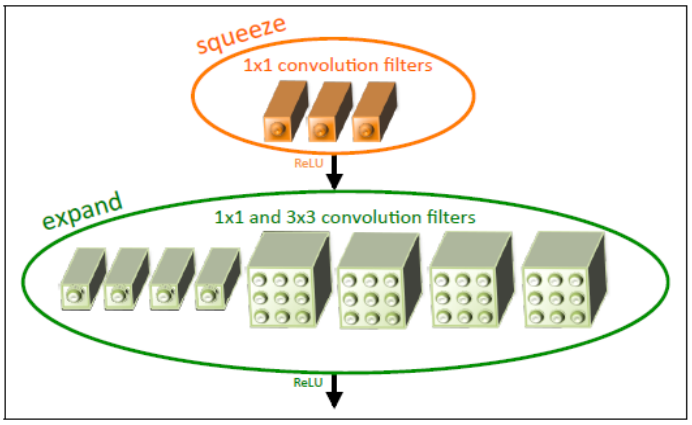


Figure 19: Fire module: SqueezeNet's building block [29]

3 RELATED WORK ON ENERGY EFFICIENT DNN

This chapter is dedicated in presenting the various workloads used for training and inference of DNN workloads. A description of the study that has been done in the domains of Approximate and Near-Threshold Computing in DNNs is also given.

3.1 Architectures for DNN Workloads

When talking for both Conv and FC Layers, the fundamental computation are multiple-and-accumulate (MAC) operations. These operations have negligible dependencies and can be considered as commutative. This fact gives to MAC operations the characteristic of flexibility on how can be scheduled and easily parallelized. Therefore, high parallel computing paradigms are used frequently to achieve high performance for DNNs. We can categorize these architectural paradigms in either temporal or spatial, as we see in Figure 20.

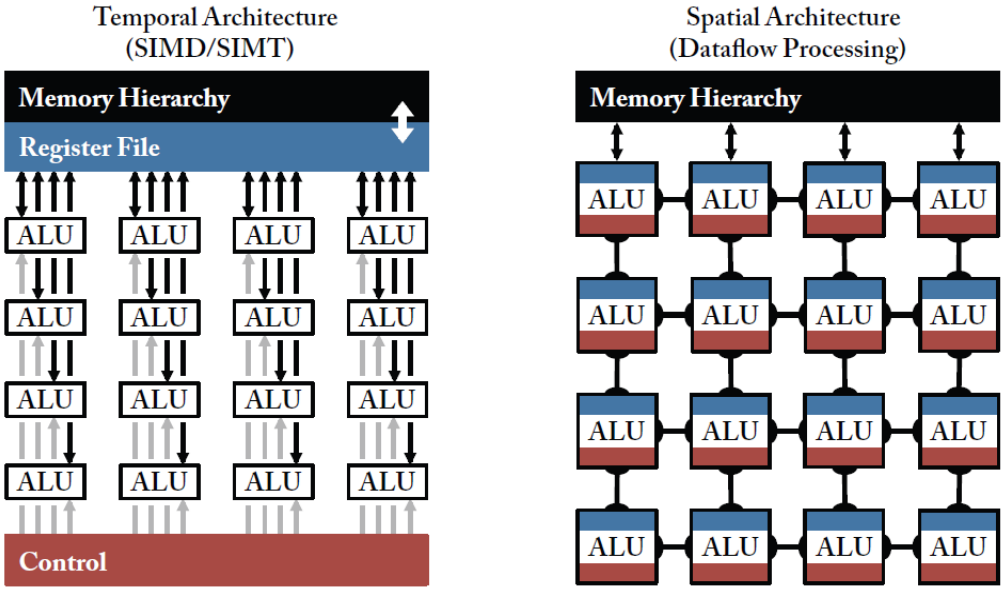


Figure 20: High parallel architectural paradigms [1]

In a temporal architecture, arithmetic logic units (ALUs) fetch data from the memory hierarchy and cannot communicate directly with each other. Central Processing Units (CPUs) and Graphics Processing Units (GPUs) employ such architectures and use a variety of technics to improve parallelism such as vector instructions or parallel threads. On the contrary when we talk about spatial architecture, communication between ALUs is allowed, and the use of dataflow processing is implemented. There are cases that an ALU can have its own control logic and local memory which

is called scratchpad. In this case one or more ALUs form a Processing Element (PE). Special designs of Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) used for the process of DNNs are commonly based on spatial architectures.

3.1.1 CPUs

CPU has been for years the most essential component in computers of any era. It is sometimes simply called a processor and is considered as the brain of a computer. Figure 21 displays a simple block diagram of a uniprocessor-CPU and its building blocks: a Control Unit (CU) that directs the operation of the processor, a Combinational Logic Known such as ALUs and Floating Point Units (FPUs) thar

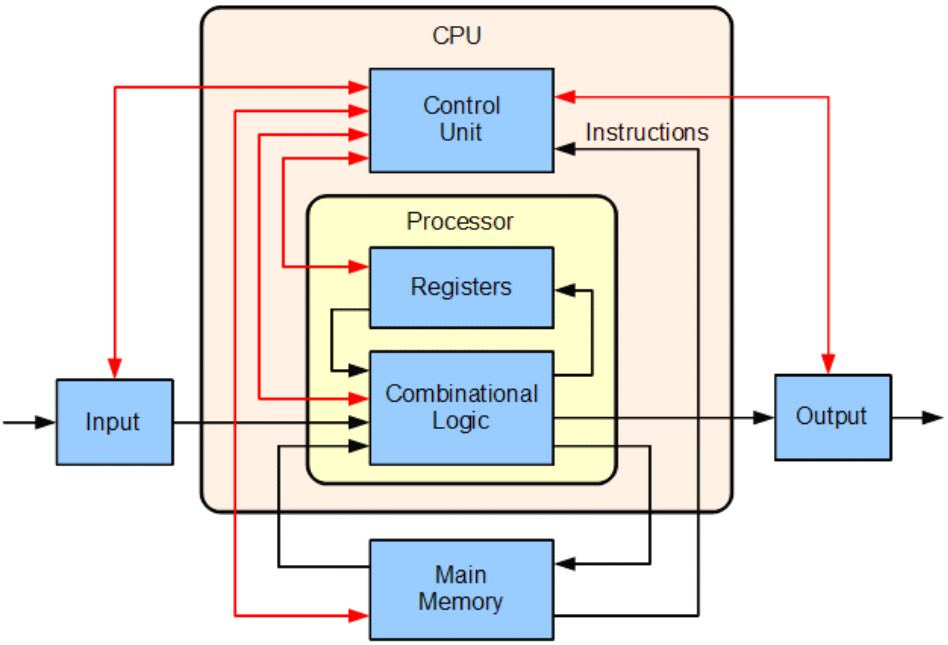


Figure 21: Block diagram of a uniprocessor-CPU [31]

perform the mathematical and logic operations and Registers which provide a quickly accessible location for fast storage. CPU cores have been enhanced to further support parallelism and specific type of computations. Some of these are listed below [32]:

- *Single Instruction, Multiple Data* (SIMD) units: These are hardware components that perform the same operation on multiple data operands concurrently. Typically, a SIMD unit, as seen in Figure 22(a), receives as input two vectors (each one with a set of operands), performs the same operation on both sets of operands (one operand from each vector), and outputs a vector with the results. SIMD operations include arithmetic operations (such as addition, subtraction, multiplication, negation) and other operations such as absolute (abs) and square root (sqrt). The increased performance of SIMD units is based on the fact that multiple data items can be

simultaneously loaded/stored from/to memory exploiting the full width of the memory data bus.

- *Fused Multiply-Add* (FMA) units: As Figure 22(b) depicts, these units perform fused operations such as multiply-add and multiply-subtract. The main idea is to provide a CPU instruction that can perform operations with three input operands and an output result. It is common for an FMA unit to support single, double precision floating-point and integer operations, and depending on the data types, to include a rounding stage following the last operation. Depending on the processor architecture, the input/output of the FMA units might be associated with four distinct registers or three distinct registers, with one register shared between the result and one of the input operands of the FMA unit.

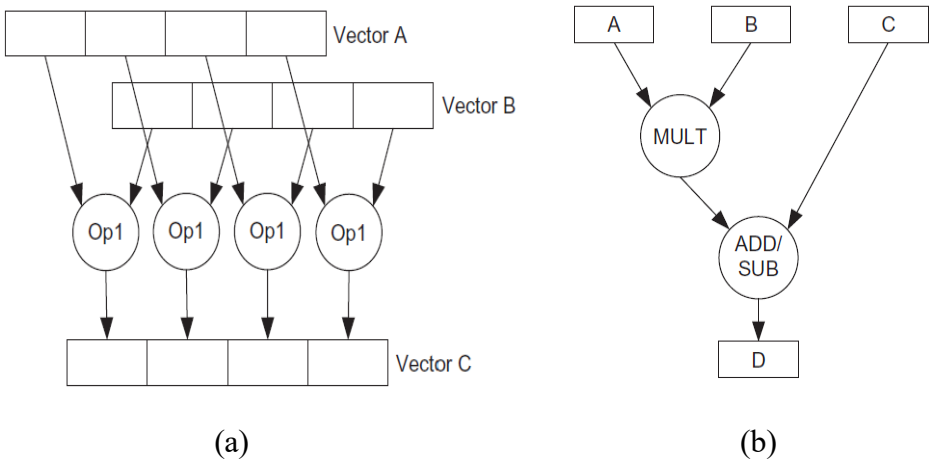


Figure 22: (a) A SIMD unit (b) An FMA unit [32]

- *Simultaneous Multithreading* (SMT): It is a basic characteristic of modern microprocessors that aims to provide multiple cores and to allow native support of parallel thread execution by duplicating hardware in a single core. The execution of multiple threads within the same core is realized by time multiplexing its hardware resources and by fast context switching. An example of such a technology is Intel-Hyper Threading that succeeds to increase processor throughput, improving overall performance on threaded software.

There are various factors for which CPUs are considered to play a crucial role in accelerating DNNs. First of all, they offer a high memory capacity. Since CPU-managed hosts in cloud and datacenter scenarios have much larger memory capacities, running memory hungry operations such as 3D Conv on CPUs is not merely attractive, but often imperative [34, 35]. CPUs are also very useful for medium-parallelism and sparse DNNs as in some workloads such as Recurrent Neural Networks (RNN), the number of computations increases with rising sequence length. However, the

parallelization of RNN is challenging because of the dependencies between the steps and the use of small batch size. Similarly, DNNs such as InceptionNet variants have filter shapes of 1x1, 3x3, 1x3, 3x1, etc., which lead to irregular memory accesses and variable amount of parallelism across the layers. Such applications with limited parallelism fit more naturally to CPUs, which have few fast cores than to GPUs, which have many slow cores [36]. Another advantage is that CPUs are used widely in mobile systems where sometimes can provide similar or higher performance than GPUs. Also, for applications requiring frequent or continuous inference, GPUs may not be most suitable as they can quickly dissipate the battery [37]. Additionally, CPUs remain the processing system of choice for executing DNNs in extreme environments [38].

3.1.2 GPUs

GPUs are considered to be the most prominent DNN accelerators. Its massive parallel architecture and computational power is a big advantage that serves the requirements of DNNs. They have been used for more than a decade in the acceleration of AI algorithms for training and inference [39]. NVIDIA, Intel, and AMD are some of the leading manufacturers that have succeeded in this domain. The internal structure of GPUs is hugely complex. Streaming Multiprocessors (SMs) are the fundamental idea of the parallelism in GPUs. Each SM may have hundreds to thousands of cores, which are the fundamental processing units. Also, the GPUs' memory hierarchy is highly parallelized and shared among various resources. However, this degree of parallelism in GPUs makes them extra vulnerable to faults. Figure 23 displays the block diagram of a high-end GPU-based accelerator where LDST identifies a load/store unit, SFU identifies a special function unit, and Tex identifies a Texture mapping unit [32].

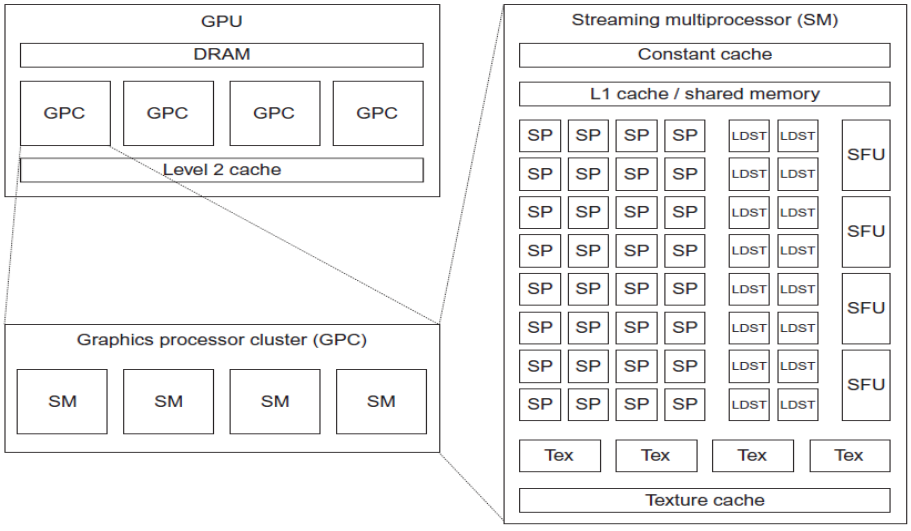


Figure 23: A high-end GPU-based accelerator (NVIDIA Fermi GPU) [32]

GPUs have their limitations, such as high-power consumption, and that is where innovation in other accelerators is emerging. Due to the wide adoption of edge devices that provide mobility features produced by IoT, the power consumption and response time are two substantial reasons that make GPUs unsuitable accelerators in some scenarios. However, GPUs are very useful in high-end scientific and engineering computing where the focus is on high computation throughput.

3.1.3 FPGAs

FPGA technology has become over time a wise choice to accelerate DNN algorithms in certain scenarios. Its main advantages are reconfigurability, versatility, and low-power consumption. An FPGA, as seen in Figure 24, is an array of carefully designed and interconnected digital subcircuits that efficiently implement common functions while also offering very high levels of flexibility. The digital subcircuits are called configurable logic blocks (CLBs), and they form the core of the FPGA's programmable-logic capabilities. Each CLB includes look-up tables (LUTs), storage elements (flip-flops or registers), and multiplexers that allow the CLB to perform Boolean, data-storage, and arithmetic operations. FPGAs also use SRAM or Block RAM (BRAM). BRAMs are small and very fast memories and are more efficient than using LUTs. A big FPGA has nearly 100Mb of BRAM, chained together as needed.

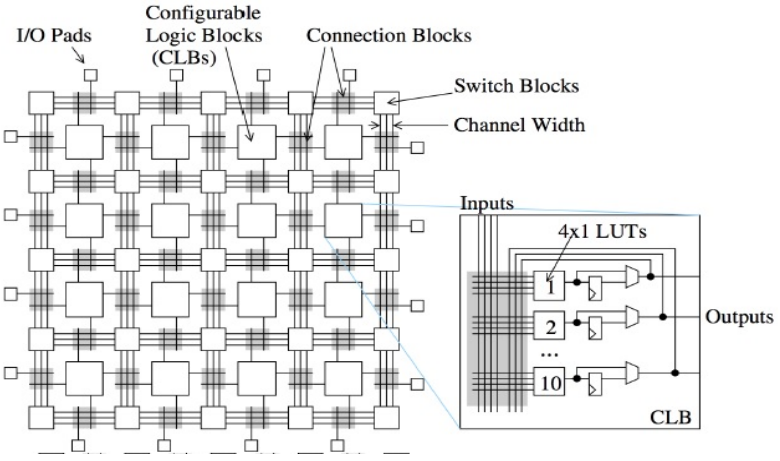


Figure 24: Basic structure of an FPGA [40]

Deep learning algorithms, such as CNNs and Multilayer Perceptrons (MLPs), are executed in FPGAs through specialized analog blocks. CLBs and Digital Signal Processing (DSPs) slices are the basic blocks for the implementation of MAC operations. However, both of these elements are prone to soft errors that can eventually lead to a failure in the DNN model's output, making FPGAs unreliable in processing of DNNs. Another problem is that FGPAs run at low clock frequencies and

are often difficult to deploy and maintain [41].

3.1.4 ASICs

ASIC accelerators can be considered as the most promising and reliable components for accelerating AI algorithms. These processors are explicitly customized to serve one task which cannot be changed over time. The biggest purpose to utilize ASICs for DNN processing is to solve the power constraints imposed by GPUs [42]. Memory accesses is recognized as the key bottleneck in DNN computations and ASICs can use with great success a data reuse pattern to reduce off-chip memory access which makes them superfast.

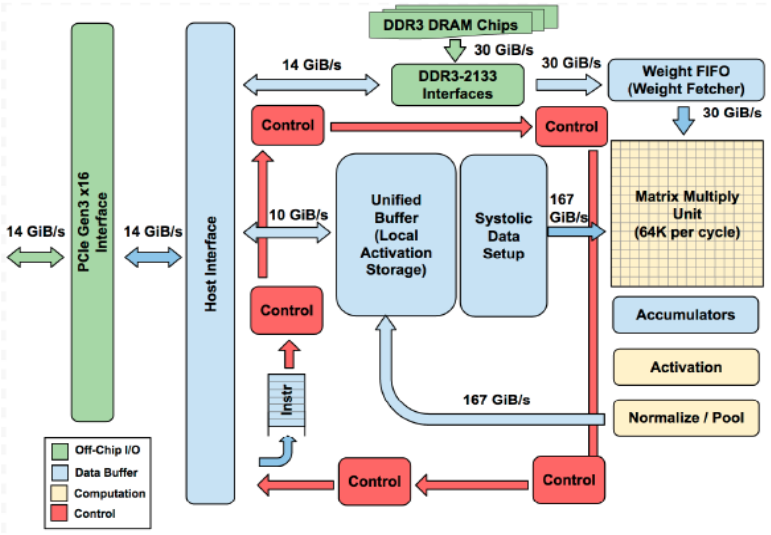


Figure 25: Block diagram of a TPU [43]

Google's Tensor Processing Unit (TPU) is an example of an ASIC accelerator that is used for training and inference of DNN models in Google's cloud platform and data centers. Figure 25 displays the block diagram of a TPU, where the yellow component is the Matrix Multiply Unit. This unit is responsible for the main computation. Figure 26 displays its printed circuit card, which can be inserted into the slot of a SATA disk in a server. Its inputs are the blue Weight FIFO and the blue Unified Buffer, and its output is the blue Accumulators. The nonlinear functions are performed by the yellow Activation Unit on the Accumulators and the results go to the Unified Buffer.

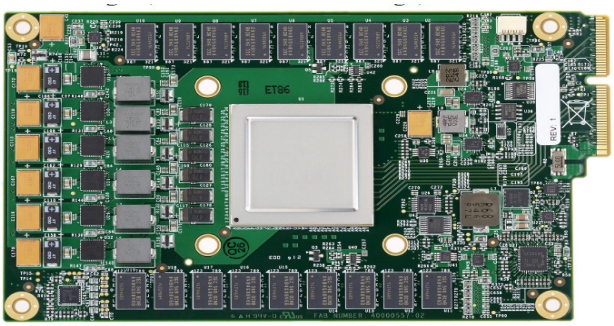


Figure 26: The Printed Circuit Board of a TPU [43]

The heart of the TPU, which is the 65,536 8-bit MAC matrix multiply unit, offers a peak throughput of 92 TeraOps/second (TOPS) and a large (28 MiB) software-managed on-chip memory [43]. It contains 256x256 MACs that can perform 8-bit multiply and adds on signed or unsigned integers. The 16-bit products are collected in the 4 MiB of 32-bit Accumulators below the matrix unit. The 4 MiB holds 4096, 256-element, 32-bit accumulators. TPU instructions follow the CISC computer architecture and its average clock cycles per instruction (CPI) is typically 10 to 20. It has in total about a dozen of instructions overall, but the most important ones are the following:

- *Read_Host_Memory*: Reads data from the CPU host memory into the Unified Buffer (UB).
- *Read_Weights*: Reads weights from Weight Memory into the Weight FIFO as input to the Matrix Unit.
- *MatrixMultiply/Convolve*: Orders the Matrix Unit to perform a matrix multiply or a convolution from the Unified Buffer into the Accumulators. A matrix operation takes a variable-sized B × 256 input, multiplies it by a 256 × 256 constant weight input, and produces a B × 256 output, taking B pipelined cycles to complete.
- *Activate*: Performs the nonlinear function of the artificial neuron, with options for ReLU, Sigmoid, and so on. Its inputs are the Accumulators, and its output is the Unified Buffer. It can also perform the pooling operations needed for convolutions using the dedicated hardware on the die, as it is connected to nonlinear function logic.
- *Write_Host_Memory*: Writes data from the Unified Buffer into the CPU host memory.

The philosophy of the TPU microarchitecture is to keep the matrix unit busy.

As reading a large SRAM uses much more power than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer. It relies on data from different directions arriving at cells in an array at regular intervals where they are combined. Figure 27 shows that data flows in from the left, and the weights are loaded from the top. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wavefront. The weights are preloaded and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. From a correctness perspective, software

is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit [43].

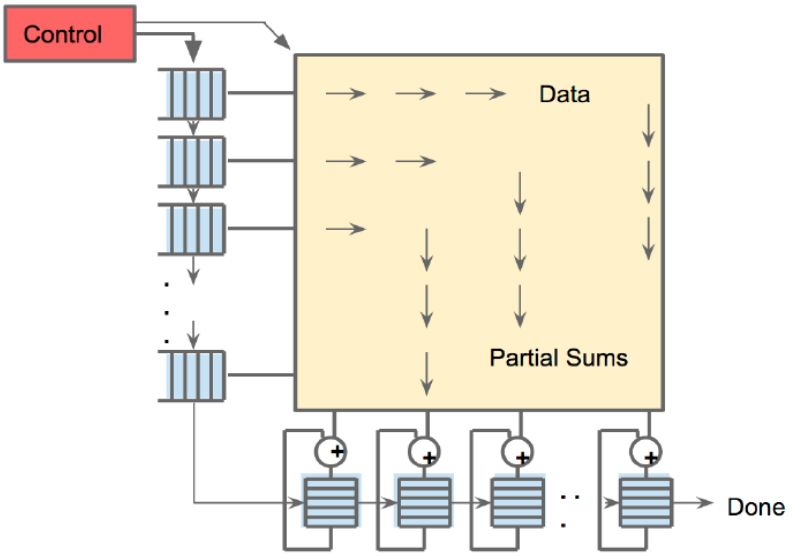


Figure 27: The Systolic dataflow inside the Matrix Multiply Unit [43]

ASIC accelerators give us a very good performance, which is very close to GPUs, but have a higher energy efficiency. This thesis will focus on this type of accelerators for the study and creation of a Near-threshold Approximate framework, which will help in the design of better energy efficient DNN accelerators. In general, this category uses a systolic array with a 256 × 256 grid of MAC units as its core and a large on-chip memory. However, with such density per chip in advanced technologies, soft error rates will escalate as well [9].

3.2 Approximate Computing in DNNs

One of the main problems of DNNs that prevent them from being widely adopted, is that they achieve superior accuracy at the expense of high computational complexity [44]. The models of state-of-the-art DNNs are very large as they require hundreds of MBs of data storage. As newer and larger DNN topologies immerge, the demand for compute is expected to grow. Despite the recent advances in computing systems that focus in optimizing DNN implementations and the introduction of custom accelerators for DNNs, for state-of-the-art neural networks on large datasets it takes days to weeks to train. All the above lead to a great interest in new opportunities to improve compute efficiency of DNN implementations.

DNNs have an important attribute, which is their resiliency in errors (e.g., the production of acceptable application-level output despite errors that occur during their constituent computations).

This attribute comes also from the fact that DNNs are used in applications where less-than-perfect results are acceptable. The nature of computations also performed within a neuron enhances this resilience. More specifically, each neuron in the network evaluates a weighted sum of its inputs, followed by a saturating (or thresholding) non-linear activation function (e.g., sigmoid, ReLU). Errors in the positive and negative directions compensate for each other during the weighted summation and any residual errors are attenuated by the activation function. Hence, approximate computing techniques can substantially benefit DNN implementations without sacrificing their classification accuracy.

Shikai Li et al. [45], introduced *Sculptor*, a flexible approximation with selective dynamic loop perforation. Sculptor, who is displayed in Figure 28, is based on loop perforation which is one of the most well-known software techniques in approximate computing. This technique transforms loops to periodically skip subsets of their iterations. During a thorough analysis, Shikai Li et al., discovered that this technique only considers the number of instructions to skip, but does not consider the differences between instructions and loop iterations. Based on their observation, these differences have considerable influence on performance and accuracy. To improve traditional perforation, they introduced selective dynamic loop perforation, a general approximation technique that automatically transforms loops to skip selected instructions in selected iterations. Across evaluated applications, selective dynamic loop perforation achieves an average speed up of 2.89x and 4.07x with less than 5% and 10% accuracy loss, and finally can be used to replace traditional loop perforation to achieve better performance improvements under the same error budgets.

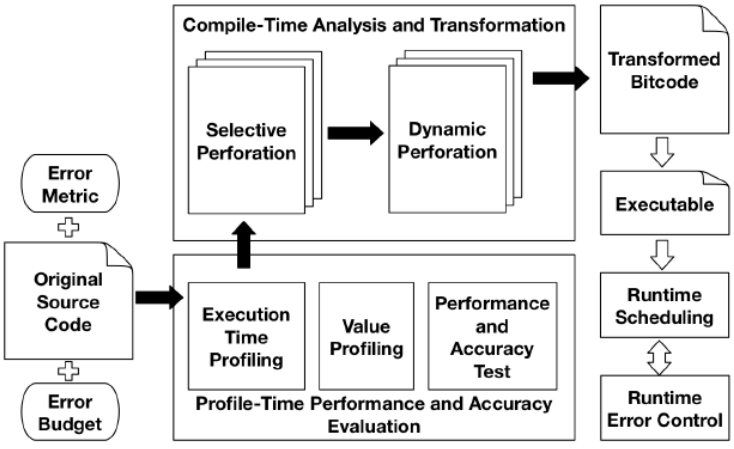


Figure 28: Overview of Sculptor [45]

In 2017, Jiecao Yu1 et al. [46], proposed *Scalpel*, a framework, as displayed in Figure 29, that uses weight and node pruning, an approximation technique that reduces DNN model size and the computation by removing redundant weights and nodes. Scalpel customizes DNN pruning to the underlying hardware by matching the pruned network structure to the data-parallel hardware

organization. It consists of two techniques: SIMD-aware weight pruning and node pruning. For low-parallelism hardware (e.g., microcontroller), SIMD-aware weight pruning maintains weights in aligned fixed-size groups to fully utilize the SIMD units. For high parallelism hardware (e.g., GPU), node pruning removes redundant nodes, not redundant weights, thereby reducing computation without sacrificing the dense matrix format. For hardware with moderate parallelism (e.g., desktop CPU), SIMD-aware weight pruning, and node pruning are synergistically applied together. Across the microcontroller, CPU and GPU, Scalpel achieves mean speedups of 3.54x, 2.61x, and 1.25x while reducing the model sizes by 88%, 82%, and 53%. In comparison, traditional weight pruning achieves mean speedups of 1.90x, 1.06x, 0.41x across the three platforms.

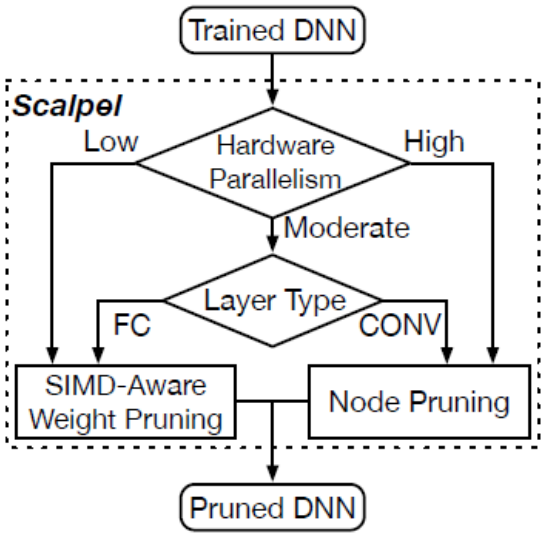


Figure 29: The Scalpel framework [46]

Chenzhuo Zhu et al. [47], aimed at the quantization technique, an approximation method to reduce the bit-precision of data used in arithmetic computation of DNN training and inference. Reduction in bit-precision results in more compact and energy efficient computing units in hardware. Based on this technique they proposed *Trained Ternary Quantization* (TTQ), a method, as displayed in Figure 30, that can reduce the precision of weights in neural networks to ternary values. This method has very little accuracy degradation and can even improve the accuracy of some models (32, 44, 56-layer ResNet) on CIFAR-10 and AlexNet on ImageNet. Experiments on CIFAR-10 dataset show that the ternary models obtained by trained quantization method outperform full-precision models of ResNet-32,44,56 by 0.04%, 0.16%, 0.36%, respectively. Respectively on ImageNet dataset, TTQ outperforms full-precision AlexNet model by 0.3% of Top-1 accuracy and outperforms previous ternary models by 3%.

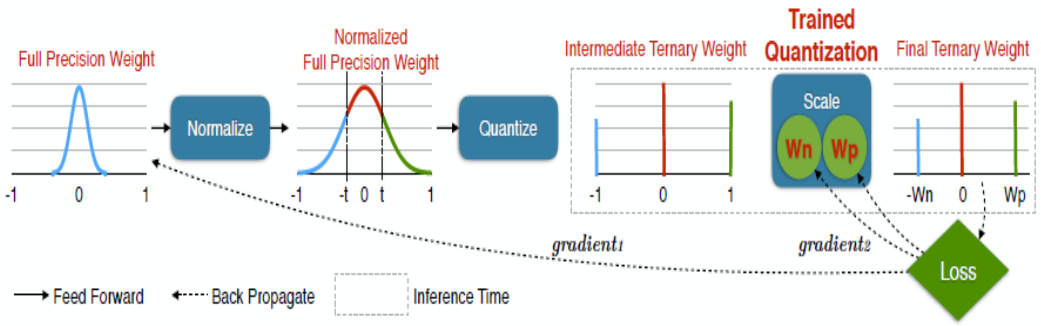


Figure 30: Overview of TTQ [47]

One promising approach to alleviate the computational challenges is implementing DNNs using low-precision fixed point (<16 bits) representation. However, the quantization error inherent in any Fixed Point (FxP) implementation limits the choice of bit-widths to maintain application-level accuracy. Shubham Jain et al. [48], presented *Compensated-DNN*, an approximation method that can dynamically compensate the error introduced due to quantization during execution. Their method introduces a new fixed-point representation named as Fixed Point with Error Compensation (FPEC). As seen in Figure 31, the bits in FPEC are split between computation bits vs. compensation bits. The computation bits use conventional FxP notation to represent the number at low precision. On the other hand, the compensation bits (1 or 2 bits at most) explicitly capture an estimate (direction and magnitude) of the quantization error in the representation. For a given word length, since FPEC uses fewer computation bits compared to FxP representation, a near-quadratic improvement in energy is achieved in the multiply-and-accumulate (MAC) operations. The compensation bits are simultaneously used by a low-overhead sparse compensation scheme to estimate the error accrued during MAC operations, which is then added to the MAC output to minimize the impact of quantization. During this study, Compensated-DNNs were built for a suite of 7 popular image recognition benchmarks and the energy evaluation conducted revealed a 2.65x – 4.88x and 1.13x – 1.7x improvement over 16-bit and 8-bit fixed point implementations with <0.5% accuracy difference.

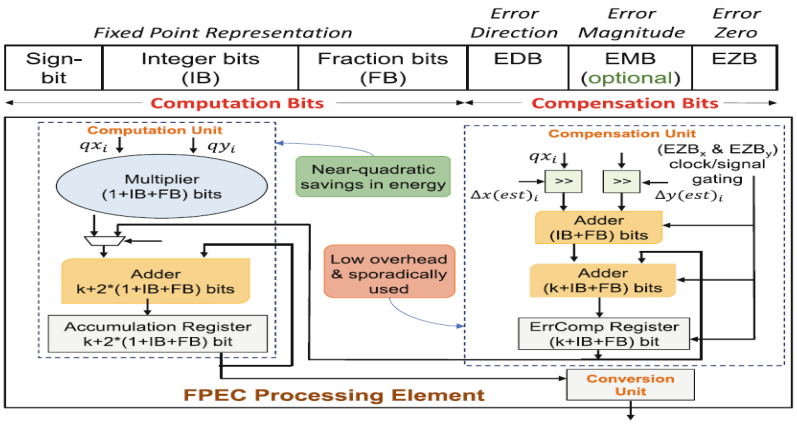


Figure 31: FPEC number format and processing element design [48]

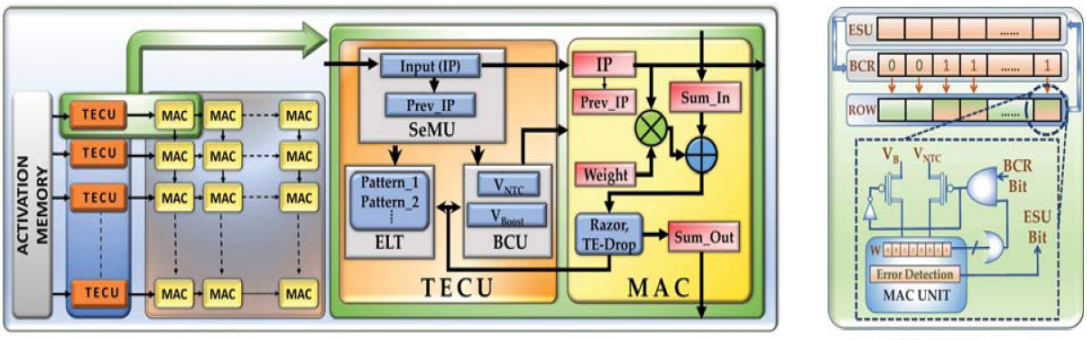
3.3 Near Threshold Computing in DNNs

While Moore's law continues to add more and more transistors, power consumption has become a great disadvantage for those devices prohibiting them from been used. Near Threshold Computing (NTC) comes as a solution to this problem, offering voltage scaling techniques as a design space where the supply voltage is approximately equal to the threshold voltage of the transistors. However, voltage scaling is limited for a lot of reasons, some of which are the following [5]:

- Process variations (PVs) play an important role in affecting the gains of voltage scaling.
- The sensitivity of circuits in voltage variations can lead to a failure much before supply voltage reaches the subthreshold voltage.
- Subthreshold leakage power starts to rise becoming a substantial portion of the total power.
- The appearance of soft errors (e.g., timing errors) during calculations in MACs and read/write operations in memories.

Nevertheless, the NTC design paradigm can serve as a great solution for providing the required energy efficiency for DNN accelerators that are at the forefront of supporting the immense throughput required for AI computation [49]. Yet, NTC operation is prone to a very high sensitivity to process and environmental variations, resulting in excessive increase in delay and delay variation. This leads to a slowdown in performance and induces high rate of timing errors in the DNN accelerator.

In 2019, Pramesh Pandey et al. [50], proposed GreenTPU, a low-power near-threshold (NTC) TPU design paradigm. Google Tensor Processing Unit (TPU) has transpired to be the best-in-class DNN accelerator, offering more than 15 × speedup over the contemporary GPUs. The work of GreenTPU is to identify the patterns in the error-causing activation sequences in the systolic array and prevent further timing errors from the same sequence by intermittently boosting the operating voltage of the specific MAC units in the TPU. Compared to other timing error mitigation techniques, GreenTPU manages to succeed a 2 × - 3 × higher performance in an NTC TPU, having in parallel a minimal loss in prediction accuracy.



(a) A conceptual block diagram of GreenTPU. (b) MACs-BCU interaction.

Figure 32: Block diagram of GreenTPU [50]

As we see in Figure 32(a), a Timing Error Control Unit (TECU) is pipelined between the activation memory and the row of each systolic array of MACs. A TECU has three main components: Error Log Table (ELT), Sequence Monitor Unit (SeMU), and Boost Control Unit (BCU). The ELT logs the timing error causing input sequence pattern. Simultaneously, the BCU is alerted to boost the operating voltage of the subsequent MACs in the row, to prevent any future timing error. The SeMU monitors the sequence of inputs and tries to find a matching pattern in the ELT in every clock cycle. If a match is found, SeMU communicates with the BCU to preclude future timing errors in all the MAC units of a row. Inside each MAC a timing error is detected and tackled using Razor and TE-Drop techniques. Figure 32(b) displays the interaction between MACs and BCU. BCU houses two 256-bit registers: Boost Control Register (BCR) and Error Sensing Unit (ECU). Each bit of these registers corresponds to each MAC unit in a row. As we see, every MAC unit has access to two voltage rails, V_{ntc} and V_b, representing a near-threshold and a boost voltage, respectively. The reset (set) value in any bit of the BCR, indicates the corresponding MAC unit to operate with the V_{ntc} (V_b) voltage. V_{ntc} and V_b is set to 0.45V and 0.65V, respectively.

High throughput architectures like General Purpose computing on GPUs (GPGPUs) can also significantly improve their performance through NTC techniques. GPGPUs provide excellent computing power for massively parallel applications and, while originally were developed for accelerating graphics processing, can dramatically speed up computational processes for deep learning. Nevertheless, NTC is more sensitive to PVs as it complicates power delivery. Rafael Trapani Possignolo et al. [51], proposed GPU Stacking, a novel method based on voltage stacking, to manage the effects of PV and improve the power delivery simultaneously. Voltage stacking improves the efficiency of power delivery. When n units are stacked, they are placed in a series fashion, rather than the conventional parallel scheme. Thus, the current in the power delivery network is reduced by a factor of n in a system. This allows voltage regulators (VRs) with increased efficiency, smaller areas and fewer package pins dedicated to power. GPU Stacking methodology lets the voltage node between the stacked elements (V_{MID}) float. This floating node is the key to PV compensation. GPU Stacking alleviates the current delivery challenges, and intrinsically mitigates PV effects without requiring multiple voltage domains. GPU Stacking automatically creates a voltage domain per level in the stack without the cost of multiple power rails. This method has a great success in increasing performance under process variation at near threshold, on average, by 37% compared to the traditional (not stacked) configuration, delivering 80% of the performance compared to the no variation (ideal) conditions.

While studying GPUs, Prabal Basu et al. [52], resulted in finding two crucial factors that significantly undermine their efficacy at NTC: (a) Delays provoked from NTC make the GPU

applications severely sensitive to Multi-cycle Latency Datapaths (MLDs) within the GPU pipeline and (b) PVs at NTC induces a substantial performance variance. To dela with these challenges, they proposed SwiftGPU, an energy efficient GPU design paradigm at NTC. SwiftGPU dynamically adjusts the degree of parallelization, and the speed of the MLDs within each stream core of the GPU. To do this, SwiftGPU employs Self-Adaptive Sprint (SAS), that dynamically sprints the MLDs based on the dimensions of the GPU kernel, as well as the MLD usage pattern during the kernel execution. As seen in Figure 33, the SAS Controller dynamically manages the execution speed of the Compute Unit (CU) (or SM) MLDs. To tackle the impact of PV, several crucial design strategies are adopted, ranging from the use of tunable voltage rails to a meticulous selection of the MLD speeds. To support several datapath speeds, the underlying power-delivery network is augmented to allow three different supply voltage rails: Vdd_H, Vdd_M and Vdd_L, respectively. The SAS controller monitors the runtime hardware utilization of various CU MLDs, and dynamically adjusts the MLD speed to improve the energy efficiency of the entire system.

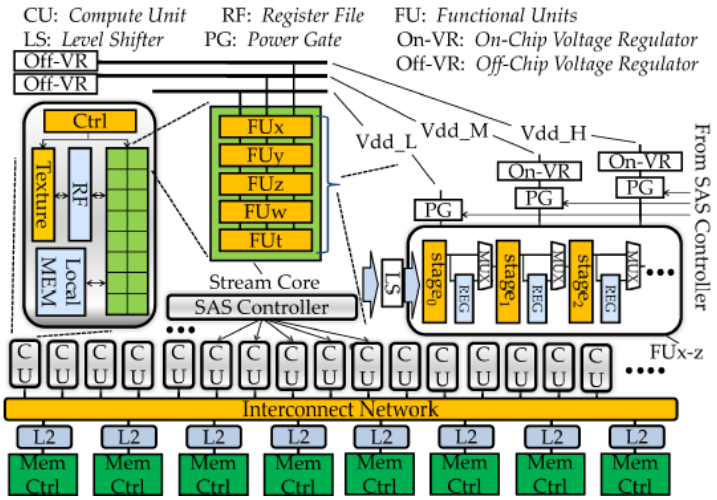


Figure 33: Overview of SAS [52]

4 HARDWARE ARCHITECTURES FOR DNN PROCESSING

When talking about efficient processing of DNNs, it is important to consider the key metrics that are needed to evaluate and compare the strengths and weaknesses of different designs and proposed techniques. Efficiency is commonly associated with the number of operations per second per Watt (e.g., FLOPS/W, TOPS/W) but there are many more metrics including accuracy, throughput, latency, energy consumption, power consumption, cost, flexibility, and scalability. Another important factor that must be considered is data reuse. Data reuse concerns data movement (e.g., accessing data to memory) and plays a crucial role in energy consumption of modern compute systems. In this chapter we present the basic key metrics and their importance, we explain why the case of DNN accelerators is important for DNN processing and finally we present two methods for exploiting efficiently data reuse.

4.1 Basic Key Metrics

DNNs became very popular and widely used because of the fact that they can offer state-of-the-art *accuracy* [1] on a wide range of tasks. Accuracy is an indicator of the quality of the result for a given task and is a key metric that must be considered when designing efficient specialized hardware to process DNNs workload. During training or inference of a DNN model, the units used to measure accuracy depend on the task. If for example we talk about image classification, accuracy is reported as the percentage of correctly classified images. There are two factors that affect accuracy: the difficulty of the task (e.g., object detection) and the dataset (e.g., ImageNet). Therefore, a DNN model that performs well on MNIST dataset may not necessarily perform well on ImageNet. In conclusion, when someone is called to evaluate the efficiency of different hardware in processing DNNs, accuracy must seriously be taken into account as it is a crucial metric for the effectiveness of DNN models.

Throughput and *latency* [1] are also two important metrics for evaluating hardware efficiency. Throughput indicates the amount of data that can be processed or the number of executions of a task that can be completed in a given time period. On the other hand, latency measures the time between when the input data arrives to a system and when the result is generated. Throughput is often generically reported as the number of operations per second. When talking about inference, throughput is reported as inferences per second or in the form of runtime in terms of seconds per inference. Latency is typically reported in seconds. Throughput and latency are actually quite distinct even if we often assume that are directly derivable from one another. For example, when we use batching for input data (e.g., batching multiple images or frames together for processing) this increases throughput since it amortizes overhead, such as loading the weights. However, batching

also increases latency which is not acceptable in real-time applications such as high-speed navigation where it would reduce the time available for course correction.

High *energy efficiency* [1] is another important key metric when processing DNNs specially at the edge in embedded devices with limited battery capacity (e.g., smartphones). This metric is used to indicate the amount of data that can be processed or the number of executions of a task that can be completed for a given unit of energy. Energy efficiency is often generically reported as the number of operations per joule. *Power consumption* [1], which is linked to energy efficiency, is used to indicate the amount of energy consumed per unit time. Thermal design power (TDP) is a design criterion that dictates the maximum power consumption of specialized hardware, which is the power that the cooling system is designed to dissipate. Power consumption is typically reported in watts or joules per second. There are various design considerations for the hardware that will affect the energy per operation (e.g., joules per operation). The energy per operation can be broken down into the energy required to move the input and output data, and the energy required to perform the MAC computation as following:

$$Energy_{total} = Energy_{data} + Energy_{MAC} \qquad (7)$$

Each component performs a joules per operation that is computed as:

$$\frac{joules}{operation} = a \times C \times V_{dd}^{2} \qquad (8)$$

where α is the switching activity, C is the total switching capacitance and V_{dd} is the supply voltage.

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Multiply	0.2
32b Multiply	3.1
16b FP Multiply	1.1
32b FP Multiply	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640

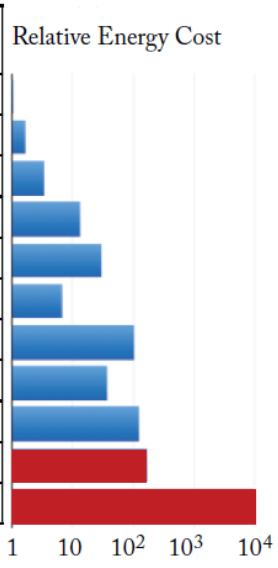


Figure 34: The energy consumption for various arithmetic operations and memory accesses in a 45nm process [1]

In Figure 34, the relative energy cost (computed relative to the 8b add) is shown on a log scale. As we see, the energy consumption of data movement (red) is significantly higher than arithmetic

operations (blue), thus energy consumption is dominated by the data movement as the capacitance of data movement tends to be much higher that the capacitance for arithmetic operations such as a MAC [1]. Moreover, the switching capacitance increases the further the data needs to travel to reach the PE, which consists of the distance to get out of the memory where the data is stored and the distance to cross the network between the memory and the PE. Consequently, larger memories and longer interconnects (e.g., off-chip) tend to consume more energy than smaller and closer memories due to the capacitance of the long wires employed. So, in order to reduce the energy consumption of data movement, we can exploit data reuse where the data is moved once from distant large memory (e.g., off-chip DRAM) and reused for multiple operations from a local smaller memory (e.g., on-chip buffer or scratchpad within the PE). Optimizing data movement is a major consideration in the design of DNN accelerators.

4.2 The case of DNN accelerators

A typical DNN accelerator has several processing elements (PEs) and various on-chip buffers. As shown in Figure 35, the arrays of Processing Elements (PE) fetch the pixels (T_n pixels) of the input feature maps (IFMs) from the input buffer, the weights from the weight buffer, the partial sums (PSUMs) from the output buffer, and then compute the PSUMs or the pixels (T_m pixels) of the output feature maps (OFMs), which are stored in the output buffer.

Designing specialized hardware, such as DNN accelerators, is a great challenge specially for the fact that, with the end of Moore's law [53], big computational needs coming from DNNs require to employ domain-specific hardware/software co-design (e.g., domain-specific languages such as Pytorch) in computing systems to continue to improve performance and energy efficiency. Co-design of hardware and software refers to the development of new software and languages that improve the user experience [54]. In addition, the compiler can better map such workloads to domain-specific hardware to enable improvements in performance and energy efficiency.

DNN accelerators provide a large improvement in key metrics such as performance and power efficiency over general-purpose processors across a wide range of DNN computations. This is because when considering hardware organizations for DNN acceleration, design space for specialized DNN hardware is quite large. As a result, there are no constraints on the execution order of MAC operations within a DNN layer. This leads the hardware designer to have wide latitude in choosing the execution order of operations and optimizing the hardware for the targeted metrics given certain resource constraints (e.g., memory capacity).

A major advantage of DNN accelerators is that they have some degree of fault tolerance due to the fault tolerance of the DNN algorithms themselves. For example, the ReLU, normalization, and max-

pooling layers help mask the effects of errors. But in order to achieve high data reuse, so as to improve performance, the memory in DNN accelerators is accessed repeatedly, and due to this, a faulty value may be reused several times.

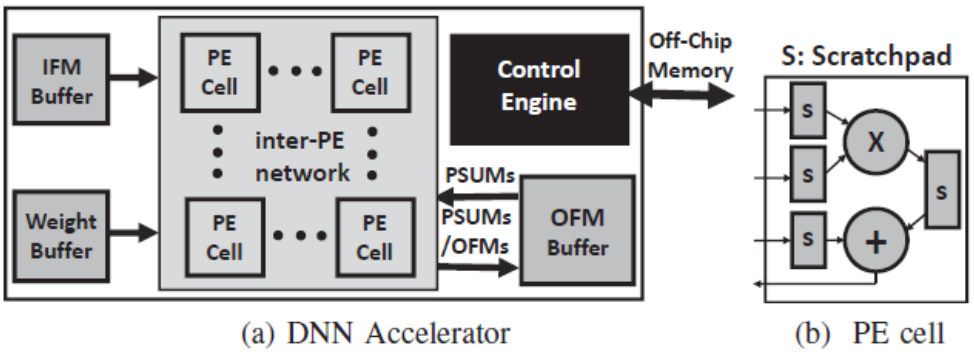


Figure 35: DNN accelerator architecture [56]

The energy efficiency of DNN accelerators has also attracted much attention when they are used in many low-power environments such as IoTs [55]. As silicon fabrication technologies become smaller and smaller, the static power caused by leakage current accounts for a large portion of the overall chip power. In modern DNN accelerators, a significant portion of the chip (e.g., 75%) is used for on-chip buffers. These on-chip buffers are usually implemented by static random-access memories (SRAMs). Therefore, a large part of the current is consumed by the leakage current of the SRAMs.

The situation is worsened by the increasing trend of using compact data representations in DNNs to improve efficiency, as more chip area is required to use SRAMs [56]. This trend leads to memory oriented DNN accelerators as the computing units of the accelerators are simplified due to the compact data representation. However, larger on-chip buffers are used to store more data on chip and reduce off-chip traffic. Thus, in addition to the reliability of DNN accelerators, the static performance of on-chip memory is also a non-negligible factor in the development of energy-efficient DNN accelerators.

4.3 Examining Data Reuse

Data reuse is a perfect way to reduce the cost of moving data. For DNN accelerators, data reuse is a key behavior that improves both latency and energy via reducing the number of remote buffer accesses (i.e., global buffer). This section presents two architectural techniques, *temporal reuse* and *spatial reuse*, and describes how they are applied in hardware.

4.3.1 Temporal Reuse

Temporal reuse is an architectural technique thar focuses on the fact that the same data value is used

more than once by the same consumer (e.g., a PE). This technique can be applied by adding an intermediate memory level with a smaller storage capacity than the level that acts as the original source of the data. Since smaller memories consume less energy to access than larger memories, the data value is transferred once from the source level (i.e., larger memory) to the intermediate level (i.e., smaller memory), and used multiple times at the intermediate level, which reduces the overall energy cost.

We can distinguish two modes: *temporal multicast* and *temporal reduction* [57]. Temporal multicast occurs for input tensors (e.g., filter and input activation) where in this case the reused data can be multicasted to multiple PEs over time. In the temporal multicast example of Figure 36, the same data tile 1 appears over time in the same PE (PE1). That is, we send the data to the future for reuse in the future, that means store the data from the Global Buffer Memory (GBM) to a smaller memory (e.g., a buffer) and read it in the future. Therefore, temporal multicast, which is reading the same stored data over time, requires a buffer, as shown in Figure 36. On the other hand, during temporal reduction, the computed partial sums over time are accumulated within the same location. This type of reuse requires a buffer since intermediate results need to be stored and read again in the future, which effectively indicates multiple read-modify-write to a buffer. The example in Figure 36 shows such a reuse pattern, where the output tile 1 appears at the same PE over time.

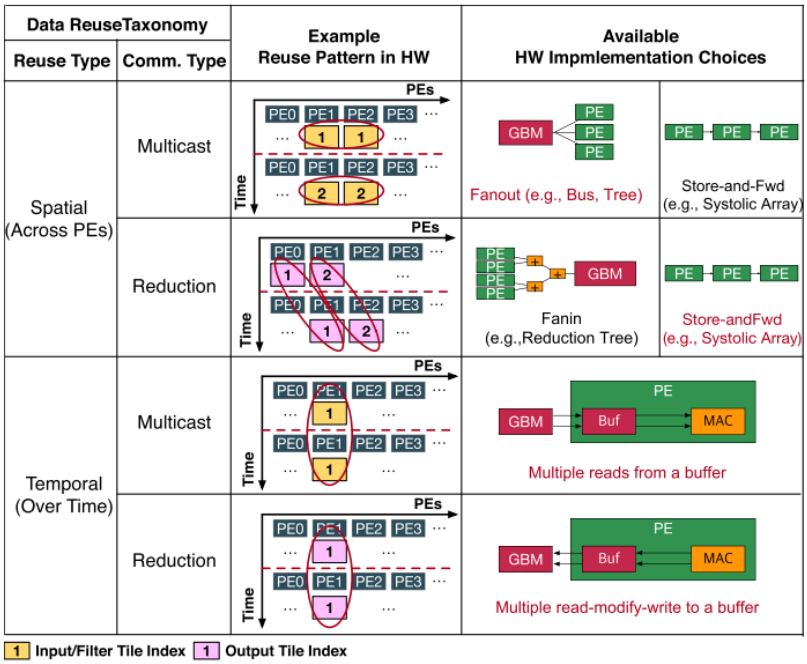


Figure 36: Overview of data reuse in DNN accelerators [57]

4.3.2 Spatial Reuse

Spatial reuse occurs when the same data value is used by more than one consumer (e.g., a group of

PEs) at different spatial locations on the hardware. It can be exploited by reading the data once from the source memory layer and transmitting it to all consumers via *multicast* or *reduction* [57]. Utilizing spatial reuse has the advantage of:

- reducing the number of accesses to the source storage layer, which lowers the overall energy cost, and
- reducing the bandwidth required by the source storage layer, which helps keep the PEs busy and thus improves performance.

In spatial multicast, which occurs for input tensors, data is delivered to multiple PEs at the same time. In the spatial multicast example of Figure 36, tiles 1 and 2 are delivered to PE1 and PE2 at the same time leveraging the multicast capability of fanout hardware such as Bus or Tree. Alternatively, store-and-forward style implementation such as systolic arrays is available with tradeoff of hardware cost and latency. Now as for spatial reduction, which occurs for output activation tensors, partial outputs (or partial sums) are accumulated for an output across multiple PEs. Figure 36 shows an example reuse pattern based on store-and-for- ward hardware. We observe that the output tiles 1 and 2 are moving to the next PE over time, which illustrates pipelined accumulation to the right direction assuming that PEs are receiving new operands from above (i.e., a row of a systolic array). Alternatively, fanin hardware such as Reduction Tree can support the spatial reduction.

4.4 Why Dataflows are important

For DNNs, the bottleneck for processing is memory access. As shown in Figure 37, each MAC needs three actions of read, each one for filter weight, fmap activation, and partial sum and one action of write for the updated partial sum. In the worst case, all memory accesses must be made via the off-chip DRAM, which severely compromises both throughput and energy efficiency [58]. For example, to support the 724 million MACs in AlexNet, nearly 3000 million DRAM accesses are required. In addition, the DRAM accesses require up to several orders of magnitude more energy than the computations. To address these challenges, it is of great importance to design a compute scheme called *dataflow*, which decides what data get read into which level of the memory hierarchy and when are they getting processed.

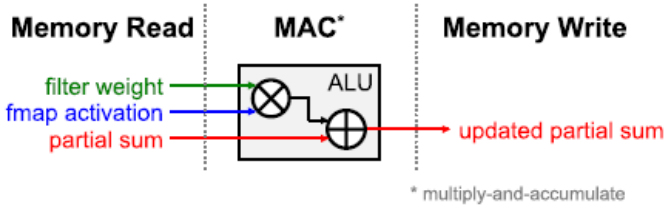


Figure 37: The actions of read/write in a MAC [58]

DNN accelerators offer the opportunity to reduce the energy cost of data movement by introducing multiple levels of local storage hierarchy with different energy costs, as shown in Figure 38. These include a large global buffer with a size of several hundred kilobytes connected to DRAM, a network between PE, which can pass data directly between ALUs, and a register file (RF) within each PE with a size of a few kilobytes or less. The different levels of the memory hierarchy help improve energy efficiency by providing low-cost data access. Retrieving data from the RF or the neighboring PEs offers one or two orders of magnitude less energy than from DRAM.

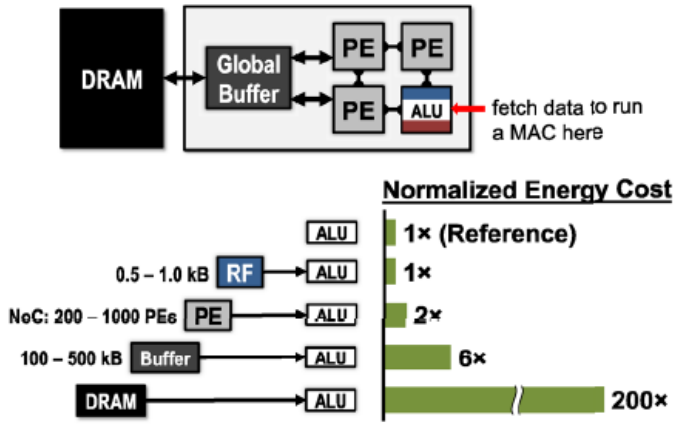


Figure 38: Levels of local storage hierarchy with different energy costs [58]

Since there is no randomness in the processing of DNNs, it is possible to design a fixed data flow that can adapt to the shapes and sizes of DNNs and optimize them for the best energy efficiency. The optimized data flow minimizes access from the more energy consuming levels of the storage hierarchy.

Large storage, which can store a significant amount of data, consumes more energy than smaller storage. For example, DRAM can store gigabytes of data, but consumes two orders of magnitude more energy per access than a small on-chip memory of a few kilobytes. Each time a portion of data is moved from an expensive tier to a tier with a lower energy cost, we want to reuse that portion of data as often as possible to minimize subsequent accesses to the expensive tiers. The challenge, however, is that the storage capacity of these low-cost stores is limited. Therefore, we need to explore different data flows that maximize reuse under these constraints.

For DNNs, we study dataflows that exploit three forms of input data reuse (convolution, fmap, and filter), as shown in Figure 39. Convolutional data reuse uses the same fmap activations and filter weights within a given channel, just in different combinations for different weighted sums. In fmap reuse, multiple filters are applied to the same fmap, so that the input fmap activations are used multiple

times for all filters. Finally, in filter reuse, when multiple input fmaps are processed at once (referred to as a batch), the same filter weights are used multiple times for all input fmaps.

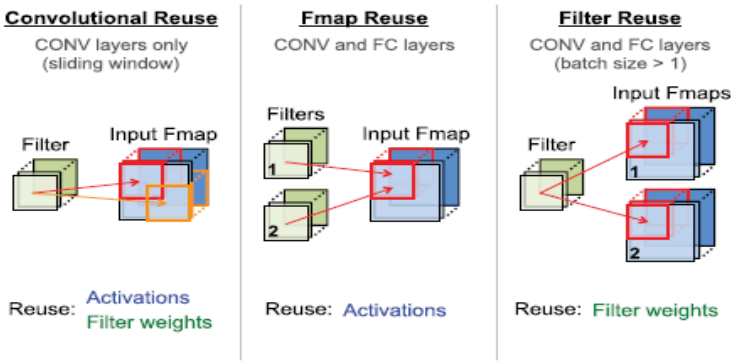


Figure 39: Forms of input data reuse [58]

Overall, we can distinguish the following three types of dataflows:

- *Weight stationary* (WS): Aims to minimize the energy consumption of reading weights by maximizing the reuse of weights from the register file (RF) at each PE. As shown in Figure 40(a), each weight is read from the Global Buffer (e.g., DRAM) into the RF of each PE and stays stationary for further accesses. Processing will run as many MACs using the same weight as possible, if the weight is present in RF; this maximizes convolutional and filter reuse of weights. The inputs and partial sums must move through the spatial array and global buffer. The input fmap activations are broadcast to all PEs and then the partial sums are spatially accumulated across the PE array. Google's TPU is a design that features a weight-stationary dataflow.

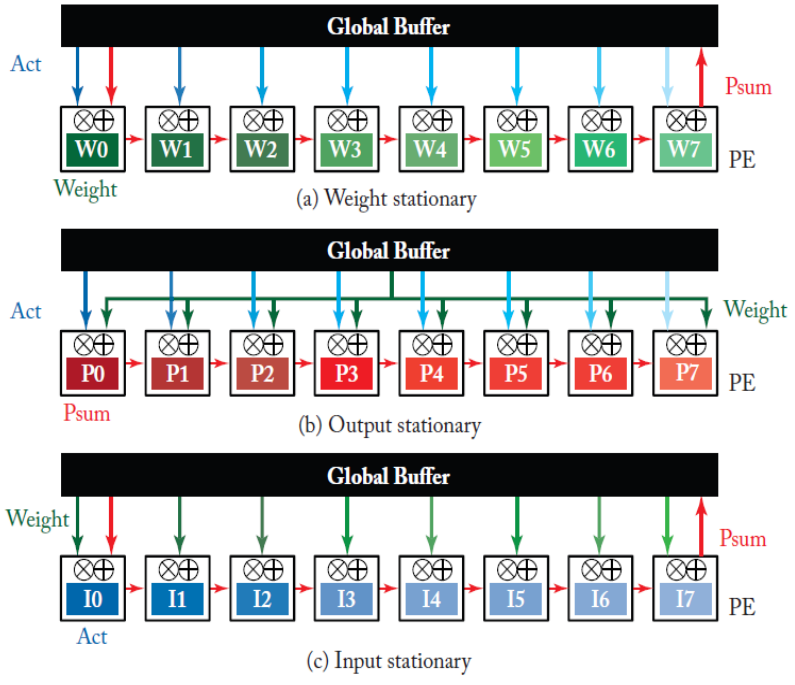


Figure 40: Types of dataflows [1]

- *Output stationary* (OS): This type of dataflow is designed to minimize the energy consumption of reading and writing the partial sums. As shown in Figure 40(b), OS keeps the accumulation of partial sums for the same initial activation value local in RF. To keep the accumulation of partial sums stationary in RF, a common implementation is to stream input activations across the PE array and send the weight to all PEs in the array. We can distinguish multiple possible variants of output stationary, as shown in Figure 41, since the output activations that get processed at the same time can come from different dimensions [58]. For example, the variant OS_A targets the processing of CONV layers and therefore focuses on processing output activations of the same channel at the same time to maximize the possibilities of reusing convolutional data. The OS_C variant targets the processing of FC layers and focuses on generating output activations from all different channels, since each channel has only one output activation. The variant OS_B lies roughly between OS_A and OS_C.

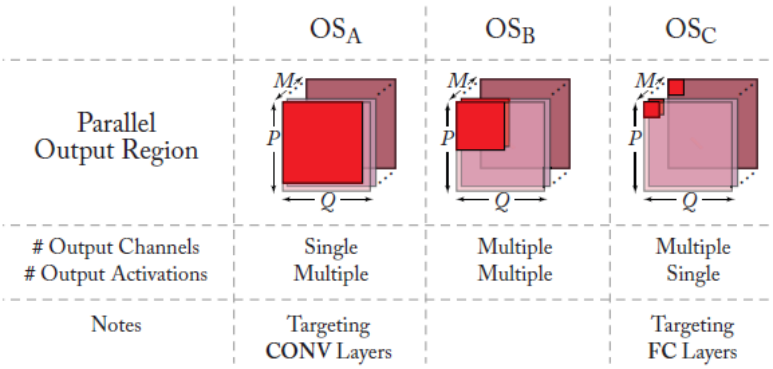


Figure 41: Variants of Output Stationary Dataflow [58]

- *Input Stationary* (IS): Like the other two types of dataflows, this type is designed to minimize the energy consumption of reading input activations. As seen in Figure 40(c), each input activation is read from DRAM and put into the RF of each PE and stays stationary for further access. Then, it runs through as many MACs as possible in the PE to reuse the same input activation. It maximizes the convolutional and input fmap reuse of input activations. While each input activation remains stationary in RF, unique filter weights are transferred to the PEs at each cycle, while the partial sums are spatially accumulated across the PEs to produce the final output activation.
- *Row Stationary* (RS): This type of dataflow focuses on maximizing the reuse and accumulation at the RF level for all types of data (weights, input activations, and partial sums)

for the overall energy efficiency. During RS dataflow, processing of a 1-D row convolution is assigned into each PE for processing. It keeps the series of filter weights stationary in the RF of the PE and then directs the input activations to the PE. The PE performs the MACs for each sliding window at once, which uses only one memory location for accumulating the partial sums. Since there are overlaps of input activations between different sliding windows, the input activations can be stored and reused in RF. By going through all the sliding windows in the row, it completes the 1-D convolution and maximizes the reuse of data and local accumulation of data in that row. The above process can be seen in Figure 42.

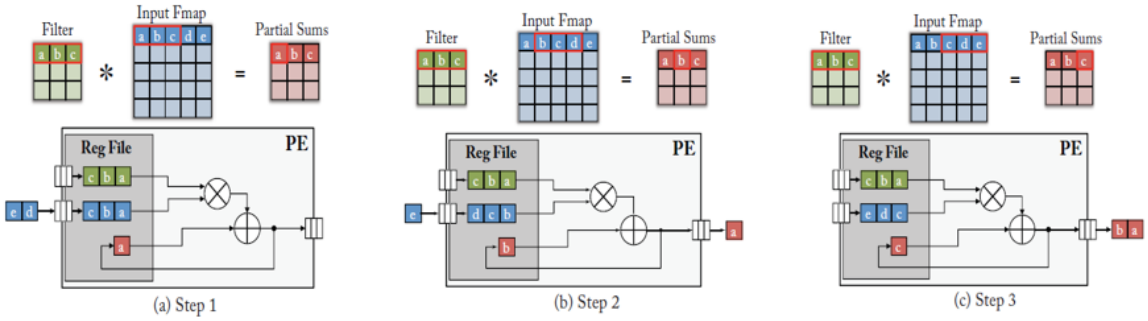


Figure 42: An overview of Row Stationary Dataflow [58]

5 THE PROPOSED NTV-DNN FRAMEWORK

This chapter focuses on presenting a proposed Near-Threshold Voltage DNN framework (NTV-DNN) that could serve as an auxiliary tool for designing performance and energy efficient DNN accelerators. Experiments are conducted for the assessment of this tool, during which we also check the resilience of DNN models in errors provoked from scaling the supply voltage (V_{dd}).

5.1 Summary

As DNN accelerators came to the fore, this led to an improvement in the speed of DNN inference by several orders of magnitude. TPU by Google is considered, among other DNN accelerators, the best in class offering more than 15 × over the contemporary GPUs [50]. Nevertheless, the growth of DNN workloads causes excessive computation, resulting in increased energy consumption in TPU-based data centers. To reduce power and energy consumption, while balancing performance and accuracy, we propose NTV-DNN, a tool for early assessment of energy at various voltage variation levels. NTV-DNN uses MAESTRO [57] as a DNN Accelerator Architectural Model to produce more than 20 statistics including total latency, energy, power, throughput, etc., as outputs in Super-Threshold Voltage Computing (STC) regime. These statistics are fed to a Near-Threshold Voltage Computing (NTC) Analysis framework, to calculate power and energy consumption, performance, and relative accuracy in NTC regime. During our research, we simulate a 16 x 16 TPU-based accelerator, trying to find the best operating voltage of the working PEs without causing errors during the computations of a DNN inference.

5.2 The NTV-DNN architectural model

Figure 43 depicts the design overview of our NTV-DNN framework. There are three crucial elements that make up the basic function of the framework: the DNN accelerator architectural model (MAESTRO), the NTC Analysis and the Error Model. MAESTRO is an open-source tool for modeling and evaluating the performance and energy-efficiency of different dataflows. This tool takes as input hardware parameters (e.g., total numbers of PEs, size of L1 scratchpad memory, etc.), the dataflow (e.g., weight stationary) and the DNN model (e.g., AlexNet), makes a thorough tensor, cluster, reuse, performance, and cost analysis and exports the results in a report in STC regime. This report is fed to an NTC analysis tool based in a python script. The NTC analysis tool is used to calculate the power, energy consumption and performance based on a voltage scaling schema.

To produce accuracy results we first run inference of the selected DNN model using the CIFAR-10 dataset. Then, based on PyTorchFI, which is a runtime perturbation tool for DNNs [59], we produce an Error Model. This model takes as input some Fault Injection (FI) parameters (e.g., NTC cluster size, positions for FI, etc.), the DNN model and the dataset. The Error Model is tweaked to perturb the binary output value of a conv2d (neuron) operation before applying nonlinearity. Then a new inference is run, based on the test dataset of CIFAR-10 and the FI positions produced from the NTV analysis tool. This FI procedure is based on an NTC frequency scaling schema that aims to calculate the accuracy of the examined DNN model under relaxed errors. During frequency scaling and based on the resilience of the DNN model in errors, we calculate a new performance.

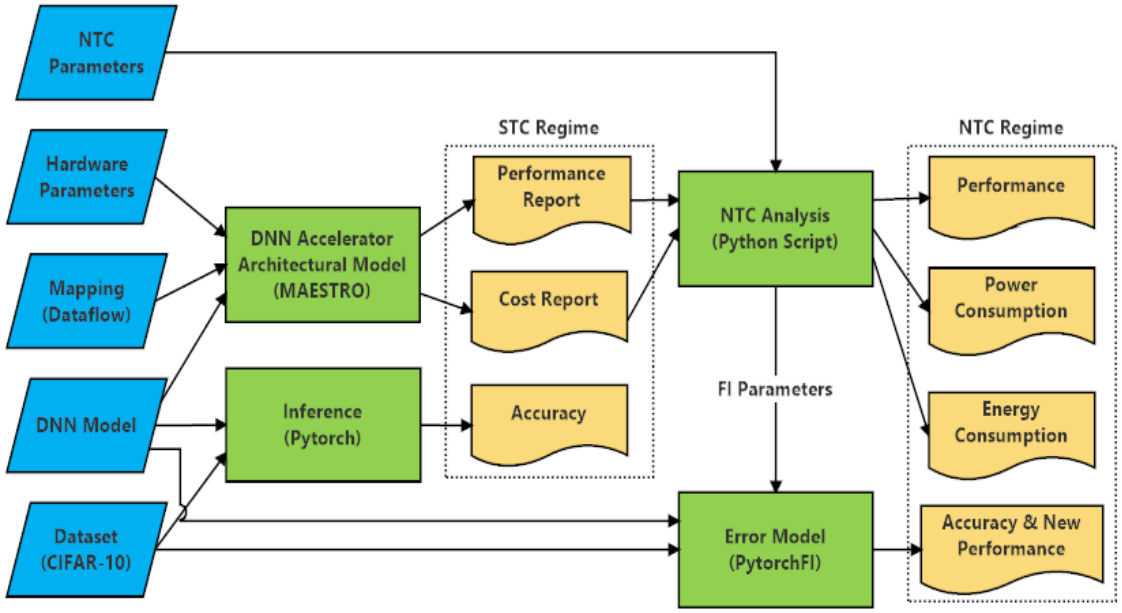


Figure 43: An overview of the proposed NTV-DNN framework

The NTC analysis tool shown in Figure 44, is a python-based script which has a main function that runs all the necessary operations to produce a report for power, energy consumption and performance of the examined DNN model in NTC regime. The operation of each function is described in Table 3.

Table 3: List of functions in NTC Analysis tool

Function name	Input	Parameters	Output	Operation
power_and_enegy_calc	.csv file (MAESTRO), Vdd in NTC regime	file, Vdd_ntc	Total power in NTC regime, Total energy in NTC regime.	Calculates the total power and energy consumption in NTC regime for the given DNN model and cluster size.

performance_calc	.csv file (MAESTRO), Vdd in NTC regime	file, Vdd_ntc	Total performance in NTC regime.	Calculates the total performance in NTC regime for the given DNN model and cluster size.
fi_injection_pos	FI_step = int(num_PEs / ntc_cluster_size)	k_step	fi_injection_list = [layer, C_out, X_out, Y_out]	Produces the locations of the error (layer_num, dim1, dim2, dim3) that are necessary for the declaration of each neuron injection.
plot_graph	Vdd in NTC regime, Total power/energy/ performance in NTC regime	vdd_ntc, (total_pwr_ntc, total_en_ntc, perf_ntc), (total_pwr_stc, total_en_stc, perf_stc)	The graphs of power, energy consumption and performance in NTC and STC regime.	Plots the power, energy consumption and performance in NTC and STC regime for the given DNN model and cluster size.

The power_and_energy_calc function determines the MAC power and energy leakage for idle PEs using a power gating mechanism. Power-gating consists in switching-off an NMOS footer (or a PMOS header) connected in series with the logic in order to cut the leakage current flow. This function calculates the power and energy consumption of PEs, Network On Chip (NOC) and SRAM (L1 Scratchpad & L2 Shared Buffer) in NTC regime. The performance_calc function calculates the total execution time based on runtime per layer and relative frequency in NTC regime. Fi_injection_pos is used to generate neuron fault injection positions. Finally, the plot_graph function is responsible for the plotting of the graphs that display power, energy consumption and performance in NTC regime.

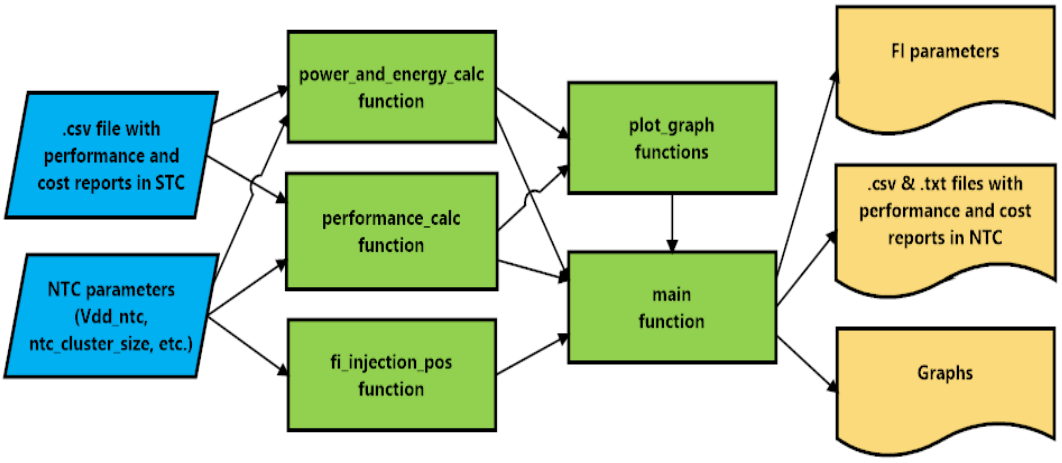


Figure 44: The NTC Analysis tool

5.3 Voltage allocation and scaling for NTV-DNN

During recent years, designers are dealing with the problem of the so-called power/utilization wall. Moore's law has led to the adoption of manycore architectures as the principal strategy to increase performance, ignoring the dark silicon problem connected to power usage, which is closely related to heat dissipation. NTC comes as a promising technique to encounter this problem as it takes advantage of the quadratic relation between the supply voltage $\left(V_{dd} \right)$ and the consumed power, by lowering the operating V_{dd} to a region slightly larger than the transistors' threshold voltage $\left(V_{th} \right)$. NTV-DNN hosts a voltage allocation and scaling technique which is based on the formation of voltage islands (VIs) for the minimization of the impact of within-die variations, which are more evident at NTC, in both performance and power. This technique, proposed in 2014 from I. Stamelakos, S. Xydis, G. Palermo et al. [60], was developed for manycore CPU architectures and proved that when moving to the NTC regime for a 128-core architecture, average power gains close to 65% are delivered while sustaining the performance values obtained by a 16-core architecture at STC.

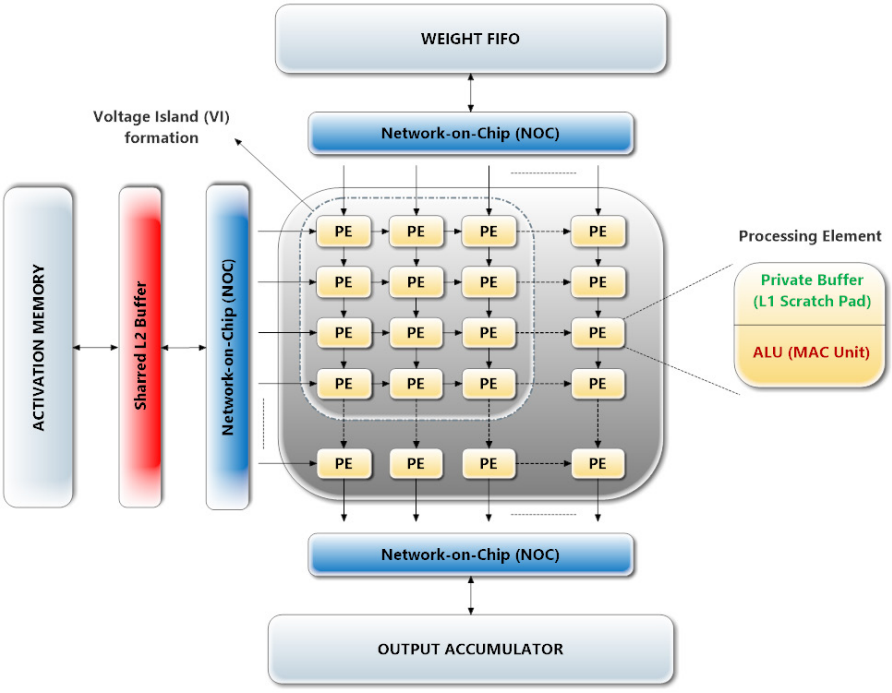


Figure 45: The NTV-DNN TPU based systolic array accelerator with a VI formation

The scope of the NTV-DNN voltage allocation and scaling scheme is to create, as Figure 45 depicts, VIs of PEs in a TPU-based accelerator. Every PE contains a MAC unit and a L1 scratchpad memory. The frequency of every MAC in NTC regime is calculated as follows:

$$f_{NTC} = \left(\frac{V_{dd,STC}}{V_{dd,NTC}} \right) \times \left(\frac{V_{dd,NTC} - V_{th}}{V_{dd,STC} - V_{th}} \right)^{b} \times f_{STC} \qquad (9)$$

where $V_{dd,STC}$, $V_{dd,NTC}$ are the operating voltages of a MAC in STC regime and NTC regime, V_{th} is the threshold voltage of a MAC, f_{STC} is the frequency of every MAC in STC regime and b is technology-dependent constant (≈ 1.5). Given the corresponding V_{dd} allocation per VI, we can calculate the power of each component in NTC. The dynamic (DP) and leakage (LP) power scaling factors are:

$$SF_{DP} = \left(\frac{V_{dd}}{V_{dd,STC}}\right)^2 \times \left(\frac{f_{NTC}}{f_{STC}}\right) \quad (10)$$

$$SF_{LP} = \left(\frac{V_{dd}}{V_{dd,STC}}\right) \times \exp\left(\frac{V_{th,STC} - V_{th} + DIBL}{n \times V_{thermal}}\right) \quad (11)$$

$$DIBL = \lambda \times \left(V_{dd} - V_{dd,STC}\right) \quad (12)$$

where $DIBL$ is a coefficient modeling the Drain-Induced Barrier Lowering effect, $V_{thermal}$ is the thermal voltage, n is the sub-threshold slope coefficient (≈ 1.5), and λ is a constant connected to $DIBL$ (≈ 0.16).

The $DIBL$ effect is related to the reduction of the threshold voltage as a function of the drain voltage. When we lower the supply voltage, we cause an exponential reduction in sub-threshold current because of the $DIBL$ effect. As shown in Figure 46, the impact of $DIBL$ effect is important when moving from an STC multicore (16 cores) to an NTC manycore (128 cores) architecture as it counts for a significant portion of the total power of the system. The thermal voltage represents the flow of electric current and electrostatic potential across a p-n junction based on the temperature (T) and is calculated as follows:

$$V_{thermal} = k \times T \quad (13)$$

where k is the Boltzmann's constant (= 8.617 * 10^{-5}) and T corresponds to the room temperature in Kelvins (= 297.35K).

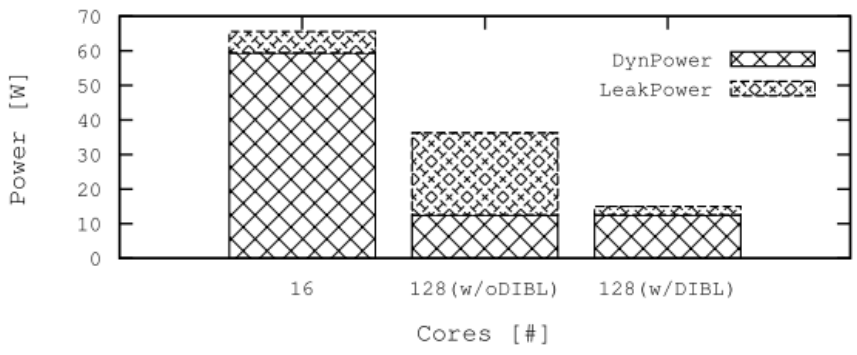


Figure 46: The Power breakdown of an STC-16core and an NTC-128core architecture with and without DIBL effect [60]

During the scaling process, we first determine the cluster size that will work in NTC regime. The cluster size, as shown in Figure 47, is calculated as follows:

$$Cluster\_size_{NTC} = Number\_VI_{NTC} \times PEs_{per\_VI} \quad (14)$$

Scaling is done for a range of $V_{dd,NTC}$ between 0.45 to 0.85V during which we compute the following values:

- The f_{NTC}, based on equation (9), of the PEs that belong to the VI of the cluster that works in NTC regime.
- The DP and LP power scaling factors, based on equations (10) and (11), for every MAC and SRAM.
- The dynamic and leakage power for every MAC and SRAM in NTC as follows:

$$Dynamic\_power_{NTC} = SF_{DP} \times Dynamic\_power_{STC} \quad (15)$$

$$Leakage\_power_{NTC} = SF_{LP} \times Leakage\_power_{STC} \quad (16)$$

This results to the calculation of the power in NTC regime with the following equation:

$$Power_{NTC} = Dynamic\_power_{NTC} + Leakage\_power_{NTC} \quad (17)$$

- The energy consumption of each MAC in NTC as follows:

$$MAC\_energy_{NTC} = \frac{MAC\_power_{NTC}}{f_{NTC}} \quad (18)$$

- The leakage power consumption of a MAC in NTC, in case of power gating the idle PEs, which equals to:

$$Leakage\_power_{per\ idle\ PE,NTC} = \frac{Leakage\_power_{NTC}}{1000} \quad (19)$$

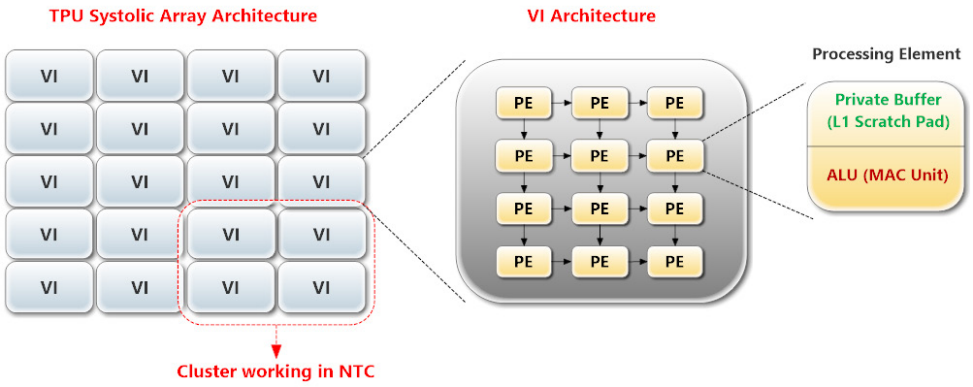


Figure 47: The NTV-DNN TPU based systolic array architecture with a cluster of VIs working in NTC

Additionally, for each layer of the examined DNN model, and for the range of $V_{dd,NTC}$ mentioned above, we calculate the power consumption in NTC as follows:

- Network On Chip (NOC) (see Figure 45) consists of a structure of routers and links, implementing a packet-switched communication fabric between the PEs and the L2 shared buffer memory. The power consumption of NOC per layer in NTC, is calculated as follows:

$$NOC\_power_{per\ layer,NTC} = NOC\_power\_dynamic_{NTC} + NOC\_power\_leakage_{NTC} + NOC\_power\_remaining_{STC} \quad (20)$$

where $NOC\_power\_dynamic_{NTC}$, $NOC\_power\_leakage_{NTC}$ are the dynamic and leakage power in NTC accordingly, while $NOC\_power\_remaining_{STC}$ is the remaining power of NOC for the PEs working in STC which is calculated according to the following equation:

$$NOC\_power\_remaining_{STC} = \left(Num\_PEs - Cluster\_size_{NTC}\right) \times \frac{NOC\_power_{per\ layer,STC}}{Num\_PEs} \quad (21)$$

where Num_PEs is the total number of PEs of the 16 x 16 TPU based accelerator and $NOC\_power_{per\ layer,STC}$ is the power of NOC per layer in STC.

- For the L1 SRAMs (scratchpad memory), which is located inside each PE, the power consumption per layer is calculated as follows:

$$L1\_power_{per\ layer,NTC} = L1\_power\_utilized_{NTC} + L1\_power\_remaining_{NTC} + L1\_power\_remaining_{STC} \quad (22)$$

where $L1\_power\_utilized_{NTC}$ is the power consumption of L1 SRAMs obtained from the multiplication of the number of active PEs ($Num\_PEs\_utilized$) that were used for MAC calculations in NTC by the required SRAM size in bytes ($L1\_size\_req$) and by the L1 power in NTC per cell (byte), as follows:

$$L1\_power\_utilized_{NTC} = Num\_PEs\_utilized \times L1\_size\_req \times L1\_power_{NTC} \quad (23)$$

The $L1\_power\_remaining_{NTC}$ addresses to the power consumption of L1 SRAMs obtained from the calculation of the remaining number of active PEs ($Cluster\_size_{NTC} - Num\_PEs\_utilized$) in NTC that rest idle, with ($Cluster\_size_{NTC} - Num\_PEs\_utilized$) > 0 , by the $L1\_size\_req$, which is the required SRAM size in bytes, and by the L1 leakage power consumption of each cell of SRAM, as follows:

$$L1\_power\_remaining_{NTC} = (Cluster\_size_{NTC} - Num\_PEs\_utilized) \times L1\_size\_req \times L1\_leakage\_power_{NTC} \quad (24)$$

Finally, the remaining power consumption of L1 SRAMs in STC equals to:

$$L1\_power\_remaining_{STC} = (Num\_PEs - Cluster\_size_{NTC}) \times L1\_size\_req \times L1\_leakage\_power_{STC} \quad (25)$$

where ($Num\_PEs - Cluster\_size_{NTC}$) is the total number of the idle active PEs that work in STC and $L1\_leakage\_power_{STC}$ is the L1 leakage power consumption of each cell of SRAM in STC ($V_{dd,STC} = 0.9V$). To avoid errors, the V_{dd} supply voltage of L1 is scaled only above a specific retention supply voltage $\left(V_{dd\_retention} = 0.6V \right)$ which assures that the SRAM is error resilient [61].

- For the L2 shared buffer SRAM memory, to avoid data corruption, the power consumption in NTC equals STC in case of $Cluster\_size_{NTC} < Num\_PEs$. That means:

$$L2\_power_{per\ layer,\ NTC} = L2\_size\_req \times L2\_power_{STC} \quad (26)$$

In case that all SoC works in NTC, then the power consumption of L2 equals to:

$$L2\_power_{per\ layer,NTC} = L2\_size\_req \times L2\_power\_ret_{NTC} \quad (27)$$

with $L2\_size\_req$ the required L2 SRAM size in bytes and $L2\_power\_ret_{NTC}$ the power consumption of each cell of L2 SRAM for the same retention voltage $\left(V_{dd}\_retention = 0.6V \right)$ as for L1.

- For the active PEs (MACs), the power consumption for each layer is calculated as follows:

$$PEs\_power_{per\ layer,NTC} = PEs\_power\_utilized_{NTC} + PEs\_power\_remaining_{NTC} + PEs\_power\_remaining_{STC} \quad (28)$$

where $PEs\_power\_utilized_{NTC}$ is the power consumed from the active PEs ($Num\_PEs\_utilized$) in NTC that were used for MAC calculations multiplied by the power consumption of each MAC unit working in NTC.

$$PEs\_power\_utilized_{NTC} = Num\_PEs\_utilized \times PE\_power_{NTC} \quad (29)$$

The remaining power consumption of the active PEs in NTC that stay idle, equals to the total number of these PEs by the leakage power consumption of a PE working in NTC, as follows:

$$PEs\_power\_remaining_{NTC} = (Cluster\_size_{NTC} - Num\_PEs\_utilized) \times PE\_leakage\_power_{NTC} \quad (30)$$

In addition, the STC power consumption of the active PEs that stay idle is product of the multiplication of their total number by the leakage power consumption of each one working in STC.

$$PEs\_power\_remaining_{STC} = (Num\_PEs - Cluster\_size_{NTC}) \times PE\_leakage\_power_{STC} \quad (31)$$

In conclusion, the total power consumption per layer in NTC equals to:

$$Total\_power_{per\ layer,NTC} = PEs\_power_{per\ layer,NTC} + L1\_power_{per\ layer,NTC} + L2\_power_{per\ layer,NTC} + NOC\_power_{per\ layer,NTC} \quad (32)$$

The energy consumption of each layer of the DNN model is calculated based on the computation

energy, which corresponds to the energy of performing MACs, and the data movement energy, which is linked to the movement of data (read/write) to and from the SRAMs [72] , that is:

$$E_{layer} = E_{comp} + E_{data} \tag{33}$$

According to equation (33) we have the following:

- For the L1 scratchpad memory, the NTC energy consumption is calculated as follows:

$$L1\_energy_{per\ layer,\ NTC} = L1\_read\_write\_ops \times L1\_energy\_mult \times MAC\_energy_{NTC} \tag{34}$$

where $L1\_energy\_mult$ is a technology dependent multiplier taken from MAESTRO [57], $L1\_read\_write\_ops$ is the total of read/write operations to L1, and MAC_energy_{NTC}, which comes from (18), is the MAC energy cost of accessing one bit at that memory level in NTC [72].

- For the L2 shared buffer, the energy consumption is calculated in the same way as for L1, but, in order to assure that there will be no errors produced during the data movement in L2 for NTC regime, we accept the assumption that:
  - If $Cluster\_size_{NTC} < Num\_PEs$, that is not all PEs work in NTC, then:

$$L2\_energy_{per\ layer,\ NTC} = L2\_read\_write\_ops \times L2\_energy\_mult \times MAC\_energy_{STC} \tag{35}$$

  - Else, if all PEs work in NTC:

$$L2\_energy_{per\ layer,\ NTC} = L2\_read\_write\_ops \times L2\_energy\_mult \times MAC\_energy_{NTC} \tag{36}$$

- In terms of energy consumption carried out by the MAC unit inside each PE, the formula that calculates the total amount of energy per layer in NTC is the following:

$$PEs\_energy_{per\ layer,\ NTC} = MAC\_ops \times MAC\_energy_{NTC} \tag{37}$$

This leads to the calculation of total energy consumption of each layer in NTC as follows:

$$\begin{aligned} Total\_energy_{per\ layer,\ NTC} &= L1\_energy_{per\ layer,\ NTC} + L2\_energy_{per\ layer,\ NTC} \\ &+ PEs\_energy_{per\ layer,\ NTC} + Energy\_overhed \end{aligned} \tag{38}$$

where $Energy\_overhead$ is, according to our assumption, the energy consumed because of the power gating of the PEs that work in f_{STC} frequency and finish the MAC calculations prior of these that work in f_{NTC}.

The performance of each DNN layer in NTC is calculated with the above equation:

$$Perf ormance_{per\ layer, NTC} = \frac{Runtime_{per\ layer, STC}}{f_{NTC}} \tag{39}$$

where $Runtime_{per\ layer, STC}$ is the total number of cycles executed per layer during inference and f_{NTC} is the working frequency in NTC.

The V_{dd} supply voltage is selected according to the problem each researcher is dealing with. For example, if we want to find the V_{dd} for which we get the minimum energy consumption, then we have:

$$min\left\{\left(\sum_{i=1}^{k}\left(Total\_energy_{per\ layer, NTC}\right)_{k}\right)_{j}\right\} \tag{40}$$

with k the total number of layers for the examined DNN model (e.g., k = 8 for AlexNet) and j the $V_{dd, NTC} \in [0.45,\ 0.85]$ with a step of 0.5V.

5.4 NTV-DNN error model

When scaling the V_{dd} supply voltage of a PE unit, this can cause erroneous MAC operations which can lead to a great impact on the accuracy of a DNN model. To measure and assess this impact, we created an NTV-DNN error model based on PyTorchFI [59]. As displayed in Figure 48, the operation of this model is based on four functions, whose detailed description can be seen in Table 4. Our main goal is to produce errors only in the level of MAC calculations, that is to the fmap outputs produced from the MAC units working in NTC.

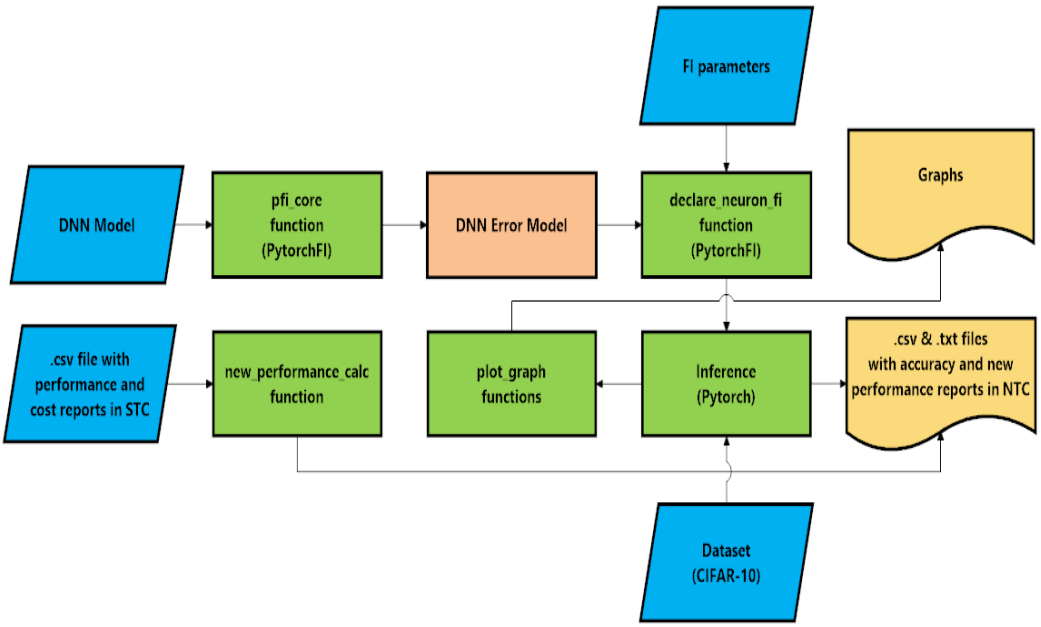


Figure 48: An overview of the NTV-DNN Error Model

Table 4: List of functions in NTV-DNN Error Model

Function name	Input	Parameters	Output	Operation
pfi_core	DNN model	DNN_model, batch_size, input_shape	A DNN fault injection model.	Creates an error model to perform error injections dynamically (i.e., during an inference).
declare_neuron_fi	FI parameters	batch, layer_num, dim1, dim2, dim3, value	A DNN fault injection model.	Declares a neuron injection by passing the location of the error.
new_performance_calc	A .csv file (MAESTRO)	file, f_ntc	A .txt file with new performance.	Calculates the new performance of the examined DNN Model based on a frequency scaling schema.
plot_graph	Vdd in NTC, Frequency in NTC, accuracy	vdd_ntc, ntc_freq, new_ntc_freq, accuracy_before_FI, accuracy_after_FI	The graphs of accuracy in NTC.	Plots the accuracy of the DNN Model in NTC according to frequency scaling.

For each V_{dd} supply voltage in NTC, an f_{NTC} is calculated through (9). This frequency is used to produce our frequency scaling scheme which uses a step of 5MHz. To decide when to produce an error, we proceeded in a path distribution analysis by designing a 45nm technology node ALU, using the Synopsys design software [73] and the TSMC cell library. We discovered that, when increasing

the frequency of the designed ALU, keeping V_{dd} stable, a delay is produced during calculations of 29th and 31st bit of a 32-bit floating point number leading to an erroneous result. According to the impact of the delay caused by the increase of frequency, we produce a bit flip (of 29th bit, 31st bit or both) on the 32-bit floating point number output result of each selected neuron. For the $f_{stc} = 700MHz$ operating frequency for each PE of our TPU-based accelerator, the 29th bit and 31st bit are calculated after 1.39nsec and 1.33nsec respectively, so we have:

$$f_{29th\ bit,STC} = \frac{1}{1.39 \times 10^{-9}} \simeq 0.719 \times 10^{9} Hz = 719MHz \quad (40)$$

$$f_{31th\ bit,STC} = \frac{1}{1.33 \times 10^{-9}} \simeq 0.752 \times 10^{9} Hz = 752MHz \quad (41)$$

Therefore, for each NTC frequency during scaling, an $f_{29th\ bit,NTC}$ and $f_{30th\ bit,NTC}$ is calculated. If the new selected frequency overpasses one or both of the above 29th bit and 31st bit frequency values, then each of these bits is flipped in the output locations of neurons fed to our Error Model, through inference using the CIFAR-10 dataset which contains 10.000 test images.

```
=================== PYTORCHFI INIT SUMMARY ====================

Layer types allowing injections:
----------------------------------------------------------------
   - Conv2d

Model Info:
----------------------------------------------------------------
   - Shape of input into the model: (3 224 224)
   - Batch Size: 4
   - CUDA Enabled: True

Layer Info:
----------------------------------------------------------------
Layer #            Layer type       Dimensions         Output Shape
----------------------------------------------------------------
    0                 Conv2d                4      [1, 64, 55, 55]
    1                 Conv2d                4     [1, 192, 27, 27]
    2                 Conv2d                4     [1, 384, 13, 13]
    3                 Conv2d                4     [1, 256, 13, 13]
    4                 Conv2d                4     [1, 256, 13, 13]
================================================================
```

Figure 49: PyTorchFI output summary of the AlexNet error model

The locations of the neurons, where the perturbation takes effect, are taken through the Fault Injection (FI) parameters fed from the NTC Analysis Tool. Each location of FI depends on the mapping of the calculations between filter weights and input activations to each VI of the TPU-based accelerator. Figure 51 displays for time step 0, an example of mapping of the data to each of the 8 VIs during inference for the 1st conv2d layer of AlexNet DNN model. The dataflow strategy chosen is the *kcp_ws* and the dimension of batch N, which is of size 4, equals to 1. The *kcp_ws* mapping is using multi-level parallelism via clustering. This is achieved by creating clusters of PEs during the computation of partial sums. As we see in Figure 50, the *kcp_ws* mapping divides the total number of PEs into clusters of size 32, which equals the size of our designated VIs. Table 5 displays an

example of the mapping of data for *input* and *weight tensors* in each cluster. The numbers are indices of the data in each tensor. Each tensor is in essence an fmap. As we observe for the 1st conv2d layer of AlexNet, only 3 of the total 32 PEs of the cluster (or VI) are used during the MAC calculations, as the dimension of the *input channel* (C) for the input and filter tensors equals to 3.

```
Network torchvision.models.alexnet {
Layer Conv2d-1 {
Type: CONV
Stride { X: 4, Y: 4 }
Dimensions { K: 64, C: 3, R: 11, S: 11, Y: 224, X: 224 }
Dataflow {
        // This is a NVDLA-like dataflow
        SpatialMap(1,1) K;
        TemporalMap(32,32) C;
        TemporalMap(Sz(R),Sz(R)) R;
        TemporalMap(Sz(S),Sz(S)) S;
        TemporalMap(Sz(R),1) Y;
        TemporalMap(Sz(S),1) X;
        Cluster(32, P);
        SpatialMap(1,1) C;
        TemporalMap(Sz(R),1) Y;
        TemporalMap(Sz(S),1) X;
        TemporalMap(Sz(R),Sz(R)) R;
        TemporalMap(Sz(S),Sz(S)) S;
}
```

Figure 50: The kcp_ws NVDLA-like dataflow for the 1st conv2d layer of AlexNet

Table 5: The mapping of data on the 1st cluster for the kcp_ws dataflow

Cluster 1							
	Time step = 0 Layer: Conv2d-1 Cluster size: x = 32	**PE 1**	**PE 2**	**PE 3**	**PE 4**	**PE 5**	**PE x**
Input Tensor	Batch (N)	1	1	1	-	-	-
	Input Channel (C)	1	2	3	-	-	-
	Input Height (Y)	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	-	-	-
	Input Width (X)	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	-	-	-
Weight Tensor	Output Channel (K)	1	1	1	-	-	-
	Input Channel (C)	1	2	3	-	-	-
	Weight Height (R)	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	-	-	-
	Weight Width (S)	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11	-	-	-

Output Tensor	Batch (N)	1
	Output Channel (K)	1
	Output Height (Y')	1
	Output Width (X')	1

For $cluster\_size\_ntc = 64$, which means that two of our VIs (each VI englobes 32 PEs) work in NTC, the FI_step equals to:

$$FI\_step = \frac{Num\_PEs}{Cluster\_size\_NTC} = \frac{256}{64} = 4$$

As we see in Figure 51, every perturbation takes place with a step of 4, meaning that for every $K = K + FI\_step$, $with\ 0 \leq K \leq 64$ (for the 1st layer), an FI takes place till all the elements of the 55 x 55 tensor output, that come the MAC calculations of the PEs working in NTC, are perturbed. Table 6 displays the part of Python code, which is part of the fi_injection_pos function, that produces the locations of neurons where the perturbation take effect. This part of code addresses to the 1st conv2d layer of AlexNet. A different for loop exists for every conv2d layer. Figure 49 displays the layers and output shapes of the error model produced from PyTorchFI based on AlexNet.

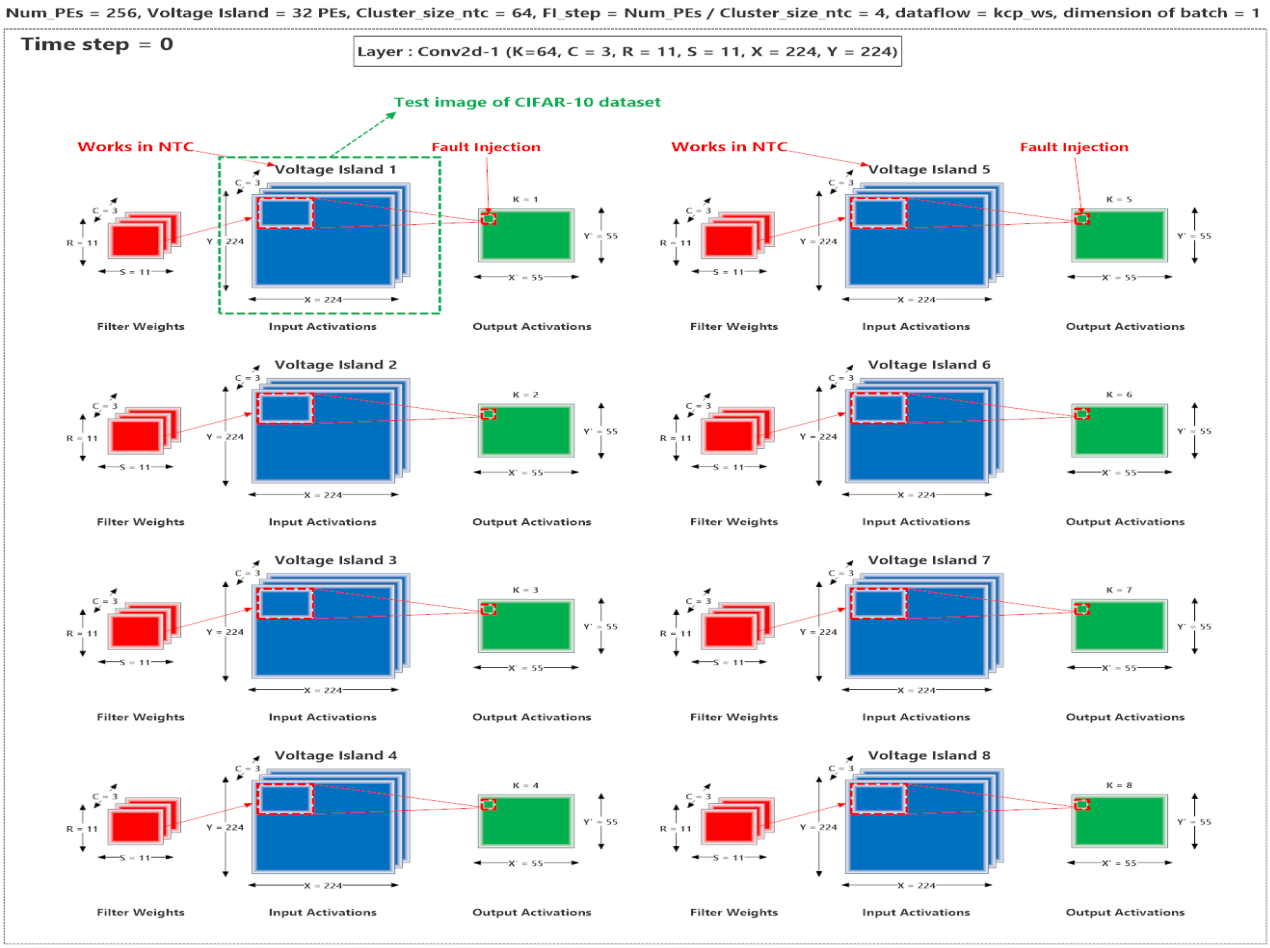


Figure 51: The FI procedure in the 1st conv2d layer of AlexNet

Table 6: FI locations for the 1st Conv2d layer of AlexNet

Python Code	Description

for g in range(0, 64, k_step):
for f in range(55):
for w in range(55):
layer.append(0)
C_out.append(g)
X_out.append(f)
Y_out.append(w)
	This for loop produces the FI locations (see example Figure 50) for the 1st Conv2d layer of AlexNet and is part of the fi_injection_pos function. The k_step is equal to the FI_step. The values of every list (layer, C_out, X_out, Y_out) are fed to the declare_neuron_fi function. The variables g, f, w represent K=64, X'=55, Y'=55 respectively which are the dimensions of output tensor of the 1st layer.

It is proved that in general DNN accelerators have a certain degree of fault tolerance due to the fault tolerance of DNN algorithms themselves [62]. According to S. Hong et al. [63], who studied the vulnerability of DNN parameters to single bit flips, as for the direction of bit-flip, only $0 \rightarrow 1$ flip causes large accuracy loss. A $1 \rightarrow 0$ flip can only reduce the parameter value. Due to the normal distribution of parameters, most parameters are inside $[-1, 1]$ range. Hence, a $1 \rightarrow 0$ flip in exponent bit can reduce the magnitude of a parameter by no more than one. Along similar lines, both $0 \rightarrow 1$ or $1 \rightarrow 0$ flips in sign bit cannot lead to large accuracy loss since they alter the magnitude by no more than two. By comparison, a $0 \rightarrow 1$ flip in exponent bits dramatically increases the value of a parameter. Hence, during inference, unduly high activation produced by the faulty parameter value overrides the remaining activations. They find that nearly 50% of the parameters of the DNNs are vulnerable to single bit-flips.

5.5 NTV-DNN assessment tool flow

In this section we present the tools that were used to build out NTV-DNN framework. All our tools are open source and are carefully selected to implement each functionality.

5.5.1 PyTorch: An open-source machine learning framework

PyTorch is one of the biggest libraries in deep learning research. It is based on the Python programming language and the Torch library, makes debugging easy and is consistent with other popular scientific libraries [64]. During computations, this optimized framework uses tensors that are accelerated by GPUs and CPUs. Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, tensors are used to encode the inputs and outputs of a model, as well as the model's parameters [65]. Figure 52 displays the training of a CNN Model using PyTorch.

Efficient interoperability is one of the key aspects of PyTorch, because it allows users to use Python's vast ecosystem of libraries as part of their own projects. It, for example, provides a mechanism for converting NumPy arrays to PyTorch tensors using the *torch.from_numpy()* function and *numpy()* tensor method. As a result, those operations are extremely cheap and take constant time regardless of how large the converted arrays are. Most importantly, users are free to replace any PyTorch component that does not meet their project's needs or performance requirements. They are all intended to be completely interchangeable, and PyTorch takes great care not to impose a specific solution.

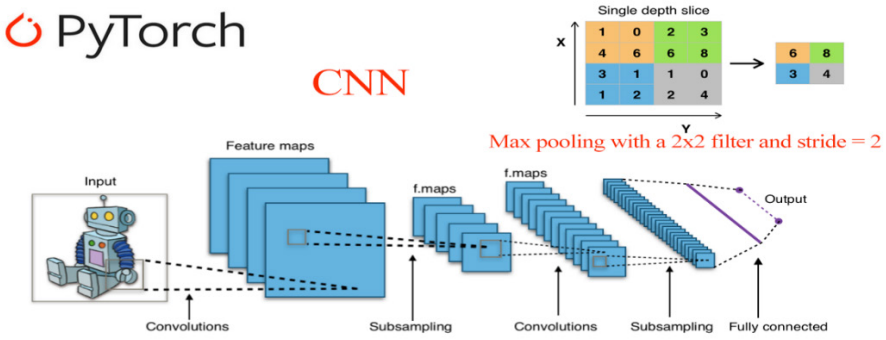


Figure 52: Training a CNN Model with PyTorch [67]

PyTorch keeps its control (e.g., program branches, loops) and data flow completely separate (e.g., tensors and the operations performed on them). The control flow resolution is handled by Python and optimized C++ code running on the host CPU, resulting in a linear sequence of operator invocations on the device. Operators can run on either the CPU or the GPU. PyTorch is designed to run operators asynchronously on GPUs by using the Nvidia CUDA stream mechanism [66] to queue CUDA kernel invocations to the GPU's hardware FIFO. This enables the system to run Python code on the CPU alongside tensor operators on the GPU. Because tensor operations typically take a long time, we can saturate the GPU and achieve peak performance even in an interpreted language with relatively high overhead, such as Python. It is worth noting that this mechanism is nearly invisible to the user. Unless they implement their own multi-stream primitives, the library handles all CPU-GPU synchronization.

5.5.2 MAESTRO cost model

The efficiency of an accelerator is determined by three factors: mapping, deep neural network (DNN) layers, and hardware, constructing extremely complicated DNN accelerator design space. MAESTRO, who's high-level overview is shown in Figure 53, is an analytical cost model tool that aims in guiding a DNN accelerator design for better efficiency [57]. As inputs, MAESTRO receives a list of DNN model descriptions and hardware resource information, as well as mapping described in a data-centric representation we propose. The data-centric representation is made up of three

directives that allow for concise mapping descriptions in a compiler-friendly format. MAESTRO quickly analyzes various forms of data reuse in an accelerator based on inputs and generates more than 20 statistics as outputs, including total latency, energy and throughput.

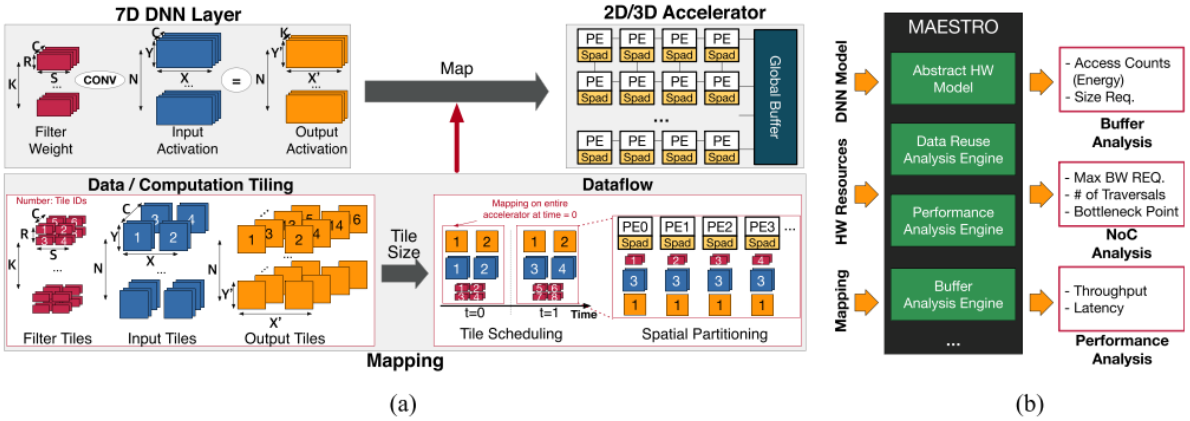


Figure 53: (a) An overview of mapping CONV2D to an accelerator (b) High-level Tool flow of MAESTRO [57]

MAESTRO consists of five preliminary engines: Tensor, cluster, reuse, performance analysis, and cost analysis. It supports, as seen in Figure 54, a diverse set of accelerators, including global shared scratchpad (L2 SRAM), local PE scratchpad (L1 scratchpad), NoC, and a PE array organized into any number of hierarchies or dimensionalities. MAESTRO implements a hardware design space exploration (DSE) tool that searches four hardware parameters (the number of PEs, L1 buffer size, L2 buffer size, and NoC bandwidth) optimized for either energy efficiency, throughput, or energy-delay-product (EDP) within given hardware area and power constraints. The DSE tool takes the same inputs as MAESTRO, but with hardware area/power constraints and the area/power of building blocks synthesized with the target technology. A float/fixed point multiplier and adder, bus, bus arbiter, and global/local scratchpad are implemented in RTL and are all synthesized using 28-nm technology to reduce the cost of building blocks. Regression is also used to fit the costs of the bus and arbiter into a linear and quadratic model because the bus cost increases linearly and the arbiter cost increases quadratically (e.g., matrix arbiter).

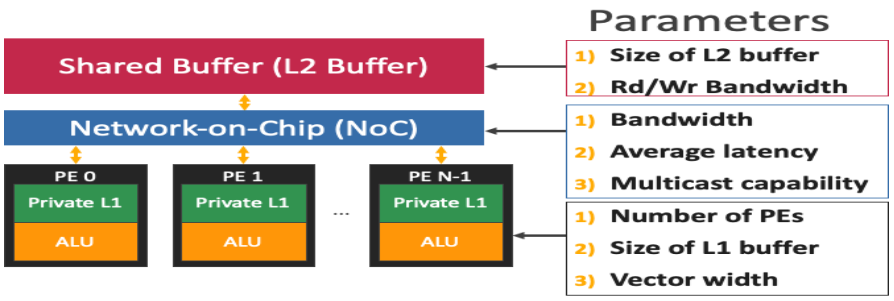


Figure 54: An overview of the supported hardware in MAESTRO [68]

5.5.3 PyTorchFI runtime fault injector

Searching for an effective and reliable open-source perturbation tool is not easy. During our research, we found many FI tools but most of them were deprecated as they use a Python version which is below version 3 (e.g., Ares Fault Injection Framework). PyTorchFI is a very promising DNN runtime perturbation tool for the popular PyTorch deep learning platform [59]. As seen in Figure 55, users can implement PyTorchFI to perform runtime perturbations on DNN weights or neurons. It is designed with the programmer in mind, with a simple and easy-to-use API that can be used with as few as three lines of code. It also has an extensible interface that allows researchers to choose from various perturbation models (or design their own custom models), allowing them to study the propagation of hardware error (or general perturbation) to the software layer of the DNN output.

As a first step, PyTorchFI can be inserted into our project as a python package with pip install pytorchfi and contains:

- Core.py: This file contains the core functionality for fault injections. We have tweaked this file to implement our fault injection schema which consists of flipping the 29th and 31st bits according to the selection of the error policy from our Error Model tool.
- Error_models.py: It provides different error models out-of-the-box for use.

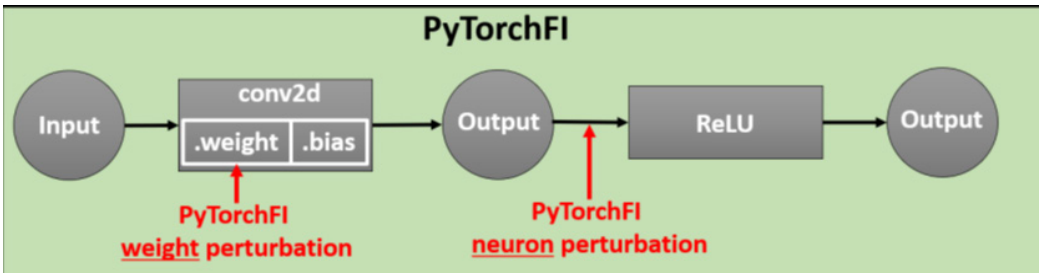


Figure 55: An overview of PyTorchFI [59]

As a second step, initializing takes place during which PyTorchFI selects the model on which the perturbations will be performed. Other arguments include the height and width of the input image, as well as optional parameters such as batch size, model data type (e.g., FP32 or FP16), and whether to run on the CPU or GPU. PyTorchFI then performs a single dummy inference to profile the model and collects all the network's hyperparameters, such as the number of layers, filter sizes, and feature map sizes. This information is used to ensure that perturbations are legal and to provide the end user with detailed debugging messages.

Finally, as a third step, we choose a perturbation model and a location for the perturbation. The user is provided with a default set of perturbation models (from *error_models.py*) to choose from, such as a random value, a single bit flip, or a zero-value. For our project, we used only the tweaked

core.py mechanism. The user can also easily create their own perturbation model. Along with the perturbation model, the user must specify the location of the perturbed weight/neuron. This can be a single location (specified in the tensor by the layer, feature map, and neuron's coordinate position) or multiple locations to cause multiple perturbations throughout the network. The user can also choose whether to apply the same perturbation to all elements in a batch or to apply a different perturbation to each element.

The actual perturbation happens during runtime by taking the location of the incorrect neuron/weight and appending it to a list of tensor positions to change. The forward hook will then iterate through all of the locations on each layer, corrupting the corresponding value based on the perturbation model chosen.

5.5.4 CIFAR-10 dataset

As a dataset for the training and inference of our DNN Models we chose the Canadian Institute For Advanced Research - 10 (CIFAR-10). This dataset consists of 60000 32 x 32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images [69]. The dataset is divided into five training batches and one test batch, each of which contains ten thousand images. The test batch contains exactly 1000 images from each class, chosen at random. The remaining images are distributed in random order in the training batches, but some training batches may contain more images from one class than another. The training batches each contain exactly 5000 images from each class.

This collection of images is commonly used to train machine learning and computer vision algorithms, and CNNs seems to be the best at recognizing the images in CIFAR-10. The archive of this dataset contains some data batch files as well as a test batch file. Each of the batch files contains a dictionary with the following elements:

- Data: This is a 10000 x 3072 NumPy array. Each row of the array stores a 32 x 32 color image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image.
- Labels: a list of 10000 numbers in the range 0-9. The number at index *i* indicates the label of the i^{th} image in the array data.

The file of the dataset called *batches.met*, is a Python dictionary object and contains the label names that correspond to a 10-element list which gives meaningful names to the numeric labels in the labels array described above.

5.6 Experimental Setup

We use MAESTRO to simulate a TPUv1 systolic array accelerator of 28nm. To run MAESTRO, we must first install all the package dependencies (e.g., g++, scons). We then run the *scons* command in terminal to compile the code and, we set the parameters of MAESTRO which are displayed in Table 7.

Table 7: List of parameters input to MAESTRO

Parameter type	Input	Description
HW_file	accelerator_1.m	The hardware parameters file.
Mapping_file	alexnet_pytorch_kcp_ws_64.m	The target dataflow and layer description file.
print_res	true	MAESTRO prints out detailed cost information to the screen.
print_res_csv_file	true	MAESTRO prints out a csv file that contains various statistics.
print_log_file	false	MAESTRO prints out a log file that contains various information of detailed computation patterns to "log.txt".

The hardware parameters file *accelerator_1.m* contains the values listed in Table 8.

Table 8: List of parameters of accelerator_1.m file

Parameter type	Input	Description
num_pes	256	Number of PEs.
l1_size_cstr	1024 (bytes)	L1 buffer size constraint.
l2_size_cstr	32768 (bytes)	L2 buffer size constraint.
noc_bw_cstr	1024 (bytes)	NoC bandwidth constraint.
offchip_bw_cstr	2048 (bytes)	Off-chip memory bandwidth constraint.

The parameters of L1 and L2 buffer are optional and if not specified, MAESTRO will assume infinite resources and compute the required amount of resources, which are reported in the .csv file. MAESTRO will also check if the constraints are met. If not, it will print out warning message. Furthermore, we tweaked the *API_user-interface-v2.hpp* and *DSE_csv_writer.hpp* of MAESTRO source code to export in the .csv file of the statistics the cluster size of the dataflow mapping and the energy consumption of L1, L2 and MAC in STC. In Table 9 we see the values that should be corrected in each source file considering that the clock speed of our TPU-based accelerator is $f = 700MHz$ and each PEs contains only one ALU.

Table 9: List of values altered in MAESTRO source code

File	Values altered	Description
options.hpp	num_simd_lanes = 1	The number of ALUs in each PE.
DSE_cost-database.hpp	$\text{mac\_energy} = \frac{1.2223\text{mW}}{700\text{MHz}} = 0.00345\text{nJ}$	The energy consumption of an ALU (MAC unit) in STC.

Next, for each DNN Model we generate a MAESTRO DNN Model file from Pytorch with *frameworks_to_modelfile_maestro.py*, as shown in Table 10, with input size 3 x 224 x 224. Since we

Table 10: List of MAESTRO DNN Models

PyTorch Model	MAESTRO DNN Model
AlexNet	alexnet_pytorch.m
GoogLeNet	googlenet_pytorch.m
ResNet-18	resnet18_pytorch.m
SqueezeNet	squeezenet1_1_pytorch.m

are going to use CIFAR-10 dataset for the training of our DNN Models, which has 60000 32 x 32 color images in 10 classes, we must alter the *K* output dimension of the last layer as shown in Table 11. The mapping analysis convention can be seen in Figure 56.

Table 11: The dimensions of the last layer for each MAESTRO DNN Model

MAESTRO DNN Model	Layer	Dimensions
alexnet_pytorch.m	Linear-8	K = 10, C = 1024, R = 1, S = 1, Y = 1, X = 1
googlenet_pytorch.m	Linear-64	K = 10, C = 1024, R = 1, S = 1, Y = 1, X = 1
resnet18_pytorch.m	Linear-21	K = 10, C = 512, R = 1, S = 1, Y = 1, X = 1
squeezenet1_1_pytorch.m	Conv2d-26	K = 10, C = 512, R = 1, S = 1, Y = 13, X = 13

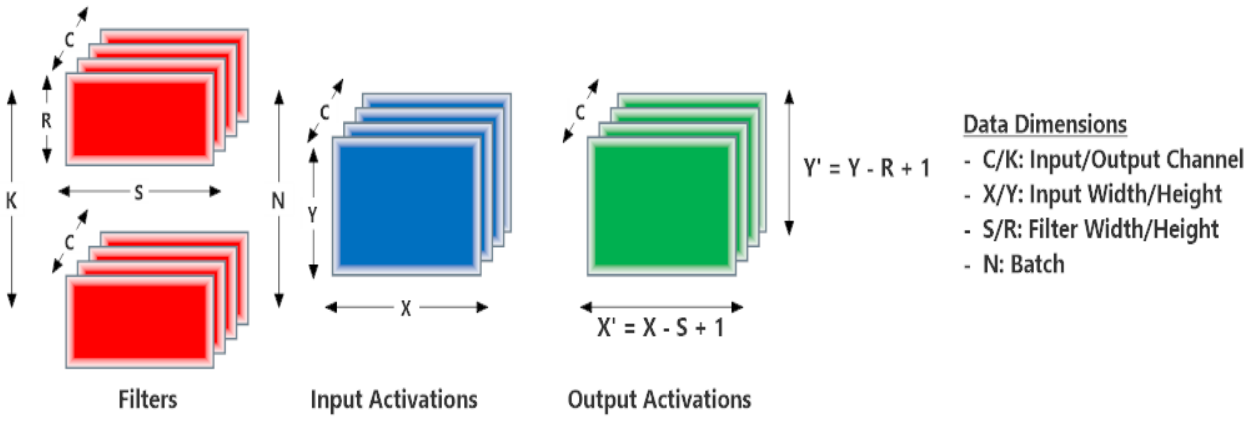


Figure 56: The mapping analysis convention

As next step, we create four MAESTRO Mapping files using the MAESTRO DNN Model file and each specific dataflow, for every DNN Model, as displayed in Table 12. The mappings (dataflow strategy) used is shown in Table 13.

Table 12: List of MAESTRO Mapping files for each DNN Model

PyTorch Model	MAESTRO DNN Model
AlexNet	alexnet_pytorch_kcp_ws.m alexnet_pytorch_rs.m alexnet_pytorch_maeri.m alexnet_pytorch_yxp_os.m alexnet_pytorch_yrp_rs.m
GoogLeNet	googlenet_pytorch_kcp_ws.m googlenet_pytorch_rs.m googlenet_pytorch_maeri.m googlenet_pytorch_yxp_os.m googlenet_pytorch_yrp_rs.m
ResNet-18	resnet18_pytorch_kcp_ws.m resnet18_pytorch_rs.m resnet18_pytorch_maeri.m resnet18_pytorch_yxp_os.m resnet18_pytorch_yrp_rs.m
SqueezeNet	squeezenet1_1_pytorch_kcp_ws.m squeezene1_1_pytorch_rs.m squeezenet1_1_pytorch_maeri.m squeezenet1_1_pytorch_yxp_os.m squeezenet1_1_pytorch_yrp_rs.m

Table 13: Mappings used to create each MAESTRO Mapping file

Partitioning Strategy	Mapping	Characteristics
Eyeriss-like row stationary dataflow [70]	**SpatialMap**(1, 1) Y' **TemporalMap**(1, 1) X' **TemporalMap**(1, 1) C **TemporalMap**(16, 16) K **TemporalMap**(Sz(R), Sz(R)) R **TemporalMap**(Sz(S), Sz(S)) S **Cluster**(Sz(R), P) **SpatialMap**(1,1) Y **SpatialMap**(1,1) R **TemporalMap**(Sz(S), Sz(S)) S	• Row-stationary • Reconfigures the computation mapping of a given shape. • High temporal reuse of input activation and filter
MAERI-like dataflow [71]	**TemporalMap**(1, 1) C **SpatialMap**(1, 1) K **TemporalMap**(1, 1) Y' **TemporalMap**(1, 1) X' **TemporalMap**(Sz(R), Sz(R)) R **TemporalMap**(Sz(S), Sz(S)) S // This is a VN of size Sz(R) x	• Constructs a Virtual Neuron (VN). • Maps VNs one by one over the PEs. • Configures the Augmented Reduction Tree (ART) for the VNs to operate in parallel.

	Sz(S) **Cluster**(Sz(R), P) **SpatialMap**(1,1) Y **SpatialMap**(1, 1) R **Cluster**(Sz(S), P) **SpatialMap**(1, 1) X **SpatialMap**(1, 1) S	
YX-Partitioned (YX-P)	**TemporalMap**(1, 1) K **SpatialMap**(Sz(R), 1) Y **TemporalMap**(Sz(S), 8) X **TemporalMap**(1, 1) C **TemporalMap**(Sz(R), Sz(R)) R **TemporalMap**(Sz(S), Sz(S)) S **Cluster**(8, P) **SpatialMap**(Sz(S),1) X	• High temporal reuse of filter. • Better spatial reuse opportunities. • 2D activation (X and Y) parallelism. • Output-stationary.
YR-Partitioned (YR-P)	**TemporalMap**(2, 2) C **TemporalMap**(2, 2) K **SpatialMap**(Sz(R), 1) Y **TemporalMap**(Sz(S), 1) X **TemporalMap**(Sz(R), Sz(R)) R **TemporalMap**(Sz(S), Sz(S)) S **Cluster**(Sz(R), P) **SpatialMap**(1,1) Y **SpatialMap**(1,1) R	• High temporal reuse of input activation and filter. • Spatial reduction opportunities. • Activation row (Y) and filter column (S) parallelism. • Row-stationary.
KC-Partitioned (KC-P) NVDLA-like dataflow	**SpatialMap**(1, 1) K **TemporalMap**(32, 32) C **TemporalMap**(Sz(R), Sz(R)) R **TemporalMap**(Sz(S), Sz(S)) S **TemporalMap**(Sz(R), 1) Y **TemporalMap**(Sz(S), 1) X **Cluster**(32, P) **SpatialMap**(1, 1) C **TemporalMap**(Sz(R), 1) Y **TemporalMap**(Sz(S), 1) X **TemporalMap**(Sz(R), Sz(R)) R **TemporalMap**(Sz(S), Sz(S)) S	• Spatial reuse of input activation. • High spatial reduction factor (32-way) on input channel (C). • Input/Output channel (C and K) parallelism. • Weight stationary.

The NTC parameters of the NTC Analysis tool are displayed in Table 14. We consider that our TPU based accelerator has a total number of 256 PEs, forming 8 VIs (8 x 32 = 256 PEs). For example, if our *ntc_cluster_size* equals to 128 (meaning that 128 PEs work in NTC), then we have 4 VIs working in NTC.

Table 14: List of NTC parameters

Parameter type	Input	Description
power_gating	true	Simulates power gating of idle PEs.
mac_power_stc	1.2223mW	The dynamic power of a MAC (ALU) unit of a PE in STC regime for TPU v1 (28nm node).
mac_power_leak	(1 / 5) * mac_power_stc	The leakage power of a MAC (ALU) unit in STC.
mac_power_dynamic	mac_power_stc - mac_power_leak	The dynamic power of a MAC (ALU) unit in STC.
l1_energy_multiplier	1.68	Taken from MAESTRO (BASE_constants.hpp). It is the constant used for the calculation of energy consumption of SRAM.
l2_energy_multiplier	18.61	Taken from MAESTRO (BASE_constants.hpp). It is the constant used for the calculation of energy consumption of SRAM.
l1_power_stc	0.00345mW	Taken from MAESTRO (DSE_cost-database.hpp). It is the power consumption of an SRAM cell (byte).
l2_power_stc	0.00345mW	Taken from MAESTRO (DSE_cost-database.hpp). It is the power consumption of an SRAM cell (byte).
Vdd_stc	0.9V	Supply voltage of a MAC unit in STC.
Vdd_ntc	[0.45, 0.50, 0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.85] in Volts	Scaling voltage in NTC.
Vth_stc	0.4V	The nominal threshold voltage in STC.
Vth	0.4V	The threshold voltage of a MAC unit.
Vdd_retention_sram	0.6V	The Vdd retention of SRAM. The minimum required supply voltage so that the memory cells retain data while consuming less leakage power. [61]
f_stc	700MHz	The operating frequency of our TPU-based accelerator in STC.
f_stc_bit_29	719MHz	The operating frequency during calculations of the 29th bit of the 32bit IEEE-754 Floating Point result.
f_stc_bit_31	752MHz	The operating frequency during calculations of the 31st bit of the 32bit IEEE-754 Floating Point result.
b	1.5	Technology dependent contant.
K	$8.617 * 10^{-5}$	Boltzman constant (eV)
temp	297.35K	Room temperature in Kelvins
Vtherm	K * temp	The thermal voltage.
n	1.5	The Sub-threshold slope coefficient.
num_PEs	512	The total number of PEs of the TPU-based accelerator.
num_layers	8	The total number of layers of the examined DNN Model (e.g., AlexNet).
dataflow	kcp_ws	The chosen mapping (dataflow).
ntc_cluster_size	128	The total number of PEs working in NTC.
fi_step	num_PEs / ntc_cluster_size	The FI step that will be used by the fi_injection_pos function for the calculation of the FI location points.

The FI parameters of the NTV-DNN Error Model are shown in Table 15.

Table 15: List of FI parameters

Parameter type	Input	Description
batch_size1	4	The batch size (N) of the input activations (e.g., 4 images of CIFAR-10 dataset)
batch_size2	[0, 1, 2, 3]	The input parameter (batch) for the declare_neuron_fi function.
input_shape	[3, 224, 224]	The input parameter (input shape of the image) for the declare_neuron_fi function.
k_step	fi_step	The FI step.
layer	injection_list[0]	This list that contains the layer numbers of the examined DNN Model where the perturbation will occur.
batch_size2	[0, 1, 2, 3]	This list contains the batch dimension of each shape [batch, C, H, W] of the output feature map, where the injection will occur.
C_out	injection_list[1]	This list contains the C dimensions of each shape [batch, C, H, W] of the output feature map, where the injection will occur.
X_out	injection_list[2]	This list contains the X dimensions of each shape [batch, C, H, W] of the output feature map, where the injection will occur.
Y_out	injection_list[3]	This list contains the Y dimensions of each shape [batch, C, H, W] of the output feature map, where the injection will occur.

For the process of inference and training to calculate the accuracy of each DNN Model, we use the parameters listed in Table 16. Training and inference are run under Window 11 operating system, CUDA version 11.3 and NVIDIA GeForce GTX 1050 Ti as the specified CUDA device.

Table 16: List of parameters for inference and training

Parameter type	Input	Description
transform	transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224), transforms.ToTensor(), transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])	We transform every image of the CIFAR-10 dataset to meet the input_shape = [3, 224, 224].
train_data	torchvision.datasets.CIFAR10	Downloads the training images of CIFAR-10 dataset.
test_data	torchvision.datasets.CIFAR10	Downloads the test images of CIFAR-10 dataset.
trainloader	torch.utils.data.DataLoader	Loads the training images for training purposes.
testloader	torch.utils.data.DataLoader	Loads the test images for inference.
device	torch.device	Instantiating CUDA device.
AlexNet_model	models.alexnet(pretrained=True)	Loads the DNN Model from PyTorch (e.g. AlexNet). This Model is trained with CIFAR-10 dataset.
AlexNet_model	torch.load('./path')	Loads the pre-trained (with CIFAR-10 dataset) DNN Model from the specified path.

5.7 Experimental Results

In this section we present the results of our research. We first explore the VI granularity which aims to present the effect of applying different $V_{dd,NTC}$ and $Cluster\_size_{NTC}$ to power/energy consumption and execution time of each layer of our examined CNN Models. We then check how

$V_{dd,NTC}$ and $Cluster\_size_{NTC}$ affect the power/ energy efficiency and performance of each DNN Model, keeping the resources (Num_PEs) constant. We proceed our experiments in examining if there is an equivalent TPU-based accelerator in NTC who has the same performance with an accelerator that works in STC. Our study continues by comparing the efficacy of our NTV-DNN tool for different operating schemes under relaxed error. Furthermore, we analyze the impact of applying different dataflow strategies to power/energy consumption and execution time of each CNN Model, for a specific scheme of $V_{dd,NTC}$, f_{NTC}, $Cluster\_size_{NTC}$ and Num_PEs. Finally, we explore how power, energy and performance efficiency of our examined DNN Models can be affected through V_{th} variability, keeping $V_{dd,NTC}$, $Cluster\_size_{NTC}$ and Num_PEs constant.

5.7.1 Exploring voltage island granularity

During the formation of VIs, we must explore the effects of applying different Vdd supply voltages during our NTC analysis through scaling. The purpose of this analysis is to explore the effect of different Vdd supply voltages to power/energy consumption and performance of each layer of the examined DNN model. The results obtained will guide as to decide if it is better to apply the same Vdd to all layers or a specific cross-layer policy (e.g., Vdd = 0.6V, cluster_size_ntc = 32). The numerical values used are shown in Table 17.

Table 17: List of parameters for the voltage island granularity analysis

Parameter	Value
num_PEs	256
PEs per VI	32
ntc_cluster_size (VIs)	32 (1), 64 (2), 128 (4), 256 (8)
Dataflow	kcp_ws
Vdd_ntc (in Volts)	0.45, 0.50, 0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.85

5.7.1.1 Intra-layer

Figure 58 displays the distribution of power consumption across each layer of the AlexNet DNN Model in STC $\left(V_{dd,STC}=0.9V \right)$ through a heatmap. In Figure 57 we see the same distribution in NTC for different $V_{dd,NTC}$ and $Cluster\_size_{NTC}$ values. We notice that, for the highest $Cluster\_size_{NTC}$ (= 256 PEs) and the lowest $V_{dd,NTC}$ (= 0.45V), we result in higher gains in power. For example, in the 1st layer of AlexNet, where we have a total $Power_{layer\ 1,\ STC}=4104.42mW$ and

a total $Power_{layer\ 1,\ NTC} = 85.84mW$, the gain equals to:

$$Gain(\%) = 100 \times \frac{(4104.42 - 85.84)}{4104.42} = 97.9\%$$

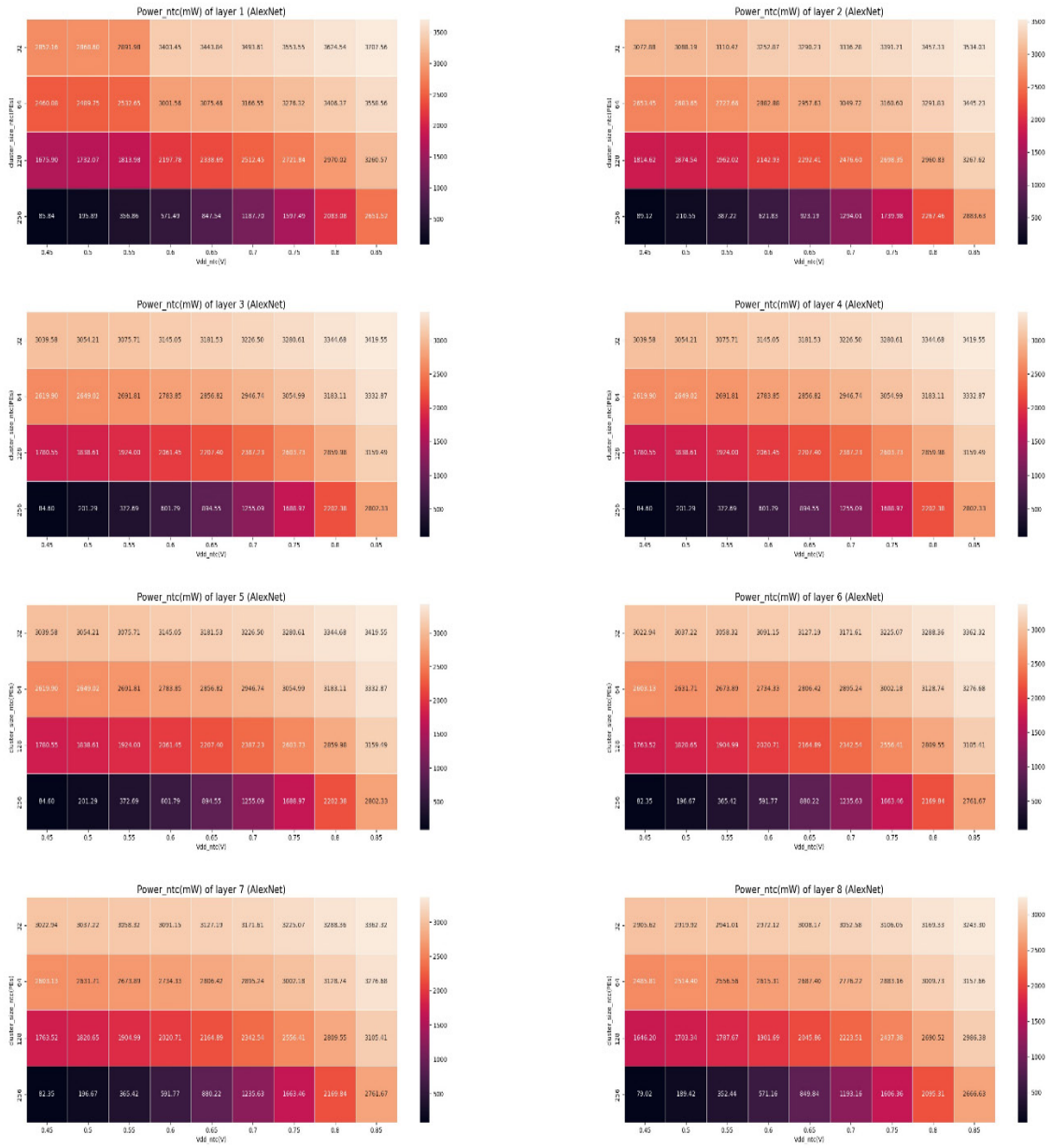


Figure 57: Power consumption per layer in NTC for different $V_{dd,\ NTC}$ and cluster_size$_{NTC}$ (AlexNet)

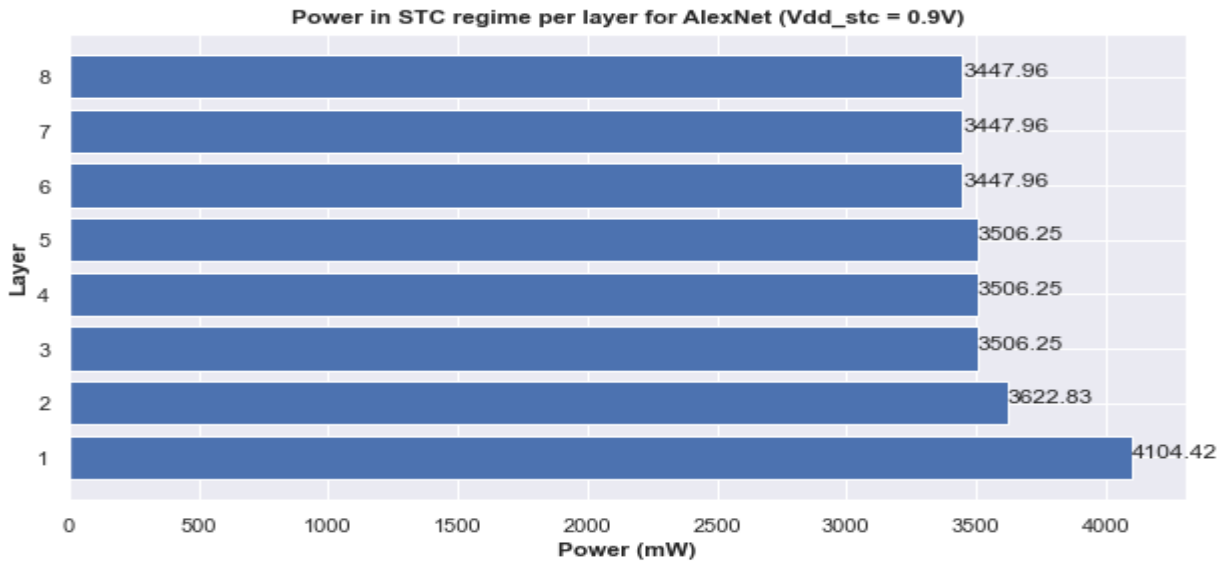


Figure 58: Power consumption per layer in STC for AlexNet

Similar to power, the gain in energy, as shown in Figures 59 and 60, for $Cluster\_size_{NTC}=256$ and $V_{dd,NTC}=0.6V$, equals to:

$$Gain(\%)=100\times\frac{\left(Energy_{layer\ 1,\ STC}-Energy_{layer\ 1,\ NTC}\right)}{Energy_{layer\ 1,\ STC}}=100\times\frac{(1073207-490630.10)}{1073207}=54.28\%$$

Each quantity of energy is measured in nJ. We can also see that in NTC, when moving to Vdd supply voltages below 0.5V, leakage power starts to dominate leading to an increase in energy consumption.

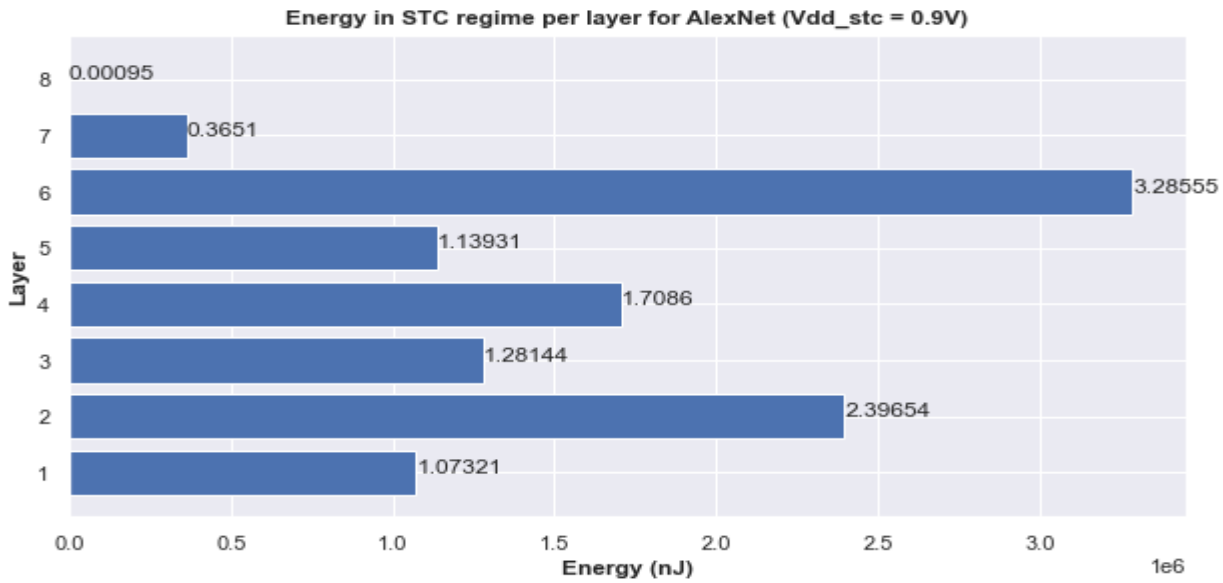


Figure 59: Energy consumption per layer in STC for AlexNet

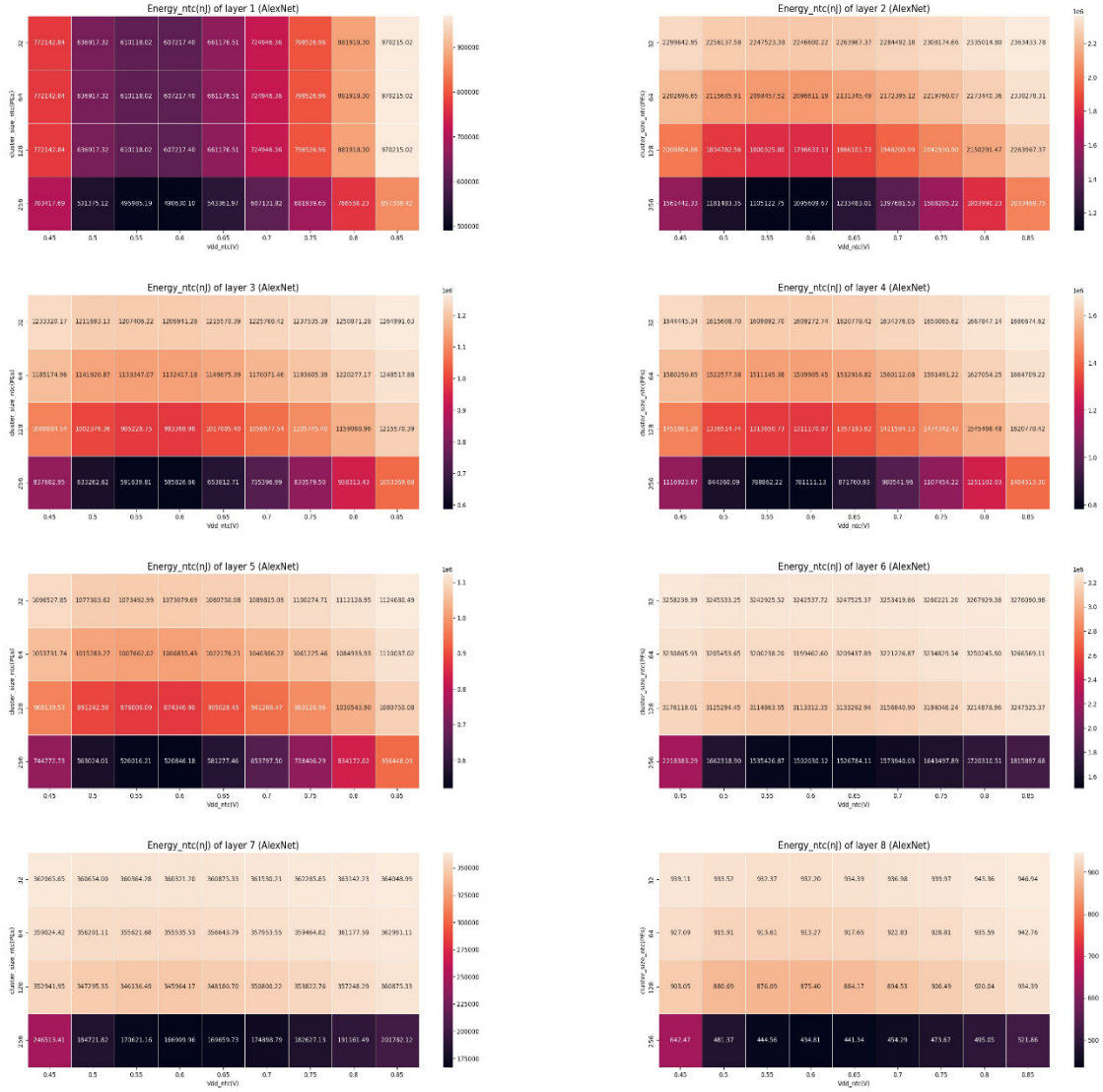


Figure 60: Energy consumption per layer in NTC for for different $V_{dd,\,NTC}$ and cluster_size$_{NTC}$ (AlexNet)

On the contrary, for low $V_{dd,NTC}$, the performance efficiency of each layer is degraded. For example, as shown in Figures 61 and 62, the loss in execution time (msec) in NTC, for the highest $Cluster\_size_{NTC} = 256$ and the lowest $V_{dd,NTC} = 0.45V$, is:

$$Loss(\%) = 100 \times \frac{\left(Performance_{layer\ 1,\ NTC} - Performance_{layer\ 1,\ STC}\right)}{Performance_{layer\ 1,\ NTC}} = 100 \times \frac{66.68 - 4.22}{66.68} = 93.67\%$$

We must also consider that the execution time of each layer is dominated by the PEs working in NTC, as the remaining PEs that work in STC complete their MAC calculations first. Table 18 displays the best gains in power and energy consumption for each layer, as well as the worst loss in execution time during inference of AlexNet CNN Model.

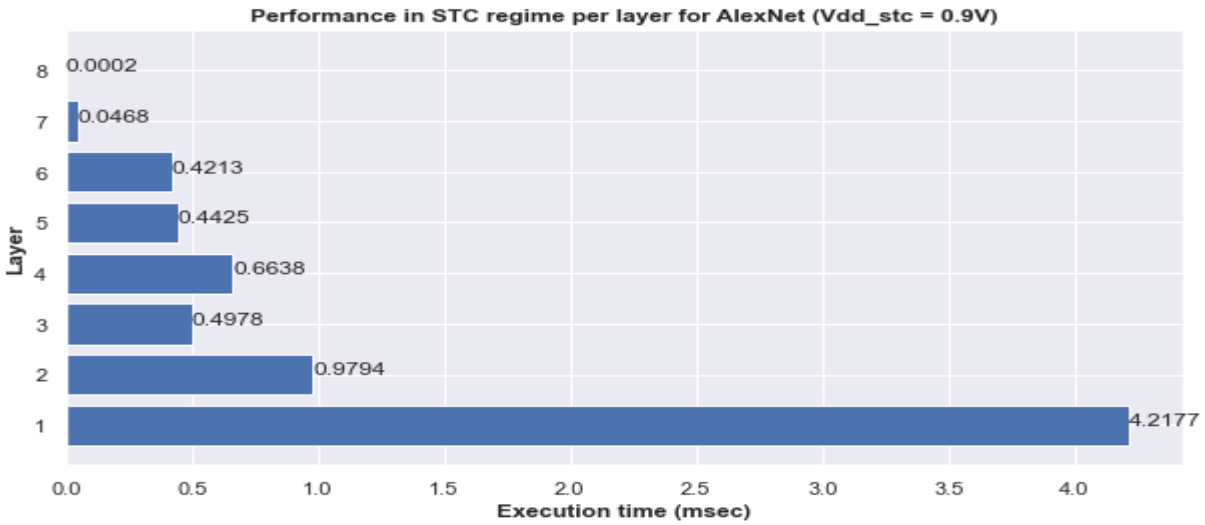


Figure 61: Performance per layer in STC for AlexNet

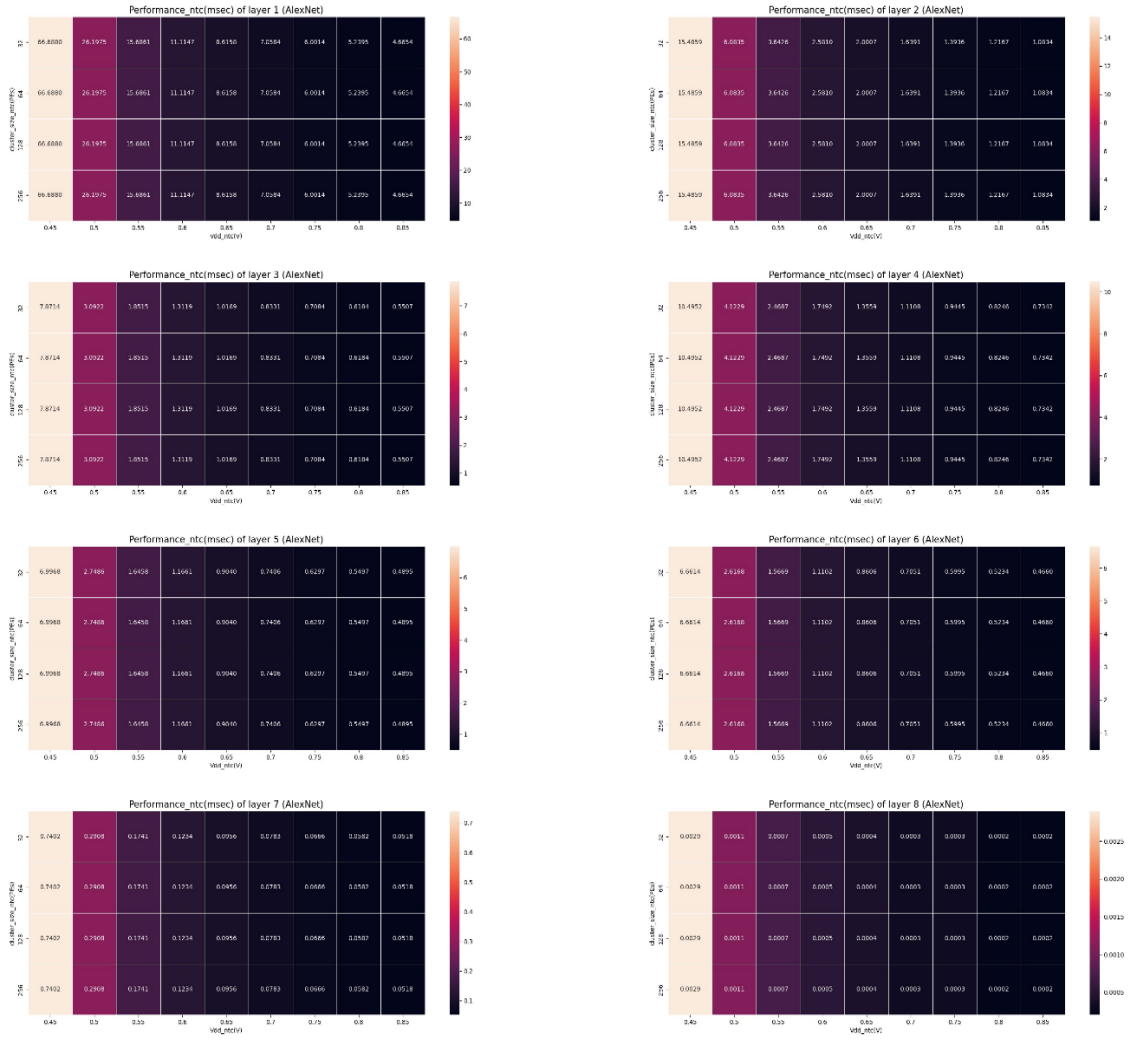


Figure 62: Performance per layer in NTC for different $V_{dd,NTC}$ and cluster_size$_{NTC}$ (AlexNet)

Table 18: Best Power/Energy efficiency gains and worst Performance loss of layers for AlexNet

DNN Model	Layer	Best Gain in Power (Vdd_ntc = 0.45, Cluster_size_ntc = 256)	Best Gain in Energy (Vdd_ntc = 0.6, Cluster_size_ntc = 256)	Worst Loss in Performance (Vdd_ntc = 0.45, Cluster_size_ntc = 256)
AlexNet	1	97.50%	54.28%	93.70%
	2	97.60%	54.32%	93.65%
	3	97.40%	54.30%	93.72%
	4	97.60%	54.45%	93.64%
	5	97.50%	54.48%	93.68%
	6	97.70%	54.33%	93.71%
	7	97.40%	54.45%	93.67%
	8	97.30%	54.38%	93.73%
	Average	97.60%	54.36%	93.65%

Studying the behavior of GoogleNet, ResNet18 and SqueezNet DNN Models, we produce the corresponding heatmaps (see Appendix A.) for power/energy consumption and performance. Similar to the analysis we made for AlexNet, the best average corresponding gains and worst losses for each DNN Model, are displayed in Table 19.

Table 19: Power/Energy efficiency gains and Performance loss of 1st layer per DNN Model

DNN Model	Layers	Best average Gain in Power (Vdd_ntc = 0.45, Cluster_size_ntc = 256)	Best average Gain in Energy (Vdd_ntc = 0.6, Cluster_size_ntc = 256)	Worst average Loss in Performance (Vdd_ntc = 0.45, Cluster_size_ntc = 256)
GoogLeNet	1 to 64	97.62%	54.32%	93.74%
ResNet-18	1 to 21	97.93%	54.25%	93.56%
SqueezeNet	1 to 26	84.51%	54.52%	93.62%

5.7.1.2 Cross-layer policies

As Figures 57 and 60 depict, our NTV-DNN framework during voltage scaling of $V_{dd,NTC}$ achieves, for each layer during inference of Alexnet, high efficiency in energy and power consumption when $Cluster\_size_{NTC} = 256$. We conclude to the same result for inference of GoogLeNet, ResNet-18 and SqueezeNet, through the heatmaps extracted in Appendix A. On the other hand, the execution time for each layer is not affected from each value of $Cluster\_size_{NTC}$ but only from the $V_{dd,NTC}$ scaling. This is because, as we aforementioned above, the execution time of each layer is dominated by the PEs working in NTC, as the remaining PEs that work in STC complete their MAC calculations prior than those working in NTC.

Figures 63 to 65 display the experimental results during voltage scaling for AlexNet, guarding $Cluster\_size_{NTC} = 256$. As we can clearly observe in Figure 64, the lowest point in energy graph is taken for $V_{dd,NTC} = 0.60V$. Below this value of $V_{dd,NTC}$, energy efficiency seems to decline as leakage power starts to rise.

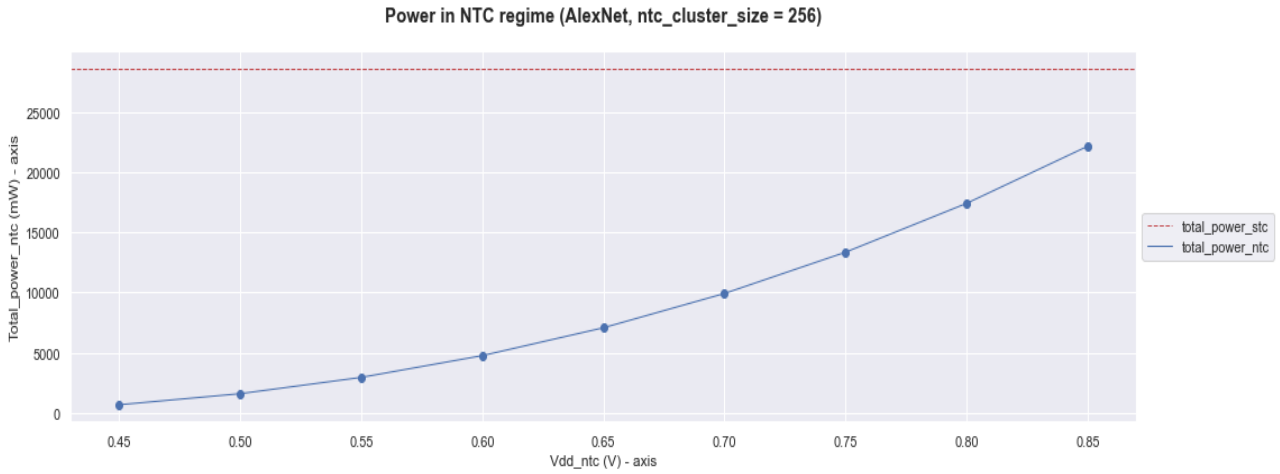


Figure 63: Power consumption of AlexNet for cluster_size$_{NTC}$ = 256

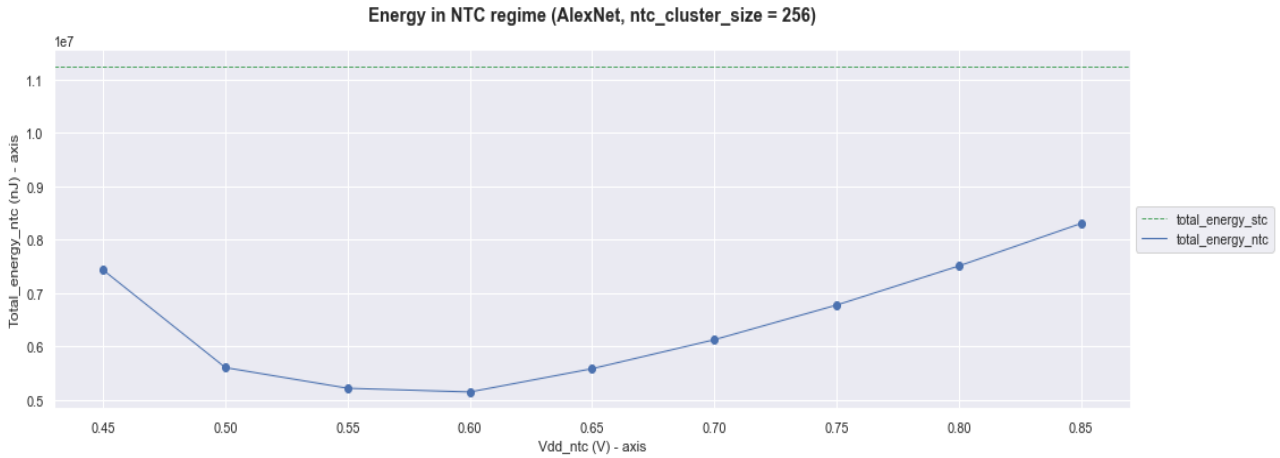


Figure 64: Energy consumption of AlexNet for cluster_size$_{NTC}$ = 256

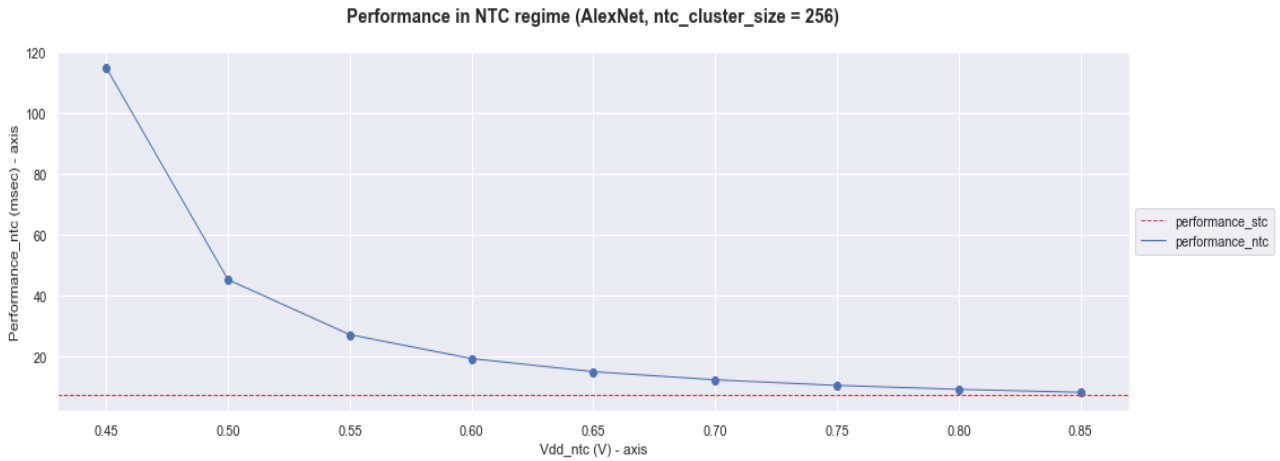


Figure 65: Execution time of AlexNet for cluster_size$_{NTC}$ = 256

Table 20: Best results for Power/Energy consumption and the related performance in NTC for all layers of different DNN Models

Cluster_size$_{NTC}$ = 256					
AlexNet			**GoogLeNet**		
NTC			**NTC**		
Vdd$_{NTC}$ (V)	0.60	0.65	**Vdd$_{NTC}$ (V)**	0.60	0.65
Power (mW)	4753.39	7064.69	**Power (mW)**	76122.82	113212.58
Energy (nJ)	5143398.64	5580581.26	**Energy (nJ)**	9653447.52	10769199.5
Execution time (msec)	19.16	14.85	**Execution time (msec)**	21.65	16.78
STC			**STC**		
Vdd$_{STC}$ (V)	0.90		**Vdd$_{STC}$ (V)**	0.90	
Power (mW)	28589.89		**Power (mW)**	446136.5	
Energy (nJ)	11250693.44		**Energy (nJ)**	21115987.58	
Execution time (msec)	7.27		**Execution time (msec)**	8.22	
ResNet-18			**SqueezeNet**		
NTC			**NTC**		
Vdd$_{NTC}$ (V)	0.60	0.65	**Vdd$_{NTC}$ (V)**	0.60	0.65
Power (mW)	25172.93	37425.48	**Power (mW)**	30703.21	45666.42
Energy (nJ)	10052777.95	11269149.04	**Energy (nJ)**	1720866.42	1913962.61
Execution time (msec)	20.34	15.77	**Execution time (msec)**	4.54	3.52
STC			**STC**		
Vdd$_{STC}$ (V)	0.90		**Vdd$_{STC}$ (V)**	0.90	
Power (mW)	147876.35		**Power (mW)**	181129.53	
Energy (nJ)	21989481.29		**Energy (nJ)**	3764231.68	
Execution time (msec)	7.72		**Execution time (msec)**	1.72	

Moreover, from Figures 57 to 60 and Appendix A., we conclude that, for the total of DNN Models, all layers seem to have the lowest energy consumption for $V_{dd,NTC}$ between 0.60 to 0.65V. As displayed in Table 20, for $V_{dd,NTC}=0.65V$, all DNN Models present better execution time than for $V_{dd,NTC}=0.60V$. The differences for NTC vs. STC in %, are displayed in Table 21. From these results we conclude that, the better policy for each layer of our examined DNN Models is to choose a $V_{dd,NTC}$ between 0.60 to 0.65V, keeping $Cluster\_size_{NTC}=256$.

Table 21: Difference (%) between NTC and STC regime for the total of DNN Models

Cluster_size$_{NTC}$ = 256					
AlexNet			**GoogLeNet**		
Difference (%) NTC vs. STC			**Difference (%) NTC vs. STC**		
Vdd$_{NTC}$ (V)	0.60	0.65	Vdd$_{NTC}$ (V)	0.60	0.65
Power	-83.37%	-75.29%	Power	-82.92%	-74.59%
Energy	-54.28%	-50.40%	Energy	-54.28%	-49.39%
Execution time	163.55%	104.26%	Execution time	163.48%	104.26%
ResNet-18			**SqueezeNet**		

Difference (%) NTC vs. STC			Difference (%) NTC vs. STC		
Vdd_{NTC} (V)	0.60	0.65	Vdd_{NTC} (V)	0.60	0.65
Power	-82.98%	-74.69%	Power	-83.05%	-74.79%
Energy	-54.28%	-49.07%	Energy	-54.28%	-49.52%
Execution time	163.54%	104.27%	Execution time	163.19%	104.06%

5.7.2 Iso-resource NTC vs. STC DNNs

In Figures 66 to 68, we can see an overview of the experimental results that were obtained from the study of different DNN Models under scaling of the supply voltage $V_{dd,NTC}$ and the $Cluster\_size_{NTC}$, keeping the resources (Num_PEs) constant. We found that, when moving to a $Cluster\_size_{NTC} = 256\ PEs$, meaning that all PEs work in NTC, we have significant gains in total power and energy consumption that reach 97% and 58% accordingly for all DNN Models. In terms of performance, as shown in Figure 67, the $Cluster\_size_{NTC}$ does not affect the execution time as the PEs working in NTC dominate during MAC calculations, while the PEs working in STC terminate first and the stay idle.

On the other hand, lowering the $V_{dd,NTC}$ causes big gains in power and energy but leads to great loss in execution time of about 15 x, working in $V_{dd,NTC} = 0.45V$ for all DNN Models. To keep some gains in power and energy consumption and lower the execution time during inference, we conclude that a good scheme for our TPU-based accelerator with a total of $Num\_PEs = 256\ PEs$, is to guard the $V_{dd,NTC}$ to 0.65V with all PEs working in NTC. This leads to 75% and 50% gains in power and energy consumption and at a loss of about 104% in performance.

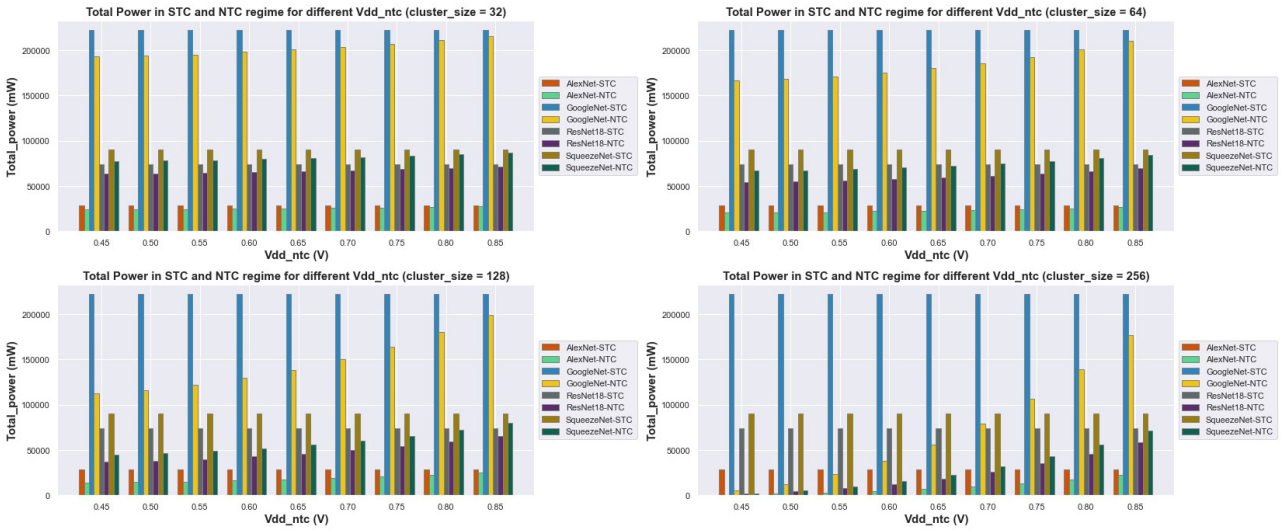


Figure 66: Total power NTC vs. STC for different DNN Models and cluster_size$_{NTC}$

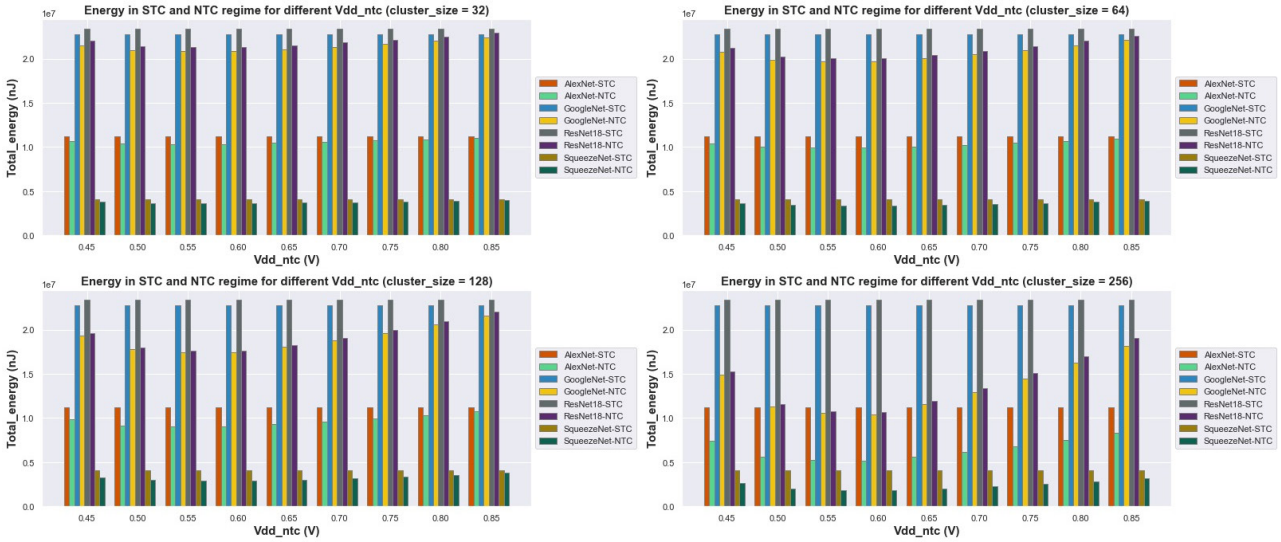


Figure 67: Total energy NTC vs. STC for different DNN Models and cluster_size$_{NTC}$

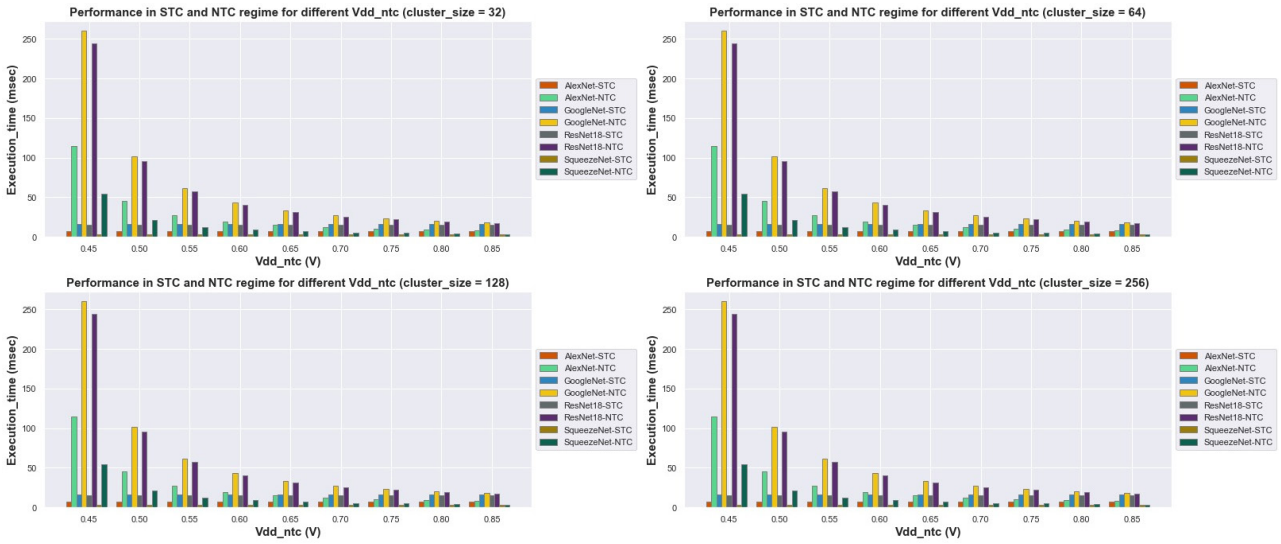


Figure 68: Total performance NTC vs. STC for different DNN Models and cluster_size$_{NTC}$

5.7.3 Iso-performance NTC vs. STC DNNs

The purpose of this research is to find if it exists an equivalent TPU-based accelerator in NTC who has the same performance with an accelerator that works in STC. Figure 71 shows that indeed, for a TPU-based accelerator with $Num\_PEs = 256\ PEs$, $V_{dd,STC} = 0.9V$ *and* $F_{STC} = 700MHz$ in STC, there is an equivalent accelerator with $Num\_PEs = 512\ PEs$, $V_{dd,NTC} = 0.65V$ *and* $F_{NTC} = 343MHz$ in NTC, with at most 2% performance loss. Speaking for power end energy consumption in NTC and STC, Figures 69 and 70 depict that we have a gain of about 50% and 52% accordingly, for all the DNN Models that were examined.

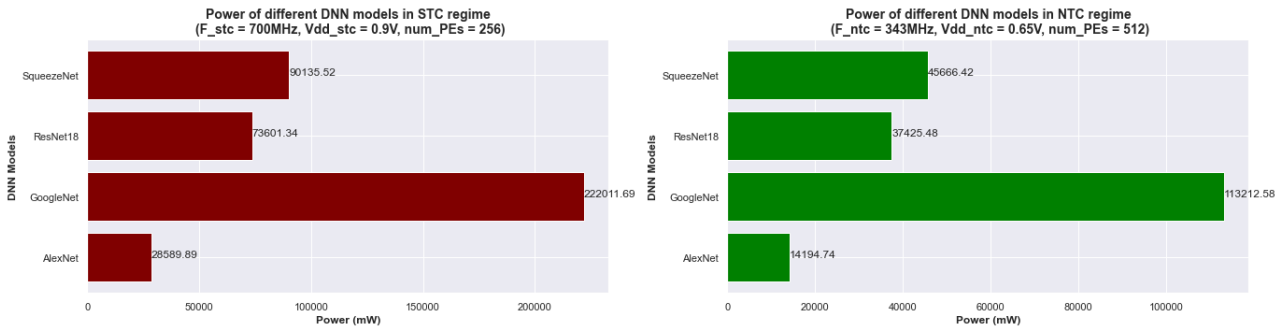


Figure 69: Power consumption for different DNN Models in NTC vs. STC

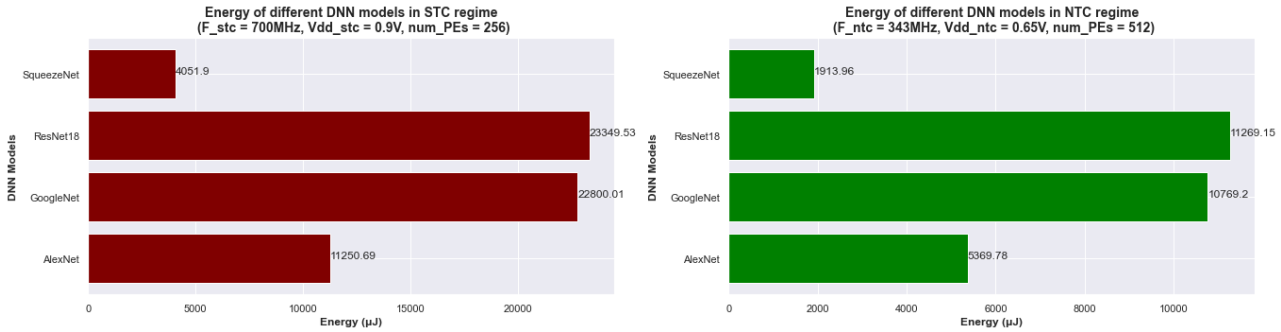


Figure 70: Energy consumption for different DNN Models in NTC vs. STC

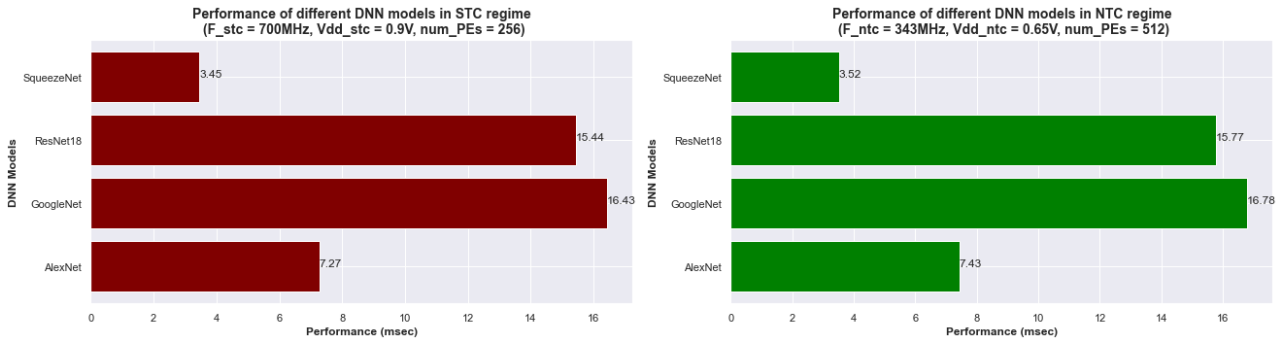


Figure 71: Performance for different DNN Models in NTC vs. STC

5.7.4 NTV-DNN under relaxed error

In this section we compare the efficacy of our NTV-DNN framework for different operating schemes under relaxed error. Each scheme has a different $Cluster\_size_{NTC}$ and is tested for various f_{NTC} operating frequencies. The efficacy is measured through the calculation of accuracy for different DNN Models. Figure 72 depicts the measured accuracy of each examined DNN Model in STC regime. As we can see, ResNet-18 and GoogLeNet show a better inference accuracy using CIFAR-10 dataset.

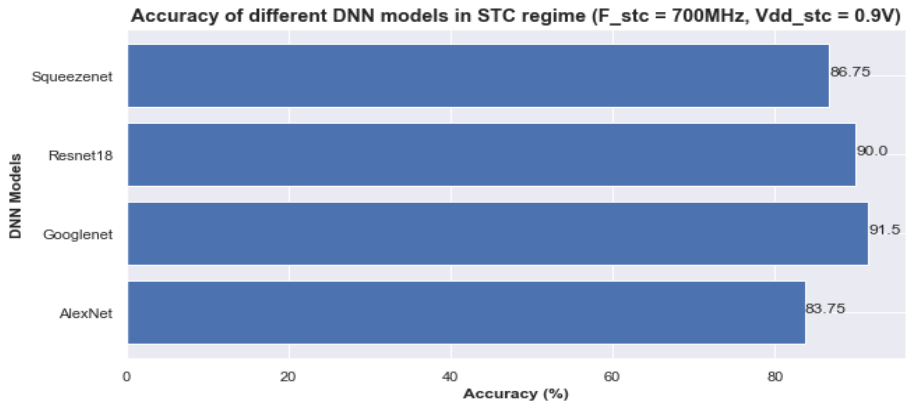


Figure 72: The measured accuracy of different DNN Models in STC

Figures 73 through 76, display the measured accuracy of different DNN Models in NTC. We can observe that, when moving from $Cluster\_size_{NTC}=32$ to $Cluster\_size_{NTC}=256$, the efficacy in accuracy of our NTV-DNN framework, for most operating frequencies, is degraded as the resilience of all DNN Models in errors begins to decline. On the other hand, for every scheme, with the increasing operating frequency, which leads in parallel to an increase in power consumption, all DNN Models seem to be more resilient in errors as we get closer to $f_{STC}=700MHz$.

Table 22 depicts the results in accuracy of different DNN Models for various $Cluster\_size_{NTC}$ and two states of FI. As we can see, GoogLeNet and ResNet-18 seem to be very sensitive during FI, specially when the number of working PEs in NTC exceeds 128. For GoogLeNet, this seems to be quite normal as it contains 57 conv2d layers, each of which is perturbed during the FI procedure. On the contrary, AlexNet, which has only 5 conv2d layers, seems to show greater tolerance even when all PEs work in NTC regime. SqueezeNet, which is the smallest DNN Model in size (=54.55 MB), has 50 times less parameters than AlexNet and counts 26 conv2d layers, displays a remarkable resilience which is close to AlexNet.

Table 22: The measured accuracy of different DNN Models during bit-flip for various cluster_size$_{NTC}$ and F$_{NTC}$

DNN Model	Without error		With error (flipping the 29th bit)		With error (flipping the 29th & 31st bit)	
	F$_{NTC}$ (MHz)	Accuracy	F$_{NTC}$ (MHz)	Accuracy	F$_{NTC}$ (MHz)	Accuracy
Cluster_size$_{NTC}$ = 32						
AlexNet	44	83.75%	118	82.7%	49	77.2%
GoogLeNet		91.5%		86.5%		10%
ResNet-18		90%		67.25%		17%
SqueezeNet		86.75%		85.75%		40.5%

Cluster_size$_{NTC}$ = 64						
AlexNet	44	83.75%	118	82.8%	49	61.8%
GoogLeNet		91.5%		74%		9.5%
ResNet-18		90%		27.5%		9%
SqueezeNet		86.75%		83.5%		10.25%
Cluster_size$_{NTC}$ = 128						
AlexNet	44	83.75%	118	81.7%	49	20%
GoogLeNet		91.5%		38.25%		9.5%
ResNet-18		90%		7.5%		7.5%
SqueezeNet		86.75%		71%		9.5%
Cluster_size$_{NTC}$ = 256						
AlexNet	44	83.75%	118	81.25%	49	9.5%
GoogLeNet		91.5%		9.75%		9.5%
ResNet-18		90%		10.5%		10.5%
SqueezeNet		86.75%		45.75%		7.5%

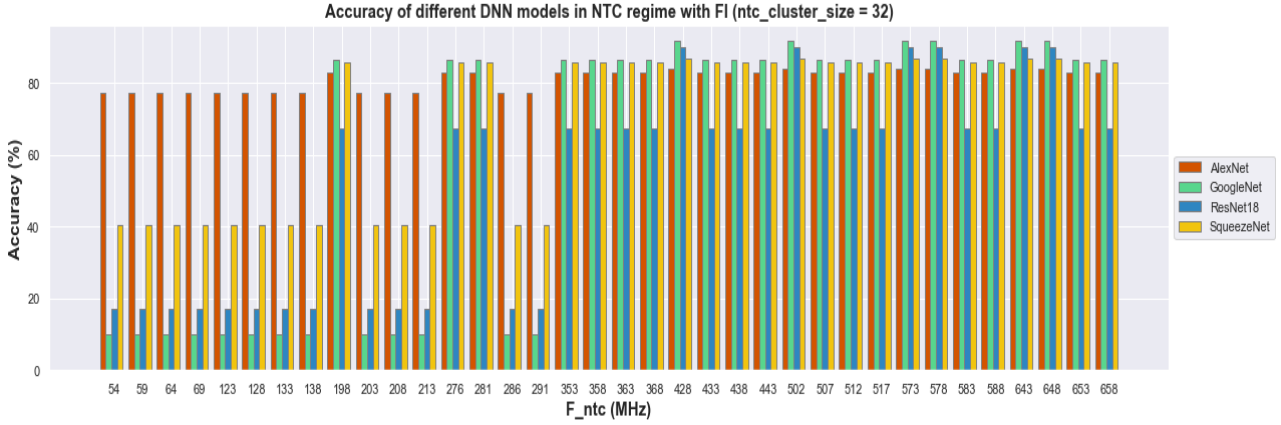


Figure 73: The measured accuracy of different DNN Models for cluster_size$_{NTC}$ = 32

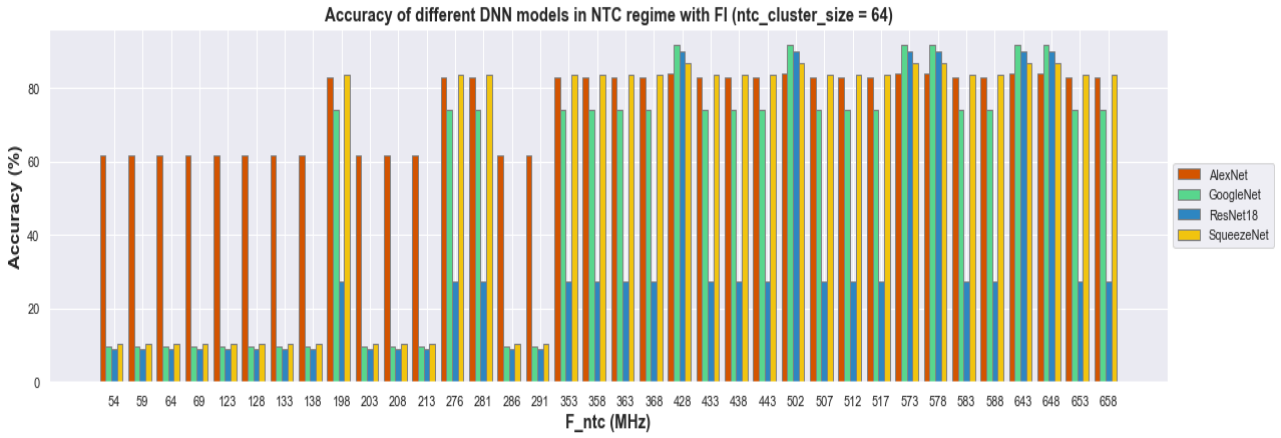


Figure 74: The measured accuracy of different DNN Models for cluster_size$_{NTC}$ = 64

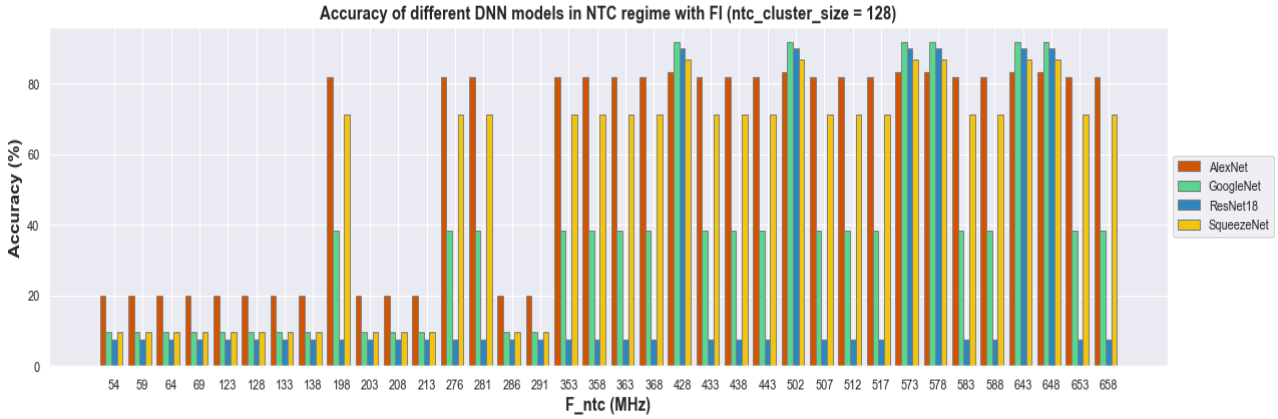


Figure 75: The measured accuracy of different DNN Models for cluster_size$_{NTC}$ = 128

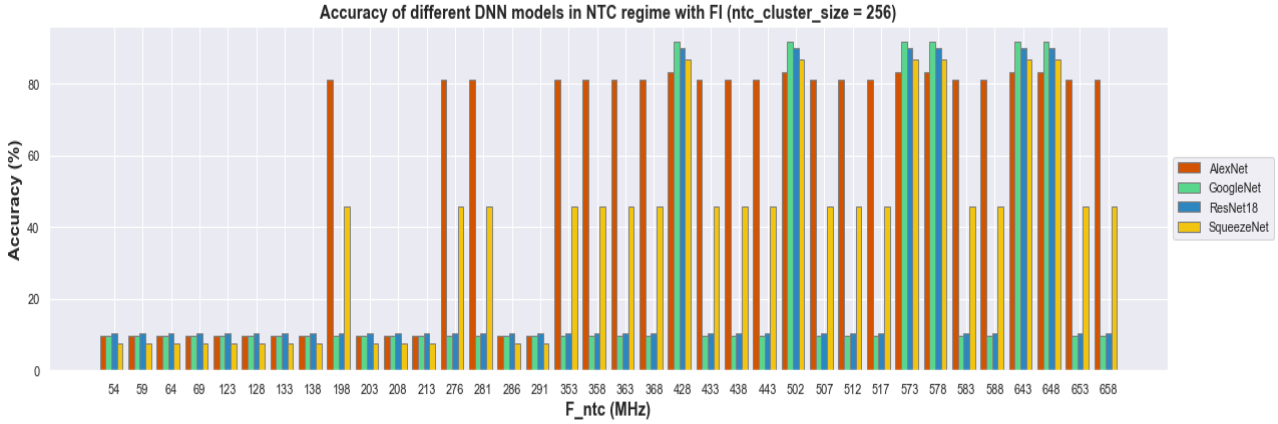


Figure 76: The measured accuracy of different DNN Models for cluster_size$_{NTC}$ = 256

In STC regime, as seen in Figure 72, AlexNet has an inference accuracy of 83.75% with an execution time that equals to 7.27msec. Figure 76 displays the new performance obtained during frequency scaling based on different $V_{dd,NTC}$. Table 23 contains the values of accuracy and performance with and without FI for various f_{NTC} and $Cluster\_size_{NTC}=256$. As we can observe, from a $V_{dd,NTC}$ between 0.45 to 0.50V, the ability to tweak f_{NTC} is very limited, as an increase of at most 5MHz causes a loss of about 89% in accuracy. However, for a $V_{dd,NTC}$ of 0.55V and above, our TPU-based accelerator seems to be more resilient in errors. Especially for $V_{dd,NTC}=0.65V$, which gives, as mentioned above, 75% and 50% gains in power and energy consumption respectively, the accuracy of AlexNet does not exceed the loss of about 3%. This gives us the opportunity, as seen in Table 23, to tweak further f_{NTC}. For example, we can increase f_{NTC} from 343MHz (=14.85msec) to 368MHz (=13.83msec) with a gain of about 7% in performance.

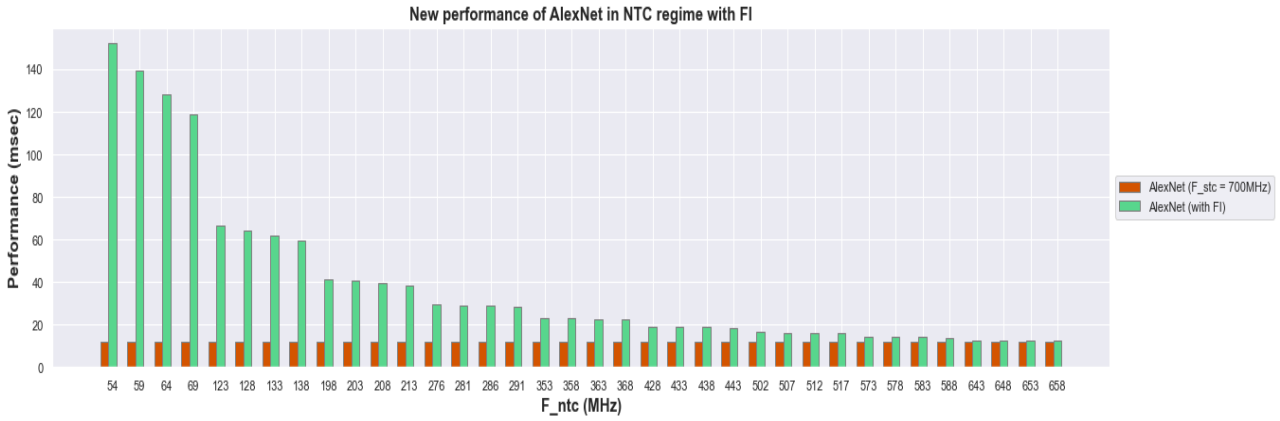


Figure 77: The new performance of AlexNet in NTC for different F_{NTC}

Table 23: The experimental results for AlexNet in NTC with/without Fault Injection

AlexNet (Cluster_size$_{NTC}$ = 256)					
Vdd$_{NTC}$ (Volts)	**F_{NTC} with no errors (MHz)**	**Scaling F_{NTC} (MHz)**	**True: With FI False: Without FI**	**Accuracy (%)**	**New Performance (msec)**
0.45	44	49	True	9.5	103.85
		54	True	9.5	94.24
		59	True	9.5	86.26
		64	True	9.5	79.51
		69	True	9.5	73.75
0.50	113	118	True	81.25	43.12
		123	True	9.5	41.37
		128	True	9.5	39.76
		133	True	9.5	38.26
		138	True	9.5	36.87
0.55	188	193	False	83.75	26.37
		198	True	81.25	25.70
		203	True	81.25	25.07
		208	True	9.5	24.46
		213	True	9.5	23.89
0.60	266	271	False	83.75	18.78
		276	True	81.25	18.44
		281	True	81.25	18.11
		286	True	9.5	17.79

		291	True	9.5	17.49
0.65	343	348	False	83.75	14.62
		353	True	81.25	14.42
		358	True	81.25	14.21
		363	True	81.25	14.02
		368	True	81.25	13.83
0.70	418	423	False	83.75	12.03
		428	False	83.75	11.89
		433	True	81.25	11.75
		438	True	81.25	11.62
		443	True	81.25	11.49
0.75	492	497	False	83.75	10.24
		502	False	83.75	10.14
		507	True	81.25	10.04
		512	True	81.25	9.94
		517	True	81.25	9.84
0.80	563	568	False	83.75	8.96
		573	False	83.75	8.88
		578	False	83.75	8.80
		583	True	81.25	8.73
		588	True	81.25	8.65
0.85	633	638	False	83.75	7.98
		643	False	83.75	7.91
		648	False	83.75	7.85
		653	True	81.25	7.79
		658	True	81.25	7.73

5.7.5 NTV-DNN under different dataflows

The results presented in this section are focused on testing our NTV-DNN Model under different dataflow strategies. For our research, we adopted the scheme of Table 24, which seems, from our thorough analysis till now, to be the most efficient for high gains in power, energy consumption and low losses in performance.

Table 24: The scheme of F_{NTC}, Vdd_{NTC} , num_PEs and cluster_size used for our research

$V_{dd, NTC}$ (V)	F_{NTC} (MHz)	Num_PEs	Cluster_size$_{NTC}$ (PEs)
0.65	343	256	256

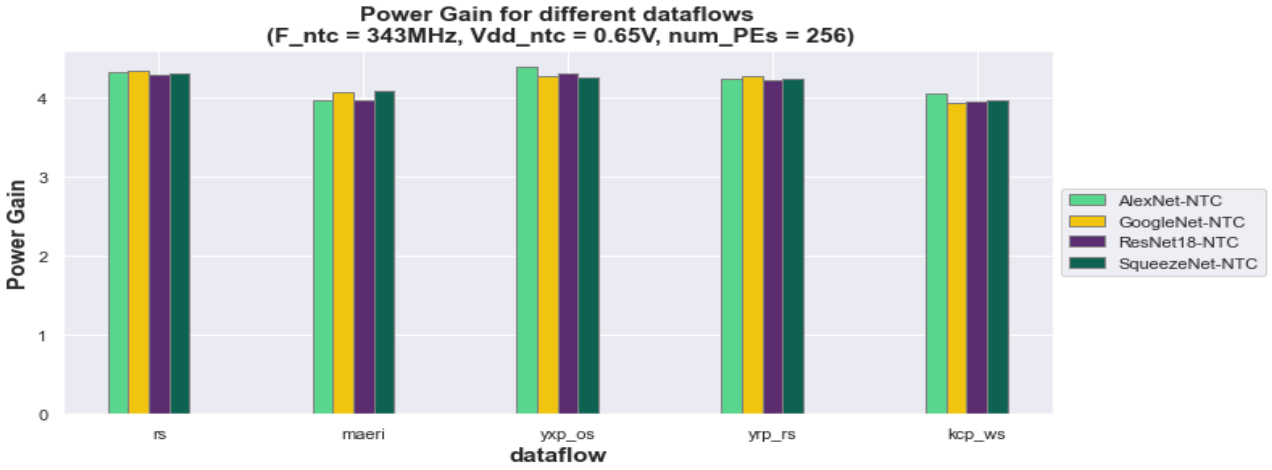


Figure 78: The power gain of different DNN Models for various dataflow strategies

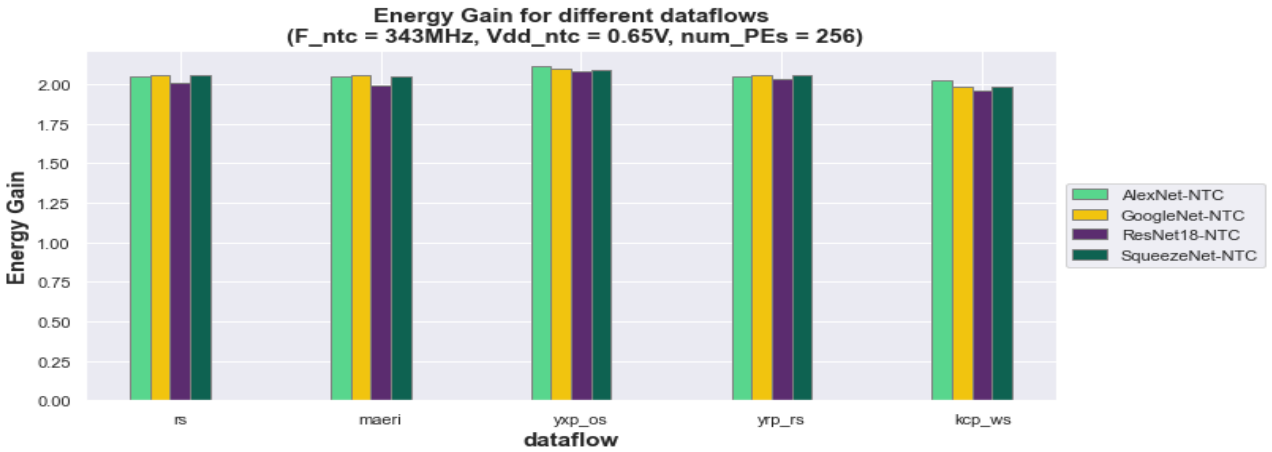


Figure 79: The energy gain of different DNN Models for various dataflow strategies

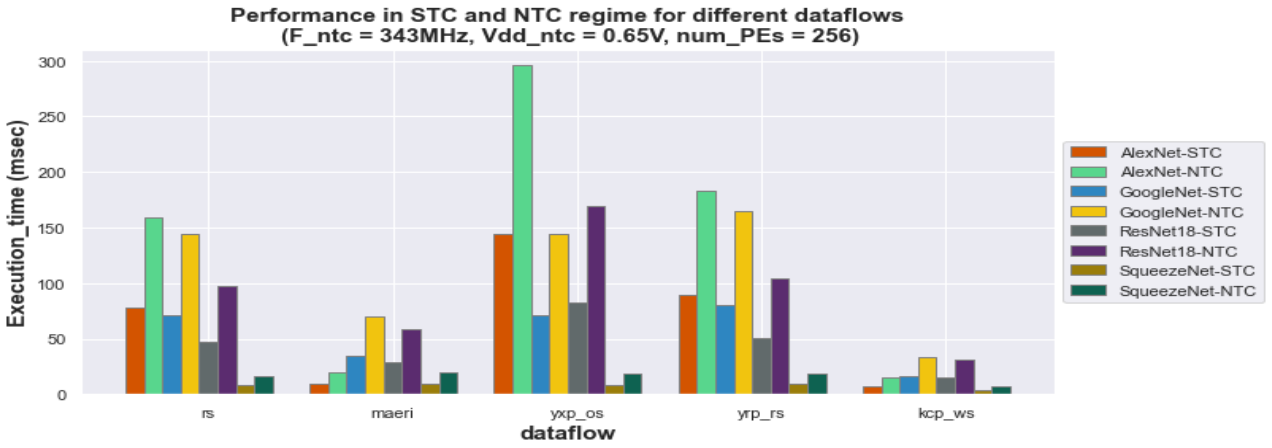


Figure 80: The performance of different DNN Models for various dataflow strategies

Figures 78 to 80 display the power $\left(\frac{Total\_power_{STC}}{Total\_power_{NTC}}\right)$, energy $\left(\frac{Total\_energy_{STC}}{Total\_energy_{NTC}}\right)$ gain and execution time of different CNN Models and various dataflow strategies. Table 25 hosts the experimental results from inference through our NTV-DNN Error Model. In terms of energy efficiency and performance, the *kcp_ws* dataflow outperforms other strategies. On the contrary, the *yxp_os* dataflow seems to be more power efficient over the other dataflow strategies.

Table 25: The experimental results NTV-DNN under different dataflows

	Dataflow				
	rs	**maeri**	**yxp_os**	**yrp_rs**	**kcp_ws**
AlexNet					
Power (mW)	6742.99	6935.45	6489.55	6581.85	7064.69
Energy (nJ)	6836276.61	6603692.53	9058295.06	6862247.5	5580581.26
Execution time (msec)	158.65	19.54	295.62	183.54	14.85
GoogLeNet					
Power (mW)	52141.96	54158.82	51729.91	51774.2	56407.78
Energy (nJ)	18049825.37	17052791.06	21962953.99	18640785.19	11539679.12
Execution time (msec)	144.23	69.53	144.31	164.85	33.56
ResNet-18					
Power (mW)	17612.29	18198.33	16990.78	17323.71	18626.22
Energy (nJ)	14889009.57	13485661.19	20508384.76	16242378.91	11890863.41
Execution time (msec)	97.17	58.66	169.47	104.3	31.54
SqueezeNet					
Power (mW)	21397.14	21915.78	21089.53	21208.32	22723.29
Energy (nJ)	6324925.68	5890602.78	8136307.48	6756676.68	4051902.98
Execution time (msec)	16.63	19.32	18.18	18.75	7.04

5.7.6 The effect of Vth variability

Vth variability is another aspect that we had to explore in order to find out if it could affect the power, energy, and performance efficiency of our NTV-DNN framework. The scheme used in our experiments is shown in Table 26. Each f_{NTC} corresponds to a different V_{th}.

Table 26: The scheme used when exploring the effect of V_{th} variability

$V_{dd,NTC}$ (V)	V_{th}(V)	F_{NTC} (MHz)	Num_PEs	Cluster_size$_{NTC}$ (PEs)
0.65	[0.45, 0.50, 0.55]	[287, 223, 148]	256	256

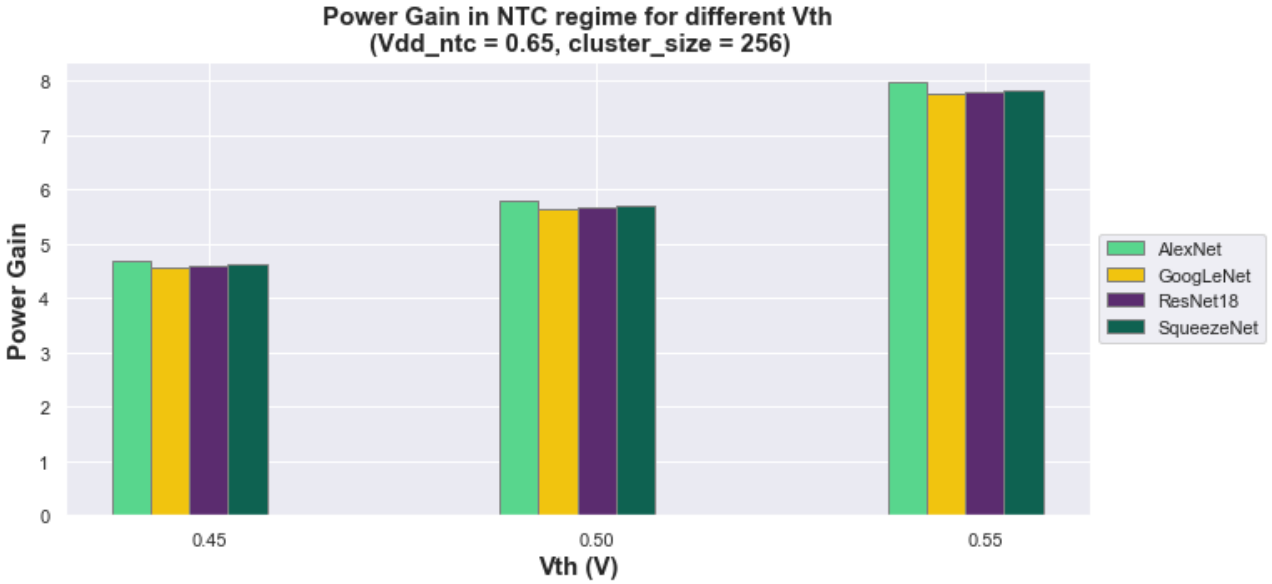


Figure 81: Power gain of different DNN Models for various V_{th}

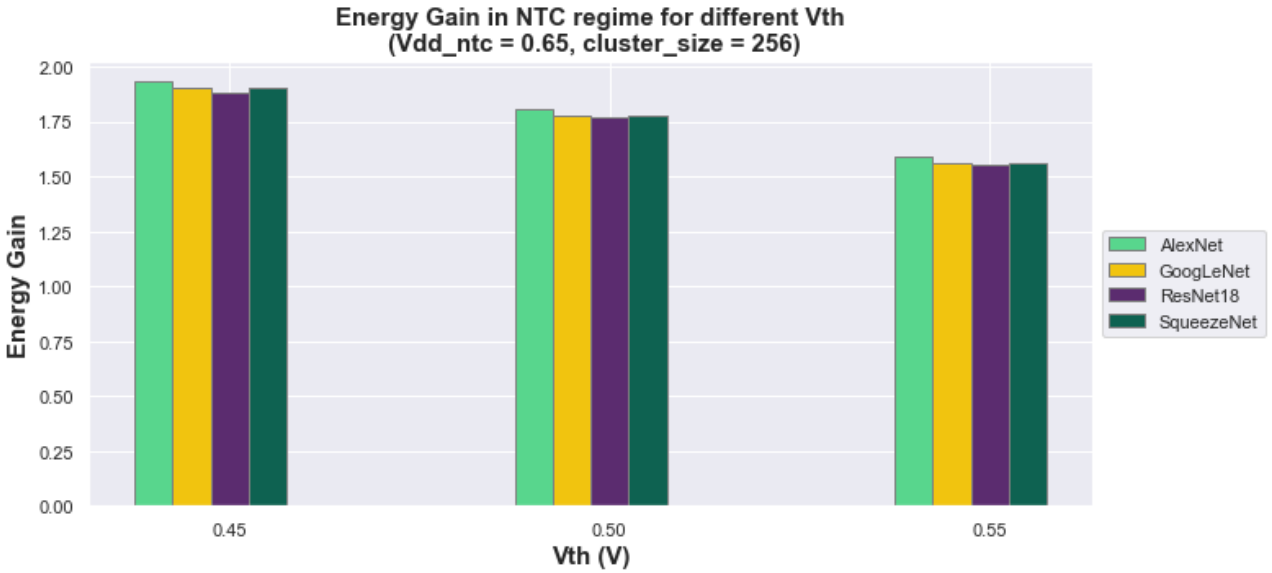


Figure 82: Energy gain of different DNN Models for various V_{th}

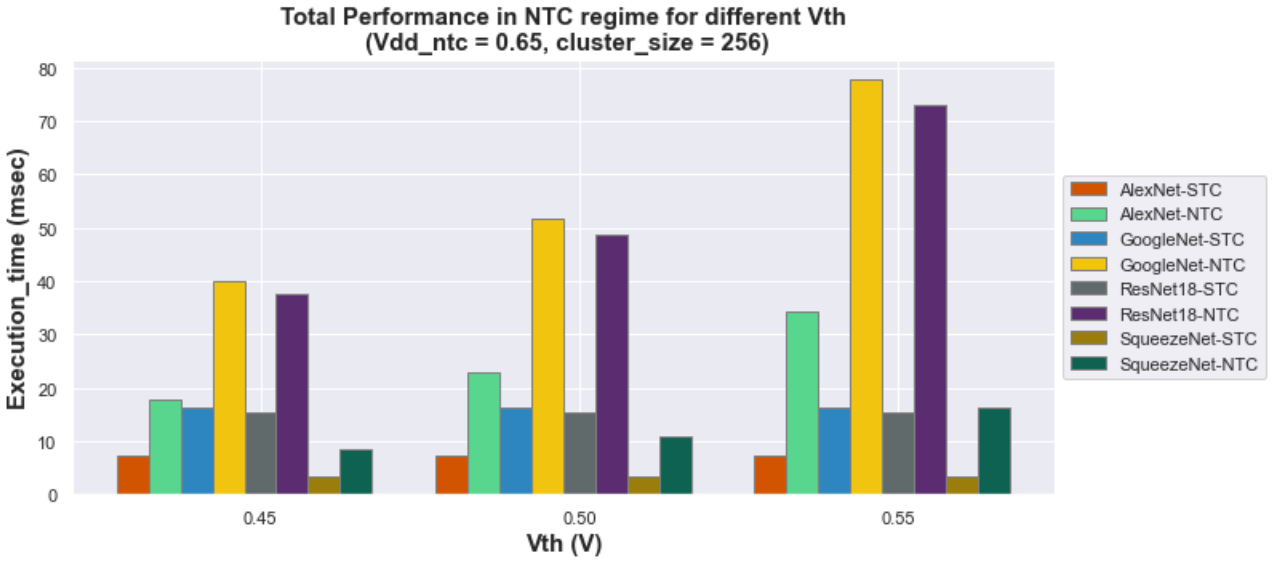


Figure 83: Performance of different DNN Models for various V_{th}

As we see in Figures 81, 82 and 83 , during scaling of V_{th}, power gain $\left(\frac{Total\_power_{STC}}{Total\_power_{NTC}}\right)$ starts to increase for all CNN Models examined. On the contrary, energy gain $\left(\frac{Total\_energy_{STC}}{Total\_energy_{NTC}}\right)$ starts to decrease. In addition, execution time during inference for all CNN Models starts to increase and most likely causing timing errors during MAC calculations which affects accuracy. Table 27 displays the exact results obtained for specific values of V_{th} categorized by DNN Model, keeping $V_{dd,NTC} = 0.65V$ constant.

Table 27: The experimental results exploring the effect of V_{th} variability in NTC

	V_{th} (V)		
	0.45	**0.50**	**0.55**
	F_{NTC} (MHz)		
	287	**223**	**148**
AlexNet			
Power (mW)	6073.42	4919.38	3587.61
Energy (nJ)	5815239.63	6200977.65	7078532.72
Execution time (msec)	17.72	22.86	34.38
GoogLeNet			
Power (mW)	48476.52	39242.71	28586.63
Energy (nJ)	12034423.56	12816138.43	14613740.99

Execution time (msec)	40.04	51.66	77.68
ResNet-18			
Power (mW)	16007.79	12959.42	9441.67
Energy (nJ)	12403620.16	13204175.8	15051191.94
Execution time (msec)	37.63	48.56	73.02
SqueezeNet			
Power (mW)	19526.04	15803.76	11508.21
Energy (nJ)	2132946.84	2271869.3	2590887.87
Execution time (msec)	8.4	10.84	16.3

5.8 Conclusions

In this thesis, we have presented and evaluated NTV-DNN, , a tool for early assessment of energy at various voltage variation levels. NTV-DNN deals with the formation of VIs in NTC through scaling of the $V_{dd,NTC}$ supply voltage, providing reduced power and energy consumption. Further, NTV-DNN performs a frequency scaling, synergistically with PyTorchFI and CIFAR-10, to further boost performance by evaluating the accuracy of different DNN Models through inference. Our experiments demonstrate significant reductions in power and energy consumption of about 80% and 50% respectively, for a range of $V_{dd,NTC}$ supply voltage between 0.6V to 0.65V for the total of our 16 x 16 TPU-based accelerator, but with a cost of about 90% reduction in execution time and 3% reduction in accuracy.

Furthermore, we have proven that for a TPU-based accelerator with a total of 256 PEs that works in STC, there is an equivalent accelerator with a double size of PEs and with similar performance that works in NTC which shows gains of about 50% and 52% in power and energy consumption respectively. We also concluded that, in terms of energy efficiency and performance, the *kcp_ws* dataflow seems to outperform other strategies like *maeri* and *yxp_os*. Finally, we concluded that, when tweaking V_{th} to greater values of 0.4V, energy consumption is increasing.

5.9 Future Work

As future work, we would like to explore the possibility of choosing dynamically the $V_{dd,NTC}$ for each VI of PEs and the $Cluster\_size_{NTC}$, according to the dataflow strategy used during inference. It is worth noting that it could be very interesting to explore the synergy between NTV-DNN and other techniques like dynamic DNN pruning, to further reduce energy consumption. We could also

use more approximate or reduced precision MAC units, which offer more potential in the accomplish of designing more energy efficient DNN accelerators. Studying how efficient could be NTV-DNN during CNN training, is another interesting aspect that we could explore. Furthermore, it would be very interesting to investigate the gains in energy consumption with the combination of other design paradigms like GreenTPU.

REFERENCES

[1] V. Sze, Y. H. Chen, T. J. Yang, J. S. Emer, Efficient Processing of Deep Neural Networks, USA, Morgan & Claypool Publishers, 2020.

[2] Alzubaidi, L., Zhang, J., Humaidi, A.J. et al., Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. J Big Data 8, 53 (2021).

[3] S. Xydis, "Lecture - Diffuse and Embedded Systems", Dept. of Informatics and Telematics, Harokopio Univ., Athens, 2020.

[4] S. Reda, M. Shafique, Approximate Circuits: Methodologies and CAD, Springer International Publishing, Cham, Switzerland, 2019.

[5] M. Hubner, C. Silvano, Near Threshold Computing: Technology, Methods and Applications, Springer International Publishing, Cham, Switzerland, 2016.

[6] Z. Abbas, M. Olivieri, Impact of technology scaling on leakage power in nano-scale bulk CMOS digital standard cells, Microelectronics Journal, Vol. 45, Issue 2, (2014) p. 179-195.

[7] Professor John McCarthy, Stanford University, http://jmc.stanford.edu/ (accessed 29/3/2022).

[8] Datacatchup, https://datacatchup.com/artificial-intelligence-machine-learning-and-deep-learning/ (accessed 29/3/2022).

[9] Y. Ibrahim, H. Wang, Junyang Liu et al., Soft errors in DNN accelerators: A comprehensive review, Microelectronics Reliability 115 (2020).

[10] A. Rassadin, A. Savchenko, Deep neural networks performance optimization in image recognition, in: Proc. 3rd Int. Conf. Info. Technol. Nano-Technol. (ITNT), Nizhny Novgorod, Russia, 2017.

[11] Tech blog, S. Bhattarai, https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/ (accessed 2/4/2022).

[12] Data Science Community, https://datascience.stackexchange.com/questions/44703/how-does-gradient-descent-and-backpropagation-work-together (accessed 2/4/2022).

[13] K. Arulkumaran, M. P. Deisenroth, M. Brundage and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," in IEEE Signal Processing Magazine, vol. 34, no. 6, pp. 26-38, Nov. 2017.

[14] Nvidia blog, M. Copeland, https://blogs.nvidia.com/blog/2016/08/22/difference-deep-learning-training-inference-ai/ (accessed 2/4/2022).

[15] Mittal, S. A survey of FPGA-based accelerators for convolutional neural networks. Neural Comput & Applic 32, 1109–1139 (2020).

[16] C. Pelletier, G. I Web, F. Petitjean, "Temporal Convolutional Neural Network for the Classification of Satellite Image", IEEE, Remote Sensing (2019).

[17] Towards Data Science, https://towardsdatascience.com/understanding-relu-the-most-popular-activation-function-in-5-minutes-459e3a2124f (accessed 7/4/2022).

[18] X. Zhang, J. Trmal, D. Povey, and S. Khudanpur, Improving deep neural network acoustic models using generalized maxout networks, in International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2014.

[19] Programmathicaly, a Blog on Building Machine Learning Solutions, https://programmathically.com/what-is-pooling-in-a-convolutional-neural-network-cnn-pooling-layers-explained/ (accessed 7/4/2022).

[20] L. Huang, J. Qin, Yi Zhou, et al., Normalization Techniques in Training DNNs: Methodology, Analysis and Application, Cornell University, 2020

[21] Baeldung, https://www.baeldung.com/cs/batch-normalization-cnn (accessed 10/4/2022)

[22] V. Dumoulin, F. Visin, A guide to convolution arithmetic for deep learning, Cornel University, Jan 12, 2018.

[23] A. Vaswani, N. Shazeer, , et al., Attention Is All You Need, 31st Conf. on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet classification with deep convolutional neural networks, in Conf. on Neural Information Processing Systems (NeurIPS*)*, 2012.

[25] Nerohive, https://neurohive.io/en/popular-networks/alexnet-imagenet-classification-with-deep-convolutional-neural-networks/ (accessed 10/4/2022).

[26] C. Szegedy, W. Liu,et al., Going deeper with convolutions, Conf. on Computer Vision and Pattern Recognition (CVPR), 2015.

[27] C. K. He, X. Zhang, S. Ren, and J. Sun, Deep residual learning for image recognition, in Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[28] A. Zhang, Z. C. Lipton, M. Li, A. J. Smola, Interactive book, Dive into Deep Learning, https://d2l.ai/chapter_convolutional-modern/resnet.html (accessed 10/4/2022).

[29] F. N. Iandola, S. Han, et al., SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, International Conf. on Learning Representations (ICLR), 2017.

[30] H. Oh, H. Lee, M. Y. Kim, Comparing Convolutional Neural Network (CNN) models for machine learning-based drone and bird classification of anti-drone system, 19th International Conf. on Control, Automation and Systems (ICCAS) (2019): 87-90.

[31] Wikipedia, Central processing unit, https://en.wikipedia.org/wiki/Central_processing_unit (accessed 16/4/2022)

[32] J. M. P. Cardoso, J. G. F. Coutinho, P. C. Diniz, Embedded Computing for High Performance, Elsevier, Morgan Kaufmann Publishers, USA, 2017

[33] Intel, Hyper-Threading Technology, https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html (accessed 16/4/2022)

[34] S. Mittal, P. Rajput, S. Subramoney, A Survey of Deep Learning on CPUs: Opportunities and Co-optimizations, IEEE Transactions on Neural Networks and Learning Systems (2021).

[35] Y. E. Wang et al., "Benchmarking TPU, GPU, and CPU Platforms for Deep Learning", arXiv:1907.10701, (2019).

[36] M. Zhang et al., "DeepCPU: Serving RNN-based deep learning models 10x faster", in USENIX ATC, 2018, pp. 951–965.

[37] N. D. Lane et al., "DeepX: A software accelerator for low power deep learning inference on mobile devices", in IPSN, 2016, p. 23.

[38] P. Blacker et al., "Rapid Prototyping of Deep Learning Models on Radiation Hardened CPUs", in AHS, 2019, pp. 25–32.

[39] Y. Ibrahim et al., "Soft errors in DNN accelerators: A comprehensive review", in Microelectronics Reliability, 2020, Volume 115.

[40] OpenGenus Foundation, https://iq.opengenus.org/structure-of-field-programmable-gate-array-fpga/ (accessed 16/4/2022).

[41] S. Mittal, "A Survey of FPGA-based Accelerators for Convolutional Neural Networks", Neural computing and applications, 2018.

[42] Z. Li, Y. Wang, T. Zhi, T. Chen, A survey of neural network accelerators, Frontiers of Computer Science 11 (5) (2017) 746–761.

[43] N. P. Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit", in Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017, 12 pages.

[44] S. Reda, M. Shafique, "Approximate Circuits : Methodologies and CAD", Springer Nature Switzerland, 2019.

[45] S. Li et al., "Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation", in International Conference on Supercomputing, June 12–15, 2018, Beijing, China.

[46] J. Yu et al., "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism", in Proceedings of ISCA '17, Toronto, Canada, June 24-28, 2017.

[47] C. Zhu et al., "Trained Ternary Quantization", in Proceedings of ICLR, Toulon, France, 2017.

[48] S. Jain et al., "Compensated-DNN: Energy Efficient Low-Precision Deep Neural Networks by Compensating Quantization Errors", in DAC '18, June 24–29, 2018, San Francisco, CA, USA.

[49] P. Pandey et al., "Challenges and Opportunities in Near-Threshold DNN Accelerators around Timing Errors", in Journal of Low Power Electronics and Applications, MDPI, Basel, Switzerland, 2020.

[50] P. Pandey et al., “GreenTPU: Improving Timing Error Resilience of a Near-Threshold Tensor Processing Unit”, in Design Automation Conference (DAC), June 2–6, 2019, Las Vegas, NV, USA.

[51] R. Trapani et al., “GPU NTC Process Variation Compensation with Voltage Stacking”, in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 26, no. 9, pp. 1713-1726, Sept. 2018.

[52] P. Basu et al., “SwiftGPU: Fostering Energy Efficiency in a Near-Threshold GPU Through a Tactical Performance Boost”, in Design Automation Conference (DAC), June 05-09, 2016, Austin, TX, USA.

[53] G. E. Moore, “Cramming more components onto integrated circuits”, in Proceedings of the IEEE, Volume: 86, Issue: 1, Jan. 1998.

[54] C. E. Leiserson et al., “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?”, in Science, 2020.

[55] N. D. Lane et al., “An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices,” in International Workshop on Internet of Things towards Applications. ACM, 2015.

[56] A. Azizimazreah et al., “Tolerating Soft Errors in Deep Learning Accelerators with Reliable On-Chip Memory Designs”, in IEEE International Conference on Networking, Architecture and Storage (NAS), 2018.

[57] H. Kwon et al., “MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings” in IEEE Micro., 2020.

[58] V. Sze et al., “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”, in Proceedings of the IEEE, 2017.

[59] A. Mahmoud et al., “PyTorchFI: A Runtime Perturbation Tool for DNNs”, in 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), 2020.

[60] I. Stamelakos, S. Xydis, G. Palermo et al., “Variation-Aware Voltage Island Formation for Power Efficient Near-Threshold Manycore Architectures”, in Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, 2014, p. 304-310.

[61] K. Chen et al., “CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques”, in IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, 2011.

[62] S. Mittal, “A survey on modeling and improving reliability of DNN algorithms and accelerators”, in Journal of Systems Architecture, 2020.

[63] S. Hong et al., “Terminal brain damage: exposing the graceless degradation in deep neural

networks under hardware fault attacks", in USENIX Security Symposium, 2019.

[64] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library", in 33rd Conference on Neural Information Processing Systems, Vancouver, Canada, 2019.

[65] Pytorch, https://pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html (accessed 27/5/2022).

[66] Cuda toolkit, https://developer.nvidia.com/cuda-toolkit (accessed 27/5/2022).

[67] Medium platform, https://medium.com/@nutanbhogendrasharma/pytorch-convolutional-neural-network-with-mnist-dataset-4e8a4265e118 (accessed 27/5/2022).

[68] MAESTRO cost model, https://maestro.ece.gatech.edu/docs/build/html/hw_supported.html (accessed 27/5/2022).

[69] The CIFAR-10 Dataset, https://www.cs.toronto.edu/~kriz/cifar.html (accessed 28/5/2022).

[70] Yu-Hsin Chen et al., "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks", in IEEE Journal of Solid State Circuits, Vol. 52, 2017.

[71] H. Kwon et al., "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects", in Association for Computing Machinery, 2018.

[72] Tien-Ju Yang et al., "A Method to Estimate the Energy Consumption of Deep Neural Networks", in 51st Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, USA, 2017.

[73] Synopsys hardware design software, https://www.synopsys.com/ (accessed 26/06/2022).

APPENDIX A.

Heatmaps of different DNN Models for exploring the voltage island granularity in NTC and STC.

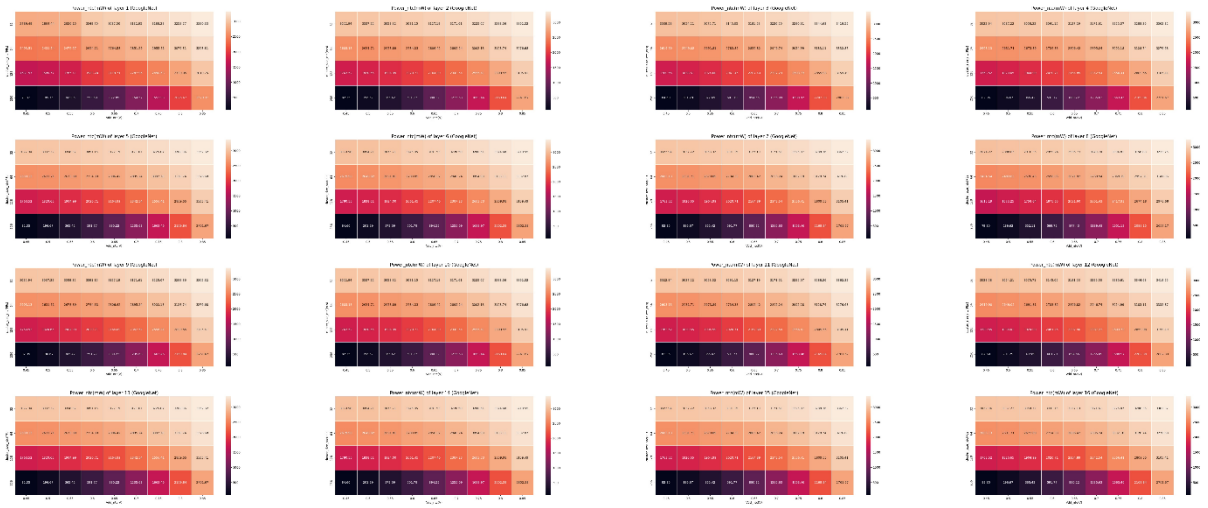


Figure A.1: Power consumption of layers 1 to 16 (GoogLeNet).

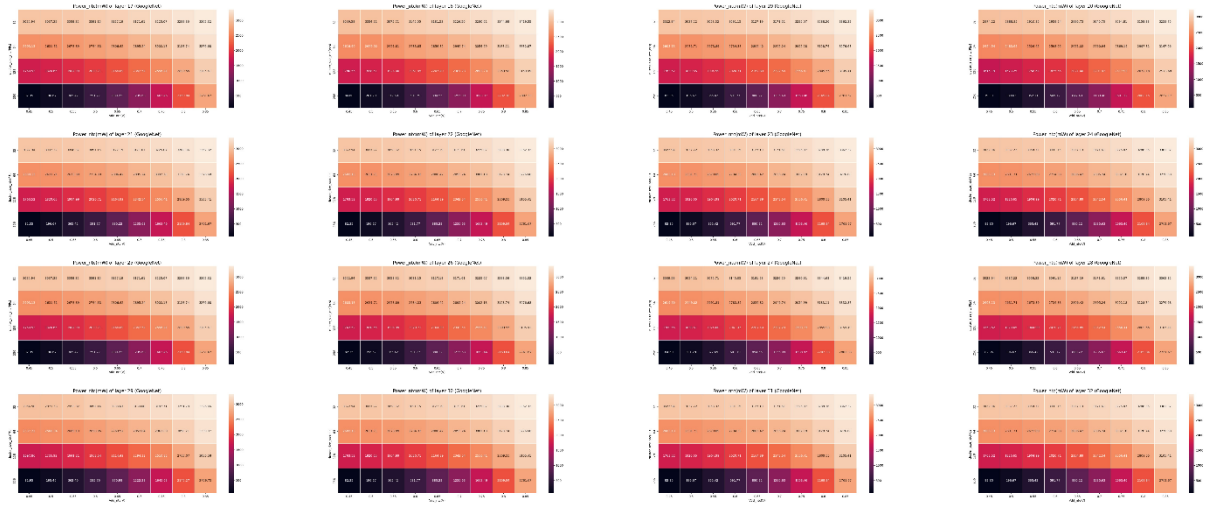


Figure A.2: Power consumption of layers 17 to 32 (GoogLeNet).

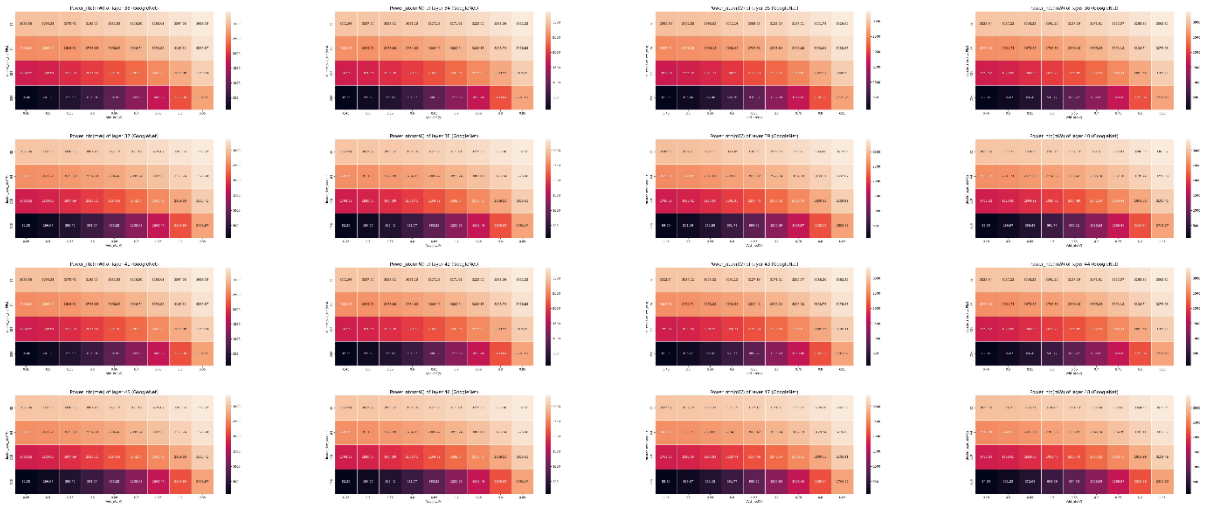


Figure A.3: Power consumption of layers 33 to 48 (GoogLeNet).

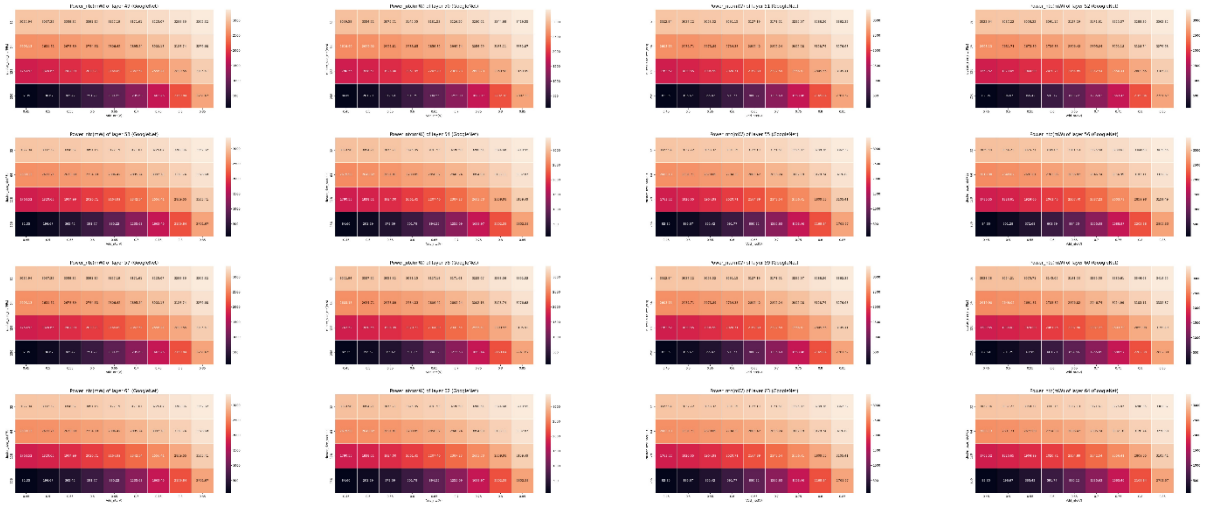


Figure A.4: Power consumption of layers 49 to 64 (GoogLeNet)

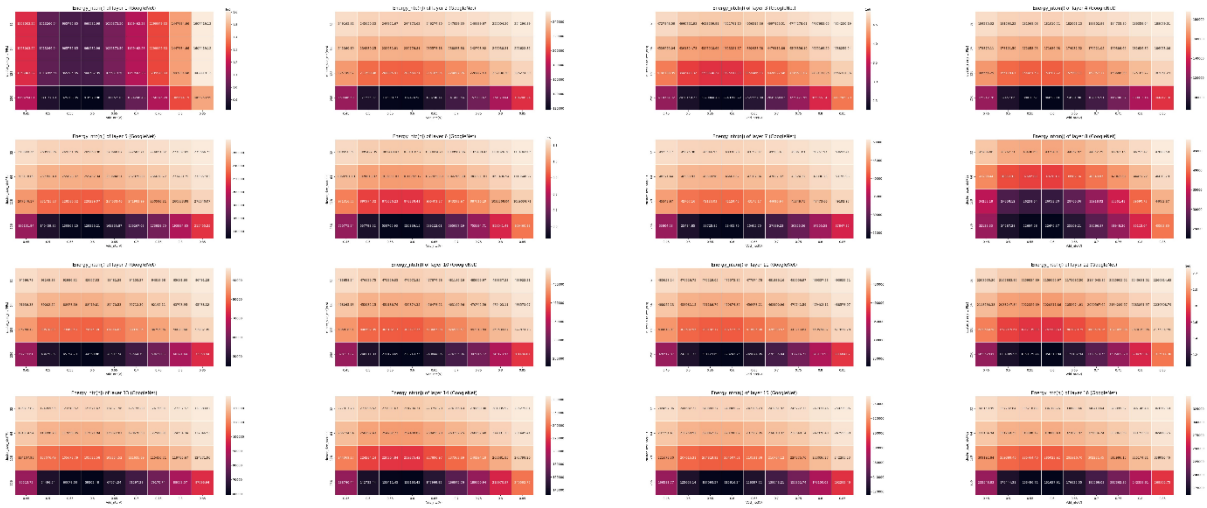


Figure A.5: Energy consumption of layers 1 to 16 (GoogLeNet)

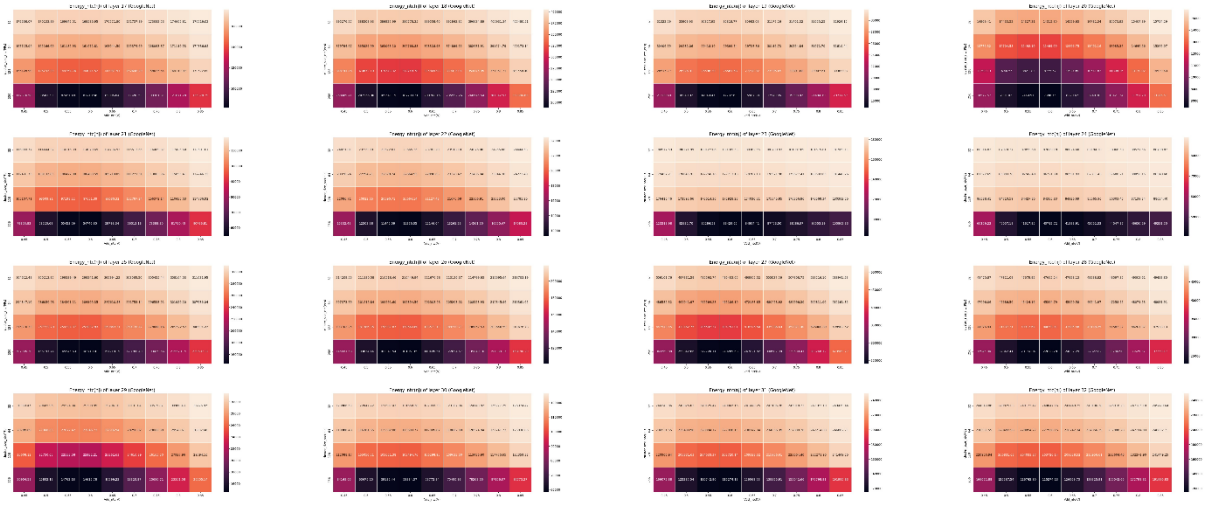


Figure A.6: Energy consumption of layers 17 to 32 (GoogLeNet)

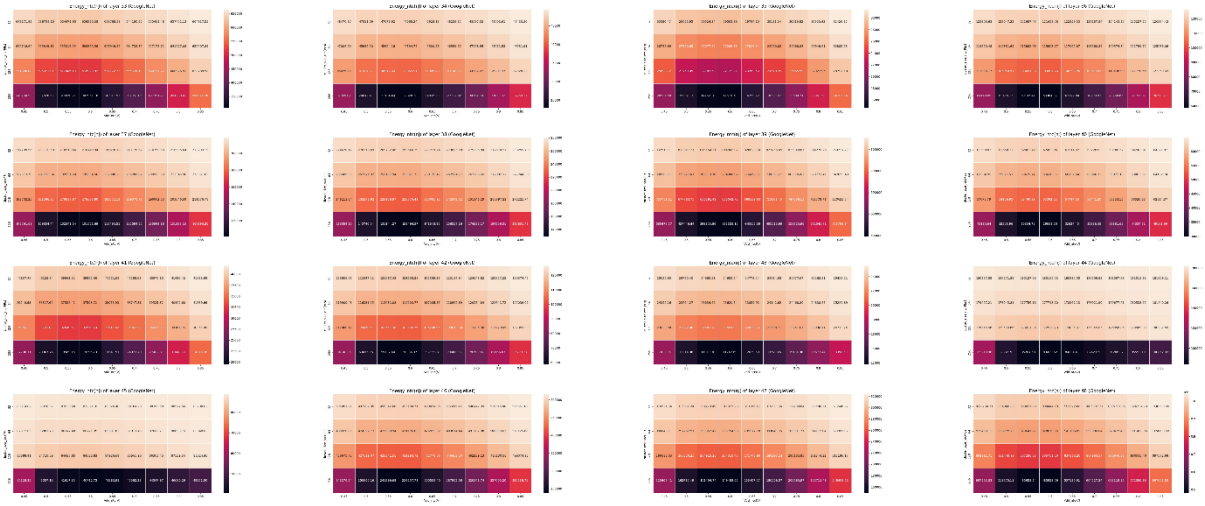


Figure A.7: Energy consumption of layers 33 to 48 (GoogLeNet)

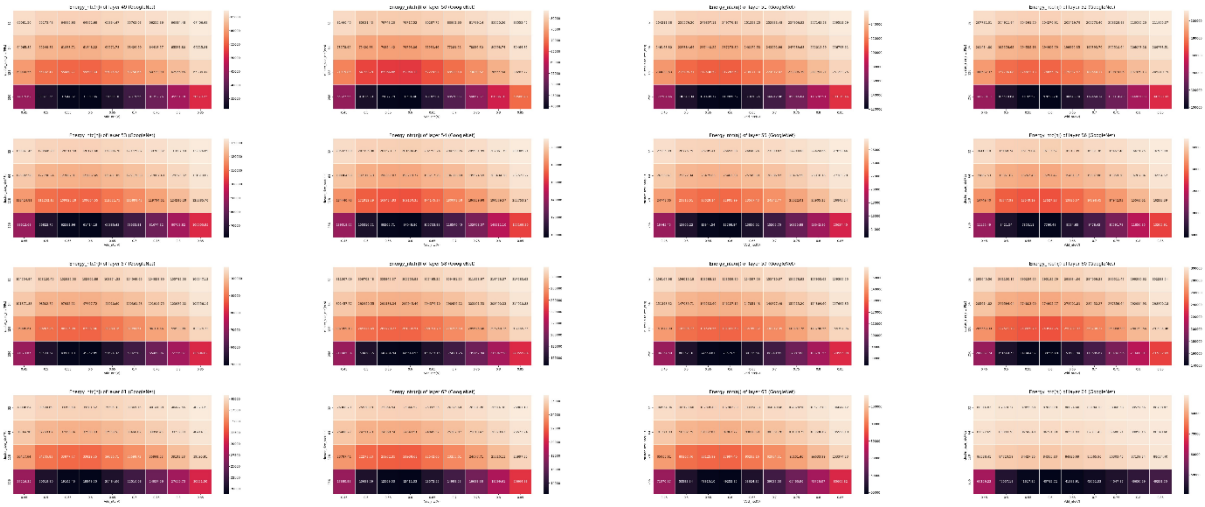


Figure A.8: Energy consumption of layers 49 to 64 (GoogLeNet)

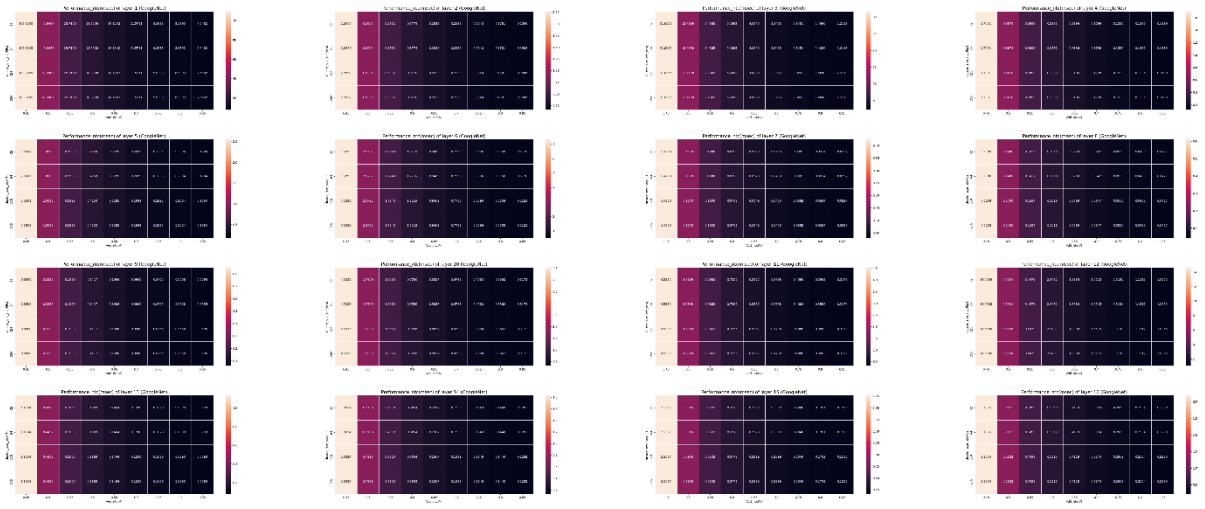


Figure A.9: Performance of layers 1 to 16 (GoogLeNet)

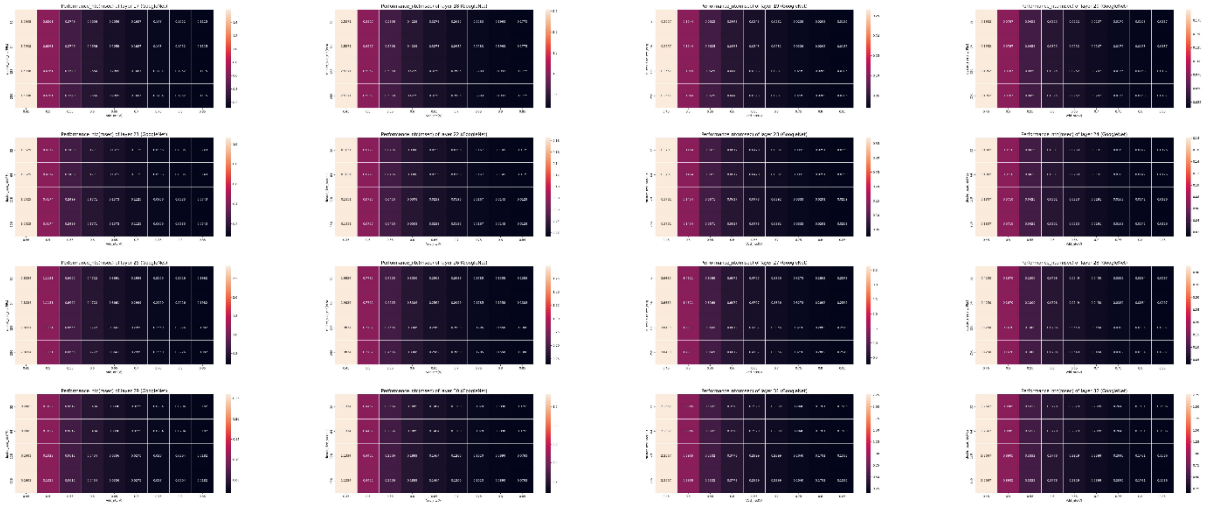


Figure A.10: Performance of layers 17 to 32 (GoogLeNet)

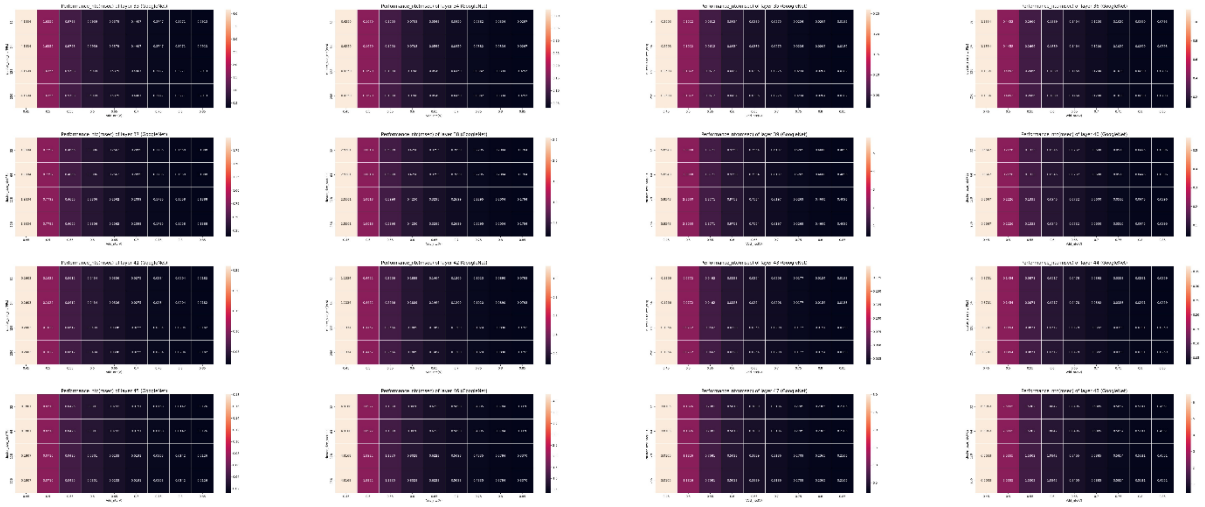


Figure A.11: Performance of layers 33 to 48 (GoogLeNet)

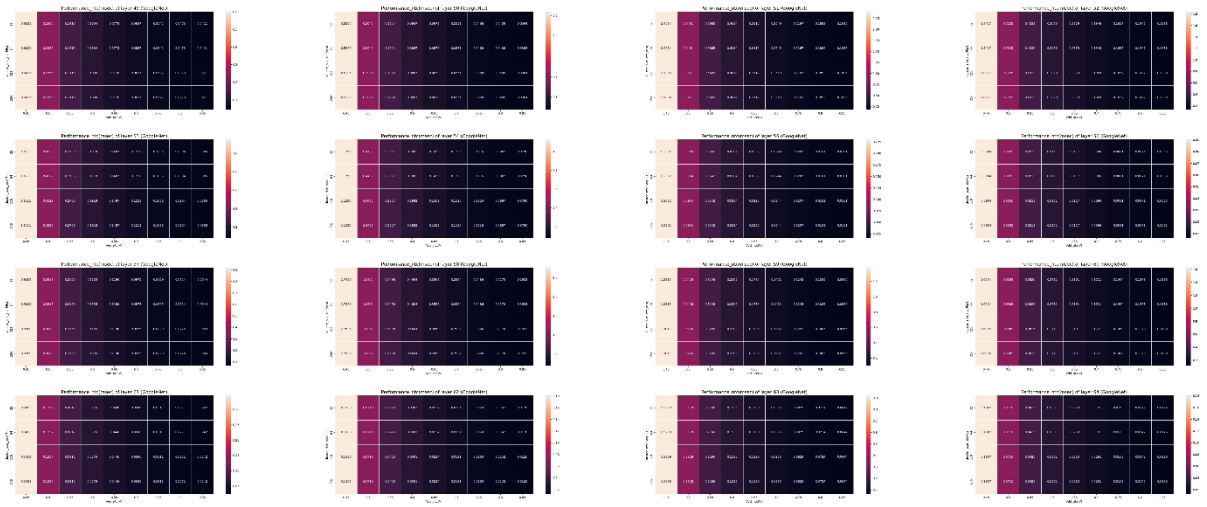


Figure A.12: Performance of layers 49 to 64 (GoogLeNet)

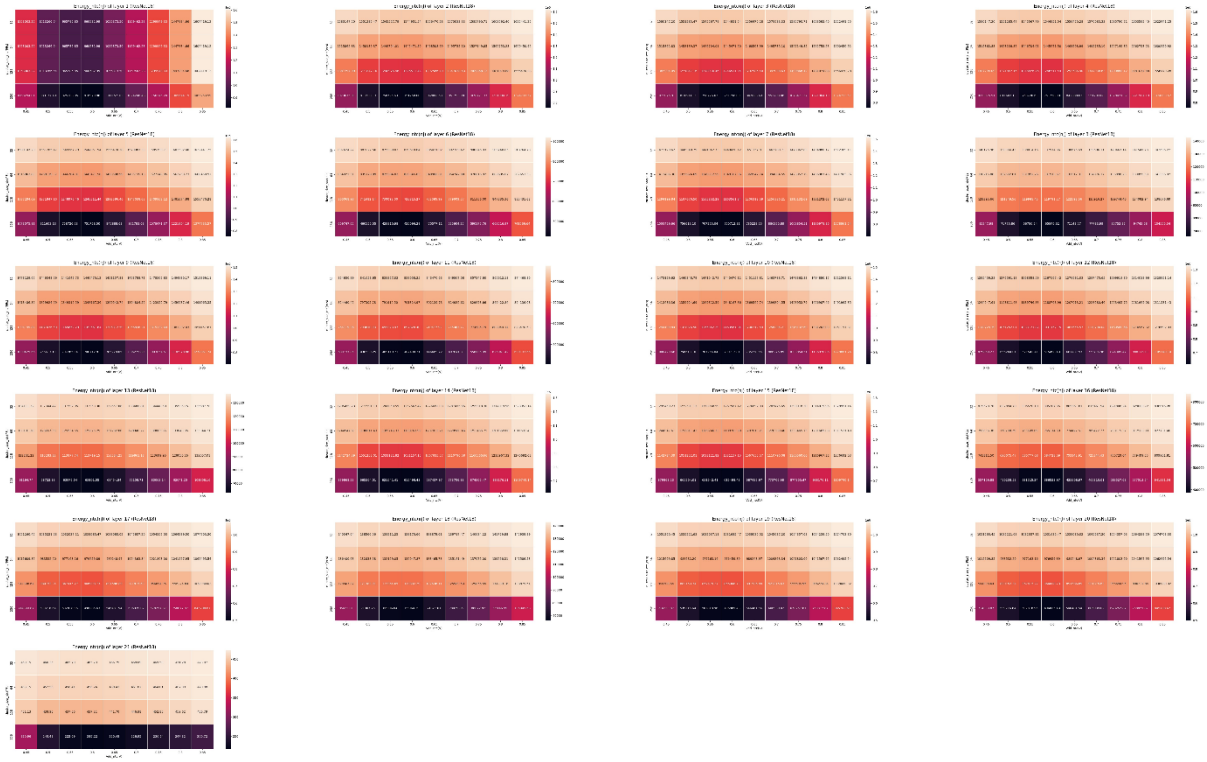


Figure A.13: Power consumption of layers 1 to 21 (ResNet-18)

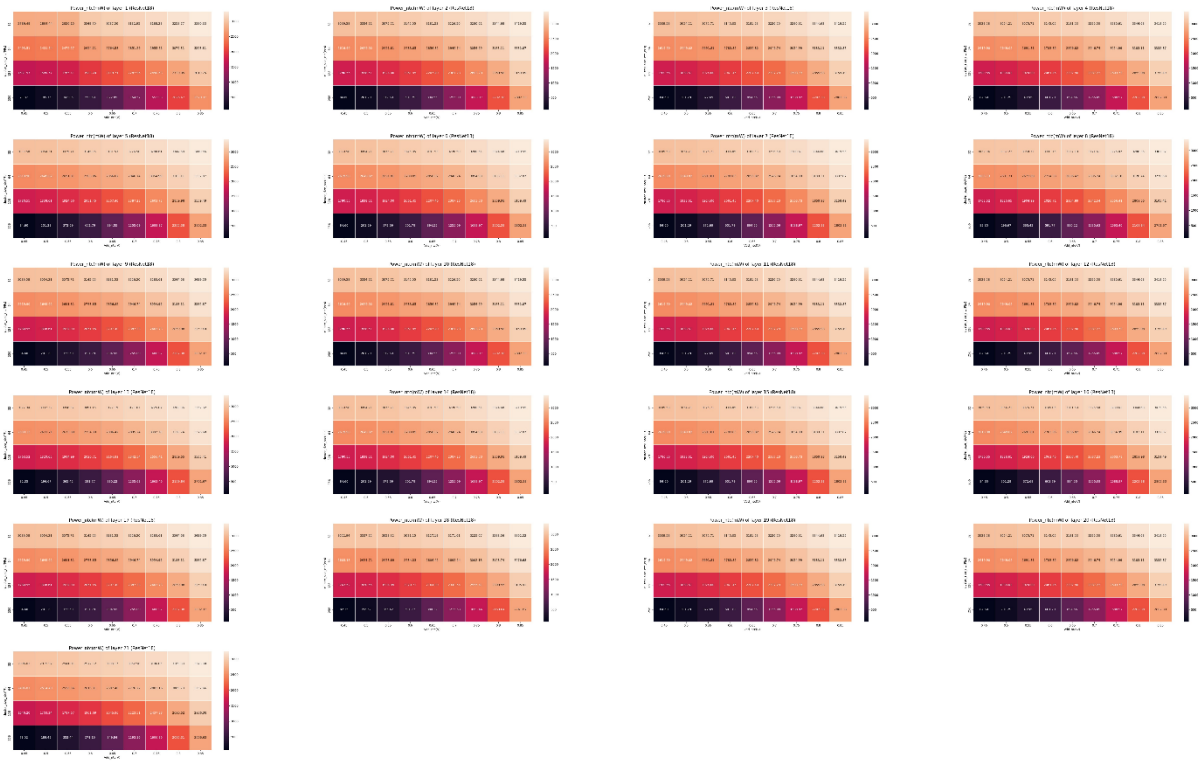


Figure A.14: Energy consumption of layers 1 to 21 (ResNet-18)

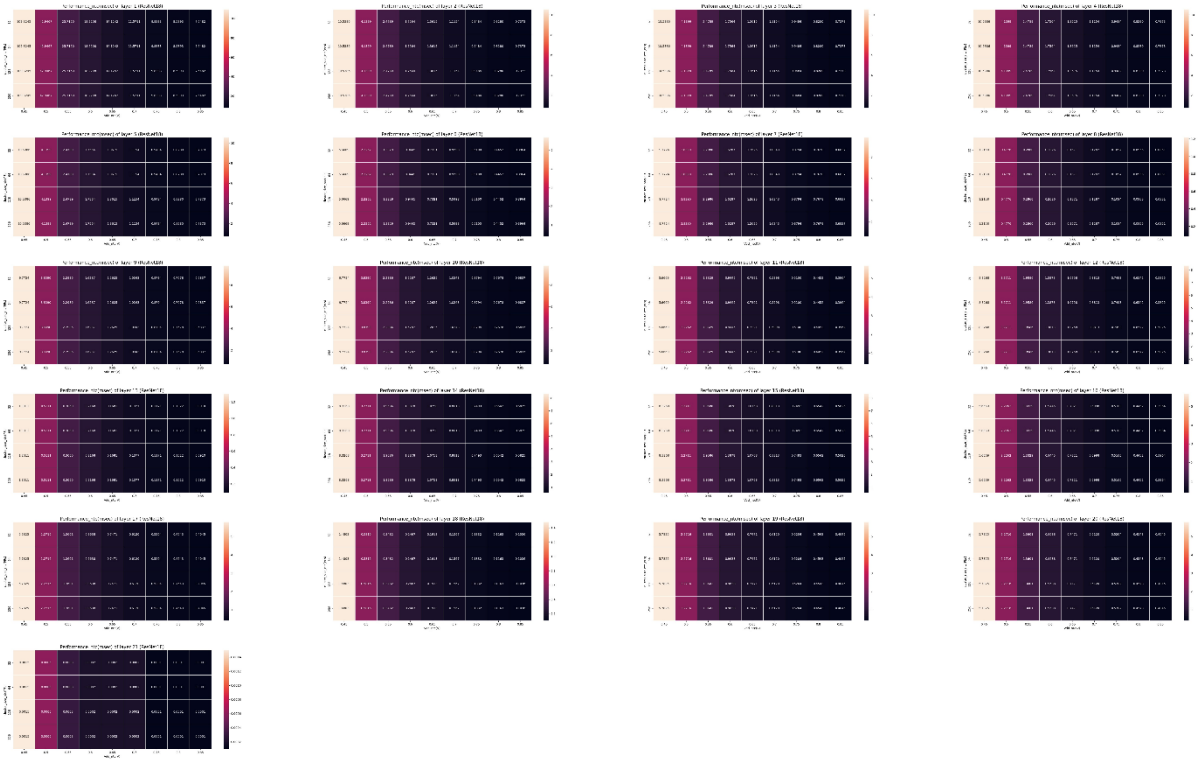


Figure A.15: Performance of layers 1 to 21 (ResNet-18)

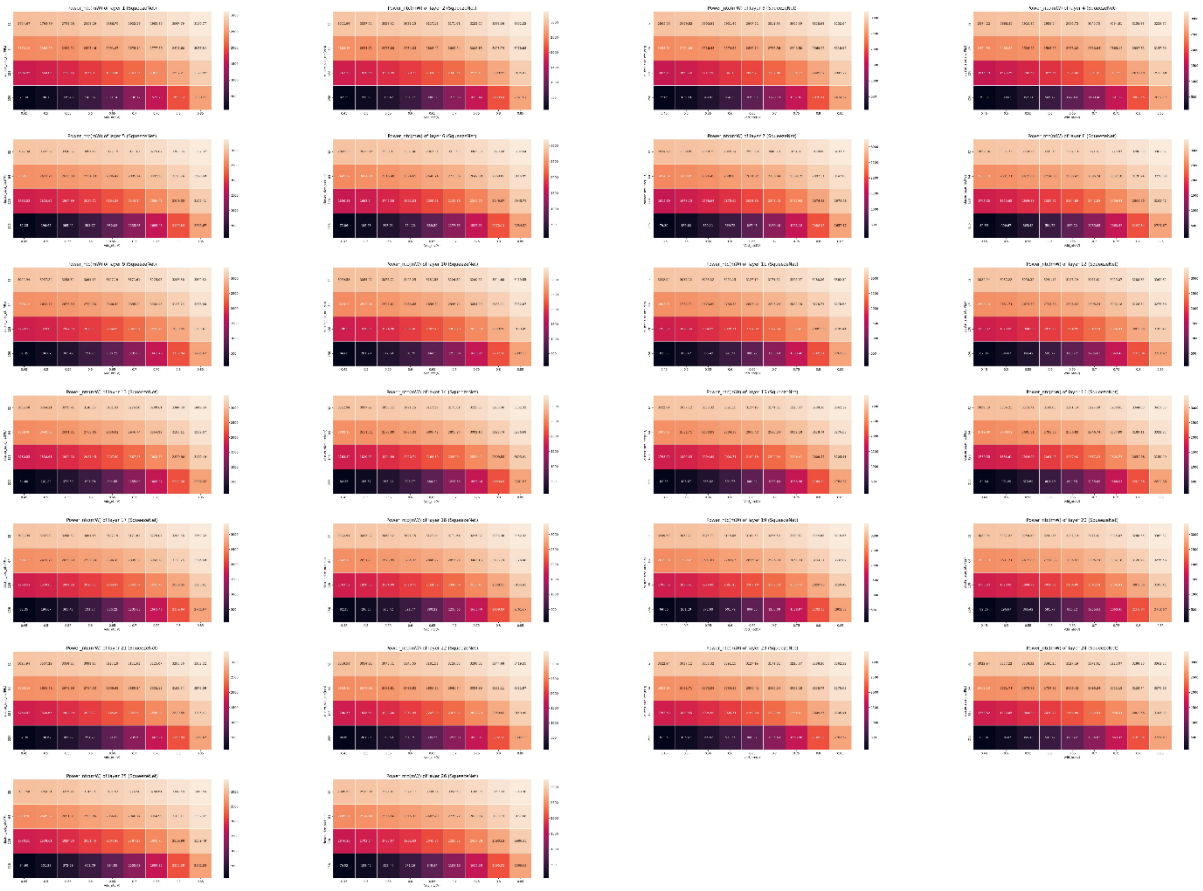


Figure A.16: Power consumption of layers 1 to 26 (SqueezeNet)

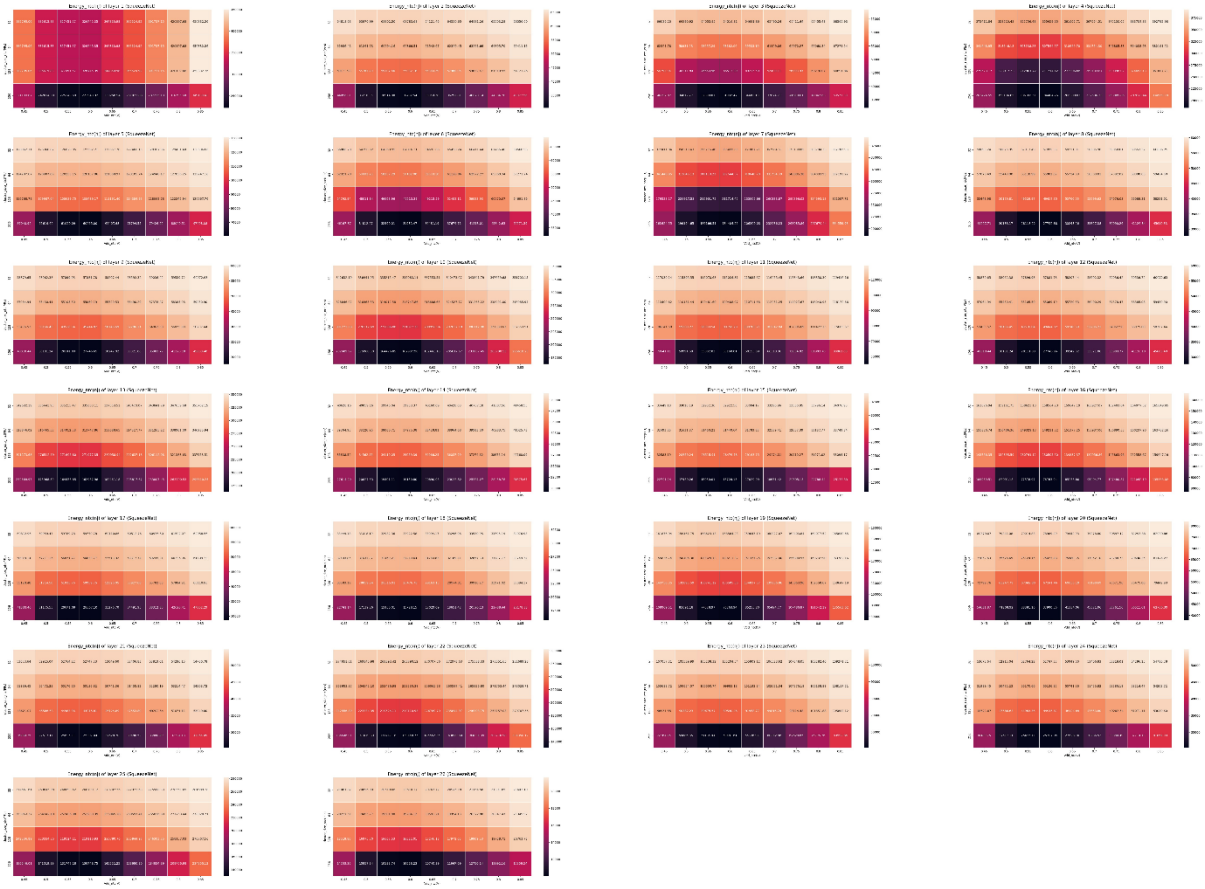


Figure A.17: Energy consumption of layers 1 to 26 (SqueezeNet)

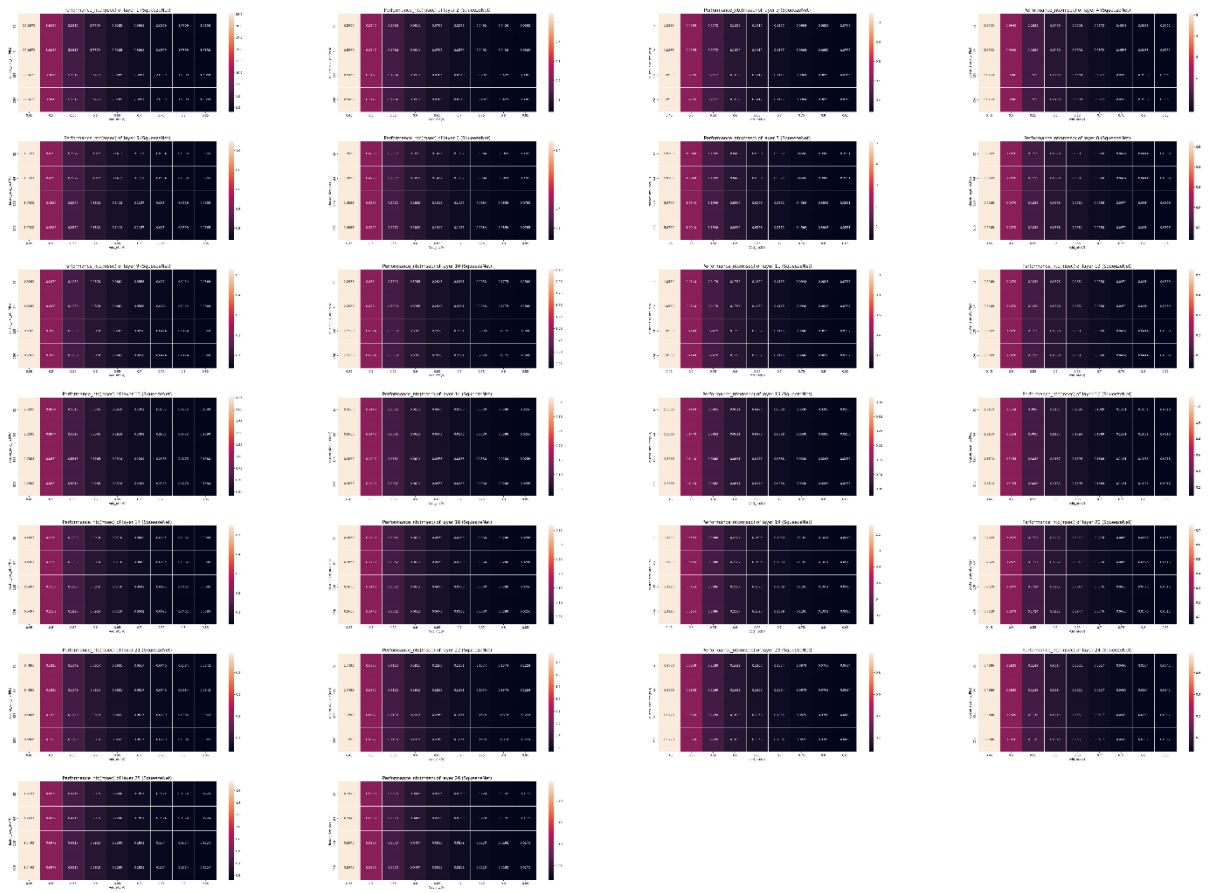


Figure A.18: Performance of layers 1 to 21 (SqueezeNet)

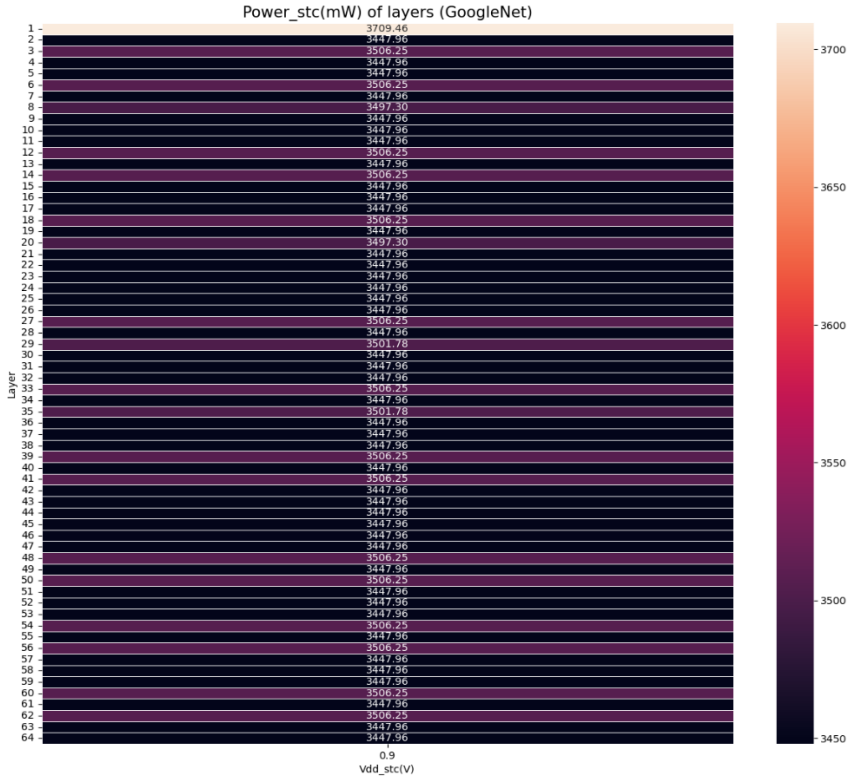


Figure A.19: Power consumption of layers 1 to 64 in STC (GoogleNet)

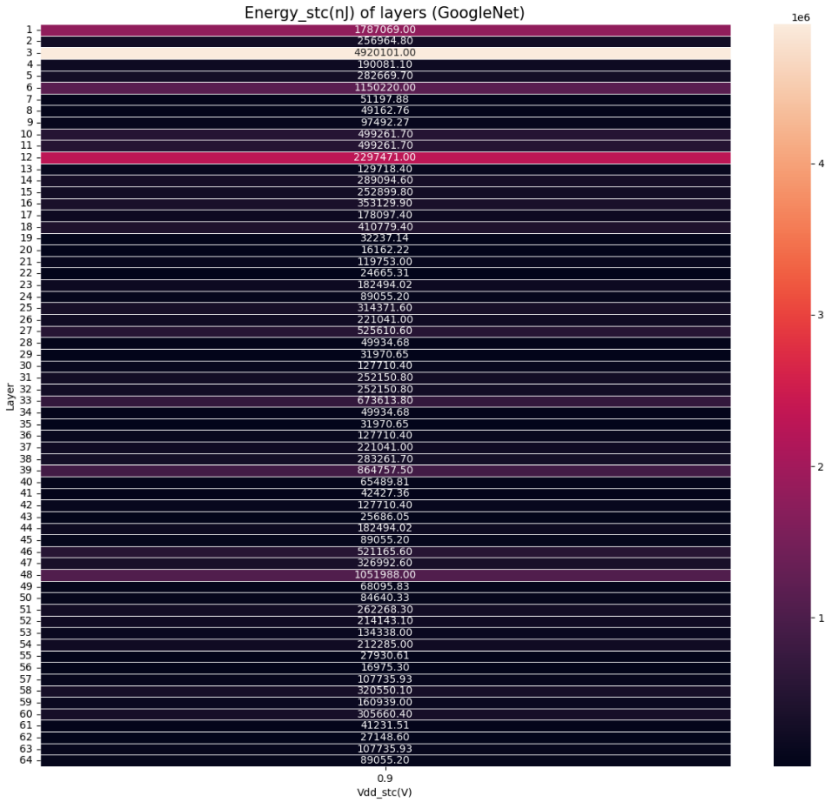


Figure A.20: Energy consumption of layers 1 to 64 in STC (GoogLeNet)

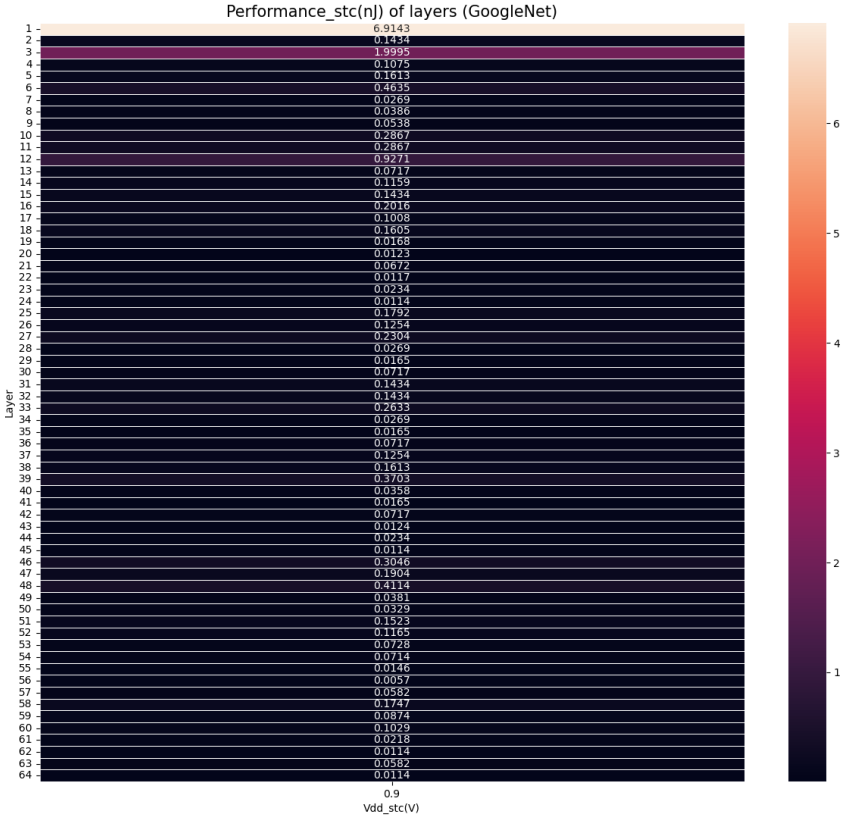


Figure A.21: Performance of layers 1 to 64 in STC (GoogLeNet)

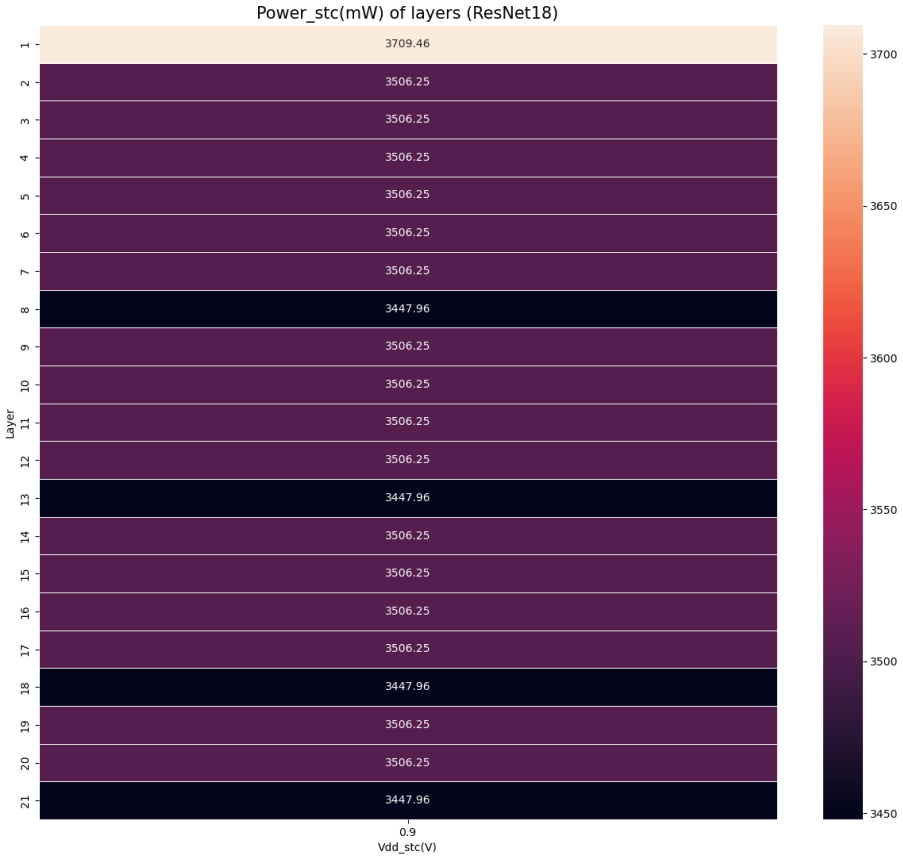


Figure A.22: Power consumption of layers 1 to 21 in STC (ResNet18)

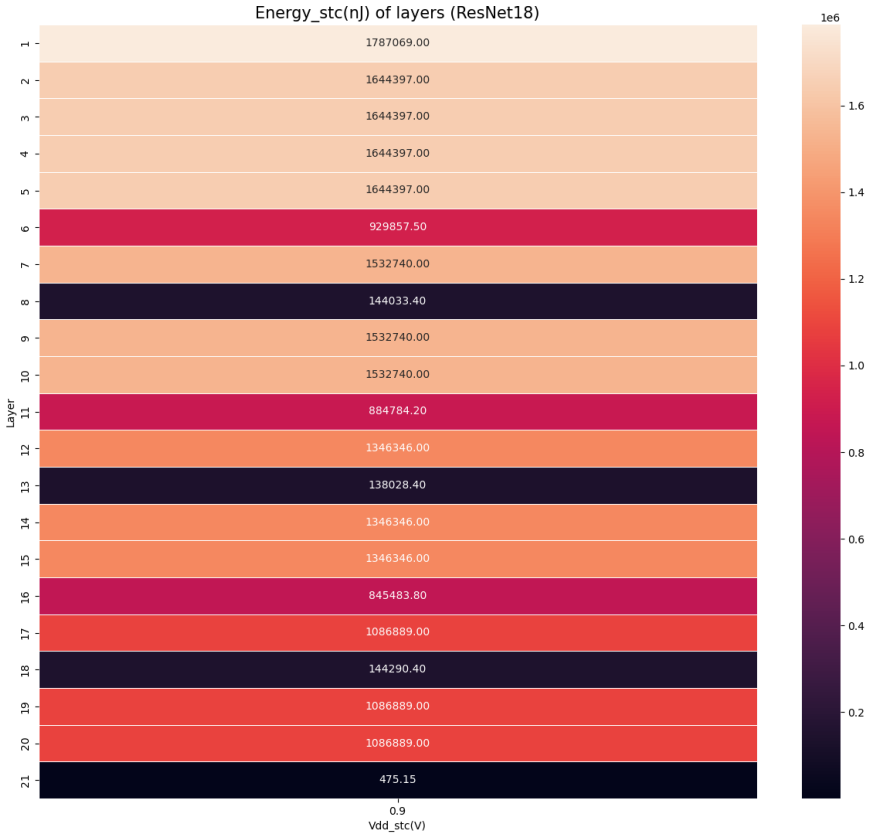


Figure A.23: Energy consumption of layers 1 to 21 in STC (ResNet18)

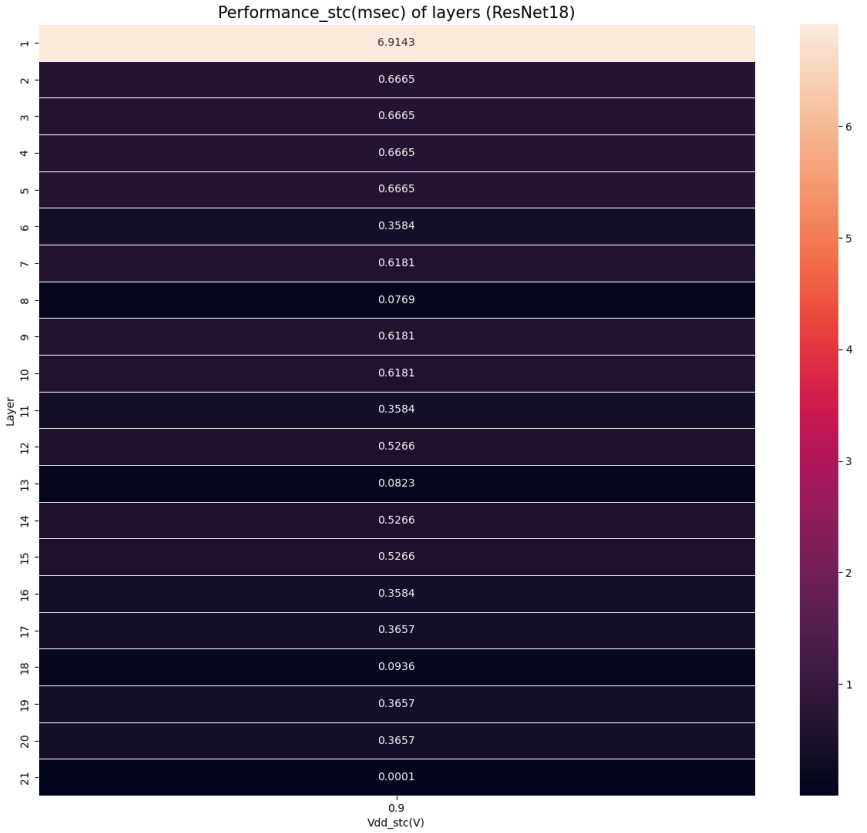


Figure A.24: Performance of layers 1 to 21 in STC (ResNet18)

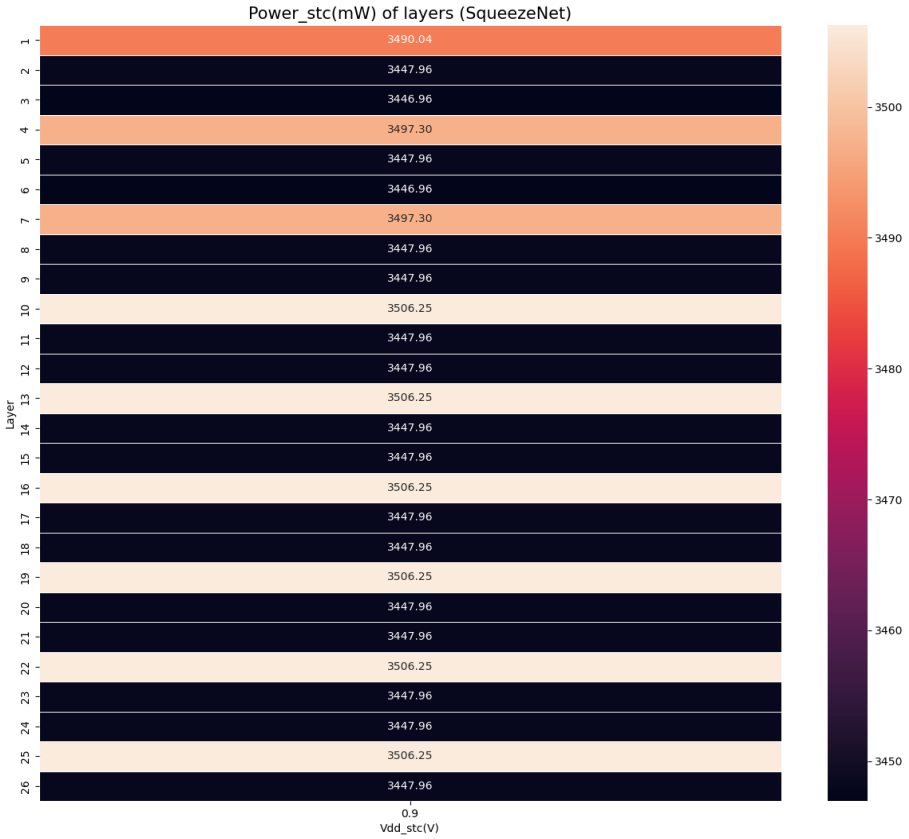


Figure A.25: Power consumption of layers 1 to 26 in STC (SqueezeNet)

Figure A.26: Energy consumption of layers 1 to 26 in STC (SqueezeNet)

Figure A.27: Performance of layers 1 to 26 in STC (SqueezeNet)

Data and code is available at:

https://github.com/Kronos78-cloud/NTV_DNN_thesis