UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

IAN DOS REIS NODARI

A Framework for FPGA-Accelerated Evaluation of Approximate Arithmetic Circuits in Systolic Arrays

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering

Advisor: Prof. Dr. Antonio Carlos Schneider Beck Filho

Porto Alegre
July 2026

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profª. Marcia Cristina Bernardes Barbosa
Vice-Reitor: Prof. Pedro de Almeida Costa
Pró-Reitora de Graduação: Profª. Nádya Pesce da Silveira
Diretor do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenador do Curso de Engenharia de Computação: Prof. Paulo Francisco Butzen
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

““Winning without taking risks
is triumphing without glory.”
— Ayrton Senna

ACKNOWLEDGEMENTS

Finally, after four long years, I am at the end of my bachelor's degree. And I would like to give my thanks to the people who have supported me along the way.

Firstly, I would like to thank Eduardo, Joélia and Aileen, my family, for always loving and supporting me along the years, in easy or hard times. You have taught me my values, and made me who I am today. I am eternally grateful.

I would also like to thank the friends I have made during my years in college, and especially the people from my research group. My time in university has some of my fondest memories, and its thanks to you all.

During my time in research, I was guided by great teachers, who I would like to thank for the patience and guidance: Dr. Michael Jordan (not the one from basketball), Profa. Fernanda, Prof. Azambuja and, especially, Prof. Caco. I thank you for having been given the opportunity to work in CHI^{2}P.

Finally, I would like to give a special thanks to three of my dearest friends from my years in university: Carol, Mateus and Kalian. You were some of the first people I met in university, and are still my friends after all these years. Meeting you three was one of the greatest gifts my years in engineering have given me.

ABSTRACT

The growing demand for deep learning applications across edge and cloud environments has intensified the adoption of domain-specific architectures, particularly systolic array-based accelerators for efficient matrix computations. While these architectures achieve high throughput and energy efficiency, their scalability is fundamentally constrained by the cost of multiply–accumulate (MAC) units, which dominate area and power consumption. Approximate computing offers a promising solution by exploiting the intrinsic error resilience of neural networks, enabling more efficient arithmetic units with minimal impact on inference accuracy. But diverse approximate circuits have a variable impact depending on neural network architecture and dataset, thus requiring the automation of exploration to effectively determine approximation impact on a target workload. In this scenario, we propose a framework for cross-layer evaluation of approximate arithmetic in systolic DNN accelerators. The framework integrates hardware synthesis, FPGA-based prototyping, and neural network evaluation to systematically analyze the trade-offs between power, area, and accuracy. By generating Pareto-optimal design points, our framework enables informed selection of approximation strategies tailored to specific workloads and architectures. Our main results include the evaluation of the thirteen circuits of EvoApproxLib-lite with three deep convolutional neural networks.

Keywords: Approximate Computing. ASIC. Systolic Array. CNN.

Um Framework para Avaliação Acelerada por FPGA de Circuitos Aritméticos Aproximados em Arranjos Sistólicos

RESUMO

A demanda cescente por aplicações de aprendizado profundo desde a borda até a nuvem tem intensificado a adoção de arquiteturas de domínio específico, particularmente aceleradores baseados em arranjos sistólicos para a execução eficiente de computações com matrizes. Enquanto essas arquiteturas alcançam uma grande largura de execução e eficiência energética, sua escalabilidade é fundamentalmente restrita pelo custo de unidades de multiplica-acumula, que dominam o consumo de área e potência. A computação aproximativa oferece uma solução promisora para aproveitar a resiliência a erro intrínseca de redes neurais convolucionais, permitindo unidades aritméticas mais eficientes com um impacto mínimo na acurácia de inferência. Mas diversos circuitos aproximativos tem um impacto variável dependendo de arquitetura de rede neural e dataset, portanto demandando a automatização da exploração para determinar com eficácia o impacto da aproximação nas tarefas a serem executadas. Nesse cenário, nós propomos um framework para avaliação de circuitos aritméticos aproximados em aceleradores sistólicos de redes neurais profundas. O framework integra síntese de hardware, prototipação baseada em FPGA e avaliação de redes neurais para sistematicamente analisar os impactos de aproximação em potência, área e acurácia. Ao generalizar pontos Pareto-ótimos de design, nosso framework permite a seleção informada de estratégias de aproximação para tarefas específicas e arquiteturas. Nossos principais resultados incluem a avaliação dos treze circuitos da EvoApproxLib-lite com três redes neurais convolucionais profundas.

Palavras-chave: Computação Aproximativa. ASIC. Arranjo Sistólico. CNN.

Contents

1 INTRODUCTION **13**

2 BACKGROUND **16**

2.1 Deep Dearning and Workloads **16**

2.1.1 Challenges in Deep Learning Execution 16

2.2 AI accelerators **17**

2.3 Systolic Arrays **18**

2.4 Approximate computing **20**

2.4.1 Error Metrics 21

2.5 The Chipyard Ecosystem **22**

2.5.1 Gemmini 22

3 RELATED WORK **25**

3.1 Neural Network Resiliency **25**

3.2 Approximate Circuits **25**

3.3 Approximate Circuits and Machine Learning **26**

3.4 Approximate Circuits and AI Accelerators **26**

3.5 Frameworks for evaluation of AxC in Deep Learning **27**

4 THE FRAMEWORK **28**

4.1 Generation Flow **28**

4.2 Evaluation Flow **29**

4.2.1 Gemmini and I/O 31

4.2.1.1 The RoCC Interface 31

4.2.1.2 The Tilelink Interface 32

4.2.1.3 The PTW Interface 34

4.2.2 Software Adaptations 34

4.2.3 Validation 35

4.3 VLSI Flow **36**
4.4 Analysis Flow **38**

5 METHODOLOGY **40**
5.1 Approximate Circuits **40**
5.2 Networks **40**
5.2.1 Network Architectures 40
5.2.2 Datasets 41
5.3 Technology for mapping **42**
5.4 Evaluated Metrics **42**

6 RESULTS **43**

7 CONCLUSION **50**
7.1 Future Works **50**

ANNEX A — AXI PROTOCOL **52**

ANNEX B — TILELINK PROTOCOL **55**

List of Figures

Figure 1.1 LLM number of parameters over the years 14

Figure 2.1 Systolic architecture example 18
Figure 2.2 Representation of a OS SA 19
Figure 2.3 Distribution of network weights for MNIST and SVHN 21
Figure 2.4 Gemmini Architecture 22
Figure 2.5 TPU Architecture 23

Figure 4.1 Proposed framework illustrated 29
Figure 4.2 Alteration made in the MAC unit 30
Figure 4.3 Evaluation Flow illustrated 30
Figure 4.4 SoC implemented for Kria testing 35
Figure 4.5 VLSI Flow illustrated 37
Figure 4.6 Analysis Flow illustrated 38

Figure 6.1 Power and Area results for the circuits 43
Figure 6.2 Accuracy results for the circuits 44
Figure 6.3 Score for each circuit 45
Figure 6.4 Error heatmap for the 1KVA circuit 46
Figure 6.5 Error heatmap for the 1KVQ circuit 47
Figure 6.6 TPU Floorplan 48

Figure A.1 AXI handshake operation 52

Figure B.1 Example of Tilelink transactions 57

List of Tables

Table 4.1 RISCV64G unprivileged ISA 32
Table 4.2 Memory Map Used for the RoCC Interface 33
Table 4.3 Gemmini's Instruction Set Architecture 34
Table 4.4 Kria Utilization 36

Table 5.1 Relevant error metrics for the approximate circuits library 41
Table 5.2 Accuracies for the tested networks 41

Table 6.1 Pareto optimal set for power vs accuracy 48
Table 6.2 Results for optimal circuit analysis 49
Table 6.3 Area estimates for Google's TPUv1 49

Table A.1 AXI Signals 54

Table B.1 Tilelink signals 55
Table B.2 Tilelink transaction types 56

LIST OF ABBREVIATIONS AND ACRONYMS

AMBA	Advanced Microcontroller Bus Architecture
AxC	Approximate Computing
AXI	Advanced eXtensible Interface
DMA	Direct Memory Access
DNN	Deep Neural Network
DSA	Domain-Specific Architecture
DSE	Design Space Exploration
EP	Error Probability
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GeMM	General Matrix Multiplication
I/O	Input/Output
IS	Input Stationary
LUT	Lookup-table
MAC	Multiply-Accumulate
MAE	Mean Absolute Error
MPSoC	Multiprocessor System-on-Chip
MRE	Mean Relative Error
MxM	Matrix Multiplication
OS	Output Stationary
PE	Processing Element
PPA	Power, Performance, Area
PTW	Page Table Walker
RoCC	Rocket Core Coprocessor

RTL	Register Transfer Level
SoC	System-on-Chip
TL	Tilelink
TL-C	Tilelink Cached
TL-UH	Tilelink Uncached-Heavyweight
TL-UL	Tilelink Uncached-Lightweight
TPU	Tensor Processing Unit
WCE	Worst-Case Absolute Error
WS	Weight Stationary

1 INTRODUCTION

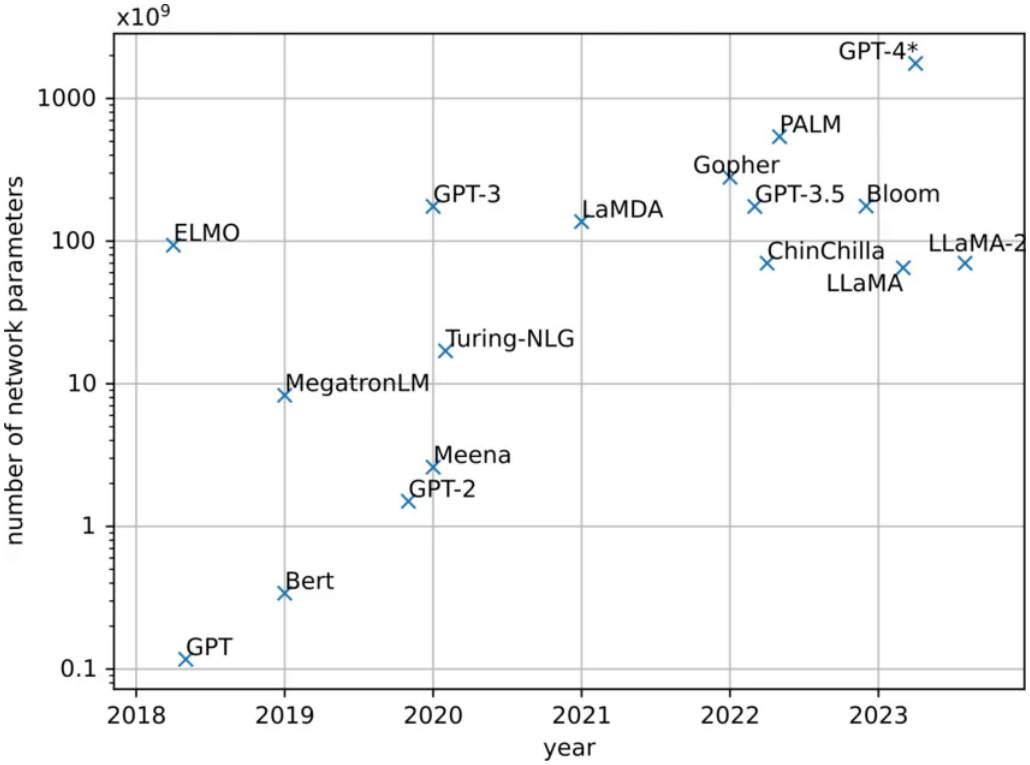
Artificial intelligence has propelled advancements in computer vision, enabling applications such as image classification, object detection, GenAI or natural language processing through Deep Neural Networks (DNNs). This performance, however, comes at a severe computational cost: these neural networks can require millions to billions of Multiply-ACcumulate (MAC) operations per execution. Figure 1.1 shows the size in parameters of several large language models over the years. The number of weights has risen in a seemingly exponential fashion, with requirements for compute energy growing progressively more relevant. This constraint has impacted the adoption and scalability of deep learning systems.

To mitigate this computational burden, the industry has adopted Domain-Specific Architectures (DSA) for deep learning acceleration, with devices such as Google's TPU since 2015 (SATO, 2017) in the cloud environment and Apple's Neural Engine in 2017 (Apple Corporation, 2017) in mobile environments. These accelerators enable the usage of deep learning in several domains, but come at a power and area footprint in the Systems-on-Chip (SoC) they are implemented in.

Some of these accelerators, such as Google's TPU, implement systolic arrays (SA) to accelerate matrix multiplication. A SA consists of an array of Processing Elements (PE) that pass data from one PE to another, executing computation as data passes through its complex. These arrays implement several hundreds or thousands of MAC operations in their system, amounting to a significant power and area footprint on the whole system: combinational logic for MACs can occupy up to 75% of the architecture (QIN et al., 2020), and scaling to accommodate larger models further exacerbates silicon and energy requirements. In this context, Approximate Computing (AxC) addresses this restriction by exploiting the intrinsic error-resilience of neural networks, allowing reduced-precision arithmetic to simplify multipliers and adders, thereby enhancing hardware efficiency without substantial accuracy loss.

Despite the availability of approximate multiplier libraries, their impact on systolic arrays affects both hardware metrics (e.g., energy consumption, silicon area, and throughput) and the accuracy of neural network computations, since errors in MAC operations propagate through the array. Importantly, while accuracy loss is correlated to circuit approximation, it is not through a linear relationship. On the same likeness, area and power

Figure 1.1 – LLM number of parameters over the years



Source: (GERSTMAYR; MANZL; PIEBER, 2024)

reduction is not linear to approximation, thus requiring an empirical exploration of circuit impact in specific networks and hardware designs, which limits Design Space Exploration (DSE) to small or simplified scenarios.

This cost is exacerbated by the difficulty of testing approximate arithmetic circuits. Off-the-shelf hardware does not support approximation; thus, workarounds need to be found to validate such systems. For these VLSI systems, there are three domains in which methods for circuit validation can be classified: Simulation based, the cycle-accurate approach using simulators such as VCS and Xcelium, emulation-based, which consists of utilizing behavioral models to execute software, and prototyping-based, which consists mainly of implementing in FPGA or engineering-sample ASICs (MASSOUD et al., 2022). Cycle-accurate simulation can take days or weeks to run even relatively compact networks, and emulation-based approaches bring forth problems with behavioral accuracy and flexibility when dealing with across-the-stack approximations.

In this scenario, one alternative to test the impact of approximation in deep learning accelerators is the usage of FPGA systems. With FPGA, we can evaluate the whole accelerator with approximation introduced, while dodging the hardship of running in cycle-accurate simulations. This allows us to keep hardware fidelity and achieve a significant throughput with complex neural networks.

With this in mind, this work presents a framework for design space exploration of approximate circuits in systolic arrays. We intend to facilitate the evaluation of libraries of approximate circuitry in deep learning workloads by harnessing FPGAs for execution of neural networks on deep learning accelerators through prototyping, giving special emphasis to real world scenarios through the use of constraints for a target application. The tool presented takes as input a set of configurations, approximate circuits, synthesis constraints, technology files, neural networks and constraints for the target application and outputs the optimal circuit to be implemented in the system.

2 BACKGROUND

2.1 Deep Dearning and Workloads

Deep Learning (DL) is a branch of machine learning which consists of the use of Deep Neural Networks (DNN) to solve complex problems which can't easily be reduced to deterministic algorithms, thriving in computer vision, natural language processing and image generation. Generally, as machine learning, deep learning can mostly be summarized as approximating a function $f(x_1, x_2, x_3..., x_n) \rightarrow (y_1, y_2, y_3, ...y_m)$, where the set of x elements are parts of a feature set and the set of y elements are some desired answer, be it classes of inference, bounding boxes or text.

There are three types of network architectures that dominate computer vision tasks: MLPs for small, simpler datasets, CNNs for more sophisticated and intricate problems and Visual Transformers (ViT) for state-of-the-art accuracy. MLPs tend to only acceptably adjust to very small or simple problems, lacking scalability, and ViTs are extremely large, which makes them unappealing or nonviable for several tasks, with CNNs being some of the most well-used network architectures for such tasks (LI et al., 2022). CNNs work by successively extracting features from intermediate, progressively more abstract feature maps, and inferring final results either by using a smaller MLP on the final layer or by taking the probabilistic distribution resultant in the last layer through the *softmax* operation.

Generally, four classes of operations are utilized in CNNs: Convolution, Activation, Pooling and Normalization. Convolution consists of a *feature extraction*, activation aims to introduce nonlinearities in inference and training to impede the model from adjusting as a single layer (known as *model collapse*), pooling aims to reduce feature size to lessen computational costs and normalization is introduced to scale features to the same interval, simplifying training and inference by making it harder for very large or very small numbers skewing the model (exploding and vanishing gradients).

2.1.1 Challenges in Deep Learning Execution

The great breakthrough in DL as a path for modeling complex problems came through the success of a deep convolutional neural network in the ILSVRC-2012 competition for image classification in the Imagenet dataset (KRIZHEVSKY; SUTSKEVER;

HINTON, 2012). Krizhevsky introduced the usage of GPUs for DL training, a first for the time, which introduced a path for major scaling in model complexity. Since then, neural networks have grown in size, with new models using millions to billions of parameters, and executing degrees of magnitude more arithmetic operations than the numbers of parameters they have.

In this context, the industry has adopted GPUs for inference and training. But notably, several system integrators have started adopting custom solutions for AI acceleration, implementing DSAs to accelerate the fundamental operations which DL consists of both in cloud and edge devices.

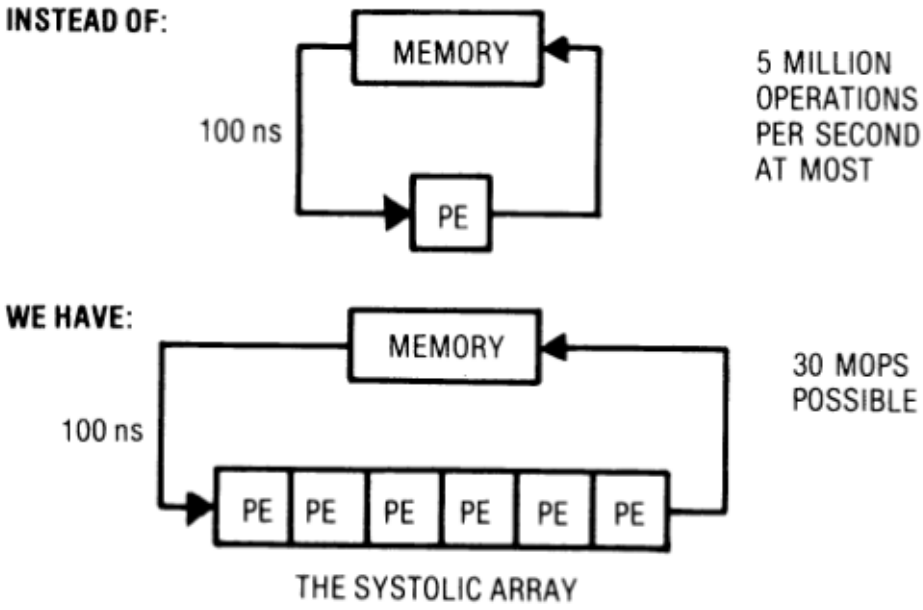
2.2 AI accelerators

While GPUs have dominated AI inference and training for their massively parallel architectures, in recent years specialized AI accelerators have been developed to tackle the particular demands of these workloads in the most diverse environments, with targets of better performance and energy efficiency. Google introduced its TPU line of cloud accelerators as early as 2015 (SATO, 2017), with recent versions powering the training and deployment of their LLMs (Google, 2025), and Qualcomm introducing their Hexagon NPU lineup in 2017 to target growing demand for DL acceleration in mobile applications (MAHURIN, 2023).

Most AI accelerators can be viewed as an array of blocks that accelerate the individual tasks to which neural networks reduce to. Google's TPUv2 consists of three main execution units: the Matrix Multiplication (MxM) unit, the scalar/vector unit (responsible for activation and normalization) and a unit for matrix transposition and reshaping (NORRIE, 2020), Qualcomm's Hexagon NPU consists of a general purpose scalar unit, a vector unit for parallel compute and a tensor unit for matrix multiplication, and several similar DSAs have been implemented over the years in diverse fields (VENKATARAMANI et al., 2021)(CHEN et al., 2017)(NVIDIA Corporation, 2018).

One of the most compute-intensive operations in DL is matrix multiplication. From a compute complexity standpoint, it is a $O(N^3)$ operation, challenging both memory bandwidth and arithmetic capacity of hardware. In this context, systolic arrays emerged as a popular construct to maximize compute per memory access in MxM operations.

Figure 2.1 – Systolic architecture example



Source: (KUNG, 1982)

2.3 Systolic Arrays

Systolic arrays are a hardware construct commonly used to implement a MxM unit. Initially proposed in the 80s (KUNG, 1982), they were proposed to combat the von Neumann bottleneck: normal computers require data and instructions to execute from memory to execute; systolic arrays, on the other hand, execute multiple operations on a single data element over their dataflow, effectively amortizing the memory access cost. Figure 2.1 shows an example of a systolic array.

Systolic arrays can be made to implement matrix multiplication very efficiently. Considering a matrix multiplication $C = A \times B$, one of the matrices is fixed on the array while the other two are passed through. Thus, systolic arrays come mainly in the form of three dataflows: Input Stationary (IS), where the A matrix is fixed, Weight Stationary (WS), where the B matrix is fixed and Output Stationary (OS), where the C matrix is fixed. To understand this, first start with the elementary matrix multiplication algorithm in the algorithm 2.1 for an operation $A \times B = C$:

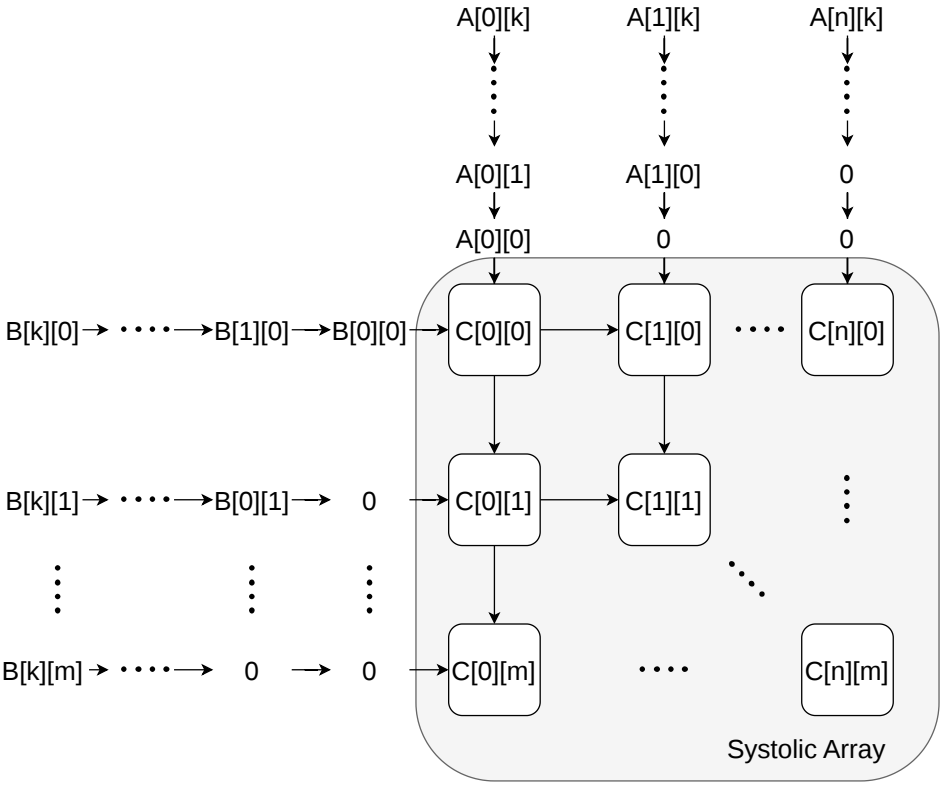
Firstly, consider the case of line matrix per column matrix multiplication for a representation of a OS dataflow. Data is passed through a single processing element, successively multiplying $A[0][k] \times B[k][0]$. Generalizing this for an A matrix with n lines and a B matrix with m columns, by passing elements through, the systolic array

Algorithm 2.1 Matrix Multiplication

$C \leftarrow 0$
for $i = 0; i < dim_i; i \leftarrow i + 1$ **do**
  for $j = 0; j < dim_j; j \leftarrow j + 1$ **do**
    for $k = 0; k < dim_i; k \leftarrow k + 1$ **do**
      $C[i][j] \leftarrow A[i][k] * B[k][j] + C[i][j]$
    end for
  end for
end for

reutilizes each n_{th} line with each m_{th} column, achieving compute parallelism for the array's size given that m and n have greater or equal size to that of the array, but compute latency linear to the internal dimension of the matrix multiplication plus the delay of the zero-padded propagation.

Figure 2.2 – Representation of a OS SA



Source: The Author

We can observe from figure 2.2 that the output stationary serializes the matrix multiplication with respect to the internal dimension K while paralelizing the M and N dimensions with the number of processing elements they implement. Similarly, both the IS and WS sequentialize one of the outer dimensions and parallelize over the remaining two. Instead of passing weights through the complex, an IS/WS systolic array fixates the

input/weight matrix and passes the partial values of the target matrix. Both use the same hardware, but transpose matrices in different manners.

Systolic arrays tend to not be as big as some of the matrices they are utilized to compute. Thus, the concept of tiling is applied. The MxM operation is reduced to a series of smaller operations which fit the array, allowing them to compute greater operations through successive partial executions. The tile is divided based on the output sections of the matrix in the OS case, and in the lines/columns of the input/weight matrix in the IS/WS dataflows.

The WS dataflow is more widely utilized in the industry: both the IS and WS fixate one of the inputs of the scalar multiplications executed across the array, thus meaning one of the inputs of the matrix multiplication is always fixed, reducing switching rate across the array, and WS is prefered to IS due to the memory layout of the input matrix having better locality. Google's TPU and EdgeTPU both implement their MxM unit through a WS array.

Given that systolic arrays implement hundreds or thousands of MAC circuits in their complex, they have a significant power and area utilization at the whole chip scale. In this context, since the main target application of the hardware accelerators in which they are implemented is machine learning, we can harness the error resiliency of deep learning by utilizing Approximate Computing (AxC) to make arithmetic circuits cheaper.

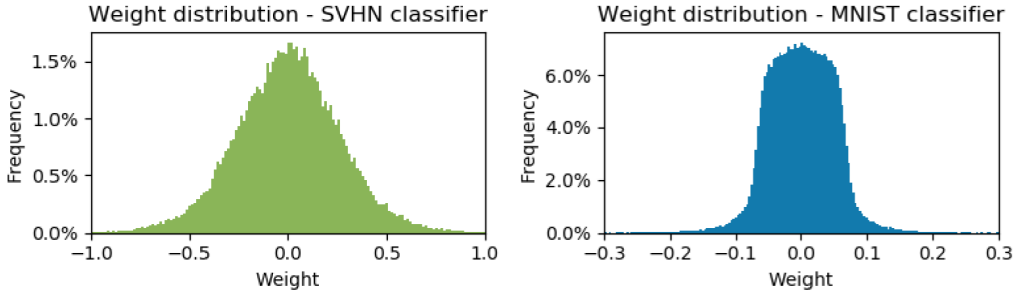
2.4 Approximate computing

AxC is a compute paradigm based around making operations cheaper by introducing some degree of error for computations which are error-resilient. It can be divided between approximate software and approximate hardware, with approximate software consisting of skipping steps or simplifying them to save time and approximate hardware for creating circuits that bring some degree of error for better Power, Performance and Area (PPA).

Approximate arithmetic focuses specifically on creating approximate arithmetic circuits capable of giving a reasonably close result for less hardware. Even small approximations can lead to respectable gains in performance and power: hardware area and power usage do not grow linearly with logical depth, with ideal logic sizing implying switching energy superlinear to logical depth (WESTE; HARRIS, 2010).

Arithmetic circuits can add up to nearly 75% of a conventional systolic array's

Figure 2.3 – Distribution of network weights for MNIST and SVHN



Source: (VASICEK; MRAZEK; SEKANINA, 2019)

area (QIN et al., 2020), with multiplication circuits amounting to about 80% of arithmetic circuit area and adder area for accumulation closing it out. Literature has shown that, for circa 1.5% average error, up to 70% of area in a multiplication can be saved, thus showing a significant potential for gains in DSAs for hardware acceleration (HASHEMI; BAHAR; REDA, 2015).

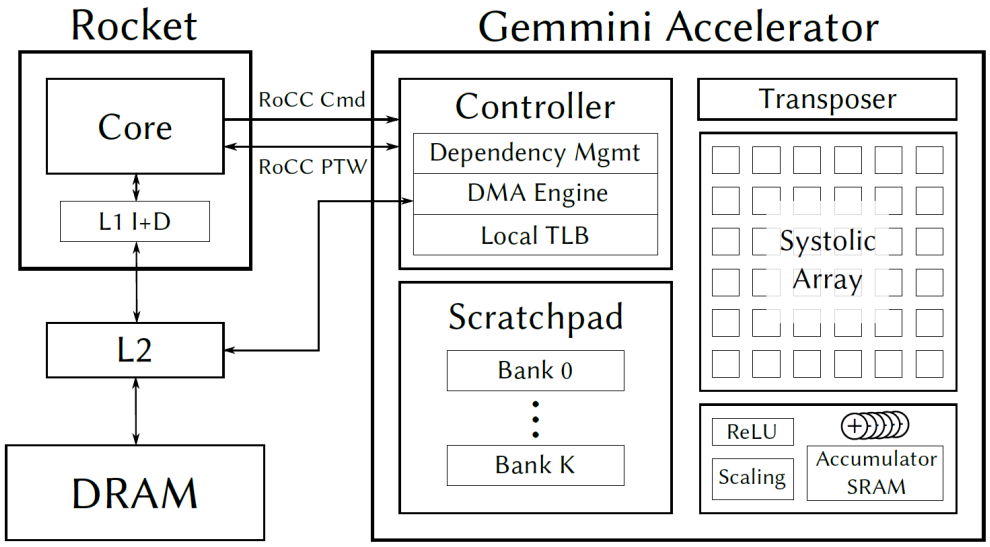
One main challenge is that the impact of employing approximate circuits is not very highly correlated to neither accuracy nor power/area. The impact of approximation on neural networks is not linear, and can vary significantly depending on network topology and dataset. On the other hand, sequential and clock tree elements can account for a significant portion of systolic array area and power; thus, power and area estimates need to account for the whole MxM unit to properly model arithmetic efficacy in reducing hardware utilization.

2.4.1 Error Metrics

Approximate circuits are commonly characterized based on formally-defined error metrics, such as their average error or chance of error for some operation. One of the ways to create approximate circuitry is through Cartesian Genetic Programming (GCP), starting with a base population of accurate circuits and progressively pruning each netlist while characterizing the resulting error, removing the lineages which explore the required accuracy metrics.

Deep learning poses a challenge for direct correlation between error and accuracy loss: network weights are not linearly distributed. Figure 2.3 shows the weight distribution of two neural networks for image classification. Since some weights are more common than others, some errors introduced by approximation can have a greater impact than others.

Figure 2.4 – Gemmini Architecture



Source:(GENC et al., 2021)

2.5 The Chipyard Ecosystem

Chipyard consists of a set of tools and designs maintained by Berkeley University to explore SoCs. It includes CPUs, peripherals and a toolflow to test through simulation and prototyping and implement ASICs, with the notable inclusion of the Gemmini systolic DNN accelerator.

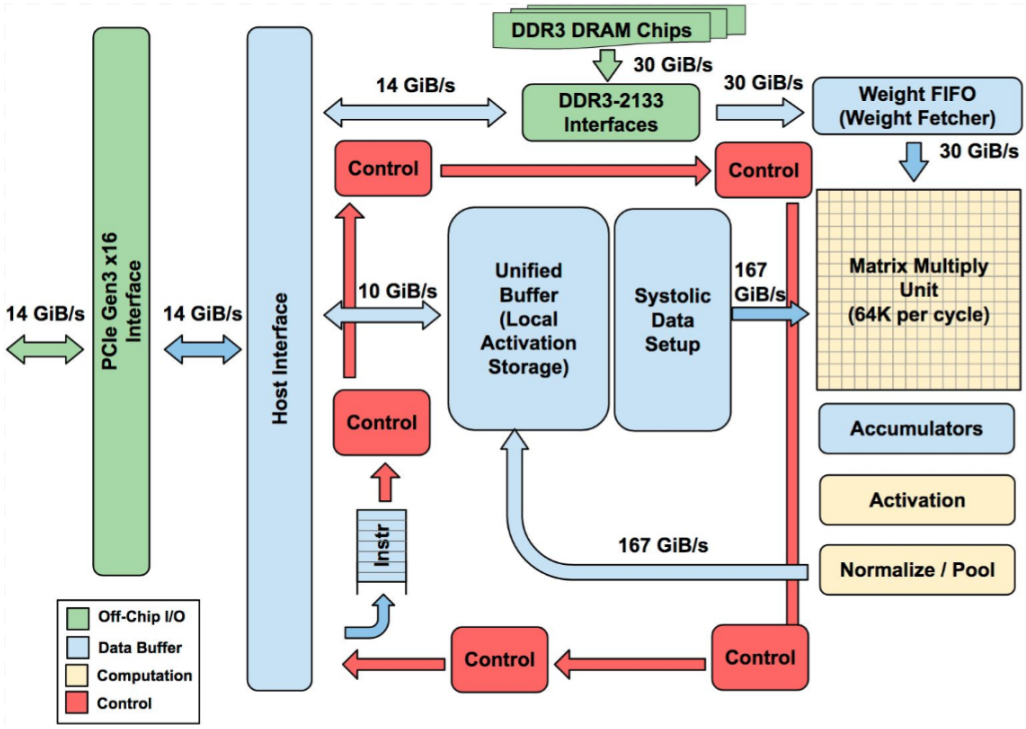
Chipyard offers IPs such as the Rocket Chip (ASANOVIć et al., 2016), the Berkeley Out of Order Machine (BOOM) (ZHAO et al., 2020), and a configurable crossbar utilizing the Tilelink communication protocol for interaction with peripherals, with JTAG, UART, SPI and memory subsystems being offered by Berkeley's solution. Among the offered IPs, the Gemmini systolic accelerator offers a platform for testing with an architecture similar to an industrially used system.

2.5.1 Gemmini

Gemmini is an Open Source AI accelerator generator described in Chisel and available as an open source IP as part of the Chipyard ecosystem. Figure 2.4 shows the block design of Gemmini's architecture (GENC et al., 2021).

Gemmini accelerates all four fundamental building blocks of convolutional neural networks: It implements convolution through the Im2Col and GeMM algorithms utiliz-

Figure 2.5 – TPU Architecture



Source: (JOUPPI et al., 2017a)

ing its systolic array, activation through non-linear activation units based on Lookup-Tables(LUT), pooling through a reshape operation embedded in its DMA, executing pooling by moving data from its private memory to the system's main memory, and normalization through a similar mechanism to that of the pooling layers.

Gemmini programming involves embedding a set of instructions in the host CPU's program through a set of C macros(the *RoCC instructions*) which implement simple operations, such as moving data from main memory to the accelerator's scratchpad, executing matrix multiplication with data in its private memory and moving data back to main system memory. It also provides an API to control accelerator behavior, including activations, pooling, normalization, stride and dilation.

A more high-level programming interface is also made available through a set of C functions implementing several common kernels in computer vision networks, including convolution, matrix multiplication, residual adds and normalization, similar to how a Tensorflow network would be implemented.

Lastly, Gemmini also provides a port for acceleration on the ONNX Runtime, a framework similar to Tensorflow or PyTorch to create and train neural networks. By calibrating the networks to properly work with Gemmini, it can accelerate networks implemented in a high-level programming language such as Python.

Gemmini is the open-sourcely available IP closest to Google's TPU architecture. Figure 2.5 shows the TPU block diagram. The accelerator itself implements a system very similar to Gemmini, but as a discrete accelerator instead of densely coupled with a CPU as a coprocessor. By utilizing Gemmini as a base, we can more realistically predict the impact of any alteration in a system utilized in the industry.

3 RELATED WORK

3.1 Neural Network Resiliency

It is common knowledge in academia that neural networks are intrinsically error-resilient. Most studies on neural network resiliency come in the context of fault-tolerant applications: Zhang et al. (2022) proposes a methodology for neural network error resilience characterization through fault injection for radiation-resilient systems, trying to minimize error characterization time through the development of a vulnerability model with a small quantity of fault injections and its use to select critical weights for fault injection.

In the context of approximate computing, studies have been conducted on characterizing neural network resiliency to approximation: Hanif, Hafiz & Shafique (2018) proposes a methodology to characterize error resiliency of neural networks under approximation, introducing gaussian noise across the network to simulate the impact of approximation. While this is effective to detect the error resilience of certain layers of the network to approximation and the general resilience of a network to approximation, it is not an accurate way to predict the impact of approximation on a neural network.

3.2 Approximate Circuits

Most of the work on approximate arithmetic circuits in the past has been made targeting mainly image processing and encoding/decoding, with several circuits being proposed and tested on these workloads. Hashemi, Bahar & Reda (2015) proposes a multiplier family with leading-one detection for input-aware reduced width multiplication, showing good results for area and power reduction. Saadat, Bokhari & Parameswaran (2018) uses the approximation of a sum of logarithms to produce a low-bias floating point multiplier and gives key insights on the challenges of approximate floating point operations.

Modern approaches to approximate circuit design generally utilize some automated approach for exploration, taking a population of initial circuit architectures and iteratively introducing approximations to find optimal designs or employing approximation during logic synthesis. Mrazek et al. (2016) proposes the usage of Cartesian Genetic

Programming (CGP) for approximation on RTL level, applying the genetic algorithm to find optimal solutions targeting neural networks, and tests multiplier effectiveness using number classification datasets. Buccolini et al. (2026) explores the design space through input aware approximate logic synthesis, achieving an average power-delay-area reduction of up to 17.5% with minimal accuracy loss.

In general, modern research points in the direction of automated circuit design with some heuristic for testing generated multiplier effectiveness. This comes with the major challenge of how to model and test the accuracy of the massive DL models employed on modern applications.

3.3 Approximate Circuits and Machine Learning

Ansari et al. (2020) provides a work on the impact of approximation on training and inference. They consider two types of multipliers: those purposefully built as approximate arithmetic circuits, and those automatically generated through the CGP. They confirm that the approximate circuit impact in neural networks is not governed by a linear relationship, meaning the utilization of approximate circuits has to be made aware of software execution, but their work is restricted to two simple datasets of number classification.

Mrazek, Sekanina & Vasicek (2020) uses CGP to generate a series of approximate circuits and tests them through a modification in Tensorflow to allow the exploration of approximation through GPU accelerated emulation. This has the positive impact of significantly accelerating testing, allowing their work to test approximations up to a Resnet-164 trained on the CIFAR-100 dataset. The major restriction of their work is the lack of accelerator accuracy, with estimates being based solely on multiplier power and area gains.

3.4 Approximate Circuits and AI Accelerators

Utilizing approximate circuits in accelerators has been explored in the past, but evaluating both the impact on hardware and accuracy has been a hardship in the past. Elbtity et al. (2022) proposes an approximate tensor processing unit and evaluates neural networks, but is restricted by only utilizing a single architecture of multiplier with variation in parameters and the use of MNIST and variations.

Spantidi et al. (2021) utilizes positive and negatively biased multipliers, alternating

to mitigate average error in DL accelerators, targeting reduced power consumption without requiring network retraining. They map multipliers by layer, thus requiring network-specific accelerators; but they lack the exploration of various multipliers, testing only three degrees of approximation between positive or negatively biased multiples.

3.5 Frameworks for evaluation of AxC in Deep Learning

Kwak, Kim & Kim (2024) leverages PyTorch to explore approximate circuit impact through GPU accelerated simulation, utilizing C or C++ programs to model approximate circuit and DL accelerator behavior, while Vaverka et al. (2020) emulates approximate circuitry with CUDA by utilizing lookup tables with quantized values. But by utilizing emulation, these works restrict framework flexibility by making it harder to model approximation impact in different structures of DNN accelerators. Both provide GPU accelerated frameworks for approximate circuit exploration with emulation, thus lacking hardware accuracy.

Jain et al. (2022) implements approximate systolic arrays with thousands of multipliers and models approximation impact in a Imagenet-trained Resnet50, but lacks the accurate mapping of hardware impact due to modeling approximation through a highly optimized implementation of matrix multiplication which estimates accuracy loss due to approximation.

In general, frameworks for exploration of approximate circuits focus on emulating approximate circuit behavior without worrying about full system accuracy, with the exploration of approximation-aware accelerator architectures posing an obstacle due to compute restrictions on simulation time or hardware accuracy of emulation.

4 THE FRAMEWORK

In this section, the proposed framework is elaborated upon with details on how it was implemented.

Figure 4.1 shows the proposed framework. It serves the purpose of evaluating approximate circuits in systolic arrays for deep learning, with an analysis on their viability for a specific application. The inputs of the framework are a library of approximate circuits, a set of neural networks, the parameters for the accelerators to be generated, a constraint set for a target application and both a standard cell library and constraint set for VLSI implementation, with the result being the optimal circuit for the application.

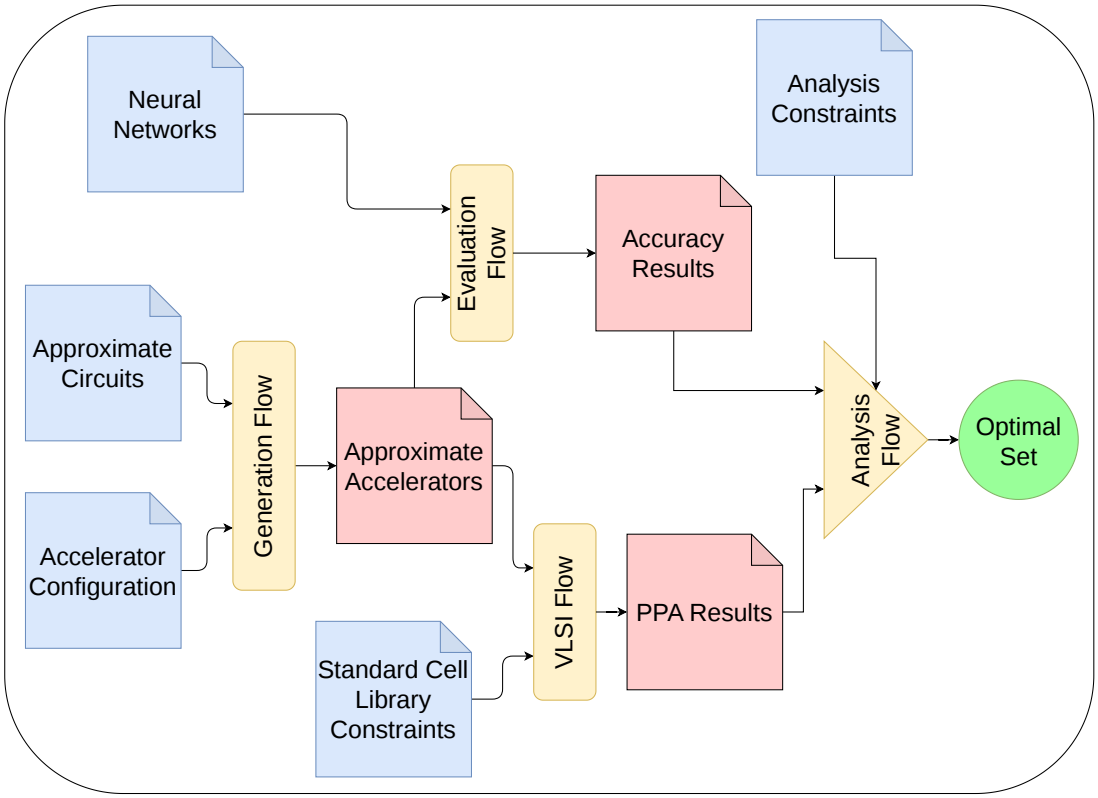
The framework consists of four major steps: the Generation flow, where the approximate hardware's description is generated from the approximate circuit library and the accelerator configuration, the Evaluation Flow, which implements and extract accuracy of the approximate hardware in the target neural networks with execution in FPGA with the accelerators generated and the networks provided by the user, the VLSI Flow, used to estimate the power and area characteristics of the generated approximate accelerators with the standard cell library and constraint set, and the Analysis Flow, responsible for selecting the optimal circuit for the user's target application with the data from the VLSI and Evaluation flows.

4.1 Generation Flow

The objective of this step of the framework is to automatically produce a series of hardware descriptions for the systolic arrays with the approximate circuits implemented. This step takes as input the accelerator configuration for Gemmini and a library of approximate circuits for evaluation, and outputs a set of RTLs of approximate accelerators. To achieve this, we harness Chipyard's hardware generation with modifications to the accelerator.

Chipyard offers multiple ways to generate hardware, but for this instance, generation using the simulation toolchain suffices, as only the Verilog is required for the subsequent steps of the flow. Implementing approximate hardware for the systolic array was achieved through modifying the Gemmini generator's Chisel source code: instead of describing the behavior of each approximate multiplier in Chisel, we implemented blackboxes for

Figure 4.1 – Proposed framework illustrated



Source: The Author

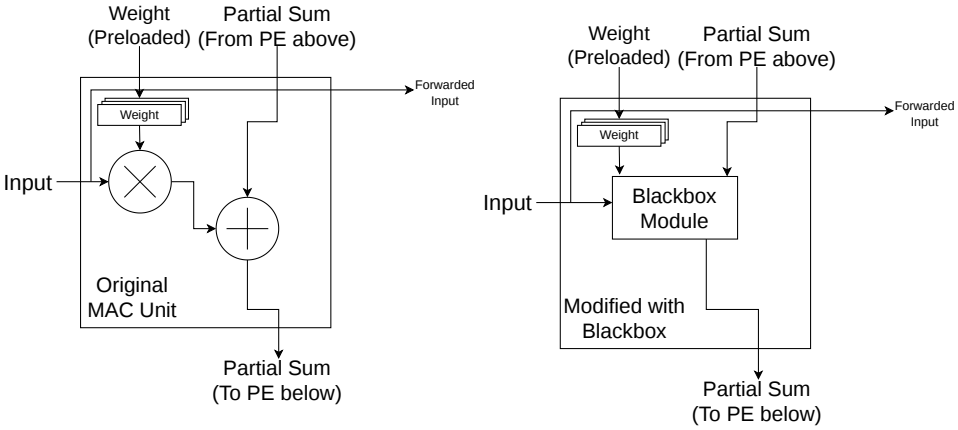
arithmetic circuits. From the hardware generators point of view it behaves as a C volatile, and this means that, from one pass of the hardware generator, any number of approximate accelerators are created using tools such as GNU AWK and SED, which are tools for text parsing and substitution, to modify the circuits in bulk. Figure 4.2 shows how the alteration behaves in a WS PE for reference.

This method for hardware generation is extensible to evaluating several forms for hardware-level approximation, and, by inserting a blackbox for any Gemmini module, it could be applied to activation, requantization or other parts of Gemmini's pipeline while taking advantage of Gemmini's programmability.

4.2 Evaluation Flow

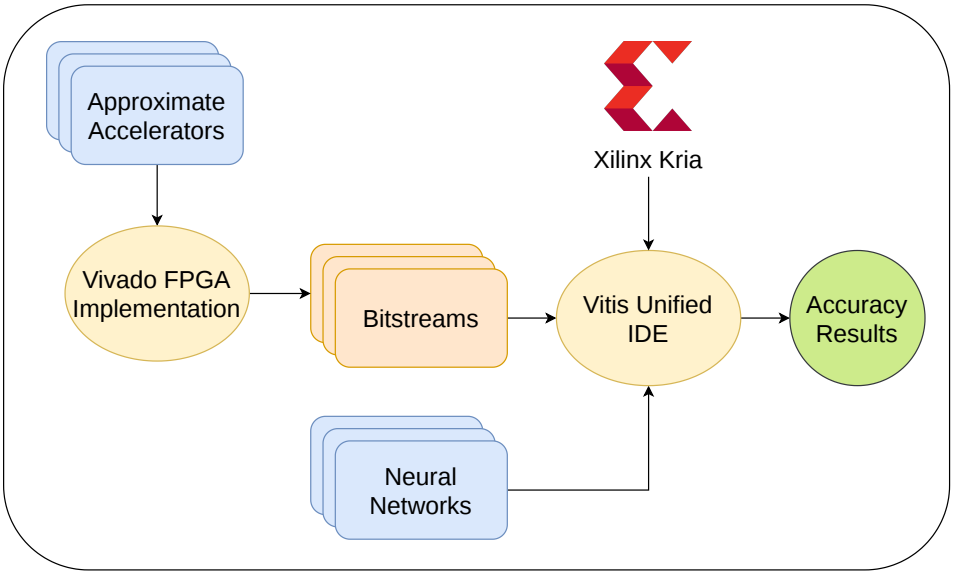
Figure 4.3 shows the evaluation flow of the framework. It serves the purpose of executing inference with the approximate accelerators on neural networks. To achieve this, we implement eacho of the the approximate accelerators from the generation flow in FPGA and execute inference with each neural network given as input to the flow on each

Figure 4.2 – Alteration made in the MAC unit



Source: (ucb-bar, 2019); Adapted

Figure 4.3 – Evaluation Flow illustrated



Source: The Author

accelerator, resulting in the accuracy results for each approximate circuit.

Chipyard provides an FPGA prototyping flow itself: it allows implementing hardware in a local device by utilizing their FPGA toolchain, which support for multiple vendors, or using Firesim, a cloud-based solution to implement and validate using Versal devices with an Amazon F2 instance. Unfortunately, both solutions lack the speed required for evaluating multiple arithmetic circuits.

The locally hosted alternative implements a full SoC in an FPGA, including a core and UART modules in soft-logic. This comes at a severe cost: moving programs to the system's memory can take a very long time. Moving the Resnet50 network with four test images to a SoC with Gemmini implemented in a Nexys Video at 10MHz (the maximum

frequency which finds timing closure, with Gemmini's requantization units being the critical path) takes approximately three hours, which means evaluating several circuits and neural networks could take weeks or months, thus severely harming the scalability of the framework.

Firesim provides an alternative approach to the issue. It offers a Versal-based solution, where respectable frequencies can be achieved. This comes at the cost of design implementation time: implementing a design for these large cloud FPGAs can take tens of hours and requires non-trivial compute capacity, thus making the evaluation of multiple arithmetic circuits non-viable.

With this in mind, our framework provides an alternative approach to implementing Gemmini in FPGA, harnessing the best of both worlds. We utilize small Multiprocessor Systems-on-Chip (MPSoC) FPGAs, implementing only the accelerator in soft-logic while using the SoC implemented in hard-logic for I/O. To do this, we needed to adapt the accelerator to work as a memory-mapped I/O.

4.2.1 Gemmini and I/O

From figure 2.4 we can see the accelerator's communication interfaces, which consist of three buses: The Rocket Core Coprocessor (RoCC) interface, to issue commands to the accelerator, the Page Table Walker (PTW) interface, used for memory virtualization, and the Tilelink (TL) Interface, which serves as the memory access interface for the Direct Memory Access (DMA) unit. Both the RoCC and PTW Interfaces are connected to the host core, while the TL interface is independent of the RoCC master.

4.2.1.1 The RoCC Interface

The RoCC interface consists of a producer-consumer interface where the host is the master and the accelerator is the slave. Every time the RISC-V host detects a RoCC instruction in its execution thread, it dispatches it to the coprocessor responsible for dealing with the instruction. RoCC instruction consist simply of instructions with one of the four custom opcodes specified in the RISC-V unprivileged ISA. Table 4.1 shows the unprivileged ISA opcodes. Gemmini, by default, is mapped to the *custom-3* opcode.

On the subject of signals, 150 bits are used for communicating data from the host to the coprocessor, with the instruction itself and two 64-bit registers being passed as

Table 4.1 – RISCV64G unprivileged ISA

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111(>32b)
00	LOAD	LOAD-FP	***custom-0***	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	*reserved*
01	STORE	STORE-FP	***custom-1***	AMO	OP	LUI	OP-32	*reserved*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	***custom-2***	*reserved*
11	BRANCH	JARL	*reserved*	JAL	SYSTEM	OP-VE	***custom-3***	*reserved*

Source: RISC-V Foundation

arguments. Another 64 bits are passed from the accelerator to the master for responses to the host (in Gemmini's case, only for reading hardware counters for performance). To close it off, there are the signals for handshaking, the busy signal (used for synchronization with accelerator execution) and the host's control state registers, which can be ignored when running the applications in baremetal mode.

Dispatching instructions without using a RoCC-compatible core was made independent of the Chipyard SoC. To do this, we mapped the RoCC I/O to a set of memory-mapped registers and execute instructions through a sequence of reads and writes in the memory region to which the accelerator is mapped on the system.

The RoCC commands issued to Gemmini can be divided into two classes: those for which no response is expected and those for which some data is given back to the host. For the purposes of this work, those with return values can be ignored: all instructions which expect a response consist of hardware counter reads, and thus are not necessary for evaluating accuracy.

Table 4.2 shows the used memory map for the application. To adapt the software, we changed the macros to, instead of generating the special *custom-X* instructions, embed code that executes the successive writes and reads required to dispatch the instruction to the accelerator. For a 32-bit processor, this would be circa seven writes and two reads, which would become a severe bottleneck if not for the CISC-like instructions offered by the accelerator, which are discussed in section 4.2.2.

To support the usage of Ultrascale FPGA, this mapping was done using a small memory connected to the MPSoC using an Advanced eXtendable Interface (AXI), with a AXI-Lite registers module from pulp-platform AXI utilities (KURTH et al., 2022). An overview of the AXI protocol and the modules used is available in Annex A

4.2.1.2 The Tilelink Interface

An overview of the TL communication protocol is available in Annex B. TL is not supported by major FPGA vendors; thus, we require Gemmini to use another protocol.

Table 4.2 – Memory Map Used for the RoCC Interface

Signal	Address from Base	Type (From Accel.)
Command	0x0-0x3	Input
RS1	0x4-0xb	Input
Command Valid	0x14-0x17	Input
Command Ready	0x18-0x1b	Output
Busy	0x1c-0x1f	Output

Source: The Author

The main candidate is AXI, with Advanced Microcontroller Bus Architecture (AMBA) peripherals being available in both Vivado and Quartus, and multiple FPGA parts having native AXI interfaces connected with the programmable logic.

Either adapting the native TL to AXI post hardware generation or modifying Gemmini's DMA to use AXI are valid options; for this work, the first approach was chosen. The target is to keep the core design as close to the original as possible, as it is easier to test and verify a protocol adapter than a DMA unit.

Gemmini implements the TL-UH variation of the protocol, but does not utilize the atomic operations nor the hints. The closest AXI version to this subset is the AXI4 variation, which supports bursts and out-of-order delivery of requests.

We developed a TL-UH to AXI4 protocol adapter to work with Gemmini. Adapting is as simple as mapping Get requests to the AR channel and the Put requests to the AW and W channels, with the B and R channels running to the TL response signals. The differences in semantics require one cycle more for issuing the Put requests through the AW channel before writing the data on the W channel, but a lot of the signals have symmetric functionalities in both protocols:

- The source TL signal is similar to the ID AXI signal
- The mask is the same to the STRB
- the corrupt and invalid signals work similarly to the RSP signals

Bursts require special attention: Tilelink uses a different semantic for signaling bursts, with the SIZE signal being the base 2 logarithm of the number of bytes transferred in the beat. This is mapped to the LEN and SIZE signals of the AXI, with LEN becoming fixed as the bus width of the Tilelink interface and SIZE being the exponentiation of the Tilelink size divided by its bus width in bytes.

Table 4.3 – Gemmini's Instruction Set Architecture

Name	Type	Lengh(Words)
mvin	memory access	1
mvout	memory access	1
config_ex	configuration	1
config_mvin	configuration	1
config_mvout	configuration	1
config_norm	configuration	1
matmul.preload	execution	1
matmul.compute.preloaded	execution	1
matmul.compute.accumulated	execution	1
gemmini_loop_ws	CISC	6
gemmini_loop_ws_spad	CISC	3
gemmini_loop_conv_ws	CISC	7

Source: Adapted from ucb-bar (2019)

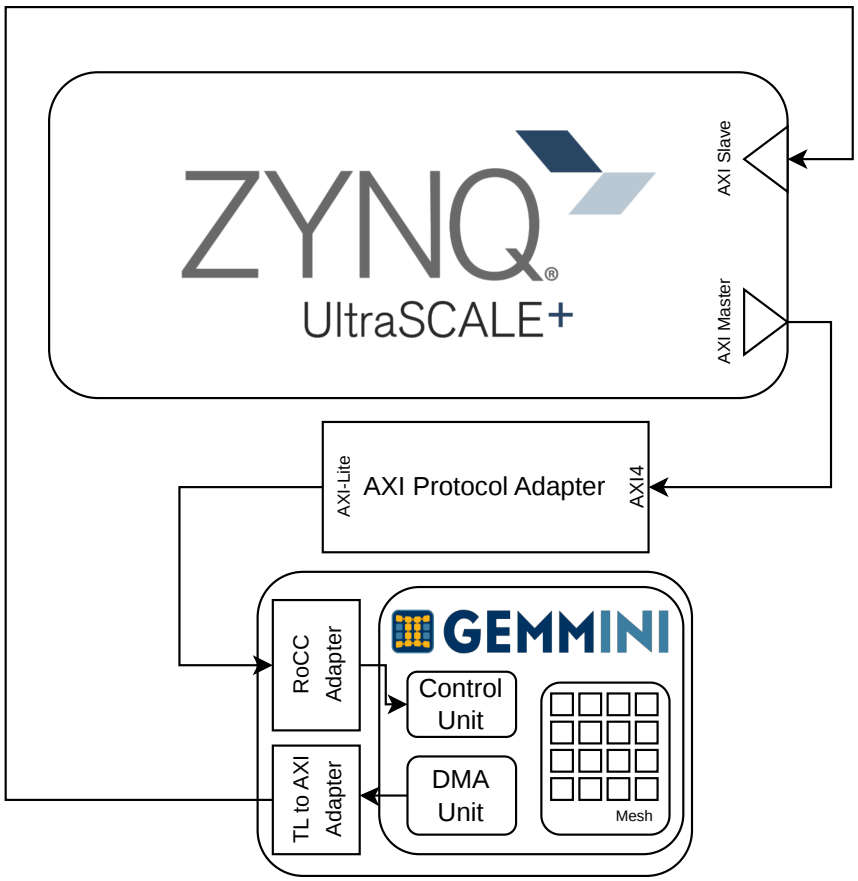
4.2.1.3 The PTW Interface

Lastly, we can mostly ignore the PTW interface. It is only required while executing code with paging; thus, tying it down to ground is enough for the required adaptation. If Linux execution was desired for this, we could simply use memory maps to mark memory as *unswappable* and use physical addresses in the instructions.

4.2.2 Software Adaptations

Programming Gemmini involves embedding custom instructions in the program to be accelerated. This is done through a series of C macros which, ultimately, call an *asm volatile* directive. To work with the proposed adaptations, instead of embedding specialized instructions, we substituted these instructions with a series of writes and reads to the address region specified in the subsection 4.2.1.1. This works, but would normally impose a heavy bottleneck: most of Gemmini's instructions are very simple operations working on small sets of data. Fortunately, the API has a series of CISC operations which execute full convolutions or matrix multiplications, with data movement, pooling and activation implicit, greatly mitigating the bottleneck imposed by the adaptations. Table 4.3 shows Gemmini's regular instruction set architecture. For any given operation, several RISC operations are necessary to implement what a single CISC operation will do. While the CISC operations take several instruction dispatches to execute, they are still faster than the RISC operations.

Figure 4.4 – SoC implemented for Kria testing



Source: The Author

4.2.3 Validation

To validate the implementation, Gemmini was implemented in the Xilinx Kria KV260 Vision AI Starter Kit. Figure 4.4 shows the SoC implemented. The command issuing is done through an AXI-Lite interface and the DMA unit is directly connected to the MPSoC. For the evaluation, an accelerator with a 16x16 WS systolic array was implemented, with a 128-bit memory interface. Table 4.4 Shows the resource utilization in the FPGA used to test the new hardware's functionality. When using these modifications to test approximate circuits, some freedom on LUT usage is needed for implementation to work: normally, multiplication is mapped in the FPGA's Digital Signal Processors (DSP), but, since approximate multiplication cannot use DSPs in the same way, it ends up in LUTs, making the approximate accelerator take more FPGA space.

The design was implemented with the accelerator running at 40MHz. The critical

Table 4.4 – Kria Utilization

	LUTs	FFs	DSPs	BRAM	CLBs
Utilization	88388	49952	203	128	14276
Percentage	75.5%	21.3%	16.3%	88.9%	97.51

Source: The Author

path is in the rescaling units (the circuit responsible for requantizing the accumulated values resulting from the matrix multiplication back to an int8 value), due to it executing an integer to float conversion, a float32 product and a float to integer conversion. Berkeley implements this circuit as a multi-cycle path, but while not constrained, it does not interfere in compute correctness due to its four cycle delay (per the accelerator's default implementation). Despite this, a smaller frequency was chosen to allow LUT combining, leading to lower design congestion and, thus, lower bitstream implementation times.

Testing the Resnet50 compiled with the proposed adaptations to the software stack and loading it to the FPGA, the new implementation takes about two minutes, vastly outperforming Chipyard's prototyping flow for this task. The design passes the compliance tests, thus safely allowing evaluation with the alternate prototyping we produced. To simplify testing for several circuits, a set of TCL scripts for automatic sourcing, synthesis and implementation were developed, allowing bitstream generation in bulk.

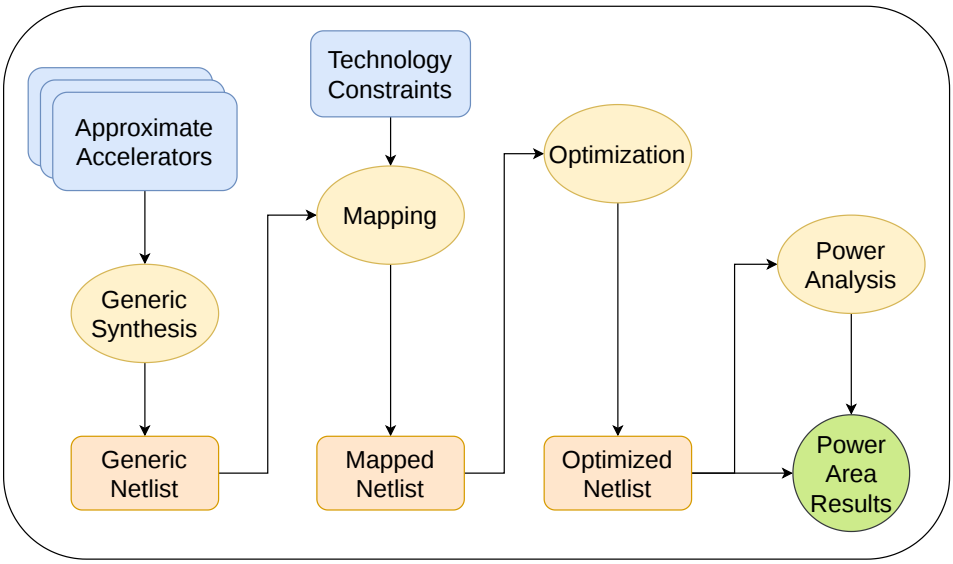
To infer in bulk, the Vitis IDE can be used to load the ELFs, extract the resulting accuracies and store them for processing later in the flow's pipeline. The neural networks themselves can be converted from an ONNX to a C program using the ONNXRuntime Python library as a parser for the files and writing the resulting C code for network topology and weights.

4.3 VLSI Flow

The VLSI flow consists of synthesizing the accelerator's systolic array to some standard cell library to estimate area and power characteristics for the construct. It takes as inputs the approximate generators and a technology file and constraint set for synthesis, resulting in power and area estimates. To achieve this, commercial tools were employed to implement the systolic arrays of the approximate accelerators generated through the Generation Flow.

Figure 4.5 illustrates the flow for area and power analysis of this step. It shows the steps used for most logical synthesis tools, which are used in this context to extract

Figure 4.5 – VLSI Flow illustrated



Source: The Author

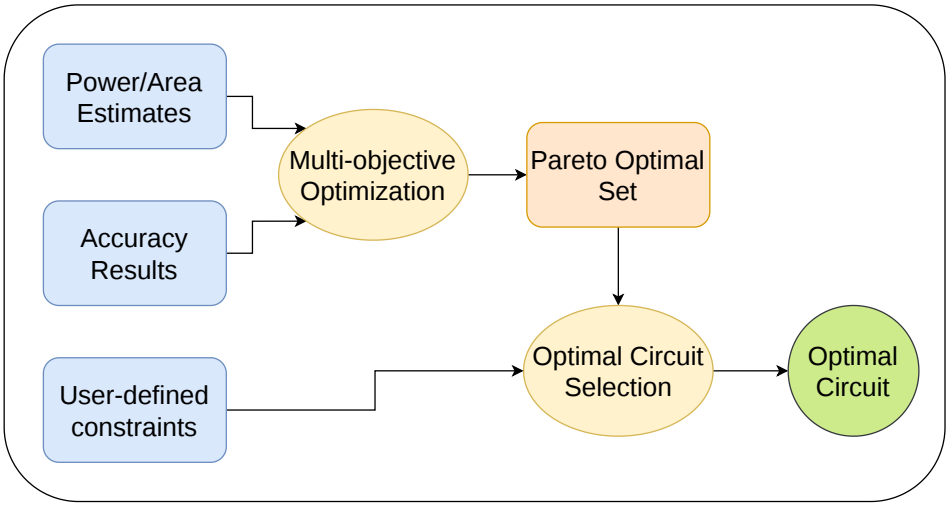
power and area estimates for the standard cell flow. It firstly synthesizes the systolic arrays with approximate arithmetic circuits to a generic netlist, maps it to a target technology given as input to the framework, applies optimizations (such as constant substitution and propagation) and extracts power data using the power analysis tool (such as Synopsys PrimePower or Cadence Voltus).

Only the arithmetic circuit's power and area is insufficient to determine its impact on the power and area utilization of the accelerator. Due to sequential elements, the clock tree and optimization during logical synthesis, power and area results do not directly correlate to the arithmetic circuits power and area. Thus, by using the whole systolic array, we can more accurately predict the impact of approximation in the MxM unit.

Firstly, some logical synthesis tool has to be chosen. Between the options, we can use either Cadence Genus, Synopsys Design Compiler NXT or Synopsys Fusion Compiler. Both Design Compiler and Fusion Compiler use physical guidance, which requires a floorplan to execute synthesis. This significantly complicates the flow, as, due to differences in area utilization between the designs, a separate floorplan needs to be generated for each one; thus, since comparative analysis is more useful for the desired DSE and to keep tool autonomy, Cadence Genus was chosen.

Genus's VLSI flow is straightforward: the tool elaborates the input RTL to a template in memory, synthesizes these templates to a generic netlist where it executes logic reordering and optimization, maps to the gates provided by the target technology and then executes DRC fixing through buffer insertion and cell sizing, while trying to match

Figure 4.6 – Analysis Flow illustrated



Source: The Author

the design constraints provided by the user. This process can mostly be automatized, demanding very little user interaction for the quality of results this analysis requires. Since synthesizing several designs is parallelizable as compute resources and tool licenses are available, we can develop a script which takes as input the set of configurations to synthesize and the number of instances to be launched and executes the tool for each design.

Some target technology and standard cell library have to be provided to the VLSI Flow, with the recommendation of using a target with an array of cells with good coverage for common functions for arithmetic circuits, such as adders and compressors; using a simpler target can make certain circuit architectures heavily underperform compared to peers which don't depend on these cells, producing an inaccurate analysis.

4.4 Analysis Flow

Lastly, using the data resulting from testing, VLSI synthesis and the user-provided constraints, the tool finds the optimal circuit for the target application. Analysis of optimality consists of taking the Pareto-optimal set of circuits, using a mix of the target networks accuracies and a mix of power and area results. To achieve this, we use some heuristic to qualify the accuracy and PPA results from the Test and VLSI flows.

Figure 4.6 shows the analysis flow. Firstly, a multi-objective optimization is applied for both accuracy against power and accuracy against power, resulting in Pareto-optimal

sets for the tested approximate circuit library. Then, the user-provided constraints are applied to take the optimal circuit for the target application.

For the implementation, the flow uses scripts to parse the results from both testing and implementation to result into a spreadsheet. This spreadsheet can then be processed through a Python script, which requires functions to evaluate the results. For example, to evaluate the overall performance of accuracy on all networks, a harmonic average for accuracy can be extracted to be used as a variable to be maximized in the analysis.

To restrain the optimal solution, a pair of constraints has to be given: one fixed constraint and a variable to be minimized or maximized. For example, one could ask for at most a loss of 2% accuracy for maximum power or area reduction, or the minimum amount of accuracy loss for 20% reduction in area. Finally, the tool can find the optimal circuit for the constraints.

5 METHODOLOGY

To show the effectiveness of our framework, we test the EvoApproxLib-lite(MRAZEK et al., 2017) for for two analysis scenarios, minimizing either power or area.

5.1 Approximate Circuits

EvoApproxLib-lite provides thirteen 8-bit signed multipliers, with varied error characteristics. Table 5.1 show the thirteen circuits in the library and their relevant error metrics, including:

- Mean Absolute Error (MAE)
- Mean Relative Error (MRE)
- Worst-Case Absolute Error (WCE)
- Error Probability (EP)

By using varied error metrics, we can cover a range of potential error characteristics. These do not strongly correlate to accuracy, but using a range of approximation levels might provide insights on how error impacts accuracy.

5.2 Networks

We test two network architectures for three datasets: The VGG16 for CIFAR10 and the Resnet18 for CIFAR100 and TinyImagenet, all quantized to 8-bit integer. Table 5.2 shows the baseline accuracy for the networks tested. We expect to see a difference in resilience both between the network topologies tested and between the datasets presented. All networks are tested with a thousand images from the validation images of the respective dataset.

5.2.1 Network Architectures

The VGG16 is a deep convolutional neural network with thirteen convolutional layers and three fully-connected layers. It was introduced in the ILSVRC 2014 competition, winning 2nd place overall in Imagenet1K for classification(SIMONYAN; ZISSERMAN, 2015).

Table 5.1 – Relevant error metrics for the approximate circuits library

Circuit	MAE	WCE	MSE	EP
1KV6	0%	0%	0%	0%
1KV8	0.0018%	0.0076%	0.28%	50%
1KV9	0.0064%	0.026%	0.9%	68.75%
1KVA	0.019%	0.075%	2.53%	74.8%
1KVM	0.049%	0.2%	2.4%	49.8%
1KVP	0.051%	0.21%	2.73%	74.8%
1KVQ	0.06%	0.25%	3.64%	84.18%
1KX5	0.15%	0.69%	8.93%	91.89%
1KXF	0.34%	1.37%	15.72%	87.16%
1L2J	0.081%	0.39%	4.41%	74.61%
1L2L	0.23%	1.16%	12.26%	93.16%
1L2N	0.52%	2.66%	27.44%	97.75%
1L12	3.08%	12.3%	135.77%	98.05%

Source: Adapted from Mrazek et al. (2017)

The Resnet18 is a residual network which was initially used for inference on the Imagenet1K dataset. It utilizes residual connections, which are layers used to allow for deeper neural networks when compared to those such as the VGG16(HE et al., 2015).

5.2.2 Datasets

CIFAR10 is a dataset of 32x32x3 images with 10 target classes. The baseline accuracy of the network tested was 88.7%, using Top 1 accuracy.

The CIFAR100 uses the same shape of the input image as CIFAR10, but has 100 target classes. Due to this added complexity, the Resnet18 DNN was used instead. The baseline accuracy of the network was 74%.

TinyImagenet is a subset of the Imagenet dataset with 64x64x3 images and 200 target classes. To test TinyImagenet, Resnet18 was utilized. Due to the added complexity of the dataset, instead of using the Top 1 accuracy of 27.9%, the Top 5 accuracy of 70.4% was used as a baseline.

Table 5.2 – Accuracies for the tested networks

Network	Dataset	Accuracy	Type
VGG16	CIFAR10	88.7%	Top 1
Resnet18	CIFAR100	74%	Top 1
Resnet18	TinyImagenet	70.4%	Top 5

Source: The Author

5.3 Technology for mapping

The technology chosen for this analysis was the TSMC65lp 9-track process. It provides a huge array of standard cells to be used for mapping, ensuring the results properly show the comparison of the circuits in real systems. We synthesize the system at 300MHz with 10% clock skew.

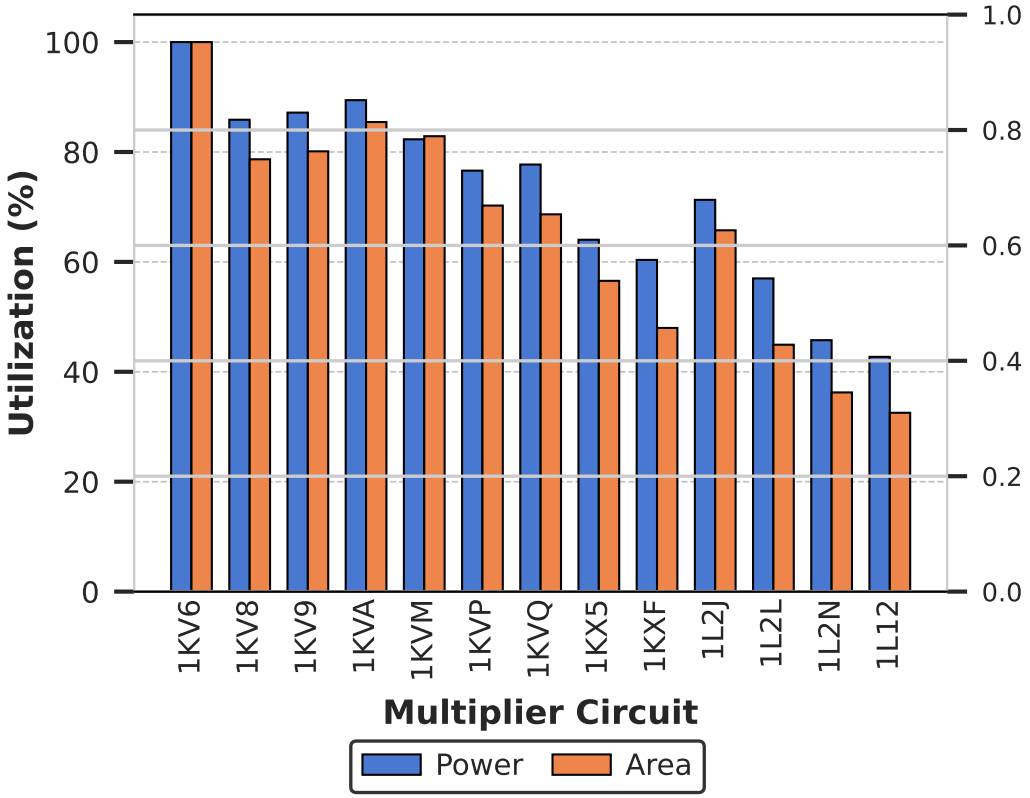
5.4 Evaluated Metrics

We test our framework with two targets for optimal circuit: the circuit which loses the least accuracy on any network for a reduction of 20% in area, and the circuit with the greatest reduction in power for at most 2% accuracy loss across the board. For our evaluation functions, we maximize the harmonic average of each network accuracy normalized to the default accuracy, and minimize area and power for the two analysis, respectively.

6 RESULTS

Figure 6.1 shows the results for power and area for the systolic arrays implemented with the approximate circuits. The X-axis shows the circuit from the EvoApproxLib-lite, with the 1KV6 being the baseline accurate circuit, and the Y-axis shows power and area utilization when compared to the accurate array. This graphic shows us that, even for a small degree of approximation, significant gains in both power and area can be achieved. The operator with the least approximation reduces the power utilization of the array by nearly 15%, with other alternatives with more aggressive approximation showing even better results.

Figure 6.1 – Power and Area results for the circuits

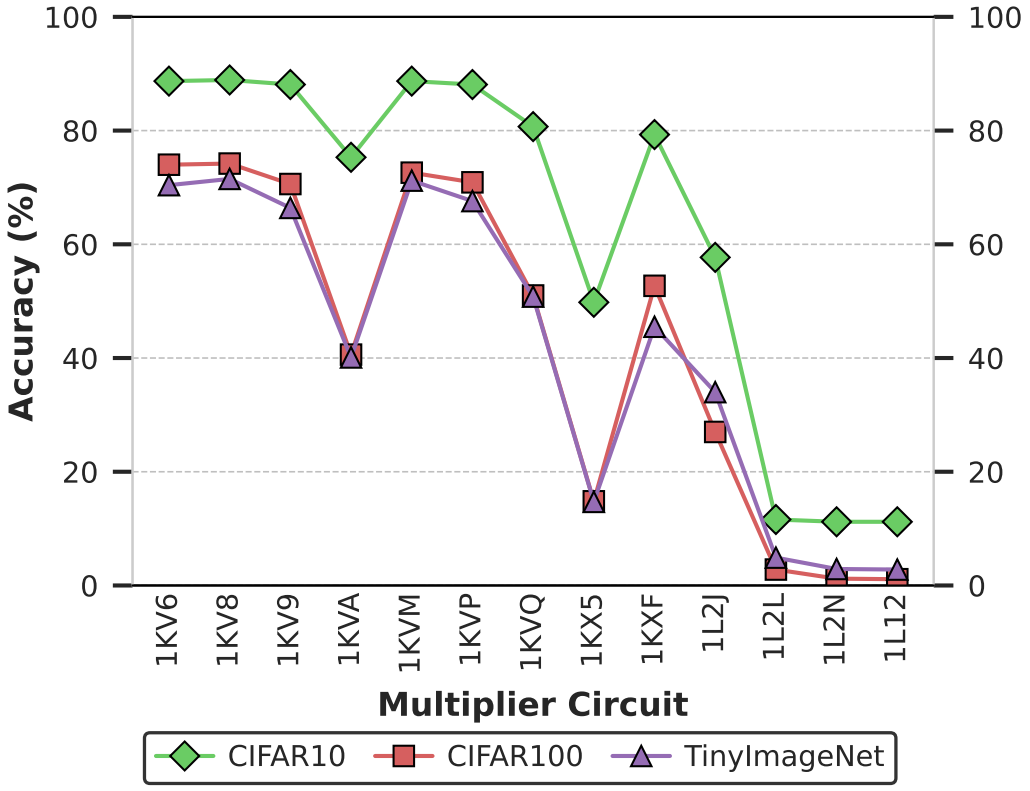


Source: The Author

Figure 6.2 shows the resulting accuracy of inference with the approximate circuits on the selected networks. The x-axis shows the circuits being tested, with the lines showing the accuracy of the three tested neural networks. Some degree of resiliency to error can be seen in all networks, with several circuits showing degradation of less than 2% across the board, but several circuits show their non-viability due to catastrophically losing inference capability or having alternatives with similar accuracy with less area and power usage.

From figure 6.2 we can also conclude that different networks have different degrees

Figure 6.2 – Accuracy results for the circuits



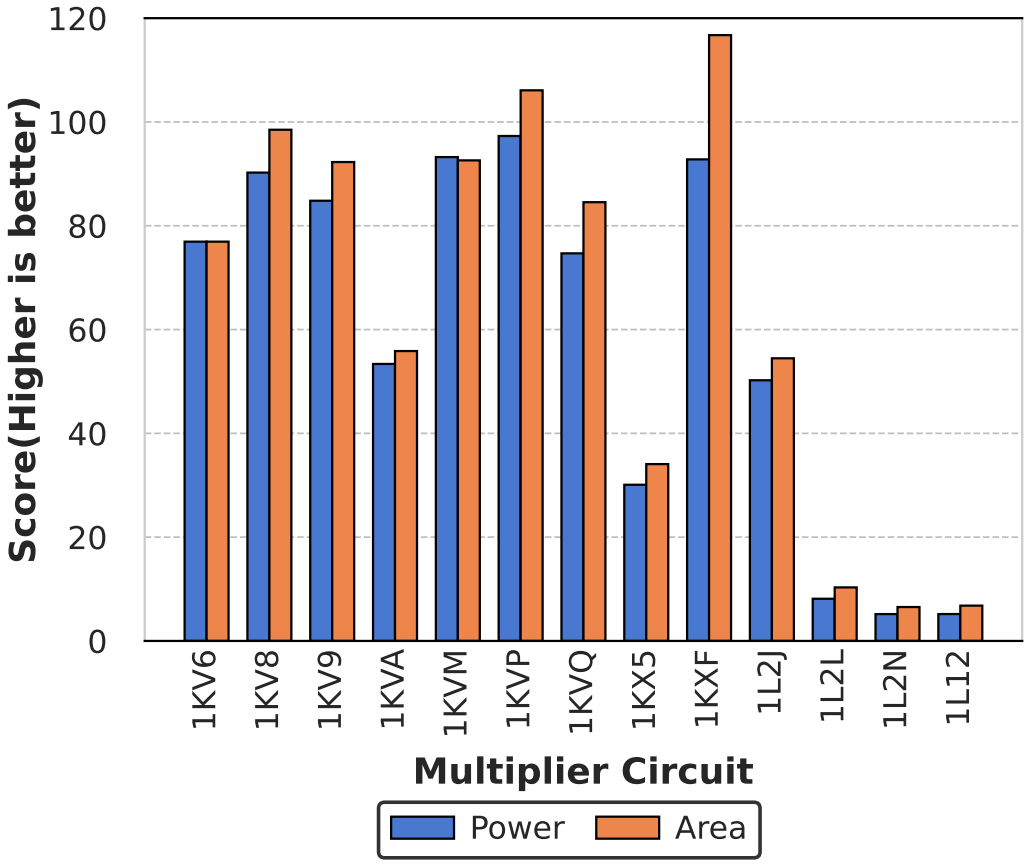
Source: The Author

of resiliency to approximation, with the VGG showing itself as more capable of tolerating errors. This can be due to either dataset complexity (with more complex datasets being more sensitive), how tight of a fit the network was during training or even how network topologies interact with approximation.

In particular, the Resnet18-based networks might be more susceptible to approximation-induced errors due to their residual add operations. Since the residue operation consists of a sum of two matrices, and approximation can introduce implicit biasing due to multiplier error, the smaller error resilience when compared to the purely convolutional VGG16 could be explained as a problem with varying scales during the successive operations in the network. As a final layer, both topologies apply the softmax function, which normalizes the final feature map of the network. Since the VGG "uniformly" deforms data, errors are masked in the softmax due to, when relative to one another, being symmetrical. On the other hand, since Resnet18 uses data from intermediate parts of the network, the errors introduced could be amplified by the intermediate residues, making it less resilient.

While this effect has not been deeply explored in the context of approximate computing, network layer resilience has been explored for fault-tolerant applications. (IBRAHIM et al., 2020) analyzes the impact of fault-injection in deep residual networks

Figure 6.3 – Score for each circuit



Source: The Author

and reaches the conclusion that residue layers and convolutional layers which antecede residue layers are more sensible to errors.

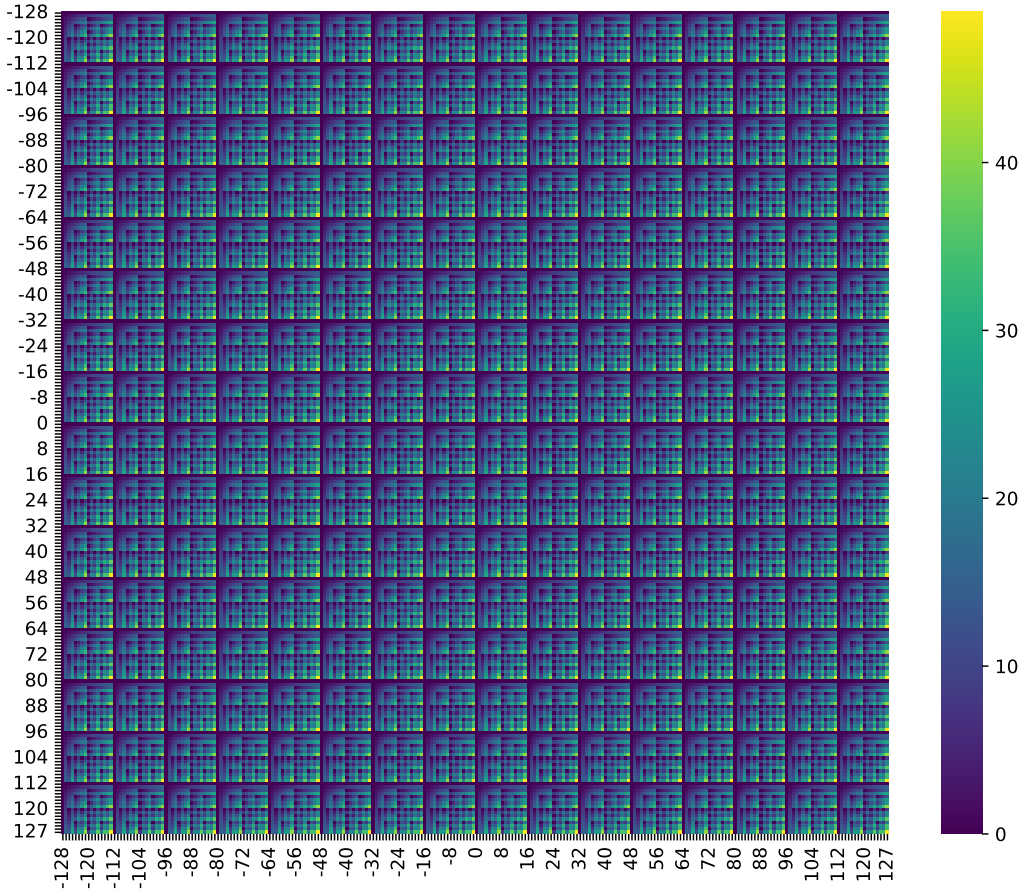
To better visualize the performance of the circuits in term of power, area and accuracy, we can compare the circuits using a score function defined as:

$$SCORE = \frac{H(acc_1, acc_2, ..., acc_n)}{utilization}$$

where H is the harmonic average function, acc refers to the accuracy on each of the networks and utilization refers to how much power or area the array utilizes when compared to the accurate baseline. Figure 6.3 shows the score for each circuit. Six circuits achieving a better score on either power and area when compared to the baseline accurate circuit. This scoring system makes the gross assumption that power and area are equally important to accuracy, but helps us compare the circuits

The circuits in EvoApproxLib-lite are all Pareto-optimal for some error metric. Despite this, multiplers such as the 1KVA, 1KX5 show very poor performance when used for deep learning. The 1KVA candidate has an area and power reduction marginally

Figure 6.4 – Error heatmap for the 1KVA circuit

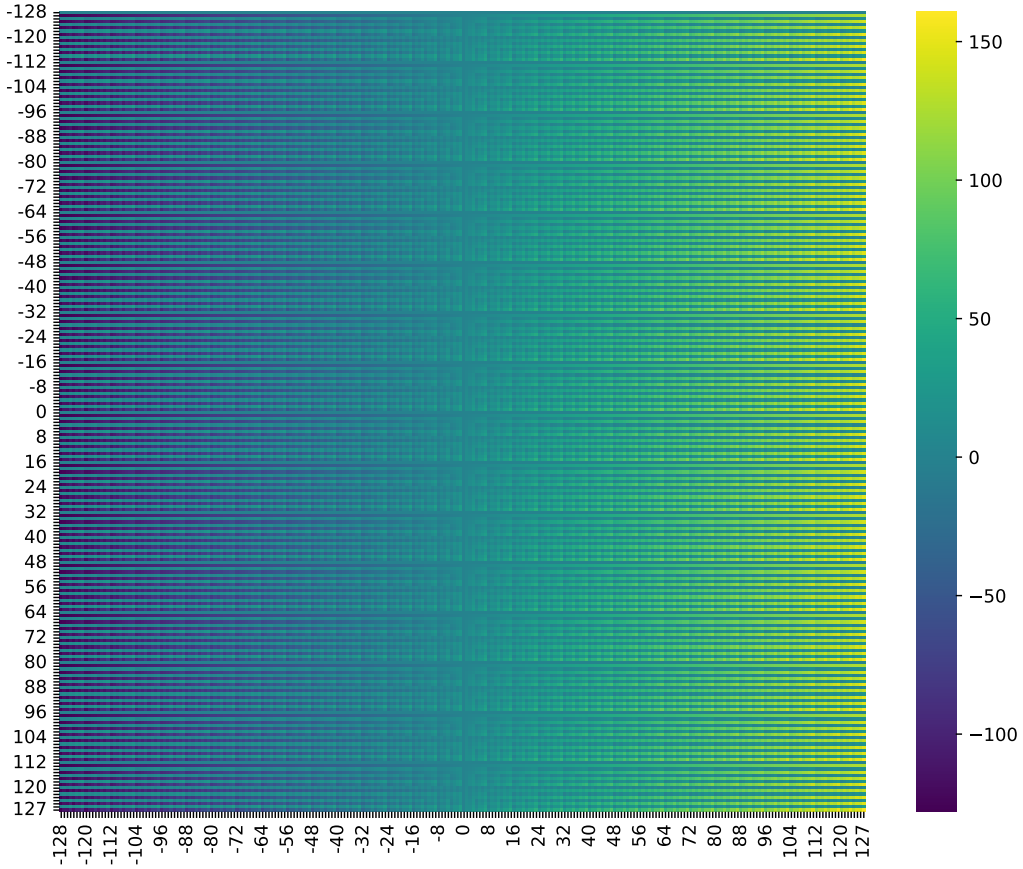


Source: The Author

smaller than the 1KV8 and 1KV9 circuits, with approximately 15% less area and 11% less power, but shows very poor performance on network accuracy, with a 13% loss on the VGG16-based CIFAR10 and a 30% accuracy loss on both the Resnet18-based benchmarks. Similarly, the 1KX5 circuit also has very poor accuracy despite reasonable power and area reductions.

The 1KVP and 1KXF circuits show the best tradeoff for power and area respectively. 1KVP has a accuracy loss of at most 3% (and having a negligible accuracy loss on the purely convolutional VGG), while giving a reduction of over 20% in power utilization, making it a prime example of how approximation can work well on neural networks. Surprisingly, the 1KXF candidate has a power and area reduction so significant that it offsets its accuracy loss of nearly 25% on the TinyImagenet dataset. In contrast, the 1KX5 and 1KXF multipliers show an interesting case, where 1KXF has worse error metrics for MAE, MSE and WCE, a greater area and power reduction but still shows better accuracy, highlighting how the evaluation proposed by our framework is necessary to properly evaluate impact of approximation on deep learning.

Figure 6.5 – Error heatmap for the 1KVQ circuit

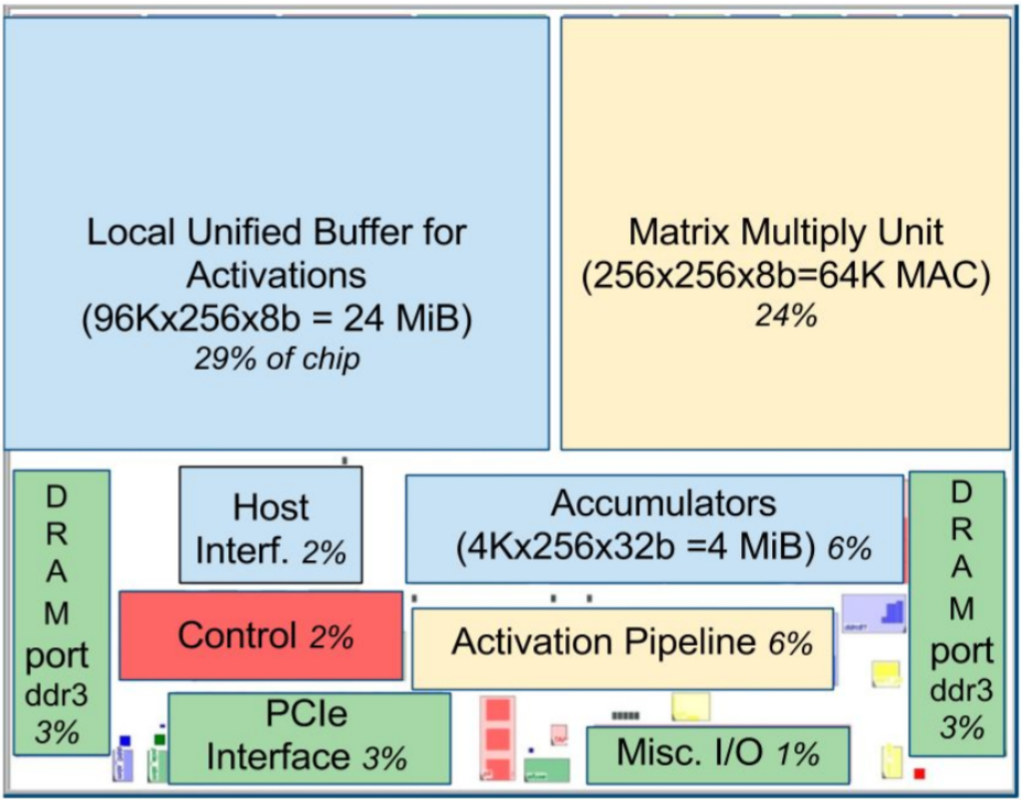


Source: The Author

From the data on figure 6.2 and in table 5.1, we can visualize the nonlinearity of error and approximation. For a case study, we will take the 1KVA and 1KVQ circuits. For all the normally evaluated error metrics, the 1KVA circuit should better, but in practice it performs worse. This sounds counterintuitive, but we can theorize on the reason by observing the arithmetic circuit's error heatmaps shown in Figures 6.4 and 6.5. These represent the error defined as the difference between the accurate result and the multiplier's result for each combination of two 8-bit operands. The x-axis represents the a input for the multiplier and the y-axis represents the b input, with the color intensity showing the value of the numeric error in the multiplication of both the operands. The 1KVQ multiplier's error across operations is more uniform, which may contribute to its better performance. This leads us to conclude that the error metrics typically used to characterize approximate multipliers might be insufficient to properly model their impact on neural networks, which could be the object of future work.

We can also point out that the 1KVQ multiplier does not obey the property of commutation, which means a product $a \times b$ does not obligatorily equal to the product $b \times a$. This is not surprising: these multipliers are automatically generated through

Figure 6.6 – TPU Floorplan



Source:(JOUPPI et al., 2017a)

Table 6.1 – Pareto optimal set for power vs accuracy

Circuit	Power(% of accurate)	Area(% of accurate)	Accuracy CIFAR10 Top-1	Accuracy CIFAR100 Top-1	Accuracy TinyImagenet Top-5
1KV8	85.87%	78.67%	88.9%	74.2%	71.5%
1KVM	82.31%	82.87%	88.7%	72.6%	71.2%
1KVP	76.61%	70.25%	88.1%	70.9%	67.6%
1KXF	47.98%	60.36%	79.3%	52.7%	45.5%
1L2L	44.93%	57.00%	11.6%	2.8%	4.9%
1L2N	36.24%	45.75%	11.2%	1.2%	2.9%
1L12	32.54%	42.72%	11.2%	1.1%	2.8%
1KV6(Accurate)	100%	100%	88.7%	74%	70.4%

Source: The Author

CGP. Thus, no guarantee can be given for their mathematical properties. Their lack of commutation implies that accuracy might depend on dataflow for a multiplier such as this, since some systolic arrays can opt to implement matrix multiplication as a product of transposed matrices due to memory locality.

Considering how different approximate circuits can have varied impacts across the board, we can conclude that software-agnostic arithmetic selection is inherently suboptimal. Thus, *the choice of approximate circuit has to be made aware of the software to be executed, thus demanding hardware-software co-design.*

From the data on both figure 6.1 and 6.2, we can extract the Pareto-optimal set for power or area. Table 6.1 shows the Pareto optimal set for power vs accuracy in bold and the accurate circuit for comparison. The Pareto optimal circuits are those for which,

Table 6.2 – Results for optimal circuit analysis

Area Constraint	Power Constraint	Accuracy Constraint	**Optimal Circuit**
At least 20% less	-	Minimize	**1KV8**
-	Minimize	At most 2% loss	**1KVM**

Source: The Author

Table 6.3 – Area estimates for Google's TPUv1

Circuit	Total Area	MxM Unit Area	Scalar/Vector/Memory/IO Area
Standard Precision	100%	24%	76%
1KV8	94.9%	18.9%	76%
1KVM	95.9%	19.2%	76%

Source: The Author

for any multiplier, no other can achieve the same accuracy for less or equal power or area utilization. This shows several circuits are not viable due to being inefficient in the target benchmarks.

Interestingly, the baseline 1KV6 circuit is not optimal due to the 1KV8 circuit showing greater than default accuracy. This phenomenon has been documented in literature (PAPPALARDO et al., 2026; JAIN et al., 2022), where minor approximation can lead to a marginal increment in accuracy. (JAIN et al., 2022) theorizes this may be due to approximation introducing a form of regularization, similarly to how pruning works.

Overall, we can see approximation is a viable method for power and area reduction of DNN accelerators. Circuits with minimal approximation show negligible degradation for significant power and area reductions, with more aggressive approximation being able to halve power and area while keeping some accuracy for the validation set.

Finally, for the two analysis proposed, the tool can find the optimal circuits for the two test scenarios. Table 6.2 shows the analysis scenarios and the two optimal circuits: for the scenario of at most 2% accuracy loss for the most power reduction, the tool finds the 1KVM circuit, with the greatest loss of 1.4% accuracy in the CIFAR100-trained Resnet18 for a reduction of nearly 18% in power. For the scenario of at least 20% area reduction, the 1KV8 circuit shows non-measurable accuracy loss for 21% less area utilization.

To better visualize the impacts of these two circuits, we can apply them in the context of an industrial accelerator through Google's TPUv1(JOUPPI et al., 2017b). Figure 6.6 shows the area utilization from the TPUv1 floorplan. The systolic array consists of approximately 24% of the chip's area. By crossing the estimated area from the systolic arrays with the MxM unit implemented in the TPU, we can extract table 6.3. Circa 5% of the whole chip area is saved for negligible accuracy loss, with the region being reduced consisting of high switching rate circuits.

7 CONCLUSION

This work presents a framework for testing of approximate circuits for deep neural networks. By implementing approximate hardware on FPGA, we greatly speed up testing when compared to simulation-based approaches while maintaining ease of modeling unlike emulation-based approaches. We prove our framework through the testing of the thirteen approximate circuits of the EvoApproxLib-lite in neural networks trained for the CIFAR10, CIFAR100 and TinyImagenet datasets.

We also analyze the impact of approximation on inference, reaching the conclusions that different network topologies have different resilience characteristics when inferred with approximation, accuracy loss is not directly linear to error metrics, instead being a harder to model phenomenon which might demand different error metrics than the typically used for approximate arithmetic circuits, and that only circuit power and area reduction is insufficient to correctly predict approximation impact in the implementation of the whole systolic array.

7.1 Future Works

The main challenge of using approximate arithmetic circuits in deep learning is validating approximation impact in complex workloads. Thus, future research should focus on the two main challenges for the area:

- **Modeling of error impact on accuracy:** Since arithmetic circuit error is not strongly correlated to network accuracy, modeling error impact on accuracy and neural network resiliency could make the implementation of approximate circuits more viable for ASICs.
- **Approximate circuits specific for deep learning:** Utilizing better heuristics for hardware exploration may lead to circuits with less accuracy degradation or better power characteristics. Thus, better heuristics to model the impact of approximation on deep learning is one of the main paths forward.

INSTITUTIONAL ACKNOWLEDGEMENTS

This work has been partially funded by the project "Venture Science - Programa de Bolsas de Formação", supported by the Center for Embedded Devices and Research in Digital Agriculture (CEDRA) - SENAI-RS, with financial resources from the PPI IoT/Manufatura 4.0 / PPI HardwareBR of the MCTI, grant number 056/2023, signed with EMBRAPII, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq, FAPERGS Techfuturo - 23/2551-0002199-4, the Office of Naval Research Global (ONR) (N629092212014), and PADIS through HT Micron Semicondutores.

ANNEX A — AXI PROTOCOL

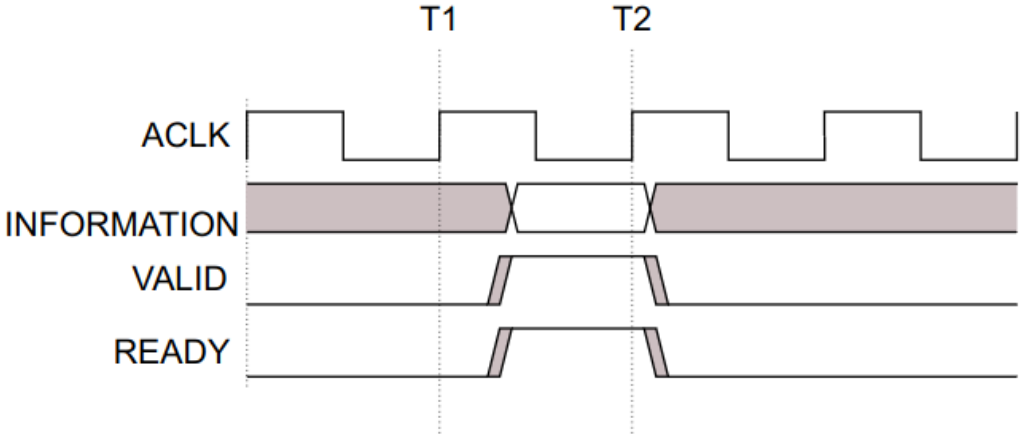
AXI, a part of the AMBA specification, is a digital communication protocol proposed by ARM. Table A.1 shows the signals used by the AXI4 variation. There are five channels:

- **AR Channel:** for master-slave read address communication
- **R Channel:** for slave-master read data communication
- **AW Channel:** for master-slave write address communication
- **W channel:** for master-slave write data communication
- **B Channel:** for slave-master write response communication

The table groups together the AR and AW channels under the umbrella of master→slave address communication, and the R and B channel's common signals under the slave→master group. The AW, W and B channels are used for the write communication, and the AR and R channels together form the channel pair for data reading.

An AXI transaction consists of a request-response pair: The request is issued either through the master-slave write or read channels, with responses in their respective counterparts, being recognized through an AXI handshake operation. Figure A.1 shows an example of an AXI handshake. The transaction is registered when both the VALID and READY signals are asserted on the same cycle.

Figure A.1 – AXI handshake operation



Source: ARM

An AXI transaction can also come as a burst; it can transfer a transaction greater than the bus width through a multi-beat transaction whose length is controlled by the LEN, SIZE and BURST signals, making a single request for an address result in multiple data transfers, saving several transaction cycles.

The additional master→slave signals control specific transaction behaviour:

- ID consists of an unique identifier for each transaction; It it used for out of order transaction ordering
- LEN controls the size of each beat in an AXI transaction
- SIZE controls the length in beats of an AXI transaction
- BURST controls the type of burst, between three types:
    - Fixed burst, where several operations are executed over the same address;
    - Incremental burst, where the address is incremented after every beat;
    - and Wrap-around burst, where the reads iterate circularly over a memory region.
- LOCK is not implemented in AXI4
- CACHE controls if memory can be intermediately cacheable for the transaction
- PROT controls access privilege
- QoS control access priority
- REGION controls SoC-specific routing constraints
- USER signaling has no specific purpose defined, and its implementation is not recommended

For data channel specific signals, besides implementing the ID and USER signals:

- WSTRB, implementing a write mask for unaligned writes (only in W Channel(=)
- LAST, signaling the last beat of a burst transaction

For more information, the specification of the protocol is available online through ARM's official website(ARM Corporation, 2014).

Table A.1 – AXI Signals

Channel group	Signal	Source	Description
Master→ Slave Address			
	AxID	Master	Transaction ID
	AxADDR	Master	Transaction address
	AxLEN	Master	Burst length
	AxSIZE	Master	Burst size
	AxBURST	Master	Burst type
	AxLOCK	Master	Lock type
	AxCACHE	Master	Memory type
	AxPROT	Master	Protection type
	AxQOS	Master	*Quality of Service*
	AxREGION	Master	Region identifier
	AxUSER	Master	User signal
	AxVALID	Master	Address validity
	AxREADY	Slave	Slave readiness
Master→Slave Data			
	WID	Master	Write ID tag
	WDATA	Master	Write data
	WSTRB	Master	Write strobes
	WLAST	Master	Write last
	WUSER	Master	User signal
	WVALID	Master	Data validity
	WREADY	Master	Slave readiness
Slave→Master			
	xID	Slave	Response ID tag
	xRESP	Slave	Response
	xUSER	Slave	Response user
	xVALID	Slave	Response validity
	xREADY	Master	Response readiness
Slave→Master Data			
	RDATA	Slave	Response read data

Source: Adapted from ARM

ANNEX B — TILELINK PROTOCOL

Tilelink is a digital communication protocol proposed by SiFive. It utilizes either two or five channels, with support for deadlock-free communication in SoCs. It comes in three compliance-sets, with the Tilelink Uncached-Lightweight(TL-UL), Tilelink Uncached-Heavyweight(TL-UH) and the Tilelink Cached(TL-C) subsets of the protocol. TL-C is not used in this work and thus, is not discussed in this annex.

Table B.1 show the Tilelink signals implemented in the TL-UL and TL-UH compliance subsets. The type column represents channel directions: C stands for control signal, D for data, V is the valid signal and R is the ready signal for the transaction. The A Channel is a master-slave channel for issuing commands and D is the response channel.

Table B.1 – Tilelink signals

Channel	Signal	Type	Width	Description
A				
	opcode	C	3	Operation code
	param	C	3	Specify caching parameters
	size	C	z	Logarithm of the operation size
	source	C	o	Per link source identifier
	address	C	a	Target byte address of the operation
	mask	D	w	Byte lane select for message
	data	D	$8w$	Data payload
	corrupt	D	1	Data on beat is corrupt
	valid	V	1	Channel data valid
	ready	R	1	Channel data acknowledged
D				
	opcode	C	3	Operation code
	param	C	3	Specify caching parameters
	size	C	z	Logarithm of the operation size
	source	C	o	Per link source identifier
	sink	C	i	Per link slave identifier
	denied	C	1	Slave unable to service req.
	data	D	$8w$	Data payload
	corrupt	D	1	Data on beat is corrupt
	valid	V	1	Channel data valid
	ready	R	1	Channel data acknowledged

Source: Adapted from SiFive

A Tilelink transaction consists of a request-response pair, where the request is issued by the master through the A channel and the response by the slave through the D channel. The transaction type is given by the opcode signal, with the transaction types being described in table B.2.

An example of how a Tilelink transaction progresses is shown in figure B.1. The

Table B.2 – Tilelink transaction types

Message	Opcode	Operation	A	D	Response message	
Get	4	Get	y	.	AccessAckData	
AccessAckData	1	Get or Atomic	.	y		
PutFullData	0	Put	y	.	AccessAck	
PutPartialData	1	Put	y	.	AccessAck	
AccessAck	0	Put	.	y		*From **TL-UL***
ArithmeticData	2	Atomic	y	.	AccessAckData	
LogicalData	3	Atomic	y	.	AccessAckData	
Intent	5	Intent	y	.	HintAck	
HintAck	2	Intent	.	y		

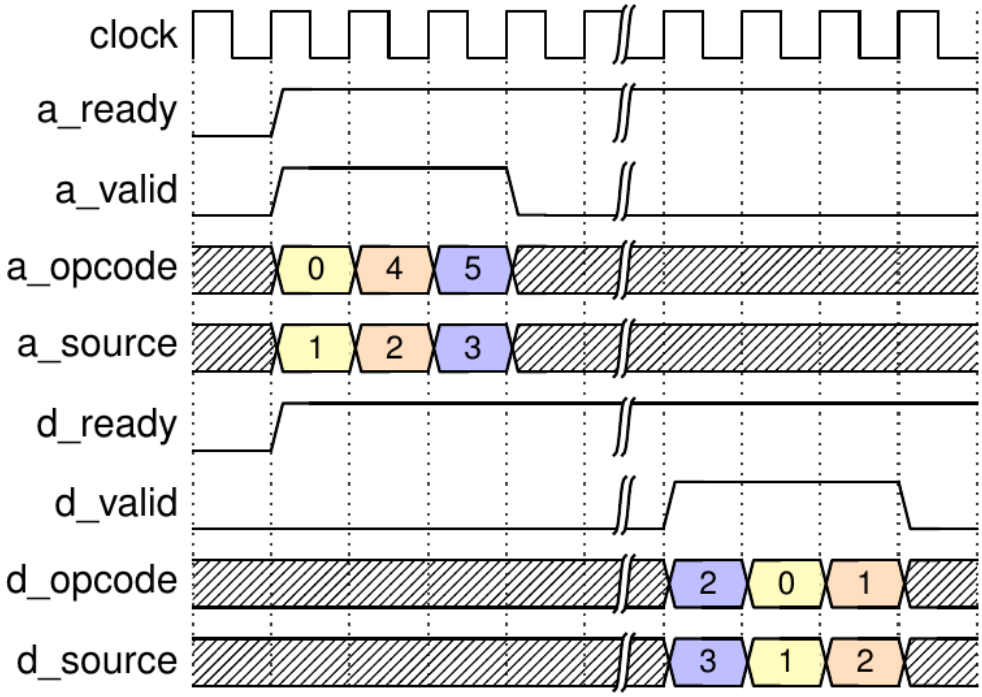
Source: Adapted from SiFive

Tilelink transaction is issued through the A channel with the simultaneous asserting of the valid and ready signals in the channel. An answer with the same source signal is issued through the D channel, with requests not obligatorily being answered in order.

Bursts are controlled through the operation size operand, with the transaction size in bytes being given as 2^n, where N is the unsigned value passed through the source operand. Bursts cannot be aborted in the middle, but beats without data can be transferred to progress the transaction by either unasserting all the mask bits (A channel write only) or asserting the corrupt bit.

For more information, the specification of the protocol is available online through SiFive's official website(SiFive Corporation, 2019).

Figure B.1 – Example of Tilelink transactions



Source: SiFive

ANNEX — REFERENCES

ANSARI, M. S. et al. Improving the accuracy and hardware efficiency of neural networks using approximate multipliers. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 28, n. 2, p. 317–328, 2020.

Apple Corporation. **The future is here: iPhone X**. 2017. Https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/. Accessed: 2026-05-24.

ARM Corporation. **AXI Protocol Specification**. 2014. Https://developer.arm.com/documentation/ihi0022/e/. Accessed: 2026-05-26.

ASANOVIć, K. et al. **The Rocket Chip Generator**. [S.l.], 2016. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>.

BUCCOLINI, A. et al. Towards input-distribution-aware approximate multiplier generation for cnns. In: **2026 Design, Automation & Test in Europe Conference (DATE)**. [S.l.: s.n.], 2026. p. 1–3.

CHEN, Y.-H. et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. **IEEE Journal of Solid-State Circuits**, v. 52, n. 1, p. 127–138, 2017.

ELBTITY, M. E. et al. Aptpu: Approximate computing based tensor processing unit. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 69, n. 12, p. 5135–5146, 2022.

GENC, H. et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In: **Proceedings of the 58th Annual Design Automation Conference (DAC)**. [S.l.: s.n.], 2021.

GERSTMAYR, J.; MANZL, P.; PIEBER, M. Multibody models generated from natural language. **Multibody System Dynamics**, v. 62, p. 249–271, 01 2024.

Google. **Gemini 3 Pro Model Card**. 2025. Https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf. Accessed: 2026-02-27.

HANIF, M. A.; HAFIZ, R.; SHAFIQUE, M. Error resilience analysis for systematically employing approximate computing in convolutional neural networks. In: **2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.: s.n.], 2018. p. 913–916.

HASHEMI, S.; BAHAR, R. I.; REDA, S. Drum: A dynamic range unbiased multiplier for approximate applications. In: **2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2015. p. 418–425.

HE, K. et al. **Deep Residual Learning for Image Recognition**. 2015. Disponível em: <https://arxiv.org/abs/1512.03385>.

IBRAHIM, Y. et al. Soft error resilience of deep residual networks for object recognition. **IEEE Access**, v. 8, p. 19490–19503, 2020.

JAIN, P. et al. Learning to design accurate deep learning accelerators with inaccurate multipliers. In: **2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.: s.n.], 2022. p. 184–189.

JOUPPI, N. P. et al. In-datacenter performance analysis of a tensor processing unit. In: **Proceedings of the 44th Annual International Symposium on Computer Architecture**. New York, NY, USA: Association for Computing Machinery, 2017. (ISCA '17), p. 1–12. ISBN 9781450348928. Disponível em: <https://doi.org/10.1145/3079856.3080246>.

JOUPPI, N. P. et al. In-datacenter performance analysis of a tensor processing unit. In: **2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2017. p. 1–12.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: **Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1**. Red Hook, NY, USA: Curran Associates Inc., 2012. (NIPS'12), p. 1097–1105.

KUNG. Why systolic architectures? **Computer**, v. 15, n. 1, p. 37–46, 1982.

KURTH, A. et al. An open-source platform for high-performance non-coherent on-chip communication. **IEEE Transactions on Computers**, v. 71, n. 8, p. 1794–1809, 2022.

KWAK, M.; KIM, J.; KIM, Y. A comprehensive exploration of approximate dnn models with a novel floating-point simulation framework. **Performance Evaluation**, v. 165, p. 102423, 2024. ISSN 0166-5316. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0166531624000282>.

LI, Z. et al. A survey of convolutional neural networks: Analysis, applications, and prospects. **IEEE Transactions on Neural Networks and Learning Systems**, v. 33, n. 12, p. 6999–7019, 2022.

MAHURIN, E. Qualcomm® hexagon™ npu. In: **HotChips**. [S.l.: s.n.], 2023.

MASSOUD, E. et al. A reusable uvm-systemc verification environment for simulation, hardware emulation, and fpga prototyping: Case studies. In: **2022 International Conference on Microelectronics (ICM)**. [S.l.: s.n.], 2022. p. 38–41.

MRAZEK, V. et al. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017**. [S.l.: s.n.], 2017. p. 258–261.

MRAZEK, V. et al. Design of power-efficient approximate multipliers for approximate artificial neural networks. In: **2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2016. p. 1–7.

MRAZEK, V.; SEKANINA, L.; VASICEK, Z. Libraries of approximate circuits: Automated design and application in cnn accelerators. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, v. 10, n. 4, p. 406–418, 2020.

NORRIE, T. Google's training chips revealed: Tpuv2 and tpuv3. In: **HotChips**. [S.l.: s.n.], 2020.

NVIDIA Corporation. **NVDLA Hardware Manual.** 2018. Https://nvdla.org/hw/contents.html. Accessed: 2026-02-23.

PAPPALARDO, S. et al. Analyzing the impact of functional approximation on the resilience of deep neural networks. **Microprocessors and Microsystems**, v. 122, p. 105259, 2026. ISSN 0141-9331. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0141933126000165>.

QIN, E. et al. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In: **2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2020. p. 58–70.

SAADAT, H.; BOKHARI, H.; PARAMESWARAN, S. Minimally biased multipliers for approximate integer and floating-point multiplication. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 37, n. 11, p. 2623–2635, 2018.

SATO, K. **An in-depth look at Google's first Tensor Processing Unit (TPU)**. 2017. Https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu. Accessed: 2026-02-27.

SiFive Corporation. **SiFive Tilelink documentation**. 2019. Https://www.sifive.com/document-file/tilelink-spec-1.9.3. Accessed: 2026-05-27.

SIMONYAN, K.; ZISSERMAN, A. **Very Deep Convolutional Networks for Large-Scale Image Recognition**. 2015. Disponível em: <https://arxiv.org/abs/1409.1556>.

SPANTIDI, O. et al. Positive/negative approximate multipliers for dnn accelerators. In: **2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)**. [S.l.: s.n.], 2021. p. 1–9.

ucb-bar. **Gemmini**. 2019. Https://github.com/ucb-bar/gemmini. Accessed: 2026-05-24.

VASICEK, Z.; MRAZEK, V.; SEKANINA, L. Automated circuit approximation method driven by data distribution. In: **2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.: s.n.], 2019. p. 96–101.

VAVERKA, F. et al. Tfapprox: Towards a fast emulation of dnn approximate hardware accelerators on gpu. In: **2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.: s.n.], 2020. p. 294–297.

VENKATARAMANI, S. et al. Rapid: Ai accelerator for ultra-low precision training and inference. In: **Proceedings of the 48th Annual International Symposium on Computer Architecture**. IEEE Press, 2021. (ISCA '21), p. 153–166. ISBN 9781450390866. Disponível em: <https://doi.org/10.1109/ISCA52012.2021.00021>.

WESTE, N.; HARRIS, D. **CMOS VLSI Design: A Circuits and Systems Perspective**. 4th. ed. USA: Addison-Wesley Publishing Company, 2010. 157-271 p. ISBN 0321547748.

ZHANG, Y. et al. Estimating vulnerability of all model parameters in dnn with a small number of fault injections. In: **2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.: s.n.], 2022. p. 60–63.

ZHAO, J. et al. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.