12-31-2023

Gen-acceleration: Pioneering work for hardware accelerator generation using large language models

Durga Lakshmi Venkata Deepak Vungarala
New Jersey Institute of Technology, dvungarala7@gmail.com

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

GEN-ACCELERATION: PIONEERING WORK FOR HARDWARE ACCELERATOR GENERATION USING LARGE LANGUAGE MODELS

by
Durga Lakshmi Venkata Deepak Vungarala

Optimizing computational power is critical in the age of data-intensive applications and Artificial Intelligence (AI)/Machine Learning (ML). While facing challenging bottlenecks, conventional Von-Neumann architecture with implementing such huge tasks looks seemingly impossible. Hardware Accelerators are critical in efficiently deploying these technologies and have been vastly explored in edge devices. This study explores a state-of-the-art hardware accelerator; Gemmini is studied; we leveraged the open-sourced tool. Furthermore, we developed a Hardware Accelerator in the study we compared with the Non-Von-Neumann architecture. Gemmini is renowned for efficient matrix multiplication, but configuring it for specific tasks requires manual effort and expertise. We propose implementing it by reducing manual intervention and domain expertise, making it easy to develop and deploy hardware accelerators that are time-consuming and need expertise in the field; by leveraging the Large Language Models (LLMs), they enable data-informed decision-making, enhancing performance. This work introduces an innovative method for hardware accelerator generation by undertaking the Gemmini to generate optimizing hardware accelerators for AI/ML applications and paving the way for automation and customization in the field.

GEN-ACCELERATION: PIONEERING WORK FOR HARDWARE ACCELERATOR GENERATION USING LARGE LANGUAGE MODELS

by
Durga Lakshmi Venkata Deepak Vungarala

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Helen And John C. Hartmann Department Of Electrical And Computer Engineering

December 2023

APPROVAL PAGE

GEN-ACCELERATION: PIONEERING WORK FOR HARDWARE ACCELERATOR GENERATION USING LARGE LANGUAGE MODELS

Durga Lakshmi Venkata Deepak Vungarala

Dr. Shaahin Angizi, Dissertation Advisor — Date
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. Durgamadhab Misra, Committee Member — Date
Professor and Chair of Electrical and Computer Engineering, NJIT

Dr. Cong Wang, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Durga Lakshmi Venkata Deepak Vungarala

Degree: Master of Science

Date: December 2023

Date of Birth:

Place of Birth:

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ 2023
- Bachelor of Technology in Electronics and Communication Engineering,
  DVR & Dr. H.S. MIC College of Technology, Kanchikacherla, AP, 2022

Major: Electrical Engineering

Presentations and Publications:

[1] D. Vungarala, M. Morsali, S. Tabrizchi, A. Roohi, and S. Angizi,"Comparative Study of Low Bit-width DNN Accelerators: Opportunities and Challenges, " *IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Phoenix, Arizona, USA, Aug 6-9, 2023 (Accepted)

[2] C. Wang, D. Vungarala, K. Navarro, N. Adwani and T. Han, "The Pinch Sensor: An Input Device for In-Hand Manipulation with the Index Finger and Thumb," "*2023 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, Seattle, WA, USA, 2023, pp. 822-827, doi: 10.1109/AIM46323.2023.10196268.

You are what you believe in. You become that which you believe you can become.

The Bhagavad-gita

ACKNOWLEDGMENT

I am immensely grateful to my professor, Dr. Shaahin Angizi, Assistant Professor of Electrical and Computer Engineering at the New Jersey Institute of Technology in Newark, for his invaluable guidance and support throughout my research journey. Without his help, I would not have been able to undertake this project.

Many thanks should also go to the defense committee members Dr. Durgamadhab Misra, Professor and Chair of Electrical and Computer Engineering, Dr. Cong Wang, Associate Professor, Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, New Jersey, who generously provided the knowledge and expertise.

This endeavor would not have been possible without the support of my family, especially my mother, Sarala Devi; father, Simha Chalam, for their endless love and support; and brother, Manoj, for making me the person I am today. I also extend my gratitude towards my cousins, who were constantly making me better. I am also grateful to my dear, loving friends who kept supporting me continuously in the ups and downs throughout my life and helped me reach this stage.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Deep Neural Networks (DNNs) have transformed the landscape of artificial intelligence (AI), showcasing impressive capabilities across diverse applications such as machine translation, computer vision, and natural language processing. Coping with the substantial computational needs of DNNs has sparked considerable exploration into hardware acceleration methods. The conventional Von-Neumann architectures, known for their distinct partition of computation and memory, encounter challenges in effectively managing the extensive data movement and memory access demands imposed by DNNs.

1.1 Background

To overcome the limitations of Von-Neumann architectures [1–3], researchers have explored various hardware acceleration techniques, including spatial array accelerators and specialized architectures for DNNs. Spatial array accelerators [3] integrate processing elements directly into memory, reducing data movement overhead and improving memory bandwidth. Specialized DNN architectures, such as GPUs and Tensor Processing Units (TPUs), optimize the hardware for the specific computations involved in DNNs.

1.1.1 Von-Neumann Architecture

The Von-Neumann architecture, also known as the Princeton architecture, is the foundational model for most modern computers. The computation unit (CPU) and the memory unit are separated and communicated by a data bus as shown in the Fig. 1.1. This architecture is the building block for a wide range of computing systems, from personal computers to supercomputers. However, the increasing complexity and

computational demands of modern applications, such as machine learning and others with, particularly those involving large datasets and parallel processing, have exposed limitations in the Von-Neumann architecture.

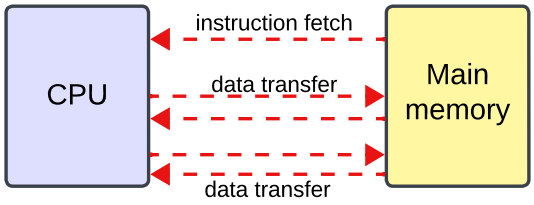

Figure 1.1 Harvard Architecture.

1.1.2 Innovative Architectures Beyond Von-Neumann

In the realm of Non-Von-Neumann architectures, we depart from the conventional model that separates computation and memory, a hallmark of Von-Neumann architectures. This departure involves seamlessly integrating processing elements directly into the memory infrastructure. The primary objective is to curtail data movement overhead and enhance memory bandwidth, thereby ushering in substantial performance enhancements, especially for applications with elevated memory access demands. A noteworthy illustration of this departure is found in Processing-In-Memory (PIM) architectures, where processing elements are interwoven within the very fabric of memory, facilitating in-memory computations.

PIM architectures bring about several advantages in contrast to traditional Von-Neumann counterparts: 1. **Reduced Data Movement:** The integration of processing elements into memory significantly diminishes the necessity for transferring data between distinct components, effectively minimizing data movement overhead.

2. **Enhanced Memory Bandwidth:** PIM architectures grant direct access to memory, resulting in elevated memory bandwidth and reduced latency.

3. **Increased Parallelism:** Accommodating a multitude of processing elements, PIM architectures foster parallel execution of computations, leading to noteworthy performance enhancements.

Beyond PIM, a spectrum of emerging Non-Von-Neumann architectures includes:

Near-Sensor Computing (NSC) Placing processing elements in close proximity to memory reduces data movement as studied at [4], although integration directly into the memory fabric is not a central feature.

Logic-in-Memory (LIM) Here, logic gates seamlessly blend with memory, paving the way for more intricate in-memory computations.

Reconfigurable Processing Units (RPUs) These flexible hardware accelerators can be tailored to execute a diverse array of computations, making them adaptable to the unique demands of Non-Von-Neumann architectures [5].

Collectively, these Non-Von-Neumann architectures offer promising avenues to surmount the constraints posed by traditional Von-Neumann structures, presenting opportunities for heightened performance and energy efficiency across demanding computing applications.

1.2 Survey on the Large Language Model for Hardware Design

ChipGPT is a novel language model that Chang et al. recently introduced [6], [7].Emerging from the intersection of artificial intelligence and hardware design, ChipGPT offers a novel approach to chip development. By leveraging the power of large language models, ChipGPT translates natural language specifications into Verilog code, automating the initial stages of hardware design and significantly accelerating the development process. This groundbreaking technology holds immense

potential to revolutionize the chip industry, democratizing chip design and enabling engineers to focus on higher-level tasks.

In parallel, Shailja et al. [8] put forth a research with a fine-tune pre-trained LLMs on Verilog datasets collected from Github and textbooks. This fine-tuning enables the LLMs to adapt specifically to the intricacies of Verilog code generation. Evaluates framework then analyzes the generated code for both correctness and compliance with Verilog syntax, providing a comprehensive assessment of the LLMs' performance.

1.3 Motivation

The motivation behind this research lies in simplifying the development and deployment of hardware accelerators. We aim to reduce the manual intervention and expertise traditionally required in this domain. To achieve this, we propose leveraging Large Language Models (LLMs), which empower a more accessible and data-informed approach to decision-making, ultimately enhancing performance.

This work is an innovative pioneering work in method for hardware accelerator generation. By harnessing Gemmini's capabilities, we aspire to automate and customize the creation of optimizing hardware accelerators tailored for AI/ML applications. Through this endeavor, we pave the way for a future where the complexities of hardware accelerator development are streamlined, allowing for broader accessibility and efficiency in the field.

1.4 Thesis Organization

This thesis is organized into several chapters, each focusing on specific aspects of the research and implementation. The organization is as follows:

Chapter 1: Introduction In this chapter, we provide an overview of the background, including the Von-Neumann architecture and innovative architectures

beyond it. We also conduct a survey on the utilization of Large Language Models (LLMs) for hardware design, highlighting the motivation behind our research.

Chapter 2: In this chapter we explore various types of hardware accelerators, including Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs), and Non-Von-Neumann Processing-in-Memory. A detailed exploration of Gemmini, a state-of-the-art hardware accelerator, is presented, covering its architecture, software stack, and benefits.

Chapter 3: Here, we conduct a comparative study of low bit-width Deep Neural Network (DNN) accelerators, focusing on Gemmini generated accelerators and a Processing-in-Memory (PIM)-based accelerator. A comprehensive comparative analysis, including experiment setup, results, and security implications, is provided.

Chapter 4: This chapter explores the role of Large Language Models (LLMs) in automating hardware accelerators. We discuss limitations in the utilization of Gemmini and propose a database generation approach for LLM training, outlining key considerations in database construction.

Chapter 5: In this section, we present our contributions, emphasizing how Gemmini an accelerator generation tool is leveraged as a crucial component in automating the hardware accelerator development. We delve into extracting the modules from the SoC in Verilog outputs and its specific focus on input and training of the LLM.

Chapter 6: The thesis concludes with a summary of findings, highlighting the innovative approach for hardware accelerator generation and its implications for AI/ML applications. Future directions and potential areas for improvement are also discussed.

CHAPTER 2

HARDWARE ACCELERATORS

To overcome the Von-Neumann architecture bottlenecks such as memory wall and achieving the high-level parallelism to deploy Machine Learning and Artificial intelligence applications, hardware accelerators [1–3] are developed as a specialized computing device designed to perform tasks more efficiently than general-purpose processors. Unlike general-purpose computers, including machine learning, scientific computing, and data analytics, they are used in many applications with high-level parallelism. In this chapter, the following is discussed.

2.1 Types of Hardware Accelerators

Hardware accelerators can be classified into several categories based on their design and functionality. Some of the most common types of hardware accelerators include:

2.1.1 Field Programmable Gate Arrays

FPGAs are programmable logic devices that can be configured to perform specific tasks. They are highly flexible and can be reprogrammed to perform different tasks.[9, 10]

Design Contemporary FPGAs have ample logic gates and Random Access Memory (RAM) blocks to implement complex digital computations. FPGAs can be used to implement any logical function that an ASIC can perform. The ability to update the functionality after shipping, partial re-configuration of a portion of the design and the low non-recurring engineering costs relative to an ASIC design (notwithstanding the generally higher unit cost), offer advantages for many applications

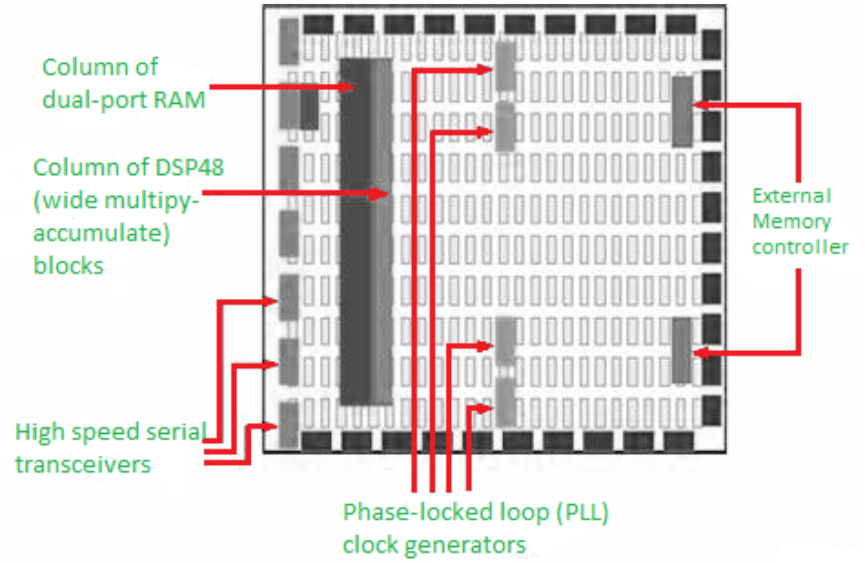

Figure 2.1 Field Programmable Gate Arrays Top level hierarchy.
Source: [11].

2.1.2 Graphics Processing Units (GPUs)

Hardware Perspective: GPUs are specialized processors optimized for parallel execution, featuring thousands of cores, high-bandwidth memory, and dedicated units for graphic tasks. This architecture enables efficient handling of complex calculations and rendering, making them ideal for graphics processing and other computationally intensive workloads.

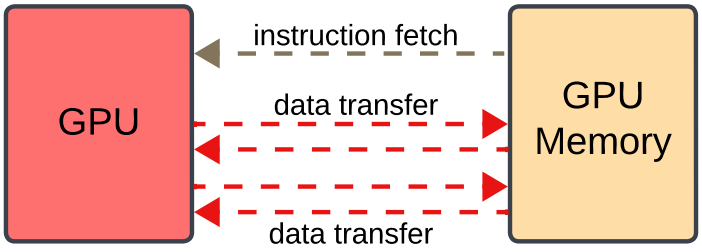

Figure 2.2 GPU block level implementation.

Advantages: GPUs offer significant advantages over CPUs, including faster graphics rendering, improved performance for parallel applications, increased energy

efficiency, and reduced workload on the CPU. These benefits have led to their widespread adoption in diverse fields, including gaming, scientific research, and machine learning.

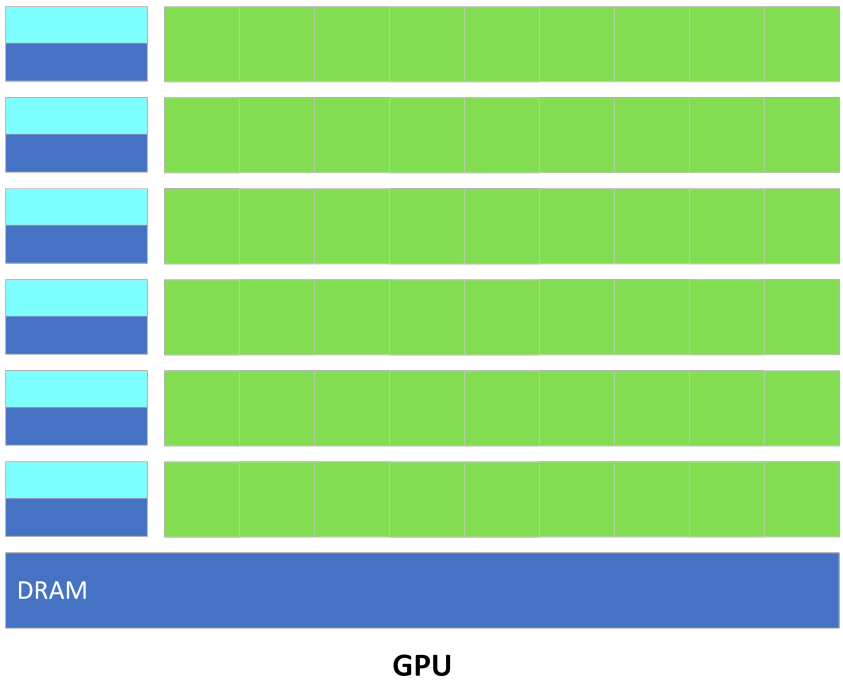

Figure 2.3 Architrcture of Standard Graphic Processing Unit.

The colour convention is that green represents the computational units or core and blue signifies the memory, light blue depicts control units.

2.1.3 Application-Specific Integrated Circuits (ASICs)

Application-Specific Integrated Circuits (ASICs) are specialized electronic chips designed and built for a particular application or task. Unlike general-purpose processors like CPUs, which are designed for a broad range of functionalities, ASICs are meticulously crafted to address specific needs and requirements. This inherent focus translates to significant advantages in terms of efficiency, performance, and cost. In deploying these for their applications few areas can be customized such as the architecture, power consumption, area on chip, performance.

2.2 Gemmini

Gemmini is an hardware accelerator development tool consists of architecture that offers full-system and full-stack integration of the DNN accelerator. Full-system integration considers the impact of the SoC and the operating system running on it. System-level effects that degrade the accelerator's performance must be considered in such integration. Some of these reasons are attributed to the system's memory hierarchy. In this hierarchy, each accelerator possesses its own memory, known as the scratchpad, while the Rocket core or the CPU has its cache. Each core has access to shared memory, and cache hits/misses significantly influence performance.

The integration also considers performance effects caused by Memory Management Units (MMU). Operating system issues, such as context switching between threads and interrupts, are all system-level factors that affect the accelerator's performance.

With Full-stack integration, Gemmini provides a programming stack that allows acceleration programming using higher-level APIs like the ONNX runtime. It also provides access to low-level ISA.

2.2.1 Rocket Chip

Looking into the accelerator requires understanding the SoC or the full system integration provided with the assistance of the Rocket Chip generator. This integration involves a set of parameterized chip-building libraries to generate various SoC variants.

The Gemmini project encompasses two distinct host CPU configurations. One is a low-power, in-order Rocket core, and the other is a high-performance, out-of-order BOOM [12] core.

2.2.2 Architecture

Gemmini accelerators are based on a spatial architecture that utilizes systolic arrays to perform matrix multiplications, which are the core operations of Deep Neural Networks (DNNs). Systolic arrays are a type of parallel processing architecture that efficiently exploits data parallelism and locality to achieve high performance. Gemmini's systolic arrays are designed to be highly configurable, allowing designers to trade off performance, energy efficiency, and area to suit specific application requirements.

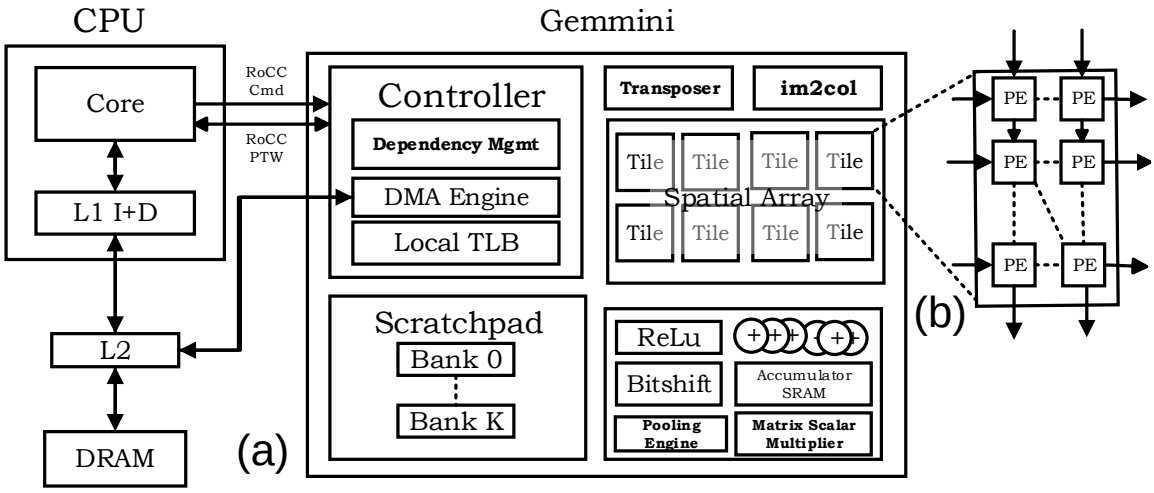

Figure 2.4 Gemmini's template: (a) Overview of Gemmini , (b) Two-level spatial array.
Source: [3].

Systolic Core Gemmini's spatial array design adopts a two-level hierarchy to offer a versatile template for different structures. Initially, the spatial array consists of tiles interconnected through explicit pipeline registers. Each of these individual tiles can be further subdivided into an array of Processing Elements (PEs). PEs within the same tile are connected combinationaly without pipeline registers. Each PE executes a single Multiply-Accumulate (MAC) operation per cycle, utilizing either the weight or the output stationary dataflow.

Data Movement The inputs are supplied to the systolic array, and the generated outputs are stored in scratchpads, which are composed of banked SRAM. A DMA engine facilitates data transfer from the main memory to these scratchpads.

The local memory, or scratchpad, has several rows identical to the number of Processing Elements (PEs) in a row of the systolic array. This means a 4x4 systolic array would have four rows in the scratchpad memory. The accumulator stores the calculation output and is more expansive, as the Multiply-Accumulate (MAC) results would require more bits for representation.

Load Pipeline Gemmini includes three load instructions for moving Inputs, Weights, and biases. These instructions are used to transfer matrices of the size of the systolic array from the main memory to the scratchpad.

Execution The execution of the actual matrix multiplication involves two distinct instructions. The Preload instruction is responsible for loading the stationary data, such as Weights for Weight Stationary data flow and biases from the Output Stationary dataflow. Following the Preload instruction, the Compute instruction is used to perform multiplication on the stationary data that has been loaded.

2.2.3 Gemmini Template Configuration

The GemminiConfig.scala file establishes the default configuration for Gemmini, and the subsequent parameters within this file outline the characteristics of the systolic core to be synthesized.

```
Important parameters:
%Supports both WS and OS dataflows
dataflow: Dataflow.Value = Dataflow.BOTH,

% Creates 16 x 16 Systolic Array
tileRows: Int = 1,
tileColumns: Int = 1,
meshRows: Int = 16,
meshColumns: Int = 16,
```

```
% Specify the data types for different parts of the accelerator
inputType: T,
spatialArrayOutputType: T,
accType: T,

% Define the number of banks in the scratchpad and accumulator
sp_banks: Int = 4,
acc_banks: Int = 2,

% Total memory in terms of KiB
sp_capacity: GemminiMemCapacity = CapacityInKilobytes(256),
acc_capacity: GemminiMemCapacity = CapacityInKilobytes(64),
```

2.2.4 Modules of Gemmini SoC

The Gemmini hardware is divided broadly into three "controllers": one for "execute" instructions, another for "load" instructions, and a third for "store" instructions.

Execute Controller This module executes "execute"-type ISA commands, such as matrix multiplications. It incorporates a systolic array for dot-products and a transposer.

The ExecuteController instantiates the MeshWithDelays module, which includes the SystolicArray (Mesh module) and a Transposer. The MeshWithDelay module processes the three matrices (A, B, and D) one row at a time per cycle and outputs the result $C = A \cdot B + D$ one row at a time per cycle.

Two dataflows are supported:

- Weight Stationary: In weight-stationary mode, the B values are "preloaded" into the systolic array, and A and D values are sequentially fed through.
- Output Stationary: In output-stationary mode, the D values are "preloaded" into the systolic array, and A and B values are fed through.

Store Controller This module's instructions are to move data from Gemmini's private SRAMs into the main memory. In this module, operations such as "max-

polling" instructions are performed because the Gemmini performs pooling when moving the unpooled data from the private SRAMs into the main memory

Load Controller This module assigns instructions to move data from the main memory into Gemmini's private scratchpad or accumulator.

The generated Verilog consists of Gemmini SoC, which we need to understand. The top module consists of 506 submodules that make up the system. In the entire system, we like to focus on the Gemmini module. This module follows a hierarchy, and specifically, we look at the implementation of the execute control since that consists of the Processing Elements and the operation controlling as illustrated in the Fig.2.5.

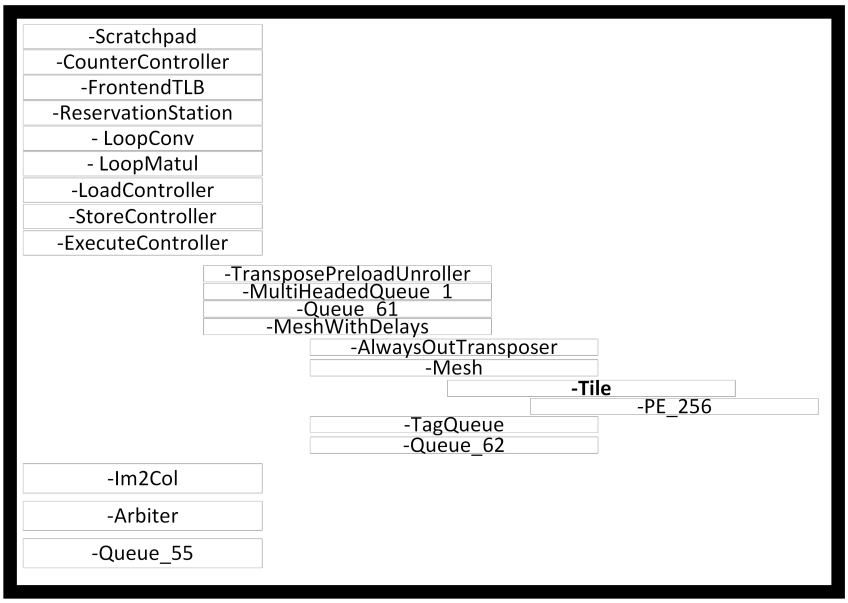

Figure 2.5 Modules in Gemmini.

2.2.5 Software Stack

Gemmini's software stack provides a comprehensive set of tools for programming, simulating, and deploying generated accelerators. The software stack includes a compiler that translates DNN models into Gemmini's ISA, a simulator that allows

developers to test and debug their code before deploying it to hardware, and a runtime system that manages the execution of DNN models on the accelerator.

2.2.6 Advantages

Gemmini offers several benefits over traditional DNN accelerators:

Flexibility: Gemmini's flexible architectural template allows designers to explore a wide range of configurations to find the optimal design for their specific application. Full-stack visibility: Gemmini provides full-stack visibility into the performance and efficiency of DNN accelerators, allowing developers to identify and optimize bottlenecks. Gemmini is an open-source project, which allows researchers and developers to contribute to its development and extend its capabilities.

Gemmini's template is shown in Fig.2.4(a). With a spatial array architecture at the heart of the system, the template uses the 2-D array of Tiles consisting of the 2-D array of Processing Elements (PE) (shown in Fig.2.4(b)) that are responsible to process multiply-accumulate (MAC) operations in parallel. To optimize the area, power, and performance trade-offs, the size of the spatial array can be adjusted. Computationally, the two-level hierarchy of Gemmini supports the implementation of a fully pipelined Tensor Processing Unit-like architecture or an NVDLA-like parallel vector engine inside the tiles as illustrated in Fig.2.6. These are connected with different connectivities between MAC units and do four multiply-accumulates per cycle. The systolic array's inputs and outputs are held in an SRAM scratchpad. A DMA engine facilitates the transfer of data between the main memory and the scratchpad.

2.2.7 Processing-in-Memory

On the other side, digital Processing-in-Memory (PIM) architectures as shown in Fig. 2.7, potentially offering a solution to the memory wall challenge, have been widely

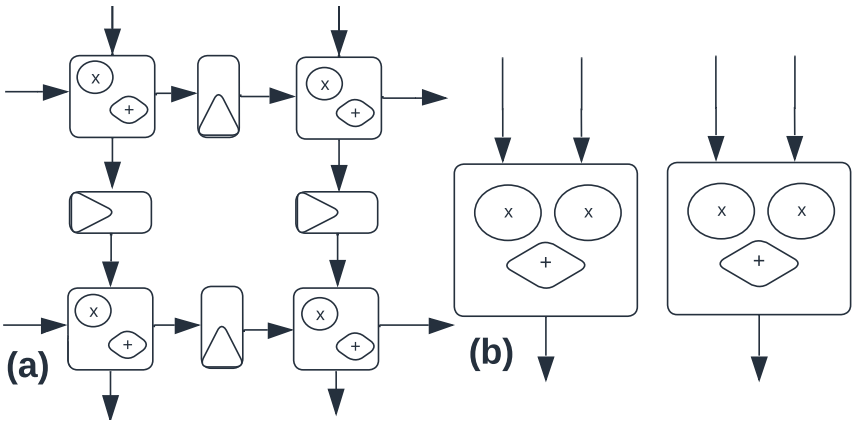

Figure 2.6 Spatial architectures generated by Gemmini: (a) Systolic Spatial Array, (b) Parallel Vector Engines.

explored at the edge devices [13–16]. The key notion behind PIM is to realize certain operations in memory by leveraging the inherent parallel computing mechanism and exploiting large internal memory bandwidth. It could lead to remarkable savings in off-chip data communication energy and latency. A PIM architecture should not only enable DNN acceleration but also other data-intensive applications, including graph processing, data encryption, etc. [17, 18].

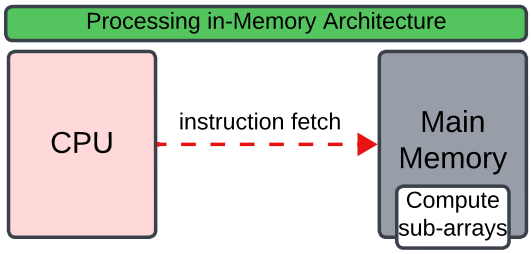

Figure 2.7 Non-Von-Neumann Architecture.

2.3 Non-Von-Neumann architecture

In the DNN acceleration domain, for instance, the Neural Cache [16] presents an 8T transposable SRAM bit-cell and supports digital bit-serial in-cache MAC operation. XNOR-SRAM [19] accelerates ternary-XNOR-and-accumulate operations in binary/ternary DNNs without row-by-row data access. C3SRAM [20] leverages

capacitive-coupling computing to perform XNOR-and-accumulate operations for binary.

CHAPTER 3

RESEARCH AND IMPLEMENTATION

.

3.1 Comparative Study of Low Bit-width DNN Accelerators: Opportunities and Challenges

A comparative study of Gemmini generated ASIC accelerators that follow classic Von-Neumann architecture are developed, another emerging platform [21] is considered along with other state of the art GPU and FPGA in quest to find the robust architectures in metrics such as execution time and power consumption

3.1.1 Gemmini Generated Accelerators

These spatial array architectures are cloned and interfaced with the Gemmini template files from Gemmini's open-sourced GitHub. It is developed in Chipyard Environment using Chisel hardware description language [22]. The generated Gemmini accelerators in this work are developed as systolic array-based matrix multiplication designs optimized for energy efficiency and for achieving high throughput via exploiting data-level parallelism. The accelerators are supported by add-on computation units, including activation functions (ReLU), bitshift, pooling engine, accumulator SRAM, and matrix scalar multiplier to process DNN workloads fully. We develop various spatial array tiles of 16×16, 32×32, and 64×64 to run while inferring to a DNN topology with the CPU: Rocket Custom Coprocessor Interface (rocket) as a RISC-V machine [3].

3.1.2 PIM-based Accelerator

The overall architecture of the under-test PIM platform is shown in Fig. 3.1(a), where each memory bank consists of multiple memory sub-arrays that are repurposed to perform not only memory operations but also in-memory computing based on a set of circuit-level designs. Every sub-array is developed on top of the recently taped-out generic and programmable in-SRAM computing design [21] as shown in Fig. 3.1, which is built on an 8T-SRAM array and can execute all 2- and 3-input Boolean logic operations (OR/NOR, AND/NAND, XOR/XNOR, NOT).

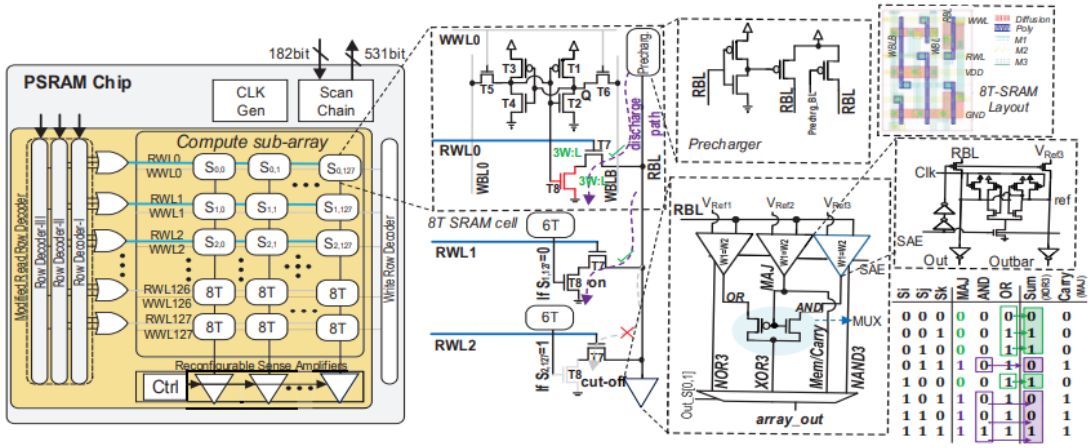

Figure 3.1 PSRAM chip with 8T SRAM cell as the operand memory and the proposed single-cycle logic-SA design.
Source: [23].

The PIM design benefits from the charge-sharing feature of the conventional 8T SRAM cell on the Read Bit-Line (RBL) to achieve different logic operations. As shown in Fig. 3.1(b), by activating three different memory rows via Read Word Line (RWL), e.g., RWL0-RWL2, the RBL remains pre-charged only if all data stored in the selected cells are in '0' state which results in deactivating read access transistors (T8). In contrast, if one or more of the selected SRAM cells store '1', the related T8s activate, and consequently the RBL discharges through them. The discharge voltage value of the RBL is associated with the combination of the values stored in SRAM cells, the activation period of T8s, and the supply voltage. The voltage value

on the RBL is measured through the presented Logic-SA design which is capable of performing logic operations by selecting various voltage references (Fig. 3.1(c)). As shown, the re-configurable Logic-SA consists of three sub-SAs with different voltage references (i.e., V_{Ref1}<V_{Ref2}<V_{Ref3}) to perform distinct logic functions. In this way, after the activation of two or three memory rows, the voltage that remains on the RBL is compared to the reference voltages of sub-SAs. As a result, (N)OR3, (MAJ)MIN, and (N)AND3 logic outputs are produced simultaneously, as shown in Fig. 3.1(c). Moreover, the XOR3's output is the same as the OR3's output when the majority output is '0'. In contrast, when the output of the majority is '1', XOR3's output is equal to AND3. Therefore, to implement XOR3, a 2:1 multiplexer with MAJ output as the selector is utilized. The Boolean logic of such an in-memory XOR3 can be given as $XOR3 = MAJ(S_i, S_j, S_k).AND(S_i, S_j, S_k)+ MIN(S_i, S_j, S_k).OR(S_i, S_j, S_k)$. With these functions, the PIM platform can readily support low-bit-width DNN operations in various configurations. We develop a hardware mapping interface employing C++ to quantize the trained weights and then map them into 128×128 sub-arrays. We use a parallelism degree of 128 sub-arrays. As for data mapping, we follow the bit-serial in-cache acceleration mechanism in [16], which provides high performance and supports various bit-widths for weight and input values.

3.2 Comparative Analysis

3.2.1 Experiment Setup

Platforms: The under-test platforms consist of Gemmini's accelerators with spatial array tiles of 16×16, 32×32, and 64×64 indicated by Ge16, Ge32, and Ge64, respectively, and the generic processing-in-SRAM accelerator in [21]. Moreover, we use the high-end NVIDIA RTX A2000 with 104 third-generation tensor cores that can deliver 63.9 TFLOPS performance and the low-end PYNQ-Z2 board [24]

that uses a Xilinx Zynq-7000 SoC containing an XC7Z020-1CLG400C FPGA for comparison. **Dateset:** For the evaluations, we exploit the CIFAR-10 [25] dataset with 50,000 32×32 color images for training and 10,000 images for validation. **Model:** To have a reasonable comparison among the under-test platforms, we consider a similar Convolutional Neural Network (CNN) topology to run CIFAR-10 consisting of 6 - 3×3 convolutional layers (with 64-64-128-128-256-256 channels) followed by 3 fully-connected layers. The CNN also includes 2 - 2×2 max-pooling layers after the second and fourth layers. **Bit-width configuration:** We consider 2 bit-width configurations of weight and input <W:I> = (<1:1>, and <2:2>) for the evaluation. The 8-bit gradient is applied to all configurations. **Training:** The DoReFa-Net open-source algorithm [26] that utilizes bit-wise convolution of fixed-point integers is used for training. The CIFAR-10 dataset underwent 500 epochs of training and validation. The model was optimized using the Adam optimizer, with various configurations and a learning rate of 0.001.

3.2.2 Results

Execution time: The execution time required to process the CNN model is reported in Fig. 3.2 for various under-test designs. We observe the superior performance of the PIM to process low-bit-width CNN compared to other designs. PIM offers an execution time of 0.002ms and 0.011ms respectively for <1:1>, and <2:2> configurations, whereas the fastest Gemmini's generated accelerator (Ge64) takes ∼6× longer execution time to process the same CNN. PIM design also achieves ∼158× and ∼98× speed up compared with the GPU and FPGA, respectively, on average across two configurations. The key architectural constraint of the PIM accelerator is the number of activated sub-arrays that can work in tandem. The more sub-arrays are involved, the shorter execution time and higher power consumption are anticipated.

As for the generated accelerators, the main constraint is the under-utilization of PE elements that lead to longer execution time.

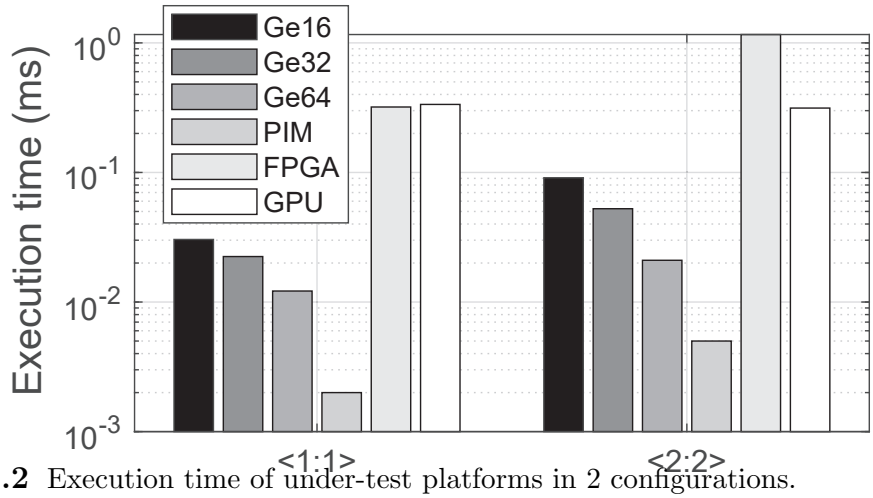

Figure 3.2 Execution time of under-test platforms in 2 configurations.

Table 3.1 Execution Time and Speedup Comparison for Under-Test Platforms

Design	**Execution Time (ms)**	**Speedup vs. PIM**
PIM (<1:1>)	0.002	-
PIM (<2:2>)	0.011	-
Ge64	∼0.012	∼6x slower PIM
GPU	∼0.316	∼158x slower than PIM
FPGA	∼0.204	∼102% slower than PIM

Power Consumption: Our estimated on-chip power consumption results for various platforms are reported in Fig. 3.3. To measure the power consumption of Gemmini's accelerators, the extracted HDL code is synthesized with the Design Compiler in an industry-standard 45nm library. For the FPGA platform, a voltage tester multimeter with a range of 3.7-30V and 0-4A current measurement capability is used as an intermediary device between the power source and the board. For the GPU platform, NVIDIA's system management interface is used. To exclude power costs due to cooling, voltage regulators, etc., the results are then conservatively scaled by 50%. Since no existing framework currently supports fixed-point CNNs on GPUs,

floating-point results are aggressively scaled by $O = W \times I$. Our result underlines the power efficiency of the PIM platform as opposed to other under-test platforms. We observe the PIM consumes 1.41W and 2.12W power to process the whole CNN on <1:1>, and <2:2> configurations, respectively. The PIM platform reduces the power consumption on average by a factor of 2.5 compared to Gemmini's accelerators. Besides, PIM lowers the power consumption respectively by a factor of 18.5× and 14.7× compared to GPU on the two configurations. Compared to the FPGA platform, PIM obtains ∼40% on power efficiency. It is noteworthy that while Ge64 is, in fact, the fastest-generated accelerator, it consumes the highest power consumption.

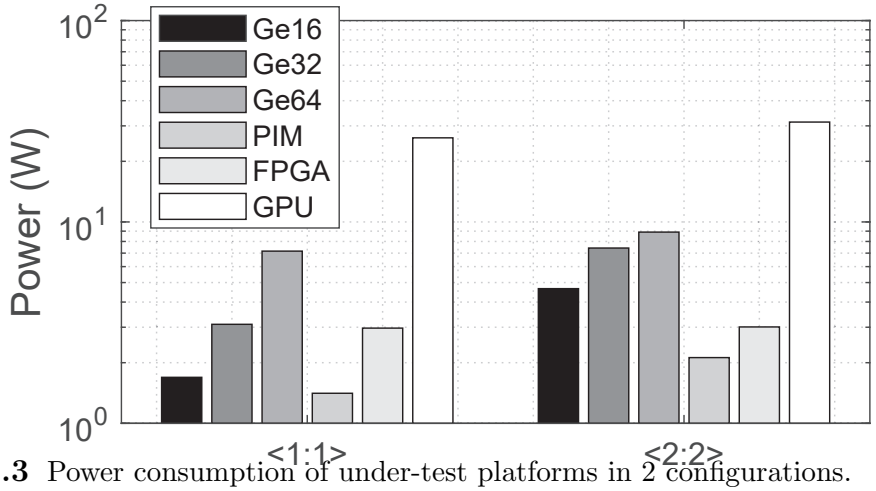

Figure 3.3 Power consumption of under-test platforms in 2 configurations.

Table 3.2 Power Consumption Comparison for Under-Test Platforms

Platform	**Power Consumption (W)**	**Power Efficiency vs. PIM**
PIM (<1:1>)	1.41	-
PIM (<2:2>)	2.12	-
Ge64	∼4.14	∼2.5x higher than PIM
GPU	∼28.62	∼18.5x higher than PIM
FPGA	∼70.6	∼40% higher than PIM

3.2.3 Security Implications

Although third-party accelerators are gaining popularity, their design is independent of the CPU and operating system, which means they may pose security risks. Moreover, since the accelerators depend on commands from a host, the CPU's security measures may be bypassed potentially. There is a large space for threats in ASIC accelerators [27–30]. The virtual memory provided by Gemmini can prevent any unauthorized components from accessing sensitive data. Gemmini, similar to some other accelerators, can be helpless against side-channel attacks. Side-channel attacks exploit the physical characteristics of the device, like power consumption or electromagnetic discharges, to obtain sensitive data. Hardware Trojans are fraudulent modifications to a design that can be included during the design or manufacturing process. These Trojans are capable of disclosing private data, obstructing system operation, or granting illegal access to the system. Since Gemmini is a complicated system with several components and interacts with other hardware and software systems, hardware Trojans may easily infect it. However, the Gemmini architecture also provides several opportunities. Since the Gemmini design is open-source, for instance, the community can audit and verify it, which can assist in identifying and resolving vulnerabilities. The security of the Gemmini architecture may also be increased by integrating it with additional security methods like access control and encryption.

On the other side, the PIM accelerators typically use localized and stationary weights that are subject to various side-channel attacks. It has been recently demonstrated that the attacker can readily extract DNN model information from power trace measurements having no prior knowledge of the network topology [31, 32]. PIM may be vulnerable to memory-based attacks, such as buffer overflow attacks [33], which could compromise the integrity of the data being processed. For example, DRAM-based PIM designs [34] can be vulnerable to row-hammer attacks, where

attackers can repeatedly access specific rows of memory to cause bit flips in adjacent memory cells. Generally, because the computation and memory are integrated, any vulnerability in the memory system could potentially be exploited to attack the computation unit. However, PIMs can also pave the way for new attack mitigation and protection techniques. Although the processor is typically considered the trusted base for encryption, high-assurance computing systems with attested and verified memory logic can rely on it for encryption. In such a design, PIM can be used for encryption without the need to send the data all the way to the processor chip, decrypt it, and then encrypt it again with a new key before writing it back. Instead, encryption can be performed directly on the spot.

CHAPTER 4

AUTOMATING HARDWARE ACCELERATORS

4.1 Large Language Models

Based on our findings in Chapter 3, we have identified Non-Von-Neumann architectures as a potential solution to overcome the bottleneck barrier and create more efficient circuits. Our team is interested in exploring the potential of Artificial Intelligence and its Large Language Model to research and develop robust circuits. In this chapter, we delve into the origins of Large Language models, their training process, and the requirements for crafting them for our needs from the hardware dimension perspective. The realization of language models can be divided into four different stages of development.

4.1.1 Statistical Language Models (SLM)

Emerging in the 1990s, Statistical Language Models (SLMs) [35–37] utilize statistical learning techniques to predict the next word based on the preceding context, adhering to the Markov assumption. These models, also known as n-gram language models (e.g., bigram and trigram models), utilize a fixed context length (n). Though widely applied in information retrieval (IR) [38, 39] and natural language processing (NLP) [40–42], SLMs face the"curse of dimensionality" where estimating high-order models becomes increasingly challenging due to the exponential growth in transition probabilities required. To address this data sparsity issue, specialized smoothing techniques like backoff estimation [43] and Good-Turing estimation [44] have been developed.

4.1.2 Neural language Models (NLM)

Neural language models (NLMs) [45] revolutionized NLP by using neural networks like multi-layer perceptrons and recurrent neural networks to analyze word sequences. They introduced the concept of "distributed word representation,"where words are represented by vectors, and enabled the learning of effective features for NLP tasks. A unified, end-to-end solution for various NLP tasks was developed using NLM concepts. Additionally, word2vec, a shallow neural network, demonstrated the effectiveness of distributed word representations across a variety of NLP tasks. By applying NLP for representation learning beyond just word sequence modeling, these studies have had a profound impact on the field of NLP.

4.1.3 Pre-trained Language Models (PLM)

Early attempts like ELMo [46] and BERT pioneered the use of bi-directional Long Short Term Memory (LSTM) and Transformer architectures with self-attention mechanisms to capture context-aware word representations. These pre-trained models with general-purpose semantic features significantly improved NLP performance and established the"pre-training and fine-tunin" learning paradigm. This has led to the development of numerous Large Language Models (PLMs) with various architectures and improved pre-training strategies, all requiring fine-tuning for specific downstream tasks.

4.1.4 Large language Models (LLM)

Researchers have observed that scaling Pre-trained Language Models (PLMs) by increasing their size or the size of their training data often leads to better performance on downstream tasks, following a trend known as the scaling law [47]. Several studies have investigated the performance limits of PLMs by training increasingly larger models, including the 175B-parameter GPT-3 and the 540B-parameter PaLM.

Large language models (LLMs) have been successfully applied to a wide variety of tasks, often outperforming state-of-the-art methods. While their power in generating HDL code has undergone comprehensive investigation [6], [7], their capacity for designing high-performance and energy-efficient deep-learning hardware acceleration remains unexplored. In this work, we propose a new approach for prompt optimization aiming at eliciting knowledge from prior art of the design of hardware accelerators such as using the OpenAI's Generative Pre-trained Transformer (GPT) with a capability to understand the human languages and perform operations has acclaimed its name as ChatGPT [48]. We achieve this through designing multi-shot prompts. These are optimized to select the most relevant prior work to the problem of interest and will command ChatGPT to produce a desired design. The design is then verified using external verification tools.

4.2 Limitations in the Utilization of Gemmini

Gemmini, as proposed in Genc et al.'s work [3], represents a high-speed matrix multiplication accelerator with the potential to deliver and improve performance enhancements. For Gemmini to be utilized the potential limitations should be considered.

The limitations of Gemmini would be ***(1)*** Mastery in tools like chisel [49] a Scala embedded language and the Chipyard framework [50] is needed. ***(2)*** To utilize Gemmini effectively, one must have a deep understanding of its low-level design. This transition from high-level programming language to its architecture can be quite challenging, particularly for hardware engineers, creating a steep learning curve.***(3)***Gemmini and similar architectural accelerators[2, 51] have limitations when focused on flexibility. Gemmini relies on memory access for data transfer, which affects both performance and energy efficiency[52]. Specifically, this results in:*(i)* load/store instructions utilizing valuable processor time, impacting overall

performance. *(ii)*These instructions consume more energy compared to direct register accesses, leading to higher energy costs [52].

4.3 Understanding Large Language Model Training

Large language model needs to be trained using a large dataset, Structured on unstructured data for the LLM (Language Model for Hardware Description) to operate optimally in generating Hardware Description Language (HDL) representations of accelerators, a critical prerequisite is the availability of a well-curated and diverse dataset. This dataset serves as the foundation upon which the LLM hones its ability to comprehend and generate accurate hardware descriptions. Large language models come in diverse forms, differentiated primarily by their training methods and intended applications.They can be categorized into three main types: zero-shot models, fine-tuned models, and edge models. Each type offers distinct advantages and characteristics, catering to specific applications.

Zero-shot Models: Zero-shot LLMs are trained on a vast corpus of text and code, enabling them to generate responses to a wide range of prompts and questions without any additional training. Their versatility makes them suitable for diverse tasks, including question answering, text summarizing, and creative writing. However, their performance may not be optimal for highly specialized domains or tasks requiring context-specific knowledge.

Fine-tuned Models: Fine-tuned models are derived from zero-shot LLMs by subjecting them to additional training on specific data sets or tasks. This targeted training process allows them to develop more profound domain expertise and achieve higher accuracy within their designated applications. However, they typically have a smaller scope and are less versatile than their zero-shot counterparts.

Edge Models: Edge models are a specialized subset of fine-tuned models designed to operate on resource-constrained devices such as smartphones and edge

computing systems. They are typically much smaller and require less computational power than other LLM types, making them suitable for real-time applications. Their scope is often confined to a specific task or domain, enabling them to provide immediate feedback based on user input.

In this study, we are trying to implement a Multi-short approach for a Fine-tuned Model, which will be detailed in the next chapter 5.

LLMs for hardware design: A review of the few existing works ChipGPT [6], [7] In a study et al., [8] sources of Verilog is taken and then given as inputs with the code and the questions from the widely used opensourced text books

The process of database generation for LLM training involves meticulous curation of relevant information and examples representative of the targeted hardware accelerators. This dataset should encompass a wide range of architectures, functionalities, and design paradigms to ensure the LLM's adaptability to various scenarios.

4.4 Parameters in Constructing the Database

The construction of a database for training any artificial intelligence model requires careful consideration of several critical parameters in which few are stated below for our Large Language Model in hardware domain.

Architectural Diversity The dataset must cover a spectrum of hardware accelerator architectures, ranging from conventional designs to state-of-the-art innovations. This diversity ensures that the LLM learns to handle a broad array of design patterns and can adapt to emerging trends in hardware acceleration.

Functional Complexity: To enable the LLM to handle intricate hardware functionalities, the dataset should incorporate examples of accelerators with varying levels of complexity. This includes designs for diverse applications such as machine learning, signal processing, and scientific computation.

Programming Paradigms: Since the LLM is tasked with generating HDL code from natural language descriptions, the dataset should encompass a variety of programming paradigms. This includes the description of HDL that are generated to make sure it is learning or referring to the corresponding code to make the learninig more efficient, enabling the LLM to understand and translate diverse input sources.

Performance Metrics: The dataset should include information about the performance metrics of different accelerators. This helps the LLM learn to optimize not only for correctness but also for efficiency, taking into account factors like throughput, latency, and power consumption.

Edge Cases and Constraints: Including examples that represent edge cases or scenarios with specific constraints is crucial. This ensures that the LLM can handle unique design challenges and limitations, fostering robustness in its output.

Once the dataset is meticulously constructed, the LLM undergoes a training process where it learns to associate natural language descriptions with corresponding HDL representations. The training involves iterative exposure to examples from the dataset, allowing the model to refine its understanding and improve its capacity to generate accurate and contextually relevant HDL code.

CHAPTER 5

FRAMEWORK FOR THE LARGE LANGUAGE MODEL

The development of domain-specific Large Language Models (LLMs) for our hardware domain necessitates the creation of a specialized training dataset. This chapter delves into the generation of such a dataset, exploring various clustering techniques, detailing the LLM training framework, and addressing the challenges encountered during implementation.

5.1 Generating the Database using Gemmini

Gemmini, implemented in Chisel, stands as a cornerstone in our pursuit of advancing hardware accelerator descriptions. Since it is a hardware accelerator generator tool, we focus on tweaking its modifiable parameters and extracting the various methods of differently structured hardware accelerators to create a robust training database. By leveraging the verilator tool [53], Gemmini transforms abstract Chisel code into tangible implementations. This process generates essential components comprising C system files and corresponding Verilog code, culminating in a comprehensive System-on-Chip (SoC) design. Fig 5.1 represents how the clustering and classification of the family are defined.

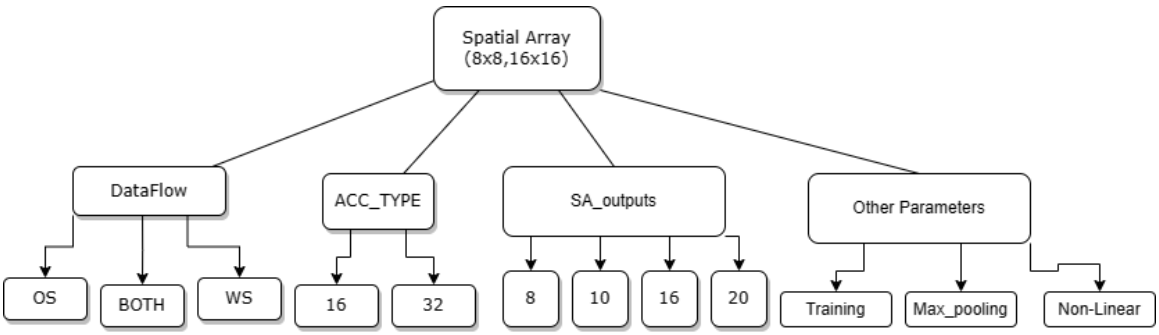

Figure 5.1 A family cluster.

Here, the heart of the gemmini, the systolic array responsible for the matrix multiplication, Multiply and Accumulate (MAC) operation, are considered the head while defining a set in our database. In this study, we are calling it a family. A family comprises different variations of the same spatial array accelerators but in different variations since they all have different parameters, as shown in Fig 5.1. Any fixed constant can be the dataflow or the ACC_TYPE, or any other parameters can be changed. Each family consists of 24 different accelerators, and we can consider them as points in a cluster. Now, we have 8 families in this study, totaling 192 different data points and eight different clusters for our model to train.

5.1.1 Bottlenecks in Our Approach

In our research, the limelight falls squarely on the hardware accelerator component. The major advantage of the gemmini is that it can be generated as a complete System-on-Chip (SoC) here the Verilog code provided by the verilator tool comprises of hundreds of thousands of lines making it physically impossible for us to train our LLM since there are certain limitations as token inputs which is discussed in the following sections this data is very large to infer the training can be compromised ideally an LLM can take upon few hundreds of lines to train for one example and can take many of such but in our case the one data point to load is itself enormous in size. To overcome this, we have considered to neglect the full-system and extracting of only the Gemmini Hardware Accelerator alone from 508 Verilog modules in the entire SoC. It is now of few thousand lines not an ideal case as we discussed, an be reliable if carefully fed to the LLM. A python code has been implemented in extracting the Gemmini Verilog module.

After observing the module, we have found it is instantiating for different modules declared in the main file (SoC), making our work even more difficult. The Gemmini module is calling approximately ~ 104 modules. We have stripped the code

snippets from every family branch and created a pool of knowledge for our LLM to learn. We have a proposal for our model creation.

5.1.2 Working of Large Language Model

A database is also referred to as a pool of knowledge. Abundant Gemmini Verilog files and their description are stored in it along with applications they can deploy. A pre-trained model is taken to train with our database in a ***(prompt, code)*** so learning of our model can be optimized. Fig 5.2 illustrating the entire diagram from the training to the end tape-out.

> *prompt: "A gemmini with a spatial array of 2x2 means with the meshRows and meshColumns of 2 with accumulator input of 32-bit SINT and spatial array output as 10 SINT with dataflow of BOTH consisting of training convolution, max-pooling, and non-linear activation function."*

Above mentioned prompt is an example with which the Verilog code is complemented.

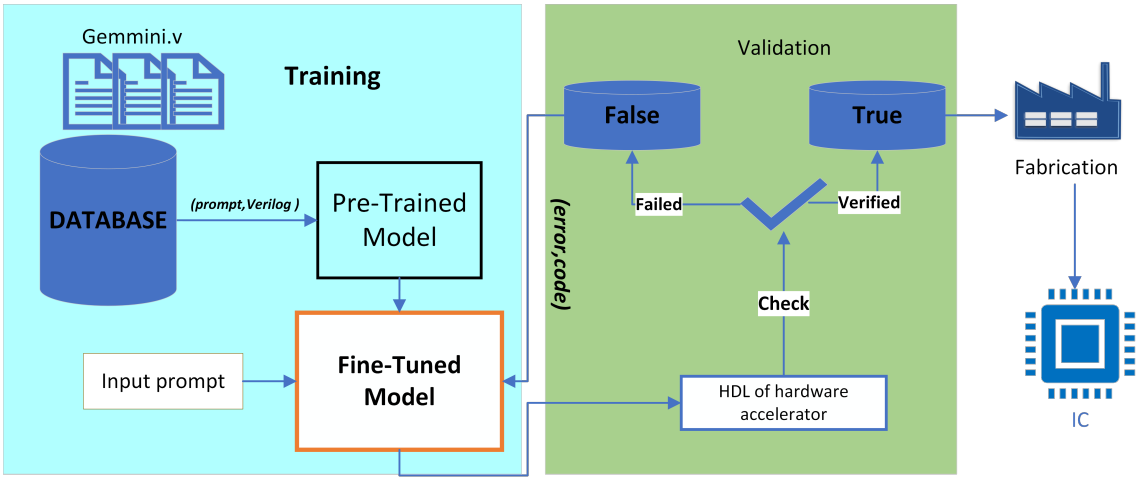

Figure 5.2 Implementation of the Large Language Model.

The pre-trained model after training can now be called a Fine-Tuned Model in which our desired inputs are given as prompts based on our application. We expect it

to generate an HDL code, in this case, a Verilog corresponding to our input request. It is then verified manually with the existing descriptions. If true, the results are stored in a new database, which can proceed to fabrication and IC; if false, then the error is identified and is reverted to the Fine-Tuned model as ***(error, code)*** error consists of where and why the error exists, we assume that the training process might have been different or the model tried out of the box and failed. Such a feedback system enables us to create an ideal model for our implementation.

CHAPTER 6

CONCLUSION

In the culmination of this master's thesis, our exploration began with a comprehensive investigation into hardware accelerators residing within both Von-Neumann and Non-Von-Neumann architectural frameworks. We scrutinized the innovative Processing-In-Memory (PIM) chip documented by [21], unraveling latent potentials and accentuating key features. Our findings invite our academic community, urging a more profound exploration into the intricacies of Non-Von-Neumann (NVM) architectures.

Shifting our focus toward practical implementation, we harnessed the formidable capabilities of Large Language Models (LLMs) to deploy hardware accelerators. This endeavor led to the creation of a meticulously curated library housing Hardware Description Language (HDL) representations of accelerators, establishing a standardized repository for future research endeavors. However, it is not without challenges, as the quantification of the extensive token input demanded by the Large Language Model revealed inherent bottlenecks. These challenges, a testament to the rigorous nature of our pursuit, underscore the need for further refinement and exploration in the realm of LLM capabilities.

This thesis serves as a foundation, unraveling the complexities of spatial array accelerators and their integration with Large Language Models and a framework is proposed in implementing it. The presented library contributes to the existing body of knowledge. It paves the way for future innovations in hardware accelerator development and LLM applications within the realm of Von-Neumann architectures.

BIBLIOGRAPHY

[1] Renzo Andri et al. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. In *ISVLSI*, pages 236–241. IEEE, 2016.

[2] Yu-Hsin Chen et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1): 127–138, 2016.

[3] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774. IEEE, 2021.

[4] Sepehr Tabrizchi, Mehrdad Morsali, Shaahin Angizi, and Arman Roohi. Nese: Near-sensor event-driven scheme for low power energy harvesting sensors. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2023. doi: 10.1109/ISCAS46773.2023.10181329.

[5] Shaojun Wei, Xinhan Lin, Fengbin Tu, Yang Wang, Leibo Liu, and Shouyi Yin. Reconfigurability, why it matters in ai tasks processing: A survey of reconfigurable ai chips. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(3):1228–1241, 2023. doi: 10.1109/TCSI.2022.3228860.

[6] Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. Chipgpt: How far are we from natural language hardware design. *arXiv preprint arXiv:2305.14019*, 2023.

[7] Pablo Antonio Martínez, Gregorio Bernabé, and José Manuel García. Code detection for hardware acceleration using large language models. *arXiv preprint arXiv:2307.10348*, 2023.

[8] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2023. doi: 10.23919/DATE56975.2023.10137086.

[9] Chenren Xu, Shuang Jiang, Guojie Luo, Guangyu Sun, Ning An, Gang Huang, and Xuanzhe Liu. The case for fpga-based edge computing. *IEEE Transactions on Mobile Computing*, 21(7):2610–2619, 2020.

[10] Saman Biookaghazadeh, Ming Zhao, and Fengbo Ren. Are fpgas suitable for edge computing? In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[11] Mar 2022. URL `https://www.geeksforgeeks.org/xilinx-fpga-architecture/`.

[12] UC Berkeley. Chipyard. `https://chipyard.readthedocs.io/en/latest/Generators/BOOM.` 2023. Accessed on 1 Nov 2023.

[13] Shaahin Angizi et al. Cmp-pim: an energy-efficient comparator-based processing-in-memory neural network accelerator. In *DAC*, pages 1–6, 2018.

[14] Shubham Jain et al. Rx-caffe: Framework for evaluating and training deep neural networks on resistive crossbars. *arXiv preprint arXiv:1809.00072*, 2018.

[15] Shaahin Angizi et al. Imce: Energy-efficient bit-wise in-memory convolution engine for deep neural network. In *ASP-DAC*, pages 111–116. IEEE, 2018.

[16] Charles Eckert et al. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *ISCA*, pages 383–396. IEEE, 2018.

[17] Shuangchen Li et al. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *DAC*, pages 1–6, 2016.

[18] Shubham Jain et al. Computing in memory with spin-transfer torque magnetic ram. *IEEE TVLSI*, 26(3):470–483, 2017.

[19] Shihui Yin et al. Xnor-sram: In-memory computing sram macro for binary/ternary deep neural networks. *IEEE Journal of Solid-State Circuits*, 55 (6):1733–1743, 2020.

[20] Zhewei Jiang et al. C3sram: An in-memory-computing sram macro based on robust capacitive coupling computing mechanism. *IEEE Journal of Solid-State Circuits*, 55(7):1888–1897, 2020.

[21] Amitesh Sridharan et al. A 1.23-ghz 16-kb programmable and generic processing-in-sram accelerator in 65nm. In *ESSCIRC*, pages 153–156. IEEE, 2022.

[22] Jonathan Bachrach et al. Chisel: Constructing hardware in a scala embedded language. In *DAC*, page 1216–1225, 2012.

[23] Amitesh Sridharan, Shaahin Angizi, Sai Kiran Cherupally, Fan Zhang, Jae-Sun Seo, and Deliang Fan. A 1.23-ghz 16-kb programmable and generic processing-in-sram accelerator in 65nm. In *ESSCIRC 2022- IEEE 48th European Solid State Circuits Conference (ESSCIRC)*, pages 153–156, 2022. doi: 10.1109/ESSCIRC55480.2022.9911440.

[24] Xilinx. Python productivity for zynq, 2018. URL `http://www.pynq.io/`.

[25] Alex Krizhevsky et al. The cifar-10 and cifar-100 datasets. URL `https://www.cs.toronto.edu/ kriz/cifar.html`.

[26] Shuchang Zhou et al. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[27] Lena E Olson et al. Security implications of third-party accelerators. *IEEE Computer Architecture Letters*, 15(1):50–53, 2016. doi: 10.1109/LCA.2015.2445337.

[28] Mihir Bellare et al. Security of symmetric encryption against mass surveillance. In *CRYPTO*, pages 1–19. Springer, 2014.

[29] Scott Spanbauer. Pentium bug, meet the ie 4.0 flaw. *PC World*, 16(2):55–55, 1998.

[30] Sangho Lee et al. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *2014 IEEE S&P*, pages 19–33, 2014. doi: 10.1109/SP.2014.9.

[31] Ziyu Wang et al. Side-channel attack analysis on in-memory computing architectures. *IEEE TETC*, 2023.

[32] Sina Sayyah Ensan et al. Scare: Side channel attack on in-memory computing for reverse engineering. *TVLSI*, 29(12):2040–2051, 2021.

[33] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security*. Version: http://www. usenix. org/events/sec01/larochelle. html, 2001.

[34] Ranyang Zhou, Sepehr Tabrizchi, Arman Roohi, and Shaahin Angizi. Lt-pim: An lut-based processing-in-dram architecture with rowhammer self-tracking. *IEEE Computer Architecture Letters*, 21(2):141–144, 2022.

[35] Jianfeng Gao and Chin-Yew Lin. Introduction to the special issue on statistical language modeling, 2004.

[36] Ronald Rosenfeld. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278, 2000.

[37] Andreas Stolcke. Srilm-an extensible language modeling toolkit. In *Seventh international conference on spoken language processing*, 2002.

[38] Xiaoyong Liu and W Bruce Croft. Statistical language modeling for information retrieval. *Annu. Rev. Inf. Sci. Technol.*, 39(1):1–31, 2005.

[39] ChengXiang Zhai et al. Statistical language models for information retrieval a critical review. *Foundations and Trends® in Information Retrieval*, 2(3): 137–213, 2008.

[40] Scott M Thede and Mary Harper. A second-order hidden markov model for part-of-speech tagging. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics*, pages 175–182, 1999.

[41] Lalit R Bahl, Peter F Brown, Peter V de Souza, and Robert L Mercer. A tree-based statistical language model for natural language speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(7):1001–1008, 1989.

[42] Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. Large language models in machine translation. 2007.

[43] Slava Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE transactions on acoustics, speech, and signal processing*, 35(3):400–401, 1987.

[44] William A Gale and Geoffrey Sampson. Good-turing frequency estimation without tears. *Journal of quantitative linguistics*, 2(3):217–237, 1995.

[45] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. *Advances in neural information processing systems*, 13, 2000.

[46] Justyna Sarzynska-Wawer, Aleksander Wawer, Aleksandra Pawlak, Julia Szymanowska, Izabela Stefaniak, Michal Jarkiewicz, and Lukasz Okruszek.

Detecting formal thought disorder by deep contextualized word representations. *Psychiatry Research*, 304:114135, 2021.

[47] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[48] URL `https://chat.openai.com.chat/`.

[49] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012. doi: 10.1145/2228360.2228584.

[50] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40 (4):10–21, 2020. doi: 10.1109/MM.2020.2996616.

[51] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[52] Sung Kim, Morteza Fayazi, Alhad Daftardar, Kuan-Yu Chen, Jielun Tan, Subhankar Pal, Tutu Ajayi, Yan Xiong, Trevor Mudge, Chaitali Chakrabarti, David Blaauw, Ronald Dreslinski, and Hun-Seok Kim. Versa: A 36-core systolic multiprocessor with dynamically reconfigurable interconnect and memory. *IEEE Journal of Solid-State Circuits*, 57(4):986–998, 2022. doi: 10.1109/JSSC.2022.3140241.

[53] Wilson Snyder. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, 2004.