Integrated Systems Architecture
02GQCOQ

Adaptation of a Gemmini-based Systolic Accelerator to Support Runtime Selection of Arithmetic Precision and Approximation

Project Report

Author:
Michael Elias: michael.elias@studenti.polito.it

Supervisor:
PhD Student Flavia Guella: flavia.guella@polito.it

Contents

List of Figures

Listings

CHAPTER 1

Introduction

Convolutional neural networks (CNNs) and other deep learning architectures have achieved remarkable success in various fields, including computer vision [10], robotics [9] and autonomous driving [1]. As CNNs rapidly expand across several fields, the threats targeting them are also on the rise, adversarial attacks are not just a theory as they have been demonstrated in real-world scenarios [5, 8].

In this project, we propose a hardware based approach to execute CNNs with mixed layer-wise approximation, where each layer operates with a different level of approximation. A software-based approach will then validate the design. This project does not examine the effectiveness of this solution as a defense against adversarial attacks; instead, it primarily focuses on the implementation and validation of the hardware design.

1.1 Report Structure

This report is split into the following chapters:

- **Chapter 2** offers an overview of the frameworks and the architecture of the accelerator.
- **Chapter 3** describes the implementation details of the project along with the problems encountered during the development.
- **Chapter 4** outlines the methodology employed to validate the design.
- **Chapter 5** presents the final results achieved.
- **Chapter 6** summarizes key elements of the project and suggests future work concepts.
- **Appendix A** consists of the user manual for the project. It explains how to run the scripts to recreate the results.
- **Appendix B** shows the final report produced by the scripts.

CHAPTER 2

Frameworks and Architectural Overview

2.1 Chipyard

Chipyard [2] is an open source framework for agile development of Chisel-based systems-on-chip. The first step for this project involved familiarization with the Chipyard framework, the conda package manager, and setting up the Chipyard repository.

2.2 Gemmini

Gemmini [6] is part of the Chipyard ecosystem, located in `Chipyard/generators/gemmini/`, and should be utilized exclusively within that context. Gemmini was developed using the Chisel hardware description language, it is a matrix-multiply accelerator targeting neural-networks and its architecture can be seen in Figure 2.1.

The main focus was the systolic array, as it is where the multiplier is located. So taking a closer look at Figure 2.2, it is evident that the systolic array consists of an array of tiles, each of which contains an array of processing elements (PEs), with each PE comprising several components, including the multiplier.

Having identified the location of the multiplier, the next step involves studying Chisel to effectively navigate Gemmini's RTL and swap it for an approximate one.

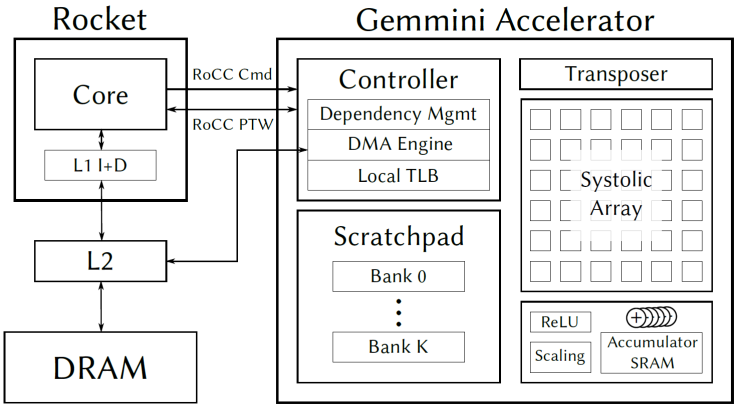

Figure 2.1: Gemmini System

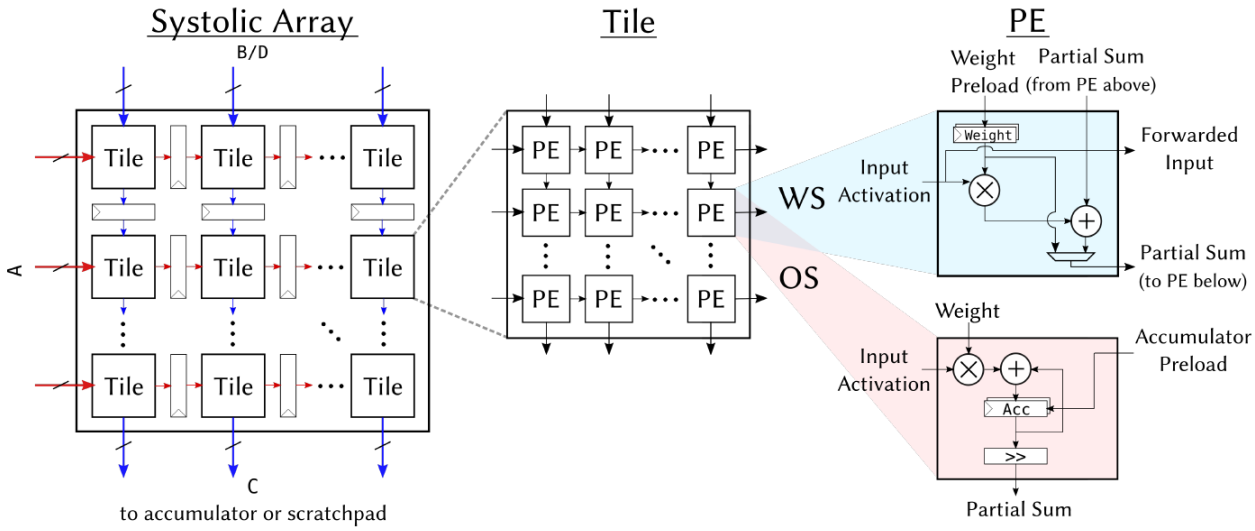

Figure 2.2: Systolic Array Architecture

The selected multiplier was the one developed and used in the MARLIN research paper [7], which features a reconfigurable multiplier with 256 approximation levels.

2.3 PyTorch/AdaPT Framework

A software-based approach was employed to validate the hardware design, utilizing a framework that incorporated PyTorch [3] and Adapt [4]. PyTorch was primarily used to generate the weights and biases for the neural network executed by Gemmini, while AdaPT was utilized to emulate approximate computation in the neural network.

CHAPTER 3

Implementation

3.1 Replacing the Multiplier

The initial challenge encountered was the language discrepancy between Gemmini's description, which is written in Chisel, and the approximation multiplier, which is implemented in SystemVerilog. The first strategy involved translating the multiplier's description from SystemVerilog to Chisel. However, it was quickly realized that this step was unnecessary, as external modules defined in Verilog or SystemVerilog can be directly inferred in Chisel using the `BlackBox` class.

3.1.1 Initial Multiplier Description

The original implementation of the multiply and accumulate (mac) operation can be found in Listing 3.1, the `mac` function can be found in Listing 3.2 at line 5. So based on this code, it is seen that the multiplication was simply described using a * operator found in `Arithmetic.scala`.

```
1  class MacUnit[T <: Data](inputType: T, cType: T, dType: T) (implicit ev: Arithmetic[T
       ]) extends Module {
2    import ev._
3    val io = IO(new Bundle {
4      val in_a  = Input(inputType)
5      val in_b  = Input(inputType)
6      val in_c  = Input(cType)
7      val out_d = Output(dType)
8    })
9
10   io.out_d := io.in_c.mac(io.in_a, io.in_b)
11 }
12 }
```

Listing 3.1: MacUnit class in PE.scala

```
1  object Arithmetic {
2    implicit object UIntArithmetic extends Arithmetic[UInt] {
3      override implicit def cast(self: UInt) = new ArithmeticOps(self) {
4        override def *(t: UInt) = self * t
5        override def mac(m1: UInt, m2: UInt) = m1 * m2 + self
6        override def +(t: UInt) = self + t
7        override def -(t: UInt) = self - t
8        ....
```

Listing 3.2: mac implementation in Arithmetic.scala

3.1.2 New Multiplier Description

First, using the `BlackBox` class required creating a `resources` directory and adding the SystemVerilog description of the approximate multiplier within it. The final directory structure appeared as follows: `src/main/resources/mul_signed_bw.sv`.

Then the `MacUnit` class had to be modified as seen in Listing 3.3, it instantiates a new class called `mul_9x9_signed_bw` which is described in Listing 3.4. This new class serves as an adapter between the `MacUnit` class and the SystemVerilog description of the approximate multiplier, since their inputs and outputs have a different size. Specifically the operands size for the `MacUnit` class are 8 bits, while the approximate multiplier expects operands on 9 bits, so a sign extension had to be implemented (lines 9 and 10 in Listing 3.4). The same was done for the output, sign extending the output of the multiplier from 18 bits to 20 bits as seen in line 17 in Listing 3.4.

Finally the SystemVerilog description can be sourced using the `BlackBox` class as seen in Listing 3.5

```
1  class MacUnit[T <: Data](inputType: T, cType: T, dType: T) (implicit ev: Arithmetic[T
       ]) extends Module {
2    import ev._
3    val io = IO(new Bundle {
4      val in_a  = Input(inputType)
5      val in_b  = Input(inputType)
6      val approximate = Input(UInt(8.W))
7      val precision = Input(UInt(14.W))
8      val in_c  = Input(cType)
9      val out_d = Output(dType)
10   })
11   val approx_mul_output = Wire(dType)
12   val approx_mul = Module(new mul_9x9_signed_bw(inputType, dType))
13   approx_mul.io.a := io.in_a
14   approx_mul.io.b := io.in_b
15   approx_mul.io.res_mask := io.precision
16   approx_mul.io.appr_mask := io.approximate
17   approx_mul_output := approx_mul.io.res
18   io.out_d := approx_mul_output + io.in_c }
```

Listing 3.3: MacUnit class in PE.scala after modification

```
1  class mul_9x9_signed_bw[T <: Data](inputType: T, dType: T) extends RawModule {
2      val io = IO(new Bundle {
3        val a = Input(SInt(8.W))
4        val b = Input(SInt(8.W))
5        val res_mask = Input(UInt(14.W))
6        val appr_mask = Input(UInt(8.W))
7        val res = Output(dType)
8      })
9      val a_inter = Cat(Fill(1, io.a(7)), io.a).asSInt
10     val b_inter = Cat(Fill(1, io.b(7)), io.b).asSInt
11
12     val mul_signed_bw_nodule = Module(new mul_signed_bw)
13     mul_signed_bw_nodule.io.a := a_inter
14     mul_signed_bw_nodule.io.b := b_inter
15     mul_signed_bw_nodule.io.res_mask := io.res_mask
16     mul_signed_bw_nodule.io.appr_mask := io.appr_mask
17     val res_inter = Cat(Fill(2, mul_signed_bw_nodule.io.res(17)), mul_signed_bw_nodule
       .io.res).asSInt
18     io.res := res_inter }
```

Listing 3.4: mul_9x9_signed_bw class in PE.scala after modification

```
class mul_signed_bw extends BlackBox with HasBlackBoxResource {
    val io = IO(new Bundle {
      val a = Input(SInt(9.W))
      val b = Input(SInt(9.W))
      val res_mask = Input(UInt(14.W)) // mask to select result precision
      val appr_mask = Input(UInt(8.W)) // mask to select result approximation
      val res = Output(SInt(18.W))
    })
      addResource("/mul_signed_bw.sv")
}
```

Listing 3.5: mul_signed_bw class inhereting from BlackBox class in PE.scala

3.2 Adding a Custom Instruction

Since the multiplier used is a runtime reconfigurable design, a new custom instruction needs to be added to Gemmini's instruction set to enable its configuration. The initial approach to adding a new instruction involved implementing it from scratch, but this caused assertion errors in the reservation station, as the station did not recognize the instruction. Attempts to bypass the reservation station and issue it immediately led to assertion errors elsewhere, so this approach was ultimately abandoned.

The second approach focused on "hijacking" the `config_mvout` instruction. According to the instruction description in the `README` section of the GitHub repository [6], when max-pooling is disabled, the bit range [63:6] of the source register rs1 is unused and set to 0, creating an opportunity for exploitation.
The format of the new instruction is `gemmini_config_multiplier(approximate_level, precision)`, it wraps the `ROCC_INSTRUCTION_RS1_RS2(x, rs1, rs2, funct)` instruction to simplify the calling process. The source registers are used as follows:

- `funct` = `0`
- `rs2` = the stride in bytes, initially fixed to a specific byte value, should be later changed to the desired setting
- `rs1[1:0]` must be `10`
- `rs1[5:4]` = max-pooling stride. If 0, then max-pooling disabled (fixed to 0 in the new instruction)
- `rs1[6]` = `gemmini_config_multiplier` is enabled (1) or disabled (0)
- `rs1[47:7]` = ignored
- `rs1[55:42]` = precision (14 bits)
- `rs1[63:56]` = approximate level (14 bits)

In this format, the new instruction must always be executed prior to `config_mvout`, as it interferes with the latter's operation; however, `config_mvout` does not interfere with the new instruction when executed first.

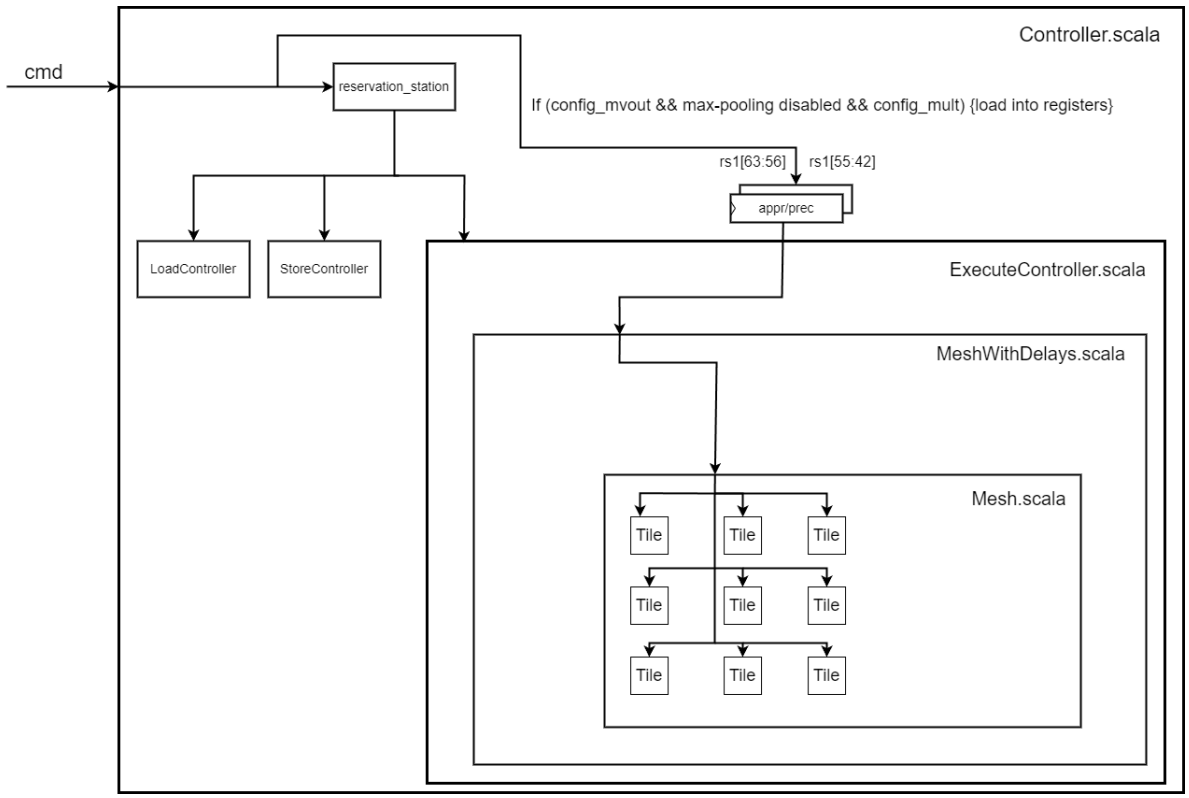

Figure 3.1: Controller Architecture

3.2.1 Modified Architecture

Several modifications were necessary to control the approximation and precision inputs of the approximate multiplier. First, two registers were added in `Controller.scala` to hold the values of the approximate level and precision, then input ports were added to the I/O interfaces of the following modules: `ExecuteController.scala`, `MeshWithDelays.scala`, `Mesh.scala` and `PE.scala` as seen in Figure 3.1 and Figure 3.2.

The registers holding the approximate and precision values are directly connected to the multipliers inside all the PEs, and they are loaded only if:

- The command is `config_mvout`, specified by `rs1[1:0]` = `10`
- Max-pooling is disabled, specified by `rs1[5:4]` = `0`
- The command is also `gemmini_config_multiplier`. specified by `rs1[6]` = `1`

The registers instantiation and the conditional loading mechanism can be seen in Listing 3.6

```
1 val approximate_reg = RegInit(0.U(8.W))
2 val precision_reg = RegInit(0.U(14.W))
3 when((io.cmd.bits.inst.funct === 0.U) && (io.cmd.bits.rs1(1,0) === 2.U) && (io.cmd.
    bits.rs1(6) === 1.U) && (io.cmd.bits.rs1(5,4) === 0.U)) {
4   approximate_reg := io.cmd.bits.rs1(63, 56)
5   precision_reg := io.cmd.bits.rs1(55, 42) }
```

Listing 3.6: Registers instantiation in Controller.scala

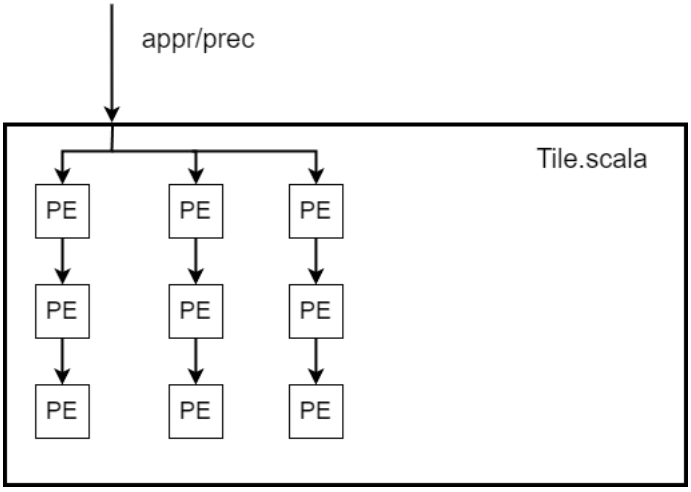

Figure 3.2: Tile Architecture

CHAPTER 4

Validation

With a preliminary functional design in place, the next step is to proceed with validation. The initial plan to validate the design was to build and run a ResNet-8 model in the Gemmini framework and compare the output matrices with the ResNet-8 model built in PyTorch. This idea was abandoned due to its complexity, as Gemmini does not offer either a Batch Normalization layer or the downsampling function required for the ResNet-8 model. The proposed solution involved executing the model in PyTorch, saving the input and output tensors, and utilizing these inputs to perform convolutional layers in Gemmini, followed by a comparison of the outputs. This chapter will elaborate on the proposed solution.

4.1 PyTorch/AdaPT Framework

A number of scripts were developed or modified to save the input and output tensors, weights, biases, approximation level for each layer and the label (prediction) for every input image.

The `resnet8_test.py` script is used to execute the forward pass of the ResNet-8 model. Two input arguments were added to the script to specify the approximation level for each layer or to apply a uniform approximation level across all layers. These arguments are `--appr_level_list` and `--appr_level`, respectively. The `get_loaders_split()` function was modified to load a number of input images equal to the batch size to ensure that the forward pass is executed exactly once. This approach prevents the input and output matrices from being overwritten during subsequent passes.

The `resnet.py` script, provided in the framework, was modified to save the input and output tensors of each layer in text files.

Two additional scripts were developed to extract the weights and biases from the `.pth` files. These scripts scale the weights and biases to `int8` according to the scaling method used in the AdaPT framework. They also export the previously saved input and output tensors, along with the weights, biases, approximation levels, labels, and predictions, as a header file for the Gemmini framework. The header file is called `resnet8_params.h`, and is saved in the `software/gemmini-rocc-tests/include/` directory

4.2 Gemmini Programs

Two Gemmini programs were developed: one to execute the convolutional layers called `conv_layer.c` and the other to perform the linear layer operations, `linear_layer.c`, they are both located in the `software/gemmini-rocc-tests/bareMetalC/` directory.

Both programs share a similar structure, with two main differences. The first program calls the `tiled_conv_auto()` function for convolutional operations, while the second program utilizes the `tiled_matmul_nn_auto()` function for linear layer operations. The second difference is that the second program calculates the index of the largest number in the final output matrix, which represents the prediction. It then compares this index with the labels stored in the header file and returns an error in the event of a mismatch.

Both programs read the input matrices, weights, biases, and approximation levels from `resnet8_params.h` to perform their operations. They then compare the results with those also stored in `resnet8_params.h`. Several functions were added to both programs to compute the error between the calculated output and the expected values stored in the header file. These functions calculate the Mean Absolute Error (MAE), Mean Squared Error (MSE), and Maximum Absolute Error (Max AE), and they print the error values at the end of execution.

4.2.1 Finding the Scale for Each Layer

The `tiled_conv_auto()` and `tiled_matmul_nn_auto()` functions accept a scaling factor as an input parameter, and each layer was identified to have an appropriate scaling factor for achieving accurate results.

The scaling factor for `tiled_conv_auto()` was determined by creating a script that tests a range of scaling factors for each layer, the one that yields the lowest error was selected. By running this script for every convolutional layer, 7 scaling factors were identified and saved in a list for later use.

The scaling factor for `tiled_matmul_nn_auto()` was determined using an alternative approach. Since prediction accuracy is the primary goal, rather than minimizing error values, the scale was selected to maximize the margin between correct predictions and other potential outcomes.

4.2.2 Choosing the Approximation Level for Each Layer

In the PyTorch/AdaPT framework, the linear layer was implemented as a standard `nn.Linear` rather than the AdaPT version. As a result, its approximation level is fixed at 0 (exact) in the Gemmini implementation. The approximation level for the remaining layers was selected based on results from NSGA-II, which explores the search space of approximate neural networks for ResNet-8. This approach identifies a set of Pareto-optimal approximation levels that balance power efficiency and accuracy.

4.3 Test Flow

With all the necessary parameters for each layer established, `conv_layer.c` can be executed seven times for the convolutional layers and `linear_layer.c` can be executed once for the linear layer, as the ResNet-8 model comprises seven convolutional layers and one linear layer. However, manually executing these programs while changing the parameter names for each layer is inconvenient. Consequently, two scripts were developed to automate this process.

The scripts are `slow_resnet8.sh` and `fast_resnet8.sh`. Both scripts contain a main loop that iterates eight times, executing either `conv_layer.c` or `linear_layer.c` based on the layer type, while dynamically changing the parameter names within these programs according to the layer being executed.

The primary difference lies in the error computation. The slow version configures the programs to write the output matrix to a text file, then invokes a Python script to compare this output with the matrix stored in the header file. In contrast, the fast version skips output matrix printing; it computes the error directly within the program and simply displays the error value, resulting in significantly faster execution compared to writing to a file.

CHAPTER 5

Results

The final run was executed using approximation levels of [43, 23, 15, 23, 92, 4, 63, 0], as determined by NSGA-II. The input size was 4x32x32x3, representing a batch of four 32x32 RGB images, with the batch size set to 4.

5.1 Computed Variance

The error for each layer between the computed output matrices and the ones imported from the PyTorch/AdaPT framework can be found in Table 5.1. For the detailed report see **Appendix B**. The errors are sufficiently small to validate the correctness of the design, additionally, `linear_layer.c` did not return a prediction mismatch error therefore the images were correctly classified.

The error values in the linear layer are intentionally high, as discussed previously. Since the primary purpose of the linear layer is prediction, the scaling factor was selected to maximize prediction accuracy rather than minimize error values.

Layer Name	**MAE**	**MSE**	**Max AE**
conv_1	0.63954	0.91850	6.0
layer1_0_conv1	0.00624	0.00624	1.0
layer1_0_conv2	0.02336	0.02336	1.0
layer2_0_conv1	0.02365	0.02365	1.0
layer2_0_conv2	0.02032	0.02032	1.0
layer3_0_conv1	0.00964	0.00964	1.0
layer3_0_conv2	0.0	0.0	0.0
linear	44.02499	2467.62500	84.0

Table 5.1: MAE, MSE and Max AE for each layer in the ResNet-8 model

5.2 Runtime

Gemmini offers two simulators for running programs: Spike and Verilator [11]. Spike generally operates at a much faster speed compared to cycle-accurate simulators like Verilator. However, Spike is limited to verifying functional correctness and cannot provide accurate performance metrics or profiling information. Furthermore, Spike does not account for any modifications made to the RTL, which makes it unsuitable for testing the new design with the approximate multiplier.

The runtime for `slow_resnet8.sh` and `fast_resnet8.sh` using Spike is approximately 15 seconds. In contrast, running a single layer using the slow version of the script in Verilator took 5 hours and 40 minutes, which would result in an estimated total runtime of around 39 hours for the entire model. The fast version of the script completed the entire model in 4 hours, making it approximately 10 times faster. A summary of these results is presented in Table 5.2.

Simulator	**slow_resnet8.sh**	**fast_resnet8.sh**
Spike	15 seconds	15 seconds
Verilator	39 hours	4 hours

Table 5.2: Runtime Summary

CHAPTER 6

Summary

This report examined the frameworks utilized and the modifications made to the RTL of Gemmini's description. These changes aimed to replace an exact multiplier with an approximate one, enabling layer-wise approximation selection, as well as to introduce a custom instruction for controlling the multiplier.

The design was subsequently validated using a framework that combines PyTorch [3] and AdaPT [4]. The discrepancies between the results generated by Gemmini and those produced by the PyTorch/AdaPT framework were minimal, indicating that the design was verified.

6.1 Future Work

While the current study has successfully integrated an approximate multiplier within the Gemmini accelerator, several potential avenues for future research and enhancement have been identified:

1. **Development of a Custom Instruction from Scratch:** Future work should consider developing a new custom instruction without exploiting another established instruction. This would require modifying the RTL design of the reservation station and potentially other modules.

2. **Script Generalization:** The existing scripts could be further refined to accommodate a wider variety of ResNet architectures, such as ResNet-14 or ResNet-20, thereby increasing their applicability.

3. **Construction of a full ResNet Model in Gemmini:** Instead of executing one layer at a time with subsequent comparisons, a complete ResNet model could be built in the Gemmini framework. This approach would streamline the validation process by allowing comparisons only at the end of the execution.

References

[1] Mohammed Al-Qizwini, Iman Barjasteh, Hothaifa Al-Qassab, and Hayder Radha. Deep learning algorithm for autonomous driving using googlenet. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 89–96, 2017.

[2] 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.

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024.

[4] Dimitrios Danopoulos, Georgios Zervakis, Kostas Siozios, Dimitrios Soudris, and Jörg Henkel. Adapt: Fast emulation of approximate dnn accelerators in pytorch. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2022.

[5] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on machine learning models. *CoRR*, abs/1707.08945, 2017.

[6] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[7] Flavia Guella, Emanuele Valpreda, Michele Caon, Guido Masera, and Maurizio Martina. Marlin: A co-design methodology for approximate reconfigurable inference of neural networks at the edge. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 71(5):2105–2118, 2024.

[8] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *CoRR*, abs/1607.02533, 2016.

[9] Harry A. Pierson and Michael S. Gashler. Deep learning in robotics: A review of recent research. *CoRR*, abs/1707.07217, 2017.

[10] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[11] Wilson Snyder, Paul Wasson, Duane Galbi, and et al. Verilator.

APPENDIX A

Guide to Running Project Scripts

This appendix will focus exclusively on scripts that are executed directly from the terminal; scripts that are invoked by other scripts will not be included.

A.1 Pytorch/AdaPT Framework

To properly generate and export the header file for the Gemmini framework, the following scripts should be executed in the specified order: `resnet8_test.py`, `extract_weights_and_biases.py`, and `export_params.py`.

A.1.1 resnet8_test.py

This script saves the input, output tensors and approximation level for each layer, along with the labels for every input image.
Usage for same approximation level for all layers:

```
python neural_networks/CIFAR10/resnet8_test.py --appr_level 5
```

Usage for different approximation level for each layer:

```
python neural_networks/CIFAR10/resnet8_test.py --appr_level_list 43 23 15 23 92 4 63 0
```

A.1.2 extract_weights_and_biases.py

As its name indicates, it extracts the weights and biases from the `.pth` file. Doesn't take any arguments.
Usage:

```
python extract_weights_and_biases.py
```

A.1.3 export_params.py

Combines all the previously saved input and output tensors, weights, biases, images labels and approximation levels into a single header file, resnet8_params.h and copies the it to the Gemmini directory.
Usage:

```
python export_params.py
```

A.2 Gemmini Framework

There are three main scripts in this framework, they can be executed in any order: `find_scale.sh`, `slow_resnet8.sh`, and `fast_resnet8.sh`.

A.2.1 find_scale.sh

Used to find the proper scale for each layer. It accepts a layer name as input argument, the range of the tested scales can be changed by modifying the for loop inside the script. Once the scale is found, it should be added to the list called `scale` in `slow_resnet8.sh`, and `fast_resnet8.sh`
Usage:

```
./find_scale.sh layer1_0_conv1
```

A.2.2 slow_resnet8.sh

Calculates and prints the discrepancies between the computed output matrix for each layer and the output matrix imported from the PyTorch/AdaPT framework. This script accepts one input argument, which can be either `spike` or `verilator`; however, executing this script with Verilator is not advisable due to the lengthy runtime.
Usage:

```
./slow_resnet8.sh spike
```

A.2.3 fast_resnet8.sh

Serves a similar function to `slow_resnet8.sh`, with the key distinction that this script does not print the output matrix.
Usage:

```
./fast_resnet8.sh verilator
```

APPENDIX B

Detailed Report

```
conv_1:
Flattening weights took 18446744073709542971 cycles
Gemmini conv took 46556 cycles
Error compuation took 46556 cycles
Mean Absolute Error (MAE) = 0.63954
Mean Squared Error (MSE) = 0.91850
Maximum Absolute Error (Max AE) = 6.0

layer1_0_conv1:
Flattening weights took 18446744073709528444 cycles
Gemmini conv took 55385 cycles
Error compuation took 55385 cycles
Mean Absolute Error (MAE) = 0.00624
Mean Squared Error (MSE) = 0.00624
Maximum Absolute Error (Max AE) = 1.0

layer1_0_conv2:
Flattening weights took 18446744073709528444 cycles
Gemmini conv took 55385 cycles
Error compuation took 55385 cycles
Mean Absolute Error (MAE) = 0.02336
Mean Squared Error (MSE) = 0.02336
Maximum Absolute Error (Max AE) = 1.0

layer2_0_conv1:
Flattening weights took 18446744073709502268 cycles
Gemmini conv took 29970 cycles
Error compuation took 29970 cycles
Mean Absolute Error (MAE) = 0.02365
Mean Squared Error (MSE) = 0.02365
Maximum Absolute Error (Max AE) = 1.0

layer2_0_conv2:
Flattening weights took 18446744073709468443 cycles
Gemmini conv took 47871 cycles
```

```
Error compuation took 47871 cycles
Mean Absolute Error (MAE) = 0.02032
Mean Squared Error (MSE) = 0.02032
Maximum Absolute Error (Max AE) = 1.0

layer3_0_conv1:
Flattening weights took 18446744073709253952 cycles
Gemmini conv took 6020 cycles
Error compuation took 6020 cycles
Mean Absolute Error (MAE) = 0.00964
Mean Squared Error (MSE) = 0.00964
Maximum Absolute Error (Max AE) = 1.0

layer3_0_conv2:
Flattening weights took 18446744073708416098 cycles
Gemmini conv took 11551 cycles
Error compuation took 11551 cycles
Mean Absolute Error (MAE) = 0.000000
Mean Squared Error (MSE) = 0.000000
Maximum Absolute Error (Max AE) = 0.0

linear:
Gemmini matmul took 1138 cycles
Error compuation took 1138 cycles
Mean Absolute Error (MAE) = 44.02499
Mean Squared Error (MSE) = 2467.62500
Maximum Absolute Error (Max AE) = 84.0

Predictions match
```