Contents

1 Introduction

There is an increase in compute demand of various neural nets and this has significant impact on the amount of energy required to train them on the current cloud architectures that largely involve general purpose microprocessors. One of the reason is the slowdown of Moore's Law and the Dennard Scaling that dictate the scaling and efficiency of the transistors on a chip. To meet these high requirements of neural nets one of the approach is to develop specialized hardware that improve the speed and efficiency of training/inference.

Gemmini[4] is a Full-system and Full-stack implementation of a custom DNN (Deep Neural Network) accelerator developed by University of Berkely, discussed in detail in section 3. It is an architecture that considers the system-level parameters that determine the overall SoC (System on Chip) and the full software stack. Meaning it enables end-to-end, implementation and evaluation of hardware accelerator.

The SoC integration is made possible with the RISC-V (Reduced Instruction Set Computer) ecosystem. The project uses the Rocket Chip[11] generator which generates SoCs based on the RISC-V Instruction Set Architecture.

Gemmini is written in Chisel (Constructing Hardware in Scala embedded language) which resembles traditional RTL like Verilog but it enables functional and object-oriented descriptions of circuits in Scala programming language. It then uses the FIRRTL (flexible intermediate representation for RTL) [12] compiler to generate its Verilog version.

In this project we study the various Verilog build files and its implementation on the Cyclone V FPGA.

Also we look into implementing the Systolic core that has some enhancements like support for Weight Stationary and Output Stationary data flows, optional im2col block, transposers, ReLU unit, pooling blocks on a DE1-SoC FPGA without the RISC-V core, and integrate it rather with ARM HPS system.

2 Background

The neural networks are composed of input layers, one or multiple hidden layers and an output layer. Each node in the layer connects the each node in the subsequent layer, if the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.

Each of these individual nodes computes a weighted sum of inputs from the previous layer, making it computationally intensive with number of multiplications and additions (MAC) performed for huge Neural nets. Hence, it very much needed to reduce the cost of performing the MAC operations.

These networks rely on training data, and improving their efficiency over time, called the learning phase. After reaching the fine-tuned accuracy they can be used to classify or cluster data with high velocity, the inference phase which is a bit faster compared to the learning phase.

2.1 CNN Architectures

Convolution Neural Networks are a class of networks that are designed for processing structured arrays of data such as images, speech or audio signals.
Three main layers of a CNN are :

- **Convolution Layer** : A key unit of a CNN which are good at picking patterns in the input data.

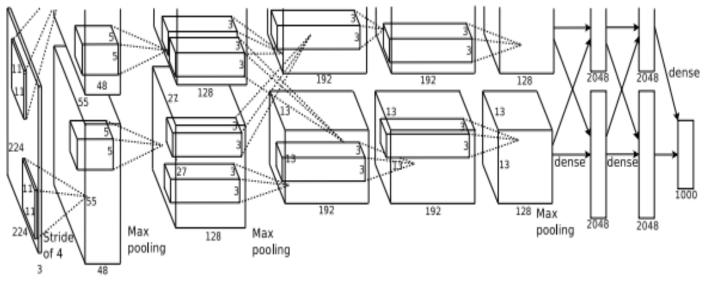
- **Pooling Layer** : Is a layer used to down sample or reduce the dimensionality on the number of inputs.
- **Fully-Connected Layer** : Is a simple feed forward network and form the last few layers of the network. The input to this layer is normally the output of a convolution or pooling layer that is fully flattened.

There are many state-of-the-art CNN architectures. In general, most deep convolutional neural networks are made of a key set of basic layers discussed above. Here, we discuss some of them and compare based on the Total MACs, error and weights needed to train/infer.

2.1.1 Alex Net

A well known Deep neural net, it composed of 5 convolution layers, 3 Max Pooling layers and 3 fully connected layers. It uses the ReLu as its activation/non-linear layer.

It was proposed by Alex Krizhevesky and won the most difficult ImageNet challenge for visual object recognition[2] called the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 .

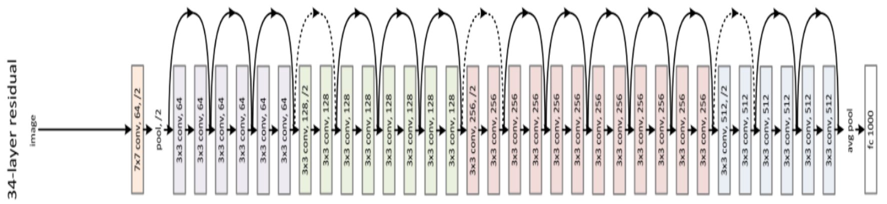


Source: From the paper, ImageNet Classification with Deep Convolutional Neural Networks
Figure 1: Architecture of AlexNet

2.1.2 ResNet

The paper[1] discussed about problems encountered while stacking more layers to create a neural network like the issues of vanishing gradients, and degradation where the accuracy gets saturated. ResNet solves these problems and can be realised as a feed-forward network with ”shortcut connections”.

With these shortcut connections or skip connections the ResNet34 architecture achieves lower training error and also reduced the top-1 error by 3.5% compared to its plain counter part[1].



Source: From the paper, Deep Residual Learning for Image Recognition
Figure 2: Architecture of ResNet34

2.1.3 MobileNet

MobileNet[5] is an efficient and portable CNN architecture that is used in real world applications. Traditionally deeper and more complicated networks are created in order to achieve higher accuracy but these advances to improve accuracy will not necessarily make networks more efficient with respect to size and speed.

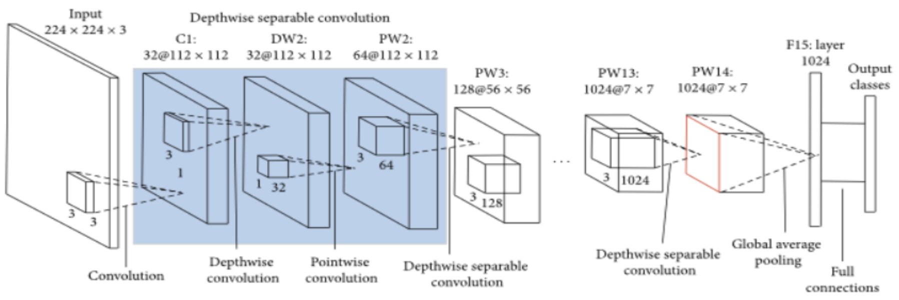
When we consider applications that are running on computationally limited platforms we need efficient neural network architectures and MobileNet defines two hyper parameters width multiplier and resolution multiplier to define smaller and efficient neural nets.

The architecture is based on depthwise separable convolutions[5] which is a form of factorized convolutions. Here, we perform convolution in two stages.

- Depthwise Convolution or the Filtering stage
- Pointwise Convolution or the combination stage

By expressing convolution as a two step process of filtering and combining we get a reduction in computation of

$$1/N + 1/D_k^2$$



Source: Form medium post explaining architecture of MobileNet [13]
Figure 3: Architecture of MobileNet

Hence, smaller and faster MobileNets can be built by trading off a reasonable amount of accuracy to reduce size and latency.

2.1.4 GoogLeNet

A standard way of increasing the performance include both increasing the depth – the number of levels – of the network and its width: the number of units at each level.

However this simple solution comes with two major drawbacks.

- Over-fitting
- Increased use of computational resources

It aims to solve these problems by moving from fully connected to sparsely connected architectures.

GoogLeNet[3] is a type of convolutional neural network based on the Inception architecture. It utilises Inception modules, which allow the network to choose between multiple convolutional filter sizes in each block.

Parameters	AlexNet	GoogleNet	ResNet	MobileNet
Total MACs	724M	1.43G	3.9G	569M
Number of Conv Layers	5	21	34	16
Number of Weights	2.3M	6M	23.5M	4.2M
Input Size	227x227	224x224	224x224	224x224
Strides	1,4	1,2	1,2	1,2

Table 1: Computational complexity

2.2 DNN Accelerators

It can be seen that there is huge computational dependence on simple linear algebra (MACs) like calculations. It would be efficient to use devices that specifically meaning perform just one task that is to compute these arithmetic operations with high efficiency and speed. This gave rise to development of these custom hardware.

DNN accelerators are custom hardware implemented to achieve efficiency in processing DNNs, by efficiency we mean they show improved accuracy, speed in inference and training with reduced energy and cost when compared to traditional systems.

Some of the key objectives [14] in the achieving this are we should try to:

- Increasing throughput and reduce latency : As MACs constitute the major computation, the custom processor has to reduce the time per MAC, which may be achieved by reducing the clock frequency or the instruction overhead required to compute.
- Avoid unnecessary MACs
- Increase the number of processing elements and also the utilization of these elements.
- Reducing energy consumption : When we look at energy cost of each operation we see that the cost of moving the data from the memory to the processing unit is more than just the cost of performing an addition or the multiplication.

DNN accelerators are normally spatial and have parallel execution units laid out in a systolic fashion. Following are some architectures currently employed.

2.2.1 Tensor Processing Units

Is Google's answer to this problem. It is an application-specific integrated circuits (ASICs) used to accelerate machine learning workloads that rely heavily on linear algebra computation. They do so by minimizing the time-to-accuracy when you train large, complex neural network models. Models that previously took weeks to train on other hardware platforms can converge in hours on TPUs[6].

The TPU [7] can't run word processors, control rocket engines, or execute bank transactions, but they can handle massive matrix operations used in neural networks at fast speeds.

Their primary task is matrix processing, which is a combination of multiply and accumulate operations. TPUs contain thousands of multiply-accumulators that are directly connected to each other to form a large physical matrix. This is called a systolic array architecture. Cloud TPU v3, contain two systolic arrays of 128 x 128 ALUs, on a single processor.

The TPU is mainly used when

- Models dominated by matrix computations
- Models with no custom TensorFlow/PyTorch/JAX operations inside the main training loop.
- Models that train for weeks or months
- Large models with large effective batch sizes

2.2.2 NVDLA

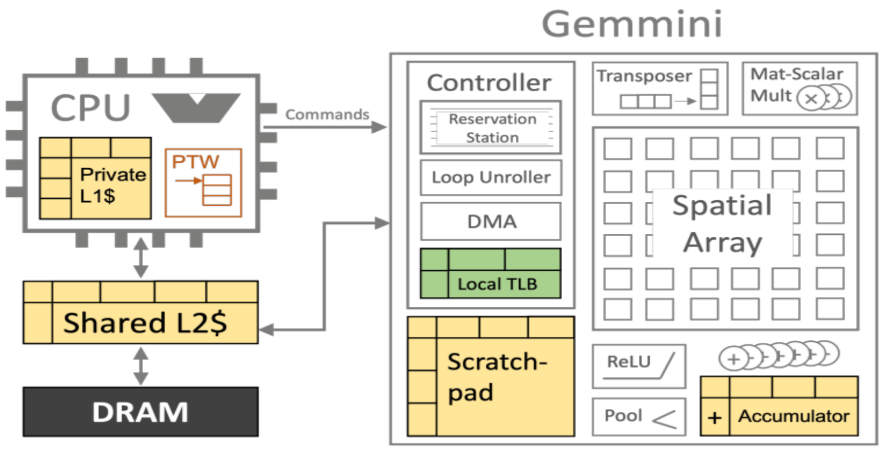
The NVIDIA Deep Learning Accelerator (NVDLA) is a free and open architecture that promotes a standard way to design deep learning inference accelerators.

It was born out of NVIDIAs Tegra Xavier[8] SoC that comprises an eight-core CPU cluster, GPU with additional inference optimizations, deep learning accelerator, vision accelerator, and a set of multimedia accelerators providing additional support for machine learning.

3 Gemmini

Gemmini as mentioned in the introduction is an architecture that provides a full-system and full-stack integration of the DNN accelerator. Here, when we talk about full-system it means that it considers the effect of the SoC and the operating system running on it. In such an integration we should consider the system-level effects that reduce the performance of the accelerator. Some those reasons are due the systems memory hierarchy. Here, each accelerator has its own memory called the scratchpad and the Rocket core or the CPU has its cache and each core will have access to the shared memory. In such an hierarchy the cache hits/misses do impact the performance. It also considers the performance effects caused by the MMU(Memory Management Units). Operating System issues like context switching between threads, interrupts are all system level issues that affects the performance of the accelerator.

By Full-stack integration Gemmini provides a programming stack in which the acceleration can be programmed using higher level APIs like the onnx runtime and also access to low level ISA.



Source: Form the presentation, Gemmini IISWC 2021
Figure 4: Gemmini SoC

3.1 Rocket Chip

Looking into the accelerator needs some understanding of the SoC or the Full system integration provided with the help of the Rocket Chip generator. It consists of a collection of parameterized chip-building libraries that we can use to generate different SoC variants.

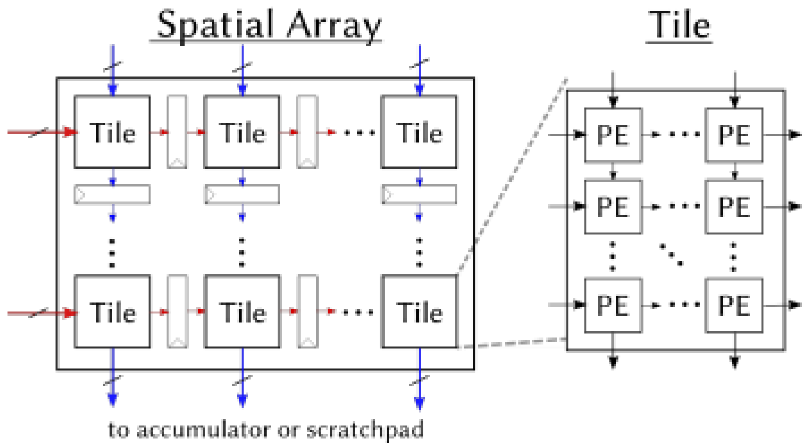
The Gemmini project has two different host CPU configurations. One is a low-power in-order Rocket core, and a high-performance out-of-order BOOM [16] core.

3.2 Architecture

3.2.1 Systolic Core

The design of Gemmini's [4] spatial array is a two-level hierarchy to provide a flexible template for different structures.

The spatial array is first composed of tiles, where tiles are connected via explicit pipeline registers. Each of the individual tiles can be further broken down into an array of PEs, where PEs in the same tile are connected combinationally without pipeline registers. Each PE performs a single multiply-accumulate (MAC) operation every cycle, using either the weight or the output stationary dataflow.



Source: From the Gemmini paper
Figure 5: Microarchitecture of Gemmini's two-level spatial array

3.2.2 Data movement

The inputs fed into the systolic array and outputs generated are stored in scratchpads that are made of banked SRAM. A DMA engine facilitates the data transfer from the main memory to this scratchpad.

The local memory or the scratchpad has number of rows that is identical to the number of PEs in a row of the systolic array. Meaning the 4x4 systolic array would have the 4 rows in the scratchpad memory. The accumulator stores the output of the calculation and is wider as the MAC results would need more bits to represent.

Load Pipeline : Gemmini has three load instruction that are intended to be for moving Inputs, Weights and biases. The are used to move matrices of the the size of the systolic array from the main memory to the scratchpad.

3.2.3 Execution

The execution or the actual matrix multiplication uses two separate instructions. **Preload** instruction is load the stationary data (i.e Weights for Weight Stationary data flow and biases from the Output Stationary dataflow) and then **Compute** instruction that is used to perform multiplication on the stationary data loaded.

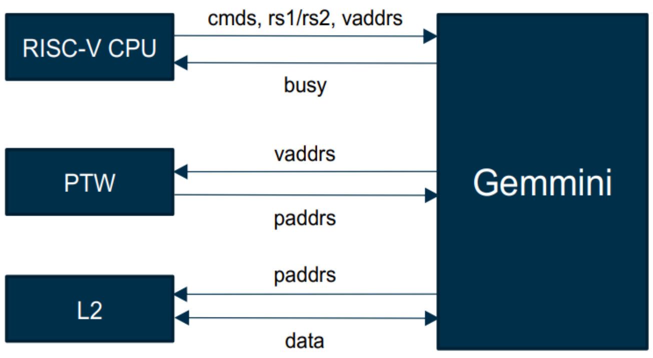
3.2.4 Storing of results

The store in performed using the mvout instruction that is similar to the load instruction, and moves data of the size of the systolic array into the main memory from the accumulator.

3.2.5 Interface with CPU

Gemmini is implemented as a RoCC(Rocket Custom Co-processor) accelerator with non-standard RISC-V custom instructions. Using the RoCC software [15] which is a set of C and RISC-V Assembly macros that help with emitting these instructions for talking with RoCCs.

How this works is whenever a CPU encounters a RoCC instruction it checks the opcode to find the target and dispatches it to the target co-processor. As the CPU does not know what the instruction is doing it continues its execution and then synchronizes with fences.



Source: Form the presentation, Gemmini IISWC 2021
Figure 6: Gemmini and CPU interface

3.3 Chisel Workflow

Chisel is a hardware construction language embedded in the high-level programming language Scala. It is a library of special class definitions, predefined objects, and usage conventions within Scala, so when you write Chisel you are actually writing a Scala program that constructs a hardware graph.

Hence, it uses the scala build tools(sbt) the defacto open source build tools to compile and package scala programs.

3.3.1 Installation

1. Install sbt on linux

   `https://www.scala-sbt.org/1.x/docs/Installing-sbt-on-Linux.html`

2. Version installed

```
root@5320060a5bd3:~# sbt sbtVersion
[info] welcome to sbt 1.4.9 (Ubuntu Java 11.0.13)
[info] loading project definition from /root/project
[info] set current project to root (in build file:/root/)
[info] 1.4.9
```

3.3.2 Chisel module

Let us look at a simple implementation of ALU, that performs basic mathematical operations depending on select signal.

1. Creating an ALU class that extends from chisel module

```
class Alu extends Module {
  val io = IO(new Bundle {
    val fn = Input(UInt(2.W))
    val a = Input(UInt(4.W))
    val b = Input(UInt(4.W))
    val result = Output(UInt(4.W))
  })
  val fn = io.fn
  val a = io.a
  val b = io.b

  val result = Wire(UInt(4.W))
  // some default value is needed
  result := 0.U
  // The ALU selection
  switch(fn) {
    is(0.U) { result := a + b }
    is(1.U) { result := a - b }
    is(2.U) { result := a | b }
    is(3.U) { result := a & b }
  }
  // Output on the LEDs
  io.result := result
}
```

2. Module instantiation and emitVerilog function to generate verilog code

```
// Generate the Verilog code
object AluMain extends App {
  (new chisel3.stage.ChiselStage).emitVerilog(new Alu(),
   Array("--target-dir", "generated"))
}

Example from : https://github.com/schoeberl/chisel-examples
```

3.3.3 Emitting Verilog code

1. Create a Makefile with the below rule to run the sbt to run the object AluMain that generates the verilog in the generated folder as specified in the emitVerilog function call.

```
SBT = sbt
# Running object using runMain command
alu:
        $(SBT) "runMain AluMain"
```

3.3.4 Verilog Code

1. Generated files consists of the firrtl representation of the code and also .json file consisting of the annotations.

```
root@5320060a5bd3:~/ChiselExamples/generated# ls
Alu.anno.json  Alu.fir  Alu.v
```

2. Generated verilog code for the Alu module

```
module Alu(
  input        clock,
  input        reset,
  input  [1:0] io_fn,
  input  [3:0] io_a,
  input  [3:0] io_b,
  output [3:0] io_result
);
  wire [3:0] _result_T_1 = io_a + io_b;
  wire [3:0] _result_T_3 = io_a - io_b;
  wire [3:0] _result_T_4 = io_a | io_b;
  wire [3:0] _result_T_5 = io_a & io_b;
  wire [3:0] _GEN_0 = 2'h3 == io_fn ? _result_T_5 : 4'h0;
  wire [3:0] _GEN_1 = 2'h2 == io_fn ? _result_T_4 : _GEN_0;
  wire [3:0] _GEN_2 = 2'h1 == io_fn ? _result_T_3 : _GEN_1;
  assign io_result = 2'h0 == io_fn ? _result_T_1 : _GEN_2;
endmodule
```

3.4 Modules

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

3.4.1 Execute Controller

This module is responsible for executing ”execute”-type ISA commands, such as matrix multiplications. It includes a systolic array for dot-products, and a transposer.

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

There are two dataflows supported:

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

3.4.2 Load Controller

This module is responsible for all instructions that move data from main memory into Gemmini's private scratchpad or accumulator.

3.4.3 Store Controller

This module is responsible for all instructions that move data from Gemmini's private SRAMs into main memory. This module is also responsible for ”max-pooling” instructions, because Gemmini performs pooling when moving unpooled data from the private SRAMs into main memory.

3.4.4 Systolic Core

The central unit of computation is the spatial array of distributed processing elements(PEs) each of which performs dot products and accumulations. The PE is implemented in PE.scala file which consists of two classes.

1. **PE Class** :
   The PEControl class defines member fields that define the different dataflow architecture that the PE is meant to perform (Output Stationary, Weight Stationary)

   ```
   class PEControl[T <: Data : Arithmetic](accType: T)
      extends Bundle {
   ...
   }
   ```

   The PE class implements the actual MAC module

   ```
   class PE[T <: Data](inputType: T, outputType: T, accType: T,
               df: Dataflow.Value, max_simultaneous_matmuls: Int)
               (implicit ev: Arithmetic[T]) extends Module {
   ...
   }
   ```

2. **Tile Class** :
   As mentioned a Tile is an array of PEs that are combinationally connected without any pipeline registers,

   ```
   class Tile[T <: Data] (...)
     (implicit ev: Arithmetic[T]) extends Module {
   ...
   }
   ```

3. **Mesh Class** :
   Mesh is an array of Tiles that are connected with pipeline registers to adjacent tiles,

   ```
   class Mesh[T <: Data : Arithmetic](...)extends Module {
   ...
   }
   ```

4. **MeshWithDelays class** :
   This module implements the Mesh, and also delay registers which shift the inputs to the systolic array.

   ```
   class MeshWithDelays[T <: Data: Arithmetic,
                        U <: TagQueueTag with Data](...)
                        extends Module {
   ...
   }
   ```

3.5 System Setup

3.5.1 Docker

A Docker image was created consisting of the tools required to run build/test the Gemmini accelerator, following commands [10] were part of the Dockerfile to generate the Docker,

1. Clone the chipyard repository, for installing tools for the RISCV core.

   ```
   git clone https://github.com/ucb-bar/chipyard.git
   cd chipyard
   git checkout dcf8da4b2d3a4deead95462fce36a6db5693ed45

   // Install ubuntu requirements
   ./scripts/ubuntu-req.sh 1>/dev/null && \

   // Install RISCV toolchain
   ./scripts/init-submodules-no-riscv-tools.sh
   ./scripts/build-toolchains.sh esp-tools

   source env.sh
   ```

2. Fetch the gemmini generator.

   ```
   cd generators/gemmini
   git fetch && git checkout v0.6.4
   git submodule update
   ```

3. Installing RISCV ISA

   ```
   cd toolchains/esp-tools/riscv-isa-sim/build
   git fetch && git checkout 090e82c473fd28b4eb2011ffcd771ead6076faab
   make && make install
   ```

3.6 Gemmini default configuration

The default Gemmini configuration is defined in the GemminiConfig.scala file, and following are parameters that define the nature 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 parts through different parts of the accelerator
inputType: T,
spatialArrayOutputType: T,
accType: T,

// Define 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),
```

3.7 Verilator Build

Verilator is a free and open-source software tool which converts Verilog (a hardware description language) to a cycle-accurate behavioral model in C++ or SystemC.

3.7.1 Verilog output

1. Run the command to build cycle-accurate Verilator simulator for the Gemmini programs

   ```
   root@:~/chipyard/generators/gemmini# ./scripts/build-verilator.sh
   ```

   This runs the Makefile in /sims/verilator folder present in the main chipyard directory, and generates an executable

2. Check the generated-src folder in the same directory for the generated verilog files. We deepdive into the file

   ```
   chipyard.TestHarness.CustomGemminiSoCConfig.top.v
   ```

3.7.2 Verilator simulation

1. Run the following command to build RISC-V binaries to run on the SoC

   ```
   root@: cd chipyard/generators/gemmini/software/gemmini-rocc-tests
   root@: ./build.sh
   ```

   The template.c is a basic baremetal C program[9] that is tested on the Gemmini acclerator. It creates a Identity matrix of size 16x16 loads data from main memory to the scratchpad memory. It then calls the gemmini_config_ex function with OS dataflow with bias as 0, and finally stores the output in the scratchpad memory back to the main memory. This is followed by a fence instruction that guarantees ordering between memory operations as the processor makes a call to the accelerator to perform computations.

2. Run verilator based simulation of the binaries in baremetal mode using the run-verilator script

   ```
   root@chipyard/generators/gemmini:./scripts/run-verilator.sh template
   ```

3. Output

   ```
   ...
   Flush Gemmini TLB of stale virtual addresses
   Initialize our input and output matrices in main memory
   Calculate the scratchpad addresses of all our matrices
   Move "In" matrix from main memory into Gemmini's scratchpad
   Move "Identity" matrix from main memory into Gemmini's scratchpad
   Multiply "In" matrix with "Identity" matrix with a bias of 0
   Move "Out" matrix from Gemmini's scratchpad into main memory
   Fence till Gemmini completes all memory operations
   Check whether "In" and "Out" matrices are identical
   Input and output matrices are identical, as expected
   ```

3.7.3 Verilog Modules

To understand the different modules that make up the Gemmini SoC, we need to check the top verilog file as mentioned in the verilog output section step 2.

1. Extract the modules into separate verilog files using the script

   ```
   root@: ./test.sh <top>.v
   ```

   It constitutes to 506 submodules that make up the system. One module of interest is the Gemmini which is implemented in Gemmini.v verilog file

   This module has the following hierarchy and in specific we look at the detailed implementation of the Execute controller:

```
-Scratchpad
-CounterController
-FrontendTLB
-ReservationStation
-LoopConv
-LoopMatmul
-LoadController
-StoreController
-ExecuteController
        -TransposePreloadUnroller
        -MultiHeadedQueue_1
        -Queue_61
        -MeshWithDelays
                -AlwaysOutTransposer
                -Mesh
                        -Tile
                                -PE_256
                -TagQueue
                -Queue_62
-Im2Col
-Arbitrer
-Queue_55
```

The Execute controller module instantiates the Systolic Core a 16x16 array(i.e each Tile instantiates a single PE module, and the Mesh module instantiates 256 Tile modules making the mesh) as mentioned in the gemmini configrations and this is the main interest as we want to decouple the Systolic core and integrate it with the ARM HPS of the DE1-SoC FPGA.

4 Inputs and Hardware

4.1 FPGA

1. **Intel Cyclone V FPGA** The DE1-SoC Development Kit presents a hardware design platform built around the Altera System-on-Chip (SoC) FPGA, which combines the dual-core Cortex-A9 embedded cores with programmable logic for design flexibility.

   Altera's SoC integrates an ARM-based hard processor system (HPS) consisting of processor, peripherals and memory interfaces tied with the FPGA fabric using a high-bandwidth interconnect backbone.

2. **Xilinx Zynq-7000 FPGA** The Zynq-7000 SoC [17] family integrates the software programmability of an ARM-based processor with the hardware programmability of an FPGA, enabling key analytics and hardware acceleration while integrating CPU, DSP, ASSP, and mixed signal functionality on a single device.

4.2 FPGA Builds

1. **Intel**

   (a) Create a New Project folder from Quartus Prime

   (b) Creating the Mesh module, copy the extracted verilog file (Mesh.v, Tile.v and PE_256.v) from the verilator build, and make the Mesh.v as the top level module in the hierarchy.

   (c) **Analysis and Synthesis** Checking the RTL viewer we see the netlist being generated. Mesh or the sys_core has huge number of primitive components and fails to utilize the DSP cores on the device due to the generated verilog file. The lowest level meaning the Processing Element PE blocks are synthesized with prmitive components like adders and multipliers.

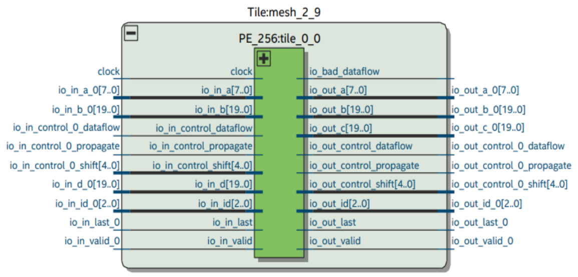


Figure 7: PE module

2. **Xilinx**
   (a) Use the build-test.tcl script to run synthesis on the Xilinx FPGA using Vivado.
   (b) A Makefile has the rule build-viv that would run the synthesis of the module mentioned in the tcl file.
   (c) The utilization and timing reports are generated for further analysis.

4.3 Reports

1. **Intel** Cyclone V FPGA:

Modules	Resources	Used	Available	Utilization %
PE_256	ALM	485	32070	1.51
	DSP	3	87	3.4
Tile	ALM	485	32070	1.51
	DSP	3	87	3.4
Mesh	ALM	83695	32070	261
	DSP	261	87	300

Table 2: Resource utilization

2. **Xilinx**

Modules	Resources	Used	Available	Utilization %
PE_256	Slice LUT	760	53200	1.43
	Slice Resgisters	65	106400	0.06
	DSP	0	220	0
Tile	Slice LUT	760	53200	1.43
	Slice Resgisters	67	106400	0.06
	DSP	0	220	0
Mesh	Slice LUT	196773	53200	369.87
	Slice Resgisters	32191	106400	30.25
	DSP	0	220	0

Table 3: Resource utilization

5 Conclusion

Building efficient DNN accelerators is a challenging task and the need to run these application on the embedded or edge devices that are resource limited make it a active area of research. It finds its application in domains like image processing, speech, audio recognition, robotics, autonomous vehicles etc.

In this project we looked into the open source DNN accelerator called Gemmini, its architecture and implementation using Chisel HDL language. Also tested some simple binaries and ran neural network models like resNet50. The Gemmini being integrated with the RISC-V core provides a platform to test its full-system stack. However, it is also worth noticing that RISC-V being free and open source ISA is used less in practice compared to the commercially available ARM devices and creating such a full-system stack around the ARM environment would give us an opportunity to test the architecture on real world applications and scenarios.

We were able to synthesize the core modules on the Cyclone V FPGA and understand the the verilog files generated from Chisel HDL and the utilization of the resources. The generated verilog for a 16x16 systolic core when synthesized made use of a very high number of primitives and is not advisable for placement and routing. It has to be optimized to be able to run the DE1-SoC FPGA board.

6 Future Work

The preliminary work did give us a good knowledge about the Gemmini implementation and the whole system integration with the RISC-V core. These are some of the steps looking forward in the project.

- The systolic core is well integrated with RISCV using the modules Load, Store and Execute controller. We look to create custom modules to integrate with the ARM HPS core.
- The calls to systolic core are performed with the help of RoCC and the step forward will be replacing it with the AXI bus interface.
- The input and parameters are stored in the Scratchpad that should be converted to M10k BRAMs available on the Altera FPGA.

References

[1] Deep Residual Learning for Image Recognition, https://arxiv.org/pdf/1512.03385.pdf

[2] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In NIPS, pp. 1106–1114, 2012

[3] Going deeper with convolutions, https://arxiv.org/pdf/1409.4842v1.pdf

[4] Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration https://arxiv.org/abs/1911.09925

[5] MobileNets: Efficient Convolutional Neural Networks for Mobile Vision-https://www.overleaf.com/project/625b3da5c2e483761fdbc1a4 Applications https://arxiv.org/abs/1704.04861

[6] https://cloud.google.com/tpu/docs/tpus

[7] https://cloud.google.com/tpu/docs/intro-to-tpu

[8] https://en.wikichip.org/wiki/nvidia/tegra/xavier#Architecture

[9] https://sites.google.com/berkeley.edu/gemminitutorialiiswc2021/

[10] https://github.com/ucb-bar/gemmini

[11] https://chipyard.readthedocs.io/en/latest/Generators/Rocket-Chip.html

[12] https://www.chisel-lang.org/firrtl/

[13] https://medium.com/analytics-vidhya/image-classification-with-mobilenet-cc6fbb2cd470

[14] Efficient Processing of Deep Neural Networks: A Tutorial and Survey by Vivienne Sze.

[15] https://github.com/IBM/rocc-software

[16] https://chipyard.readthedocs.io/en/latest/Generators/BOOM.html

[17] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

Glossary

ARM Advanced RISC Machine. 2

CNN Convolutional Neural Networks. 2

DMA Direct Memory Access. 7

DNN Deep Neural Networks. 2, 6

FIRRTL Flexible Intermediate representation for RTL. 2

FPGA Field Programmable Gate Array. 2

HPS	Hard Processor System. 2
ISA	Instruction Set Architecture. 6
MAC	Mulitply-Accumulate. 2
MMU	Memory Management Units. 6
RoCC	Rocket Custom Co-Processor. 8
RTL	Register Transfer Level. 2
SoC	Systom on Chip. 2
SRAM	Static Random Access Memory. 7
TPU	Tensor Processing Unit. 5