FPGA BASED ACCELERATOR FOR CONVOLUTIONAL NEURAL NETWORKS

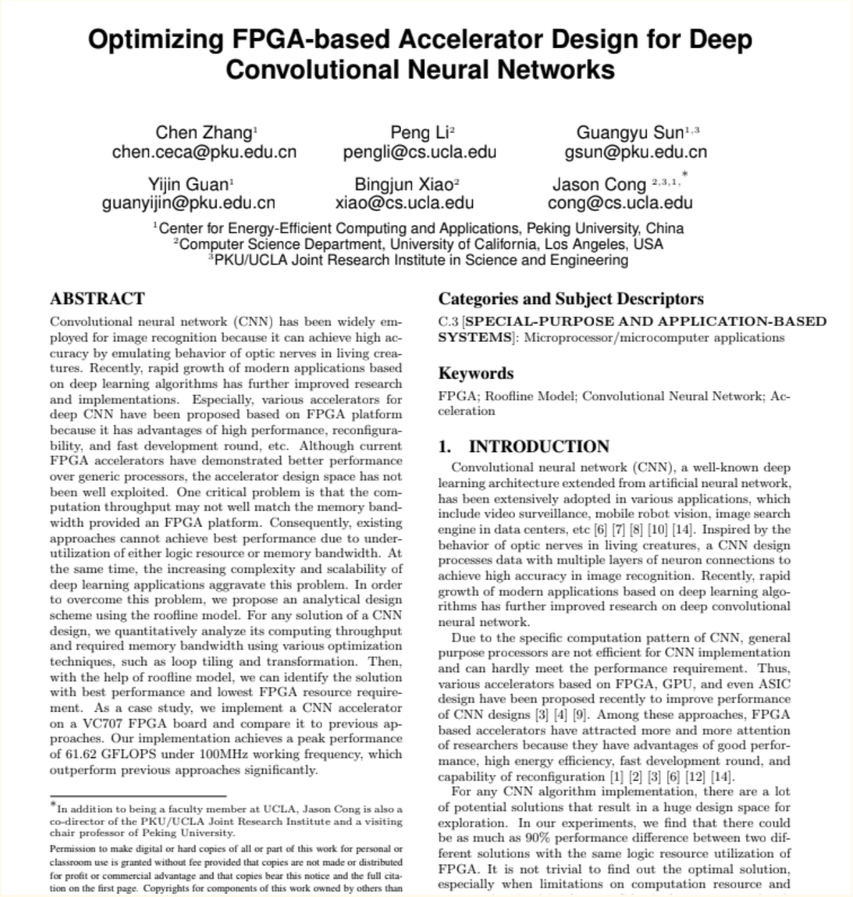
Kevin Wang
Pavan Sai Guntha

Agenda

- Project Background
- Convolution Neural Networks
- Accelerator Architecture
- Implementation
- Results | Learnings | Challenges
- AI Usage

Project Background

- Our project is based on replicating the techniques used in the research paper “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks.”
- Their research was funded by C-FAR, NSF China, National High Technology Research and Development Program of China, and RFDP.
- We decided to use a different CNN model and an FPGA board due to the complexity and resources available.



Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks

Chen Zhang[1] chen.ceca@pku.edu.cn
Peng Li[2] pengli@cs.ucla.edu
Guangyu Sun[1,3] gsun@pku.edu.cn
Yijin Guan[1] guanyijin@pku.edu.cn
Bingjun Xiao[2] xiao@cs.ucla.edu
Jason Cong[2,3,1,*] cong@cs.ucla.edu

[1]Center for Energy-Efficient Computing and Applications, Peking University, China
[2]Computer Science Department, University of California, Los Angeles, USA
[3]PKU/UCLA Joint Research Institute in Science and Engineering

ABSTRACT

Convolutional neural network (CNN) has been widely employed for image recognition because it can achieve high accuracy by emulating behavior of optic nerves in living creatures. Recently, rapid growth of modern applications based on deep learning algorithms has further improved research and implementations. Especially, various accelerators for deep CNN have been proposed based on FPGA platform because it has advantages of high performance, reconfigurability, and fast development round, etc. Although current FPGA accelerators have demonstrated better performance over generic processors, the accelerator design space has not been well exploited. One critical problem is that the computation throughput may not well match the memory bandwidth provided an FPGA platform. Consequently, existing approaches cannot achieve best performance due to under-utilization of either logic resource or memory bandwidth. At the same time, the increasing complexity and scalability of deep learning applications aggravate this problem. In order to overcome this problem, we propose an analytical design scheme using the roofline model. For any solution of a CNN design, we quantitatively analyze its computing throughput and required memory bandwidth using various optimization techniques, such as loop tiling and transformation. Then, with the help of roofline model, we can identify the solution with best performance and lowest FPGA resource requirement. As a case study, we implement a CNN accelerator on a VC707 FPGA board and compare it to previous approaches. Our implementation achieves a peak performance of 61.62 GFLOPS under 100MHz working frequency, which outperform previous approaches significantly.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Microprocessor/microcomputer applications

Keywords

FPGA; Roofline Model; Convolutional Neural Network; Acceleration

1. INTRODUCTION

Convolutional neural network (CNN), a well-known deep learning architecture extended from artificial neural network, has been extensively adopted in various applications, which include video surveillance, mobile robot vision, image search engine in data centers, etc [6] [7] [8] [10] [14]. Inspired by the behavior of optic nerves in living creatures, a CNN design processes data with multiple layers of neuron connections to achieve high accuracy in image recognition. Recently, rapid growth of modern applications based on deep learning algorithms has further improved research on deep convolutional neural network.

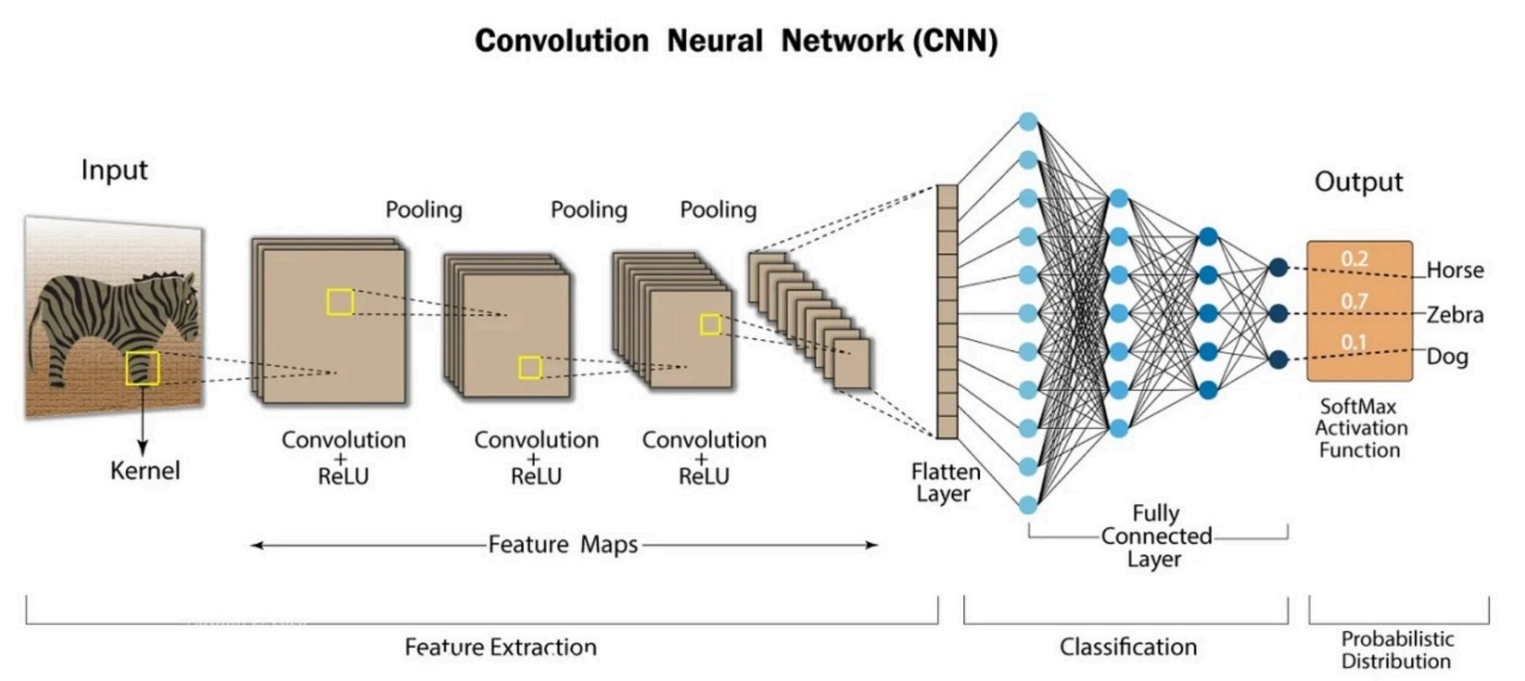
Due to the specific computation pattern of CNN, general purpose processors are not efficient for CNN implementation and can hardly meet the performance requirement. Thus, various accelerators based on FPGA, GPU, and even ASIC design have been proposed recently to improve performance of CNN designs [3] [4] [9]. Among these approaches, FPGA based accelerators have attracted more and more attention of researchers because they have advantages of good performance, high energy efficiency, fast development round, and capability of reconfiguration [1] [2] [3] [6] [12] [14].

For any CNN algorithm implementation, there are a lot of potential solutions that result in a huge design space for exploration. In our experiments, we find that there could be as much as 90% performance difference between two different solutions with the same logic resource utilization of FPGA. It is not trivial to find out the optimal solution, especially when limitations on computation resource and

*In addition to being a faculty member at UCLA, Jason Cong is also a co-director of the PKU/UCLA Joint Research Institute and a visiting chair professor of Peking University.

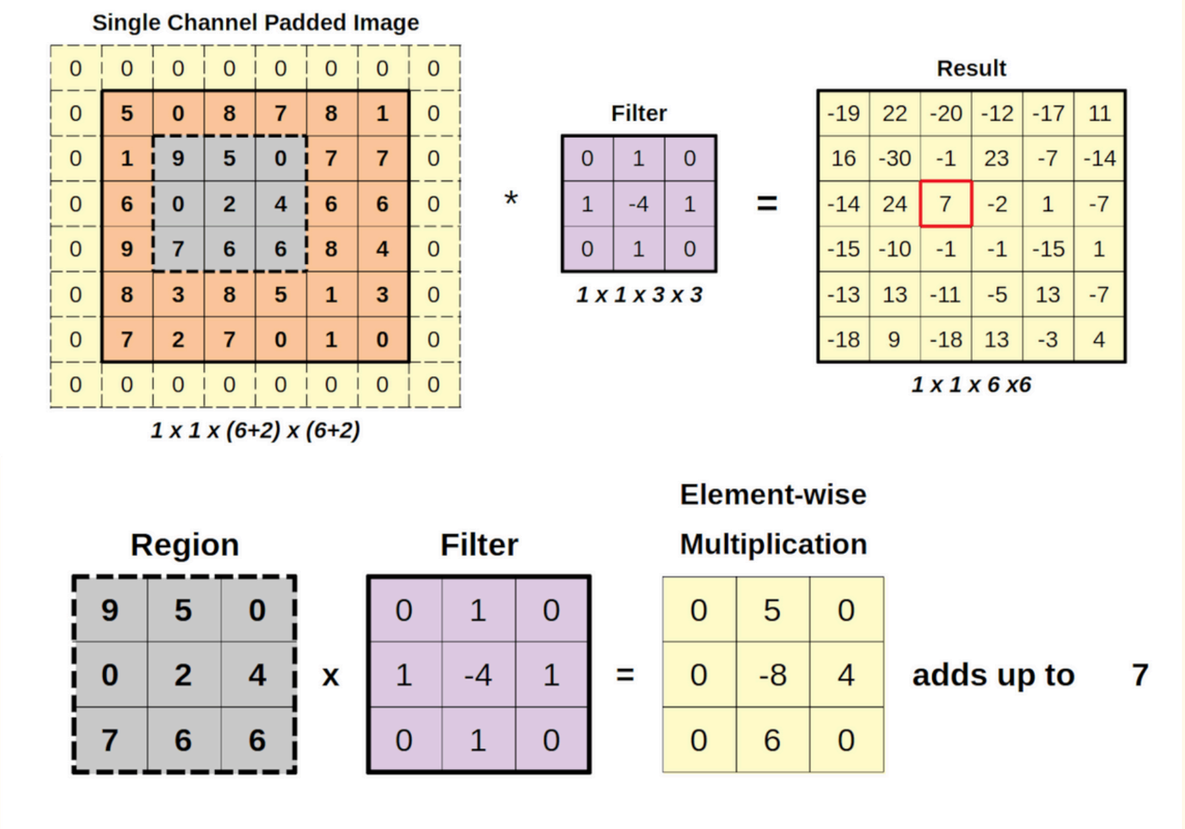
Convolutional Neural Networks

- A specialized type of deep learning algorithm designed to process data with a grid structure.
- CNNs are particularly effective for tasks such as image classification, which we focused on in this project.



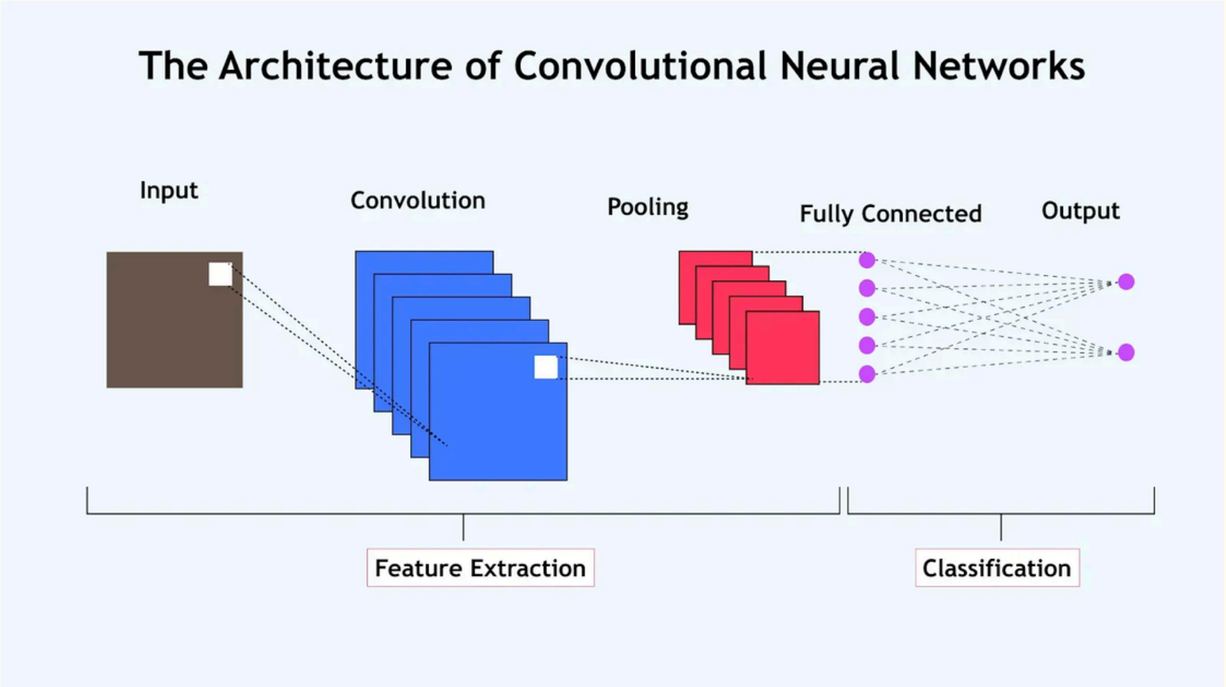
Convolutional Neural Network Layers

- CNNs have various types of layers:
  - Input Layer
  - Convolutional Layers
  - Activation Layers (e.g., ReLU)
  - Pooling Layers
  - Fully Connected Layers



Convolutional Neural Networks Architecture

- The layer workflow goes as follows:
  - Input image → Convolutional layer detects edges
  - ReLU removes negative values
  - Pooling downsamples edges
  - Fully connected layer combines edges into shapes

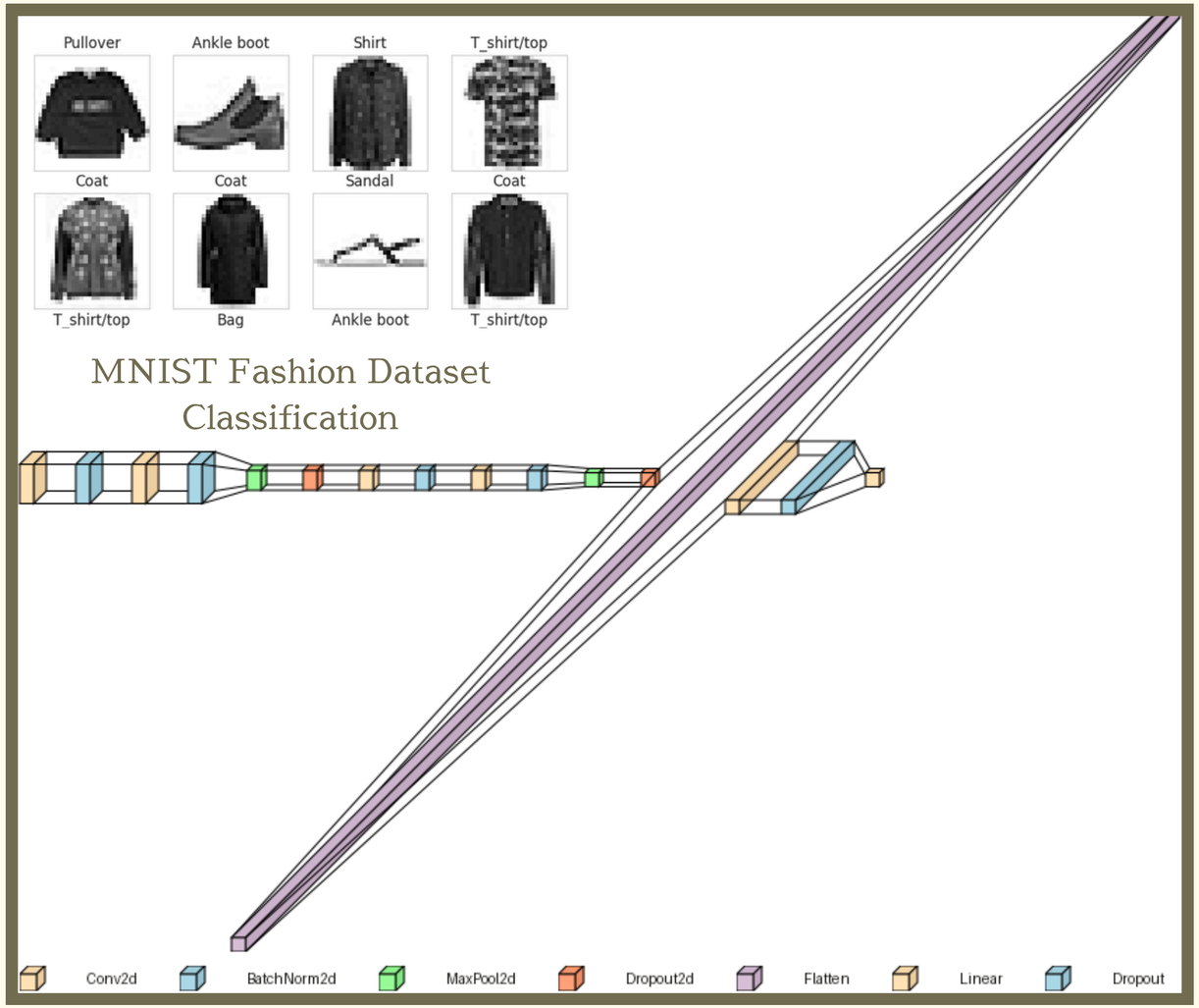


CNN Model Architecture

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
batch_normalization (BatchNormalization)	(None, 28, 28, 32)	128
conv2d_1 (Conv2D)	(None, 28, 28, 32)	9,248
batch_normalization_1 (BatchNormalization)	(None, 28, 28, 32)	128
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
dropout (Dropout)	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 14, 14, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 14, 14, 64)	256
conv2d_3 (Conv2D)	(None, 14, 14, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 14, 14, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 64)	0
dropout_1 (Dropout)	(None, 7, 7, 64)	0
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 128)	401,536
dropout_2 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 1,404,992 (5.36 MB)

Trainable params: 468,202 (1.79 MB)



Accelerator Design

- We used convolution loop unrolling for our accelerator design

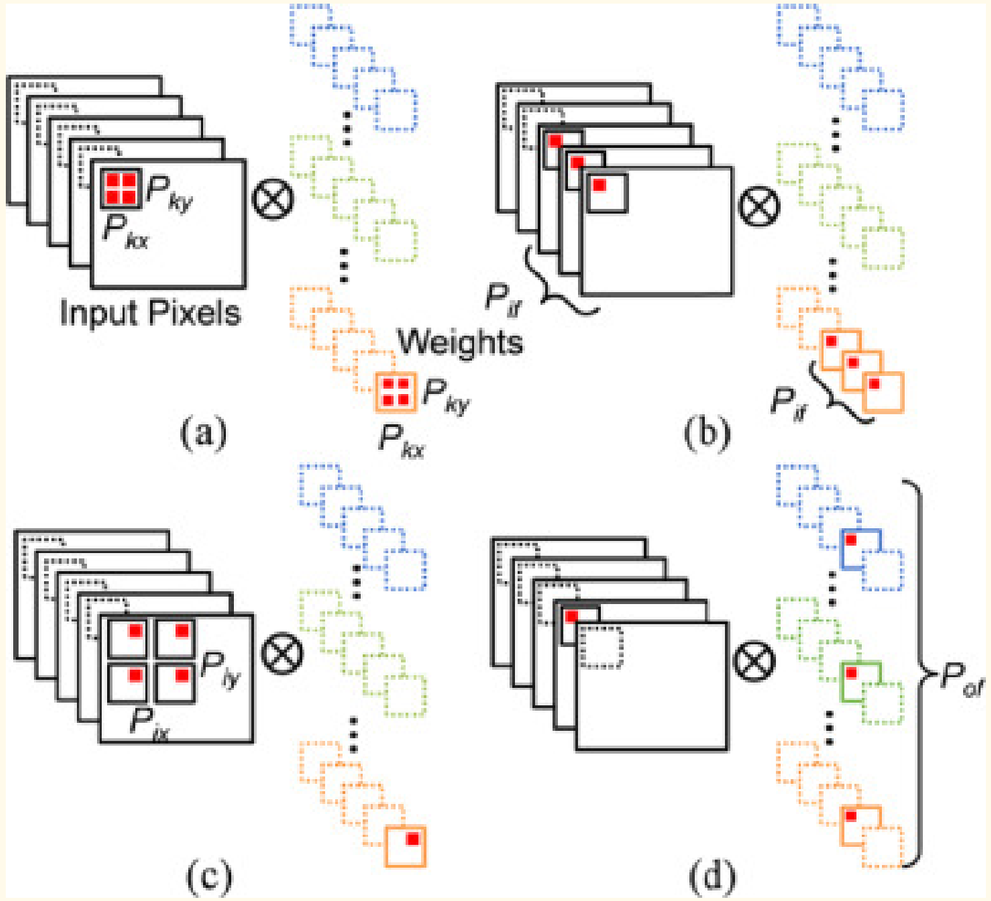
Before

```
for (i = 0; i < n; i++) {
  a[i] = a[i] + b[i];
}
```

After

```
for (i = 0; i < n; i += 4) {
  a[i]   = a[i]   + b[i];
  a[i+1] = a[i+1] + b[i+1];
  a[i+2] = a[i+2] + b[i+2];
  a[i+3] = a[i+3] + b[i+3];
}
```

Example of loop unrolling with a factor of 4



Accelerator Design

- We also utilized loop tiling and loop reordering

```
for( int i = 0; i < 4; i++)
    for( int j = 0; j < 4; j++)
        f(A[i], B[j]);
```

Without loop tiling

```
for( int i=0; i<2; i++)
    for( int j=0; j<2; j++)

        for( int ti=0; ti<2; ti++)
            for( int tj=0; tj<2; tj++)
                f(A[i*2 + ti], B[j*2 + tj]);
```

With loop tiling

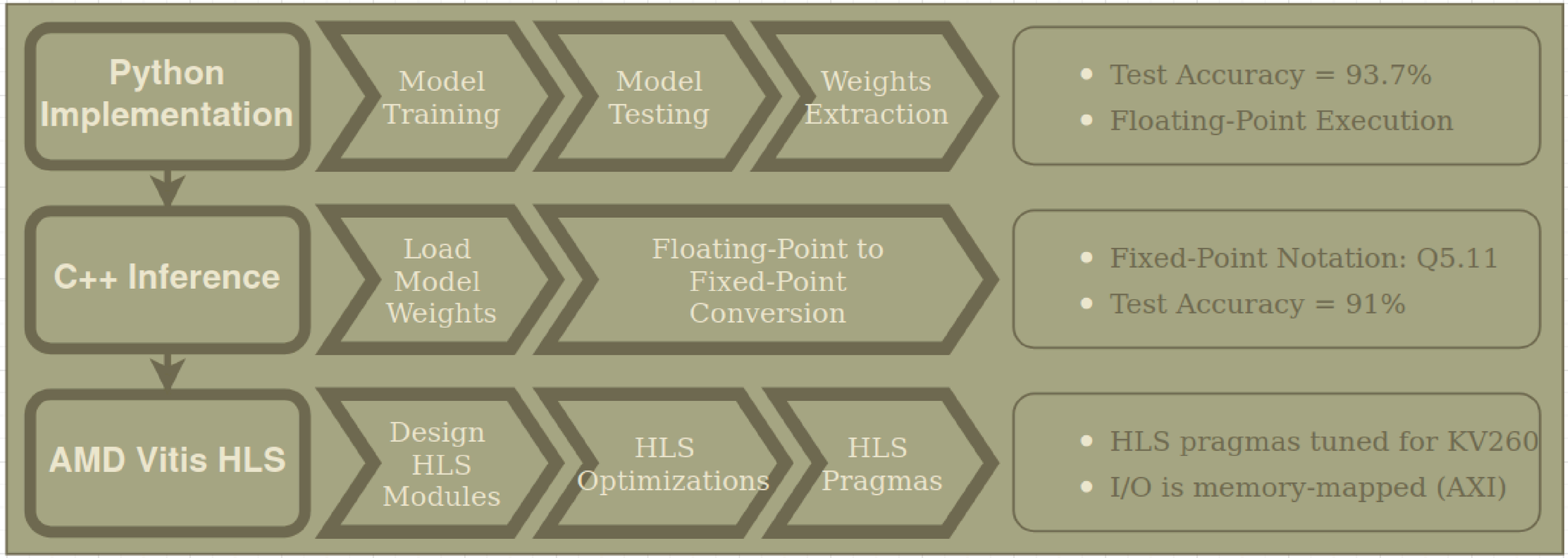
```
for (ii = 0; ii < N; ii += TILE)
  for (jj = 0; jj < N; jj += TILE)
    for (kk = 0; kk < N; kk += TILE)
      for (i = ii; i < ii+TILE; i++)
        for (j = jj; j < jj+TILE; j++)
          for (k = kk; k < kk+TILE; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Without loop reordering

```
for (ii = 0; ii < N; ii += TILE)
  for (kk = 0; kk < N; kk += TILE)
    for (jj = 0; jj < N; jj += TILE)
      for (i = ii; i < ii+TILE; i++)
        for (k = kk; k < kk+TILE; k++)
          for (j = jj; j < jj+TILE; j++)
            C[i][j] += A[i][k] * B[k][j];
```

With loop reordering

Implementation Overview



HLS Top Module - nnet

```
325  void nnet(
326      float24_t* image,
327      float24_t* conv1_weights,
328      float24_t* conv1_bias,
329      float24_t* conv2_weights,
330      float24_t* conv2_bias,
331      float24_t* fc1_weights,
332      float24_t* fc1_bias,
333      float24_t* fc2_weights,
334      float24_t* fc2_bias,
335      float24_t* predictions
336  ) {
337      // MAXI interfaces with proper alignment and burst lengths
338      #pragma HLS INTERFACE m_axi port=image offset=slave bundle=gmem0 depth=784 max_read_burst_length=256 max_write_burst_length=25
339      #pragma HLS INTERFACE m_axi port=conv1_weights offset=slave bundle=gmem1 depth=288 max_read_burst_length=256
340      #pragma HLS INTERFACE m_axi port=conv1_bias offset=slave bundle=gmem2 depth=32 max_read_burst_length=16
341      #pragma HLS INTERFACE m_axi port=conv2_weights offset=slave bundle=gmem3 depth=18432 max_read_burst_length=256
342      #pragma HLS INTERFACE m_axi port=conv2_bias offset=slave bundle=gmem4 depth=64 max_read_burst_length=16
343      #pragma HLS INTERFACE m_axi port=fc1_weights offset=slave bundle=gmem5 depth=401408 max_read_burst_length=256
344      #pragma HLS INTERFACE m_axi port=fc1_bias offset=slave bundle=gmem6 depth=128 max_read_burst_length=16
345      #pragma HLS INTERFACE m_axi port=fc2_weights offset=slave bundle=gmem7 depth=1280 max_read_burst_length=256
346      #pragma HLS INTERFACE m_axi port=fc2_bias offset=slave bundle=gmem8 depth=10 max_read_burst_length=16
347      #pragma HLS INTERFACE m_axi port=predictions offset=slave bundle=gmem9 depth=10 max_write_burst_length=16
348
349      #pragma HLS INTERFACE s_axilite port=return bundle=control
```

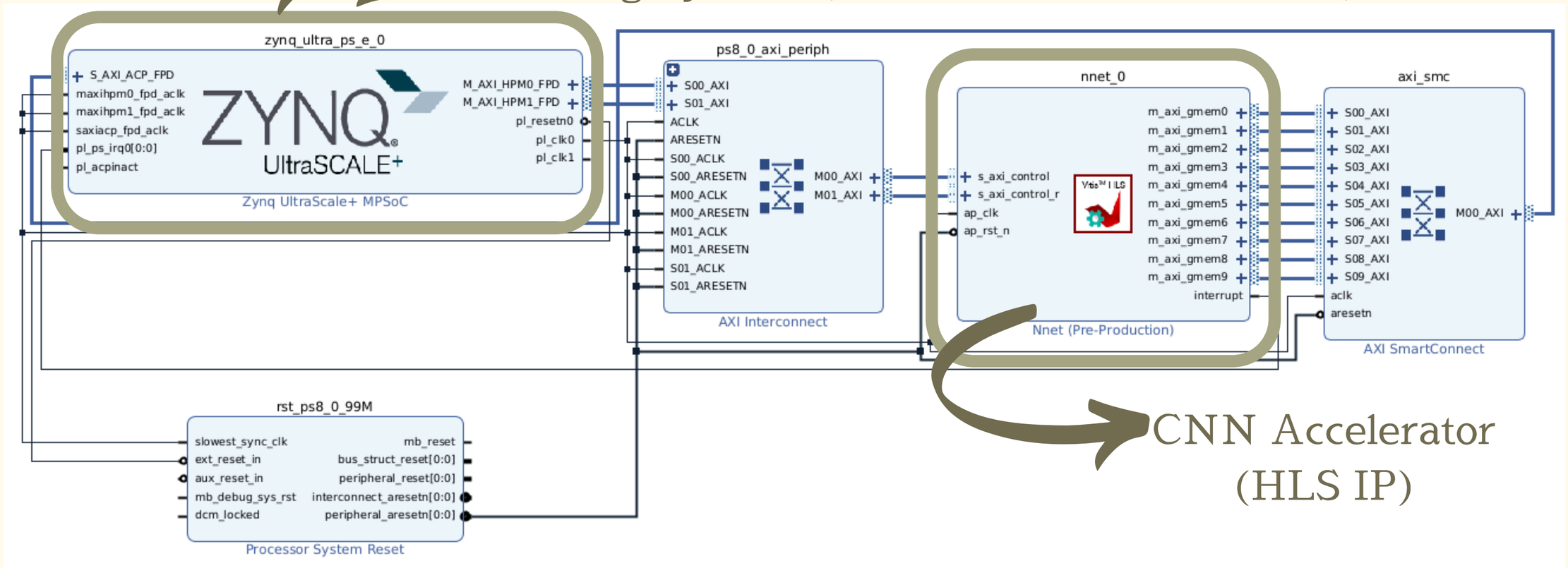
I/O Interface of Top Module

- Each port has access to specific data (like weights, bias etc.,)
- All the ports are memory-mapped.

HLS pragmas describing the AXI port mappings for all the I/O signals

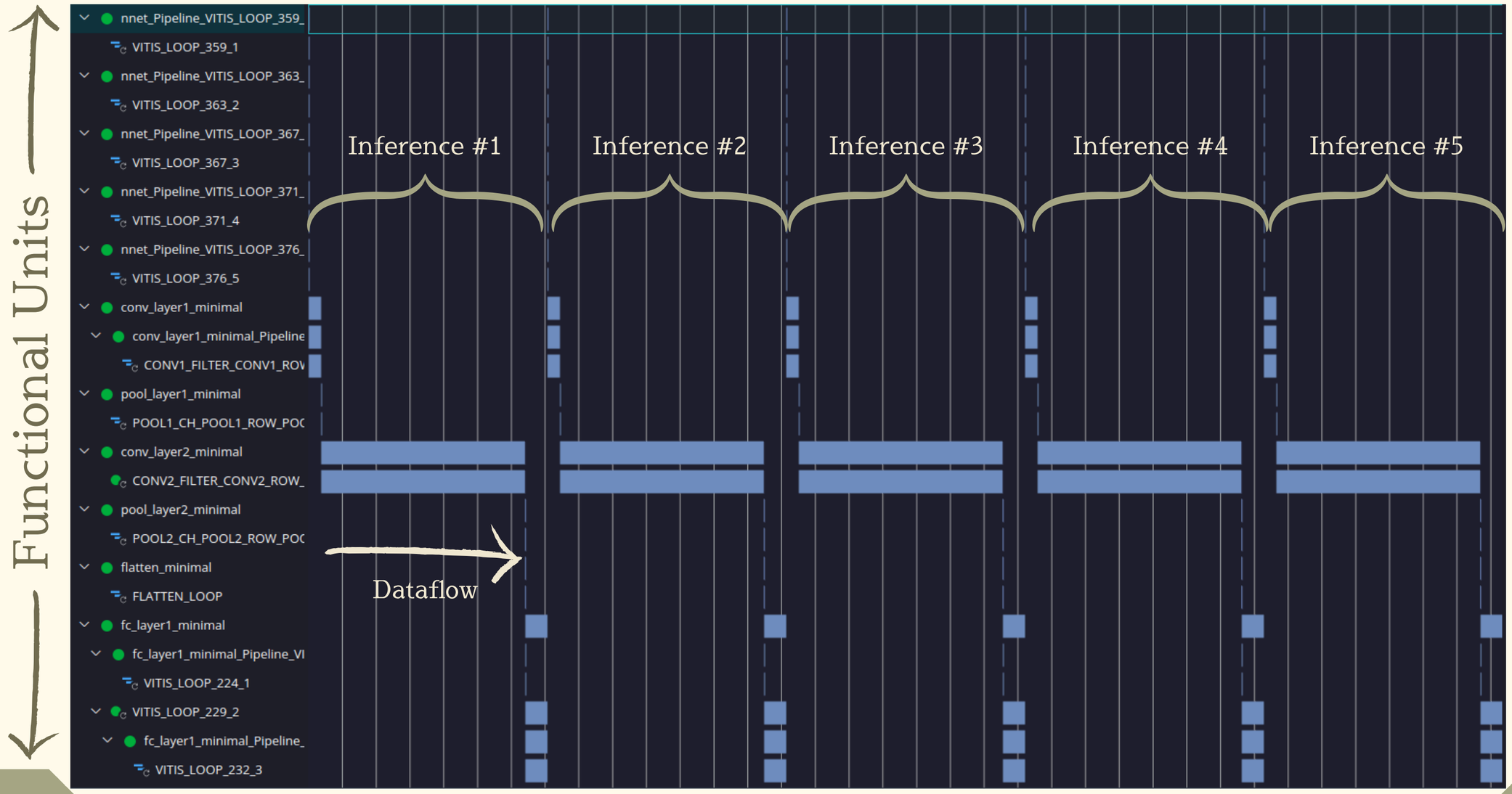
FPGA Block Design

Processing System (4 ARM Cortex A53 Cores)



CNN Accelerator (HLS IP)

C/ RTL Co-Simulation Trace View



FPGA Setup


```
GTKTerm - /dev/ttyUSB1  115200-8-N-1
File  Edit  Log  Configuration  Control signals  View  Help

Step 2: Reset accelerator...
  Initial status: 0x00000004
  Already IDLE

Step 3: Configure accelerator addresses...
  Using s_axi_control_r at 0xA0010000
  Setting image: 0x04000000 -> offset 0x10
    ✓ Verified: 0x04000000
  Setting conv1_weights: 0x08000000 -> offset 0x20
    ✓ Verified: 0x08000000
  Setting conv1_bias: 0x10000000 -> offset 0x28
    ✓ Verified: 0x10000000
  Setting conv2_weights: 0x18000000 -> offset 0x38
    ✓ Verified: 0x18000000
  Setting conv2_bias: 0x20000000 -> offset 0x40
    ✓ Verified: 0x20000000
  Setting fc1_weights: 0x28000000 -> offset 0x50
    ✓ Verified: 0x28000000
  Setting fc1_bias: 0x30000000 -> offset 0x58
    ✓ Verified: 0x30000000
  Setting fc2_weights: 0x38000000 -> offset 0x68
    ✓ Verified: 0x38000000
  Setting fc2_bias: 0x40000000 -> offset 0x70
    ✓ Verified: 0x40000000
  Setting predictions: 0x48000000 -> offset 0x80
    ✓ Verified: 0x48000000

Step 4: Start accelerator...
  Writing START bit to control register at 0xA0000000
  Starting accelerator, waiting for completion...
    0 ms elapsed, status: 0x00000001
    200 ms elapsed, status: 0x00000001
  Accelerator completed, final status: 0x0000000E

/dev/ttyUSB1 115200-8-N-1                DTR  RTS  CTS  CD  DSR  RI
```

UART Execution Log

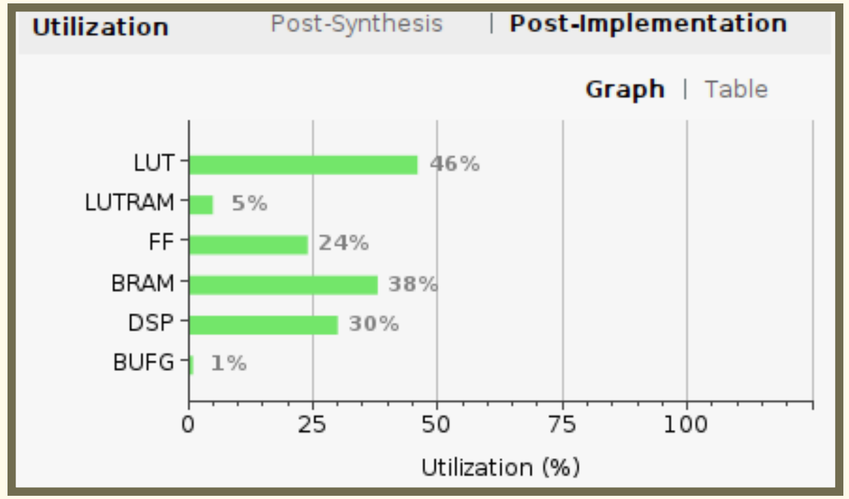
Results

Hardware Utilization | Timing | Power

Power — **Summary** | On-Chip

Total On-Chip Power:	**3.515 W**
Junction Temperature:	**33.2 °C**
Thermal Margin:	51.8 °C (22.0 W)
Effective ϑJA:	2.3 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report



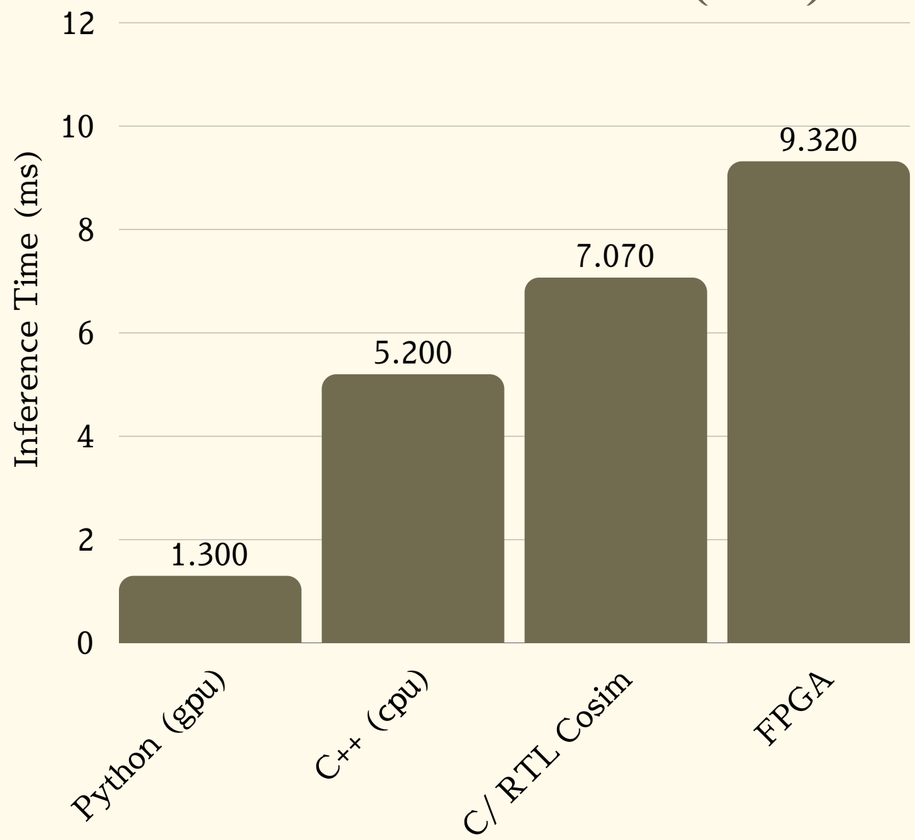
Design Timing Summary — Clock Frequency = 100 MHz

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.619 ns	Worst Hold Slack (WHS):	0.010 ns	Worst Pulse Width Slack (WPWS):	3.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	206779	Total Number of Endpoints:	206779	Total Number of Endpoints:	62929

All user specified timing constraints are met.

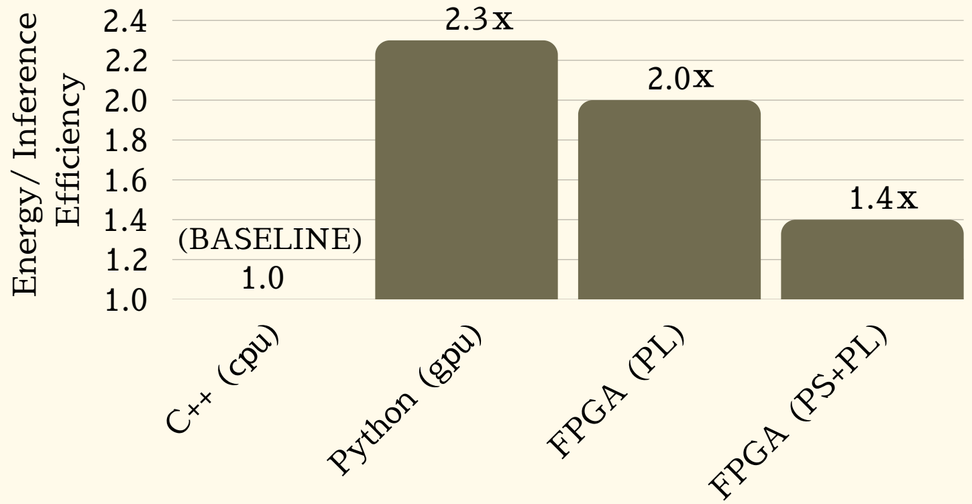
Execution Metrics

Inference Time (ms)



C/ RTL cosimulation execution time is in hours. The table illustrates the inference time from the timing diagram.

Energy Consumption/ Inference



Implementation	Power	Inference Time	Energy per Inference
CPU (i7-13700H)	~9W	7.2ms	64.8 mJ
GPU (RTX 4050)	~22W	1.3ms	28.6 mJ
FPGA (KV260)	3.5W (PL only)	9.32ms	32.6 mJ
FPGA (KV260)	~5.0W (PL+PS)	9.32ms	46.6 mJ

CPU & GPU power estimates are provided based on:

- Core Utilization
- Thermal Design Power (TDP) Rating

Learnings | Challenges

- Some of our biggest challenges included:
  - Adjusting the floating point design so it fits our board
  - Debugging the C++ code to generate the correct RTL for our board
  - Downsizing the architecture as compared to the research paper due to the complexity and resources
- Lessons learned:
  - Various loop optimization/deep learning acceleration techniques (loop unrolling, tiling, reordering)
  - Designing HLS compatible C++ code and utilizing AMD Vitis

AI Usage

- Primarily for Code Assistance:
  - C++ HLS function templates
  - Test case design
  - Debugging C++ code

- Used to plan out a timeline for the project
- Explained topics in research papers
- Assisted in the initial draft of our project write-up

THANK YOU