Designing Components for Convolutional Neural Networks on FPGA Platform

By

Aditya Agrawal (CS19B1003)

Jatin Sachdeva (CS19B1013)

Abhijeet Wankhade (CS19B1028)

Guided By,

Dr. Debasish Mukherjee

Content

- Introduction
- Summary and Results of Previous Work
- Problem Statement
- Solution using Multi-Channel Architecture
- Experiment Setup
- Experiment
- Demonstration
- Results
- Challenges Faced
- Conclusions
- References

Introduction

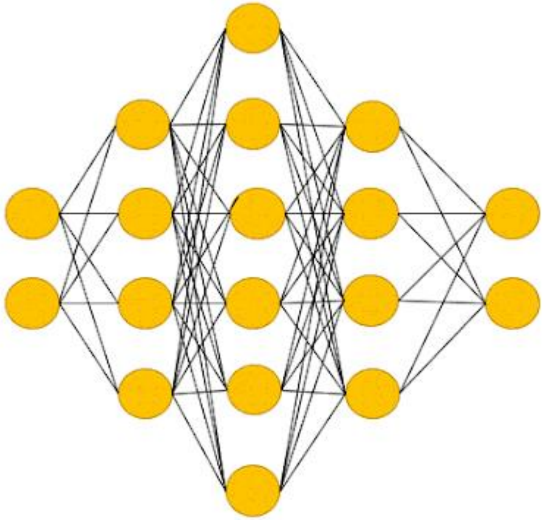
- We discuss the working of convolution neural networks and processes and techniques involved in it and how to improve upon them.

- We implement the CNNs on the CPU and FPGA platform and compare its performance among the CPU, GPU and FPGA implementations.

- We also cover the importance of CNNs and our implementation for practical purposes and their applications.

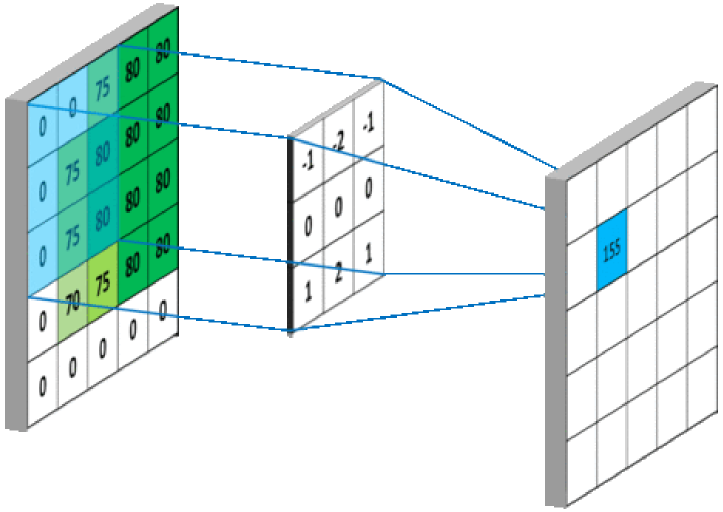
Neural Networks & Convolution Neural Networks

- Neural networks are computing systems inspired by the biological neural networks that constitute animal brains.
- Neural networks are widely used to mimic the human brain to find patterns in the data we receive.
- Convolution Neural Networks are extension of NNs where a convolution layer is part of the overall neural network.
- **Goal**: We intend to design a convolution, pooling and dense layer for the project purposes.



A fully connected layer

(Source: blogspot.com)

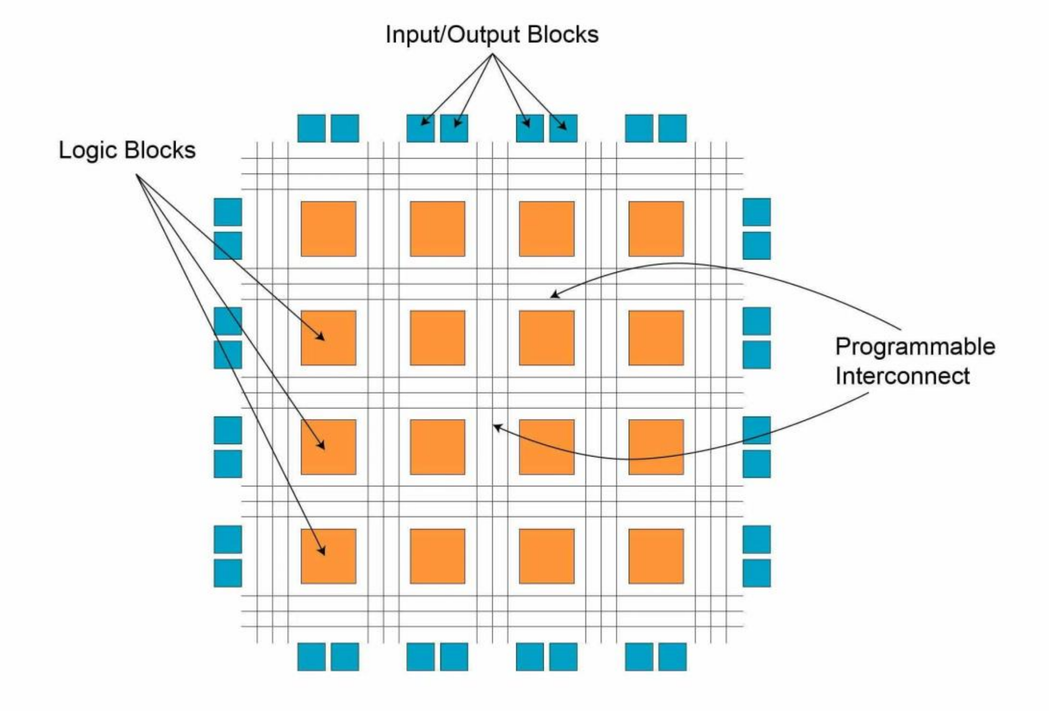


Convolution Operation

(Source: pressablecdn.com)

FPGA (Field-Programmable Gate Array)

- It is a semiconductor integrated circuit that can be programmed or reprogrammed by the user to perform variety of tasks.
- FPGAs are composed of programmable logic blocks like logic gate and flip-flops, interconnects, and input/output blocks.
- Programmable logic cells in FPGAs can be programmed using Hardware Description Languages (HDLs) such as Verilog or VHDL.
- Generally used to achieve high performance and low latency.

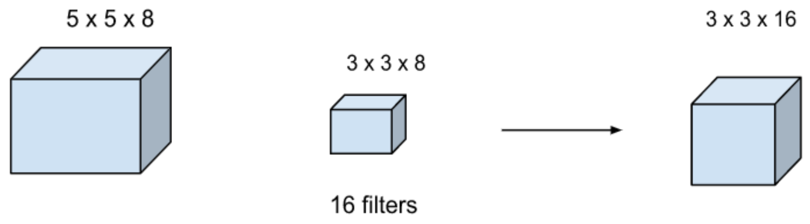


FPGA Architecture

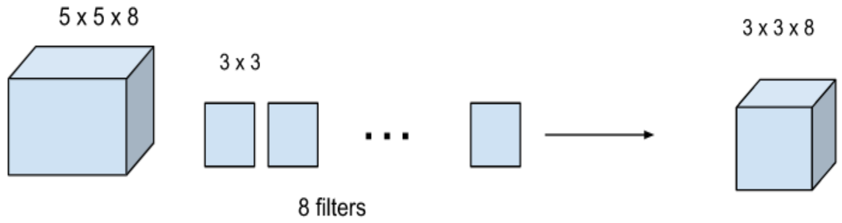
(Source: logic-fruit.com)

Literature Review

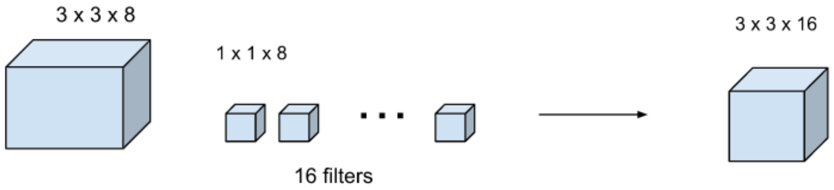
Title	Approach	Demerit
[1] A CNN Accelerator on FPGA Using Depth-wise Separable Convolution	Using Depth-wise Convolution and Point-wise Convolution operations	Complex spatial information is lost, complex hardware implementation
[2] Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster	Uses Multiple FPGAs for different layers of CNN	Increase cost and communication overhead
[3] Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA	Uses loop optimizing technique like loop unrolling, loop tiling and loop interchange	Inflexible or unscalable, needs to decide optimizing parameter for different model
[4] Fused-Layer CNN Accelerator	Conv. layers in series pipelined together to avoid writing the output and then reading again reducing off-chip mem	Only applicable to convolution layers in series



Multiplication = (3 x 3 x 16) x (3 x 3 x 8) = 10368



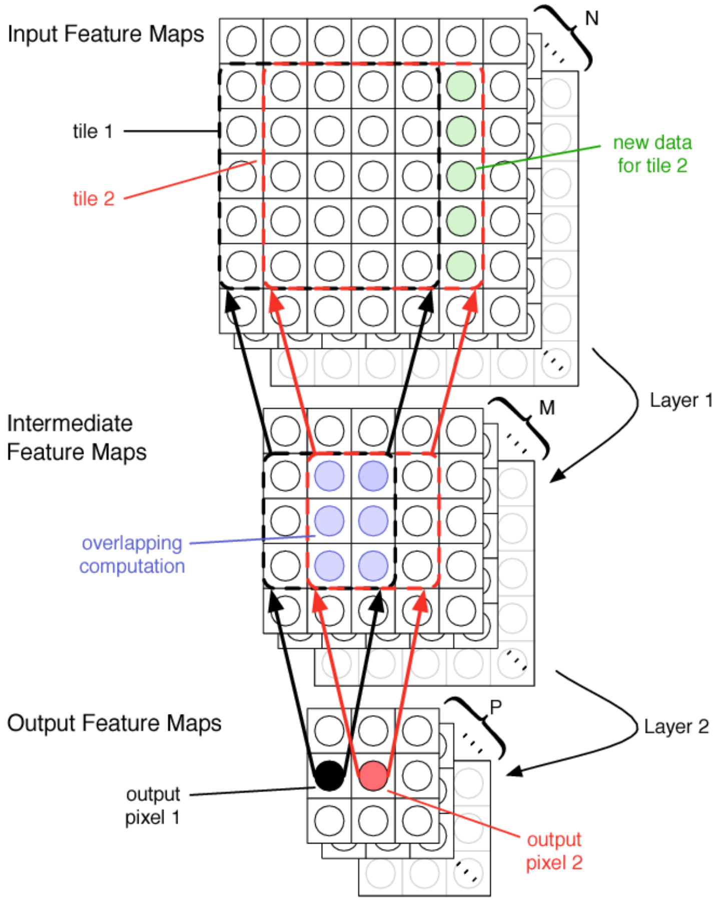
Multiplication = (3 x 3 x 8) x (3 x 3) = 648



Multiplication = (3 x 3 x16) x (1 x 1 x 16) = 2304

Total = 2304 + 648 = 2952

Source: [1]



Source: [4]

Summary of Previous Work

Previous implementations of neural networks generally focus on the factors like:

- Trying distinguished platforms: (FPGA, GPU, TPU)
- Memory optimization: (pipelining multiple conv layers, caching, using fixed points, using on chip or off chip memory)
- Using different convolution and synthesis approaches: (Depth-wise separable convolution, binarized weights)

Overall, they try to optimize the memory usage, compute heavy operation and implementation approaches.

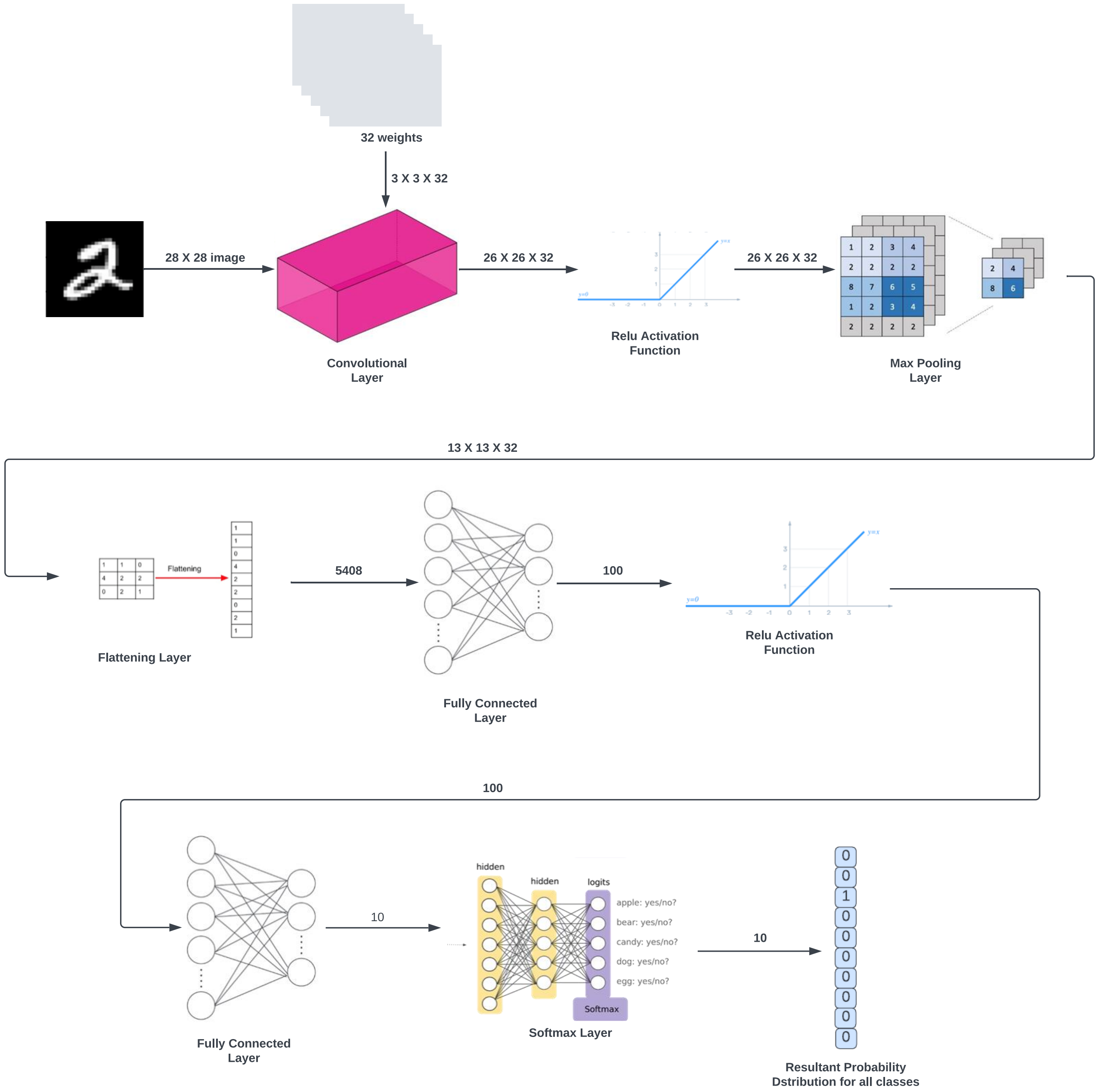
Problem Statement

- Can the implementation of CNNs on FPGA provide a more energy-efficient and high-performance solution?

- How do parameters like clock frequency, parallelism, and memory affect the performance of CNNs on FPGA, and how can they be optimized for efficient implementation?

- What are the challenges and opportunities of implementing CNNs on FPGA, and how can they be addressed?

CNN Architecture



32 weights

3 X 3 X 32

28 X 28 image

Convolutional Layer

26 X 26 X 32

Relu Activation Function

26 X 26 X 32

Max Pooling Layer

13 X 13 X 32

Flattening Layer

5408

Fully Connected Layer

100

Relu Activation Function

100

Fully Connected Layer

10

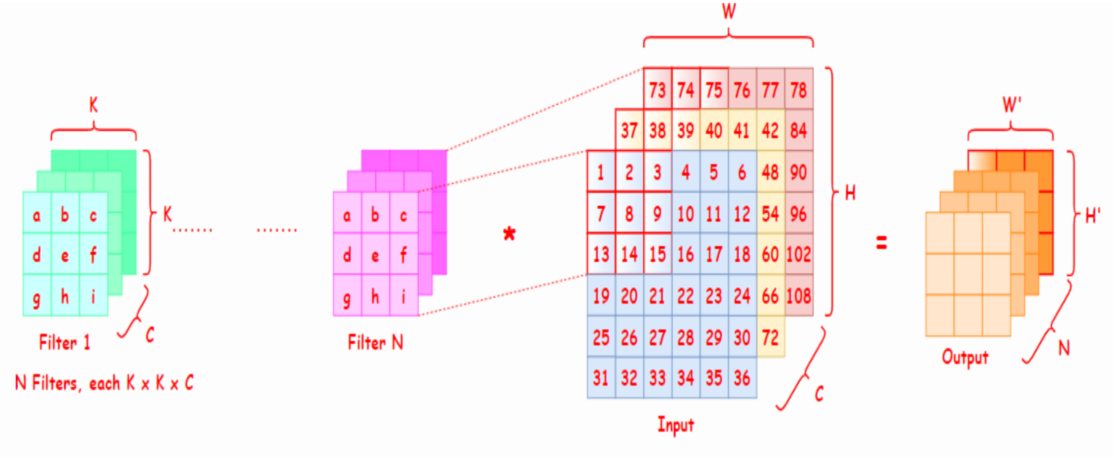
Softmax Layer

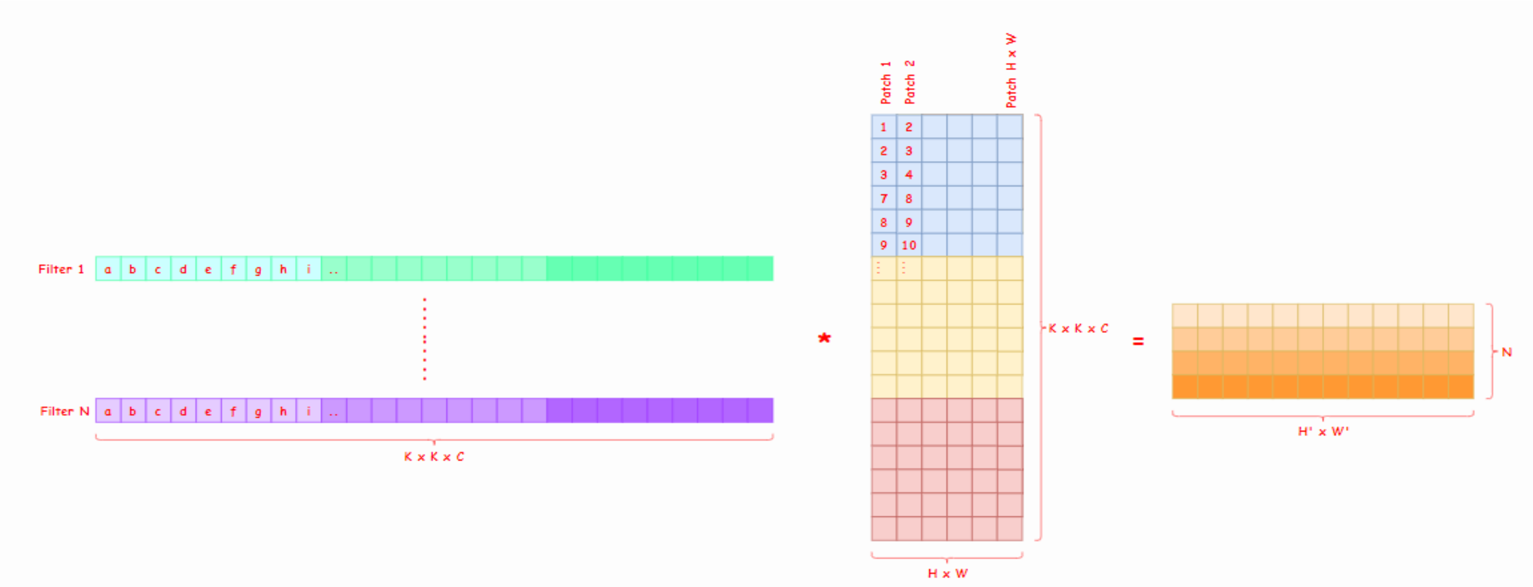
10

Resultant Probability Dstribution for all classes

Different CPU Implementation of Convolution Layer

- Using Loops
- Using NumPy Library
- Generalized Matrix Multiplication Problem





Source: https://scocoyash.github.io/speeding-up-convolutions/

Hardware Implementation

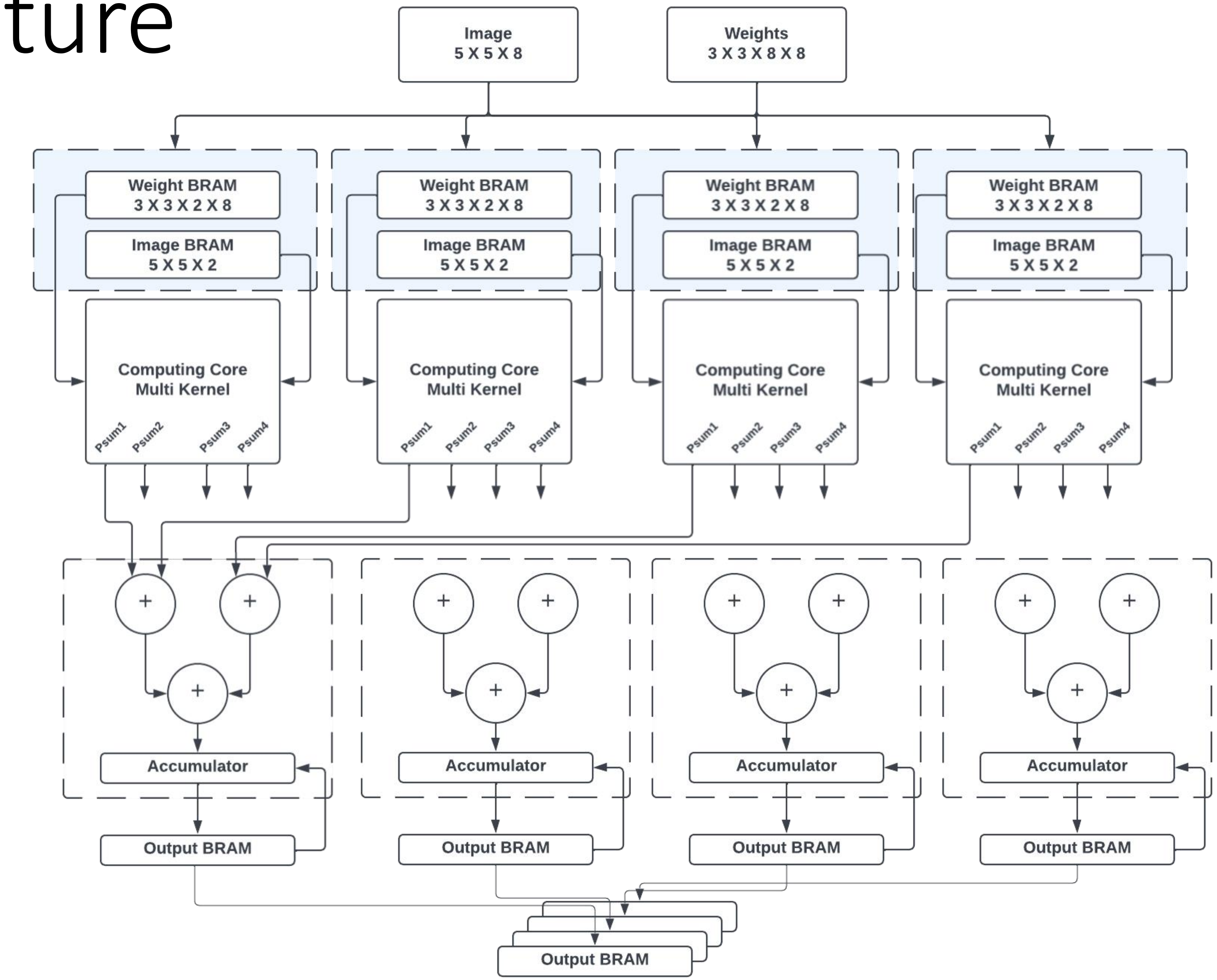
Our Solution using Multi-Channel Architecture

- Improving performance through hardware requires careful consideration of the amount of parallelization we can do and the hardware resources we can use.
- Channel and Kernel-wise computations together can be fragmented into batches.
- Generally, we are likely to see multiples of 2 and 4 in the shape/size of the inputs, weights, and biases in various CNN models like Alex-Net, VGG, etc., so here we'll divide the work and do parallel tasks in a batch of 4 or its multiple.

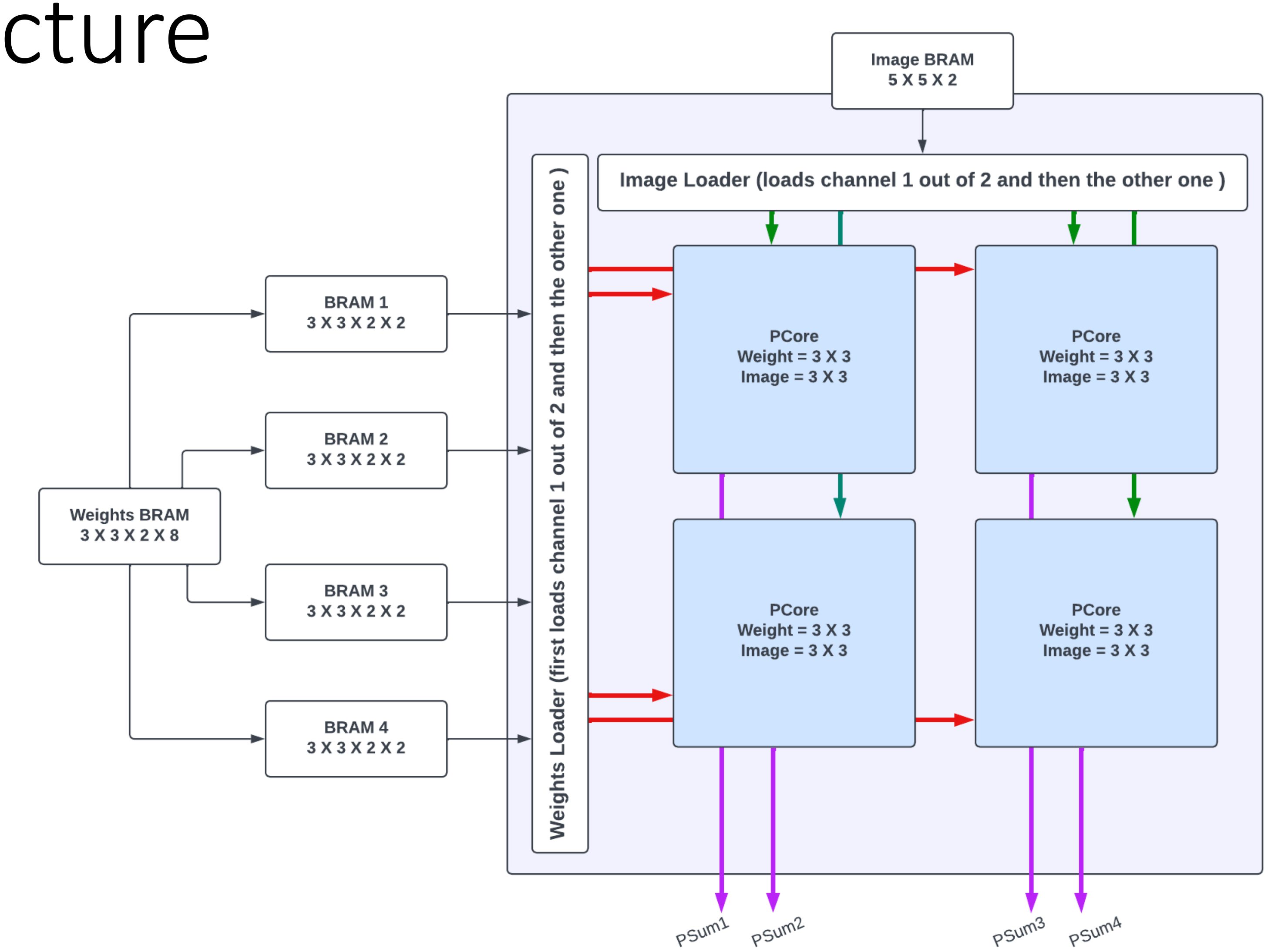
Convolution Layer

- Used to capture local features or patterns in tensor data (2D/3D) using multiple filters. We divide the channels into 4 which are applied to 4 kernels parallelly.
- Following are the components used:
  - BRAMs : Store and input and output data
  - Loaders and Address Generators : Load data at various level (Activations, Weights)
  - Computing Core : Core computing unit to multiply weights and activations in batch of 4. Total 4 comp. cores are there.
  - Accumulator : Adds and Stores results generated by Computing Core.
  - Pipeline : Controls the flow of other components.
- Overall, we achieve a speedup of **193.6x** compared to CPU NumPy.

Convolution Layer with Multi Channel Architecture



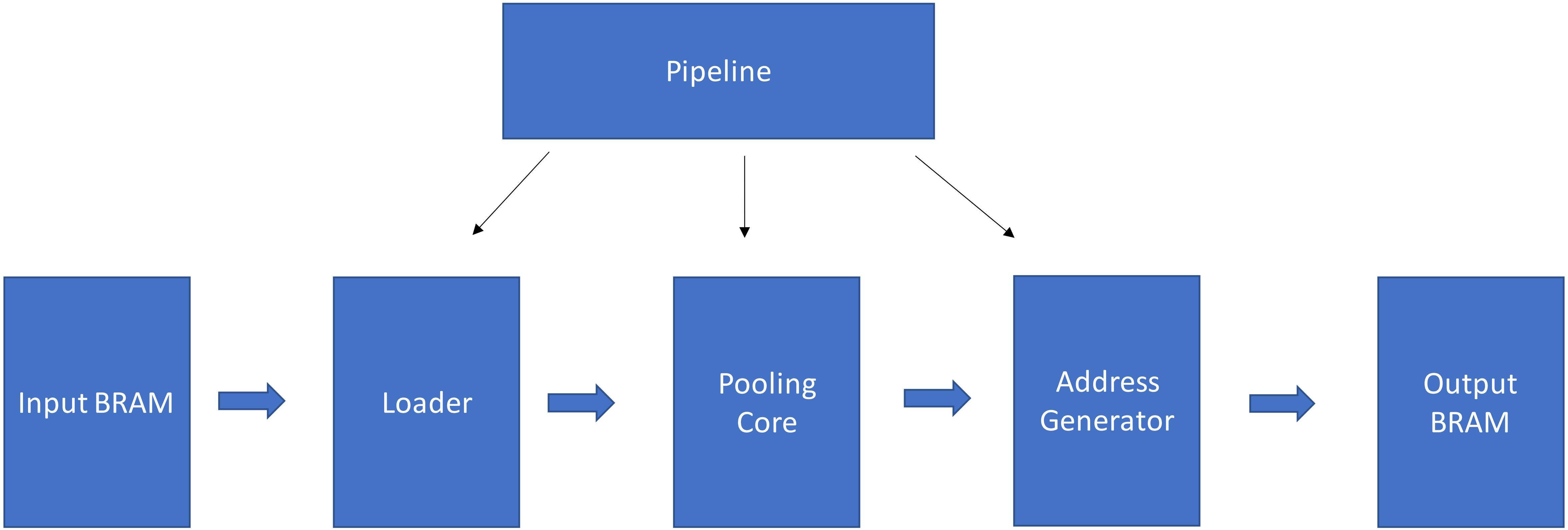
Computing Core with Multi Kernel Architecture



Pooling Layer

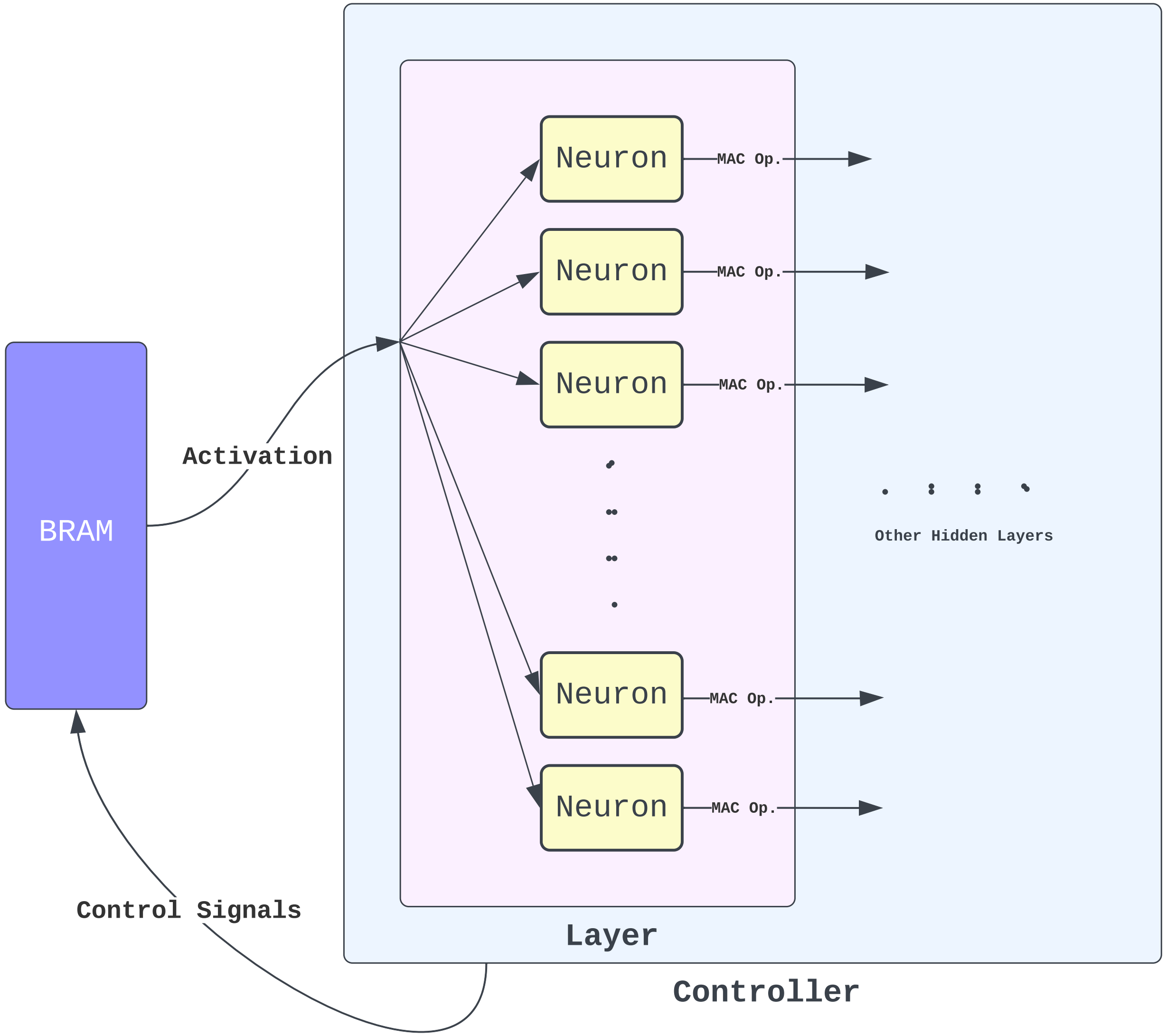
- Used to reduce spatial dimension of feature maps to diminish total computation and make the network more robust to small variations.
- Following are the components used:
  - BRAMs : Store Input and output data
  - Loaders and Address Generators : Load and stores data from previous layers.
  - Pooling Core : Core unit to take maximum out of a window.
  - Pipeline : Controls the flow of other components.
- Overall, we achieve a speedup of **138.8x** compared to CPU.

Pooling Layer



Fully Connected Layer

- Layer in which all the input features are connected to all output features with associated weights to identify the activation/relation strength b/w them. We parallelize all the output feature computations.
- Following are the components used:
  - Neurons : Store output data
  - Layer : Combines multiple neurons together.
  - Controller : Loads data from previous layer and pass it on to the layers. For multiple internal layers, controllers the loading from the outside previous layer and internal layers.
- Overall, we achieve operations per second = **0.2*n** (n = output classes)



Dense Layer Architecture

Experiment Setup

CPU Device Configuration

We've used the CPU with following configuration for testing:

- Processor: Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
- Cores: 8 Logical Cores
- Memory: 8GB Memory
- Operating System: Windows 11 - 64bit

FPGA Device Configuration

We've used Kintex - 7 Product Family, xc7k70t fbg676-1 (simulation) with following specs:

- I/O Pin: 676
- IOBs: 300
- LUT Element: 41000
- Flip Flops: 82000
- Block RAMs: 135
- DSPs: 240
- BUFGs: 32

Evaluation Parameters

We've evaluated the following parameters for each implemented component:

- Hardware Usage
- Power Usage
- Time Analysis

Demonstration

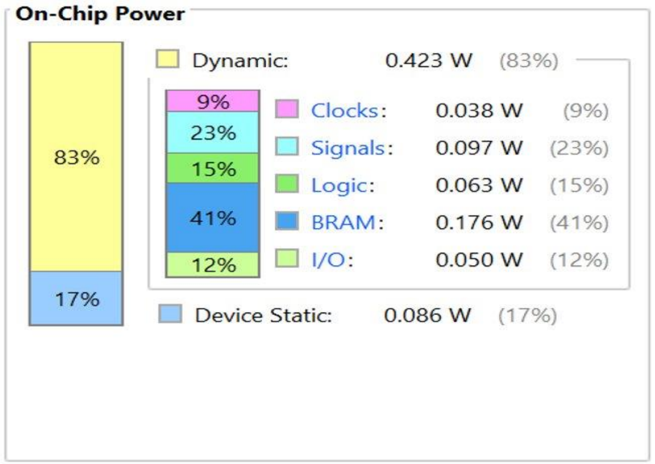
Results

Experiment

- We test the convolution layer with 8 filters of size 3 x 3 x 8 on different input sizes: 28 x 28 x 8, 64 x 64 x 8, 128 x 128x 8, 256 x 256 x 8, 512 x 512 x 8
- We test the pooling layer with 2 x 2 window with stride 1 on different input sizes: 28 x 28 x 2, 64 x 64 x 2, 128 x 128 x 2 , 256 x 256 x 2, 512 x 512 x 2
- We test the dense layer with different input x output class combinations: 1000 x 6, 100 x 60, 5408 x 100, 5408 x 10

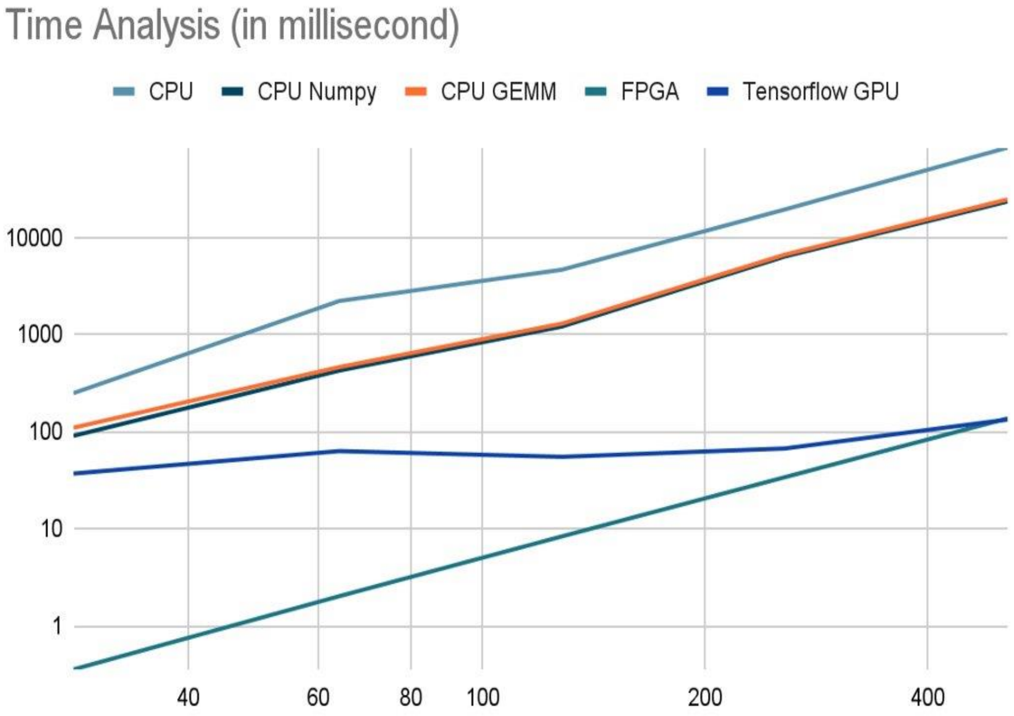
We perform these operations on CPU, TensorFlow GPU and our FPGA implementation.

Convolution Layer



On-Chip Power

Dynamic:	0.423 W	(83%)
Clocks:	0.038 W	(9%)
Signals:	0.097 W	(23%)
Logic:	0.063 W	(15%)
BRAM:	0.176 W	(41%)
I/O:	0.050 W	(12%)
Device Static:	0.086 W	(17%)


Time Analysis (in millisecond)

Resource	Utilization	Available	Utilization %
LUT	9093	41000	23.19
FF	6673	82000	8.24
BRAM	72	135	53.33
IO	285	300	95

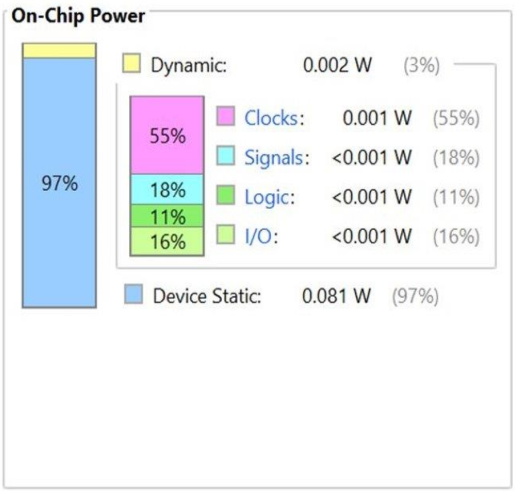
Convolution Layer

Time Analysis

	28	64	128	256	512
CPU	249.17ms	2212.24ms	4647.57ms	19358ms	83982.3ms
CPU NumPy	90.5ms	425.143ms	1209.20ms	6351.789ms	23428.614ms
CPU GEMM	109.76ms	461.84ms	1293.155ms	6635.371ms	24582.72ms
FPGA	0.3562ms	2.02ms	8.344ms	33.909ms	136.708ms
TensorFlow GPU	37ms	63ms	55ms	67ms	133ms

FPGA Speedup from CPU NumPy = **193.6x**

Pooling Layer



On-Chip Power

Dynamic: 0.002 W (3%)

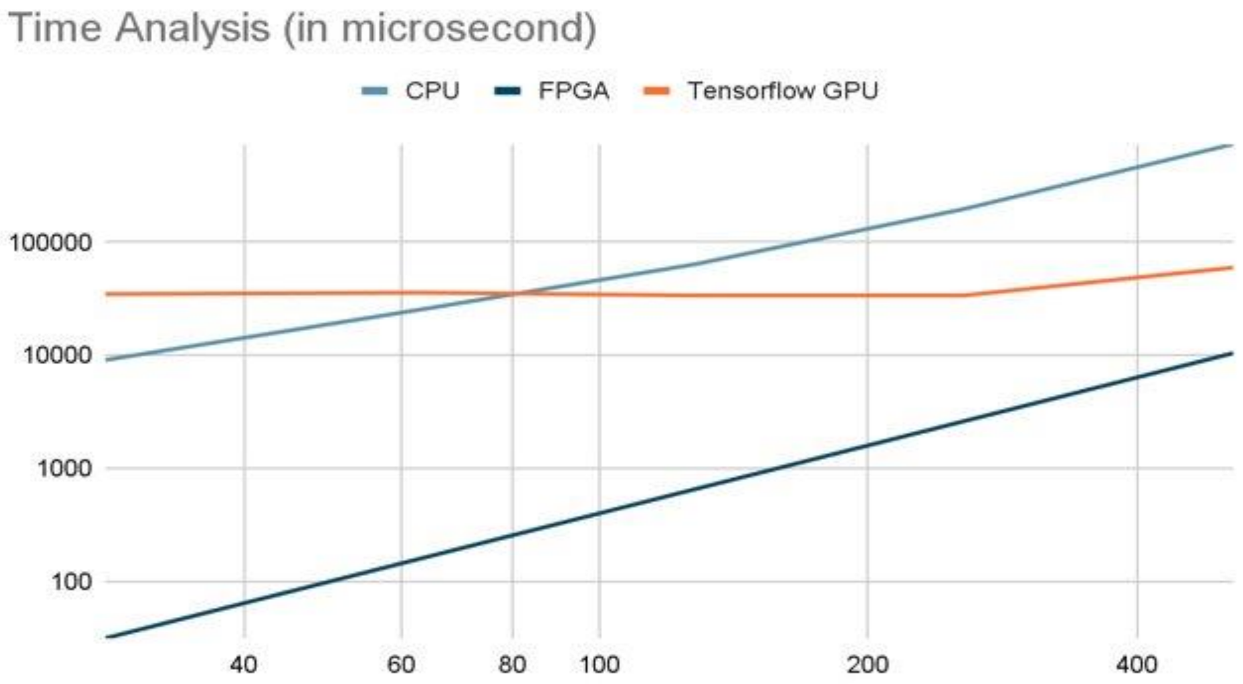
Clocks: 0.001 W (55%)

Signals: <0.001 W (18%)

Logic: <0.001 W (11%)

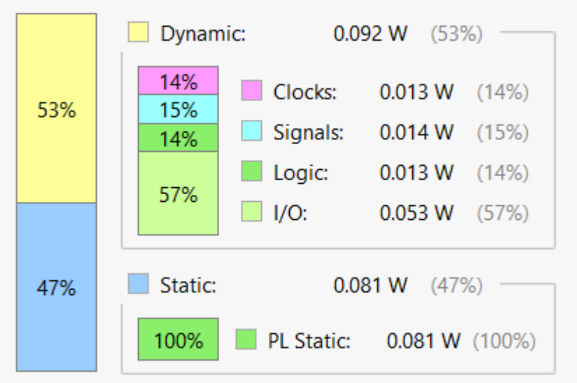
I/O: <0.001 W (16%)

Device Static: 0.081 W (97%)

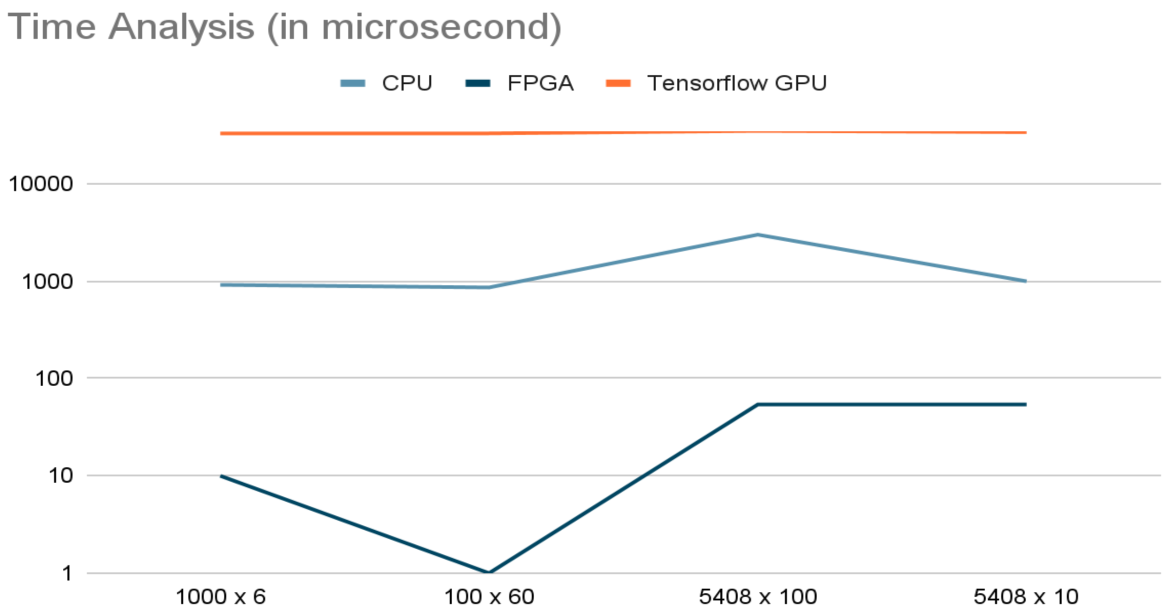


Resource	Utilization	Available	Utilization %
LUT	49	41000	0.12
FF	47	82000	0.06
IO	34	300	11.33
BRAM	10	135	7.41

Dense Layer



🟨 Dynamic:	0.092 W	(53%)	
🟪 Clocks:	0.013 W	(14%)	
🟦 Signals:	0.014 W	(15%)	
🟩 Logic:	0.013 W	(14%)	
🟩 I/O:	0.053 W	(57%)	
🟦 Static:	0.081 W	(47%)	
🟩 PL Static:	0.081 W	(100%)	



Resource	Utilization	Available	Utilization %
LUT	704	41000	1.72
FF	743	82000	0.91
IO	267	300	89.00
BUFG	1	32	3.13

Theoretical Throughput

We'll now try to calculate time to do classification for MNIST dataset

(28, 28, 4) -> CNN (26, 26, 32) -> Pooling (13, 13, 32) -> Dense (100, 1) -> Dense (10, 1)

0.7124ms + 0.06272ms + 0.05409ms + 0.00101ms

Hence, **0.83022ms** per classification.

Or in other terms: **1,204.53fps**

*Ignored time for SoftMax

Applications

- These components, stacked together can be used to form a full convolution neural network that has vast potential in terms of image pattern recognition.
- Due to the possibility of making an actual hardware device out of it and optimal performance, these components can be used across real-time applications like traffic monitoring, health monitoring, and other places.
- As the hardware utilization is minimized vastly, the manufactured hardware will be small and space efficient. This is especially useful in applications like self-driving cars, robotics applications and other fields.

Challenges Faced

- FPGAs have limited resources compared to modern CPUs or GPUs. Hence, optimizing the network design to fit within the available resources is critical. Limited resource availability forced us to not integrate the components together.
- CNNs are highly algorithmically complex, and optimizing the network for implementation on FPGAs requires careful consideration of the hardware resources available.
- Model training includes extensive calculations like differential equations and others which makes it harder to implement on hardware.
- As the size of the CNN network grows, the scalability of the implementation on FPGA platforms can become a challenge.
- FPGAs can consume significant power, especially when running complex computations like CNNs. This can limit their use in low-power devices, and optimizing the design for low power consumption is a challenge.

Conclusions

- We implemented scalable and customizable components of convolutional neural networks.
- The convolution layer was able to work on a frequency of 137MHz, while the pooling and fully connected layers ran at 166MHz and 200MHz respectively. When combined, the system ran at a frequency of 137MHz.
- On FPGA, we achieved **5 GOPS** on a single core for convolution layer (**193.6x** CPU), **0.02075 GOPS** in Pooling (**138.8x** CPU) and **0.2*n GOPS** in the dense layer, where n is the number of output classes.
- The total power consumption of the design was **0.765 W**, making it a power-efficient design.

Future Work

- We can generalize the size of the kernel for convolution layer.
- Line buffers can be used in Loaders to improve time required for them.
- We can use external memory along with data streaming to reduce on-chip memory usage.
- Pipelining the three layers (convolution, pooling, and fully connected) to improve performance further using multiple FPGAs.

Contributions

- Aditya: Fully connected Layer on FPGA, NumPy CPU, Dense Layer CPU, Accumulator
- Abhijeet: Computing core on FPGA, Load Weight
  and CTRL on FPGA, BRAM selector on FPGA,
  Convolution using NumPy, SoftMax, ReLU and Flattening Layers on CPU
- Jatin: Convolution as GEMM on CPU, Pooling Layer on CPU, Pooling Layer on FPGA, Load Activation, Pipeline

References

- [1] Bai, Lin, Yiming Zhao, and Xinming Huang. "A CNN accelerator on FPGA using depthwise separable convolution." IEEE Transactions on Circuits and Systems II: Express Briefs 65.10 (2018): 1415-1419.
- [2] Zhang, Chen, et al. "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster." Proceedings of the 2016 International Symposium on Low Power Electronics and Design. 2016.
- [3] Ma, Yufei, et al. "Optimizing the convolution operation to accelerate deep neural networks on FPGA." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 26.7 (2018): 1354-1367.
- [4] Alwani, M., Chen, H., Ferdman, M. and Milder, P., 2016, October. Fused-layer CNN accelerators. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (pp. 1-12). IEEE.

Thank You