UC San Diego

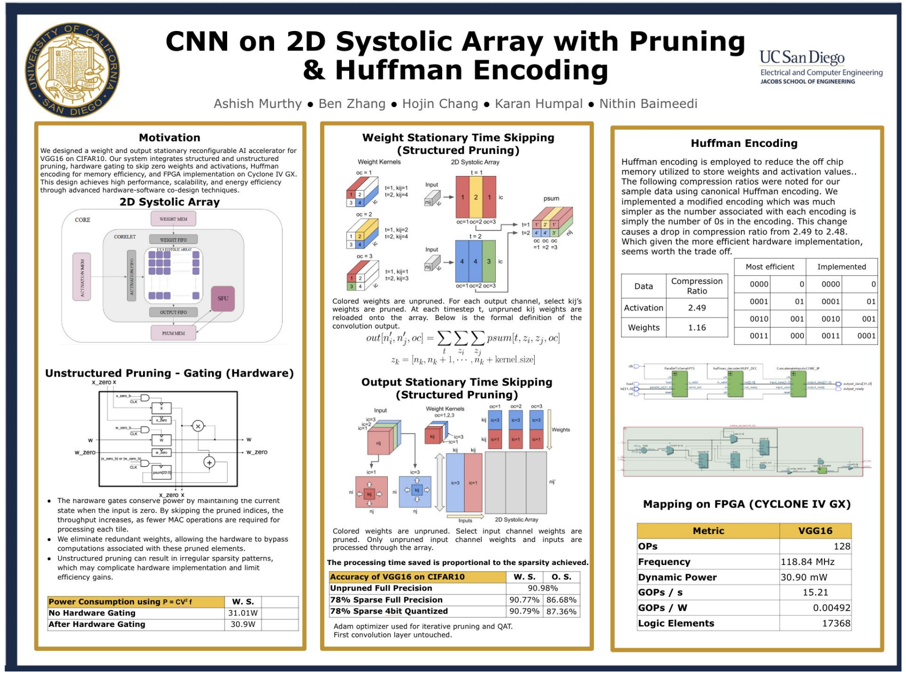
CUDA-Accelerated 2D Systolic Array for Low-Bit Quantized Matrix Multiplication

Ben Zhang

ECE 277 Project

Project Motivation:

- Originated from an ASIC project for CNN acceleration using a 2D systolic array.
- Aim: Port and extend the hardware accelerator concept to CUDA for deep learning inference.
- Low-bit (2-bit) quantized neural networks reduce computational and memory requirements.
- Importance of efficient matrix multiplication in CNN inference.



CNN on 2D Systolic Array with Pruning & Huffman Encoding

UC San Diego
Electrical and Computer Engineering
JACOBS SCHOOL OF ENGINEERING

Ashish Murthy • Ben Zhang • Hojin Chang • Karan Humpal • Nithin Baimeedi

Motivation

We designed a weight and output stationary reconfigurable AI accelerator for VGG16 on CIFAR10. Our system integrates structured and unstructured pruning, hardware gating to skip zero weights and activations, Huffman encoding for memory efficiency, and FPGA implementation on Cyclone IV GX. This design achieves high performance, scalability, and energy efficiency through advanced hardware-software co-design techniques.

2D Systolic Array

Unstructured Pruning - Gating (Hardware)

- The hardware gates conserve power by maintaining the current state when the input is zero. By skipping the pruned indices, the throughput increases, as fewer MAC operations are required for processing each tile.
- We eliminate redundant weights, allowing the hardware to bypass computations associated with these pruned elements.
- Unstructured pruning can result in irregular sparsity patterns, which may complicate hardware implementation and limit efficiency gains.

Power Consumption using $P = CV^2 f$	W. S.	
No Hardware Gating	31.01W	
After Hardware Gating	30.9W	

Weight Stationary Time Skipping (Structured Pruning)

Colored weights are unpruned. For each output channel, select kij's weights are pruned. At each timestep t, unpruned kij weights are reloaded onto the array. Below is the formal definition of the convolution output.

$$out[n_i', n_j', oc] = \sum_t \sum_{z_i} \sum_{z_j} psum[t, z_i, z_j, oc]$$

$$z_k = [n_k, n_k + 1, \cdots, n_k + \text{kernel\_size}]$$

Output Stationary Time Skipping (Structured Pruning)

Colored weights are unpruned. Select input channel weights are pruned. Only unpruned input channel weights and inputs are processed through the array.

The processing time saved is proportional to the sparsity achieved.

Accuracy of VGG16 on CIFAR10	W. S.	O. S.
Unpruned Full Precision	90.98%	
78% Sparse Full Precision	90.77%	86.68%
78% Sparse 4bit Quantized	90.79%	87.36%

Adam optimizer used for iterative pruning and QAT.
First convolution layer untouched.

Huffman Encoding

Huffman encoding is employed to reduce the off chip memory utilized to store weights and activation values.. The following compression ratios were noted for our sample data using canonical Huffman encoding. We implemented a modified encoding which was much simpler as the number associated with each encoding is simply the number of 0s in the encoding. This change causes a drop in compression ratio from 2.49 to 2.48. Which given the more efficient hardware implementation, seems worth the trade off.

Data	Compression Ratio
Activation	2.49
Weights	1.16

Most efficient		Implemented	
0000	0	0000	0
0001	01	0001	01
0010	001	0010	001
0011	000	0011	0001

Mapping on FPGA (CYCLONE IV GX)

Metric	VGG16
OPs	128
Frequency	118.84 MHz
Dynamic Power	30.90 mW
GOPs / s	15.21
GOPs / W	0.00492
Logic Elements	17368

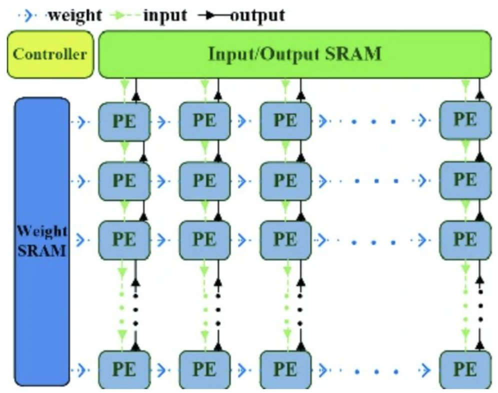
Algorithm

2D Systolic Array Simulation:

- A parallel processing architecture where processing elements (PEs) are arranged in a two-dimensional grid.
- PEs exchange data only with adjacent neighbors.
- Operations are overlapped to maximize throughput.
- Uses local shared storage to reduce global memory accesses.

Low-Bit Quantization:

- Input matrices stored as 8-bit integers but only lower 2 bits are used.
- Optimized arithmetic for quantized data to simulate low-bit operations.



https://link.springer.com/chapter/10.1007/978-3-030-05677-3_16

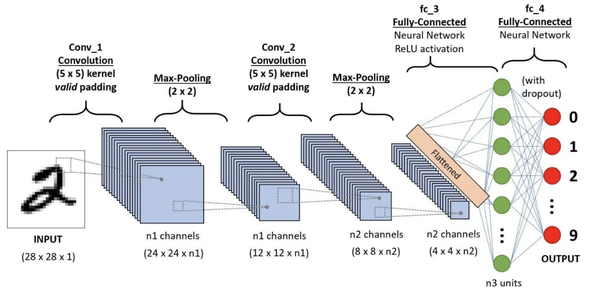
Limitations of Algorithm using CPU

- **Limited Parallelism**
  - CPUs offer significantly fewer cores compared to GPUs, reducing the level of parallel processing available for large-scale matrix multiplications.
- **Performance Bottlenecks**
  - Processing is done sequentially and only in one core, leading to lower throughput.
- **Real-Time Inference Constraints**
  - Achieving low latency and high throughput for deep learning applications is more challenging without the massive parallelism
- **Energy Efficiency**
  - More time to process usually leads to high power consumption

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

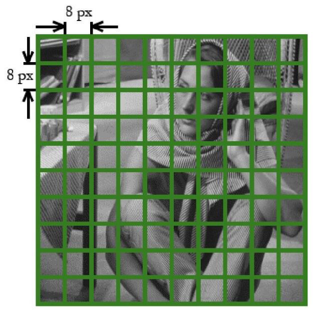
$$C = AB = \begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} & a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32} \\ a_{21} * b_{11} + a_{22} * b_{21} + a_{23} * b_{31} & a_{21} * b_{12} + a_{22} * b_{22} + a_{23} * b_{32} \end{bmatrix}$$

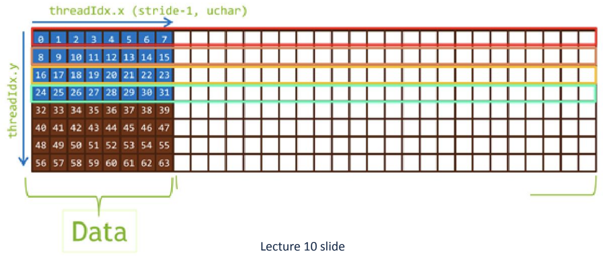


https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/

CUDA Implementation Details

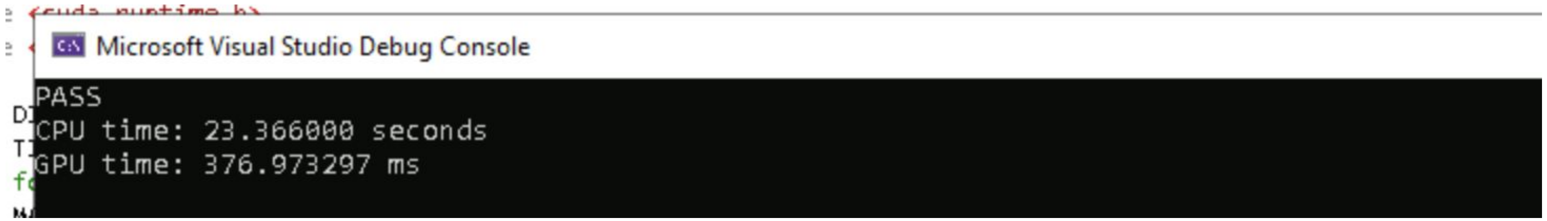
- **Memory Optimization:**
  - **Shared Memory:**
    - Tiled loading of matrices into shared memory for fast access.
    - Optionally implement a shared memory transpose to improve coalesced memory accesses.
  - **Global Memory Transpose:**
    - Row-wise read,Column-wise write
    - Store SMEM ot GMEM, load GMEM to SMEM
- **Kernel Launch Configuration:**
  - Block and grid dimensions tuned to matrix dimensions and tile size.
  - Use of CUDA events for GPU runtime measurement.
- **Data Preprocessing & Testbench:**
  - Random generation of quantized matrices.
  - CPU baseline implemented in C for result validation and performance benchmarking.





Lecture 10 slide

Demo


Microsoft Visual Studio Debug Console

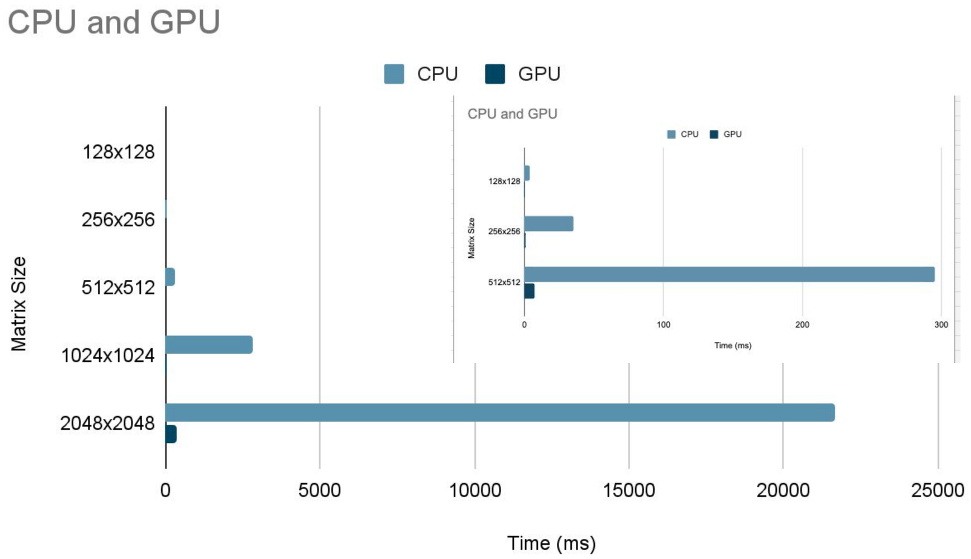
```
PASS
CPU time: 23.366000 seconds
GPU time: 376.973297 ms
```

Performance and Results

UC San Diego

- **Comparative Analysis:**
  - GPU vs. CPU implementation run times.
  - Any Matrix smaller than 128 x 128 will run instantly and CPU timer doesn't work.
  - Anything above 2048 x 2048 will take long time for CPU to execute
  - Validating correctness by comparing output arrays between CPU and GPU.
  - Input matrix are randomly generated with selected Dimension
  - Speedup = 1/(1-(CPU/GPU)) as assuming "infinite" number of parallel processors for GPU
- **Observations:**
  - Significant speed-up using CUDA, especially for larger-scale matrix multiplications.
  - Larger Matrix size will have better improvement in execution speed.

Matrix Size	CPU	GPU	Speedup
128x128	4	0.207872	94.80%
256x256	35	0.990208	97.17%
512x512	295	7.084032	97.60%
1024x1024	2836	54.031265	98.09%
2048x2048	22170	353.237	98.41%



Conclusion

- **Achievements:**
    - Successful design of a CUDA-accelerated systolic array for quantized matrix multiplication.
    - Demonstrated performance gains over CPU-based implementations.

- **Future Enhancements:**
    - Further memory and kernel optimizations , like use of concurrent kernel, multistream, and even async memory transfer .
    - Bit-packing multiple 2-bit values per byte for reduced memory footprint.

References

https://telesens.co/2018/07/30/systolic-architectures/

https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/

https://link.springer.com/chapter/10.1007/978-3-030-05677-3_16

https://docs.google.com/presentation/d/1m3y5f82RGP89rW5o2U_iYEZnltnKjtjVgIkgyb7-nYY/edit?usp=sharing

https://github.com/khumpal/ECE284-Final-Project

Lecture Slides