EIE: Efficient Inference Engine on Compressed Deep Neural Network

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, William J. Dally

Presented By Jiachen He, Ben Simpson, Jielun Tan, Xinyang Xu, Boyang Zhang

Intro/Motivation

- Large DNNs (Deep Neural Networks) powerful but consume a lot of energy
  - Energy consumption dominated by DRAM access if there is no data reuse;
  - no parameter reuse in fully connected (FC) layers in a convolutional neural network (CNN);
  - uncompressed modern DNNs are so large they must be placed on DRAM
  - Conclusion: large, uncompressed DNNs are not suitable for energy constrained applications
- SRAM access consumes much less energy than DRAM access
- Previous works focus on accelerating dense, uncompressed models, so if energy constrained can only use small models that fit on the on-chip SRAM
- Conclusion: Need to work on compressed models in order to be energy efficient

Intro/Motivation

- Processing compressed models with CPU/GPU:
  - Batching improves throughput at a cost of latency, not suitable for embedded applications
  - Irregular pattern of operation hinders effective accelaration
- This paper's contributions: EIE, an efficient inference engine
  - Dedicated accelerator
  - Processes efficiently on compressed models
  - Exploits the dynamic sparsity of activations to save computation
  - A method to parallelize a sparsified layer across multiple PEs
  - Evaluation

DNN Compression

- Fully Connected (FC) layer heavily involves matrix multiplication

$$b = f(W*a + v) = f([W\ v] * [a\ 1]^T)$$

- Pruning creates a sparse matrix

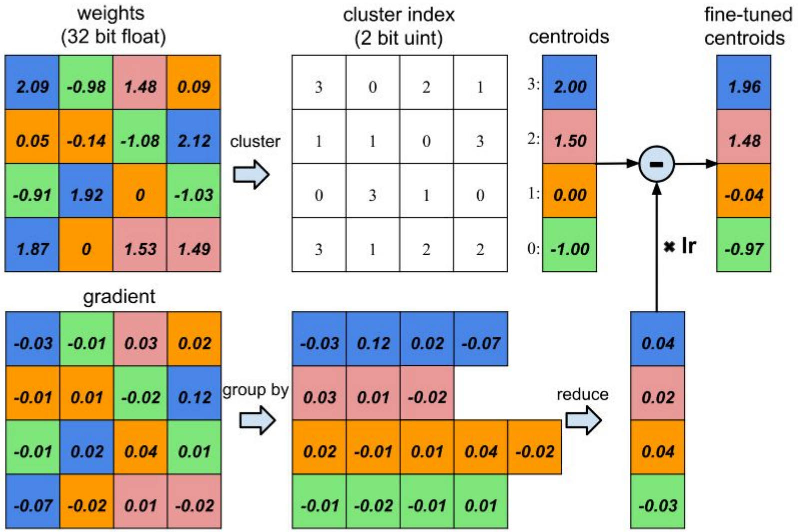
$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

DNN Compression

Weight sharing

- Weights replaced with four bit index into a table of 16 possible weight values (16 bit single-precision floating point numbers)

- Reduce memory usage



DNN Compression

Original FC Layer

$$b_i = ReLU\left(\sum_{j=0}^{n-1} W_{ij} a_j\right)$$

Compressed FC Layer

X: Set of columns $W_{ij} \neq 0$

Y: Indices $a_j \neq 0$

S: Weight table

$$b_i = ReLU\left(\sum_{j \in X_i \cap Y} S[I_{ij}] a_j\right)$$

DNN Compression

Compressed Sparse Column (CSC) Format

For each column of W:

v: non-zero weight indexes (0 if sequence of zeros longer than 4 bits of storage)

z: number of zeros before corresponding element in v

v and z are stored together as a pair. Elements in *p* point to start of each column

<u>Example</u>: W_j = $[0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \mathbf{0}, 0, 0, 3]$

v = [1, 2, 0, 3]; z = [2, 0, 15, 2]

DNN Parallelization

- Processing Element *k* (PE_k)

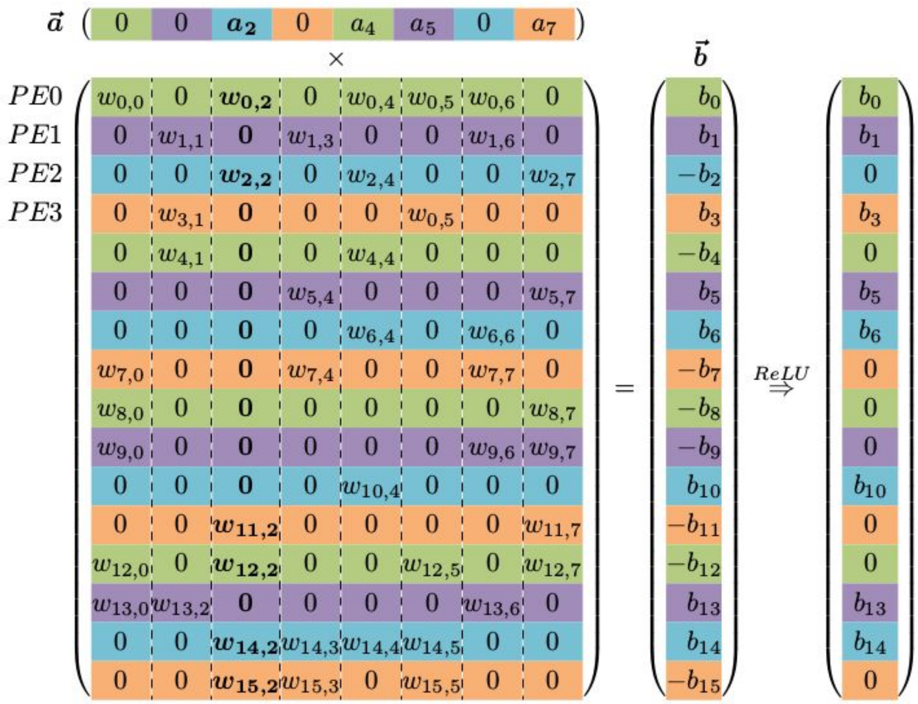
holds all rows W_i where i % N = k

- Scan *a* to find next non-zero

value (a_z), broadcast to all PE’s

- Non-zero weights in v_z multiplied

by a_z value and accumulated in b_z



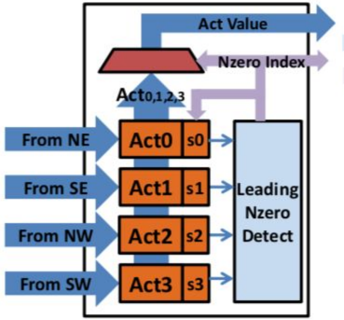
Hardware Implementation

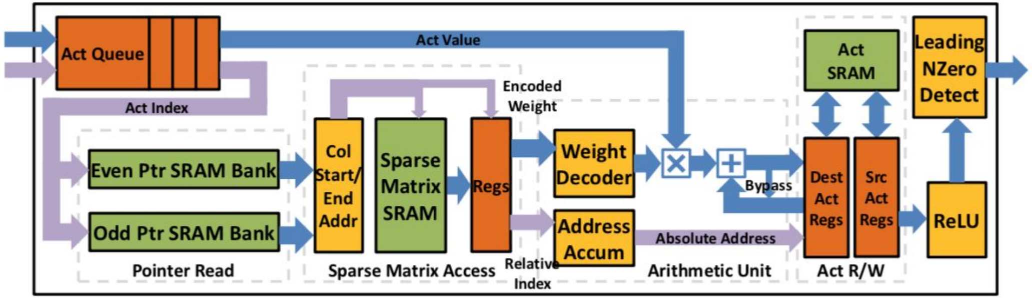
Central Control Unit

CCU

Processing Element

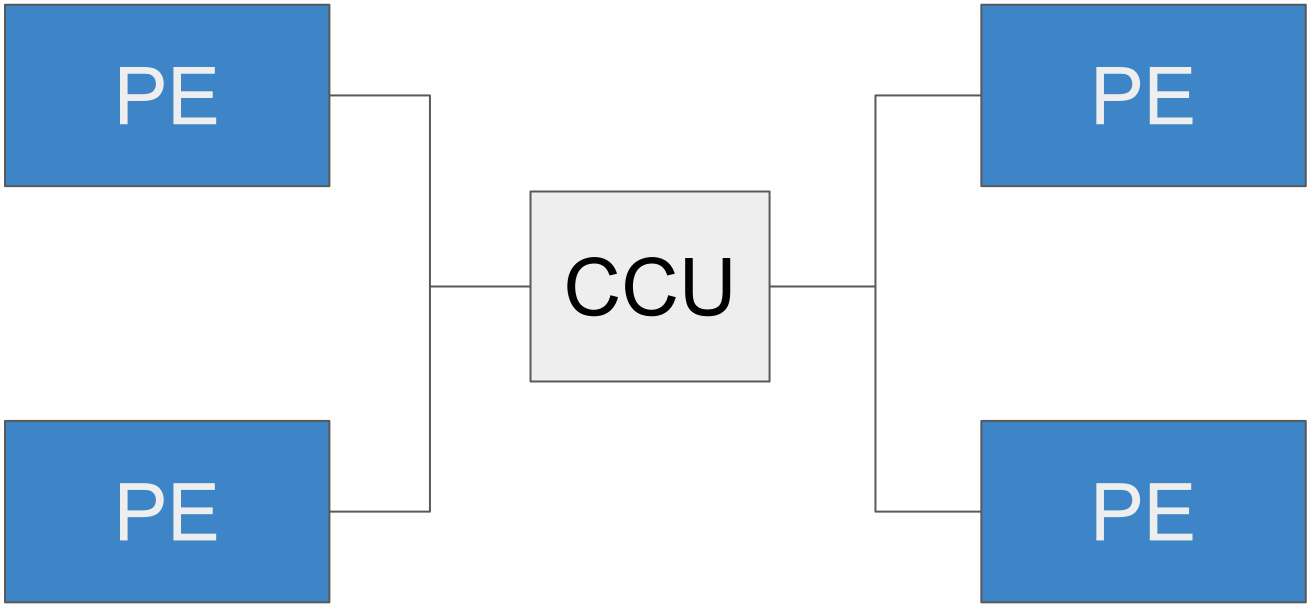
PE



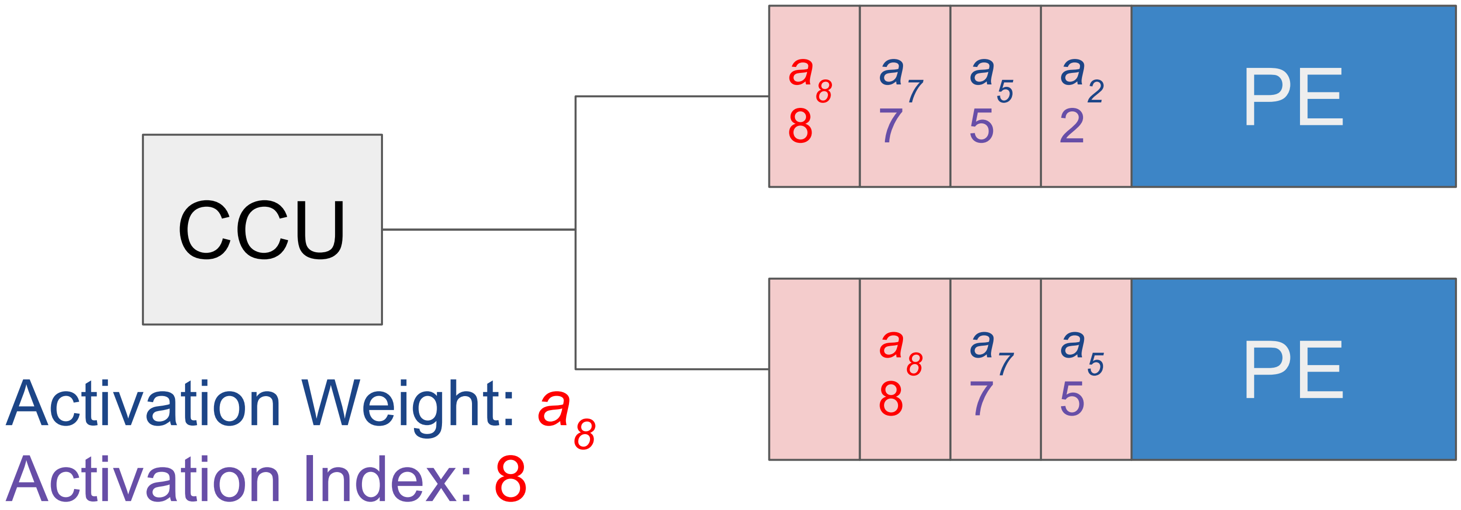


Hardware Implementation

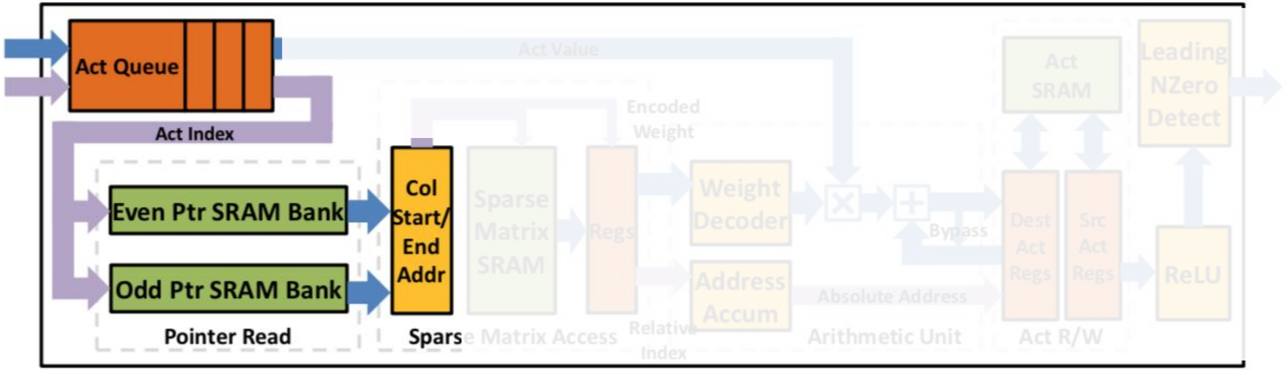
- CCU determines leading non-zero activations
- Broadcast non-zero activations to PEs



Load Balancing via Queuing



Pointer Read



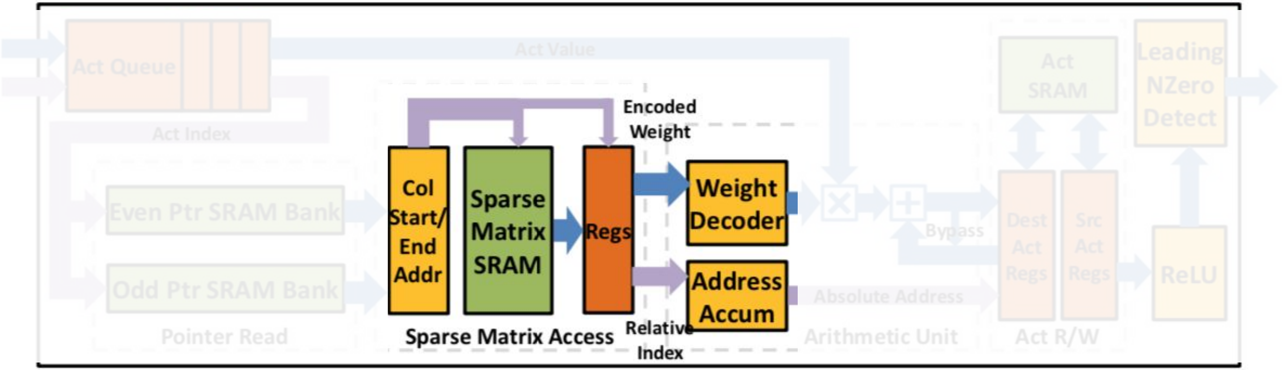
Use activation index to find pointers to weights

- Index: j
- Start Pointer: p_j
- End Pointer: p_{j+1}

Number of non-zero weights: $p_{j+1} - p_j$

	Weight Code	Separating Zeros
p_j →	1	2
	2	0
	0	15
	3	2
p_{j+1} →	5	1
	4-bits	4-bits

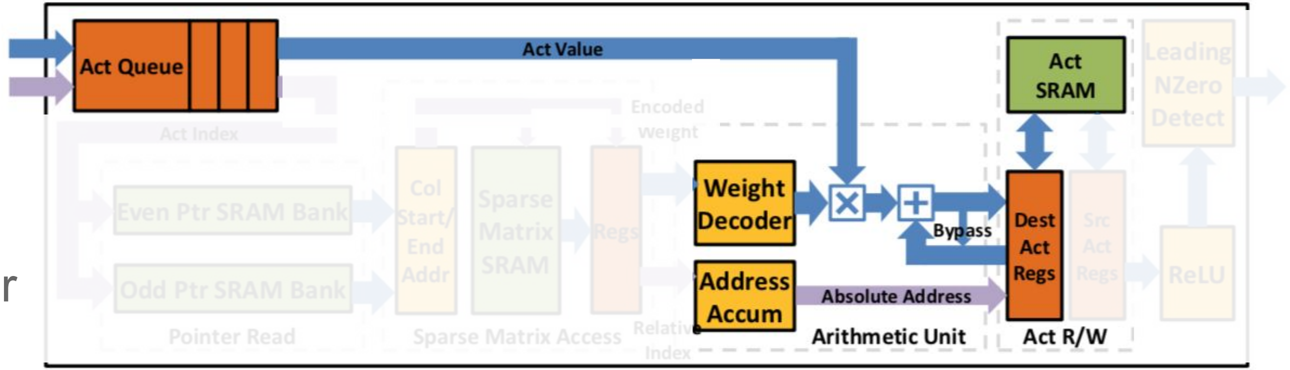
Decode



Weight Code	Separating Zeros
1	2
2	0
0	15
3	2
4-bits	4-bits

Decoded Weight, v	Accumulator Index, x
0.09375	2
12.75	3
0.0	19
4.15625	22
16-bit fixed point	

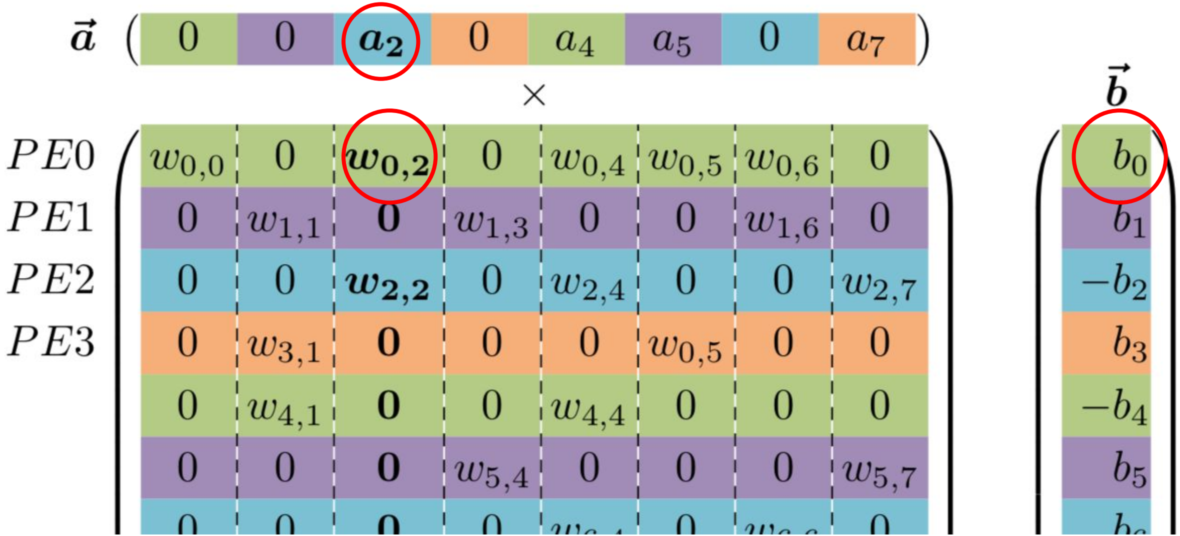
Arithmetic and Write

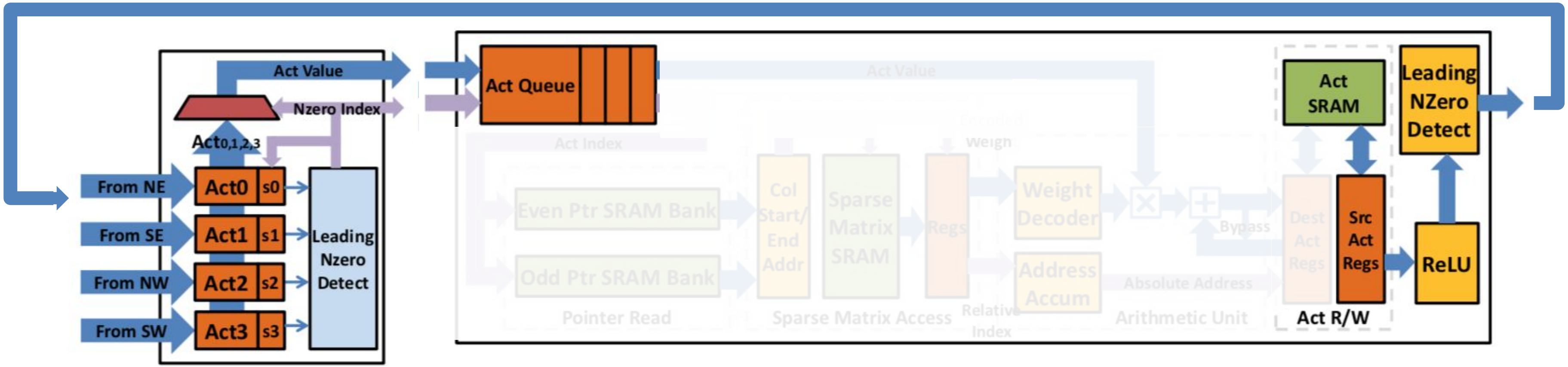


Decoded Weight, *v*	Accumulator Index, *x*
0.09375	2

$$b_x = b_x + va_j$$

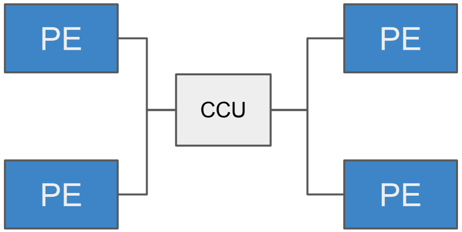
- Bypass if same accumulator is accessed consecutively to avoid pipelining hazard
- Register files hold 64 16-bit activation values per PE.
  - 4K for all 64 PEs
- Additional 2 KB activation SRAM for holding longer vectors





Distributed Leading Non-Zero Detection (LNZD)

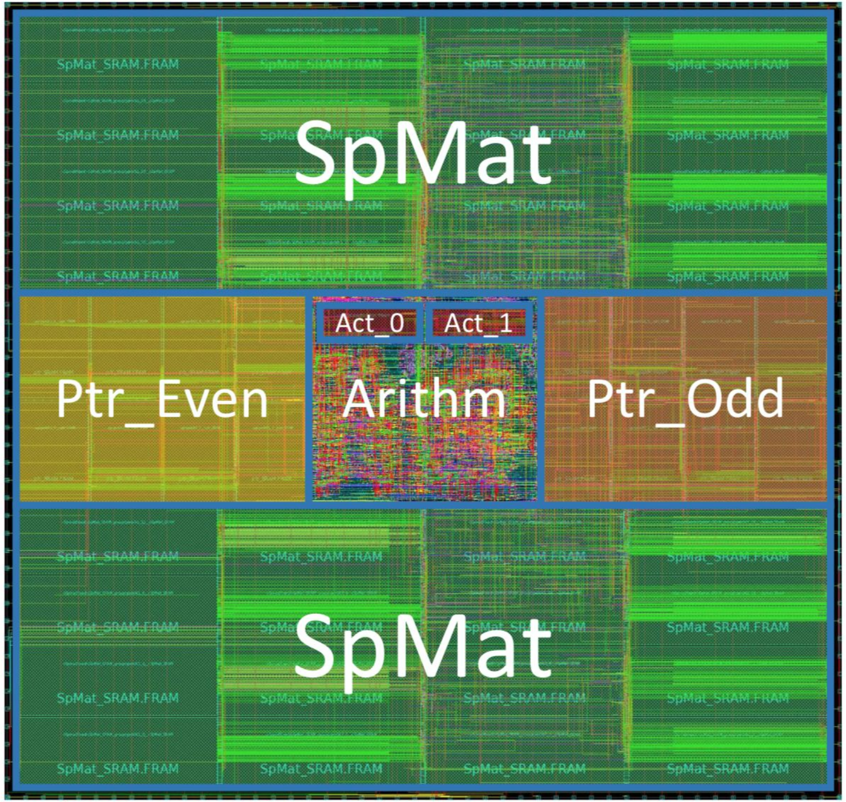
- Each PE determines leading non-zero from source activations
- One LNZD per four PEs
- Pass LNZ up quad tree to root LNZD node
- CCU is root LNZD node



Implementation, Verification, and Evaluation

RTL Implemenation	Verilog
Simulation/Verification	A custom cycle-accurate C++ simulator
Synthesis	Synopsys Design Compiler (DC) under the TSMC 45nm GP standard VT library with worst case PVT corner
Layout	Synopsys IC Compiler (ICC)
SRAM Modeling	Cacti
Power Estimation	PrimeTime PX

Implementation, Verification, and Evaluation



Specs	
Single PE Area	$0.638mm^2$
Single PE Power	9.157mW
Critical Path Delay	1.15ns
Total SRAM Capacity	162KB per PE
Performance	102 GOP/s when 64 PEs running at 800MHz

Benchmark and Comparison

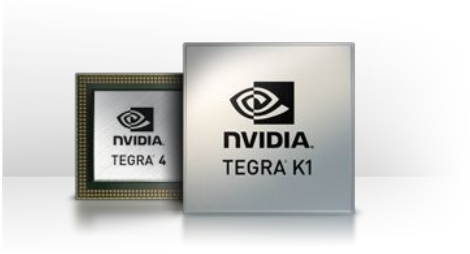
- Benchmarks

  7 DNN models from AlexNet, VGGNet, and NeuralTalk (uncompressed and compressed)

- Comparison

CPU
Intel Core i7 5930k

GPU
NVIDIA GeForce GTX Titan X



mGPU
NVIDIA Tegra K1

Comparison Results - Performance

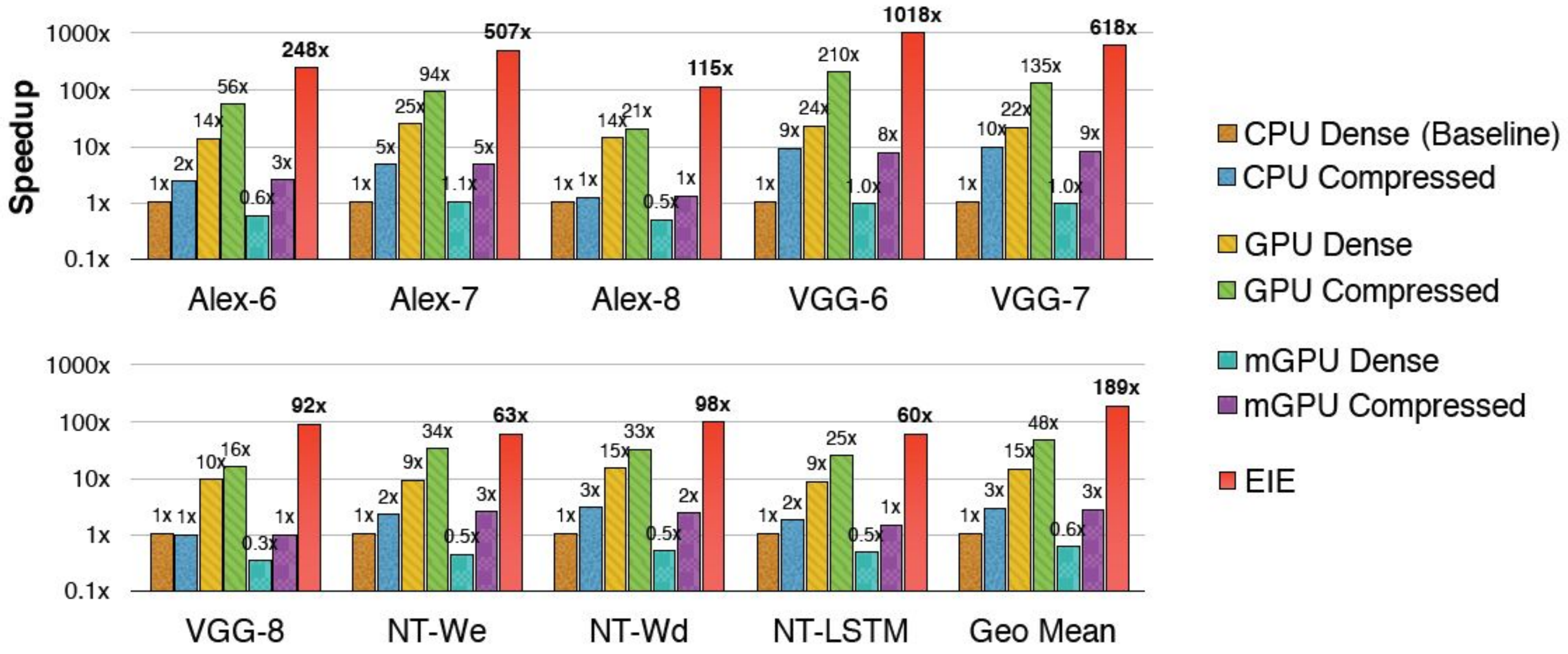


Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

Comparison Results - Energy

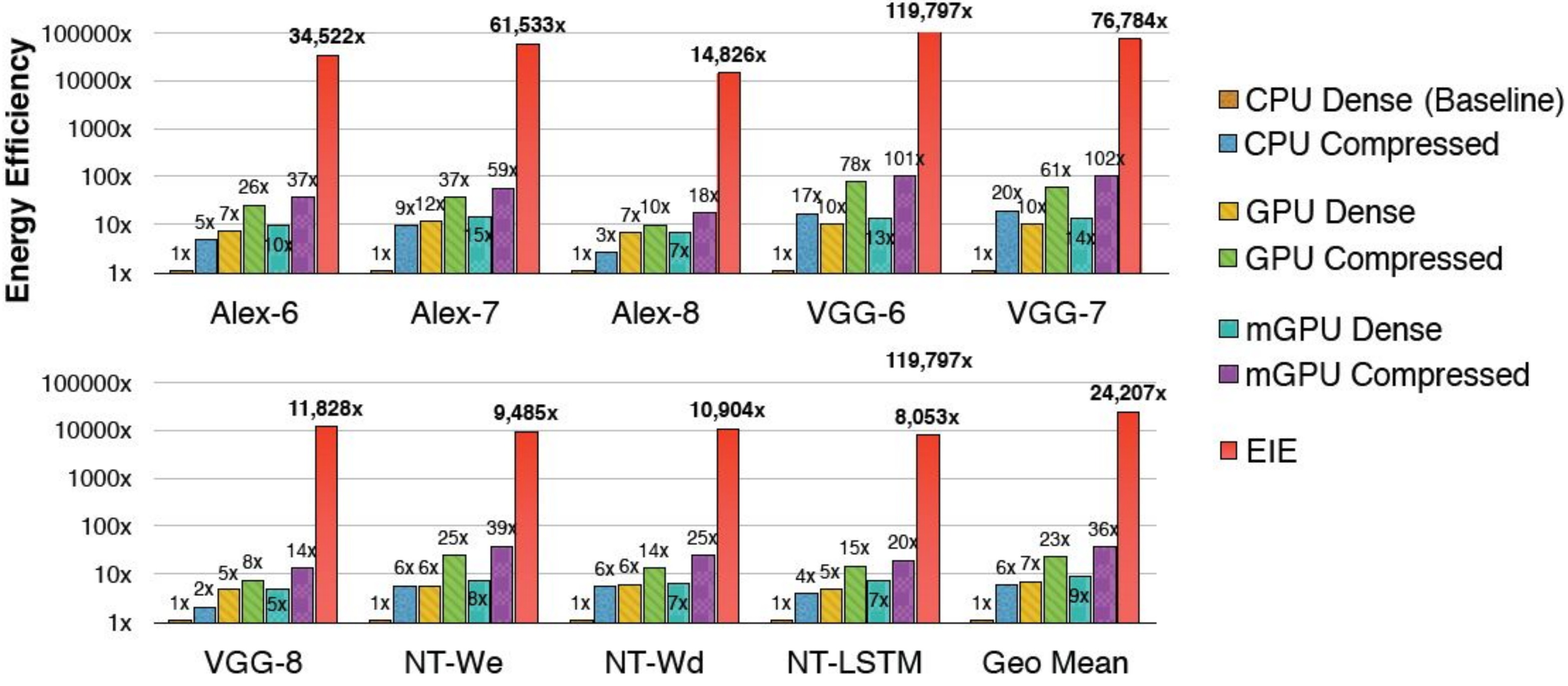


Figure 7. Energy efficiency of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

Design Space Exploration - Queue Depth

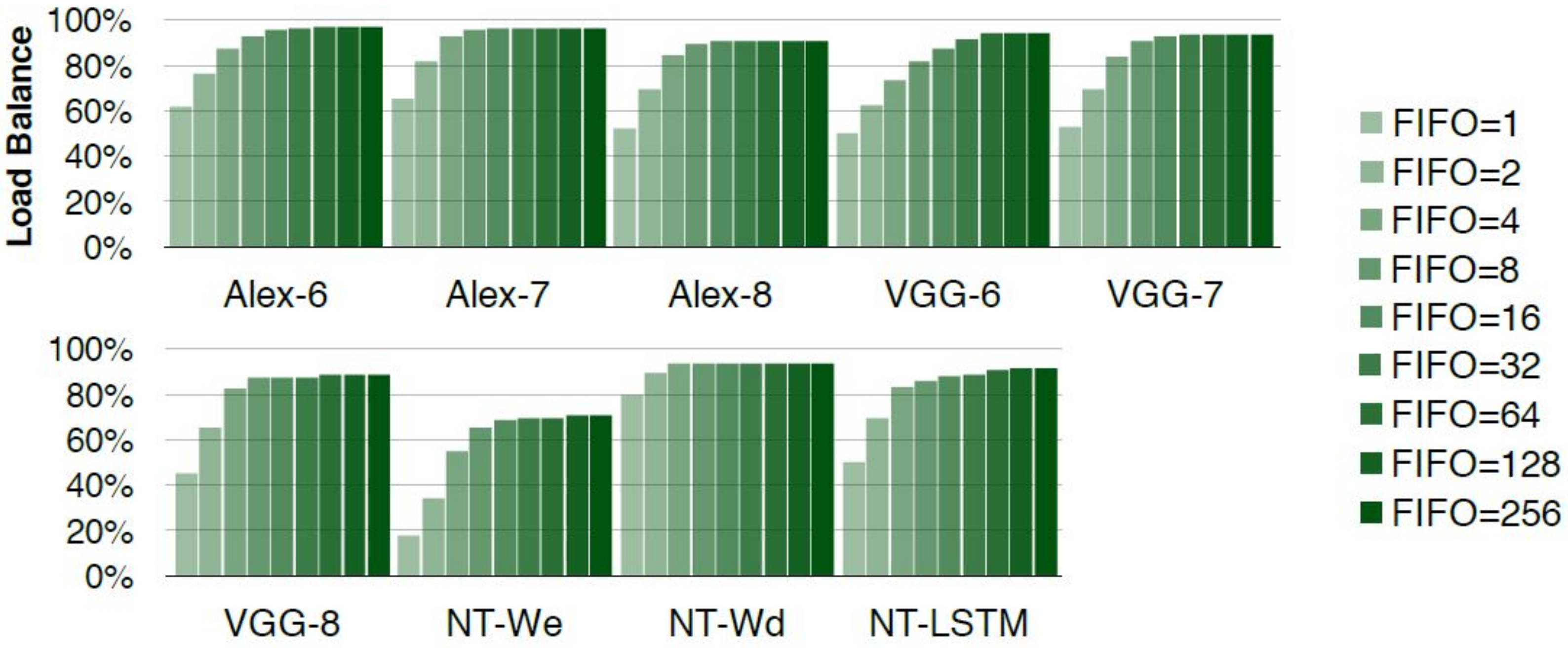
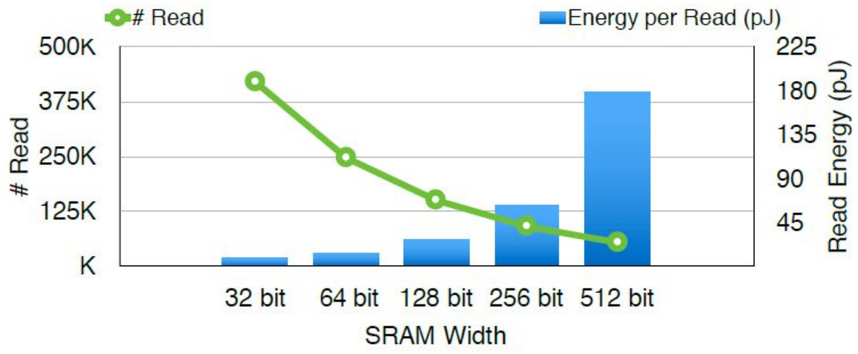
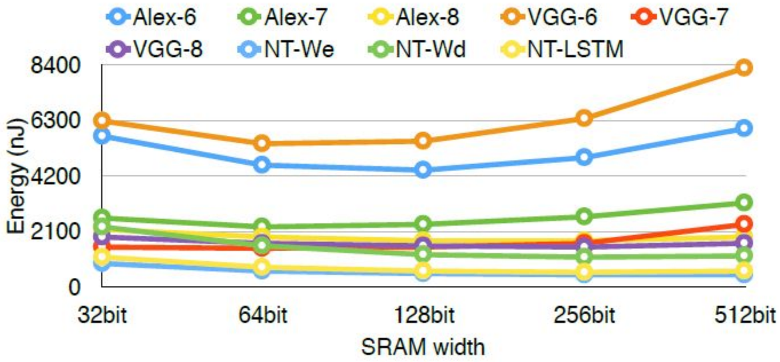


Figure 8. Load efficiency improves as FIFO size increases. When FIFO deepth>8, the marginal gain quickly diminishes. So we choose FIFO depth=8.

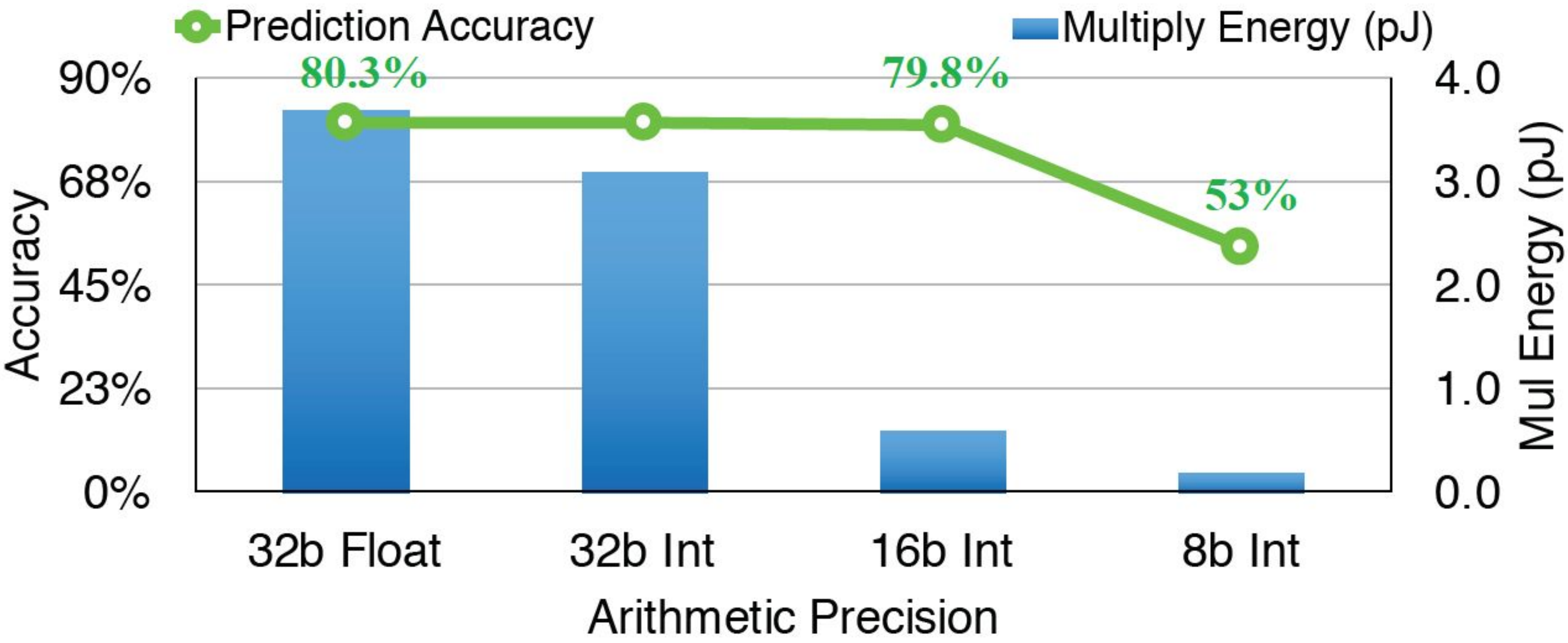
Design Space Exploration - SRAM Width





SRAM read energy and number of reads benchmarked on AlexNet. Multiplying the two curves gives the total energy consumed by SRAM read.

Design Space Exploration - Arithmetic Precision



Workload Partitioning

1. Each PE gets a column of W, still single broadcast
    a. Suffers massive load imbalancing issues, need to reduce at the end
2. The method described, each PE gets a row
3. Combined solution of block distribution
    a. Complex, still possible load balancing issues

Scalability

- Broadcast latency can be remedied via pipelining
- Larger matrix -> more PEs
- Sparsity larger than 16 zeroes can be alleviated by padding

Related Works

- This work clearly has its advantages over publications 2 years before it
  - DaDianNao and ShiDianNao, the first ones of the accelerators
- However, better performance and efficiency can be achieved by taking the outer product instead when it comes to SpMM or SpMV
  - No need to match, maximum reuse, theoretical minimum number of memory operations
  - For those that are interested, please check out S. Pal et al. “OuterSPACE: An Outer Product based Sparse Matrix Multiplication Accelerator”, HPCA 2018