Efficient Inference Engine on Compressed Deep Neural Network

By: Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark Horowitz, William Dally

Presented By: Justin Beemer, Jack Wisbiski, Yuxin Zhong

Motivation

- Many modern neural networks have millions of parameters (weights)
- With 4-byte representation (32b floating point), can easily exceed 100MB of memory required to store all parameters
- Processors used in embedded systems commonly have KBs of on-chip SRAM
- Parameters must be stored in DRAM
- DRAM accesses require >100x the energy of SRAM accesses and >1000x the energy of arithmetic operations
- **Key point: Loading parameters from DRAM dominates the energy cost of inference with a neural network.**

Key Insights

- In many neural networks, the weight matrix and activation vectors can be made sparse via **pruning**
  - If an input is 0, there's no need to load the corresponding weight from memory to perform the multiplication -- we already know the result
  - If removing a weight doesn't reduce the network accuracy, we can do so
- We can use **weight sharing** to reduce the memory required to store weights.
  - Instead of storing each individual weight, store a much smaller index into a table of shared weights
- By leveraging these ideas, neural networks can be compressed to fit in SRAM.
- This greatly improves the performance and reduces the energy required to perform inference, without sacrificing inference accuracy.

EIE (Efficient Inference Engine)

- Operates on a neural network in compressed format, allowing network to be stored entirely in SRAM
- Performs efficient matrix-vector multiplication, taking advantage of static (weight matrix) and dynamic (activation vector) sparsity
- Consists of multiple processing elements (PEs), allowing for parallel computation, load sharing, and scalability

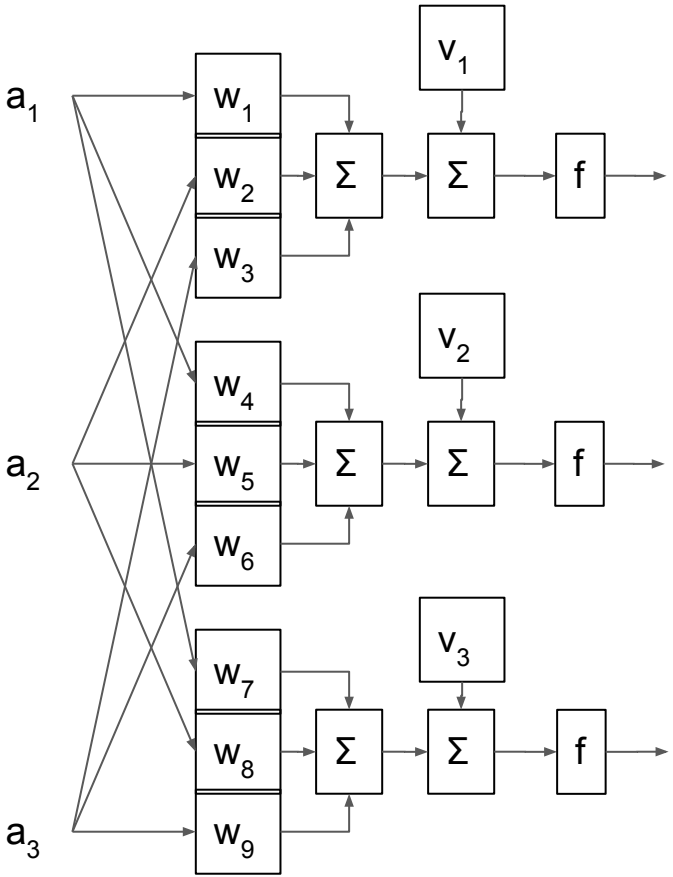
Computation

- The output vector for a fully connected (FC) layer is calculated using matrix-vector multiplication according to the following formula:

$$b = f(Wa + v)$$

- where
  - b = output vector
  - W = weight matrix
  - a = activation (input) vector
  - v = offset vector (often encoded as extra column in W)
  - f = activation function
- In EIE, **only the computations for which $W_{ij} \neq 0$ and $a_j \neq 0$ are performed.**

Example



$b_1 = f(a_1w_1+a_2w_2+a_3w_3+v_1)$

$b_2 = f(a_1w_4+a_2w_5+a_3w_6+v_2)$

$b_3 = f(a_1w_7+a_2w_8+a_3w_9+v_3)$

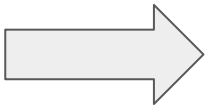
$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = f\left(\begin{bmatrix} w_1 & w_2 & w_3 & v_1 \\ w_4 & w_5 & w_6 & v_2 \\ w_7 & w_8 & w_9 & v_3 \end{bmatrix} \times \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ 1 \end{bmatrix} \right)$$

Representation

- Instead of storing weights in the weight matrix, EIE stores 4-bit indices into a shared weight table
- Instead of storing the entire matrix, stores two vectors for each column:
  - v = non-zero entries in the column
  - z = number of zeros preceding the corresponding non-zero entry
- v and z for all columns stored in a pair of vectors.
- Additional vector p contains pointers to the beginning of each column
  - The number of elements in column j is p[j+1] - p[j]

Representation: Example

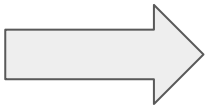
0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v							
z							
p							

Representation: Example

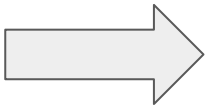
0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v	3						
z	1						
p	0						

Representation: Example

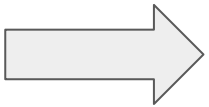
0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v	3	1					
z	1	0					
p	0						

Representation: Example

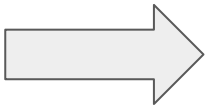
0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v	3	1	13				
z	1	0	4				
p	0						

Representation: Example

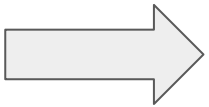
0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v	3	1	13	7			
z	1	0	4	3			
p	0	3					

Representation: Example

0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0

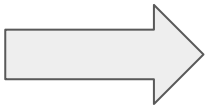


no entries for this empty column!

v	3	1	13	7			
z	1	0	4	3			
p	0	3	4				

Representation: Example

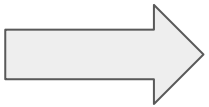
0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v	3	1	13	7	8		
z	1	0	4	3	0		
p	0	3	4	4			

Representation: Example

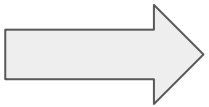
0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v	3	1	13	7	8	9	
z	1	0	4	3	0	0	
p	0	3	4	4			

Representation: Example

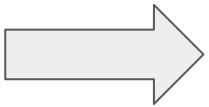
0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v	3	1	13	7	8	9	4
z	1	0	4	3	0	0	4
p	0	3	4	4			

Representation: Example

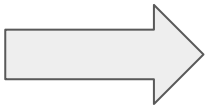
0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v	3	1	13	7	8	9	4
z	1	0	4	3	0	0	4
p	0	3	4	4	7		

Representation: Example

0	0	0	8
3	0	0	9
1	0	0	0
0	7	0	0
0	0	0	0
0	0	0	0
0	0	0	4
13	0	0	0



v	3	1	13	7	8	9	4
z	1	0	4	3	0	0	4
p	0	3	4	4	7		

Parallelization

- System contains multiple PEs, each of which stores a subset of the weight matrix, activation vector, and output vector
- PE_k stores all rows W_i, activations a_i, and outputs b_i for which i (mod N) = k
- PEs broadcast their nonzero inputs to the other PEs
- PEs receive an activation, multiply it by each piece of the corresponding weight column which they store, and accumulate the output

Parallelization Example

$w_{0,0}$	$w_{0,1}$	0	0	a_0	b_0
0	$w_{1,1}$	$w_{1,2}$	0	0	b_1
0	0	$w_{2,2}$	0	a_2	b_2
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	a_3	b_3

Green = PE_0
Orange = PE_1

Parallelization Example

$w_{0,0}$	$w_{0,1}$	0	0	a_0	b_0
0	$w_{1,1}$	$w_{1,2}$	0	0	b_1
0	0	$w_{2,2}$	0	a_2	b_2
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	a_3	b_3

Green = PE_0
Orange = PE_1

- PE_0 broadcasts a_0

Parallelization Example

$w_{0,0}$	$w_{0,1}$	0	0	a_0	$b_0 = a_0 w_{0,0}$
0	$w_{1,1}$	$w_{1,2}$	0	0	b_1
0	0	$w_{2,2}$	0	a_2	b_2
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	a_3	$b_3 = a_0 w_{3,0}$

Green = PE_0
Orange = PE_1

- PE_0 broadcasts a_0
  - PE_0 multiplies a_0 by $w_{0,0}$ and adds it to b_0
  - PE_1 multiplies a_0 by $w_{3,0}$ and adds it to b_3

Parallelization Example

$w_{0,0}$	$w_{0,1}$	0	0	a_0	$b_0 = a_0 w_{0,0}$
0	$w_{1,1}$	$w_{1,2}$	0	0	b_1
0	0	$w_{2,2}$	0	a_2	b_2
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	a_3	$b_3 = a_0 w_{3,0}$

Green = PE_0
Orange = PE_1

- PE_0 broadcasts a_0
  - PE_0 multiplies a_0 by $w_{0,0}$ and adds it to b_0
  - PE_1 multiplies a_0 by $w_{3,0}$ and adds it to b_3
- PE_1 skips a_1, since it is zero

Parallelization Example

$w_{0,0}$	$w_{0,1}$	0	0	a_0	$b_0 = a_0 w_{0,0}$
0	$w_{1,1}$	$w_{1,2}$	0	0	$b_1 = a_2 w_{1,2}$
0	0	$w_{2,2}$	0	a_2	$b_2 = a_2 w_{2,2}$
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	a_3	$b_3 = a_0 w_{3,0} + a_2 w_{3,2}$

Green = PE_0
Orange = PE_1

- PE_0 broadcasts a_0
  - PE_0 multiplies a_0 by $w_{0,0}$ and adds it to b_0
  - PE_1 multiplies a_0 by $w_{3,0}$ and adds it to b_3
- PE_1 skips a_1, since it is zero
- PE_0 broadcasts a_2
  - PE_0 multiplies a_2 by $w_{2,2}$ and adds it to b_2
  - PE_1 multiplies a_2 by $w_{1,2}$ and adds it to b_1
  - PE_1 multiplies a_2 by $w_{3,2}$ and adds it to b_3

Parallelization Example

$w_{0,0}$	$w_{0,1}$	0	0	a_0	$b_0 = a_0 w_{0,0}$
0	$w_{1,1}$	$w_{1,2}$	0	0	$b_1 = a_2 w_{1,2}$
0	0	$w_{2,2}$	0	a_2	$b_2 = a_2 w_{2,2}$
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	a_3	$b_3 = a_0 w_{3,0} + a_2 w_{3,2}$

Green = PE_0
Orange = PE_1

- PE_0 broadcasts a_0
  - PE_0 multiplies a_0 by $w_{0,0}$ and adds it to b_0
  - PE_1 multiplies a_0 by $w_{3,0}$ and adds it to b_3
- PE_1 skips a_1, since it is zero
- PE_0 broadcasts a_2
  - PE_0 multiplies a_2 by $w_{2,2}$ and adds it to b_2
  - PE_1 multiplies a_2 by $w_{1,2}$ and adds it to b_1
  - PE_1 multiplies a_2 by $w_{3,2}$ and adds it to b_3
- PE_1 broadcasts a_3

Parallelization Example

$w_{0,0}$	$w_{0,1}$	0	0	a_0	$b_0 = a_0 w_{0,0}$
0	$w_{1,1}$	$w_{1,2}$	0	0	$b_1 = a_2 w_{1,2}$
0	0	$w_{2,2}$	0	a_2	$b_2 = a_2 w_{2,2}$
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	a_3	$b_3 = a_0 w_{3,0} + a_2 w_{3,2} + a_3 w_{3,3}$

Green = PE_0
Orange = PE_1

- PE_0 broadcasts a_0
  - PE_0 multiplies a_0 by $w_{0,0}$ and adds it to b_0
  - PE_1 multiplies a_0 by $w_{3,0}$ and adds it to b_3
- PE_1 skips a_1, since it is zero
- PE_0 broadcasts a_2
  - PE_0 multiplies a_2 by $w_{2,2}$ and adds it to b_2
  - PE_1 multiplies a_2 by $w_{1,2}$ and adds it to b_1
  - PE_1 multiplies a_2 by $w_{3,2}$ and adds it to b_3
- PE_1 broadcasts a_3
  - PE_1 multiplies a_3 by $w_{3,3}$ and adds it to b_3

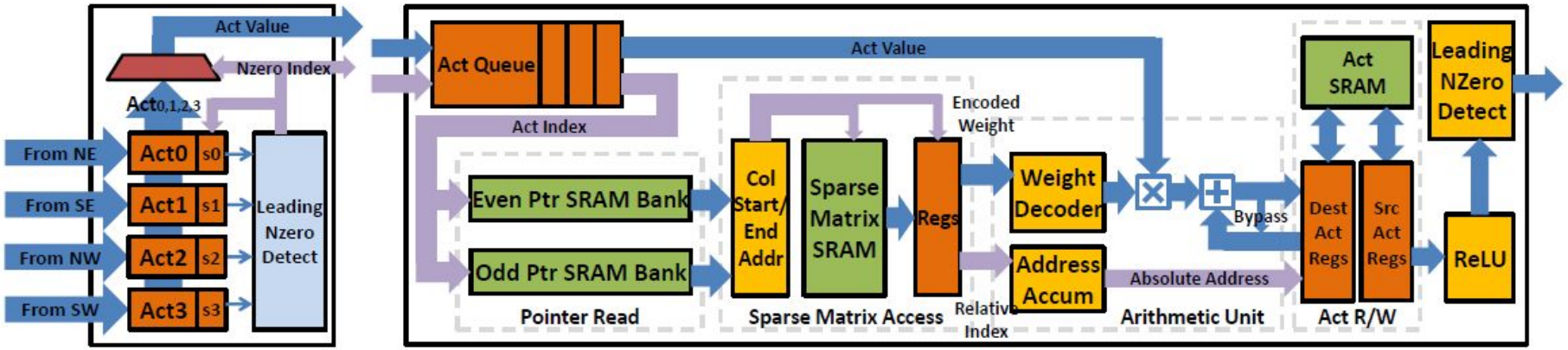
Parallelization Example

$w_{0,0}$	$w_{0,1}$	0	0	a_0	$b_0 = a_0w_{0,0}$
0	$w_{1,1}$	$w_{1,2}$	0	0	$b_1 = a_2w_{1,2}$
0	0	$w_{2,2}$	0	a_2	$b_2 = a_2w_{2,2}$
$w_{3,0}$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	a_3	$b_3 = a_0w_{3,0} + a_2w_{3,2} + a_3w_{3,3}$

Green = PE_0
Orange = PE_1

- PE_0 broadcasts a_0
  - PE_0 multiplies a_0 by $w_{0,0}$ and adds it to b_0
  - PE_1 multiplies a_0 by $w_{3,0}$ and adds it to b_3
- PE_1 skips a_1, since it is zero
- PE_0 broadcasts a_2
  - PE_0 multiplies a_2 by $w_{2,2}$ and adds it to b_2
  - PE_1 multiplies a_2 by $w_{1,2}$ and adds it to b_1
  - PE_1 multiplies a_2 by $w_{3,2}$ and adds it to b_3
- PE_1 broadcasts a_3
  - PE_1 multiplies a_3 by $w_{3,3}$ and adds it to b_3

EIE Architecture

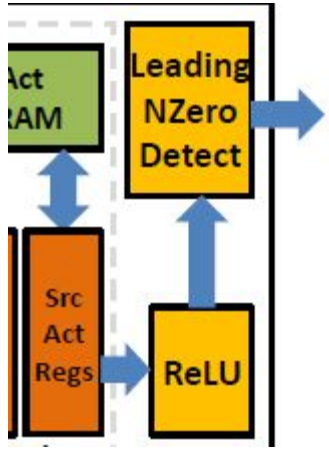


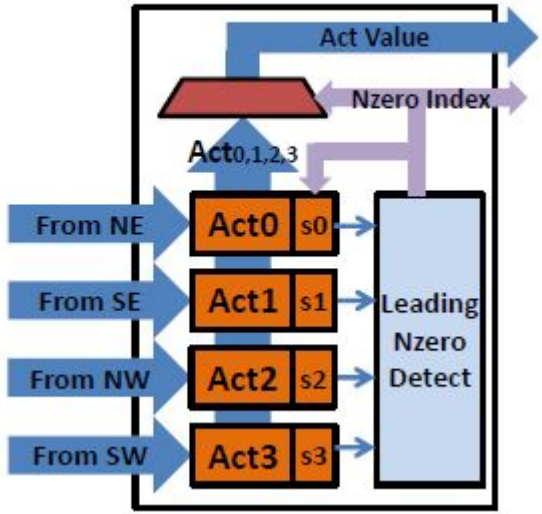
LNZD Node

PE

LNZD Node

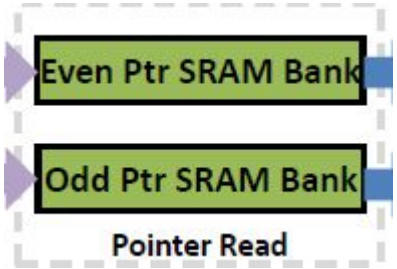
- Each group of 4 PEs do local leading non-zero detection
- Then they send the results to the connected LNZD node
- The LZND node sends the result up the quadtree
- The root LNZD Node will broadcast the result back to all PEs via separate wires placed in H-tree.
- Central Control Unit (CCU) is the root LNZD node
- CCU modes: I/O and Computing





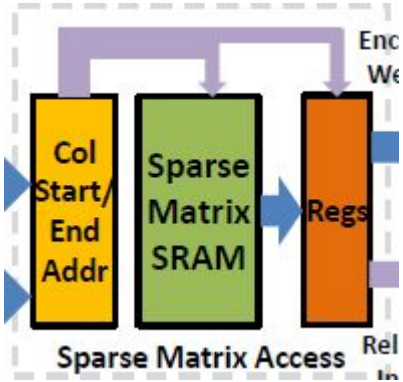
PE Architecture

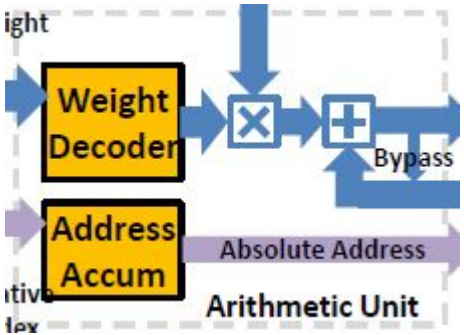
- Activation Queue(FIFO)
  - Stores the non-zero elements of an input activation vector and index broadcasted by CCU
  - Full queue > broadcast disabled
  - Load Balancing
  - PE processes activation from the head of the queue
- Pointer Read Unit
  - Index > lookup the start and end pointer of corresponding weight matrix column
  - Two pointers store in different SRAM bank > read two pointers in 1 cycle
  - Pointers are 16-bit length



PE Architecture

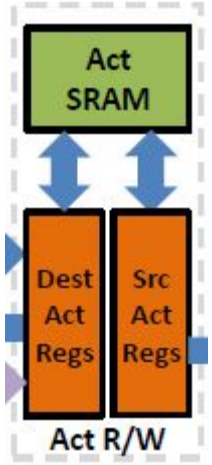
- Sparse Matrix Read Unit
  - Pointers from Pointer Read Unit > read non-zero elements in weight matrix' corresponding column
  - Sparse Matrix SRAM is of 64-bit width > store 8 (v,x)
  - v : 4-bit, x : 4-bit (CSC)
  - A single (v,x) entry is sent to Arithmetic Unit each cycle
- Arithmetic Unit
  - 4-bit encoded v > lookup table > 16-bit fixed point weight value
  - x > index accumulator array (index of output activation vector)
  - Decoded v * non-zero element of activation from activation queue
  - $b_x = b_x + v \times a_j.$





PE Architecture

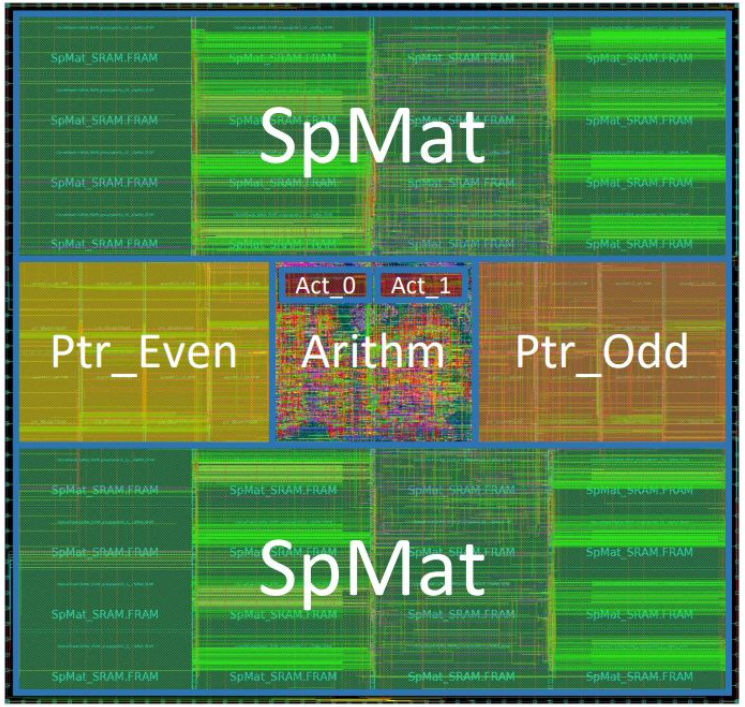
- Activation Read/Write Unit
  - Source and destination activation register file
  - Exchange roles at next layer's computation > avoid unnecessary data transfer
  - 64 16-bit activations are stored in 1 reg file > 64*64 = 4K activations in 64 PEs
  - If activation vector of length greater than 4K,
  - Then 1 matrix * vector will be completed in several 64-PE batches



Evaluation Methodology

- RTL of EIE is implemented in Verilog
- Synopsys Design Compiler and IC Compiler
- Cacti is used SRAM modeling (area & energy)
- C++ simulator
- EIE is compared with:
  - CPU: Intel Core-i7 5930k CPU
  - Desktop GPU: NVIDIA GeForce GTX Titan X GPU
  - Mobile GPU: NVIDIA Tegra K1 with 192 CUDA cores
- Comparisons are based on 2 sets of models:
  - Uncompressed DNN models from AlexNet, VGGNet, and NeuralTalk
  - Compressed DNN models

Processing Element Layout



0.638mm², 9.157mW

Table II
THE IMPLEMENTATION RESULTS OF ONE PE IN EIE AND THE BREAKDOWN BY COMPONENT TYPE (LINE 3-7), BY MODULE (LINE 8-13). THE CRITICAL PATH OF EIE IS 1.15 NS

	Power (mW)	(%)	Area (μm^2)	(%)
Total	9.157		638,024	
memory	5.416	(59.15%)	594,786	(93.22%)
clock network	1.874	(20.46%)	866	(0.14%)
register	1.026	(11.20%)	9,465	(1.48%)
combinational	0.841	(9.18%)	8,946	(1.40%)
filler cell			23,961	(3.76%)
Act_queue	0.112	(1.23%)	758	(0.12%)
PtrRead	1.807	(19.73%)	121,849	(19.10%)
SpmatRead	4.955	(54.11%)	469,412	(73.57%)
ArithmUnit	1.162	(12.68%)	3,110	(0.49%)
ActRW	1.122	(12.25%)	18,934	(2.97%)
filler cell			23,961	(3.76%)

Experimental Results

- Comparison of CPU, GPU, mGPU, EIE
- EIE targets low-latency applications
  - Performance comparison uses batch size = 1 on all hardware
  - With batch size = 64, GPU and EIE are comparable

Table IV
WALL CLOCK TIME COMPARISON BETWEEN CPU, GPU, MOBILE GPU AND EIE. UNIT: μS

Platform	Batch Size	Matrix Type	AlexNet			VGG16			NT-		
			FC6	FC7	FC8	FC6	FC7	FC8	We	Wd	LSTM
CPU (Core i7-5930k)	1	dense	7516.2	6187.1	1134.9	35022.8	5372.8	774.2	605.0	1361.4	470.5
		sparse	3066.5	1282.1	890.5	3774.3	545.1	777.3	261.2	437.4	260.0
	64	dense	318.4	188.9	45.8	1056.0	188.3	45.7	28.7	69.0	28.8
		sparse	1417.6	682.1	407.7	1780.3	274.9	363.1	117.7	176.4	107.4
GPU (Titan X)	1	dense	541.5	243.0	80.5	1467.8	243.0	80.5	65	90.1	51.9
		sparse	134.8	65.8	54.6	167.0	39.8	48.0	17.7	41.1	18.5
	64	dense	19.8	8.9	5.9	53.6	8.9	5.9	3.2	2.3	2.5
		sparse	94.6	51.5	23.2	121.5	24.4	22.0	10.9	11.0	9.0
mGPU (Tegra K1)	1	dense	12437.2	5765.0	2252.1	35427.0	5544.3	2243.1	1316	2565.5	956.9
		sparse	2879.3	1256.5	837.0	4377.2	626.3	745.1	240.6	570.6	315
	64	dense	1663.6	2056.8	298.0	2001.4	2050.7	483.9	87.8	956.3	95.2
		sparse	4003.9	1372.8	576.7	8024.8	660.2	544.1	236.3	187.7	186.5
EIE	**Theoretical Time**		**28.1**	**11.7**	**8.9**	**28.1**	**7.9**	**7.3**	**5.2**	**13.0**	**6.5**
	Actual Time		**30.3**	**12.2**	**9.9**	**34.4**	**8.7**	**8.4**	**8.0**	**13.9**	**7.5**

Performance Comparison

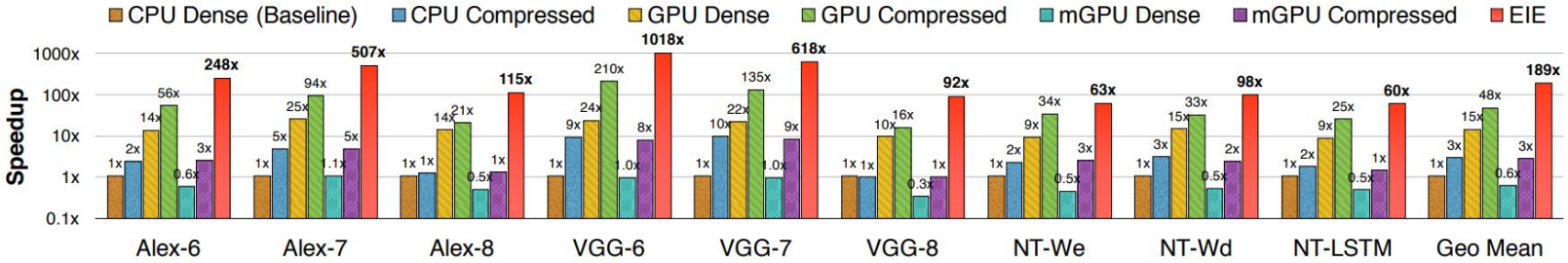


Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

- EIE outperforms general purpose hardware by factor of:
  - CPU: 189x
  - GPU: 13x
  - mGPU: 307x
- DNN Compression alone gives 3x speedup
- EIE dedicated logic exploits sparsity to eliminate 97% of GOP/s

Energy Efficiency

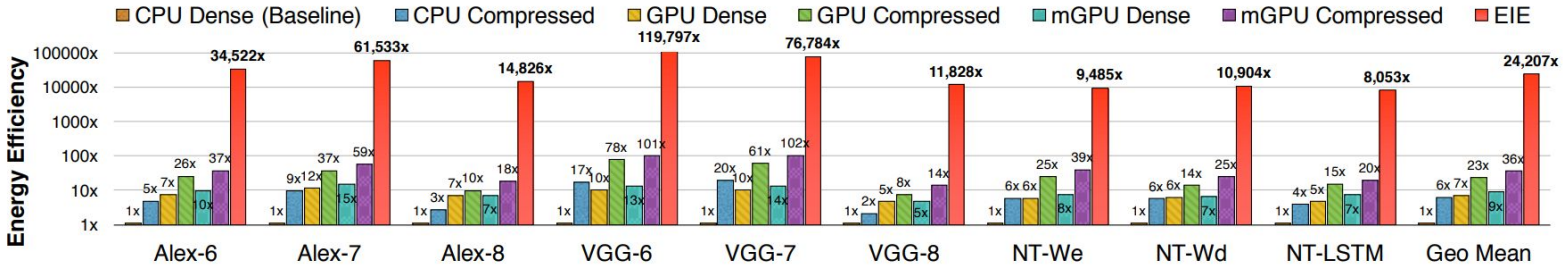
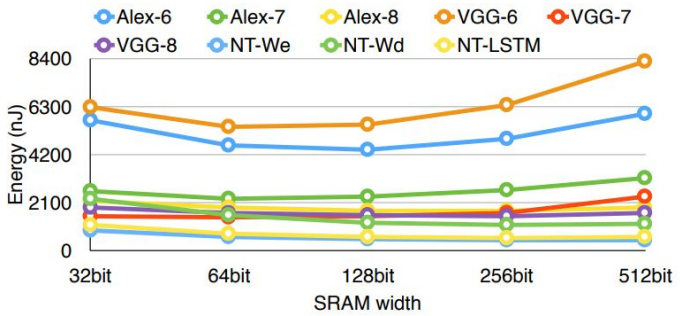


Figure 7. Energy efficiency of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.

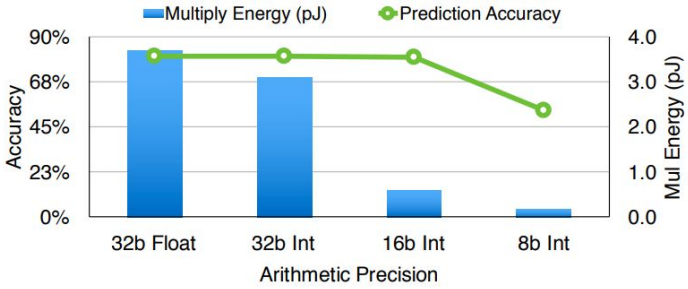
- EIE consumes less power than general purpose hardware by factor of:
  - CPU: 24,000x
  - GPU: 3,400x
  - mGPU: 2,700x
- Energy savings are accomplished in three ways:
  - Required energy per read is reduced 120x by using SRAM rather than DRAM
  - Compressed DNN model reduces required reads 10x
  - Vector sparsity reduces redundant computation cycles 65%

Design Space Exploration

- SRAM Width
  - Wider SRAM reduces total # accesses
  - Smaller SRAM reduces energy per read
  - Minimum total access occurs at 64 bits



- Arithmetic Precision
  - Higher precision increases prediction accuracy
  - Lower precision reduces energy consumption
  - 16-bit fixed-point reduces prediction accuracy by 0.5% compared to 32-bit floating-point, while using 6.2x less energy



Design Space Exploration

- FIFO queue depth:
  - Greater FIFO depth increases load balance
  - Depth 8 is chosen - diminishing returns beyond this point

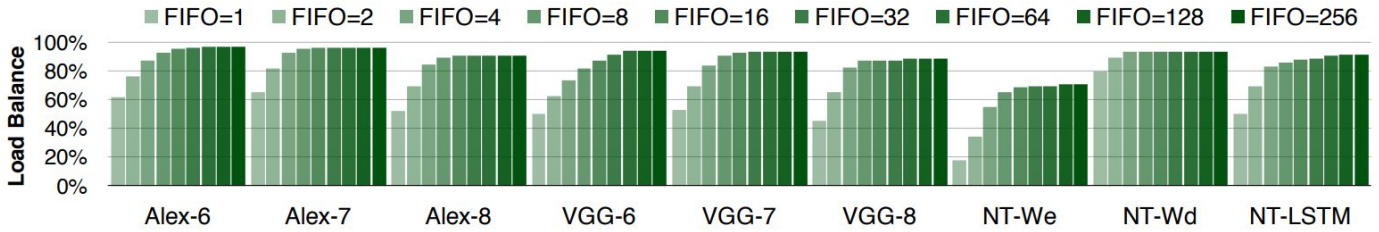


Figure 8. Load efficiency improves as FIFO size increases. When FIFO deepth>8, the marginal gain quickly diminishes. So we choose FIFO depth=8.

- MxV workload partitioning options:
  - Distribute W columns to PEs - requires reduction operation for final result
  - **Distribute W rows to PEs** - chosen method, requires broadcasting full input to all PEs
  - Hybrid approach to distribute 2D blocks of W - inherent complexity

Scalability and Flexibility

- Scalability:
  - # PEs can be scaled up with matrix size
  - More PEs reduces zero padding
  - Speedup scales near-linearly with # PEs
- Flexibility:
  - EIE handles large NN input/output and weights
  - Can also support other computation and perform convolution MxV operations

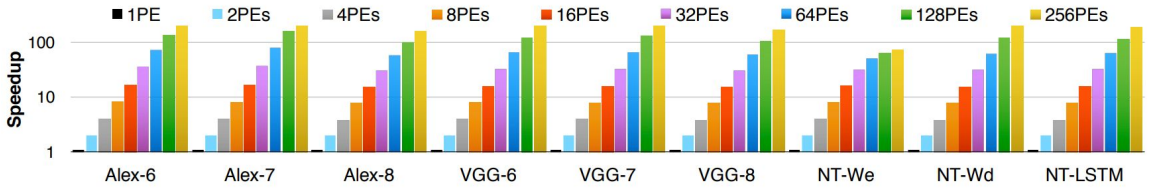
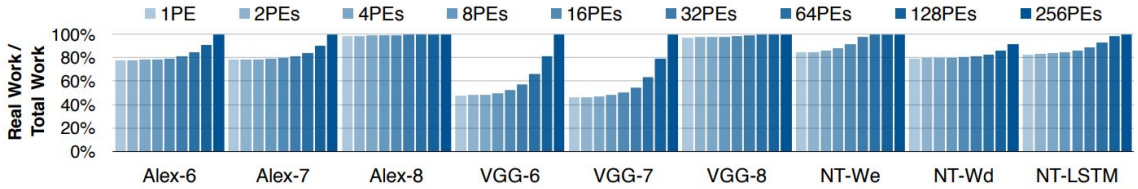


Figure 11. System scalability. It measures the speedups with different numbers of PEs. The speedup is near-linear.



Comparison to Other DNN Hardware

- Other DNN platforms considered: Core-i7 CPU, Titan X GPU, Tegra K1 mGPU, A-Eye FPGA, DaDianNao ASIC, and TrueNorth ASIC
- Other platforms have low MxV efficiency, cannot exploit weight (static) and input (dynamic) sparsity, often use uncompressed models
- EIE maintains high MxV throughput, explicitly designed for compression into on-chip SRAM and use of sparsity

Conclusions

- Matrix-vector multiplication in fully-connected DNN layers is memory-limited, especially for real-time networks with low latency requirements
- EIE utilizes pruning and weight sharing for DNN compression, and leverages resulting sparsity for speedup
- Weights are stored in on-chip SRAM for energy savings
- Implications: EIE ideal for use in real-time applications and low-power systems, but requires specialized hardware

Questions?

Backup Slide: Hardware Comparison Table

Table V
COMPARISON WITH EXISTING HARDWARE PLATFORMS FOR DNNS.

Platform	Core-i7 5930K	GeForce Titan X	Tegra K1	A-Eye [14]	Da-DianNao [11]	True-North [40]	**EIE (ours, 64PE)**	**EIE (28nm, 256PE)**
Year	2014	2015	2014	2015	2014	2014	2016	2016
Platform Type	CPU	GPU	mGPU	FPGA	ASIC	ASIC	ASIC	ASIC
Technology	22nm	28nm	28nm	28nm	28nm	28nm	45nm	28nm
Clock (*MHz*)	3500	1075	852	150	606	Async	800	1200
Memory type	DRAM	DRAM	DRAM	DRAM	eDRAM	SRAM	SRAM	SRAM
Max DNN model size (*#Params*)	<16G	<3G	<500M	<500M	18M	256M	84M	336M
Quantization Stategy	32-bit float	32-bit float	32-bit float	16-bit fixed	16-bit fixed	1-bit fixed	4-bit fixed	4-bit fixed
Area (mm^2)	356	601	-	-	67.7	430	40.8	63.8
Power (*W*)	73	159	5.1	9.63	15.97	0.18	0.59	2.36
M×V Throughput (*Frames/s*)	162	4,115	173	33	147,938	1,989	81,967	426,230
Area Efficiency ($Frames/s/mm^2$)	0.46	6.85	-	-	2,185	4.63	2,009	6,681
Energy Efficiency (*Frames/J*)	2.22	25.9	33.9	3.43	9,263	10,839	138,927	180,606

Backup Slide: QuadTree and H-Tree

- QuadTree: A quadtree is a tree data structure in which each internal node has exactly 4 children.
- An interactive explanation of quadtree:https://jimkang.com/quadtreevis/
- H-tree: It is a fractal tree structure constructed from perpendicular line segments, each smaller by a factor of square root of 2 from the next larger adjacent segment.
- H-Tree Construction process:
  - https://medium.com/algorithm-and-datastructure/h-tree-construction-3d6989186ce0

References:
[1]https://en.wikipedia.org/wiki/H_tree
[2]https://en.wikipedia.org/wiki/Quadtree#:~:text=A%20quadtree%20is%20a%20tree,into%20four%20quadrants%20or%20regions.