Deep Convolutional Neural Network in Resource-Constrained Embedded Systems: How to Fit an Elephant in a Car?

Soheil Ghiasi

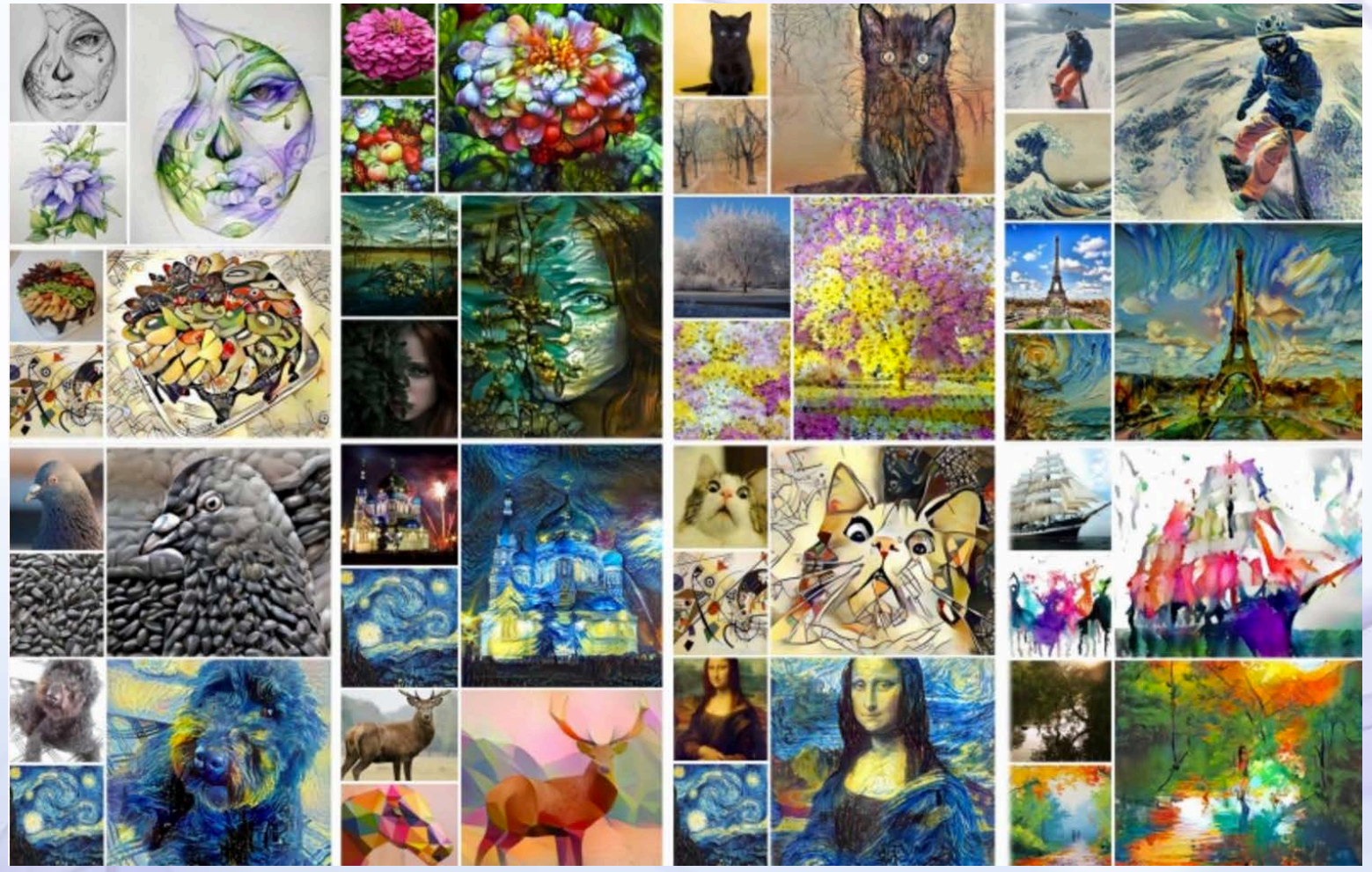
Electrical and Computer Engineering Department
University of California, Davis

LEPS
Laboratory for Embedded and Programmable Systems

UCDAVIS
UNIVERSITY OF CALIFORNIA

What's all the fuss about!

- Tremendous progress in "AI"
  - Visual recognition
  - Speech understanding
  - Natural language processing
- Two key enablers
  - Ample data
  - Ample compute capability

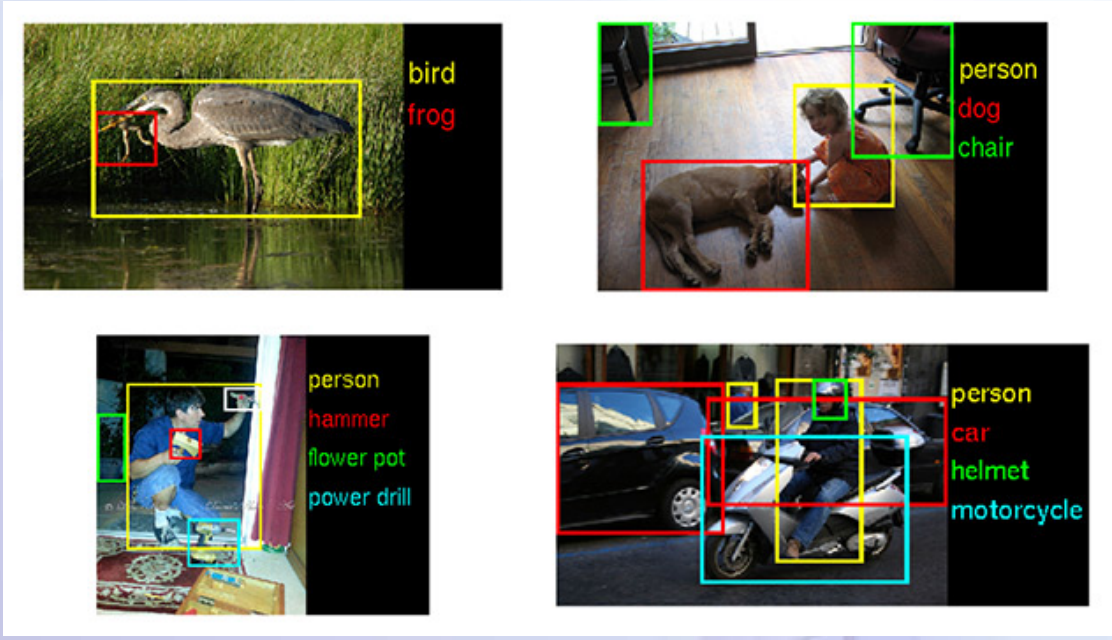


[bored panda]

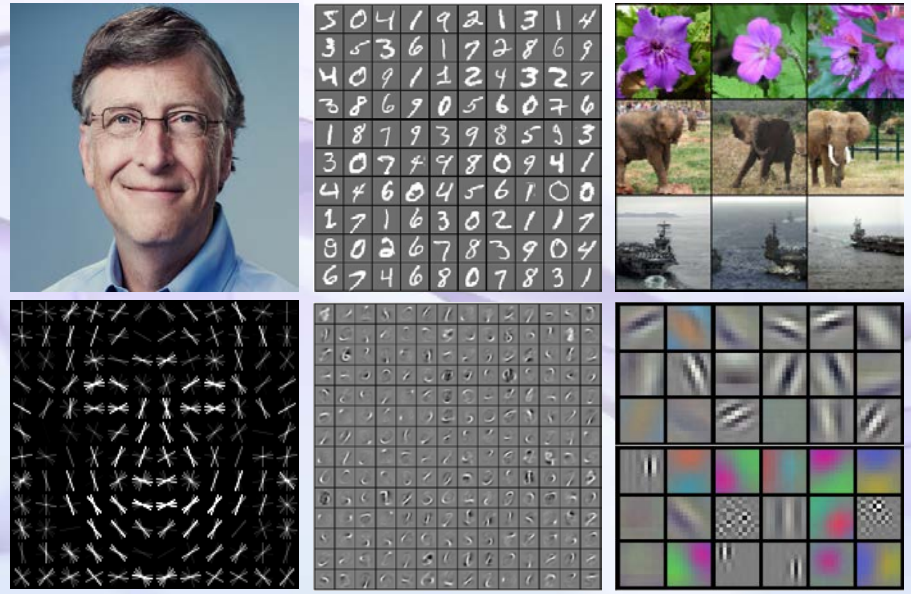
Machine Vision: Past, Present and Future!

Feature Extraction Approaches

- Hand crafted features such as HoG and SIFT
- Representation Learning (end-to-end learning, automated features extraction)

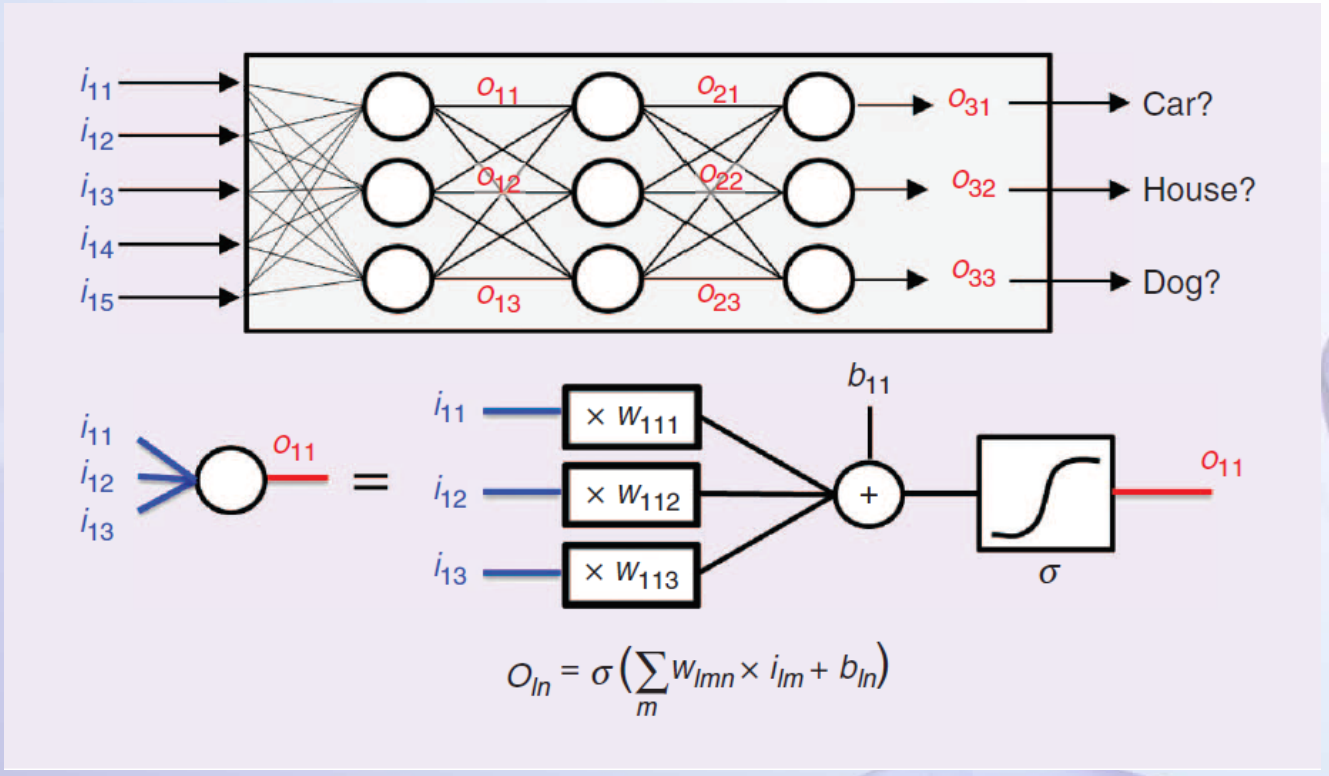


ILSVRC (Imagenet) Competition

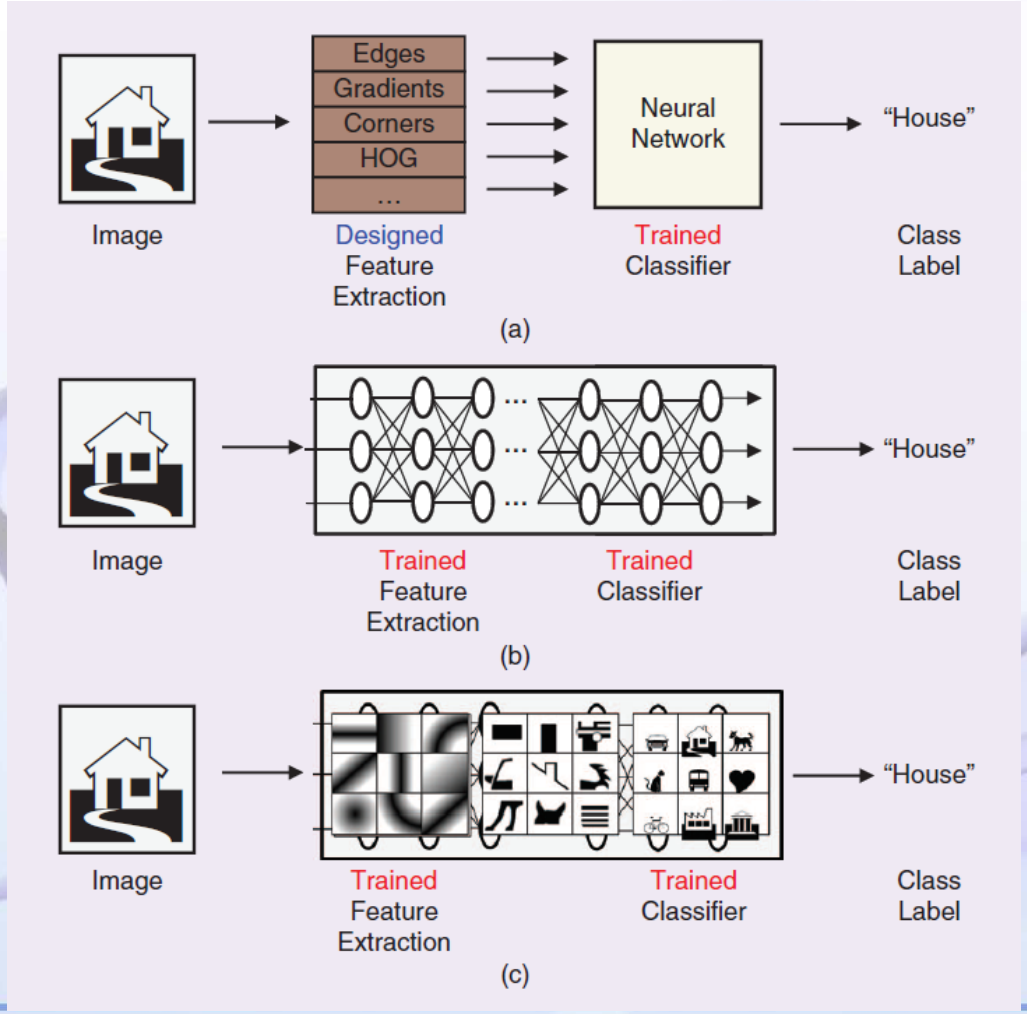


[Krizhevsky et al. 2012]

Representation Learning with Deep Neural Networks

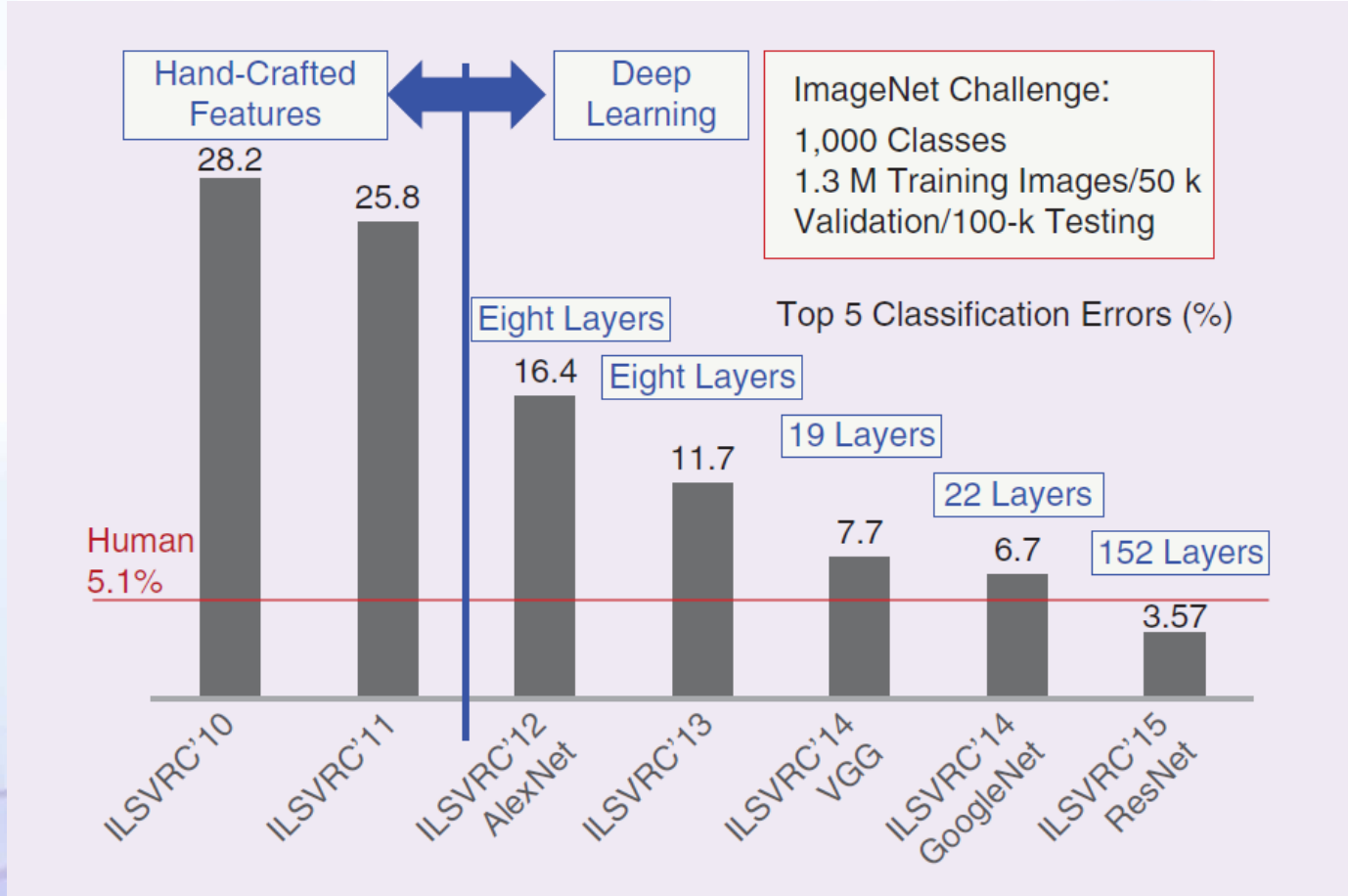


[Verhelst and Moons 2017]



ILSVRC (ImageNet) Competition

- Best Top-5 classification error in object recognition task
- Substantial progress after deep neural networks

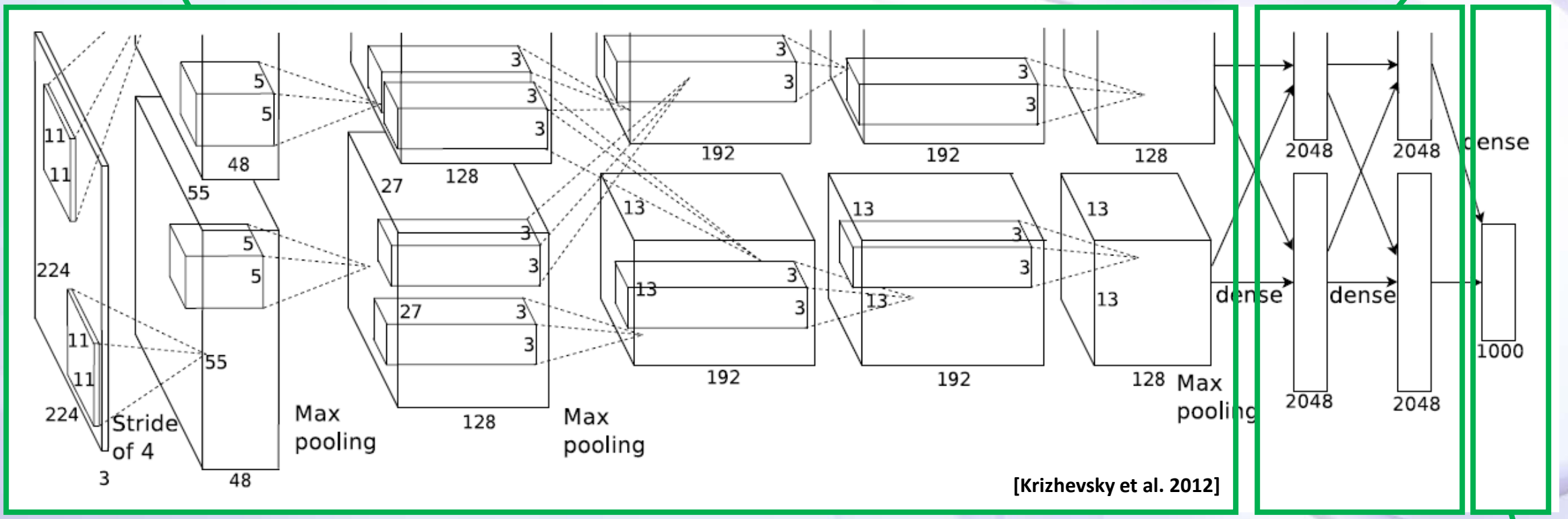


[Verhelst and Moons 2017]

AlexNet as an Example

Convolutional Layers.
They implicitly use locality of pixel dependency.

Fully Connected Layers.
They can extract local and global features.



Feature Extractor

Classifier

- 650K Neurons; 60M parameters; 630M connections
- ~1.3B operations

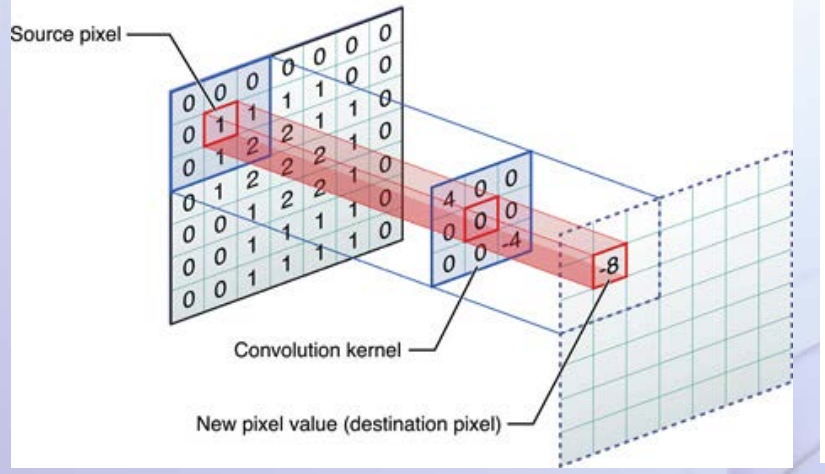
- Example AlexNet results

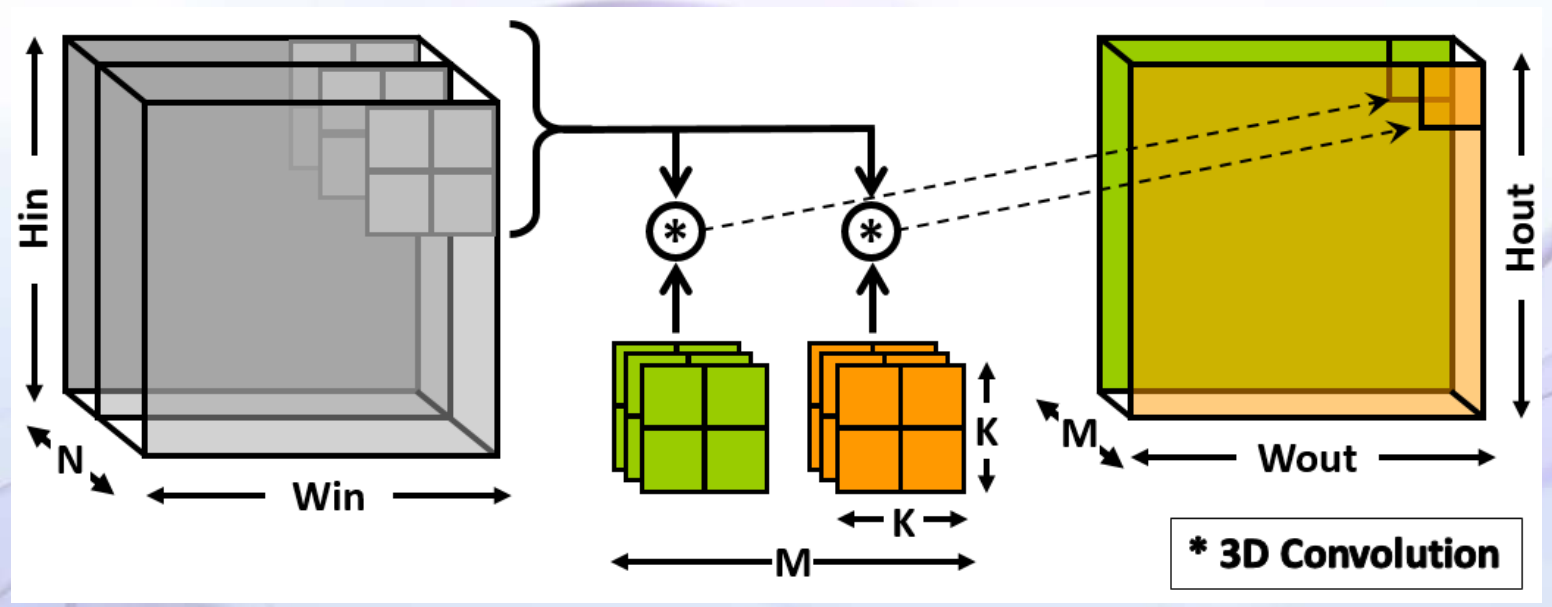
[Krizhevsky et al. 2012]

3D Convolution

- Center element of the kernel is logically placed on each of pixels in Input Feature Map (IFM)
- Each Output Feature Map (OFM) is computed by 3D convolution of a number of Input Feature Maps (IFM) with a Kernel stack.

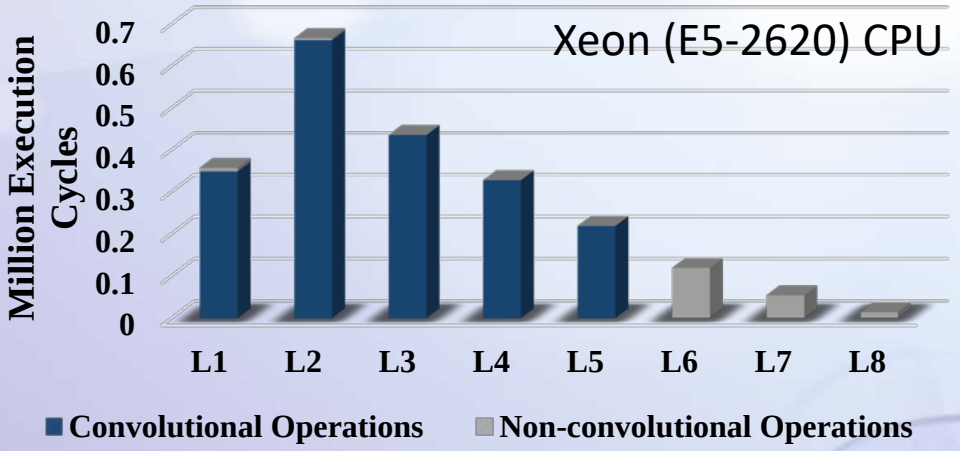
[developer.apple.com]

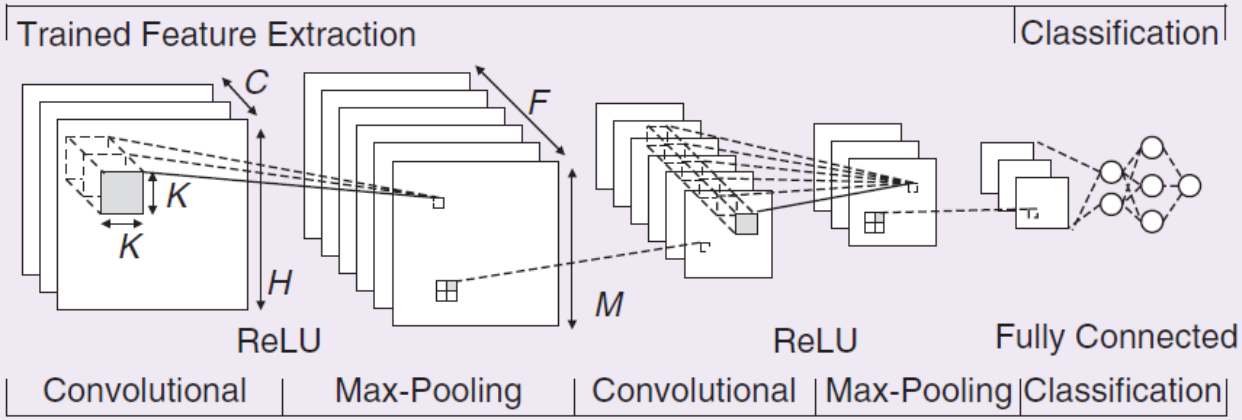




Let's Switch Hats: From ML to Systems

- Simplified view of the computation is shown
- Training vs. Inference





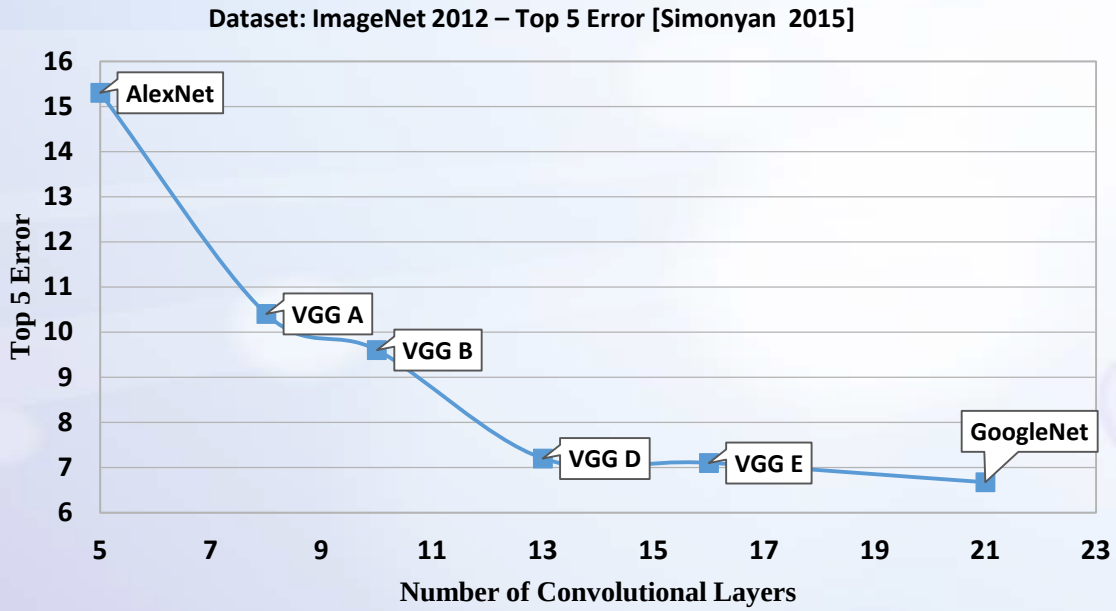

```
for (int f = 0; f < F; f++)
  for (int mx = 0; mx < M; my++)
    for (int my = 0; my < M; mx++)
      for (int c = 0; c < C; c++)
        for (int kx = 0; kx < K; kx++)
          for (int ky = 0; ky < K; ky++)
            o[c][mx][my] += w[f][c][kx][ky] · i[c][mx + kx][my + ky];
```

Per Output Pixel of a Layer:
- Load $C.K^2$ Weights
- Load $C.K^2$ Inputs
- Do $C.K^2$ MACs
- One Output Store

Repeat $F.M^2$ Times Per Layer

More than 90% of AlexNet are spent in convolutional layers (modern networks have a similar profile).

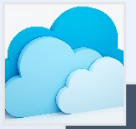
Inference in Embedded Systems



- Higher Depth
- Higher Precision
- Higher Execution Time
- Higher Power Consumption

It is challenging to fit an elephant in a car

Inference Implementation Options

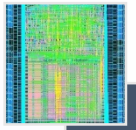


Cloud

- High energy consumption to communicate imagery
- Network may be unreliable and/or unavailable
- Latency and Privacy

GPU/CPU

- Software programmable
- Energy efficiency better than CPUs but inferior to custom HW



ASIC

- Initial Cost
- Reusability
- Complexity of design
- Superior performance

FPGA

- Promise of energy efficiency vs. cost tradeoff
- Design complexity lower than ASIC, higher than GPU

An energy efficient and flexible implementation is essential for deployment of DCNN in mobile devices.

Our work in This Area

Caffe

Training

Inference

Workstation

Compression (Ristretto)

FPGA

Verilog (PLACID)

OpenCL (FICaffe)

Mobile SoC

Android Software Synthesis (Cappuccino)

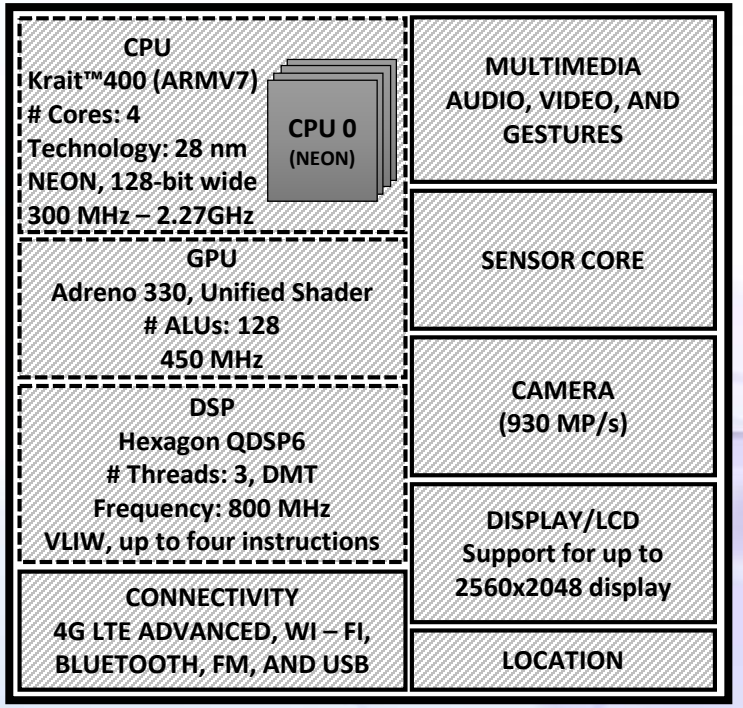
Granularity Optimization

Switching Gears …

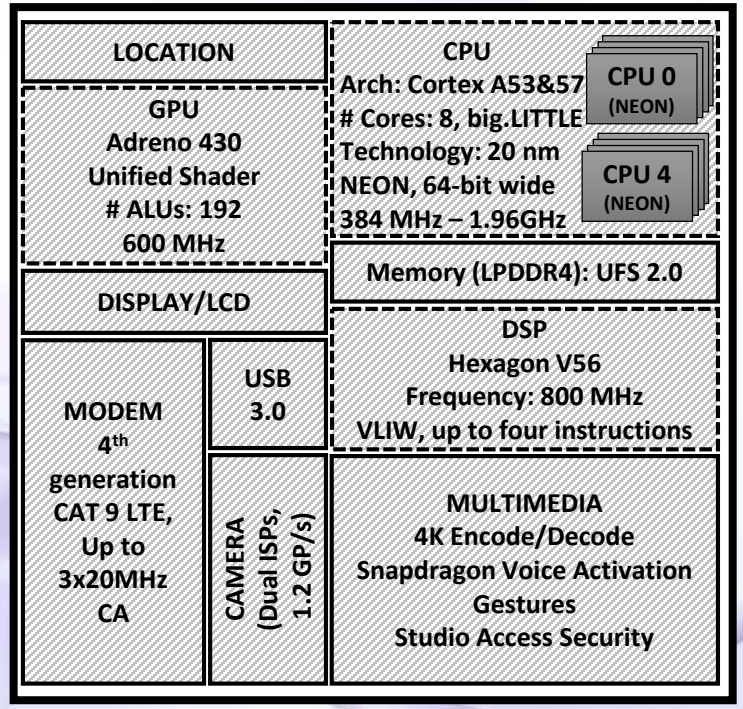
Mobile SoC Implementation

Mobile SoC as a Dominant IoT Platform

- Mobile SoCs (or their similar-looking clones) are well-positioned to be the platform of choice for many IoT applications.



Qualcomm Snapdragon 800
(Google Nexus 5)*

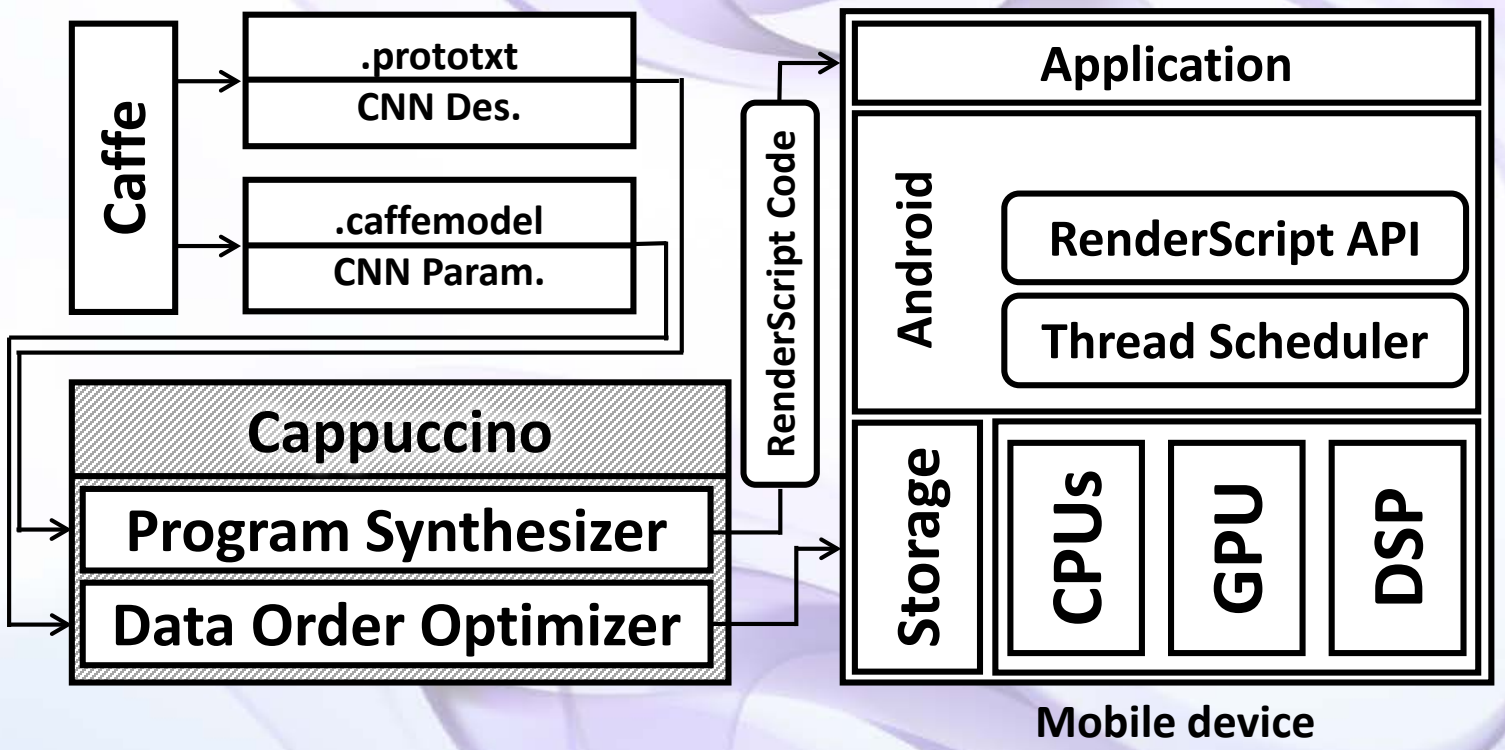


Qualcomm Snapdragon 810
(Google Nexus 6P)**

* https://www.qualcomm.com/products/snapdragon/processors/800
** https://www.qualcomm.com/products/snapdragon/processors/810

Cappuccino

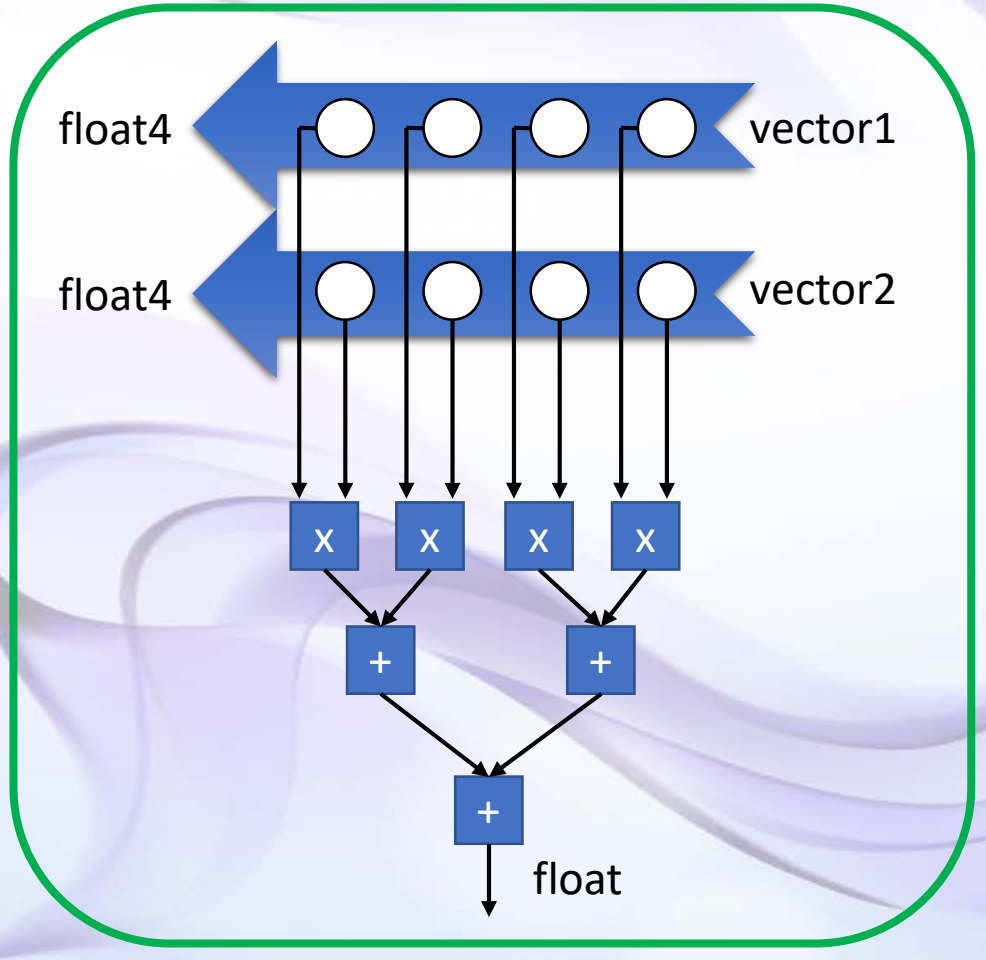
- Cappuccino is a parallel inference software synthesizer for mobile devices.
- Input
  - CNN description file
  - CNN parameter file
- Output
  - RenderScript program for SoC-based acceleration
- Sample Results
  - Accelerating GoogLeNet, SqueezeNet, and AlexNet on Nexus 6P.



CNN Name	Baseline (ms)	Parallel (ms)	Speedup
AlexNet	8626	62	139X
SqueezeNet	17299	141	123X
GoogLeNet	25570	602	42X

Vectorized Operation in RenderScript

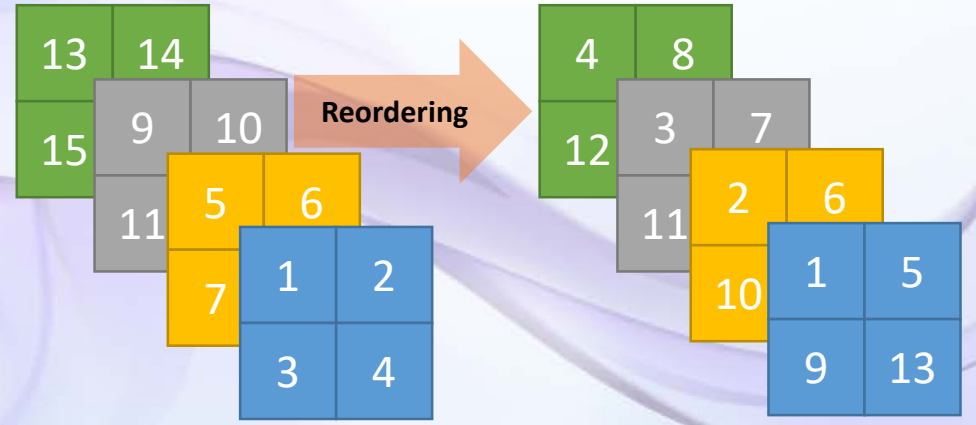
- RS offers several vector operations
  - Inputs must be vectors
- Example: vectorized DOT product
  - Inputs: float4 (128 bits)
  - Output: float
- Currently, maximum vector length is four words
- Supported only in “imprecise” mode




```
float dot(float4 vector1, float4 vector2)
```

Acceleration Strategy

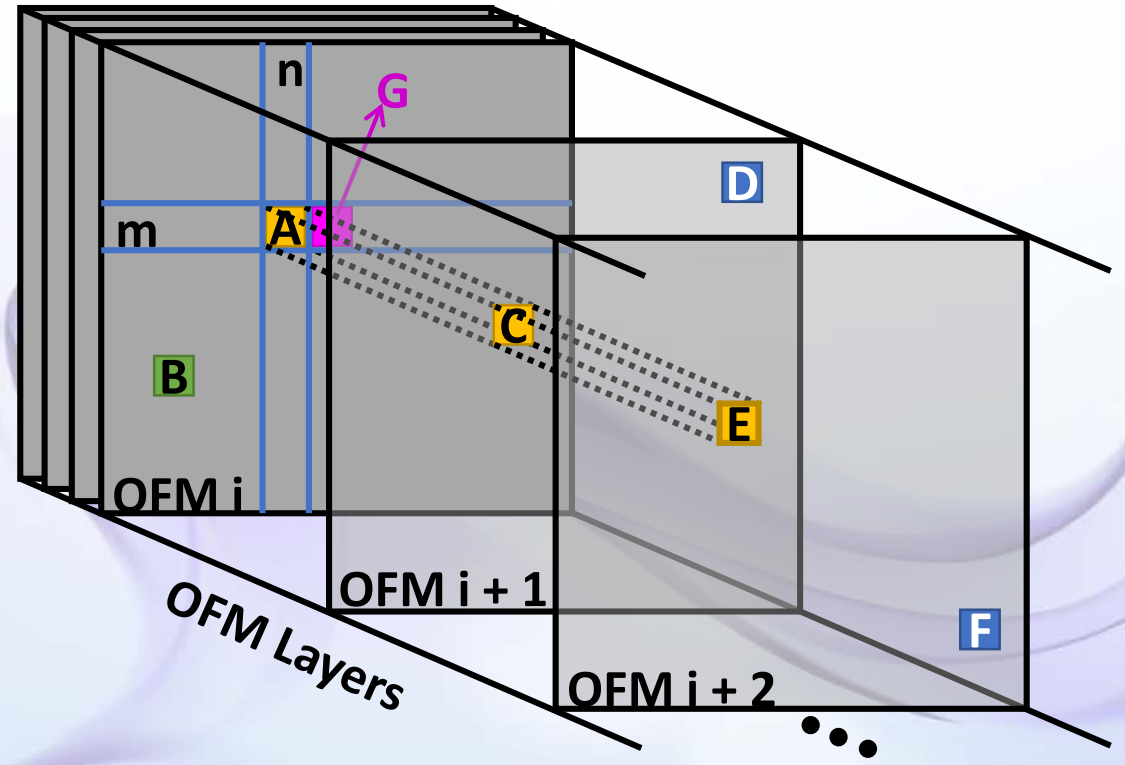
- Parallel computation of output elements
  - Computation of each output pixel is assigned to an independent thread.
- Sub-word parallelism
  - Wide ALUs provides support for vectorized operations.
  - Imprecise computing
  - Data reordering
- Kernel Fusion
  - Minimizing the overhead of kernel launch



What is the optimal number of threads to launch per convolutional layer to achieve the fastest execution of a CNN on a given platform?

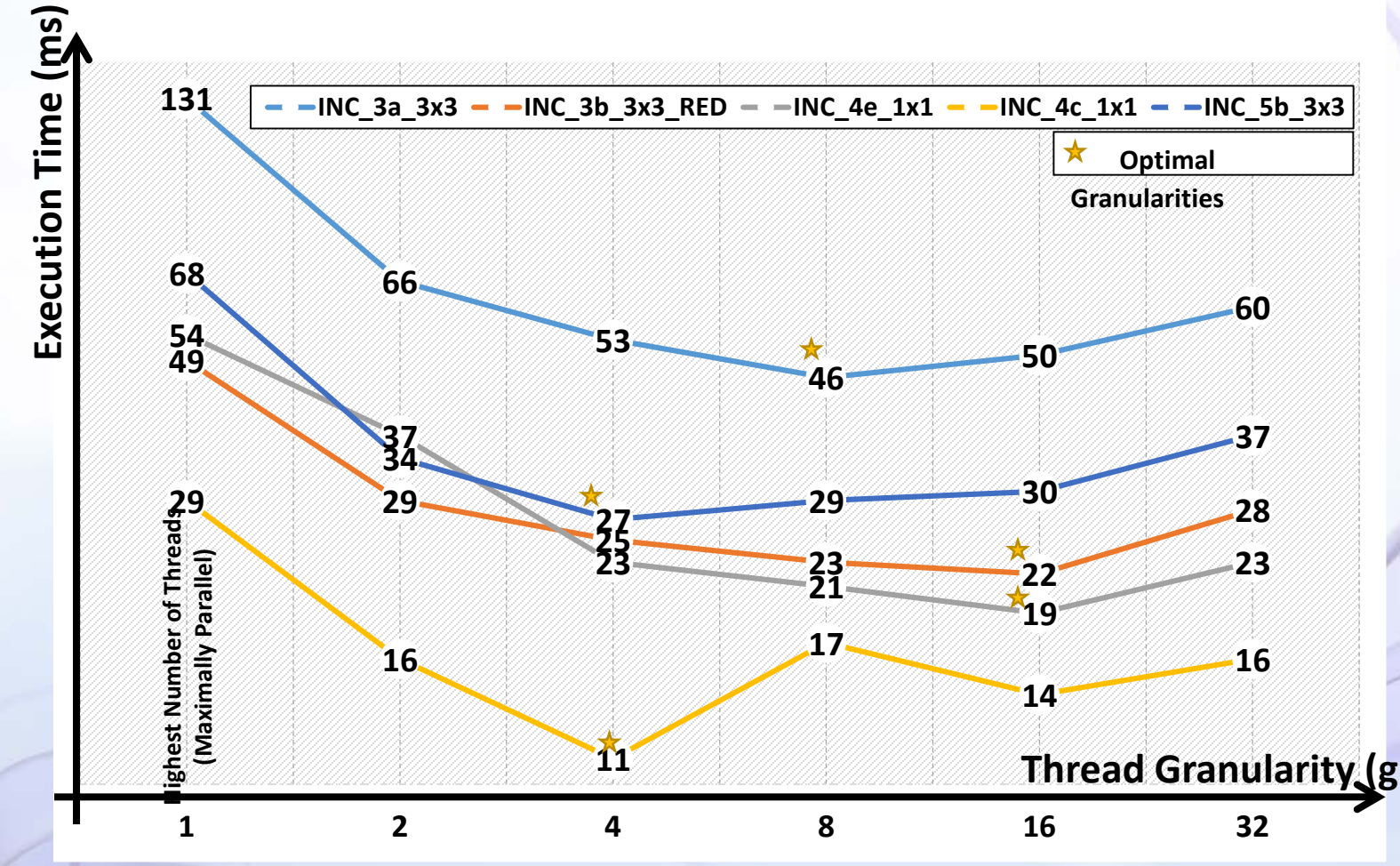
Thread Workload Allocation Policy

- Policy 1
  - Selecting a pixel from another location of the same OFM (e.g., pixel G or B)
  - Number of memory access in granularity *g*
    - $\gamma_{p1} \leq \frac{K^2}{g} + K^2 + 1$
- Policy 2
  - Selecting a pixel from the same location of another OFM (e.g., pixel C or E)
    - $\gamma_{p2} = \frac{K^2}{g} + K^2 + 1$
- Policy 3
  - Naïve selection (e.g., pixel D or F)
    - $\gamma_{p3} = 2K^2 + 1$



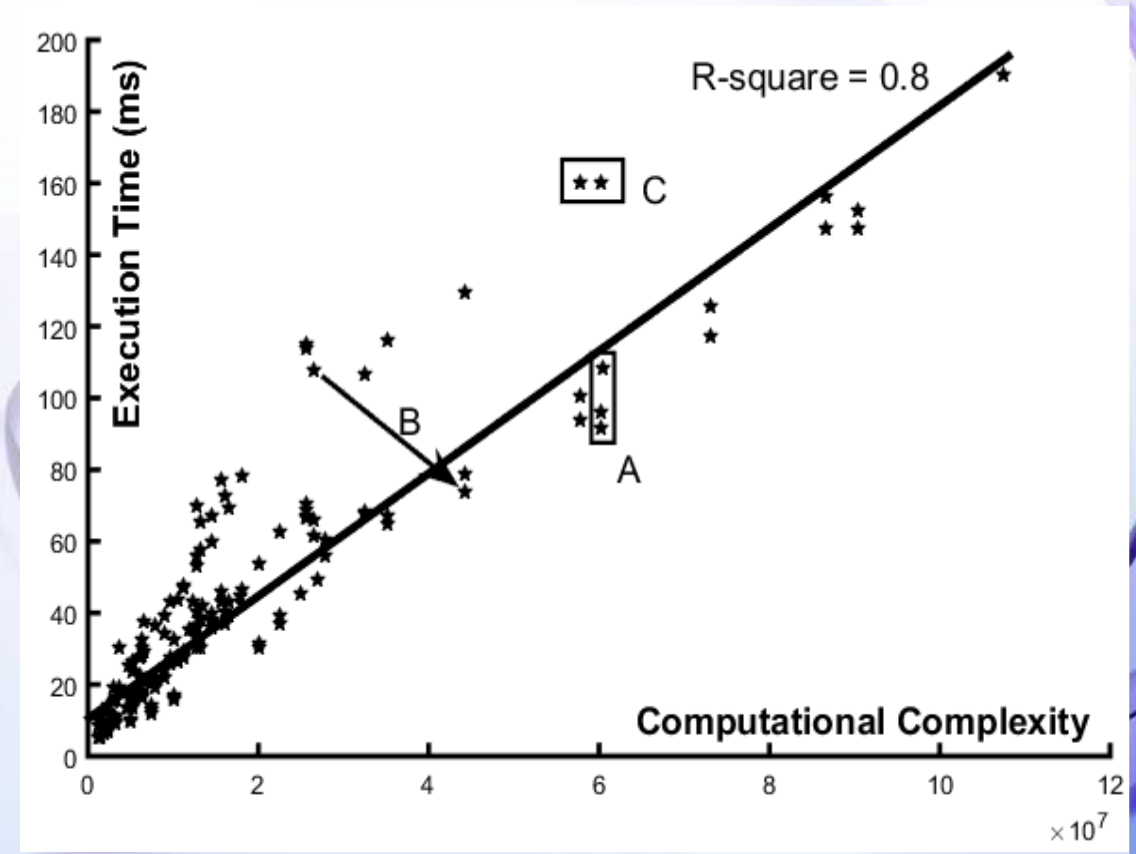
Data-Driven Approach to Granularity Optimization

- Execution time of five layers of GoogLeNet under different thread granularities on Nexus 5.
  - Average over many runs
- No single thread granularity is globally optimal.



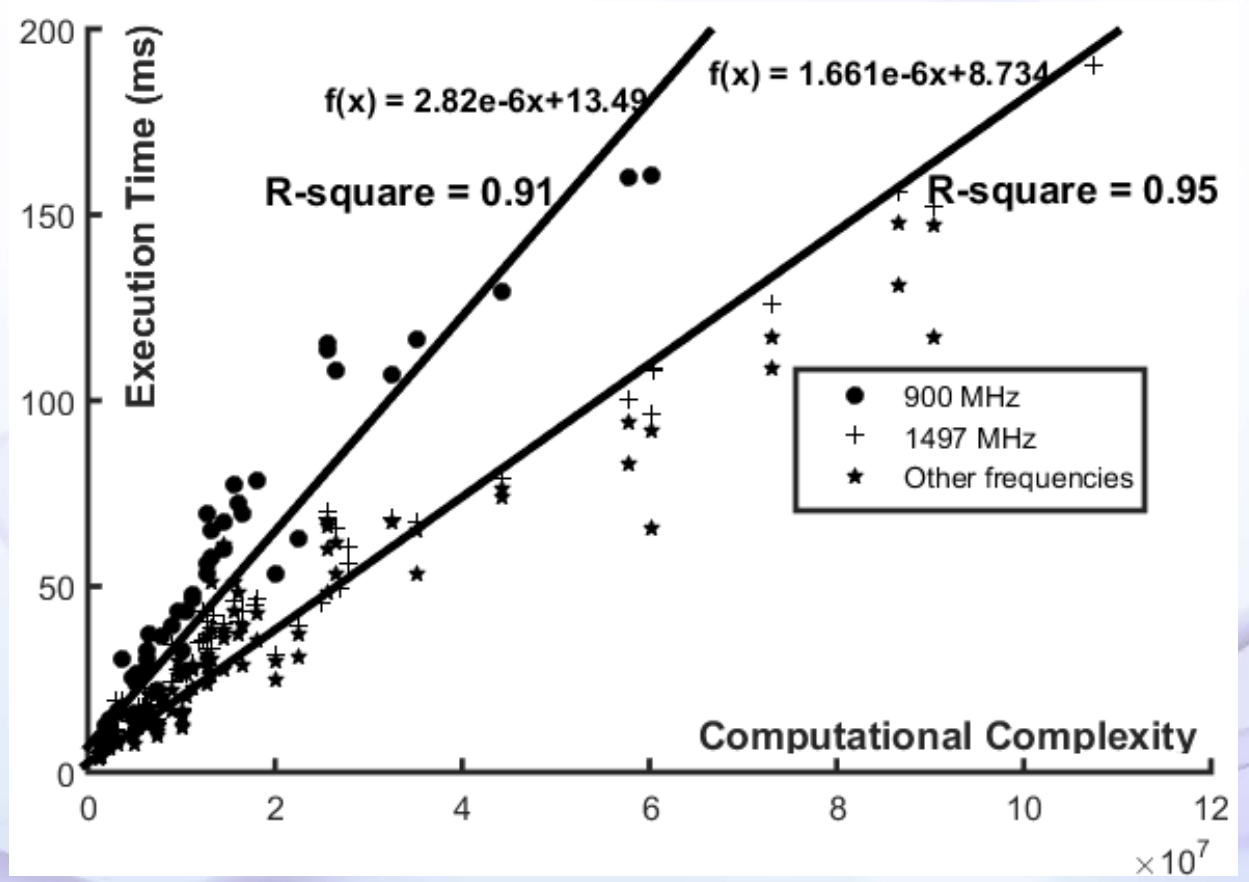
Runtime of Convolutional Layers

- Hypothesis
  - Runtime of a CNN layer can be estimated as a function of it's computational complexity.
  - Computational Complexity (CC) = Number of MAC Operations
    - $CC = \frac{N \times W_{in} \times H_{in} \times K^2 \times M}{S^2}$
- Irregularities
  - A, B, and C
- Our hunch
  - Resource sharing and multitasking
  - Dynamic Voltage and Frequency Scaling



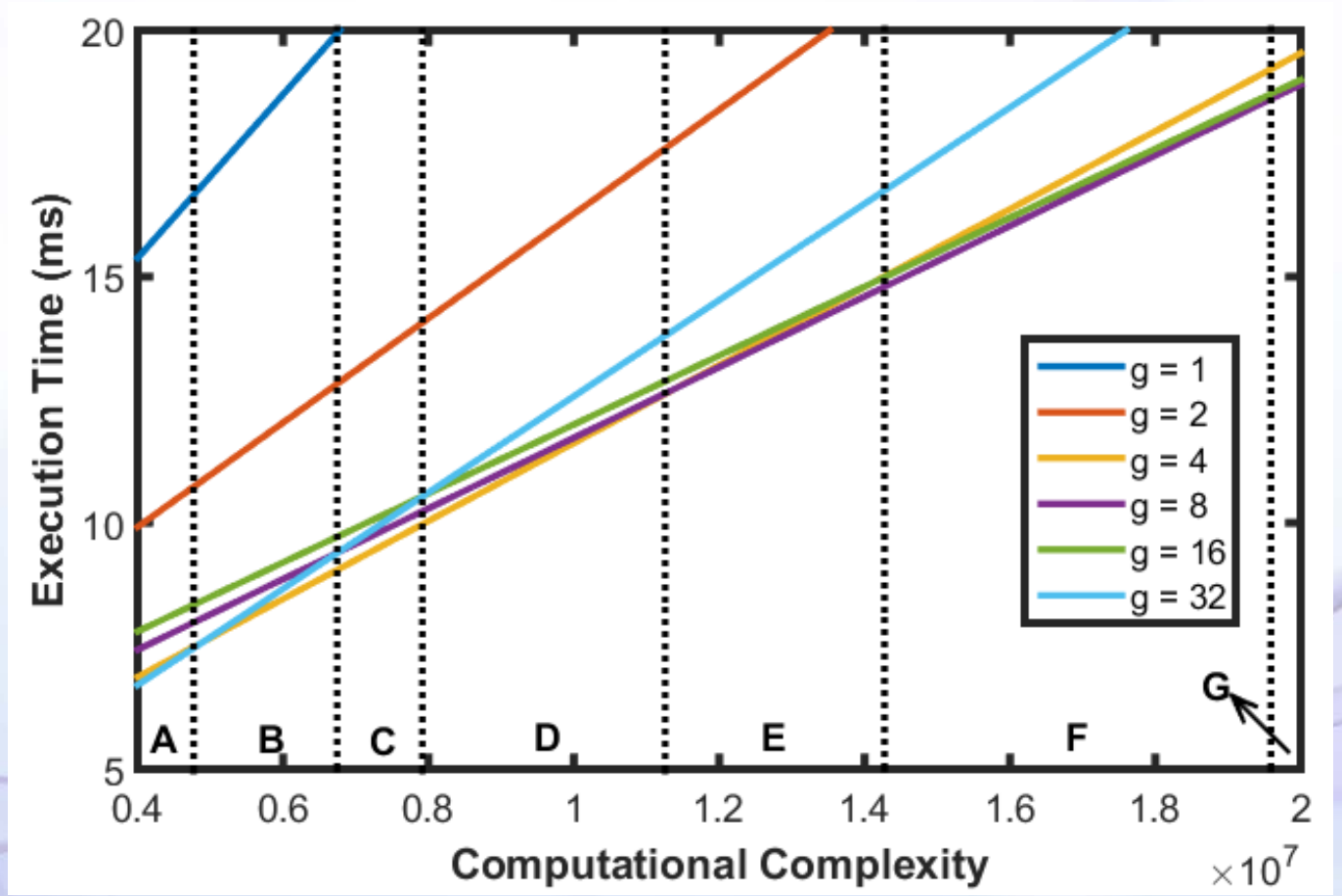
Frequency-based Clustering

- Rooted the device & enforced a fixed frequency
  - Frequency-based clustering improves the regression performance.
- Leads to a (simple) data-driven model for predicting the execution time of other CNNs under different thread granularities.



Data-Driven Model for Granularity Selection

- Linear regression results for different thread granularities.
- Optimal thread granularity varies based on the value of computational complexity.

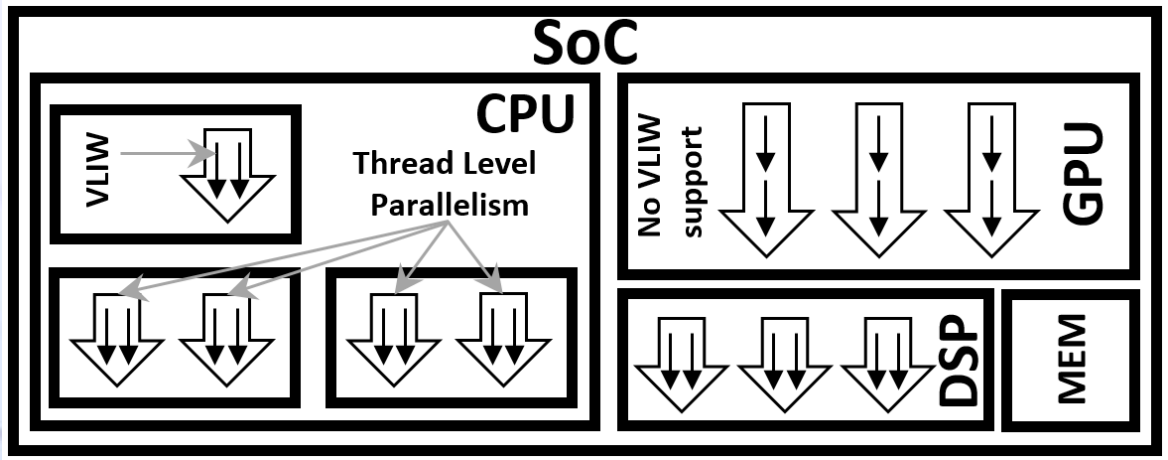


Experimental Setup: Choice of CNN and Platform

- Choice of neural network
  - AlexNet: Widely used for performance comparison in different CNN acceleration work.
  - SqueezeNet: Tailored towards resource-constrained IoT applications.
  - GoogLeNet: The Inception family has state-of-the-art performance in classification.
- Hardware
  - Google Nexus 5, Google Nexus 6P
- Framework
  - RenderScript

Experimental Setup: RenderScript

- RenderScript
  - A frame work for execution of computation intensive tasks on mobile phones.
- Heterogenous Processing
  - RenderScript runtime utilizes all available cores on a SoC to run a parallel software.
- Parallel Programming Paradigms
  - Flexible Single Instruction Multiple Data (SIMD)
  - Sub-word parallelism
- Hardware Abstraction
  - Thread Scheduling is not customizable.

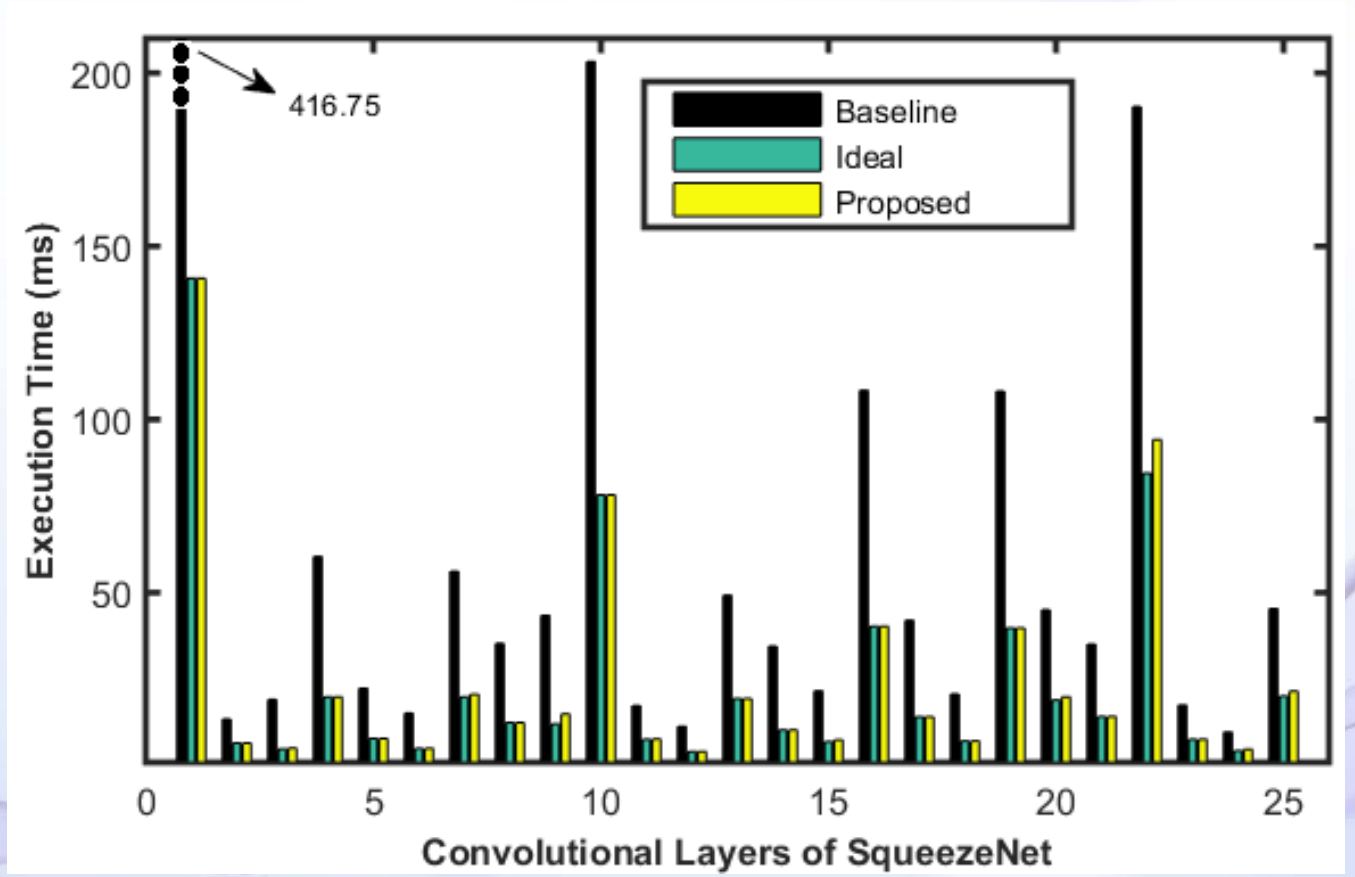


Experimental Setup: Measurement Methodology

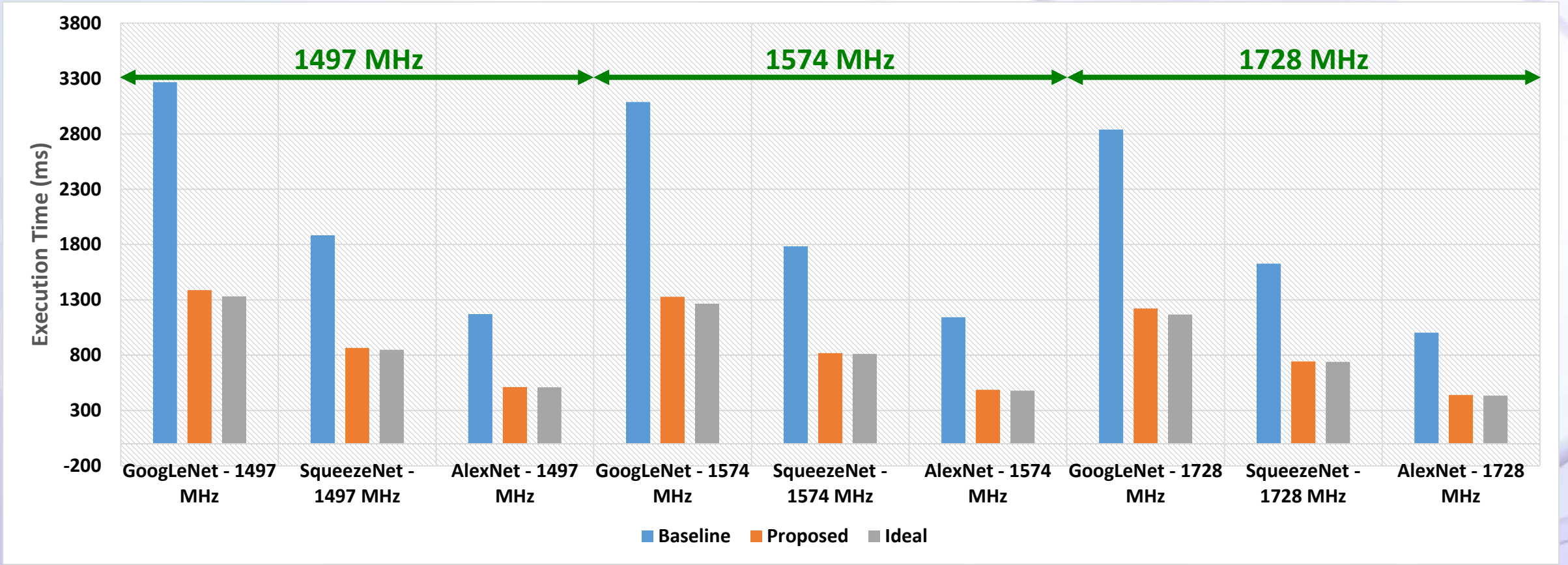
- Baseline
  - Execution time of a maximally parallel program (finest thread granularity)
- Proposed
  - Execution time of a each layer when it is executed using thread granularities found by the proposed algorithm.
- Ideal
  - Each layer has been executed with all possible thread granularities and the execution time of the smallest is measured.
- Average over 100 runs
- Memory flushed between two consecutive experiments.
- The granularity prediction model is trained using GoogLeNet.

Granularity Selection: Model Assessment

- Execution time for each convolutional layer in SqueezeNet using the baseline, ideal, and proposed granularity.
- Experiment setup
  - Frequency: 1497 MHz
  - Platform: Google Nexus 5
- The difference between the performance of the proposed solution and ideal performance is 2%.

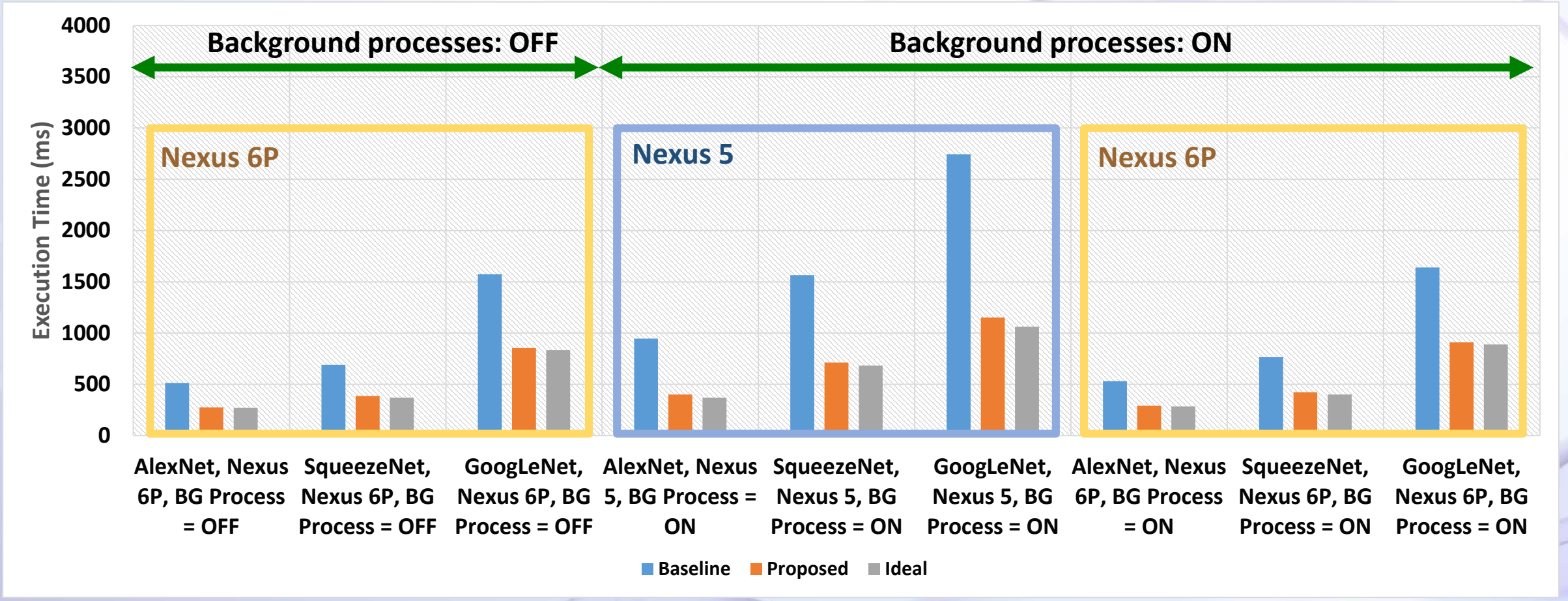


Performance Comparison – Fixed Frequency



- Experiments are performed on Google Nexus 5
- Geometric mean of acceleration: 2.27X
- Geometric mean of difference with ideal: 1.72%

Performance Comparison – Normal Condition

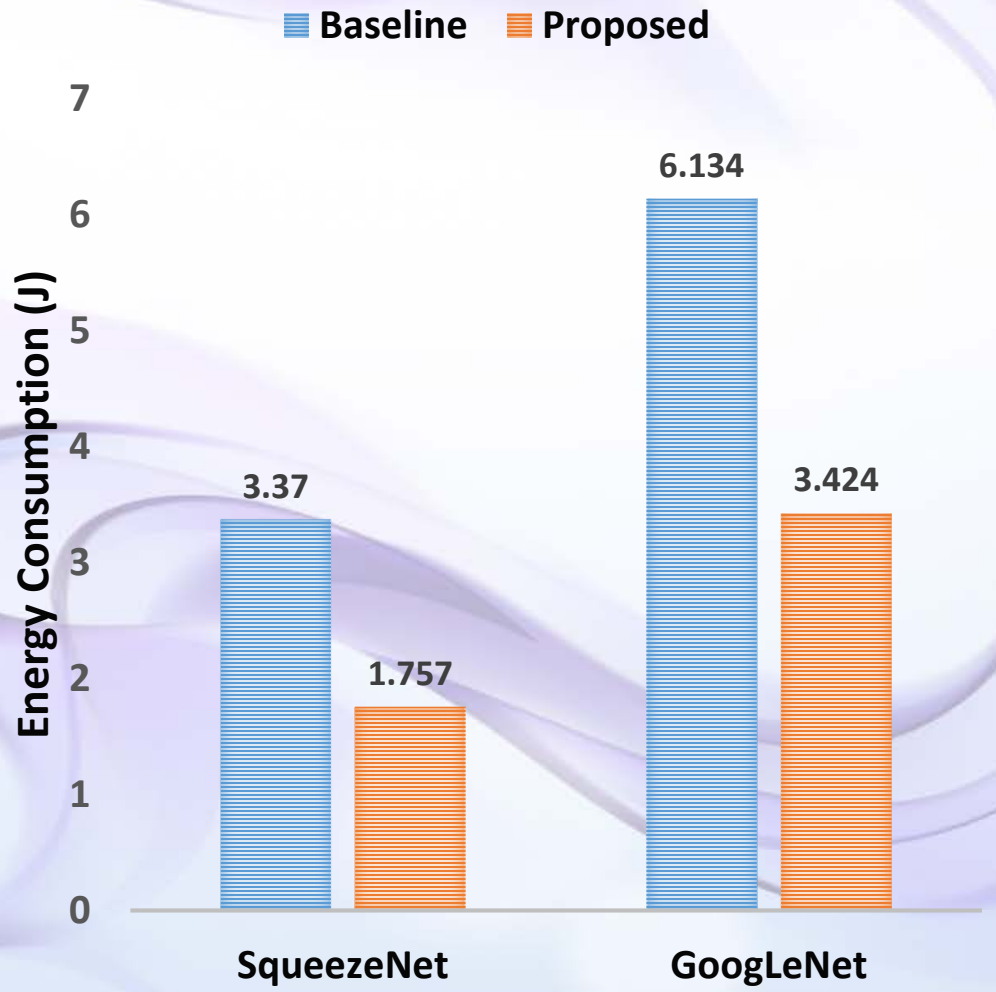


- In all experiments, DVFS was active.
- Granularity prediction models are trained using GoogLeNet.

- Acceleration geometric mean: 1.97X
- Geometric mean of difference with ideal: 3.85%

Impact on Energy Consumption

- Platform: Google Nexus 5
- Profiler: Qualcomm Trepn Power Profiler
- Average over 1000 experiments



Switching Gears …

FPGA

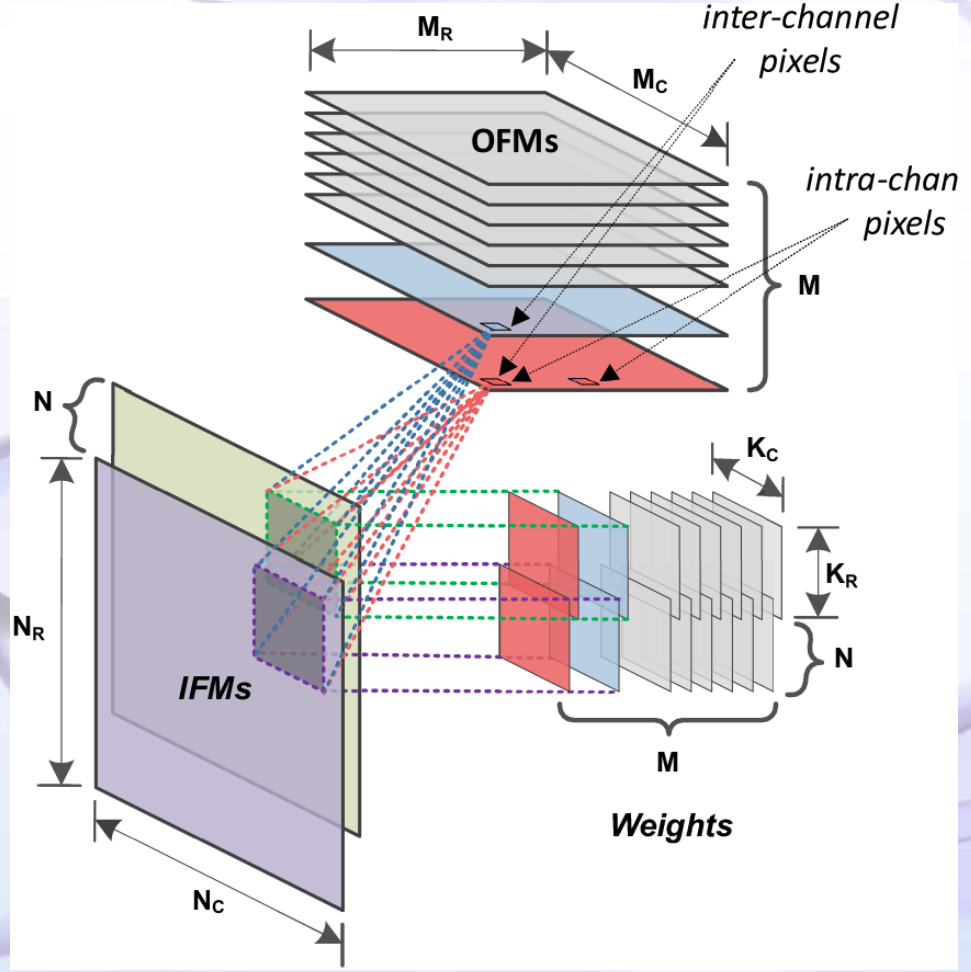
Implementation

Computation of a Convolutional Layer

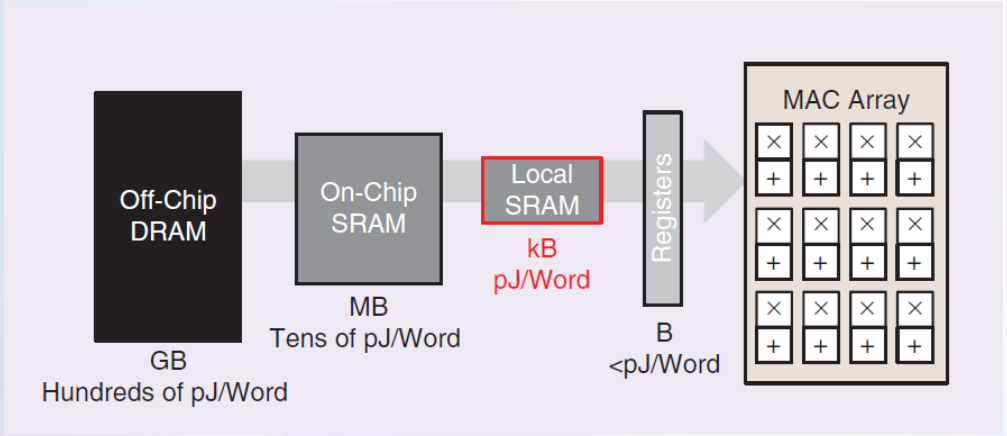
- CNN Layer parameters
  - N: number of Input Feature Maps
  - $N_R \times N_C$: size of Input Feature Maps
  - $K_R \times K_C$: size of weight Filters
  - M: number of Output Feature Maps
  - $M_R \times M_C$: size of Output Feature Maps

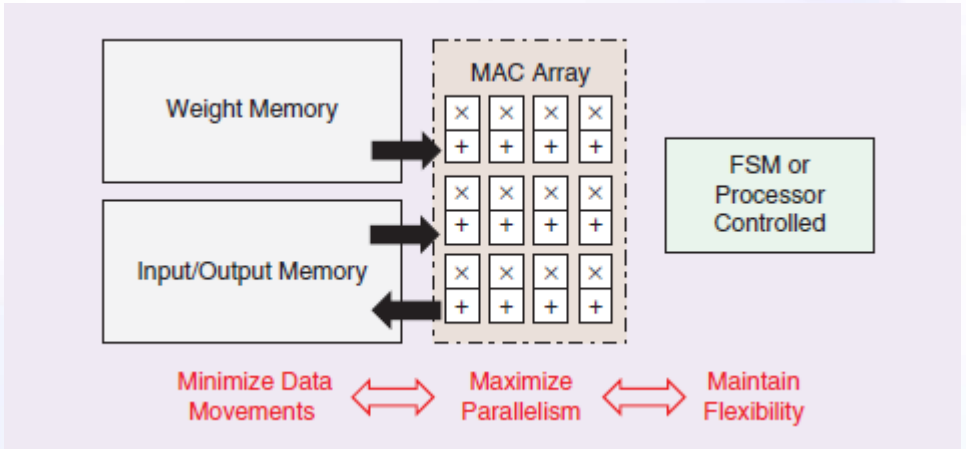
Algorithm 1 Loops involved in a convolution layer

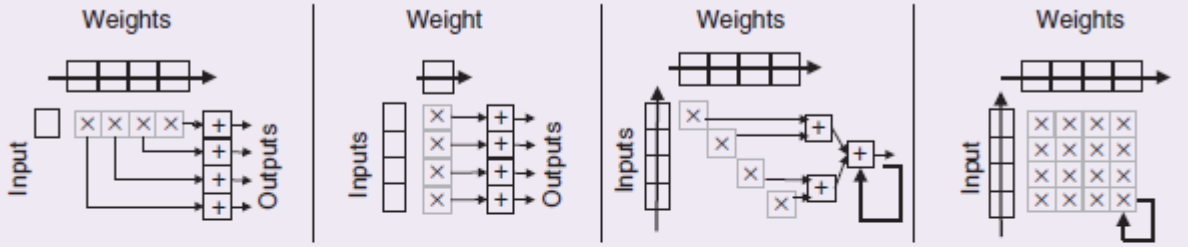
```
1: for ( m = 0, m < M, m += 1) do                                   ▷ :Loop 1
2:     for ( r = 0, r < M_R, r += 1) do                             ▷ :Loop 2
3:         for ( c = 0, c < M_C, c += 1) do                         ▷ :Loop 3
4:             out[r][c][m] = bias[m]
5:             for ( kr = 0, kr < K_R, kr += 1) do                  ▷ :Loop 4
6:                 for ( kc = 0, kc < K_C, kc += 1) do              ▷ :Loop 5
7:                     for ( n = 0, n < N, n += 1) do               ▷ :Loop 6
8:                         out[r][c][m] += in[r × S + kr][c × S + kc][n] × wt[kr][kc][n]
```



Accelerator Design: Driving Principles





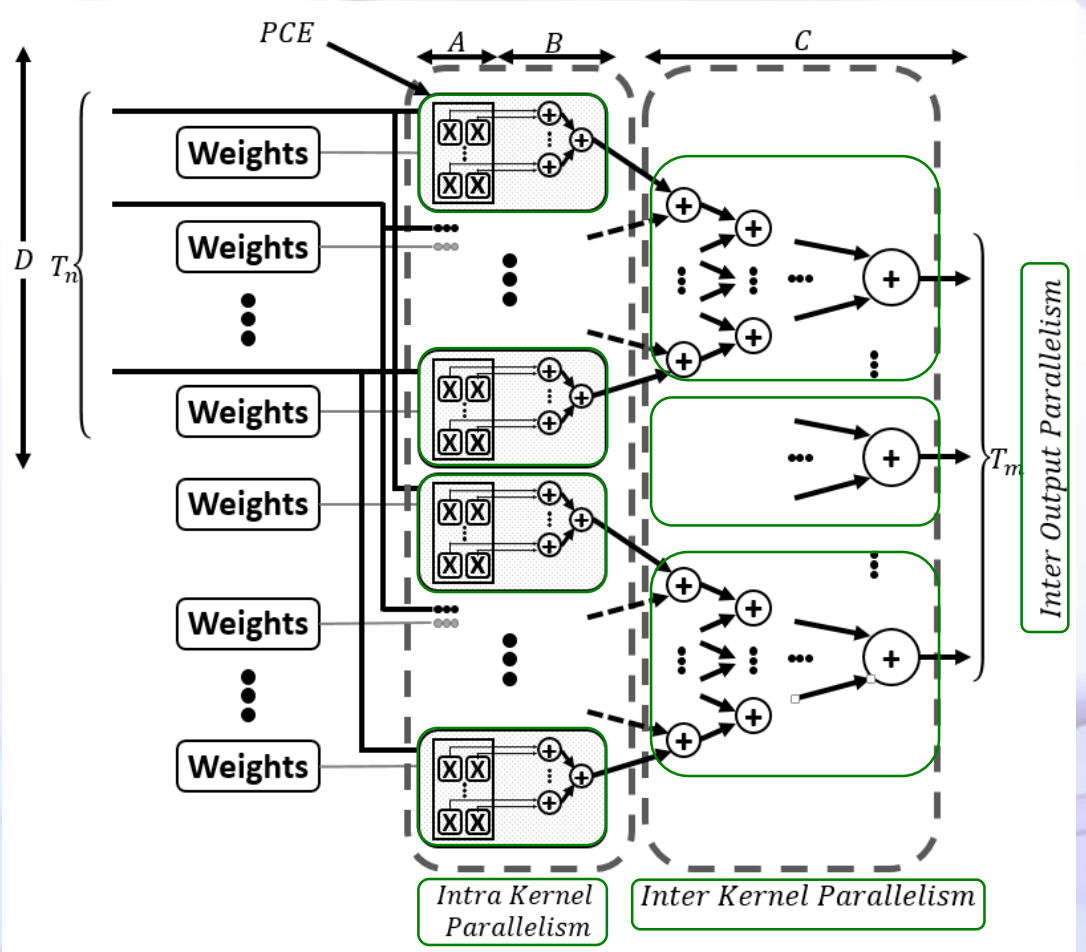


	Input Stationary (Weight Parallel)	Weight Stationary (Input Parallel)	Output Stationary	Hybrids
Input BW	Low	High	High	Medium
Weight BW	High	Low	High	Medium
Output BW	High	High	Low	Medium

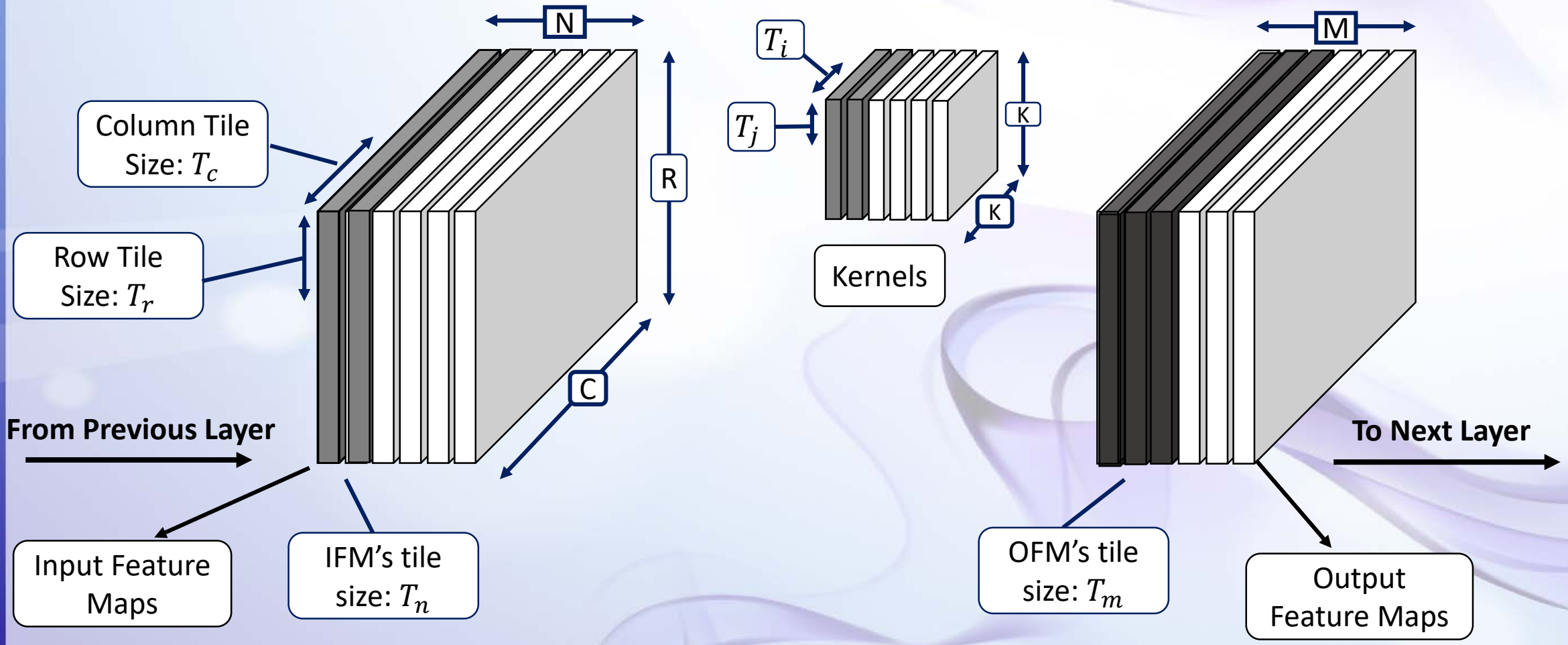
[Verhelst and Moons 2017]

Processing Engine Template

- Intra Kernel Parallelism
  - Parallel Conv. Engine (PCE)
  - For example, 4 Parallel multiplications (T_k) are depicted.
- Inter Kernel Parallelism
  - PCEs with the same weights
- Intra Output Parallelism
  - PCEs with different weights



Tilling in Convolutional Layers



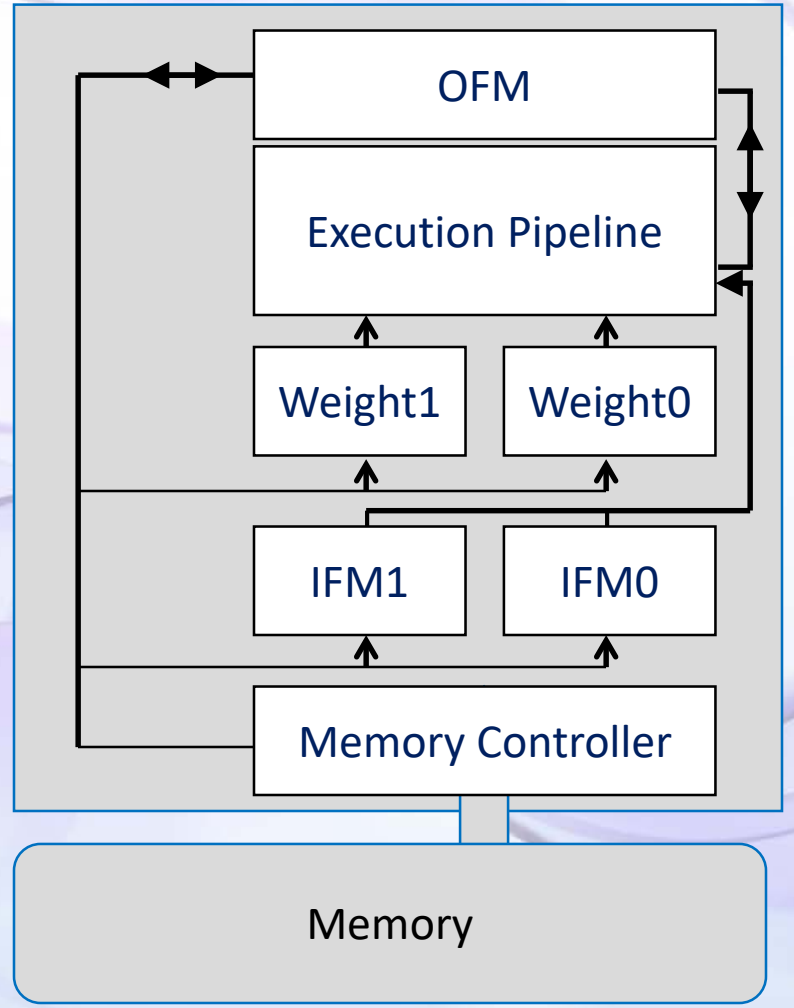
Tiled Data Transfer

- Memory access
  - Time and energy
  - Avoid unnecessary memory access
- Weight's buffer size

$$\beta_{wght} = T_m \times T_n \times T_i \times T_j$$

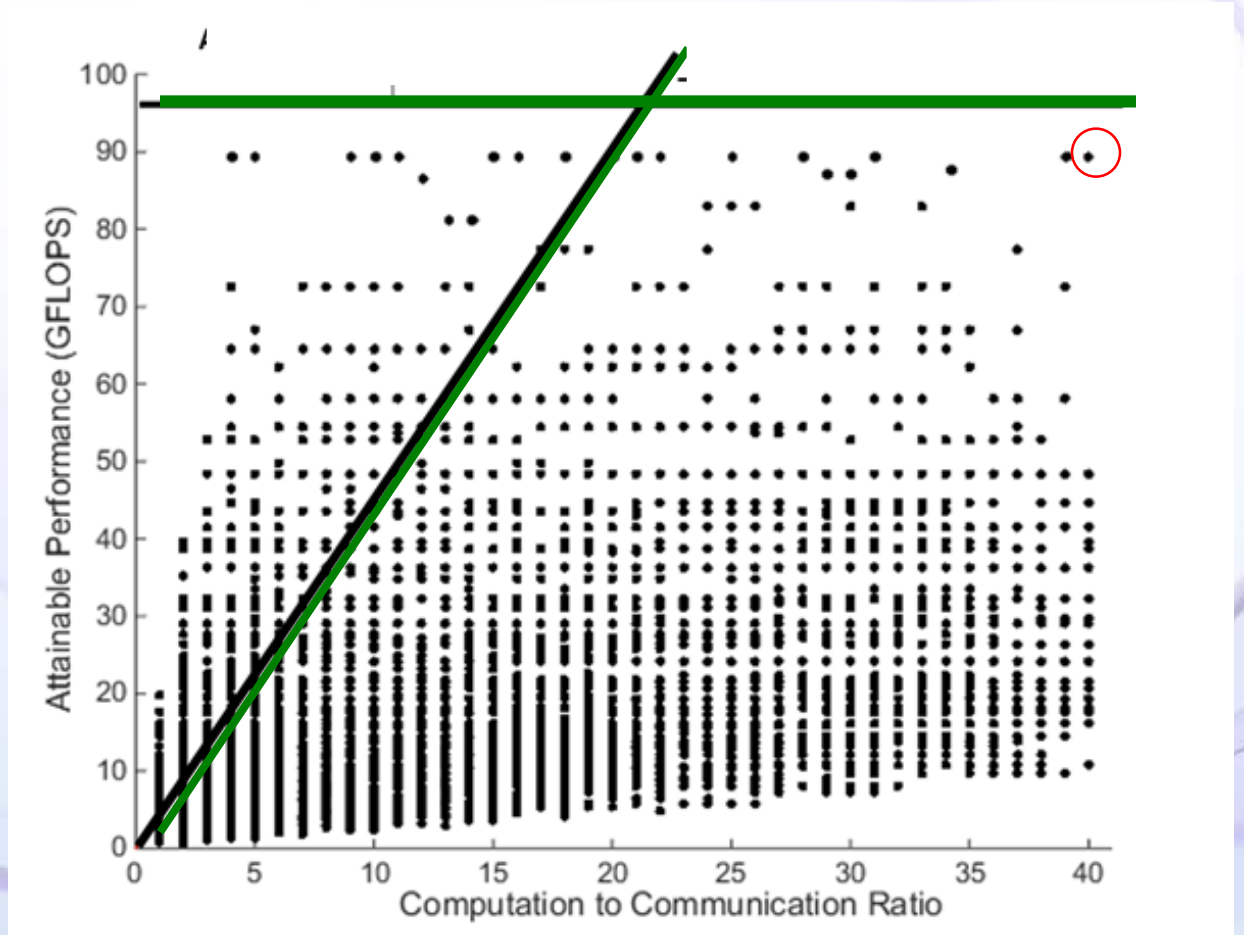
- Number of loads and stores of weights

$$\alpha_{wght} = \frac{M}{T_m} \times \frac{N}{T_n} \times \frac{R}{T_r} \times \frac{C}{T_c} \times \frac{K}{T_i} \times \frac{K}{T_j}$$



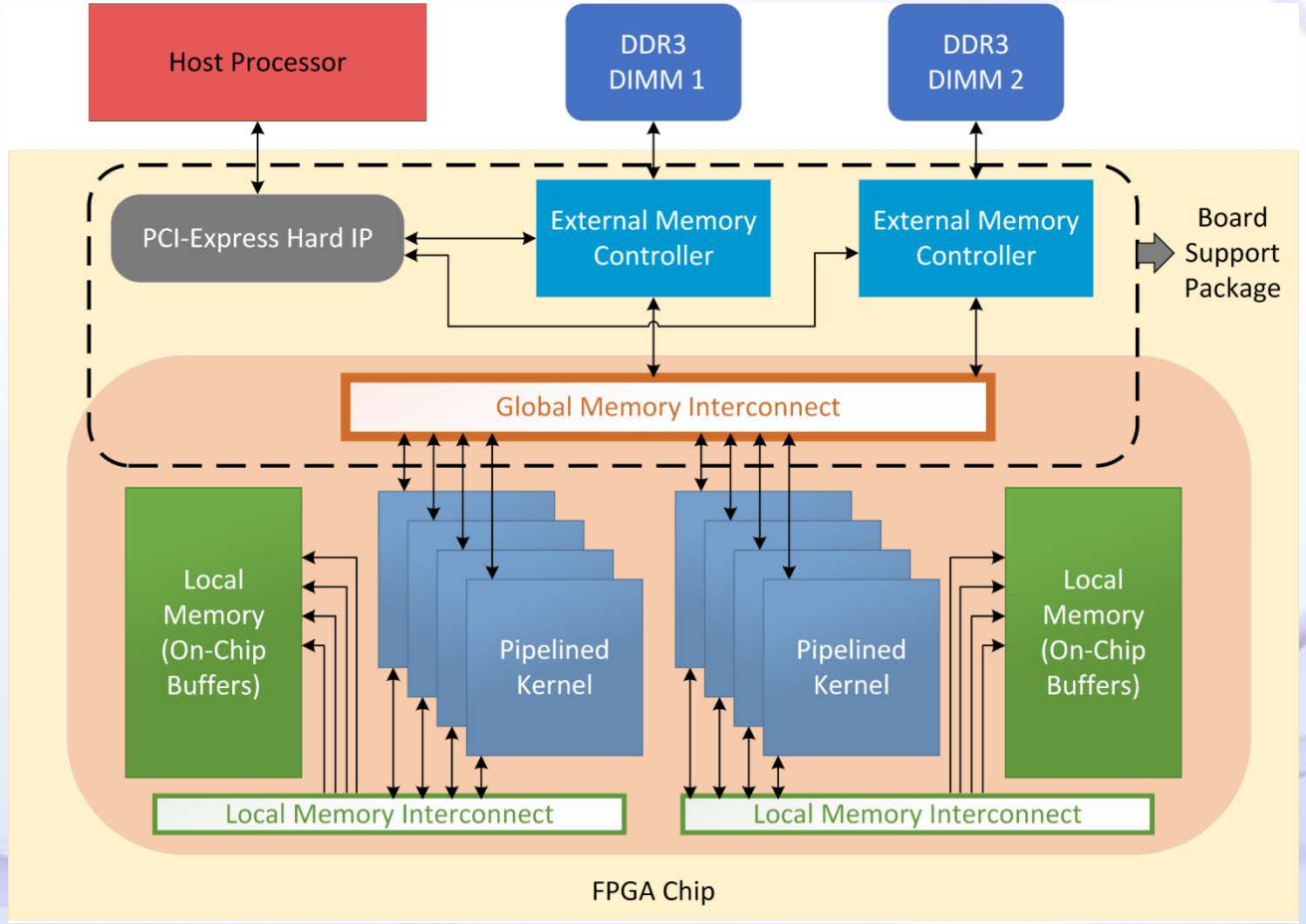
Design Space Exploration

- Objective
  - Maximize performance
- Constraints
  - Memory bandwidth
  - Logic density (area)
  - On-chip storage
- Approach
  - Exploration in the parameter space of T_m, T_n, T_r, T_c, T_i and T_j



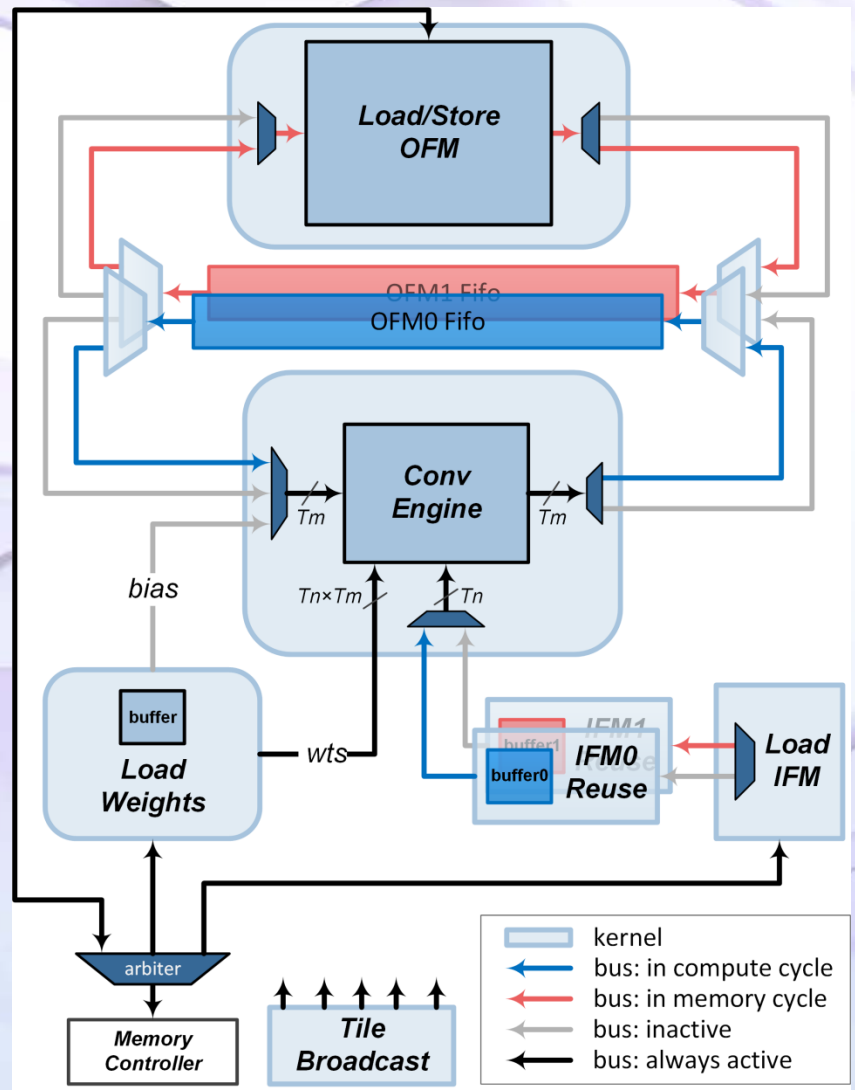
FPGA OpenCL Compute Model Overview

- Host interfaces with FPGA Device via PCI-E
- PCI-E and External Memory (off-chip) Controller IPs are packaged into a pre-synthesized, timing closed package called **Board Support Package** (BSP)
- BSP implements global memory interface while OpenCL Kernels implement on-chip memory interfaces
- OpenCL Compiler converts C/OpenCL Kernel logic into deeply pipelined RTL circuits
- Data transfer and Kernels can be scheduled and launched from Host side using Host OpenCL APIs



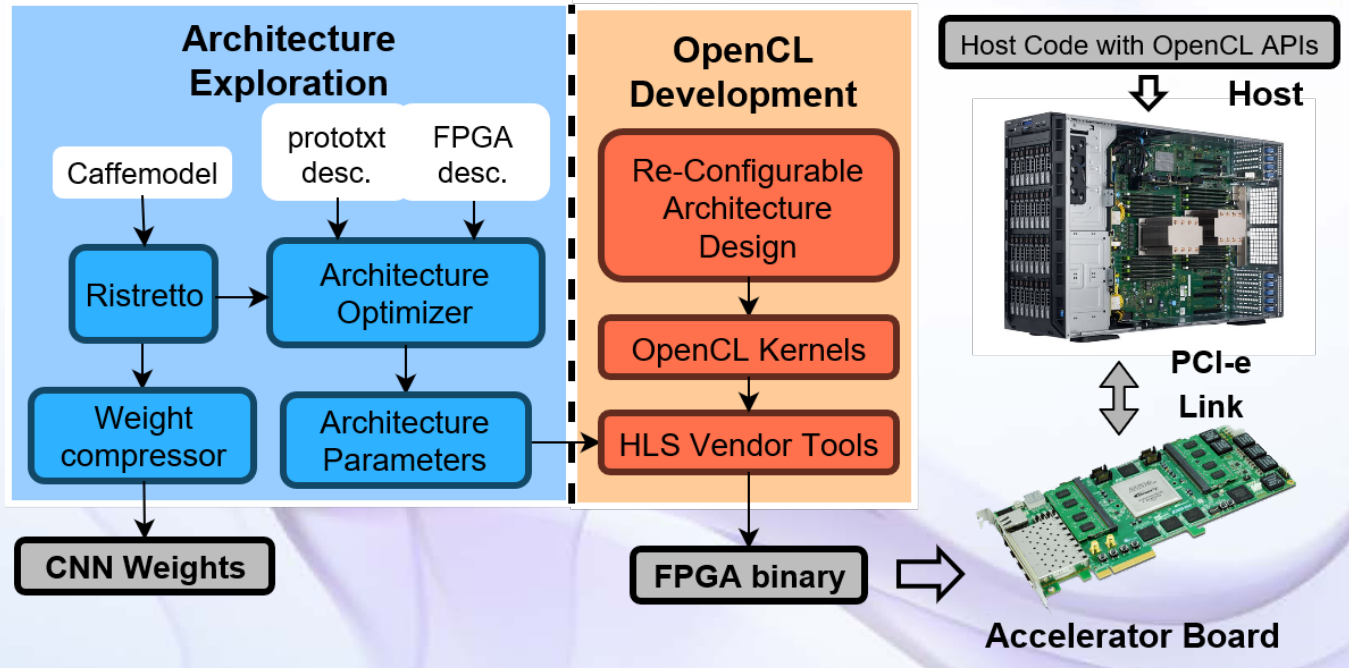
Accelerator Architecture

- Modular architecture with 11 concurrent OpenCL Kernels
- IFMs and OFMs streamed from off-chip memory overlapped with computation (double buffered)
- OFM buffers implemented as FIFOs with OFM partial sums read/written in dual buffers (red and blue blocks)
- Weight Filters are single-buffered due to low bandwidth requirement
- *Conv Engine* efficiently processes all filter sizes in ***Tn×Tm*** SIMD and pipelined parallelism manner
- *Tile Broadcast* Kernel broadcasts layer configuration parameters and schedules Kernel launch



FICaffe Workflow

- FICaffe: a framework that autonomously maps Caffe CNN models to FPGAs
- Architecture exploration for latency optimization:
  - ***Tn***: IFM channel tile
  - ***Tm***: OFM Channel tile
  - ***Tr***: IFM/OFM buffer depth tiles
- Model compression/quantization via Ristretto. One time transfer of weights is done to device DRAMs
- Optimized OpenCL library is synthesized and binary is downloaded to FPGA
- Real-time CNN inference is performed from Host by continuously transferring image data to device



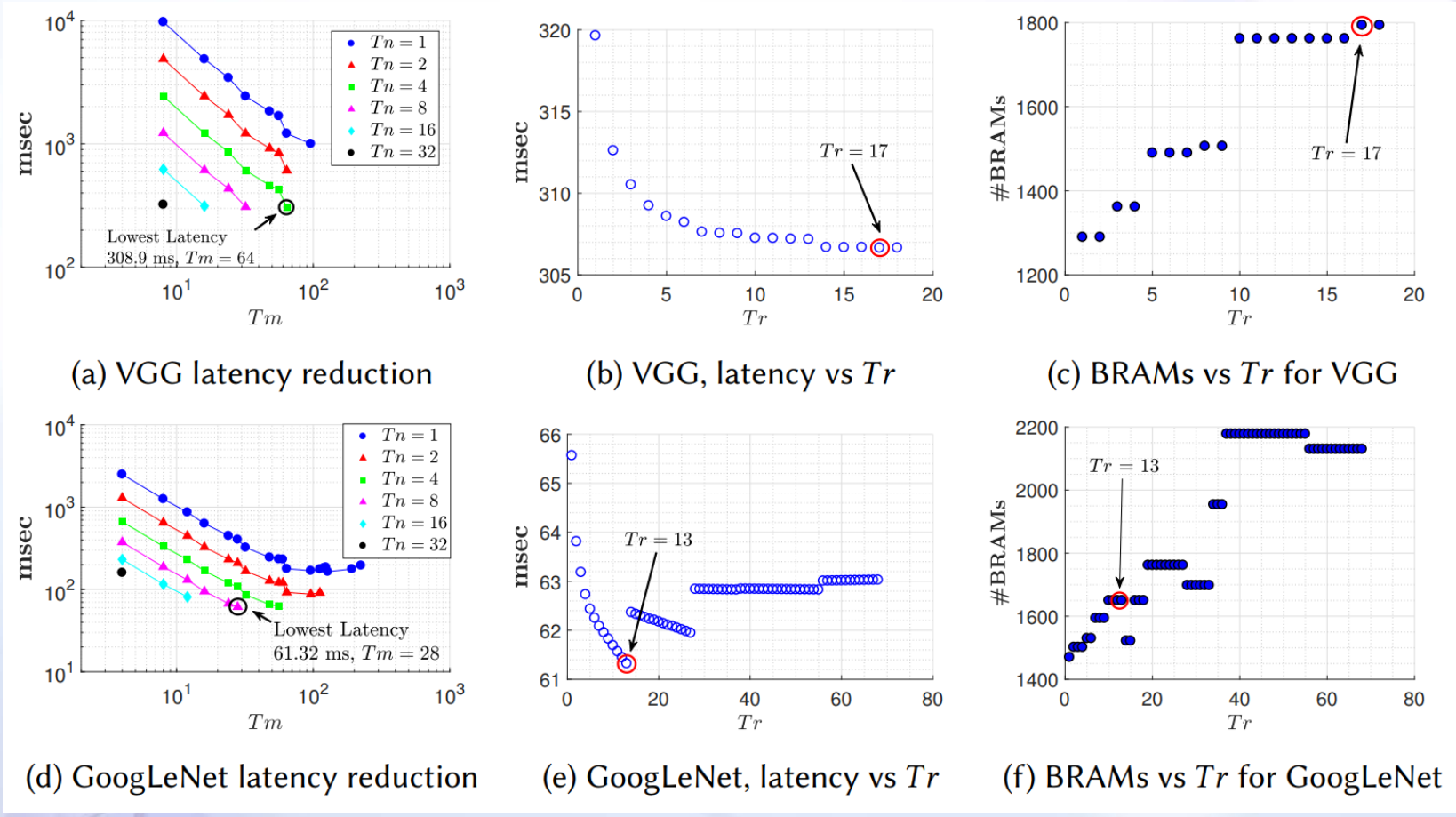
Architecture Exploration

(a) & (d) Variation of runtime latency with ***Tn*** (varies as power of two, different colors) and ***Tm*** (horizontal axis)

(b) & (e) Latency is further optimized based on on-chip cache size (controlled by ***Tr***)

(c) & (f) Variation of BRAM resource utilization with ***Tr***

***Note**: Optimal values are circled*



FICaffe Performance

CNN	#Ops (GOPs)	Tiles *<Tn, Tm, Tr>*	Freq (MHz)	Estimated, Scaled Latency (ms)	Measured Latency (ms)	Diff %
AlexNet	1.45	4, 64, 14	195	18.56, 19.04	20.59	7.53%
VGG	30.76	4, 64, 17	185	306.68, 331.57	339	2.19%
SqueezeNet	1.66	4, 56, 26	195	33.24, 34.13	37.73	9.54%
GoogLeNet	3.16	8, 28, 13	209	61.33, 58.69	62.88	6.66%

- Results on DE5-Net board with a Stratix V GXA7 FPGA, 4 GB DDR3 DRAM @ 800MHz and PCI-E Gen3 x8 interface with the host PC.
  - host is a 6-Core Intel's Xeon CPU E5-2603 v3 @ 1.60GHz, with 16 GB RAM.
- Estimated Latency is based on 200 MHz operation assumption.
- Full CNN runtime estimates are accurate within 10% difference with on-board tests.
- Performance measured with **Batch Size** = 1, i.e., runtime reflects latency/image.

Comparison with State of the Art

	ICCAD'16 [4]		FPGA'17 [5]	FCCM'17 [6]	FPGA'17 [7]	FPGA'17 [8]	FICaffe	
Framework	Xilinx HLS		RTL	OpenCL+RTL	OpenCL	OpenCL+Sys V.	OpenCL	
Precision	fixed, 16		fixed, 8-16	fixed, 16	float, 16	fixed, 8-16	fixed, 8-16	
Platform	Virtex 690t	Kintex KU060	Arria-10 GX1150	Stratix-V GSMD5	Arria-10 GX1150	Arria-10 GX1150	Stratix V GXA7	
CNN (GOPs)	VGG (30.76)		VGG (30.76)	VGG (30.76)	AlexNet (1.45)	VGG (30.76)	AlexNet (1.45)	VGG (30.76)
Freq (MHz)	150	200	150	150	303	385	195	185
#PEs, #DSPs	2833, 2833/3600	1058, 1058/2760	3136, 1518/1518	2048, 1036/1590	2504, 1476/1518	2432, 2756/3036	256, 256/256	
#BRAMs	1248/2940	782/2160	1900/2713	919/2014	2487/2713	1450/2713	1539/2560	1780/2560
Throughput (GOPs/s)	354	266	645.25	364.4	1382	1790	70.42	90.74
CNN Specific	No		Yes	No	Yes	Yes	No	
Input Batching	CL 1, FCL 32		No	Yes	CL 1, FCL 96	No	No	
Latency/img (ms)	2084.16	3236.80	> 47.67	-	> 94.12	> 17.18	20.59	339
PE Eff %	41.60%	62.80%	68.60%	59.30%	91.10%	84.30%	70.50%	95.80%

Conclusions

- (re)emergence of Artificial Intelligence (AI)
  - An essential component: representation learning
  - Deep neural network models do well in many important applications
- The interplay between Algorithms (ML) and Implementations (Systems)
  - Ultimately ideas need to impact the real world => systems constraints play a role
  - Our work on FPGAs and mobile SoCs overviewed
- There are more questions that answers => promising area

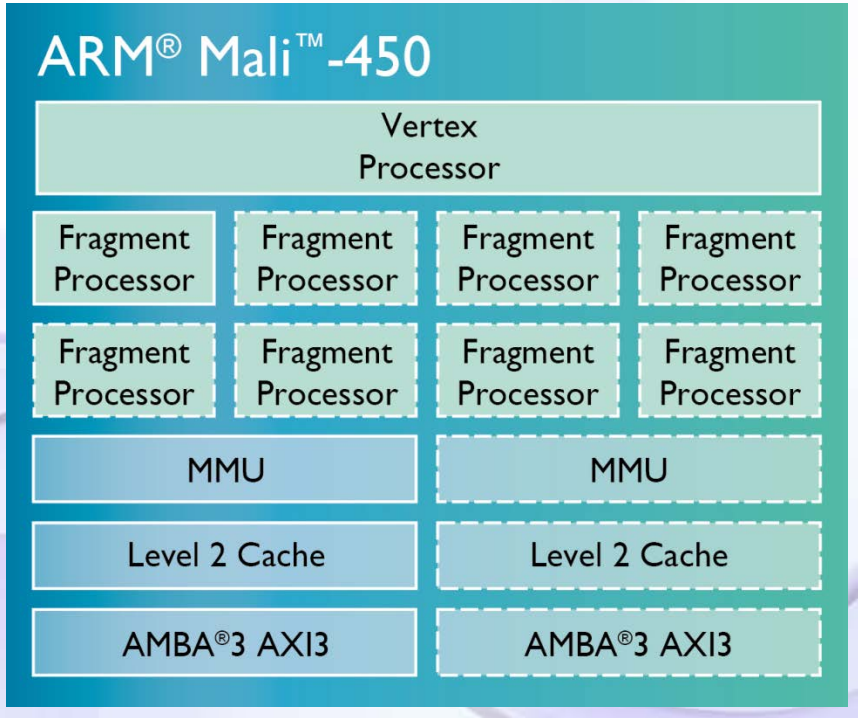
Acknowledgements

- Thanks to
  - Mohammad Motamedi
  - Philipp Gysel
  - Sachin Kumawat
  - Felix Portillo
  - Mohammad Foroozan
  - Trevor Hodges
  - Professors V. Akella, B. Baas and M. Hashemi
- NSF, SRC, CITRIS, UC-Davis, Surround, Xilinx and Altera

Backup Slides

Mobile SoCs and RenderScript (RS)

- An Android framework to run tasks on parall resoruces (CPU, GPU and DSP cores).
- Mobile GPU vs. Server GPU
  - Mobile GPU and CPU share the same memory
  - RS does not offer inter thread communication.
  - RS does not offer texture/image read.

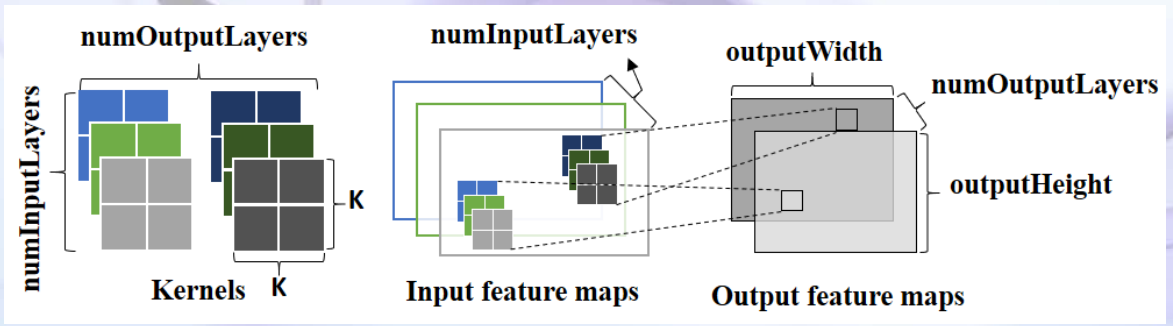


Courtesy of ARM

Sequential Implementation

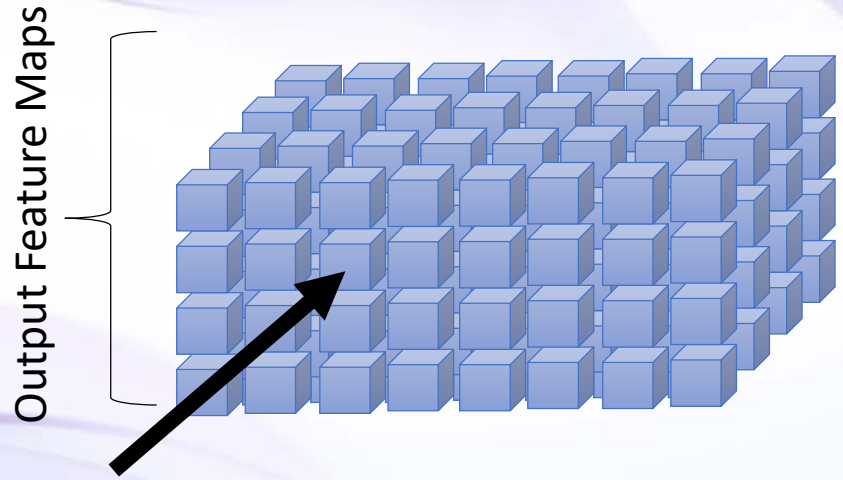
```
for (m = 0; m < numOutputLayers; m++)                    //Loop 1
   for (n = 0; n < numInputLayers; n++)                  //Loop 2
      for (h = 0; h < outputHeight; h++)                 //Loop 3
         for (w = 0; w < outputWidth; w++)               //Loop 4
            for (i = 0; i < kernelHeight; i++)           //Loop 5
               for (j = 0; j < kernelWidth; j++)         //Loop 6
                  out[m][h][w] += in[n][h * S + i][w * S + j] * kernel[m][n][i][j];
```

- Two filter bank, each with three kernels are used to generate two output feature map.



Parallel Computation of Output Feature Maps

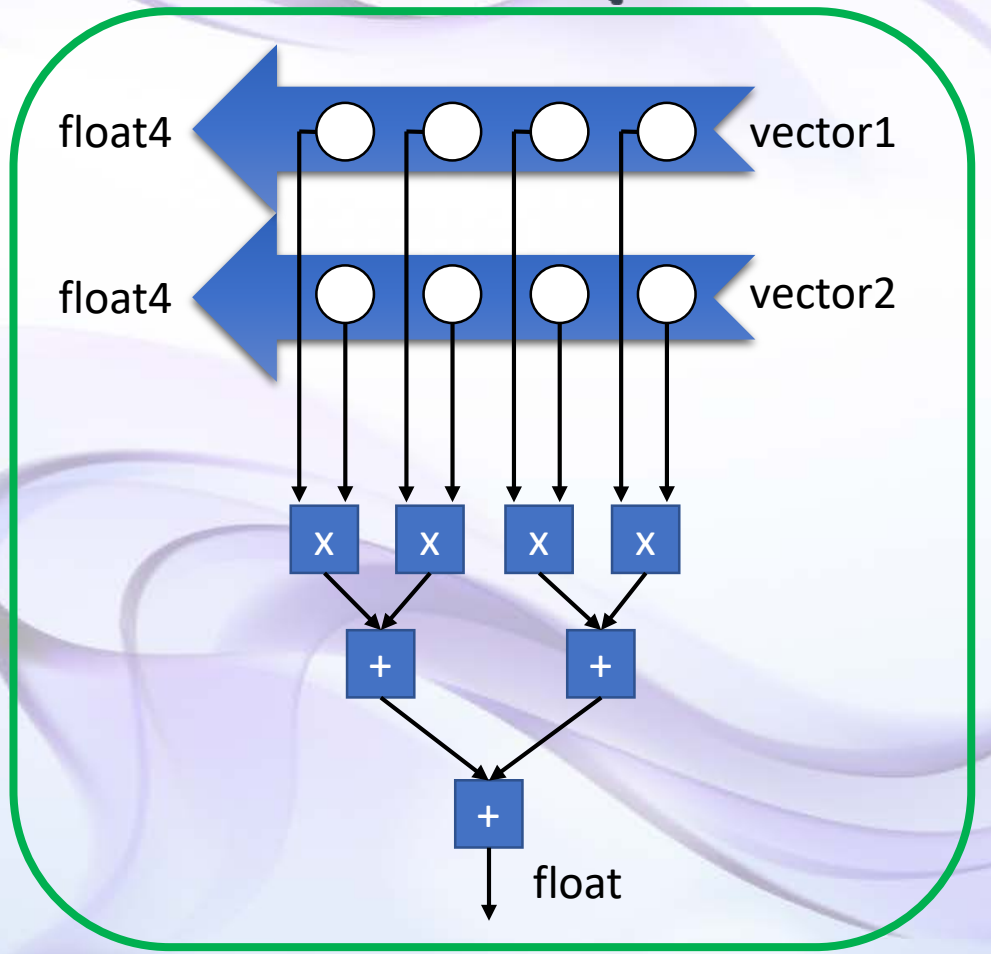
- Output of each layer is a 3D matrix.
- Compute the elements of this matrix in parallel
  - one thread per element.
- Serialization
  - $w = id \,\%\, outputWidth$
  - $h = \left\lfloor \frac{id}{outputWidth} \right\rfloor \% \, outputHeight$
  - $m = \left\lfloor \frac{id}{outputWidth \times outputHeight} \right\rfloor$




```
for (n = 0; n < numInputLayers; n++)
   for (i = 0; i < kernelHeight; i++)
      for (j = 0; j < kernelWidth; j++)
         out += in[n][h * S + i][w * S + j] *
            kernel[m][n][i][j];
```

Vectorized Operations in RenderScript

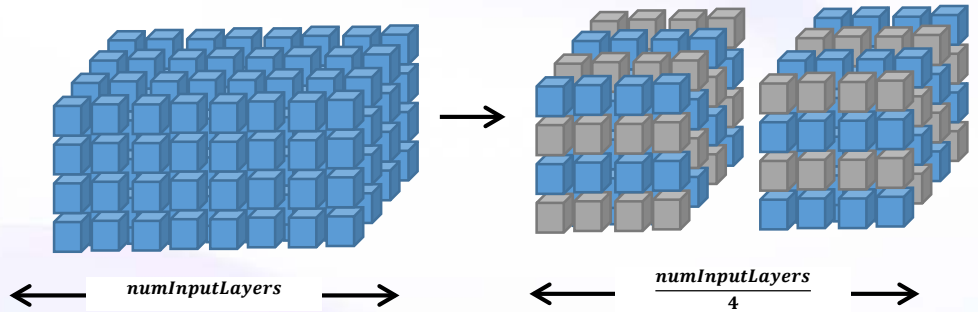
- RS offers several vector operations
  - Inputs must be vectors
- Example: vectorized DOT product
  - Inputs: float4 (128 bits)
  - Output: float
- Currently, maximum vector length is four words.




```
float dot(float4 vector1, float4 vector2)
```

Data Reordering: Visual Representation

- Elements in the same spatial location from consecutive layers form a vector.
- Rather than ‘row major’ format, generate and store data in ‘layer major’ format.
  - Reordering of the input
  - Smart address generation for subsequent layers



Vecorized Convolution

- Function **`rsGetElement_float4`** reads a vector from memory and **`dot`** performs a vectorized dot product.
- Four different layers are processed per iteration

```
int32_t w = x % outputWidth;
int32_t h = (x / outputWidth) % outputHeight;
int32_t m = (x / (outputWidth * outputHeight);
for (n = 0; n < (numInputLayers / 4); n++)
   for (i = 0; i < kernelHeight; i++)
      for (j = 0; j < kernelWidth; j++) {
         idx1 = (w * S + i) +
                outputWidth * (h * S + j) +
                 (outputWidth * outputHeight * n);
         idx2 = //Omitted for simplicity
         float4 in = rsGetElementAt_float4 (input, idx1);
         float4 wght = rsGetElementAt_float4 (weight, idx2);
         out += dot (in, wght);}
```

SqueezeNet

$base_e = 128, incr_e = 128, freq = 2, pct_{x3} = 0.5, and\ SR = 0.125$

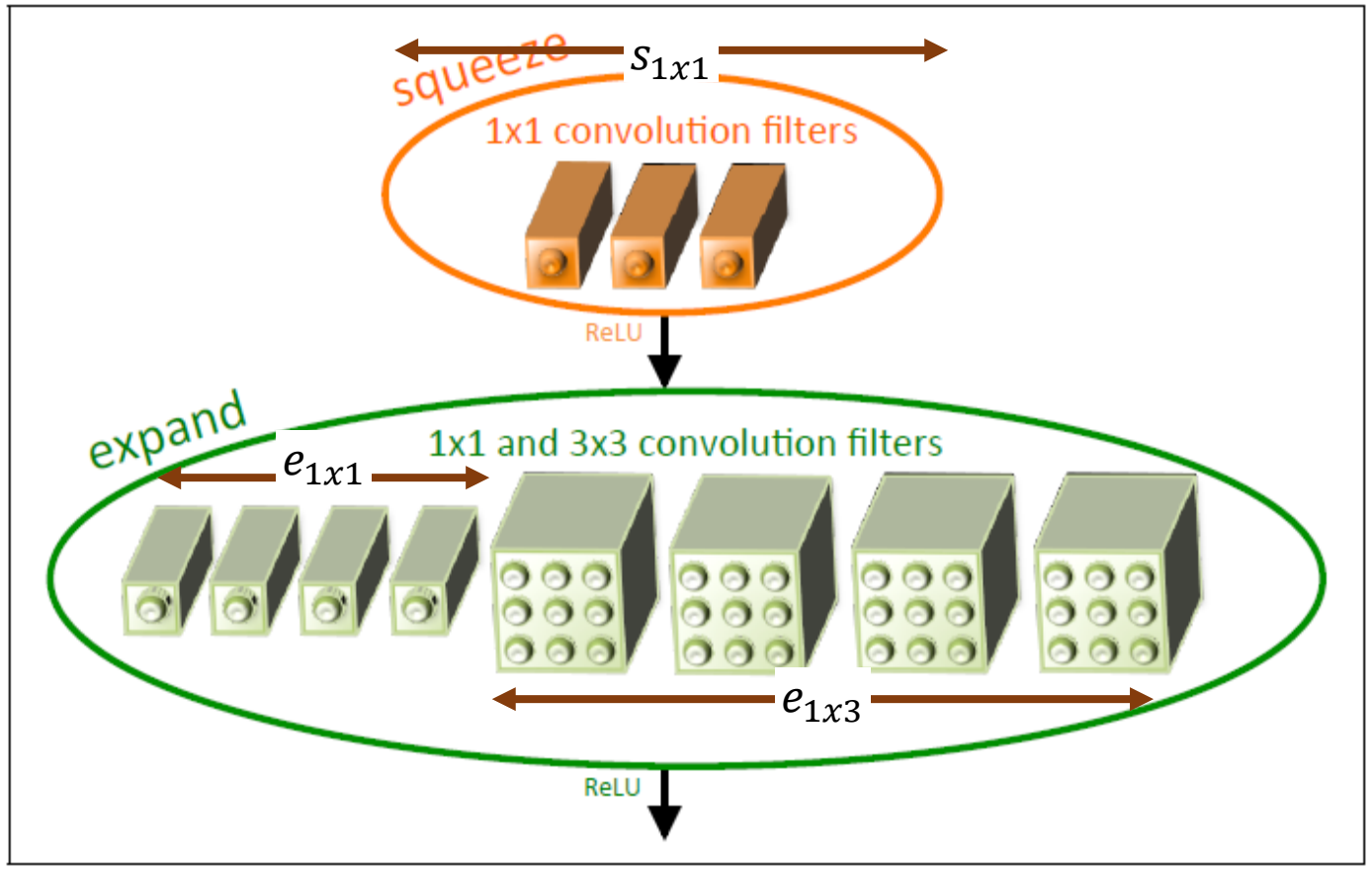
Source: [2]

layer name/type	filter size / stride (if not a fire layer)	s_{1x1} (#1x1 squeeze)	e_{1x1} (#1x1 expand)	e_{3x3} (#3x3 expand)
input image				
conv1	7x7/2 (x96)			
maxpool1	3x3/2			
fire2		16	64	64
fire3		16	64	64
fire4		32	128	128
maxpool4	3x3/2			
fire5		32	128	128
fire6		48	192	192
fire7		48	192	192
fire8		64	256	256
maxpool8	3x3/2			
fire9		64	256	256
conv10	1x1/1 (x1000)			
avgpool10	13x13/1			

Squeezenet Fire Module

- Fire Module
  - Squeeze Layer
    - Only 1×1 filters
  - Expand Layer
    - Mix of 1×1 and 3×3 filters

$$s_{1x1} < e_{1x1} + e_{1x3}$$



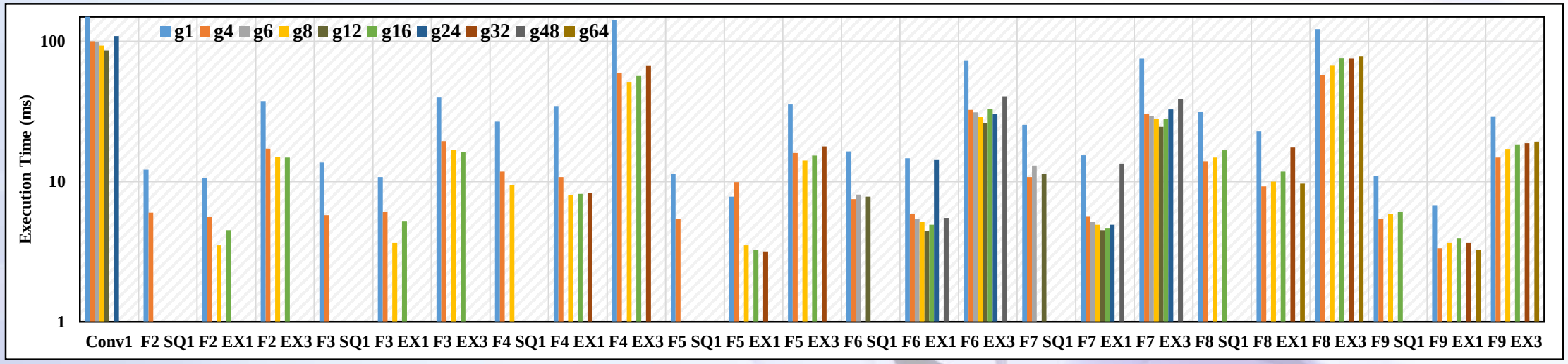
[Iandola et al.]

The effect of thread Granularity

- Tradeoff between the number of threads and the complexity of their workload.
- Higher number of threads usually yields a better execution time.
- Smaller number of thread can improve data reusability.

What thread granularity is optimum (for a given convolution task)?

Effect of Thread Granularity



- Finest thread granularity (g1) does not yield the best performance.
- Optimal granularity depends on target platform and CNN.
  - The graph relates to “Squeeznet on Google Nexus 5”

Pessimal Vs. Optimal Granularities

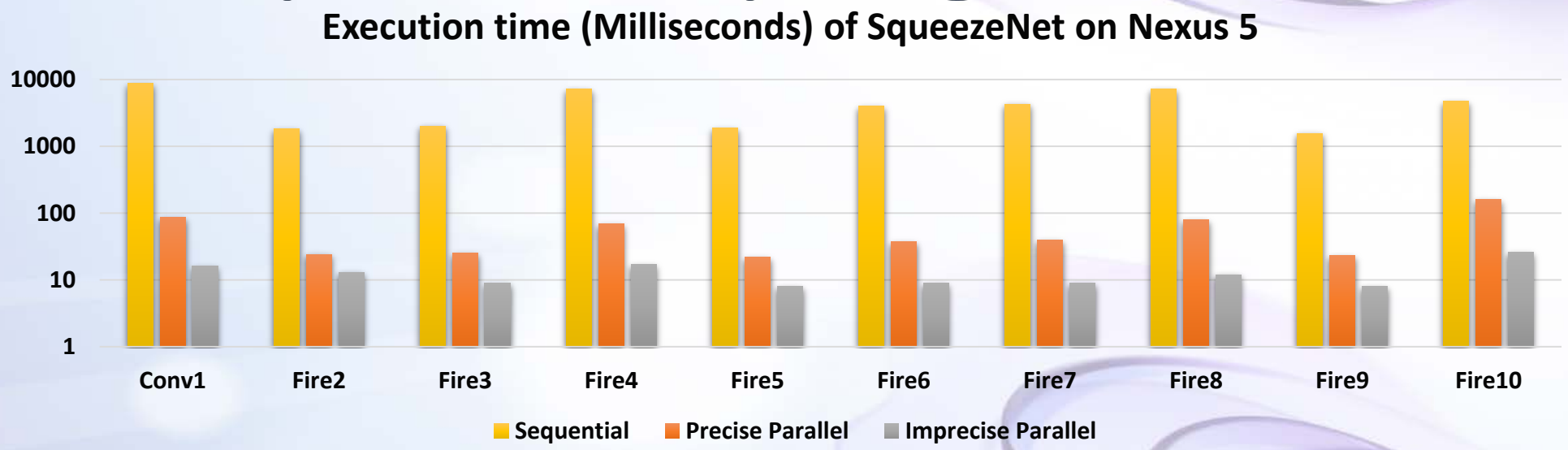
	Fire Layers (ms)			Convolutional Layers (ms)			Speedup
	Optimal	Pessimal	Speedup	Optimal	Pessimal	Speedup	
Galaxy S7	268.94	852.36	3.17X	159.55	227.46	1.43X	2.52X
Nexus 6P	234.72	543.01	2.31X	134.91	205.19	1.52X	2.02X
Nexus 5	323.6	828.35	2.56X	247.59	475.84	1.92X	2.28X

- Thread granularity impacts the execution time drastically.

Imprecise Computing

- RS offers two modes for imprecise computing.
  - Relaxed Floating Point
    - Enables flush to zero for denorms and round towards zero
  - Imprecise Floating Point
    - Enables flush to zero for denorms and round towards zero
    - Operations resulting in -0.0 can return +0.0 instead
    - Operation on INF and NAN are undefined

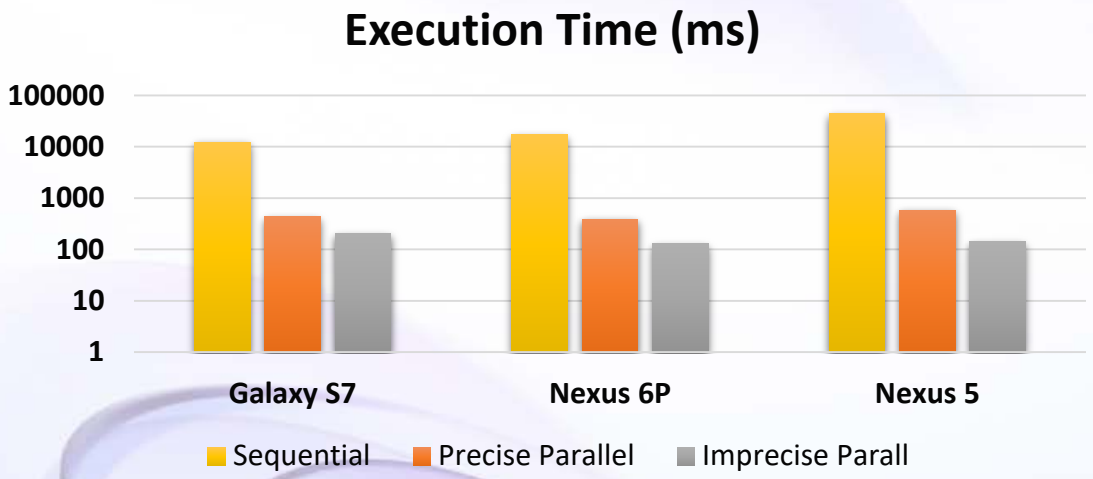
Imprecise Computing Results: Nexus 5



- Parallelism effect on different layers: Min = 29X, Max = 107X
- Imprecise computing effect: Min = 138X, Max = 573X
- Functionally, the classification accuracy remains intact
  - Tested on the first 10000 samples of ILSVRC 2012.

Speedup

- Minimum Speedup: 60X
- Maximum Speedup: 310X



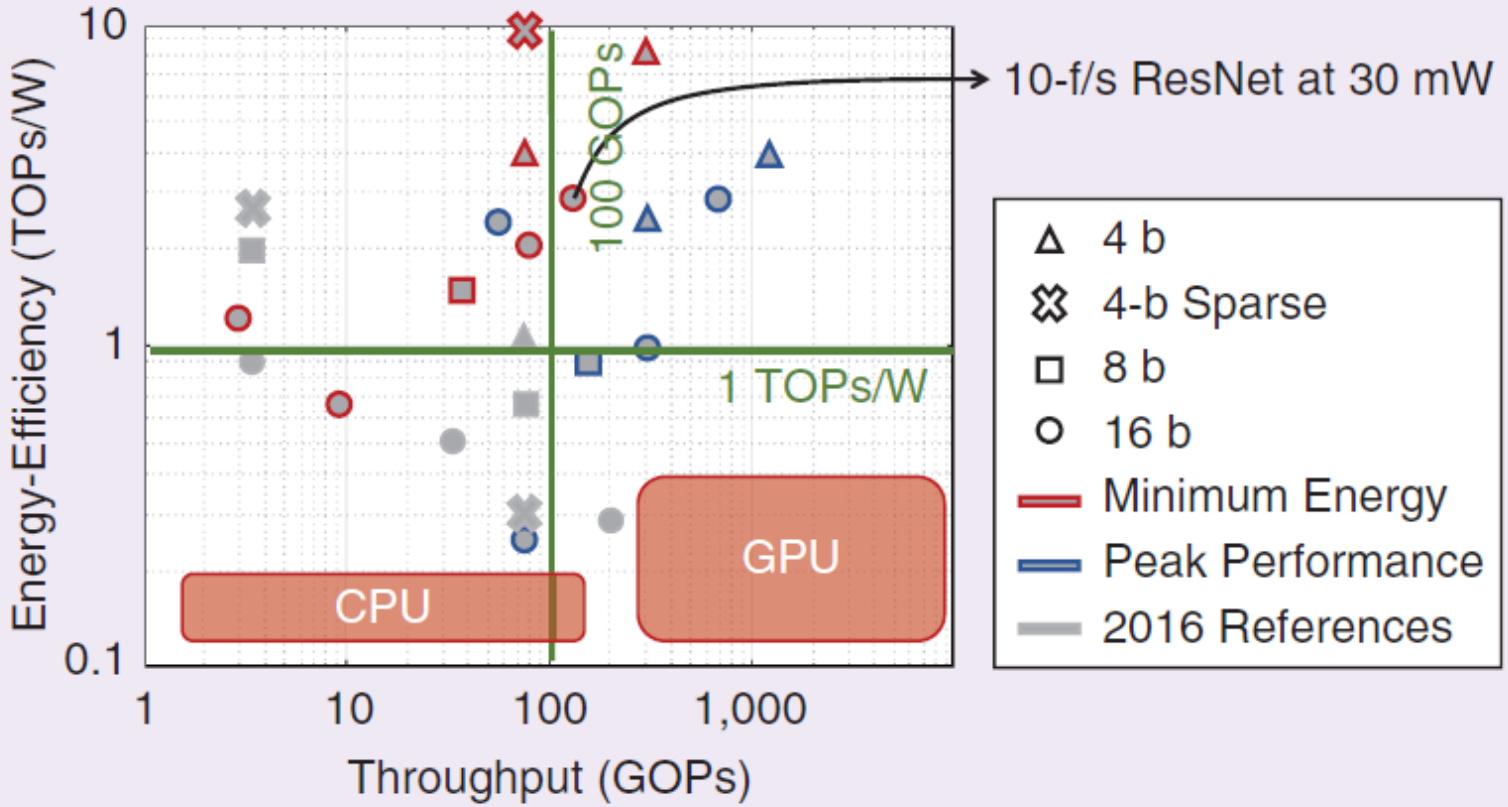
	Sequential	Precise Parallel	Speedup	Imprecise Parallel	Speedup
Galaxy S7	12331.82	436.71	28.24X	207.1	59.54X
Nexus 6P	17299.55	388.36	44.55X	129.21	133.89X
Nexus 5	43932.73	588.29	74.68X	141.38	310.74X

Energy Consumption

- Power measurements by Qualcomm Trepn profiler
- Airplane mode and minimized screen brightness.

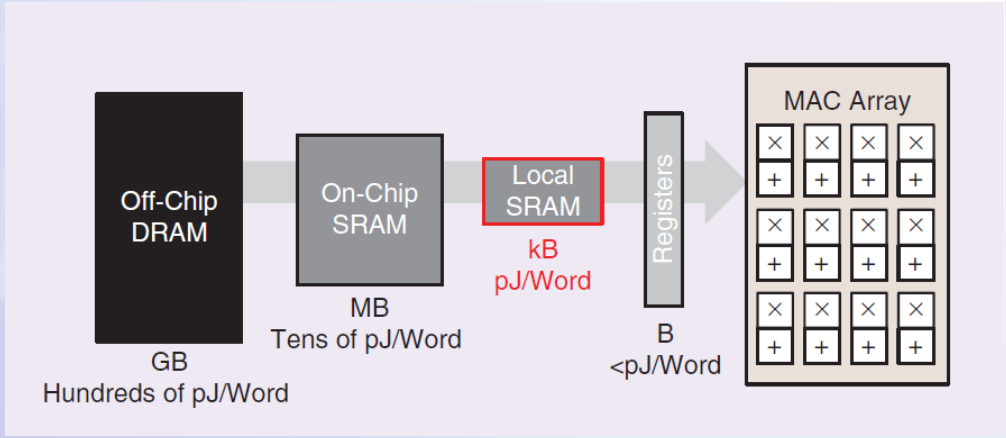
	Total Power (mW)			Energy/Image (J)		Energy Ratio
	Baseline	Sequential	Imprecise	Sequential	Imprecise	
Galaxy S7	173.18	1552.51	2921.79	17	0.569	29.88X
Nexus 6P	1480.97	1999.12	5461.89	8.96	0.514	17.43X
Nexus 5	422.71	1023.3	1170.45	26.37	0.106	249.47X

ASICs Reported in ISSCC 2016 & 2017



[Verhelst and Moons 2017]

Accelerator Optimization



	Solving the Memory Bottleneck	Solving Computational Bottleneck
Algorithmic Techniques	**A) Enhancing and Exploiting Network Structure**	
	• Spatial Data Reuse • Hierarchical Memory Exploiting Data Locality	• Highly Parallel Architectures • Distributed Processing
Tightly Linked	**B) Enhancing and Exploiting Fault Tolerance**	
	• Quantized Training • Stochastic Memories	• (Dynamic) Fixed Point • Analog and Statistical Processing
Processor Architecture Techniques	**C) Enhancing and Exploiting Network Sparsity**	
	• Network Pruning • Network Compression and Weight Sharing	• Memory and Computational Gating • Compressed Computing

[Verhelst and Moons 2017]