Boosting AI Efficiency through Accelerators, Custom Compilers and Approximate Computing

Swagath Venkataramani

IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

VLSID 2020

The evolution of AI: Past, Present and Future

Narrow AI

Single task, single domain

Superhuman accuracy and speed for certain tasks

Broad AI

Multi-task, multi-domain

Multi-modal

Distributed AI

Explainable

▼ We are here

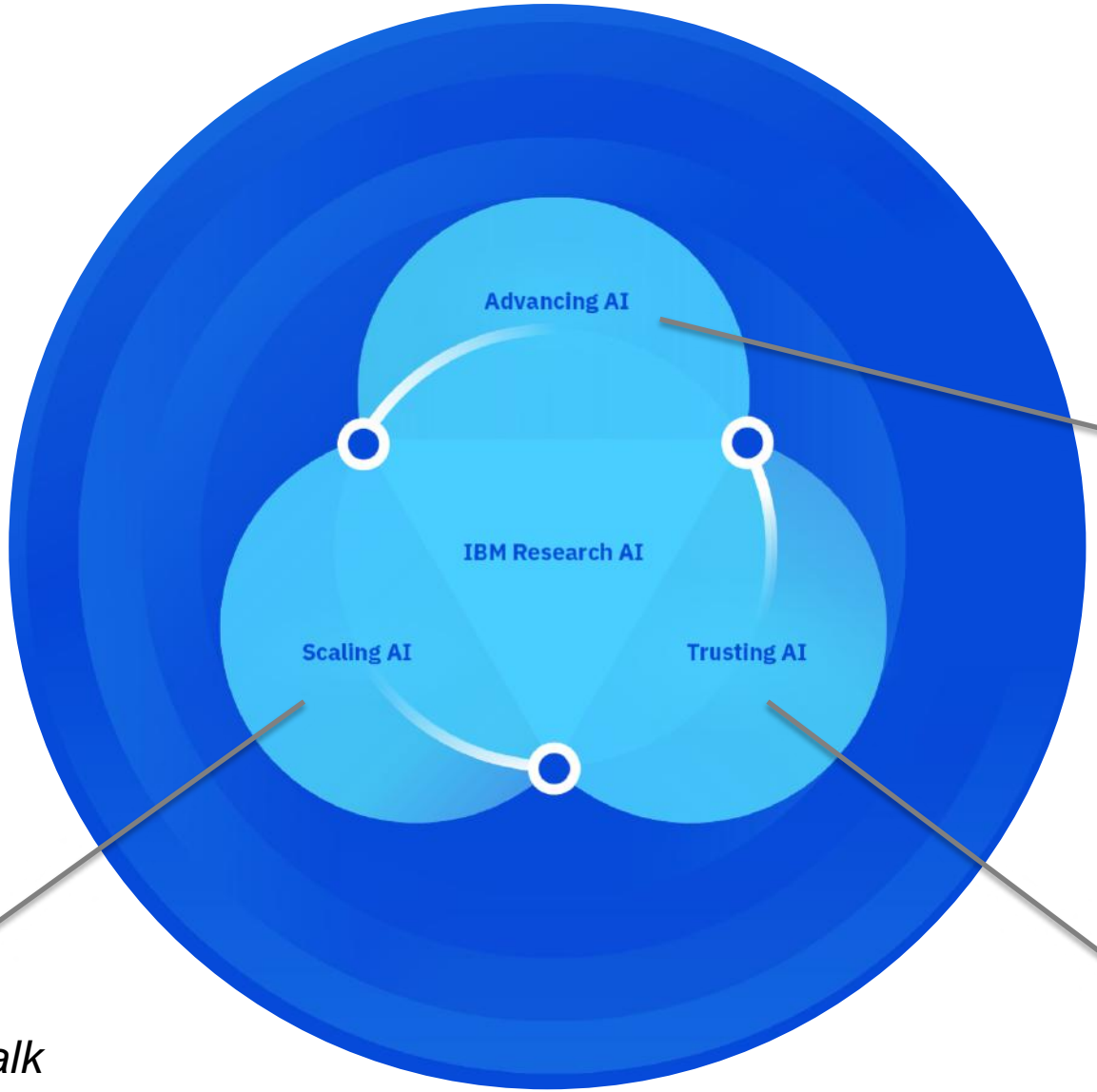
General AI

Cross-domain learning and reasoning

Broad autonomy

2050 and beyond

The path to Broad AI



Advancing AI
Powering advances in perception, reasoning and understanding to help AI address complex human-like tasks.

Scaling AI
Novel technologies across the full computing stack that make AI faster, easier, and able to scale to larger and more complex problems.

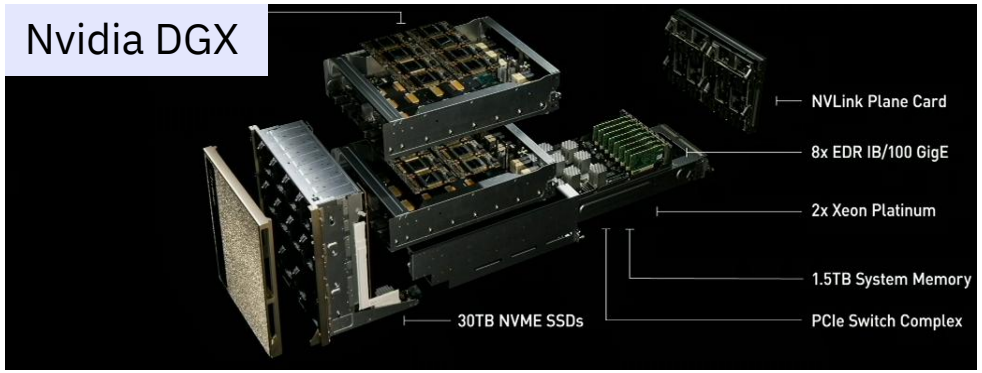
Focus of this talk

Trusting AI
Novel techniques to instrument key dimensions of trust and enable AI solutions that inspire confidence.

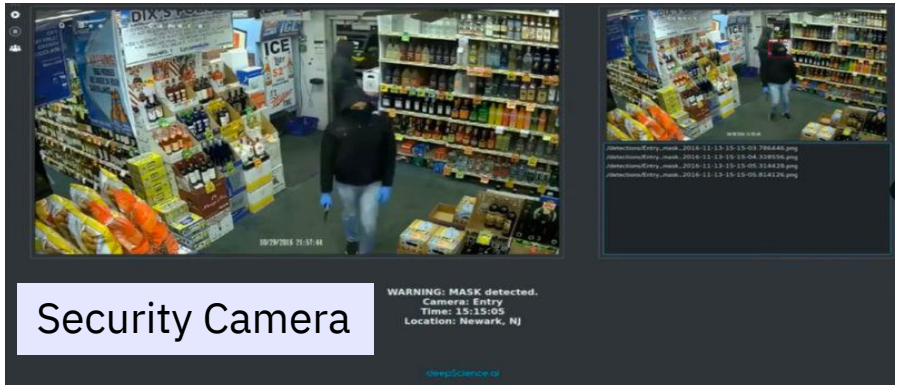
Deep Learning (DL) Training and Inference Use Cases

- **Model Training**
  - Typically on-prem
  - Customized GPU / ASIC Cluster of > 16 chips
  - Days-weeks to train DL models
- **Server Inference**
  - Most cases allow some form of batching (i.e. latency insensitive)
  - Currently CPU dominated – transitioning to PCIe attached accelerators
- **Transactional Inference**
  - Extremely latency sensitive – difficult to batch DL jobs.
  - Use cases : Financial industry, insurance,….
  - PCIe attached and on-CPU chip accelerators
- **Autonomous Driving (Latency sensitive)**
  - Largely focused around image, LIDAR / other sensor processing & fusion
  - Accuracy is extremely important
  - Huge # of Ops and extremely latency sensitive → customized SoCs
- **Inference on Mobile / IoT devices**
  - Security, Mobile, Home Appliances. Drones,….
  - Deep Learning based Object detection, Image Classification, Translation
  - Slightly lower accuracy tolerable

Nvidia DGX



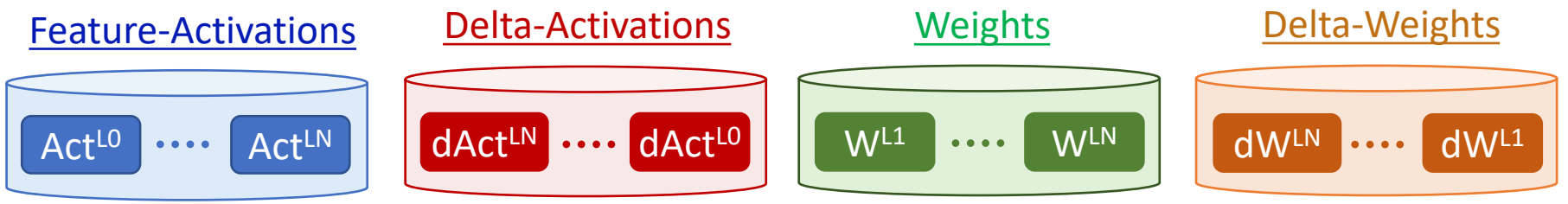
Self Driving Car

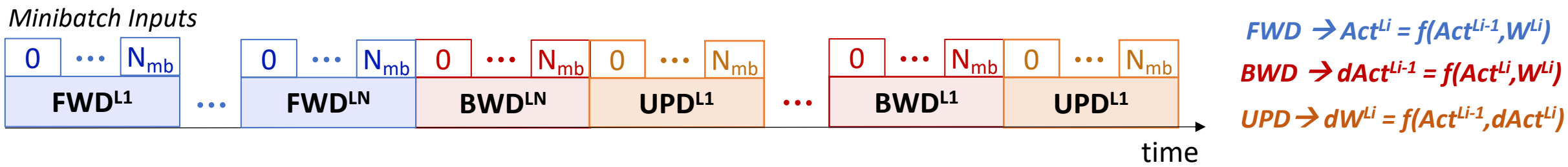


Security Camera

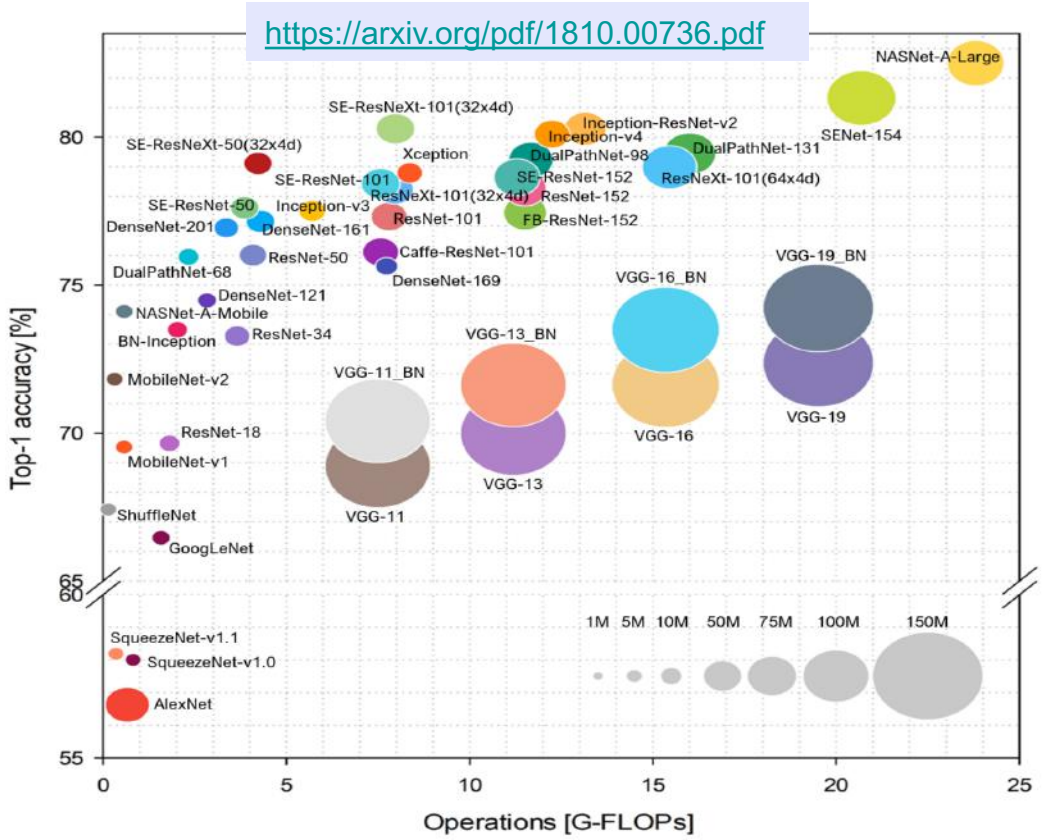
DNN Training/Inference: Execution Dataflow

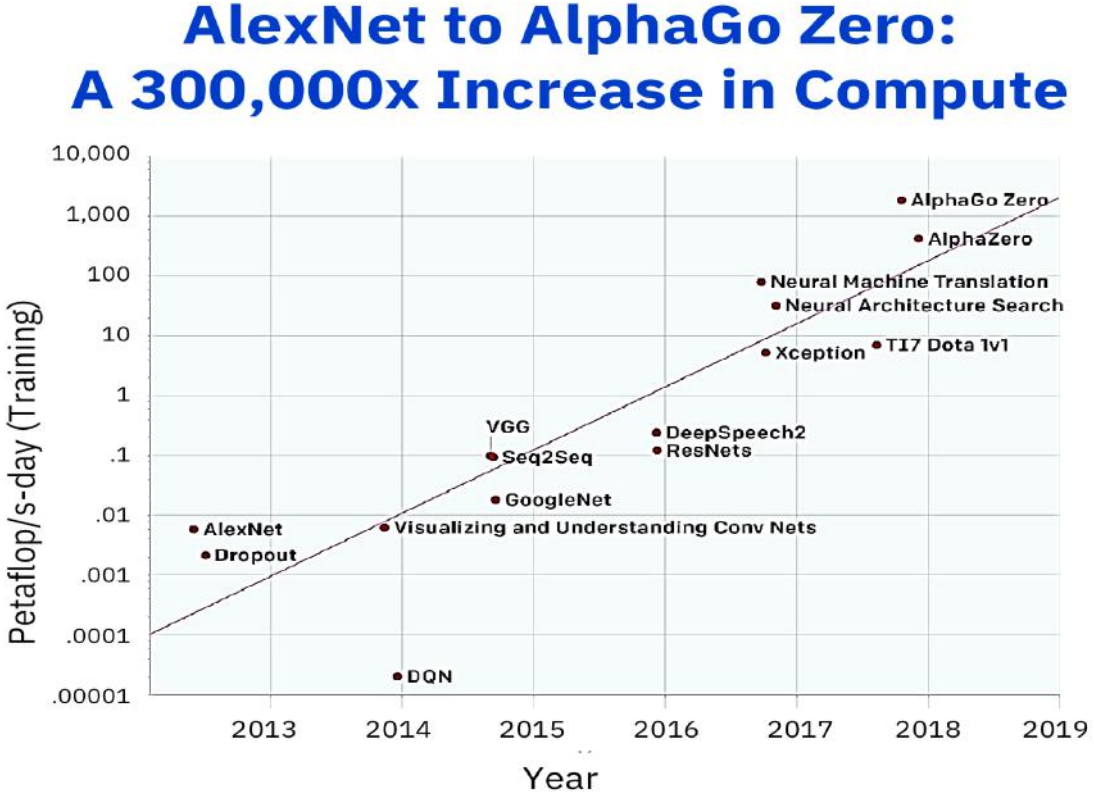
- **DNN training: Learn DNN weights by randomly initializing, and iteratively refining them based on labelled training inputs**
- **DNN training involves 3 key compute kernels**
  - Forward propagation (FWD): Evaluate neurons in each layer to compute DNN output
  - Backpropagation (BWD): Find error at output based on labelled data, and propagate it back through each layer
  - Weight Update (UPD): Use the error at output of each layer to update its weights
- **Minibatch: Training inputs are grouped into minibatches**
  - Weights are updated after gradients from all minibatch inputs are accumulated





Deep Learning Computational Complexity

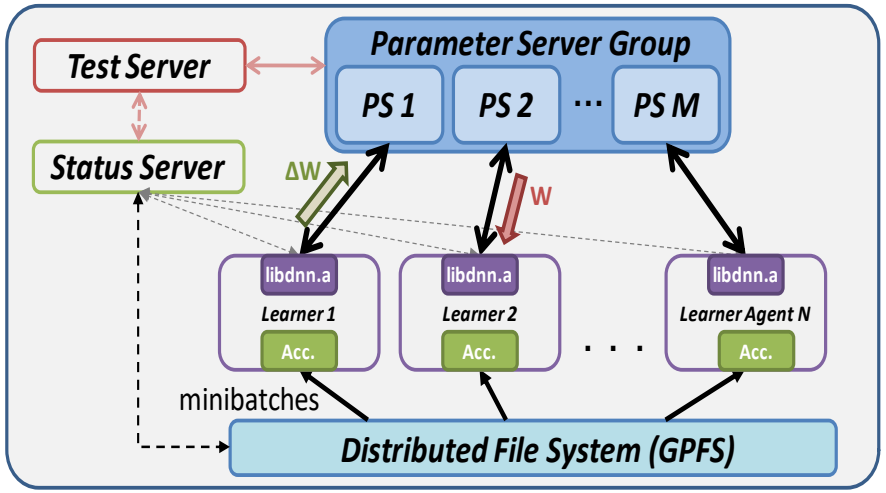




- **Deep Learning Training is computationally extremely expensive.**
  - Amount of compute needed for the largest Training job is **doubling every 4 months**
  - **10 - 100 ExaOps** of total compute for training – for typical DL models.
- **Deep Learning Inference needs (Op/s) also increasingly dramatically.**
  - Recent complex models recently for language modeling, object detection, image classification, speech

Deep Learning System Bottlenecks

Deep Learning Training



- **Large system training :**
  - Multiple **300W** GPUs/ASICs connected on a node (using **proprietary links**) + Multiple Nodes connected via Infiniband

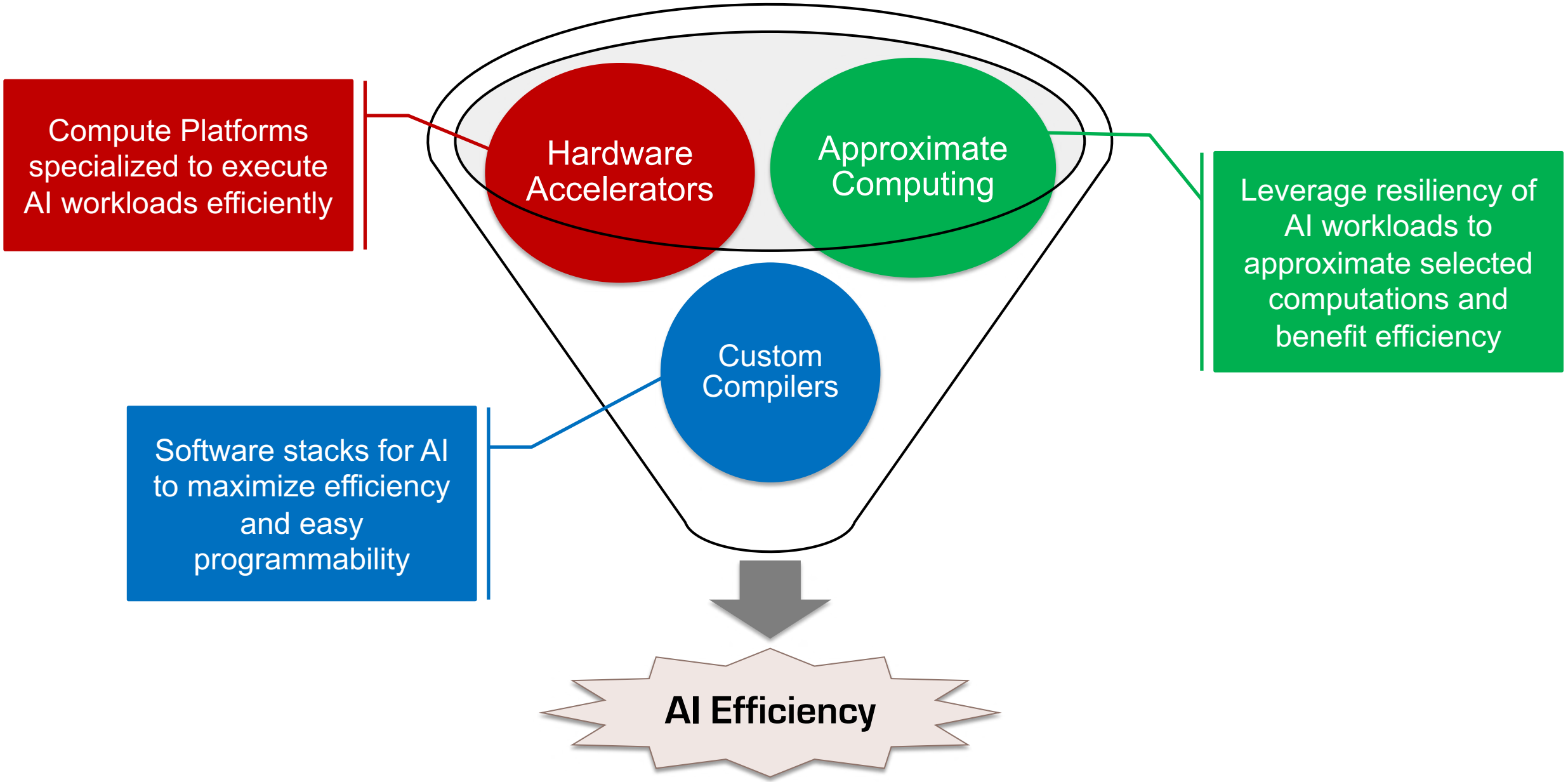
- **Training Bottlenecks:**
  - **Computation**: Minibatch SGD
    - Peak GPU / ASIC Flops capability
    - Utilization : Typical GPU utilization ~ 10 – 35% - depends on minibatch size (higher the better) – **limited by memory bandwidth**!
  - **Communication**: Different synchronization schemes possible
    - Overheads depend on superminibatch size and # of learners but limited by DL convergence - **limited by chip-to-chip bandwidth**.

Deep Learning Inference

- **Typically single-chip (board) solution**
  - **75W** PCIe attached
  - Virtualization (multi-thread /multi-process / multi-VM) support critical

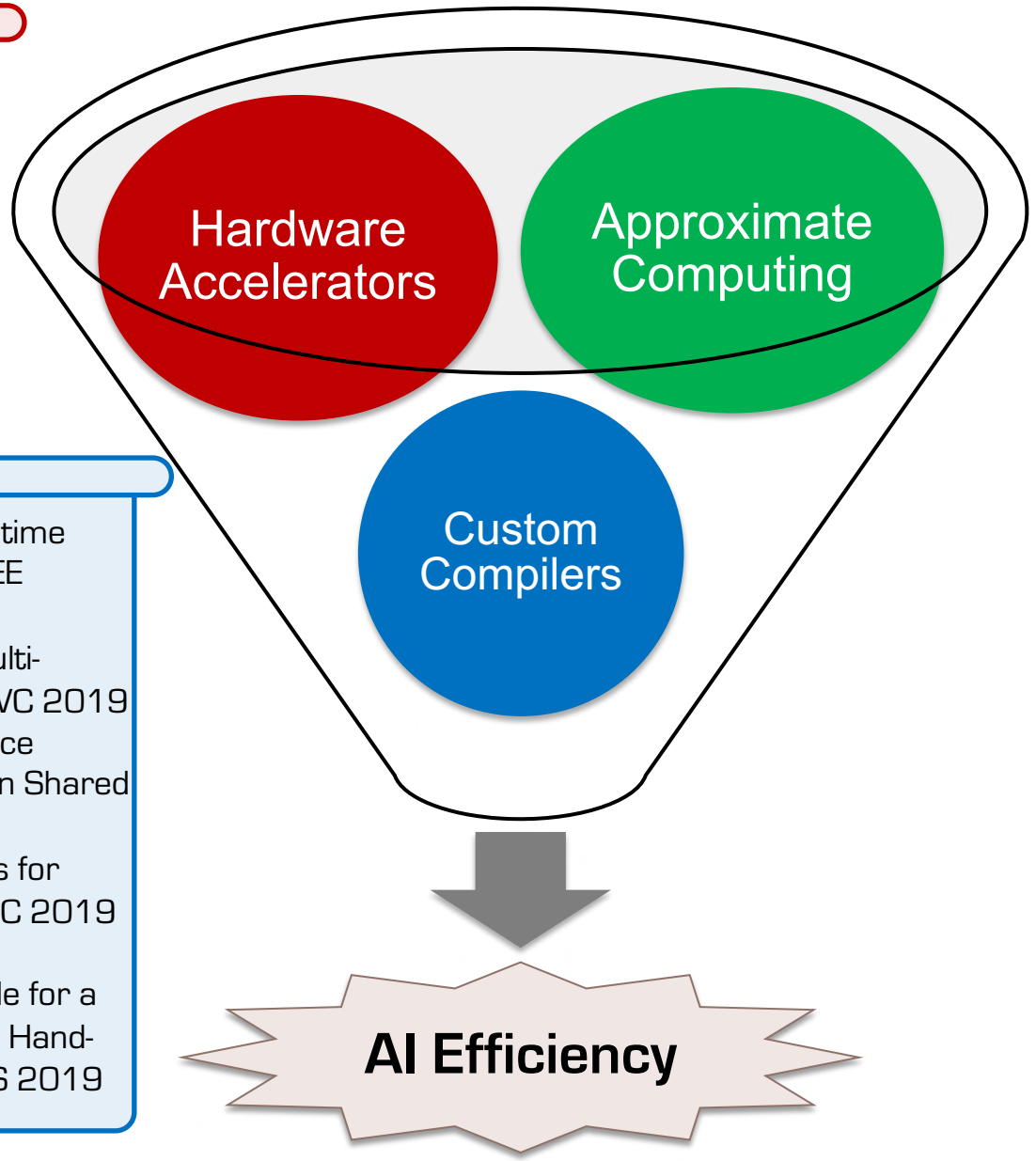
- **Inference Bottlenecks:**
  - **Computation** bottlenecks:
    - Peak GPU / ASIC Flops capability.
    - Utilization : Typically higher than training since fewer tensors – but still **limited by memory bandwidth**
  - No communication bottlenecks.
    - Multi-chip inference not common

Boosting Deep Learning Efficiency



Boosting Deep Learning Efficiency

- "A scalable multi-tera ops deep learning processor core for AI training and inference", VLSI 2018
- "DLFloat: A 16-b Floating Point format designed for Deep Learning Training and Inference", ARITH 2019



- "DeepTools: Compiler and Execution Runtime Extensions for RaPiD AI Accelerator", IEEE MICRO 2019
- Performance-driven Programming of Multi-TFLOP Deep Learning Accelerators, IISWC 2019
- Design Space Exploration for Performance Optimization of Deep Neural Networks on Shared Memory Accelerators, PACT 2017
- "Memory and Interconnect Optimizations for Peta-Scale Deep Learning Systems", HIPC 2019
- A Compiler for Deep Neural Network Accelerators to Generate Optimized Code for a Wide Range of Data Parameters from a Hand-crafted Computation Kernel, COOLCHIPS 2019

Training:

- Deep learning with limited numerical precision, ICML 2015
- Adacomp: Adaptive residual gradient compression for data-parallel distributed training, AAAI 2018
- Training deep neural networks with 8-bit floating point numbers, NeurIPS 2018
- Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks, NeurIPS 2019

Inference:

- Pact: Parameterized clipping activation for quantized neural networks, arXiv 2018
- Bridging the accuracy gap for 2-bit quantized neural networks (QNN), SysML 2018
- Compensated-DNN: energy efficient low-precision deep neural networks by compensating quantization errors, DAC 2018
- BiScaled-DNN: Quantizing Long-tailed Data structures with Two Scale Factors for Deep Neural Networks, DAC 2019

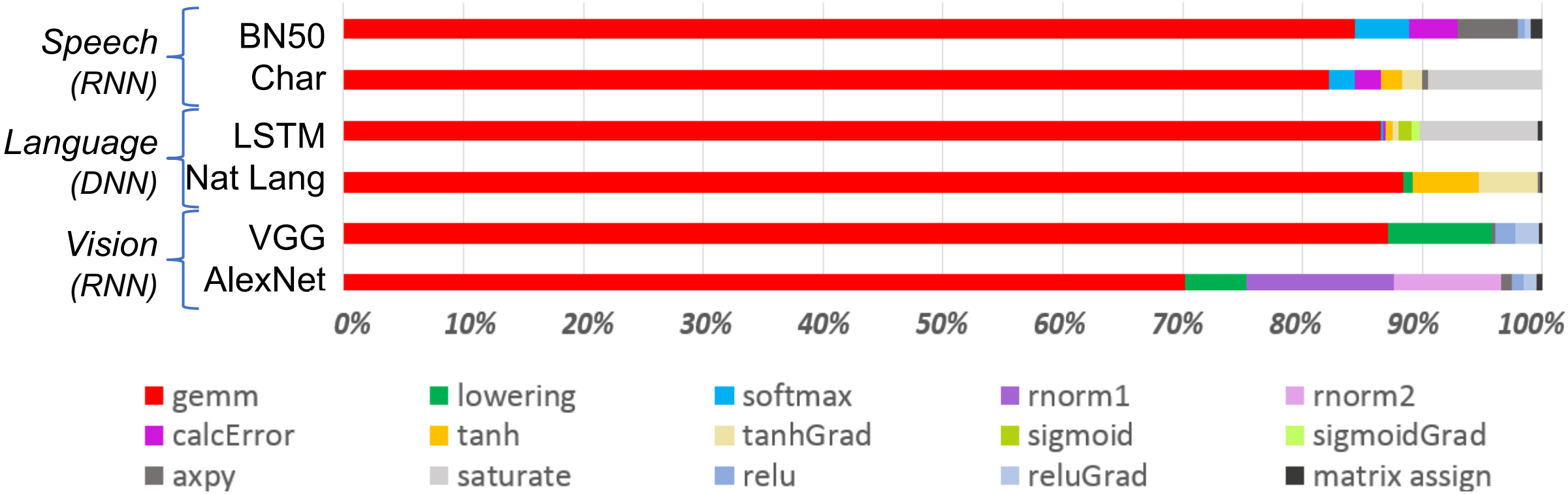
Hardware Accelerators

Deep Learning (DL) Accelerator: Hardware Design Principles

1. **End-to-end performance**
   - Parallel computation, high utilization, high data bandwidth
   - Support minibatch sizes down to as low (1 if possible)
     - Useful for transactional inference and extremely scaled training use cases.
2. **DL model accuracy**
   - DL operations require a mix of various precisions (fp32, fp16, .. INT2)
   - Design optimized for mixed-precision processing engines
     - Support for higher precision reduces efficiency of low-precision hardware
3. **Power efficiency**
   - It's an accelerator: application power should be dominated by compute elements
4. **Flexibility and programmability**
   - Support dataflow diversity and development of future algorithms
   - Architecturally maximize on-chip reuse: Access to data is as important as compute
5. **Scalability**
   - Single-core vs. Multi-core approach
   - Effective core-to-core and chip-to-chip communication

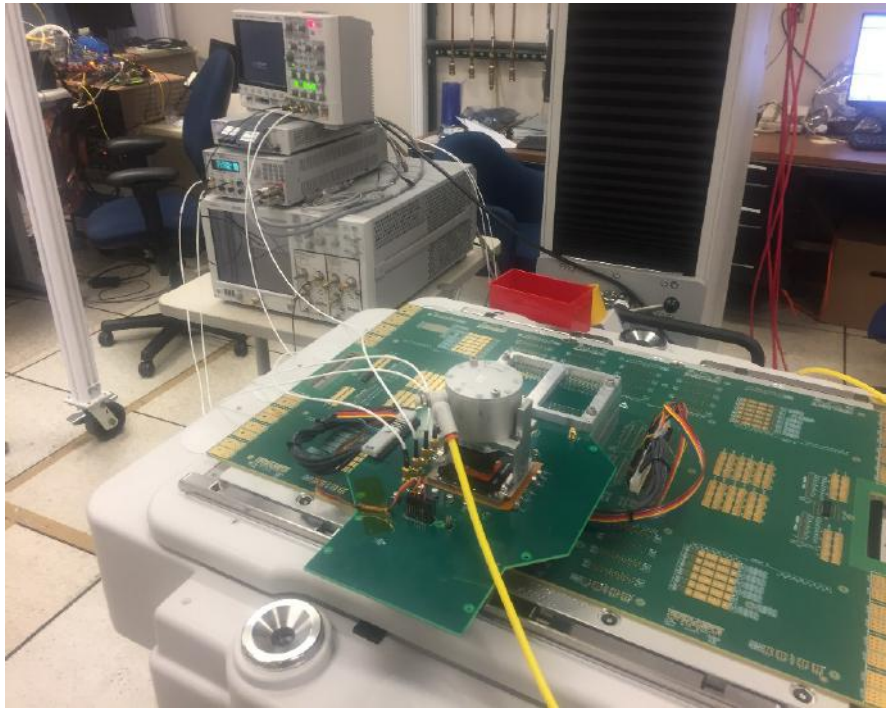
Workload Profiling

- **Op count is dominated by matrix operations and a small set of other functions**
  - Convolution/Matrix multiplication
  - Vector operations: Point-wise functions with/without reduction
- **All functions are highly parallelizable**



RaPiD: 14nm 1.5 GHz DL accelerator core

- **Many highly tuned fp pipelines and high bandwidth throughout [Performance]**
  - Customized dataflow architectures
  - Algorithm/program/ISA/hardware co-designed specifically for DL workloads
- **Balanced multiple-precision support [Accuracy]**
  - Precision chosen for each computation, for training and inference
- **Simplify logic in and around compute pipelines [Power]**
  - Carefully curated ISAs
  - Streamlined control logic
- **ISA-accessible communications network [Programmability/Scalability]**

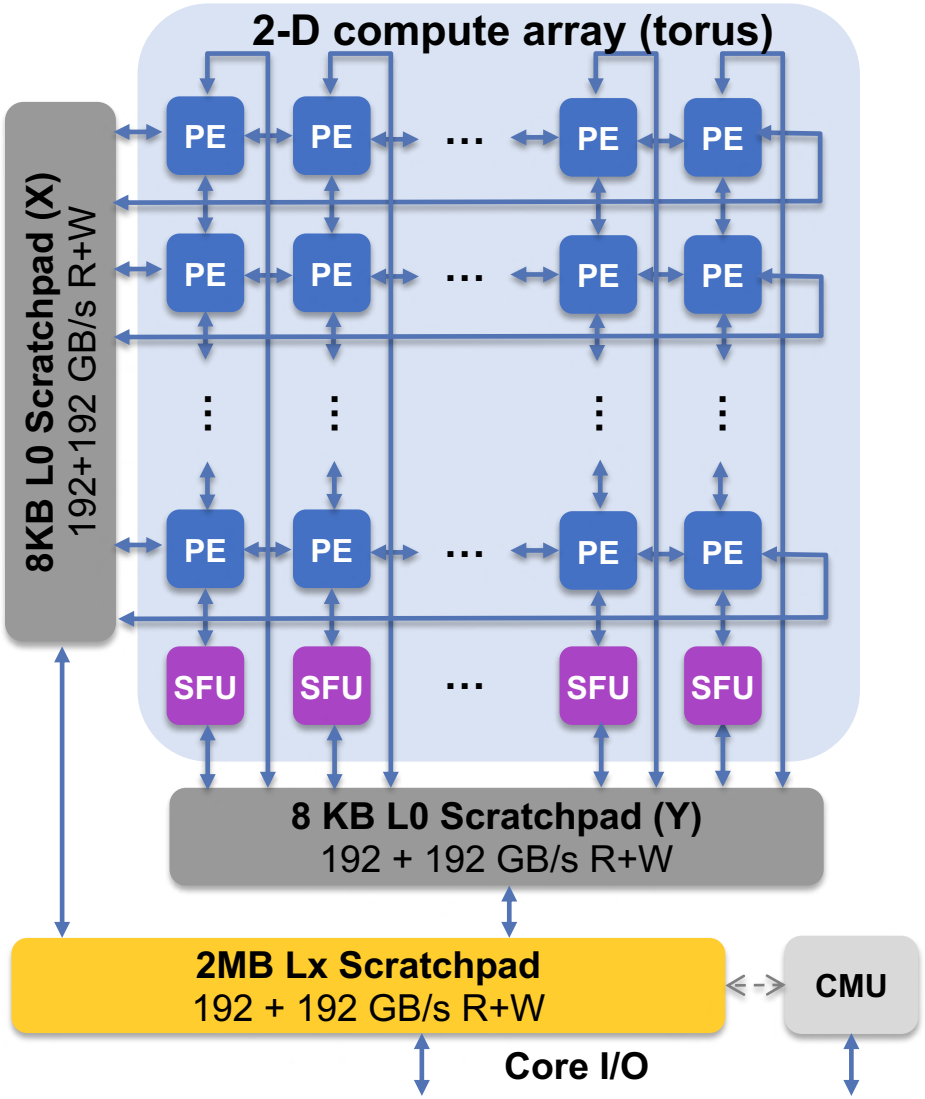


- Peak performance of 1.5 TFLOPS fp16, 12 TOPS ternary and 25 TOPS binary
- Sustained utilization >90% on multiple neural-network topologies
- Core in+out bandwidth of 96+96 GB/s for scalability

A scalable multi-teraops deep learning processor core for AI training and inference, Fleischer et al, VLSI 2018.

13

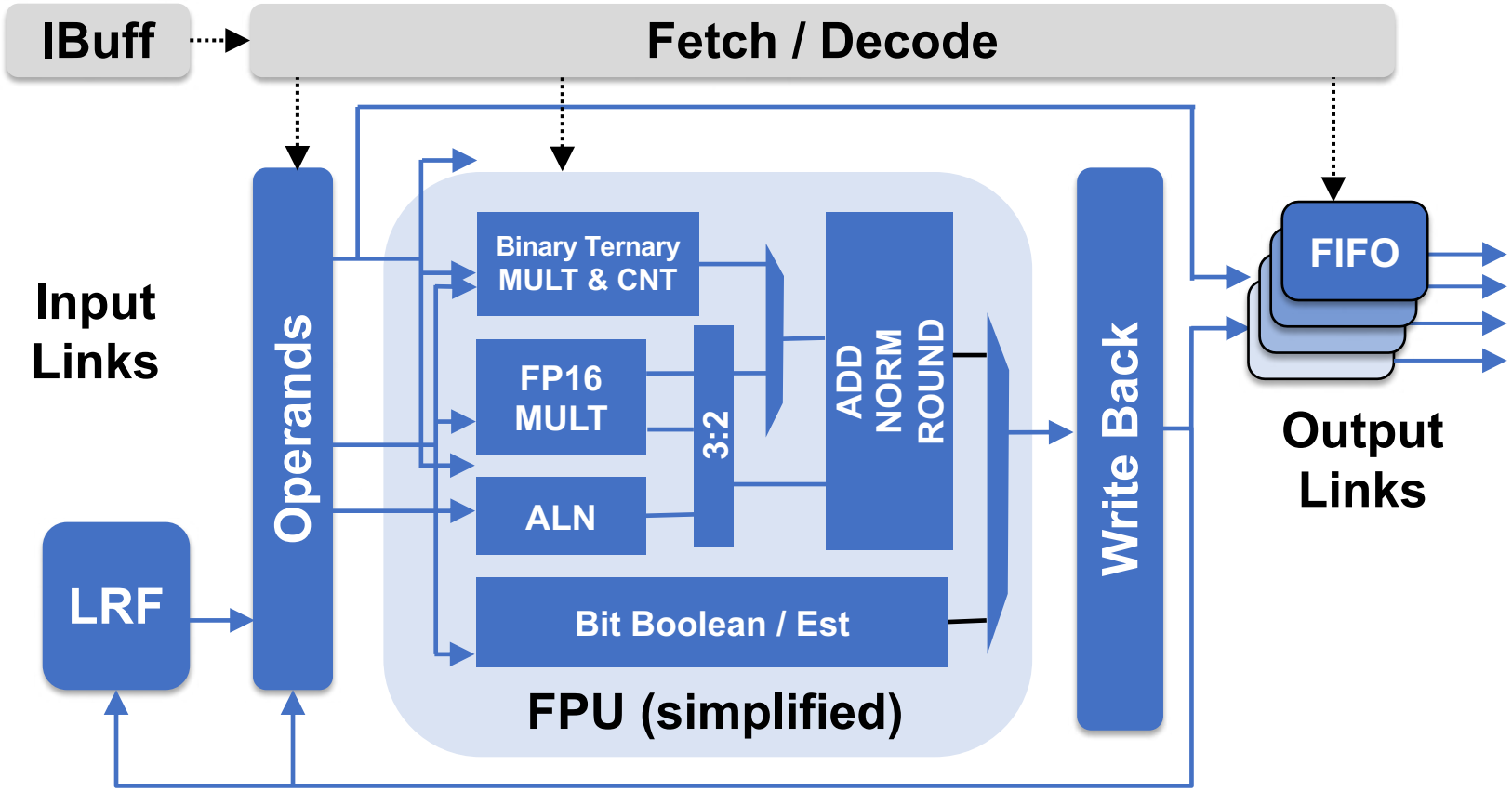
RaPiD Core Microarchitecture



- **Dataflow with scratchpad hierarchy**
- Customized dataflow architecture – **hybrid SIMD-Dataflow** vs. traditional Dataflow
- **Reduced-precision** based PEs (Processing Elements) for ***matrix / convolution ops***
  - Support for precisions down to 2-bits (FP16 / FP8 / INT4 / INT2)
  - Minimum accumulation precision (HW chunking techniques for ALUs)
- **Limited FP32 SFUs (Special Function Units)** for ***vector (linear/non-linear) ops***
  - Needed primarily for softmax, batchnorm and axpy for training
- **Directly-addressable multi-level scratchpads** (Software managed) for high utilization (core efficiency)
- Double-buffering in Lx and high bandwidth between Lx-L0-compute

A scalable multi-teraops deep learning processor core for AI training and inference, Fleischer et al, VLSI 2018.

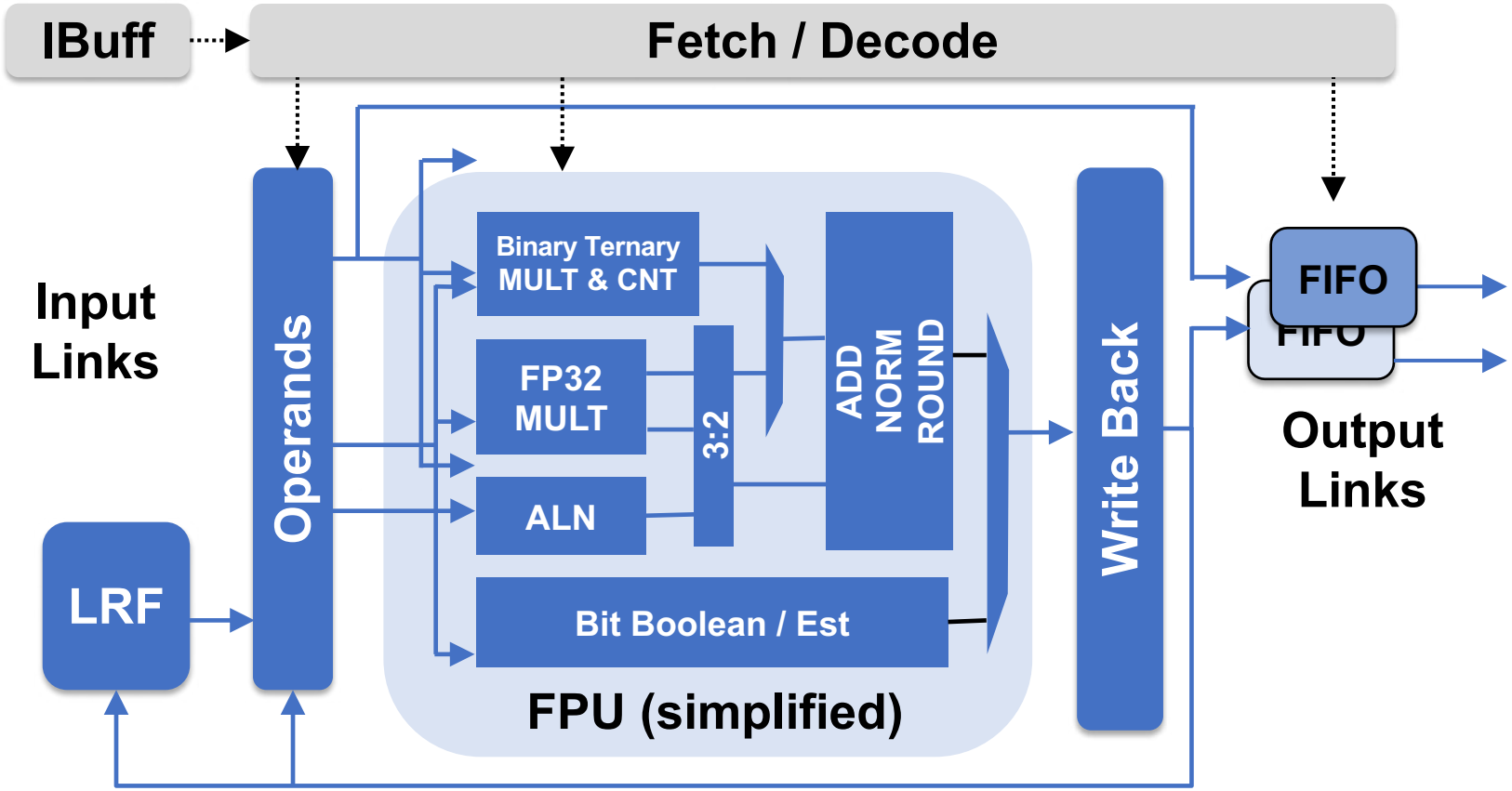
FP16 Processing Element (PE)



- Very simple I-unit with a small local program
- FP16 multiply-add, compare, min/max
- Hardware approximate recip/sqrt/exp/ln
- FP16 exponent modify
- Binary/Ternary math and Boolean

A scalable multi-teraops deep learning processor core for AI training and inference, Fleischer et al, VLSI 2018.

FP32 Special Function Unit (SFU)

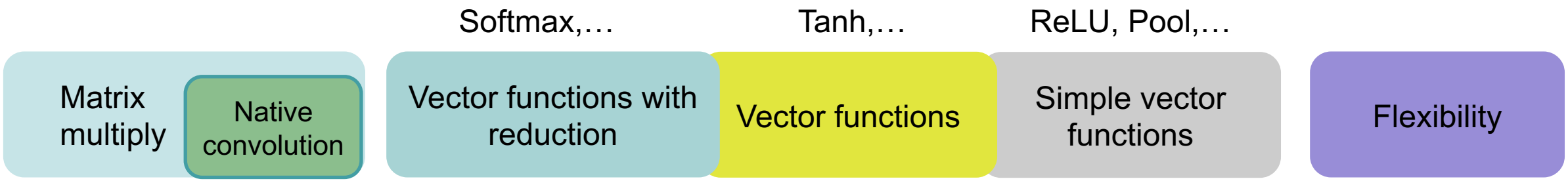


- Very simple I-unit with a small local program
- FP32 multiply-add, compare, min/max
- Hardware approximate recip/sqrt/exp/ln
- FP32 exponent modify
- Binary/Ternary math and Boolean
- FP16↔FP32 conversions

A scalable multi-teraops deep learning processor core for AI training and inference, Fleischer et al, VLSI 2018.

ISA features

Features to support algorithm variations and future development



FP16 dataflow

FMA operands from data flowing H/V and in local regs

Special-purpose data movement ops

Native support for reduction

Estimate ops for FP16 and FP32

Independent matrix and vector programs

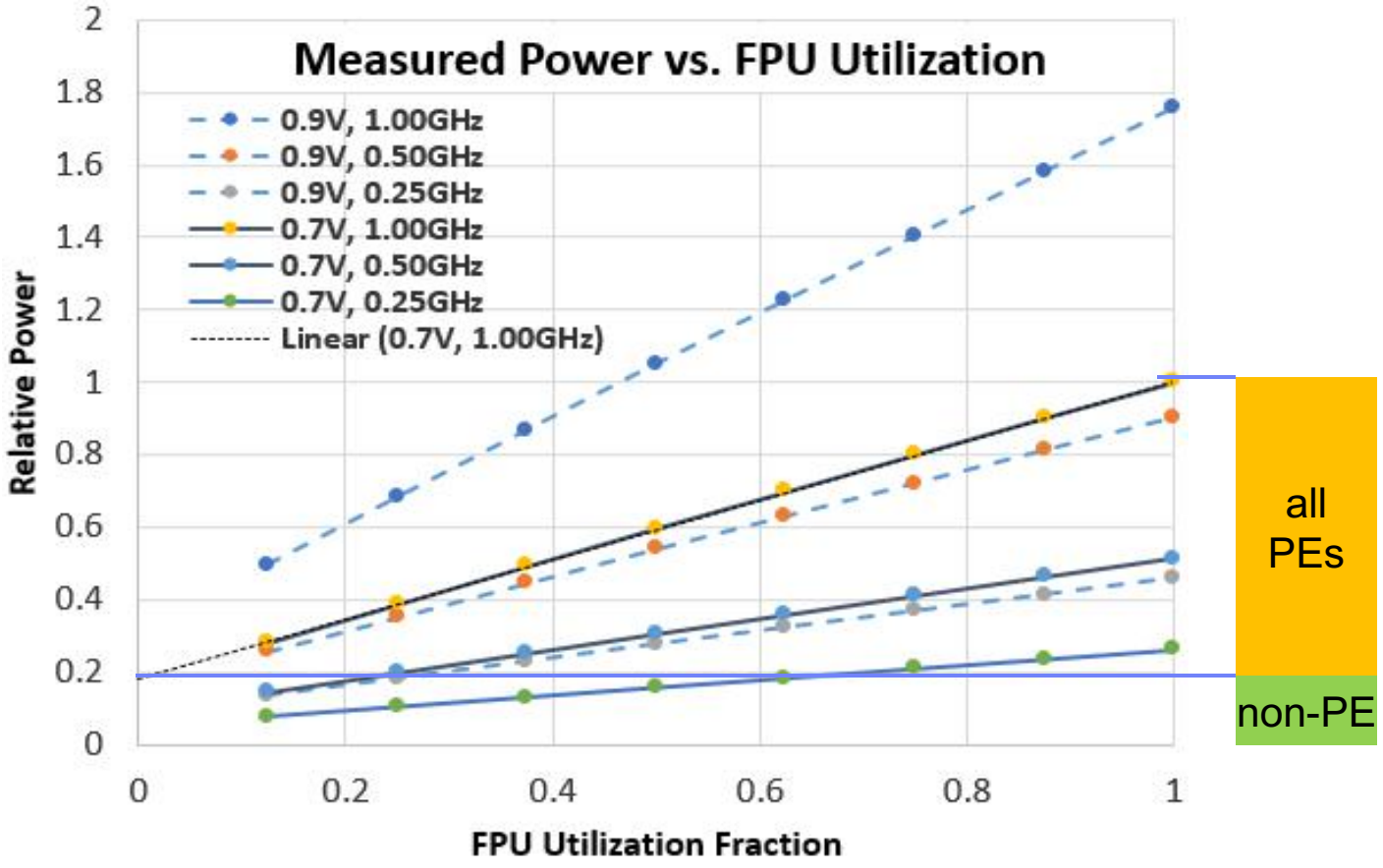
Single-instruction ReLu, etc.

Implement one algorithm with different precisions

Dataflow mapping by ISA

A scalable multi-teraops deep learning processor core for AI training and inference, Fleischer et al, VLSI 2018.

Power is dominated by PEs



Special suite of benchmark programs

- Each program uses a different fraction of the PE array
- The active PEs run at near 100% utilization
- Lx and LO supply data in every cycle in all programs

- Extrapolation to 0 active PEs gives power outside PE array
- P is linear in the utilization
- PEs use > 80% of power
- P ∝ frequency (<5% leakage)

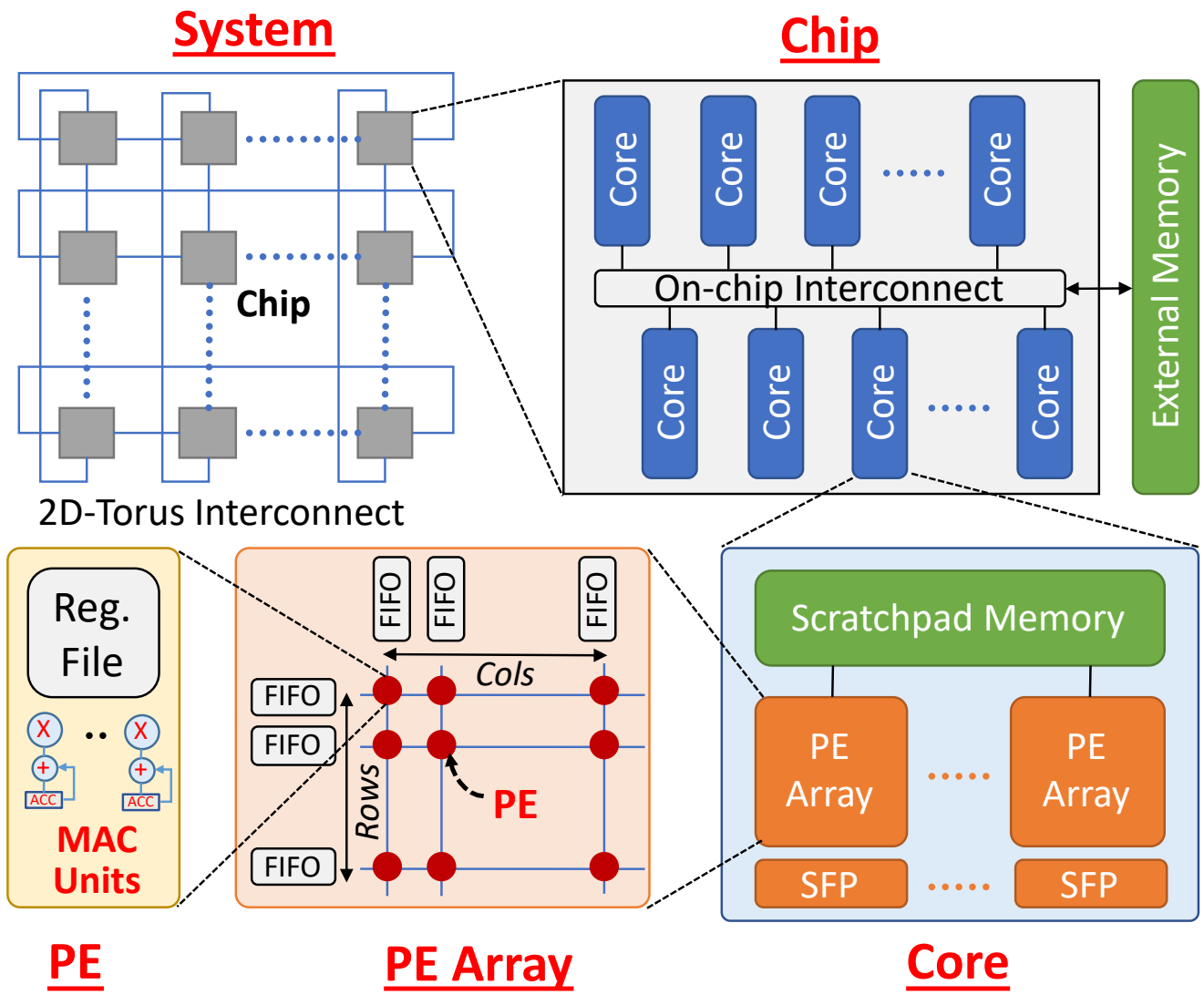
A scalable multi-teraops deep learning processor core for AI training and inference, Fleischer et al, VLSI 2018.

18

Peta-Scale Deep Learning System

- **Customized server class system for training deep learning models**

- **5 tiered hierarchy**
  - **PE**: SIMD multiply-and-accumulate engines
  - **PE array**: 2D array of Pes
  - **Core**: Multiple PE arrays with shared memory and special
  - **Chip**: Ring of cores
  - **System**: 2D torus of chips



Other Architectural Techniques: Sparsity

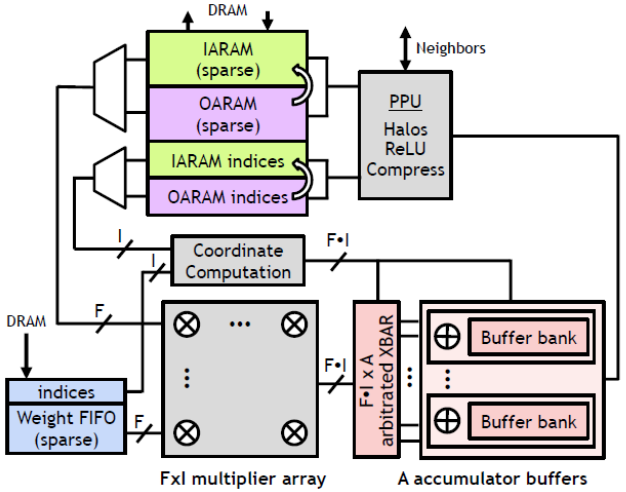
- **DNN data-structures exhibit significant amount of sparsity**
  - Static vs. Dynamic sparsity
- **Multiply-and-Accumulate operation becomes NOP when one of its operands is zero**
- **Opportunities:**
  - Identify and skip sparse computations in hardware
  - Store and fetch sparse data-structures in compressed format

50.6%	65.6%	85.1%
Activations	Weights	Errors

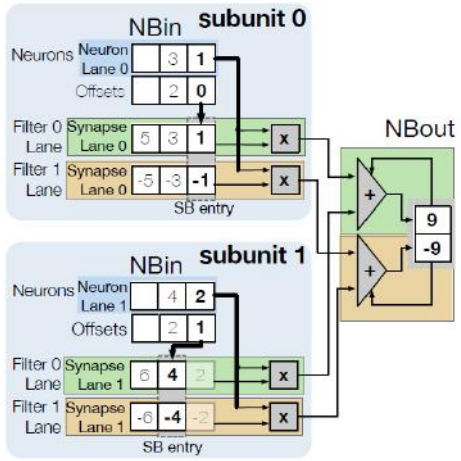
Static sparsity
- Location & no. of zeros constant across inputs
- Network pruning

Dynamic sparsity
- Location of zeros vary across inputs
- ReLU activation functions

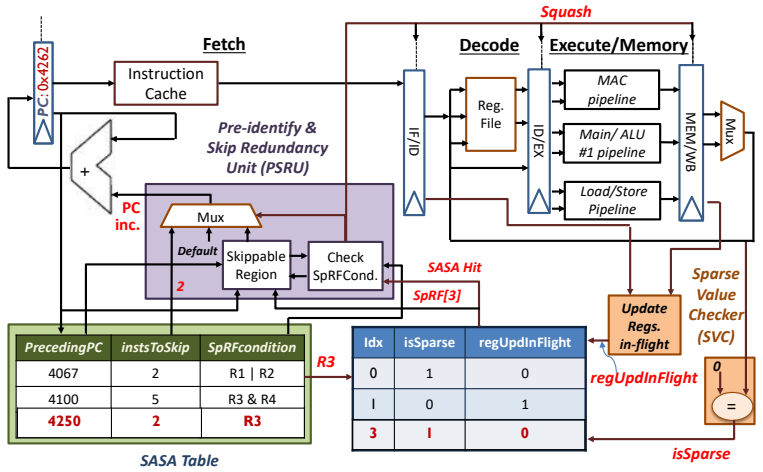
SCNN (ISCA 2017)



CNVLUTIN (ISCA 2016)

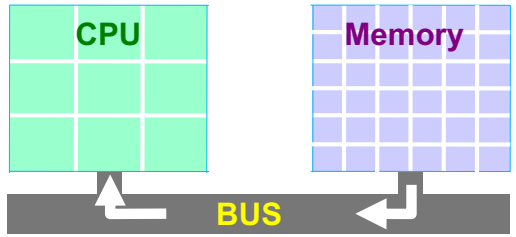


SparCE (TC 2018)

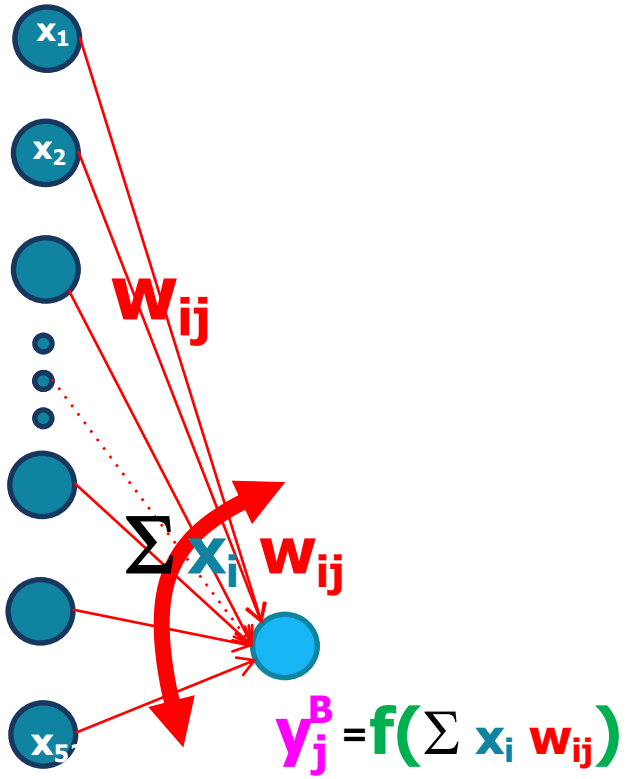


Other Architectural Techniques: In-memory Computing

- **Compute and memory are separate in traditional computer architectures**
  - Ability to supply data becomes performance/energy bottleneck
- **In-memory compute: Embed compute within memory**
- **Crossbar architecture with NVM: MRAM (Magnetic RAM), PCM (Phase-Change Memory), RRAM (Resistance RAM)**
  - Computations inherently approximate → methods to recuperate accuracy

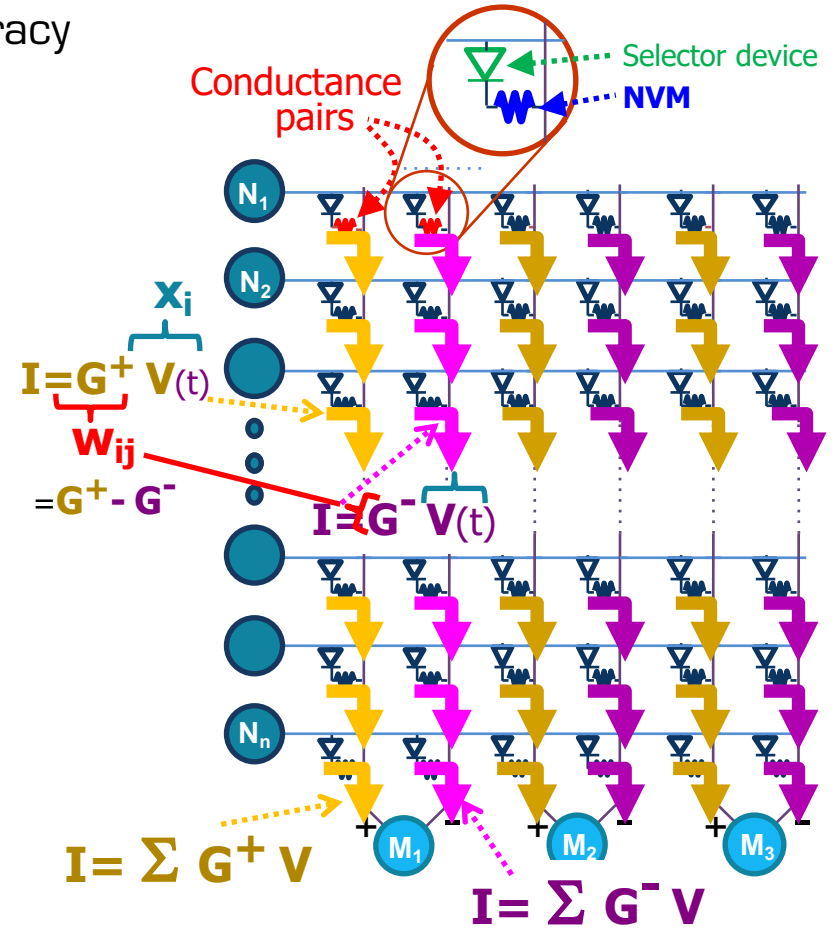


Von Neumann **"Bottleneck"**



1) Different peripheral circuitry
2) Weights **w** → conductances G^+, G^-
   (Ohm's Law: V= IR → I = GV)
3) Apply "**x**" voltages to **every** row
   (Kirchhoff's Current Law → Σ I)
4) Analog measurement

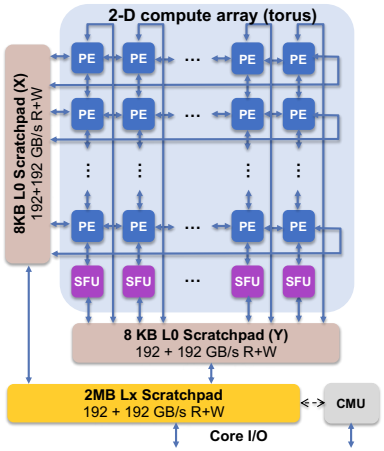
Computed at the data, by physics...



Custom Compilers

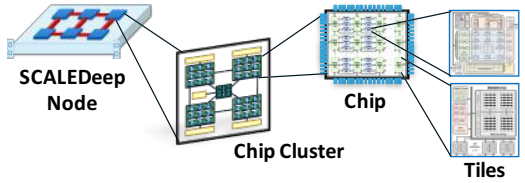
Programmable Deep Learning Accelerator Systems

Low-power IP cores ⟷ **Server-class Systems**



MAPLE [PACT10]
NeuFlow [CVPRW11]
DianNao [ASPLOS14]
Eyeriss [ISSCC16]
DANA [PACT15]
Minerva [ISCA16]
Multi-TOPS Core [VLSI18]
.....

DaDianNao[MICRO14]
NeuroCube[ISCA16]
TPU [ISCA17]
ScaleDeep [ISCA17]
.....



- **Each architecture represents a different point in energy *vs.* throughput trade-off**
  - Demonstrate impressive peak performance and processing efficiency
  - *Programmable* → Flexibility to execute DNNs of various shapes and sizes

Challenge:

- How do we program accelerators to achieve best possible ***system utilization*** for any given DNN?
- How do we achieve performance ***without sacrificing end-user (non-expert users) productivity***?

Programming Deep Learning Accelerators

- **DNNs are static dataflow graphs**
  - No data dependent execution paths, irregular memory access patterns etc.
  - Possible to define a space of mapping configurations and identify the best configuration *offline*
  - Performance estimation to reasonable accuracy through *analytical* analysis

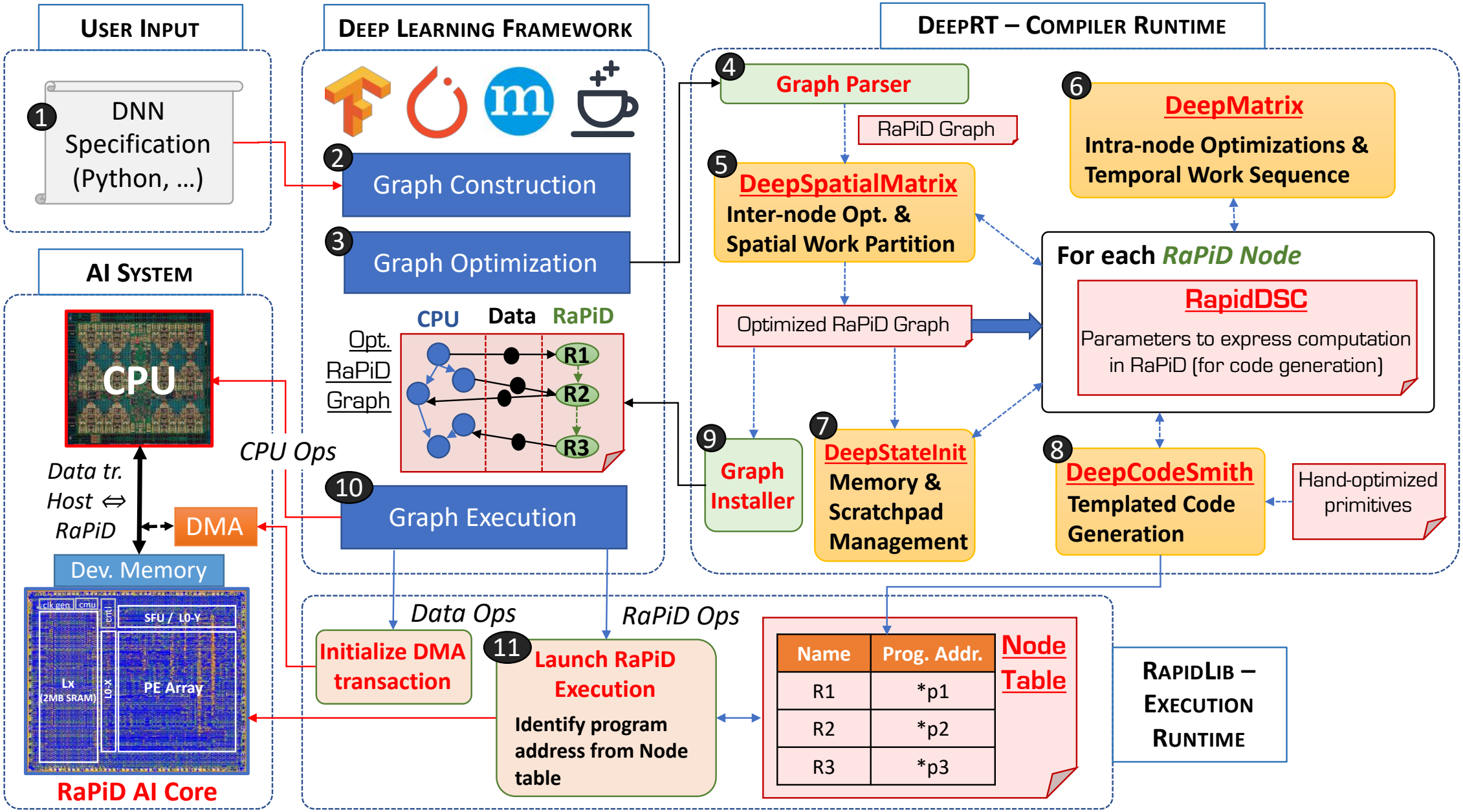
- **DNN functionality can be expressed using small set (tens) of primitives**
  - For example, VGG11 network contains **>10 billion** scalar ops, but expressible with **6** functions: *Convolution, Matrix-multiplication, ReLU, Max pooling, Softmax* & *Bias-add*
  - Complexity of how each function is optimally realized is hidden behind library/API calls

- **But, significant heterogeneity in shape and size of each data-structure, which makes each operation computationally unique**

Layer	CONV1	CONV2	…	CONV3_2	CONV4_1	…	CONV5_2	…	FC_7
Feature size	64x 224x224	128x 112x112		256x 56x56	512x 28x28		512x 14x14		4096x 1x1
Ops/By	25.78	372.58		842.51	519.06		180.63		1.00

  - **Insight**: Each layer/op needs to be *programmed differently*

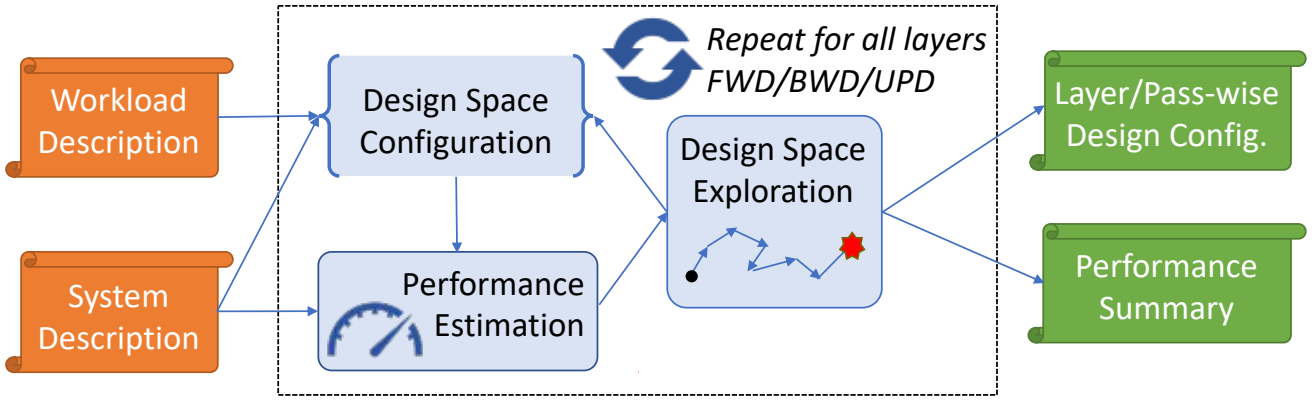
DeepTools: Software Stack for AI



DeepSpatialMatrix (DSM) +DeepMatrix (DM) : Mapping DNNs on Accelerators

- **DSM+DM: A systematic method to map DNNs on to any given accelerator system**
  - How the **compute needs to be partitioned** to across processing elements?
  - How much **data to stage** in each memory, respecting capacity constraints?
  - How the **data movement** needs to be orchestrated, given bandwidth limitations?

- **DeepMatrix can be leveraged by:**
  - System Designers: Understand limitations of architecture & explore micro-architecture space
  - Library Designers: Understand how computation/data needs to be sliced spatio-temporally
  - Users: A front-end compiler flow to map DNN applications



DeepMatrix + DeepSpatialMatrix: Steps

Design Space Characterization

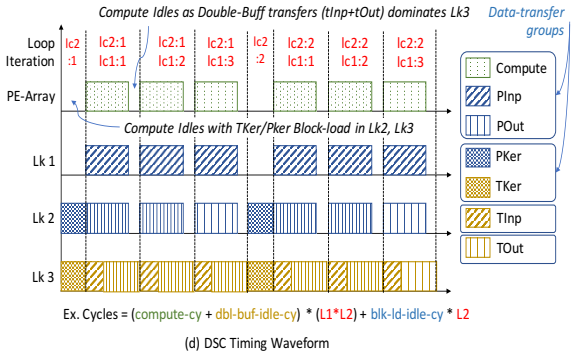
- **A parameterized design space that captures the various ways in which DNN layers can be mapped**
  - Dataflow, amount of data-staged, computation sequence, data-transfer locations

Design Space Configuration

DF - dataflow type (1 param),
(T/P)-(B/T)-(D/B){in,op,ij,kij,mb} loop orders (15 params),
P-T-B-X{Inp,Out,Ker} data-trans. locations (12 params),
P-T-B-D{in,op,ij,kij,mb} dimensions (20 params)

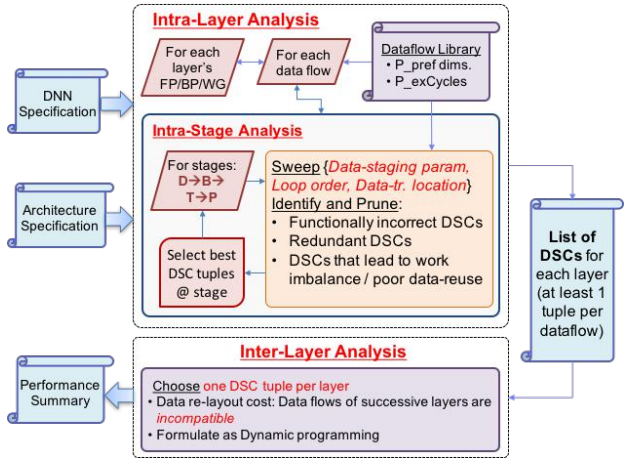
Configuration Estimation

- **A wave-form driven approach to estimate performance**
  - Explicitly accounts for each compute iteration, overlapped and visible data-transfers



Design Space Exploration

- **Methodology to search the configuration space**
  - Identify configurations that balance work and maximize reuse
  - Prune configurations that are redundant



Design Space Characterization (RapidDsc)

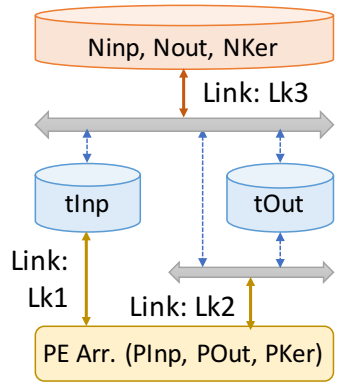
- **Spatial Work Division: Defines the work division across cores and chips**
  - Also defines how data is organized in memory and scratchpad of different cores
- **Dataflow: Orchestrate compute within PE array**
  - Data-structure/workload dimension mapped along rows, columns and held stationery in register file
  - Constrains for valid dataflows
- **Given the dataflow, the overall computation can be expressed as a nested sequence of loops**
- **Tile Sizes: Defines limits for how data-structures are chunked across memory hierarchy levels**
- **Loop Order: Determines the order in which each data-structure is traversed**
- **Data-transfer Location: Capacity and reuse of each data-structure at each level of memory**

[~100 Parameters]

attr N = <*dim tuple*>	*Total workload*
attr ChipD = <*dim tuple*> ***attr*** CoreD = <*dim tuple*>	*Spatial work division across chips and cores*
attr P = <*dim tuple*>	*Work fed to PE array*
attr B = <*dim tuple*> ***attr*** T = <*dim tuple*>	*Data-staging params for 2-level data tiling*
attr loopOrder = <*list* {stage,dim}>	*Seq. of nested loops: #loop stages * #dims*
attr dataStructures = <*list* { ***attr*** layout = <*dim tuple*> ***attr*** dataTransfers = <*list* {*src,dst,type,loopLoc*}> }>	*Info ∀ datastructure: Memory layout, data-transfers and their source, dst., and location within loops*
attr compPrimitives = <*list*>	*Primitive PE/SFP ops.*

Configuration Estimation

- **A waveform based approach to compute execution cycles for a give design space configuration**
  - Explicitly accounts for each compute iteration, overlapped and visible data-transfers

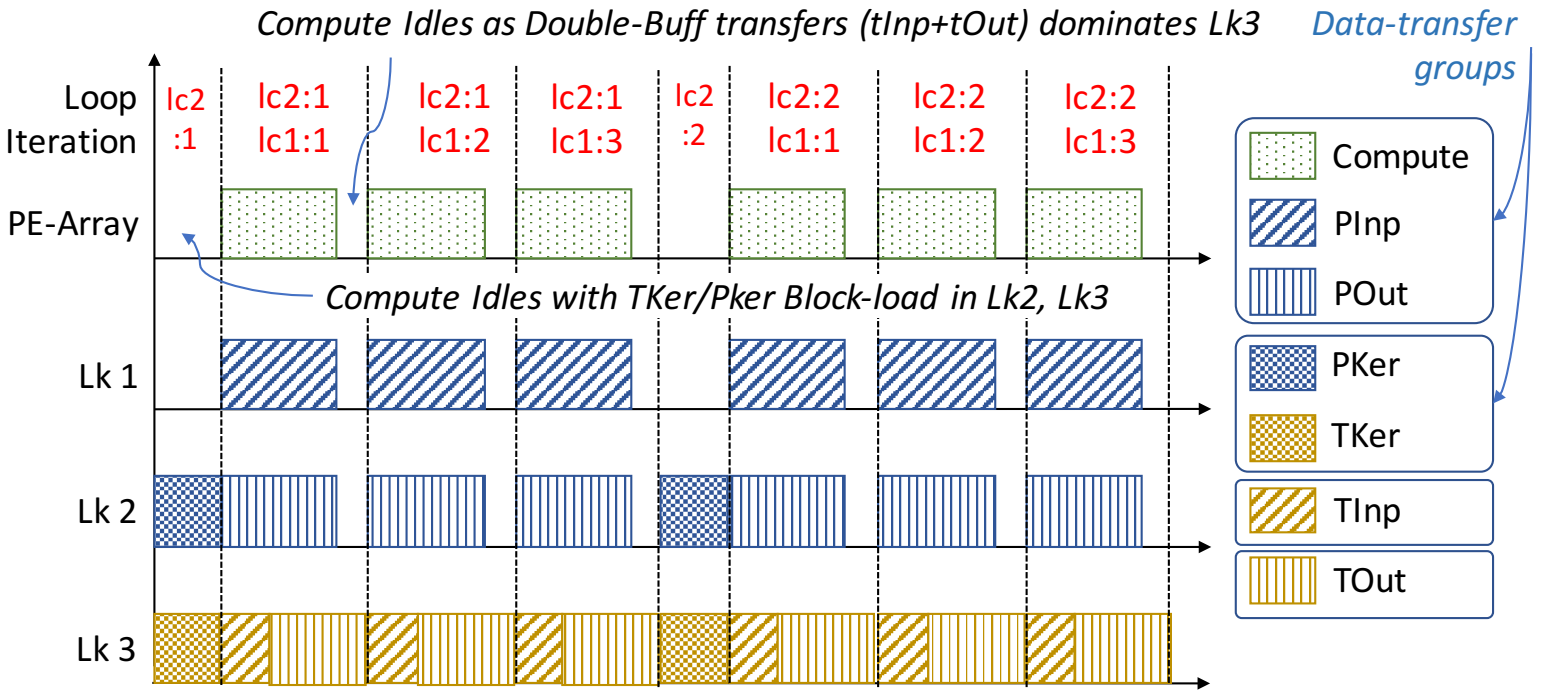


Program

1. transfer tOut
2. *for 1 → lc2: (= 2)*
3. transfer tKer
4. transfer pKer
5. transfer tInp
6. *for 1 → lc1: (= 3)*
7. transfer pInp
8. transfer pOut
9. compute()

Trans.	Type	Link
pInp	Stream.	Lk1
POut	Stream	Lk2
pKer	Blk-Load	Lk2
tInp	Dbl-Buff.	Lk3
tOut	Dbl-Buff.	Lk3
tKer	Stream.	Lk3

Example:

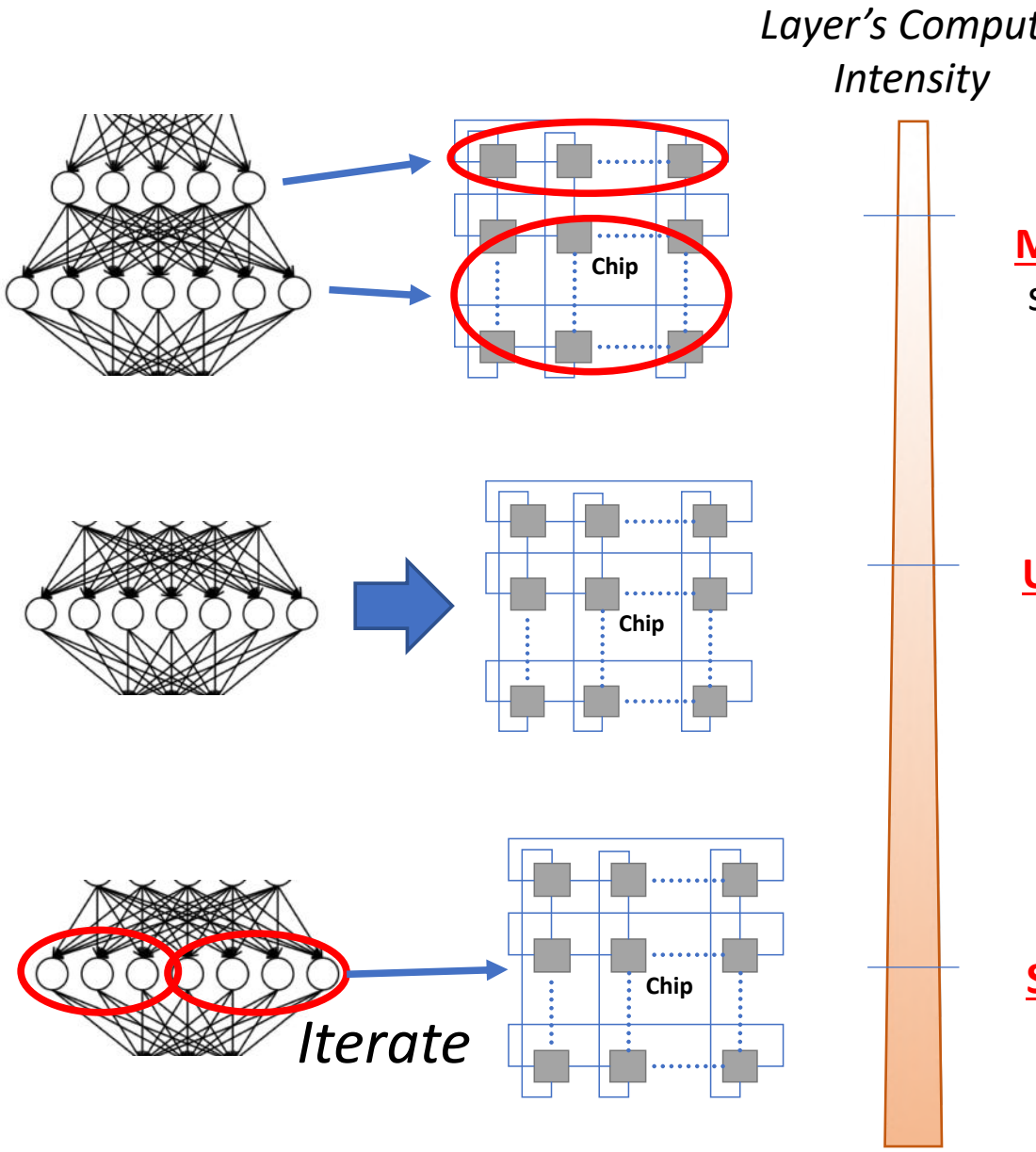


Configuration Estimation (Contd..)

- **Overall performance comprises of 5 major components:**
  - **Compute time**: Time taken by the cores to finish the work assigned to them
    - Performance impacted by how dimensions map to the PE array
      - DSM considers different data-flows & loop sequences
    - **Start-up time**: Some cycles are initially wasted to fetch the first pieces of data from memory before computations can start
  - **Overlapped Communication time**: Data-transfers that are overlapped with compute
    - E.g. Double-buffering data-structures: Pre-fetch Inputs for the second iteration, while the first iteration is in progress
  - **Non-Overlapped Communication time**: Data-transfers that cannot be overlapped with compute
    - E.g. Loading programs, gradient reductions etc.
  - **Auxiliary Compute time**: Time to compute special functions (typically exec. on SFPs)
    - E.g. Activation function, sampling etc.
  - **Pipeline Bubble**: When operations are pipelined, time to fill and drain the pipeline

- **Total Execution Time = MAX (Compute time, CommOverlap time) + CommNonOverlap time + Aux ComputeTime + PipeBubble**

Execution Scenarios



3 possible execution scenarios per layer

Multi-Layer: A layer/pass assigned subset of system resources; multiple layers executed concurrently

Uni-Layer: Every layer/pass executed spatially across entire system; Successive layers executed sequentially

Sub-Layer: Layer/Pass computations broken into subsets and executed iteratively

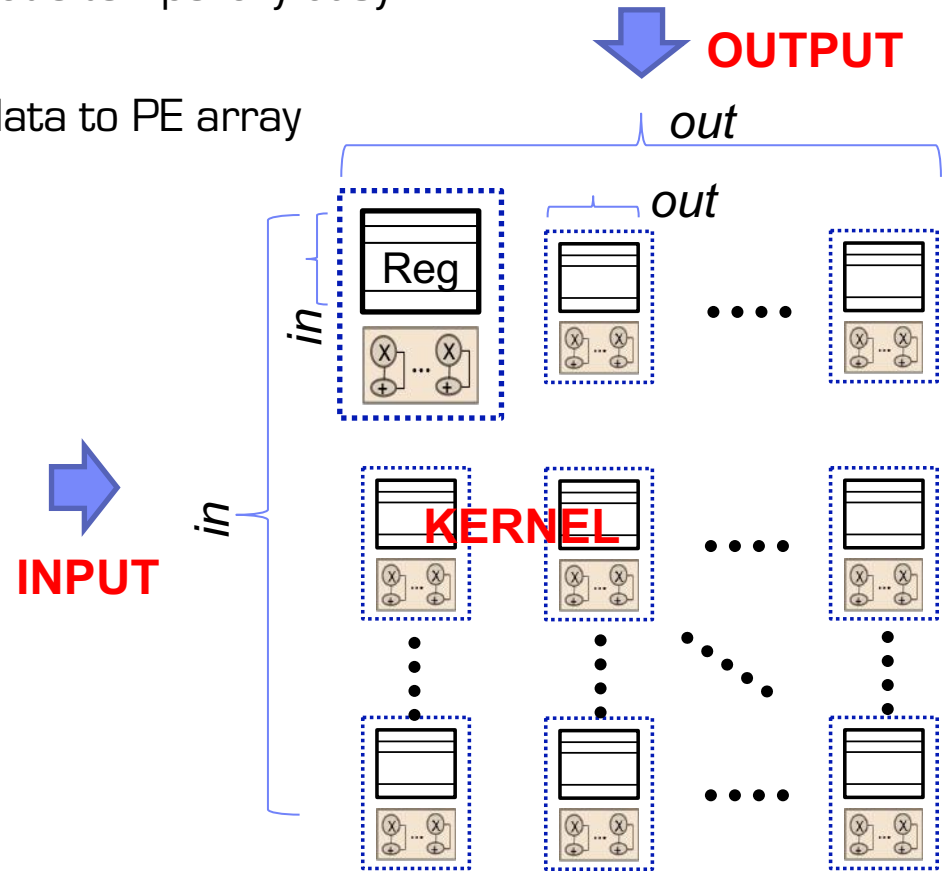
Key Optimizations

DeepTools explore

1. Choice of Dataflow
2. Type of parallelism
3. Inter-layer Memory Reuse
4. Dynamic Spatial Minibatching
5. Timestep Pipelining

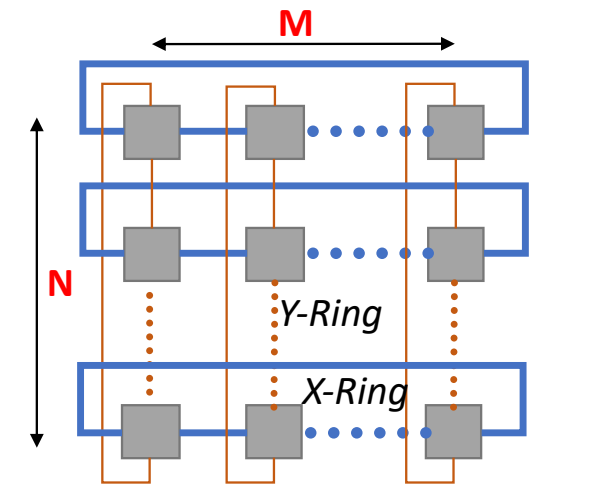
1. Choice of Data Flow

- **Data-flow: How compute is orchestrated within the PE array?**
  - The **direction** we flow the elements of each *data-structure* (INP, OUT, KER) to PEs
  - The **dimension** of that data-structure that is *spatially* mapped
  - The **preferred (minimum) granularity** of work to keep PE array spatio-temporally busy
    - DeepMatrix will try to keep P = P_prefferred, unless N < P_preffered
  - The **type (streaming vs. block-load)** for data-transfers that fetch data to PE array



- **Example: 8x8 PE array & 8 regs/PE; weight stationery dataflow**
  - Direction: INP along row, OUT along cols and KER held in the reg. file of PEs
  - Dimension: "in" → rows, "out" → cols
  - P_pref{in,out,ij,mb,kij} = {64,64,1,1,1}
  - Transfer type: Pin = streaming, Pout = streaming, Pker = block-load

3. Hybrid Parallelism



2D-tours w. asymmetric bandwidth: **X-ring BW** > **Y-ring BW**

	Data	Model
Data	**All-Data** $ChipD_{OUT} = N_{OUT}$ $ChipD_{MB} = N_{MB}/MN$ Grad. reduction along X & Y	**ModelXDataY** $ChipD_{OUT} = N_{OUT}/M$ $ChipD_{MB} = N_{MB}/N$ Fea. rotation in X Grad. reduction in Y
Model	**DataXModelY** $ChipD_{OUT} = N_{OUT}/N$ $ChipD_{MB} = N_{MB}/M$ Grad. reduction in X Fea. rotation in Y	**All-Model** $ChipD_{OUT} = N_{OUT}/MN$ $ChipD_{MB} = N_{MB}$ Fea. rotation along X & Y

Parallelism in X-dir.

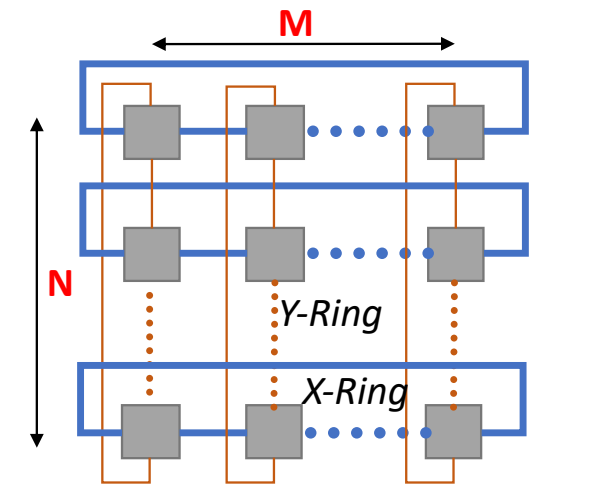
Parallelism in Y-dir.

Preferred Parallelism

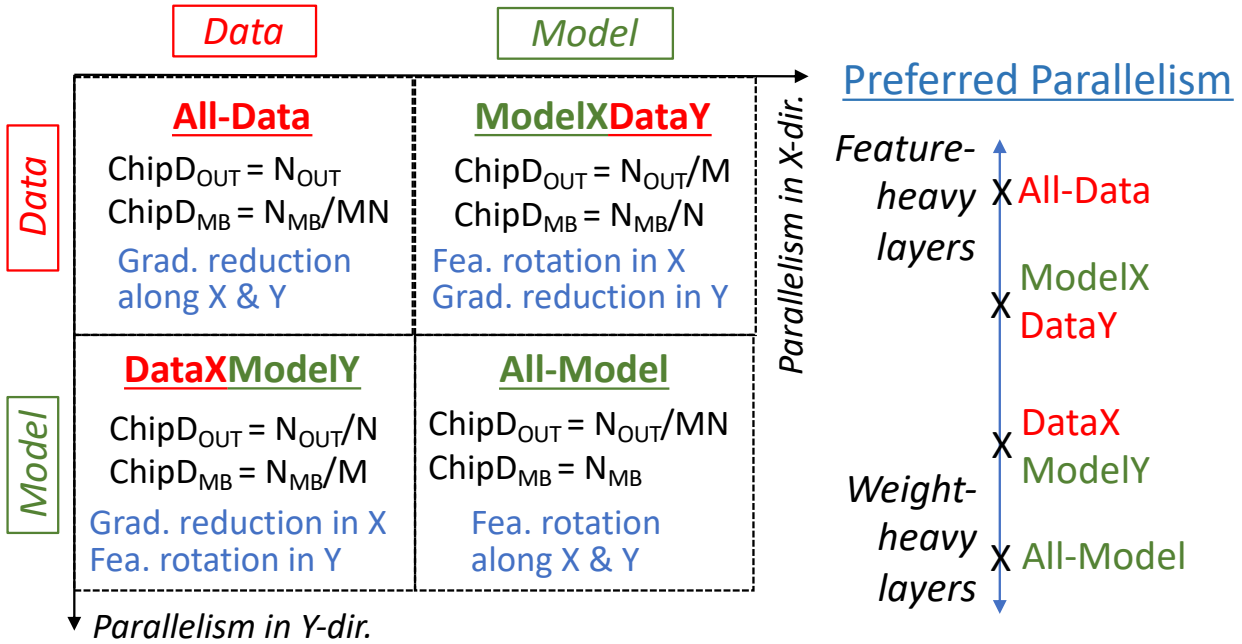
Feature-heavy layers	X All-Data
	X ModelX DataY
	X DataX ModelY
Weight-heavy layers	X All-Model

- **Propose 2 new forms of hybrid parallelism – ModelXDataY and DataXModelY**
  - Combination of data and model parallelism along the X and Y directions
  - Involves gradient direction in one direction and feature communication in the other
- **Asymmetric 2D-torus design: Bandwidth along one direction > bandwidth along the other direction**
  - Enables either fast weight gradient reduction or feature communication based on the type of hybrid parallelism chosen

3. Hybrid Parallelism Contd.



2D-tours w. asymmetric bandwidth: **X-ring BW** > **Y-ring BW**

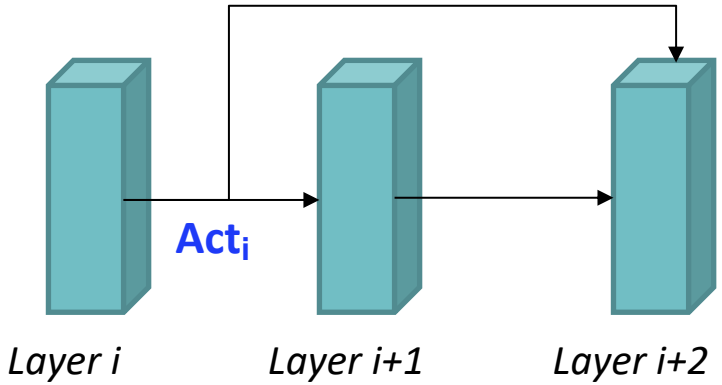


- **Advantages:**
  - In some cases, full data or model parallelism may not be feasible
    - E.g. Data parallelism is not feasible when Nmb < #chips
  - Many middle convolutional layers lie in the spectrum where they are not completely feature or weight dominant
- **DeepSpatialMatrix explores each form of parallelism and picks the best for each layer**
  - Data re-layout cost when adjacent layers use different parallelism is also accounted

4. Inter-layer and Timestep Memory Reuse

- **Typically, each layer reads/writes its inputs/outputs from external memory**
  - Performance of many layers limited by available memory bandwidth
- **Idea: Reduce bandwidth by *holding* selected data-structures in on-chip scratchpad**

- **Reuse of Activation (and De-activation):**
  - Activations produced by a given layer reused by one or more layers after it
  - Hold activations on-chip until the last consumer, which then writes it back to external memory for use in BWD pass



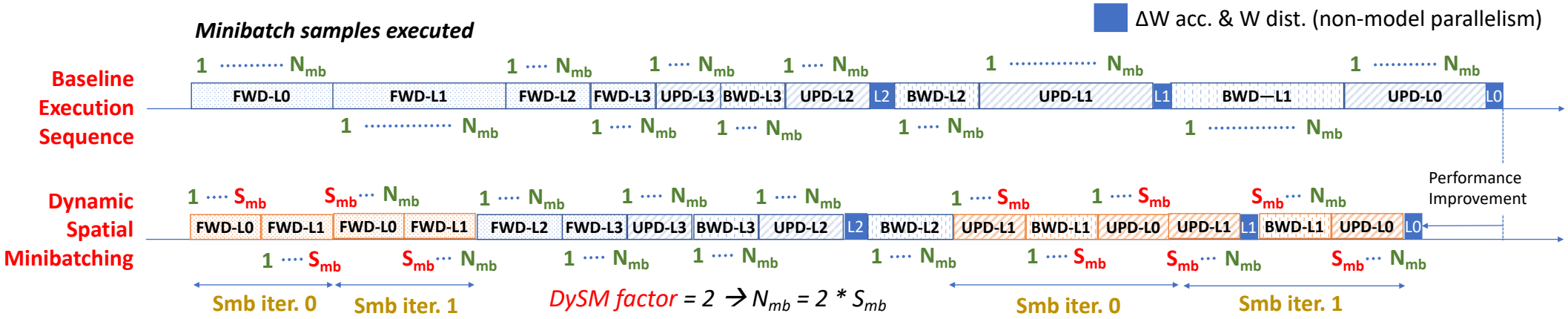
  - Given a finite spad capacity, data structures compete for the same spad space
    - Identify critical data-structures that impact performance the most
  - Work division should be compatible with data-layout in spad to performance penalty

- **Reuse of Weights**
  - Training: Weight reuse across timesteps in LSTM networks and recurrent models
  - Inference: Pre-load weights of selected layers on-chip and reuse across multiple inference jobs
  - Refer to paper for more details!

5. Dynamic Spatial Minibatching (DySM)

- **Observation: In some layers, working set (activations/deactivations) is too large to fit on-chip and prevents inter-layer memory optimizations**

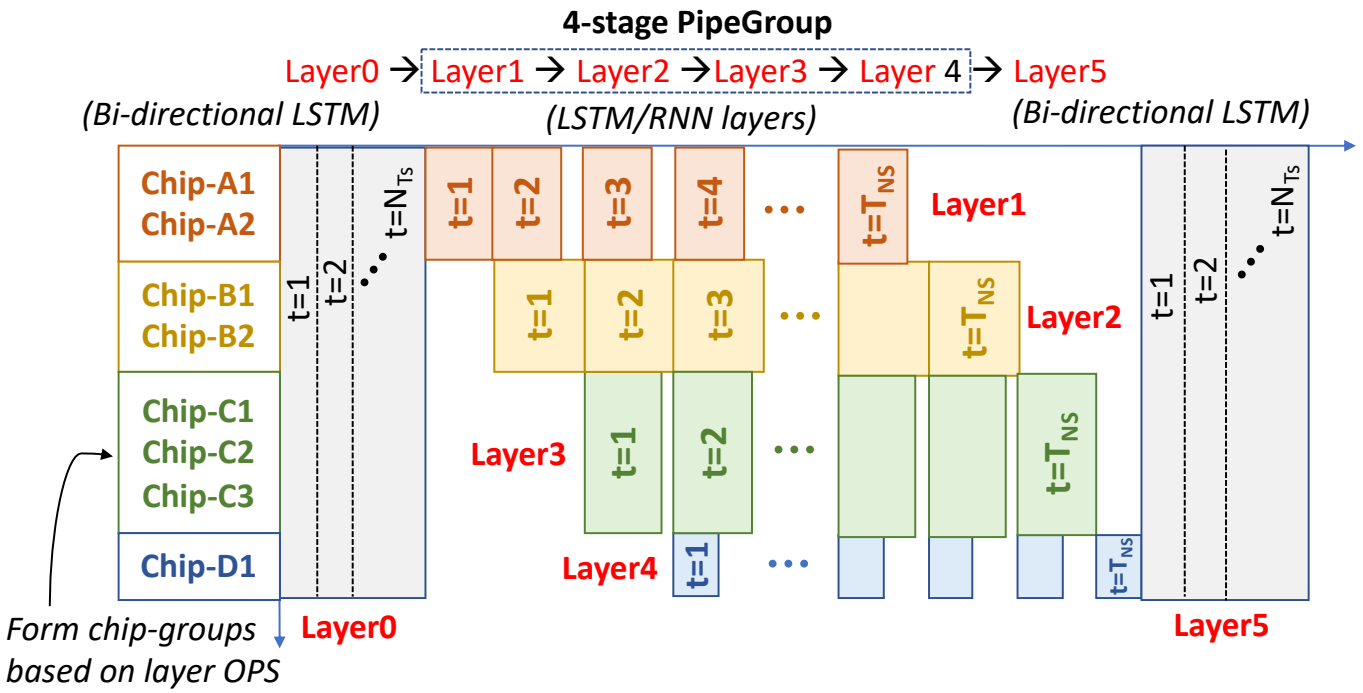
- **Idea: Break layer computation along minibatch dimension into "spatial minibatches"**
  - After a spatial minibatch, proceed to the next layer keeping activations on-chip
  - Return back to execute other spatial minibatches for the layer in sequence
  - Spatial minibatch size varied across groups of layers based on working set size
  - NOTE: Weight update happens only after all spatial minibatches are complete



- **Performance tradeoff with DySM**
  - Pro: Reduces bandwidth of activations/deactivations by fitting on-chip
  - Con: Increases bandwidth for weights → fetched once for each spatial minibatch
  - Works best for activation-dominant layers

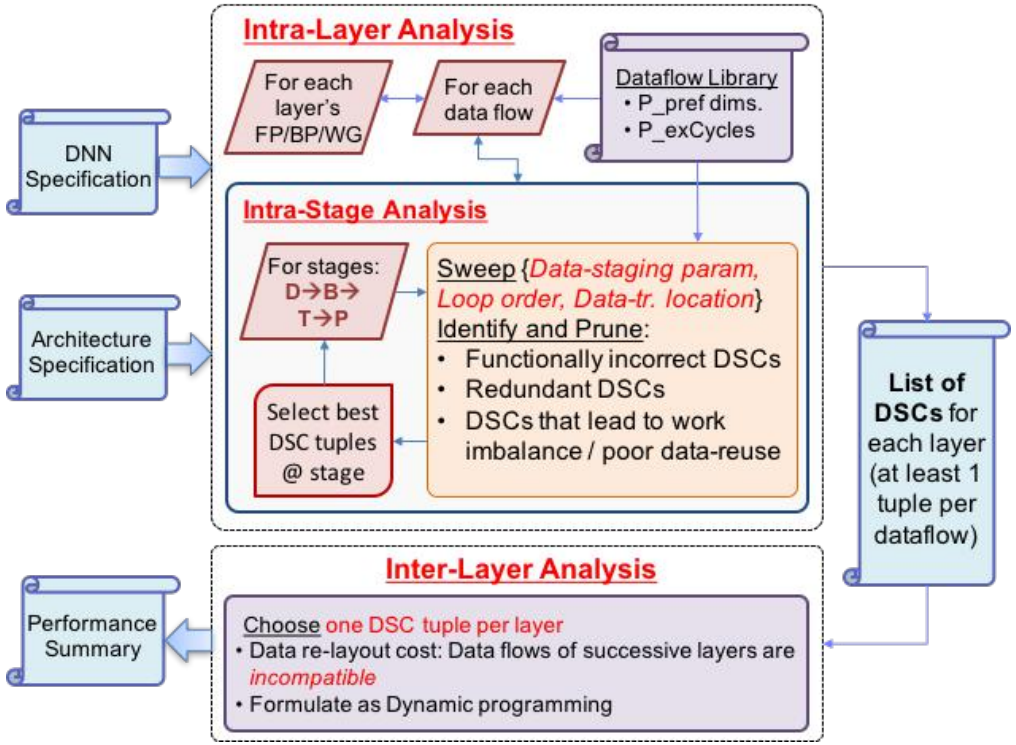
6. Timestep Pipelining

- **Pertains to applications with:**
  - "Timestep" dimension (e.g. LSTM network)
  - Work/layer is small to parallelize it across the entire system
- **Map multiple layers spatially across different chips based on their compute OPS**
  - Successive layers are placed as close as possible
- **Execution pipelined across timesteps**
  - Chips pertaining to each layer concurrently execute a different timestep



Design Space Exploration

- **Enormous search space (~100 parameters) – need a good search/pruning strategy**
- **Intra-layer analysis**
  - For each layer → For each Parallelism → For each dataflow → For each loop-stage:
    - Pruning strategies:
      - Choose combinations that lead to balance in work across cores or loops
      - Choose loop order and data-transfer locations that maximize data reuse
      - Identify redundant configurations and explore them only once
    - Get a list of "top" combinations at least one for each data-flow
- **Inter-layer analysis**
  - Select one combination for each layer while considering cost of re-laying data
    - Data-flows corresponding to successive layers may not be compatible
  - Solved optimally in polynomial time



Results

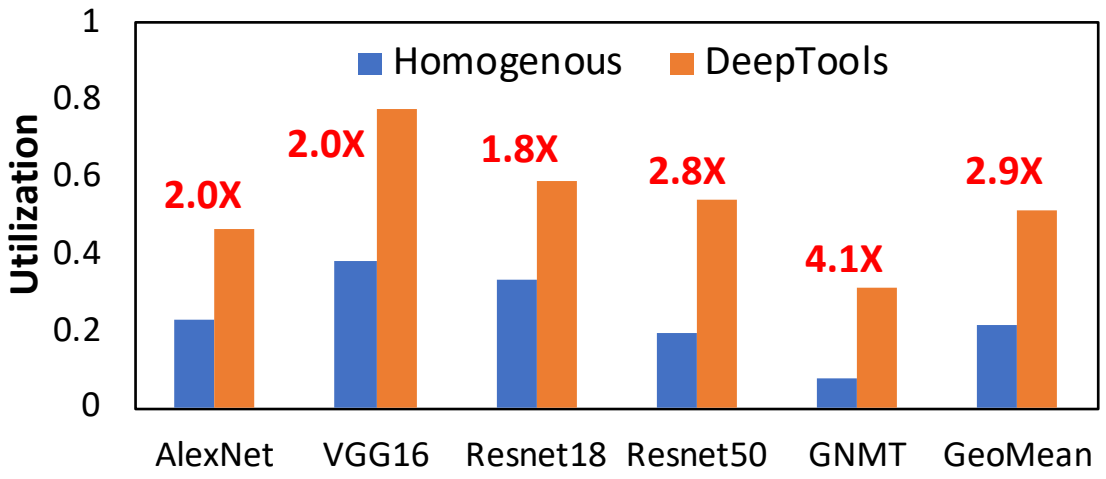
- **Architecture: System with 8 PFLOPs (half-precision) peak processing power**
  - 64 chips, 32 cores/chip, 1024 MACs/core
- **Performance model calibrated with measurements from fabricated chip at 14nm**
- **DeepTools explored over a million mapping configurations in <15 mins**

- **Benchmark: Heterogenous selection of DNNs**
  - Convolutional neural networks
    - AlexNet, VGG16: Many compute-heavy layers
    - ResNet18, Resnet50: Many lean layers that are memory-bound
  - LSTM network
    - GNMT: Very small work per layer per timestep

- **1.8X-4.1X performance improvement over hand-tuned mapping**

Bold → Baseline configuration; *{italics}* → Range used for sensitivity studies

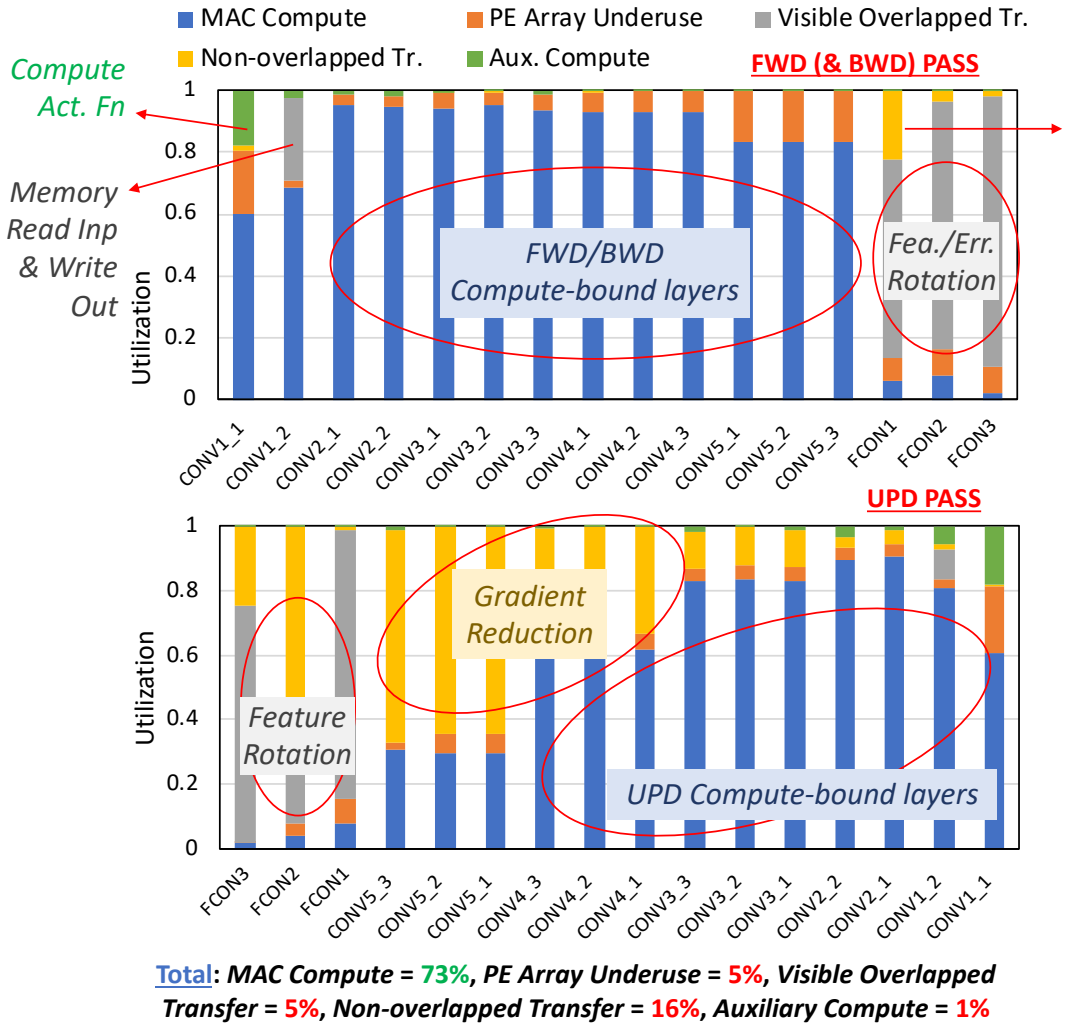
System Params.	Number of Chips			**64** *{16-256}*
	Chip Params.	Number of Cores		**32**
		Core Params.	Num. of MACs (FP16)	**1024**
			Spad Mem. (MB)	**1** *{0.5-4}*
			Spad. Bandwidth (GBps)	**128**
			Frequency (GHz)	**2**
		Chip Topology		**Ring**
		Core-to-Core Bandwidth (GBps)		**256**
		External Mem. Capacity (GB)		**8**
		External Mem. Bandwidth (GBps)		**256** @ 80% eff.
	System Topology			**2D-Torus** Chips X,Y: **4, 16***{4,64}*
	Chip-to-chip Bandwidth (GBps)			Symm. - X: **80** Y: **80**; Asymm. - X: **120** *{30-240}* Y: **40** *{10-80}*



VGG16: Layerwise study

- DSM+DM automatically modulates how each layer is mapped based on the layer's characteristics – Initial CONV vs. Mid CONV vs. Fully-connected layers
- Detailed view into performance bottlenecks for each layer

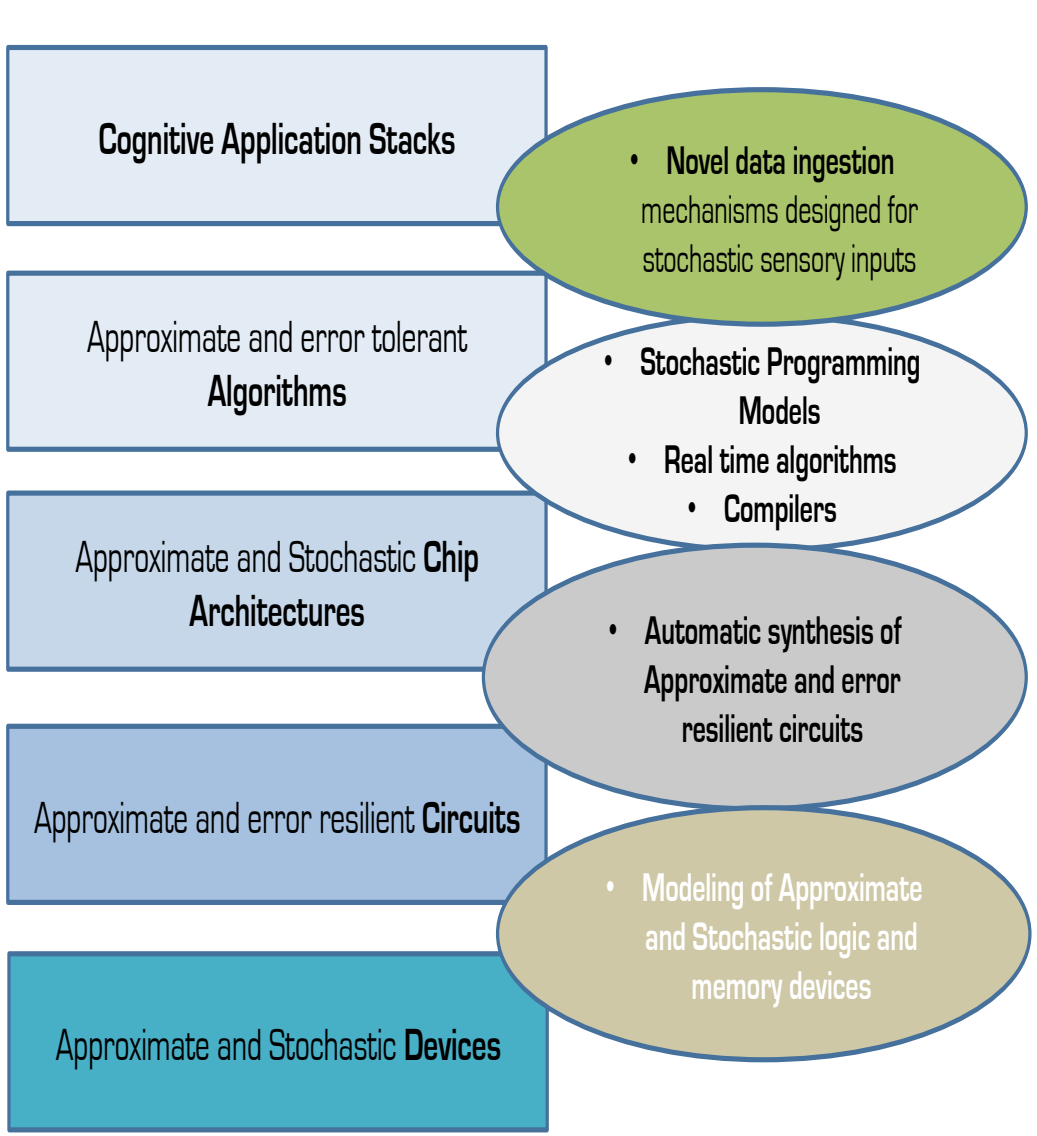
Layer FWD/BWD/UPD	Parallelism Type	CoreD Split				
		in	*out*	*ij*	*mb*	*kij*
CONV1_1	Data			32		
CONV1_2	Data			32		
CONV2_2	Data		2	8	2	
CONV3_1	Data	4	4	2		
CONV4_3	Data		8	2	2	
CONV5_1	Data	8	4			
FCON1	Model				32	
FCON2	DataX-ModelY		4		8	
FCON3	DataX-ModelY				32	



Total: *MAC Compute* = 73%, *PE Array Underuse* = 5%, *Visible Overlapped Transfer* = 5%, *Non-overlapped Transfer* = 16%, *Auxiliary Compute* = 1%

Approximate Computing

Approximate Computing Overview and Techniques



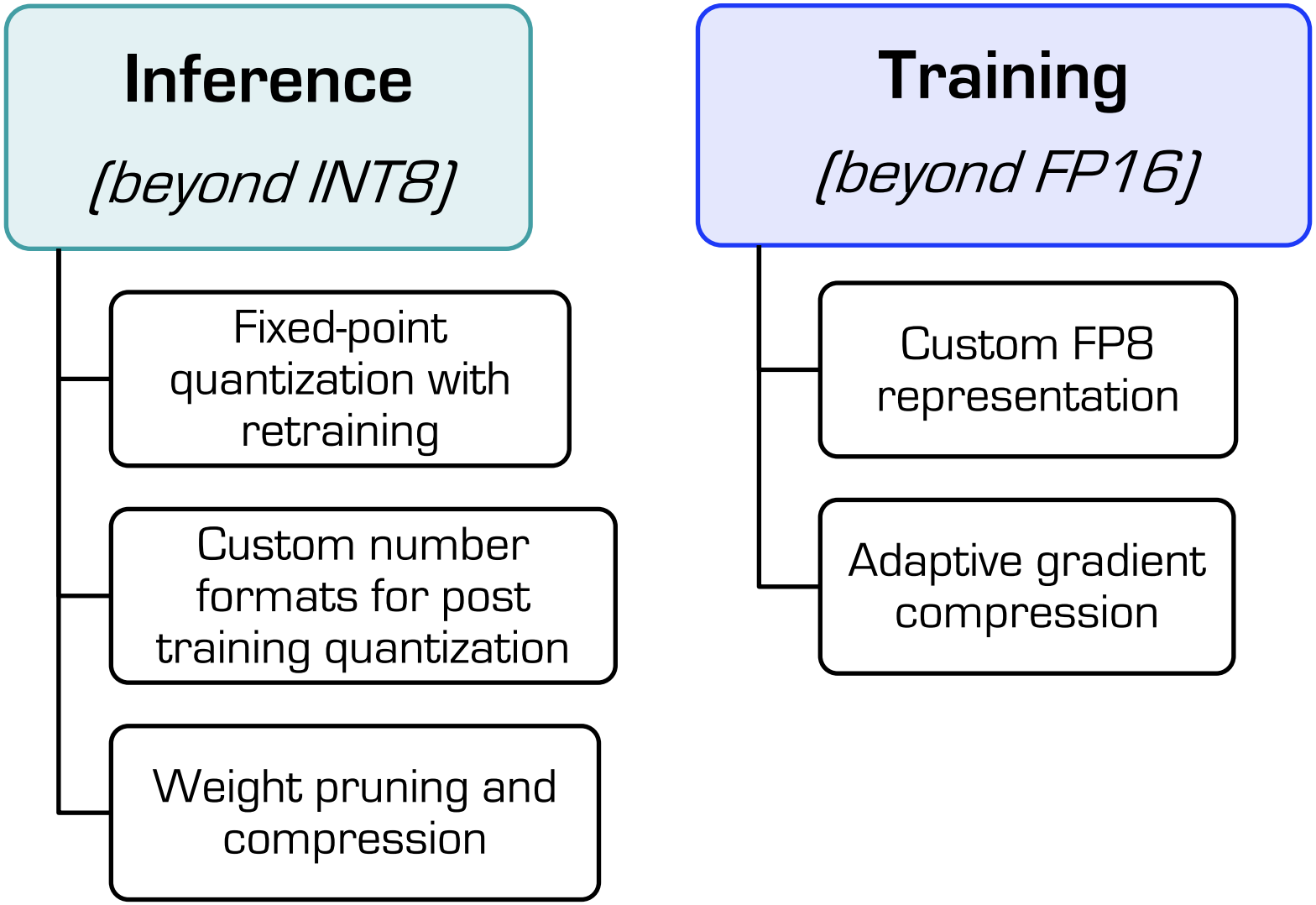
Near Term

Medium Term

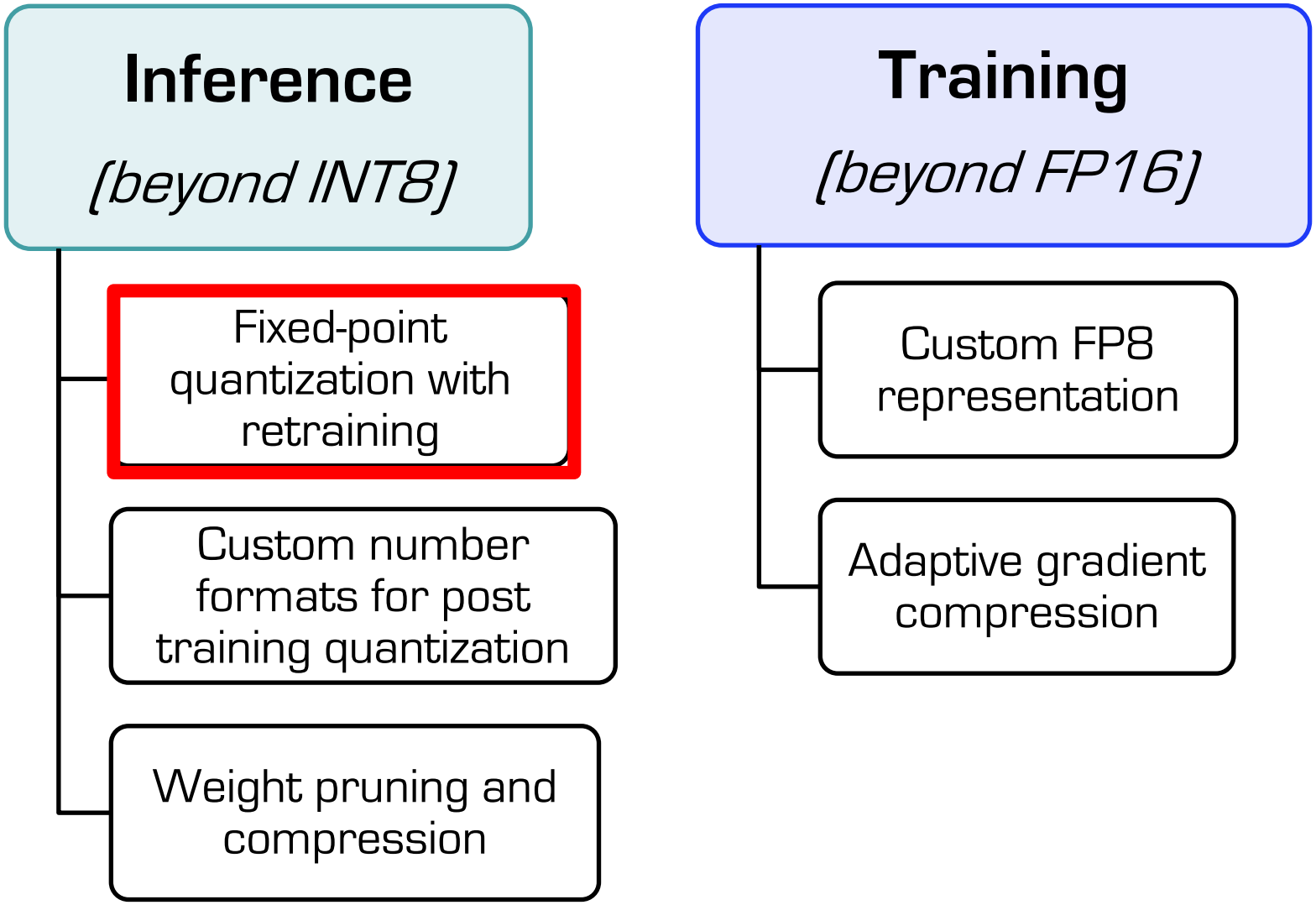
Long Term

- **Large spectrum** of cross-stack approximate computing techniques available.
- 3 Primary techniques (already) being used widely in DL
  - **Precision:**
    - **Scaled precision** for Training and Inference
    - Maximum bang for the buck (**quadratic gains in efficiency w. precision**)
  - **Compression:**
    - Lossy compression to minimize data communicated between ASICs for training.
  - **Synchronization:**
    - (Mostly) SW techniques to minimize synchronization overheads for distributed training.

Algorithmic Approximations for AI

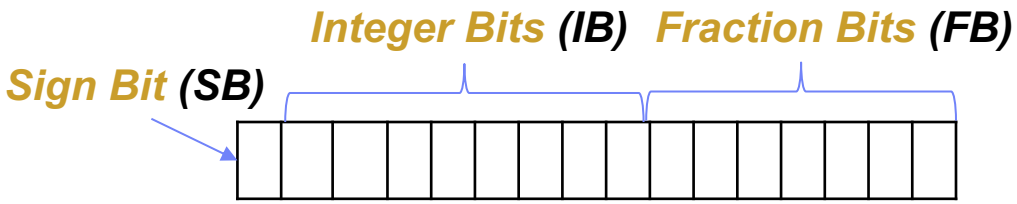


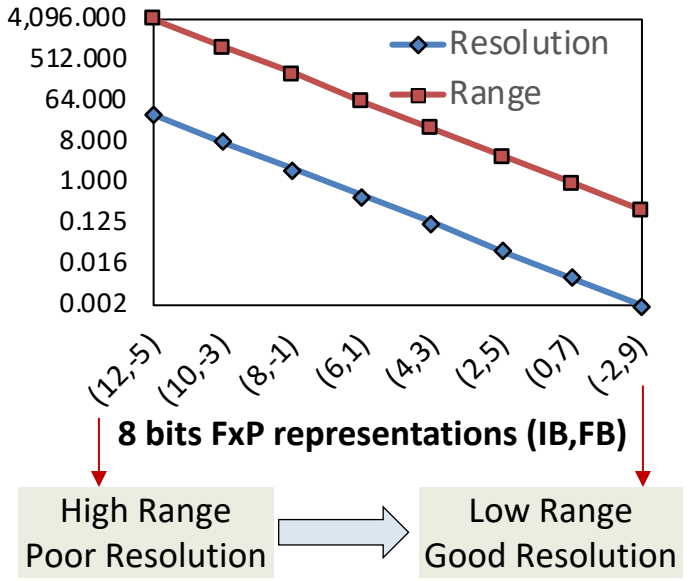
Algorithmic Approximations for AI



Quantized Deep Neural Networks

- **Implement DNN weights and activations using Fixed-Point (FxP) representation**





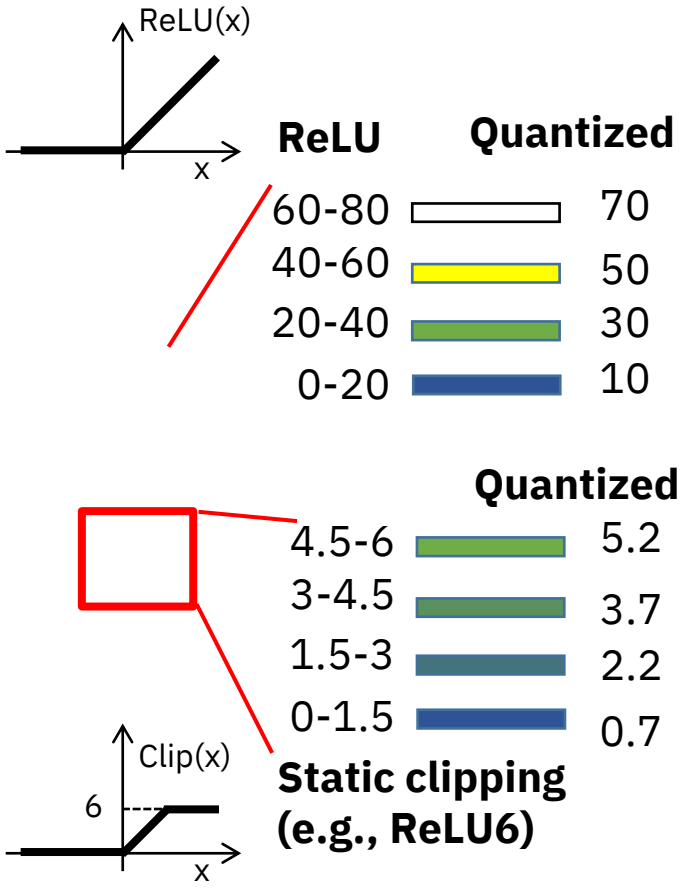
  - Number of IB and FB bits determine the *range* and *resolution* respectively
  - The dynamic range of weights and activations are different for each layer

- **Advantages**:
  - Smaller ALUs ➔ lead to power and performance benefits
  - Smaller memory footprint
  - Smaller data elements ➔ increase data transfer efficiency

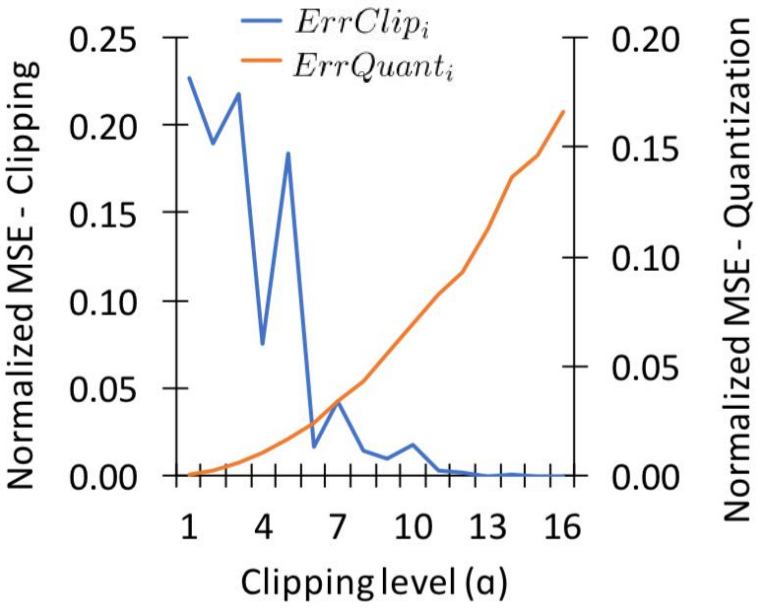
- **Challenge: Invariably suffer from quantization errors, which degrades accuracy**
  - Choose the right range and resolution for each data-structure
  - Retrain the network considering quantization errors

Inference : Challenges in Low Precision Activation Quantization

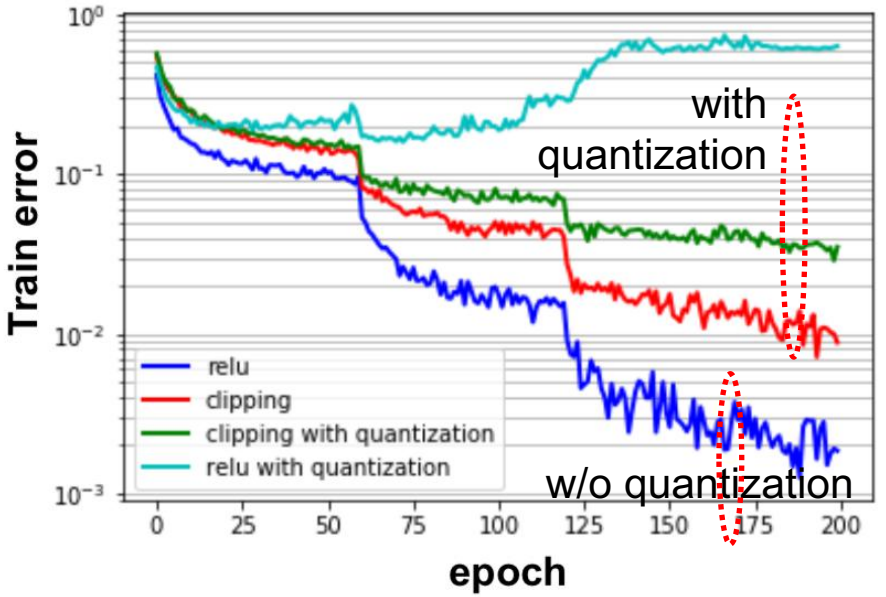
- **Trade-offs when quantizing activation**
  - **ReLU**: *Cover large dynamic range* (better convergence) ➔ **Suffer large quantization error**
  - **Static clipping**: *Clip outliers* (lower quant error) ➔ Accuracy drop due to ***diminishing gradients***



Trade-off between Quantization & Clipping Error



CIFAR10 ResNet20: **ReLU vs Clipping when quantizing activation**



Hard to find sweet spot ➔ *Automatic tuning via training!*

J. Choi et. al (presented at SysML 2019)

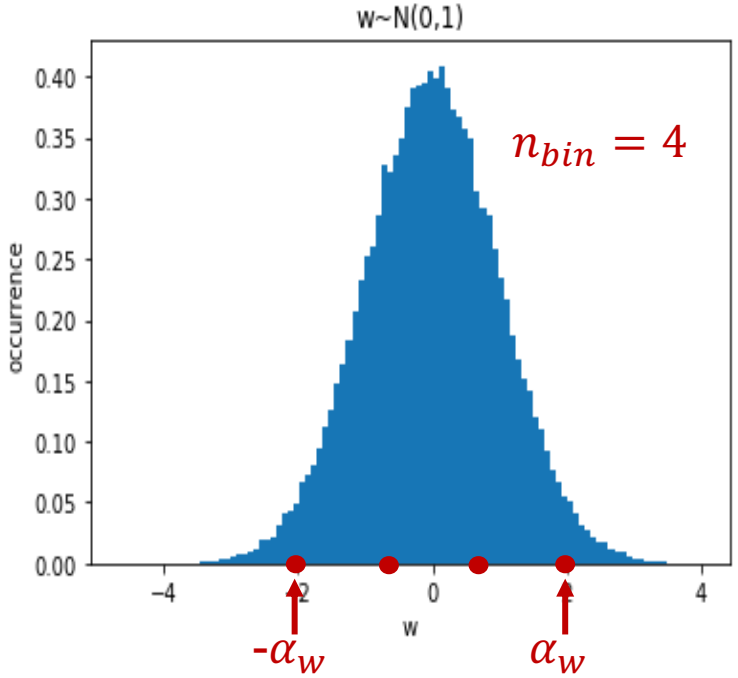
Inference : Challenges in Low Precision Weight Quantization

- **Objective: Find a good *quantization scale* (α_w)**
  - Assumption: symmetric distribution (usually true…), Uniform quantization (for simple HW)
  - Given a quantization scale, quantized weights are exclusively determined
  - Goal: find α_w that minimizes quantization error

$$\alpha_w^* = \arg\min_{\alpha_w} ||w - w_q||^2$$

- **Previous approach: Find α_w with respect to $E(|W|)$**
  - E.g., XNOR-Net, Ternary-Weight-Quant, etc.,
  - Pros: Simple (sometimes with analytic solution)
  - Cons: $E(|W|)$ is not enough to capture shape of W

Need an analytic solution that better characterizes weight distribution



Ex: XNOR-Net (Rastegari et al., 2016)

$$J(\mathbf{B}, \alpha) = \|\mathbf{W} - \alpha\mathbf{B}\|^2$$

$$\alpha^*, \mathbf{B}^* = \underset{\alpha,\mathbf{B}}{\operatorname{argmin}} J(\mathbf{B}, \alpha)$$

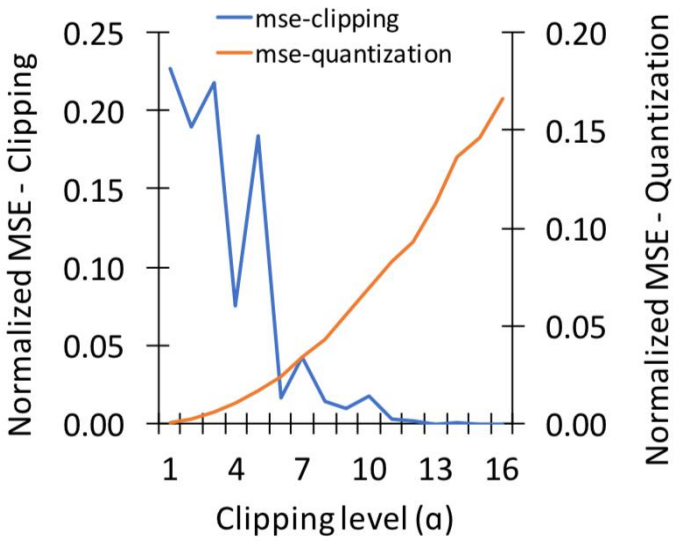
$$\alpha^* = \frac{\mathbf{W}^\top \operatorname{sign}(\mathbf{W})}{n} = \frac{\sum |\mathbf{W}_i|}{n} = \frac{1}{n}\|\mathbf{W}\|_{\ell 1}$$

J. Choi et. al (presented at SysML 2019)

New Techniques for Inference with Hyper-Scaled Precision

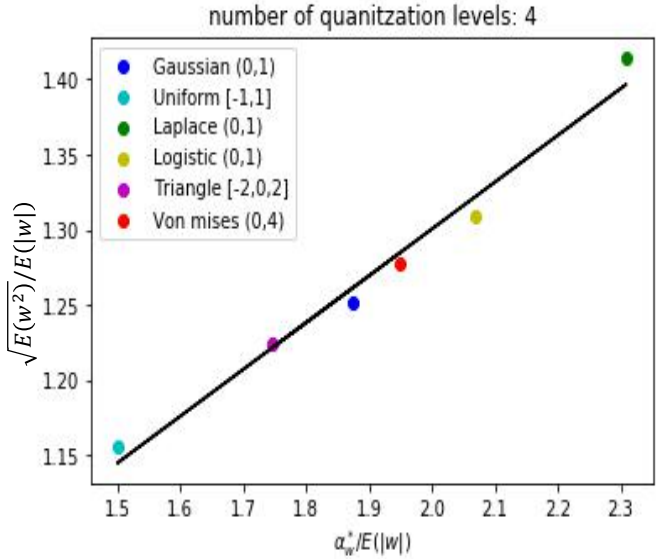
Activation Quantization
PArameterized Clipping acTivation (PACT)

– Automatic tuning of clipping level to balance clipping vs quantization error



Weight Quantization
Statistics Aware Weight Binning (SAWB)

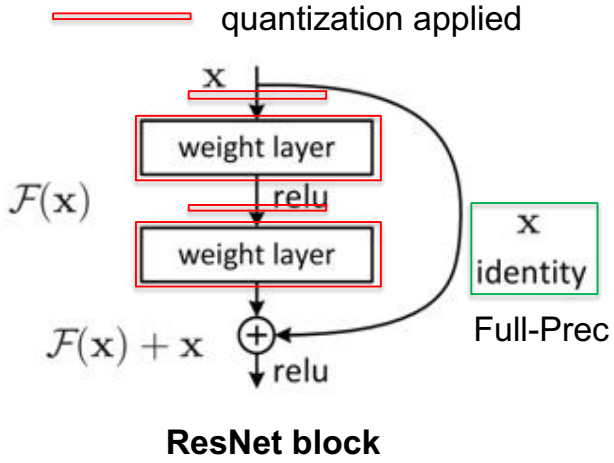
– Exploit weight statistics to better capture shape of weight



$$\alpha_w^* = c_1 \cdot \sqrt{E(w^2)} - c_2 \cdot E(|w|)$$

Quantization in the Presence of Shortcut Connections
Full-Precision Shortcut

– Enhance gradient-flow of training by not quantizing shortcut

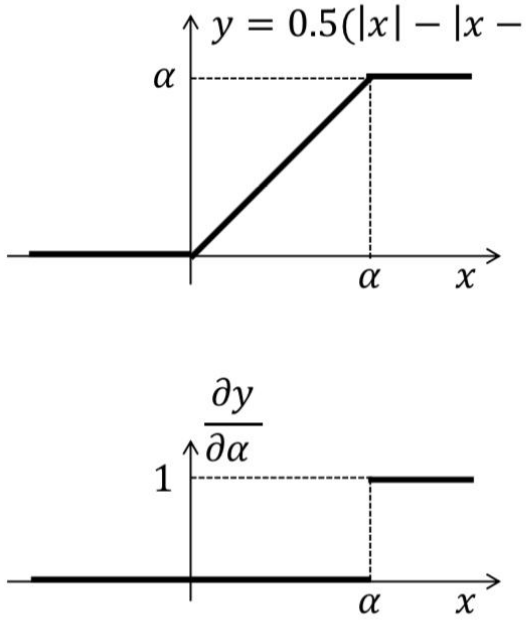


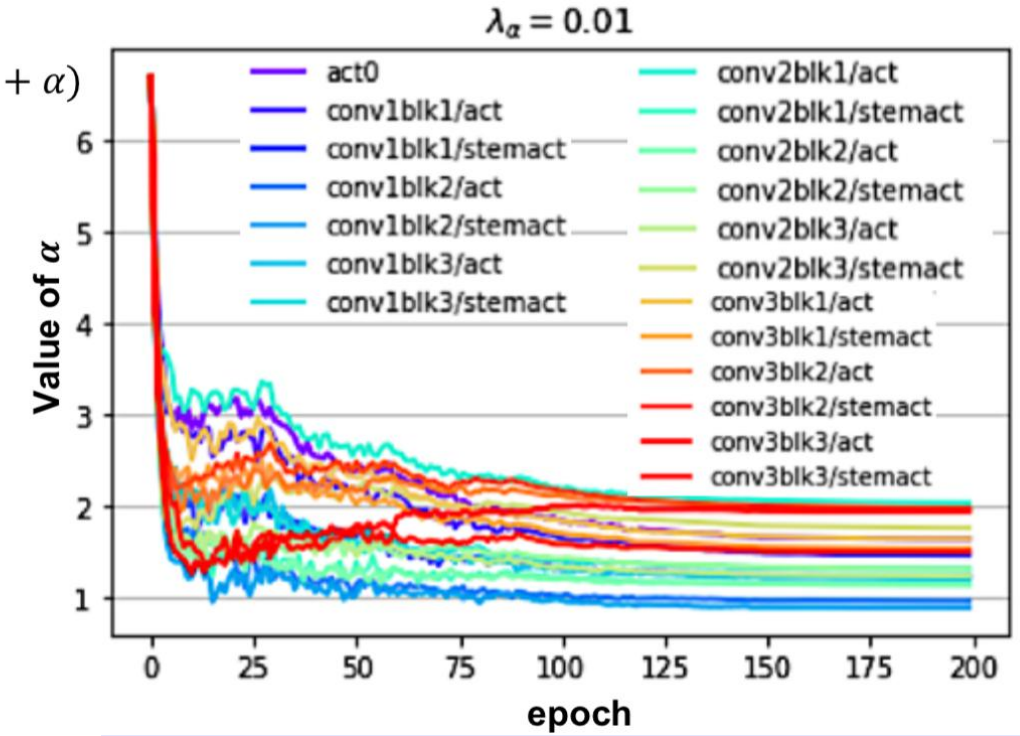
ResNet block

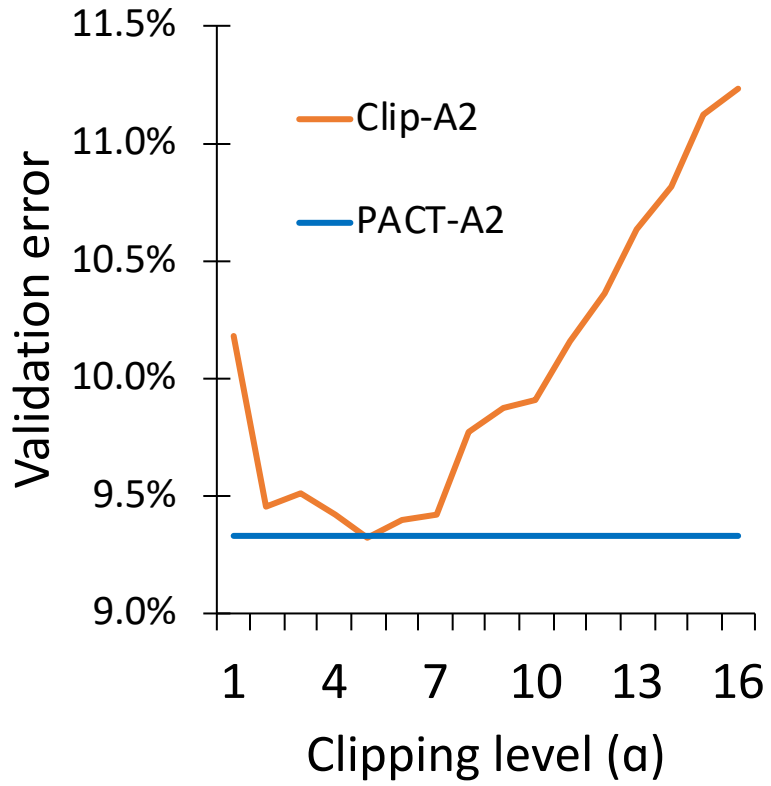
J. Choi et. al (presented at SysML 2019)

Activation Quantization : PArameterized Clipping acTivation (PACT)

- **Clipping level (= α) as a *trainable parameter* → Auto-tuned by Backprop**
  - α is *initialized with large value* to emulate ReLU in the beginning
  - L2-regularization on α → Converges to low magnitude to reduce quantization error
- **Ex: CIFAR10-ResNet20 with PACT**
  - PACT automatically finds the best clipping level without expensive sweeping over α







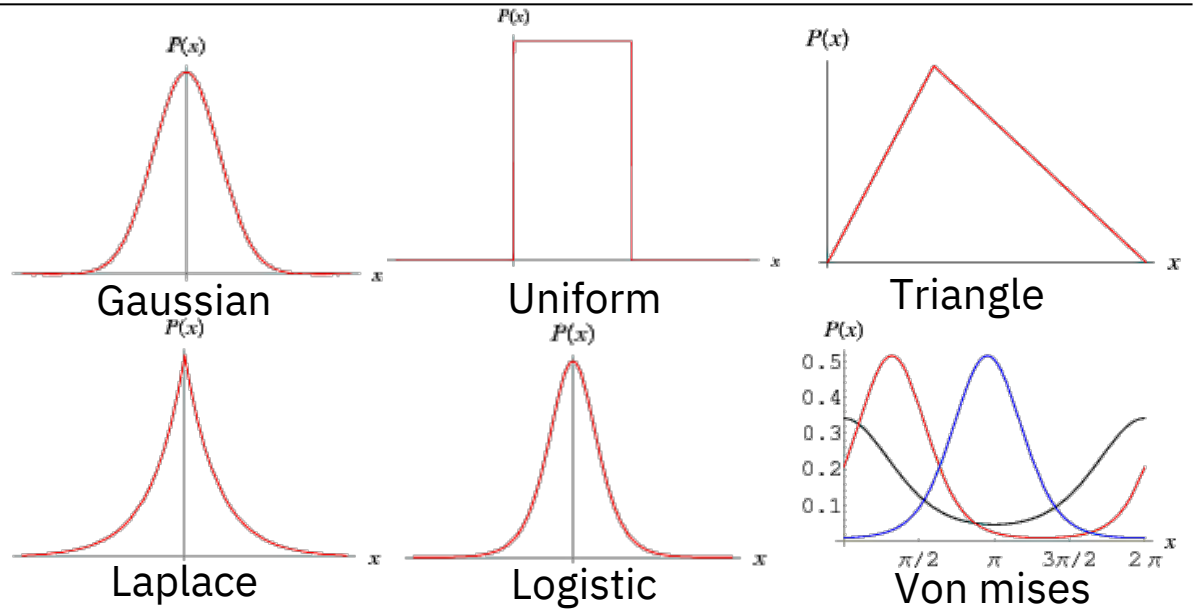
J. Choi et. al (presented at SysML 2019)

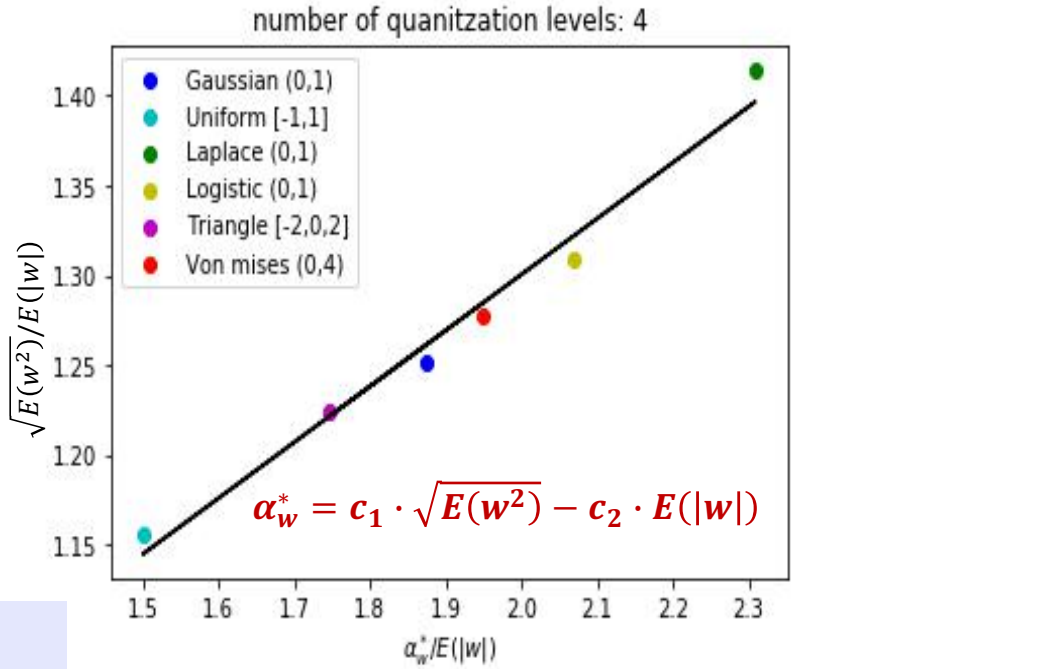
Weight Quantization : Statistics Aware Weight Binning (SAWB)

- **How to better capture shape of weight?**
  - $\boldsymbol{E(|W|)}$ captures the ***representative values***
  - $\boldsymbol{E(W^2)}$ captures the ***overall shape***
  - Use $\boldsymbol{E(|W|)}$ and $\boldsymbol{E(W^2)}$ for finding best $\boldsymbol{\alpha_w}$?

$$\boldsymbol{\alpha_w^* = c_1 \cdot \sqrt{E(w^2)} - c_2 \cdot E(|w|)}$$

- **Verification:**
  - Take 6 representative distributions with varying variance: Gaussian, Uniform, Triangle, Laplace, Logistic, Von mises
  - Sweep over $\boldsymbol{\alpha_w}$ to find one with smallest MSE(W-Wq)
  - Linear regression to find relationship among α_w^*, $E(|W|)$ and $E(W^2)$





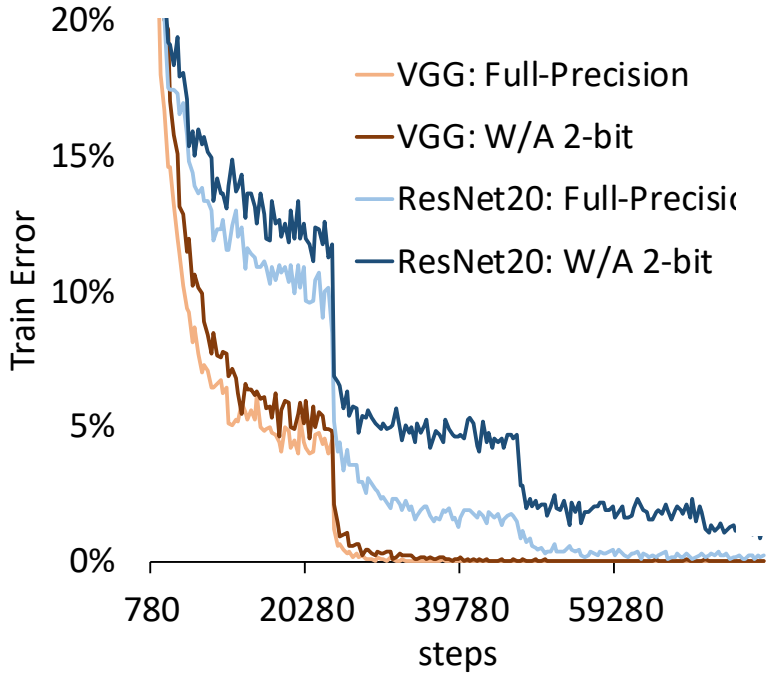
Observation: α_w^* is characterized by $E(|W|)$ and $E(W^2)$ ➔ *Analytic solution* to find the *best scale*

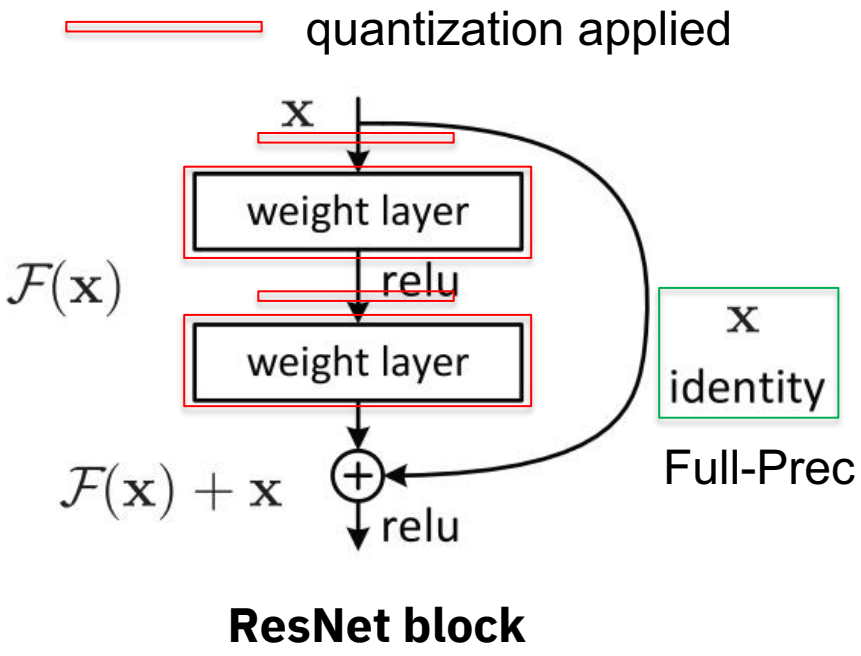
J. Choi et. al (presented at SysML 2019)

Full Precision Short-Cut for Inference (FPSC)

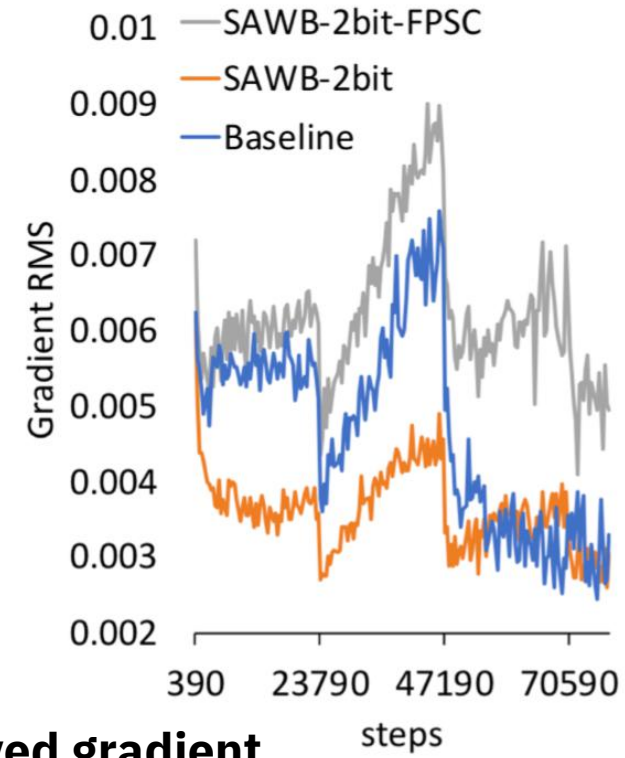
- **Observation: ResNet is more sensitive to quant-error than VGG-like networks**
  - Short-cut in ResNet helps gradients to flow → Quantization on shortcut hinders training
- **Full-Prec Short-Cut: Avoid quantization at input activation and weight in short-cut**
  - Short-cut involves small weights → Full-Prec does not harm performance (<1% ResNet18)
  - FPSC allows larger magnitude gradients → Improved gradient flow

QNN training: VGG vs ResNet





ResNet block



Improved gradient flow by FPSC

J. Choi et. al (presented at SysML 2019)

Hyper-Scaled Precision : Inference Accuracies on Models

CIFAR10 Dataset

Name	Baseline	Quantized	Degradation
	< ResNet20: W2-A32 >		
SAWB-fpsc	**91.8**	**91.6**	**0.2**
DoReFa	91.8	90.9	1.0
LQ-Nets	92.1	91.8	0.3
TWN	91.8	90.9	0.9
TTQ	91.8	91.1	0.6
	< ResNet20: W32-A2 >		
PACT-fpsc	**91.5**	**91.4**	**0.2**
DoReFa	91.5	90.1	1.4
ReLU6	91.5	91.0	0.5
	< ResNet20: W2-A2 >		
SAWB-PACT-fpsc	**91.5**	**90.8**	**0.7**
DoReFa	91.5	88.2	3.3
LQ-Nets	92.1	90.2	1.9
	< VGG: W2-A2 >		
SAWB-PACT-fpsc	**93.8**	**93.8**	**0.1**
LQ-Nets	93.8	93.5	0.3
QIP (lambda=0.5)	94.1	93.9	0.2

ImageNet Dataset

Name	Baseline	Quantized	Degradation
	< AlexNet >		
SAWB-PACT-fpsc	**58.3**	**57.2**	**1.1**
LQ-NETs	61.8	57.4	4.4
QIP (lambda=0.0)	58.1	55.7	2.4
DoReFa-Net	55.1	53.6	1.5
HWGQ	58.5	52.7	5.8
BalancedQ	57.1	55.7	1.4
WEQ	57.1	50.6	6.5
WRPN-x1	57.2	51.3	5.9
WRPN-x2	60.5	55.8	4.7
WRPN-x2, W2-A4	60.5	57.2	3.3
LearningReg	58.0	54.1	3.9
	< ResNet18 >		
SAWB-PACT-fpsc	**70.4**	**67.0**	**3.4**
LQ-NETs	70.3	64.9	5.4
QIP (lambda=0.0)	69.2	65.4	3.8
DoReFa	70.2	62.6	7.6
HWGQ	67.3	59.6	7.7
BalancedQ	68.2	59.4	8.8
LearningReg	68.1	61.7	6.4
	< ResNet50 >		
SAWB-PACT-fpsc	**76.9**	**74.2**	**2.7**
LQ-NETs	76.4	71.5	4.9
Apprentice (W2-A8)	76.2	71.5	3.4
UNIQ (W4-A8)	76.0	73.4	2.6

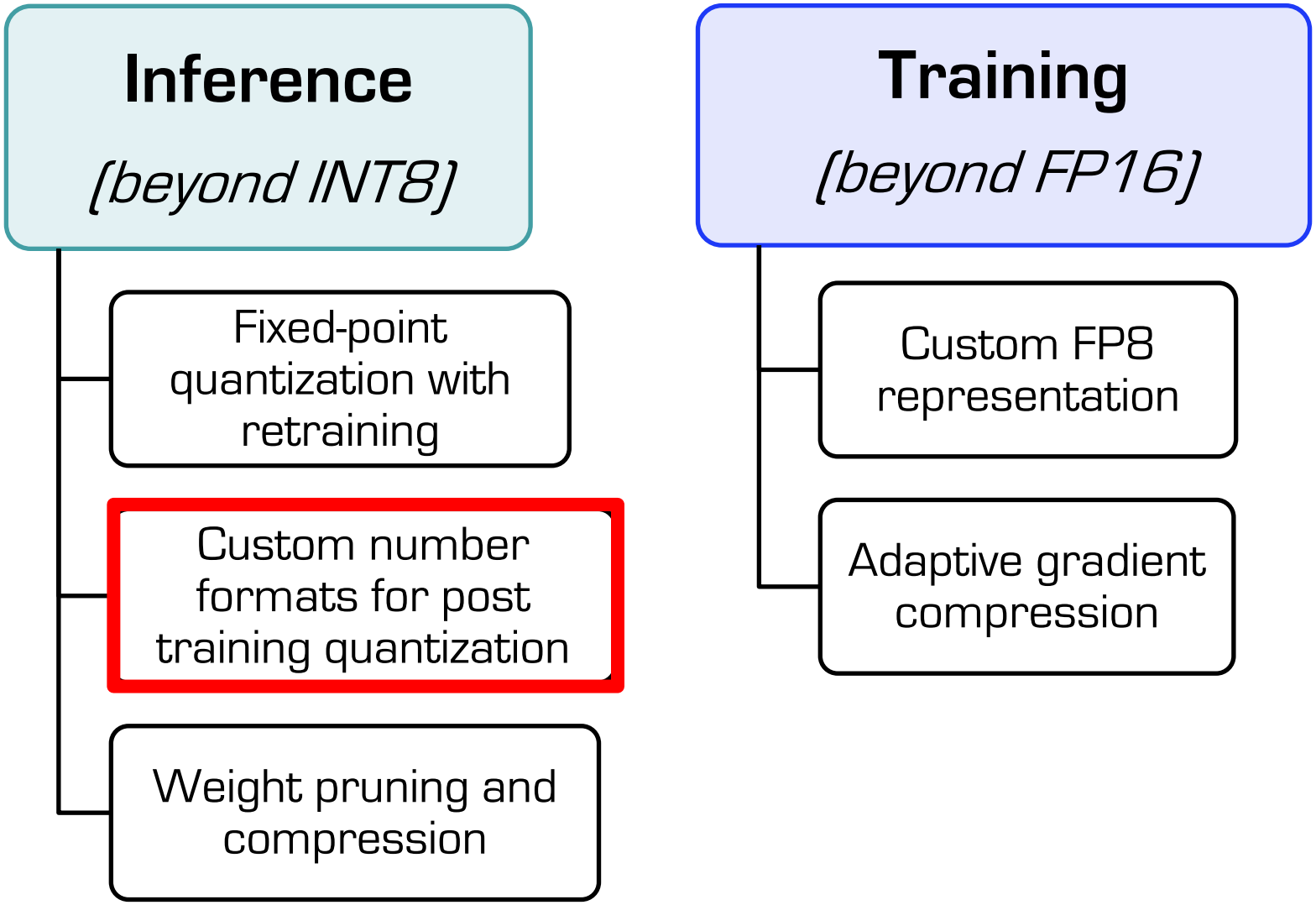
Lowest accuracy degradation

Accuracy higher than prior work

No loss of accuracy for 4-bit Inference

J. Choi et. al (presented at SysML 2019) `

Algorithmic Approximations for AI

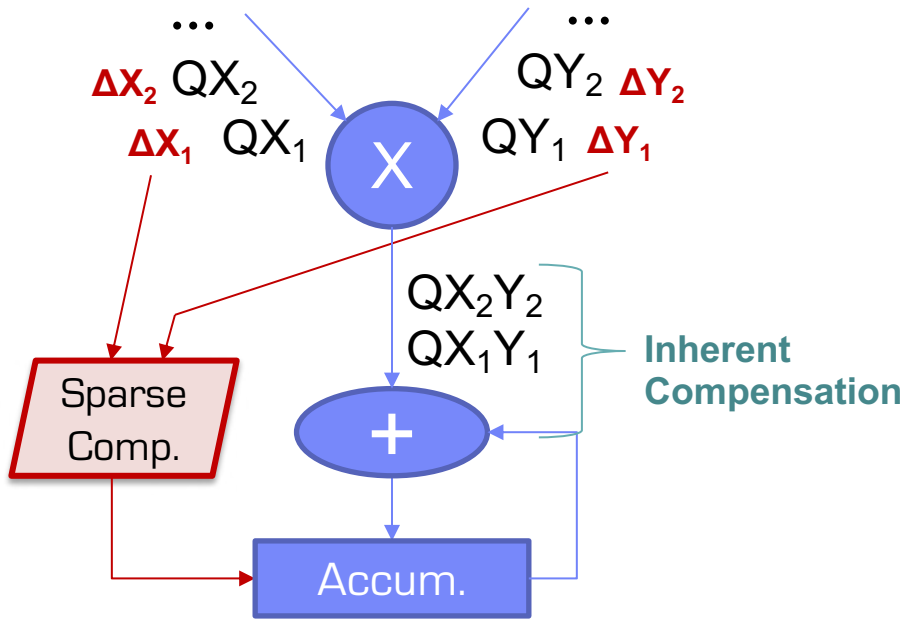


Compensated-DNN: Dynamic Compensation of Quantization Errors

Key Idea: Reduce quantization errors by estimating and compensating at runtime

- **Observation:**
  - In DNNs, the primitive operation is Multiply-and-Accumulate (MAC)
  - Error in MAC output gets accumulated over several (thousands of) multiplications
    - Inherent compensation due to rounding reduces quantization error
      - Positive errors in some multiplications cancelled by negative errors in others
  - Compensated DNNs makes compensation systematic and reliable

- **Proposal:**
  - A new number format to explicitly represent the amount of quantization error
  - A selective compensation scheme to estimate quantization error in output & apply compensation with very little overhead

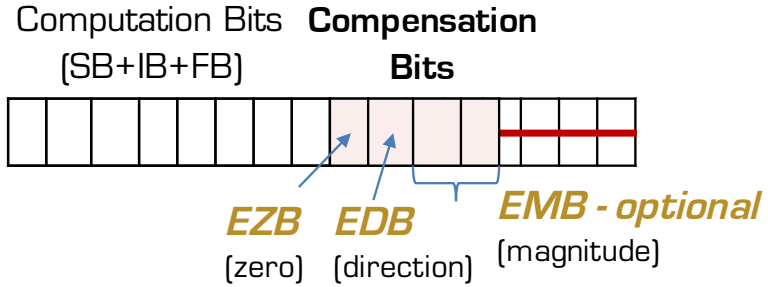


Compensated-DNN: Energy Efficient Low-Precision Deep Neural Networks by Compensating Quantization Errors, DAC 2018

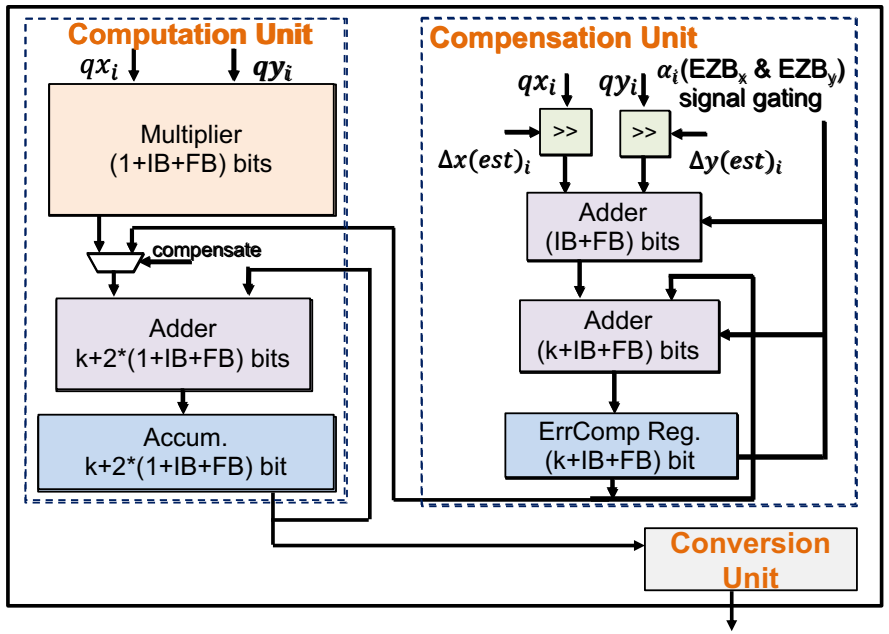
55

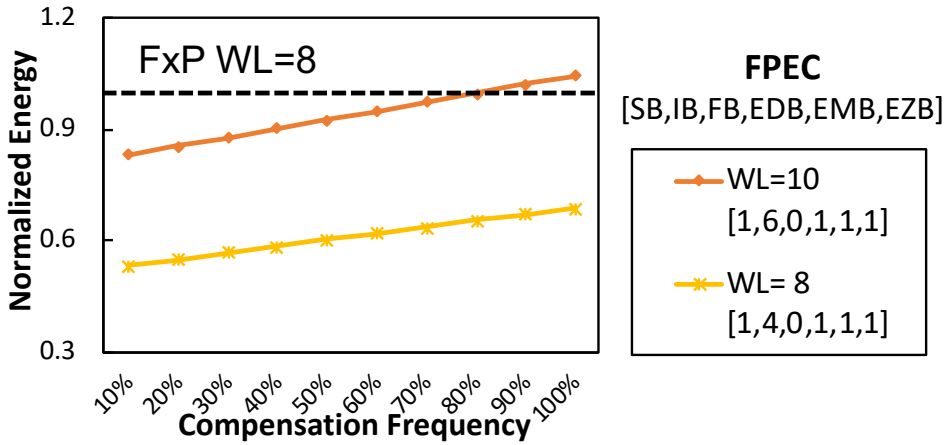
Fixed Point with Error Compensation (FPEC)

- **Shrinks the number of FxP computation bits (SB+IB+FB)**
- **Adds Compensation Bits: 3 new bit fields which capture the amount of quantization error**
  - **Error Zero Bit (EZB):** Single bit to indicate no quantization error
  - **Error Direction Bit (EDB):** Single bit to show direction of error
  - **[OPTIONAL] Error Magnitude Bits (EMB):** 1 or 2 bits to represent magnitude of quantization error
    - Based on the resolution (FB), the max. error magnitude is implicitly inferred



- **The range between two adjacent quantization levels are sub-divided to set EZB/EDB/EMB**





- FPEC vs. FxP: Energy efficiency
  - At iso-bitwidth, FPEC is more energy efficient @ *all* compensation freq.
  - Even at larger bitwidth, FPEC more efficient when comp. freq. is smaller

Compensated-DNN: Energy Efficient Low-Precision Deep Neural Networks by Compensating Quantization Errors, DAC 2018

56

Compensated MAC: Illustration

FPEC Representation: Sign Bit (SB) = 1, Integer Bits (IB) = 3, Fraction Bits (FB) = 0
Error Zero Bit (EZB) = 1, Error Dir. Bit (EDB) =1, Error Mag. Bit (EMB) = 0

Note: FB = 0 → Resolution = 1 → Max. quantization error = ± 0.5

X_i	QX_i	$\Delta X(est)_i$	FPEC (SB/IB/EZB/EDB)
3.3	3	+0.5	0 011 0 0
2.95	3	0	0 011 1 X

Y_i	QY_i	$\Delta Y(est)_i$	FPEC (SB/IB/EZB/EDB)
3.45	3	+0.5	0 011 0 0
2.75	3	-0.5	0 011 0 1

O_{Actual} = 3.3 * 3.45 + 2.95 * 2.75 = 19.4975

O_{Quant} = 3 * 3 + 3 * 3 = 18

$O_{Compensated} = O_{Quant}$ + (3 >> 1 + 3 >> 1) + (0 - 3 >> 1) = 19.5

$Y_1\Delta X_1$ $X_1\Delta Y_1$ $Y_2\Delta X_2$ $X_2\Delta Y_2$

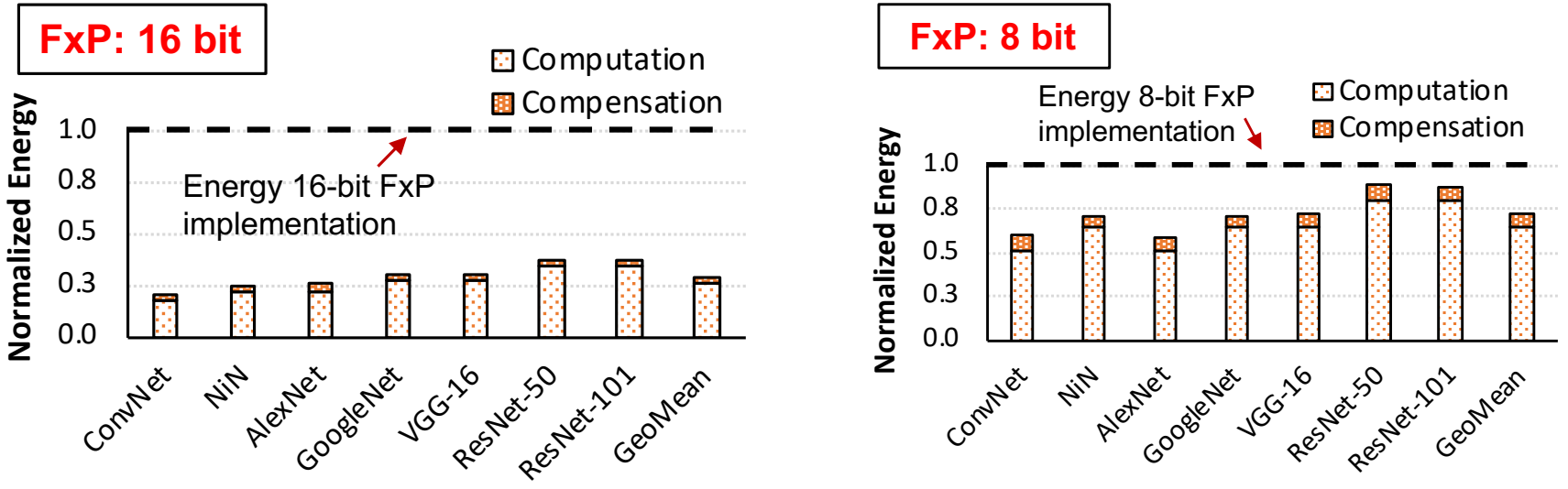
Postive compensation X_1, Y_1 rounded-down

No compensation X_2 close to QX_2

Negative compensation Y_2 rounded-up

Compensated-DNN: Energy Efficient Low-Precision Deep Neural Networks by Compensating Quantization Errors, DAC 2018

Results: Comparison to FxP16 and FxP8 Baselines



- **Constraint: Classification accuracy loss < 0.5% from FxP16 baseline**

- **Compensated-DNNs achieve:**
  - FxP16: ~60% reduction in computation bits → 2.65-4.88X energy savings
  - FxP8: ~25% reduction in computation bits → 1.13-1.7X energy savings

- **Average compensation frequency: ~30%, compensation overhead: ~5%**

58

Bi-Scaled DNN: Quantizing Long-tailed DNN Datastructures

- **Insight: Weights and activations of DNNs exhibit a long tail**
  - Many numbers of small magnitude with a few larges ones

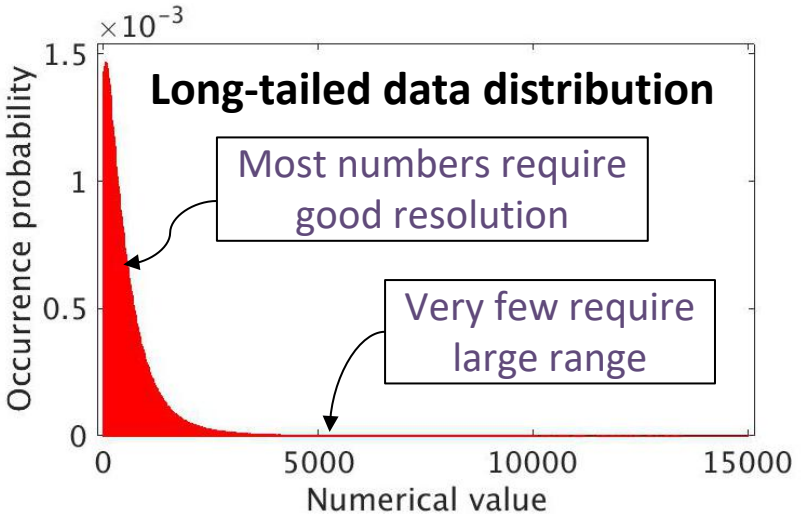
Tail length = $\lceil \log_2(P_{100}/P_{90}) \rceil$

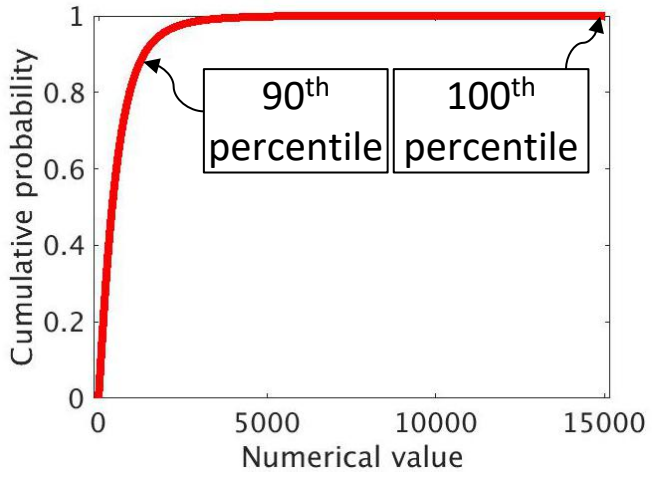
P_K: K^{th} Percentile

  - Tail length of 2-4 bits across DNNs

- **Conventional FxP is unfavorable for long-tailed data-structures**
  - Forces a choice between range or resolution

Network: VGG-16 **Layer**: CONV3-1 **DataType**: Activation (after ReLU)





Network: VGG-16

Layer Type	Tail length
Conv2-1 (Weights)	2 bits
Conv3-2 (Weights)	4 bits
FC1 (Weights)	2 bits
Conv1-2 (Activations)	3 bits
Conv4-1 (Activations)	2 bits
FC2 (Activations)	2 bits

Network: ResNet-50

Layer Type	Tail length
Conv1 (Weights)	2 bit
Conv2 (Weights)	3 bits
Conv3 (Weights)	2 bits
Conv1 (Activations)	3 bits
Conv2 (Activations)	4 bits
Conv3 (Activations)	3 bits

🙁 **Prefer range**
Captures large numbers, but majority suffer quantization errors

🙁 **Prefer resolution**:
Good for small numbers, but large numbers impact accuracy the most

BiScaled-DNN: Quantizing Long-tailed Datastructures with Two Scale Factors for Deep Neural Networks, DAC 2019

BiScaled FxP Quantization

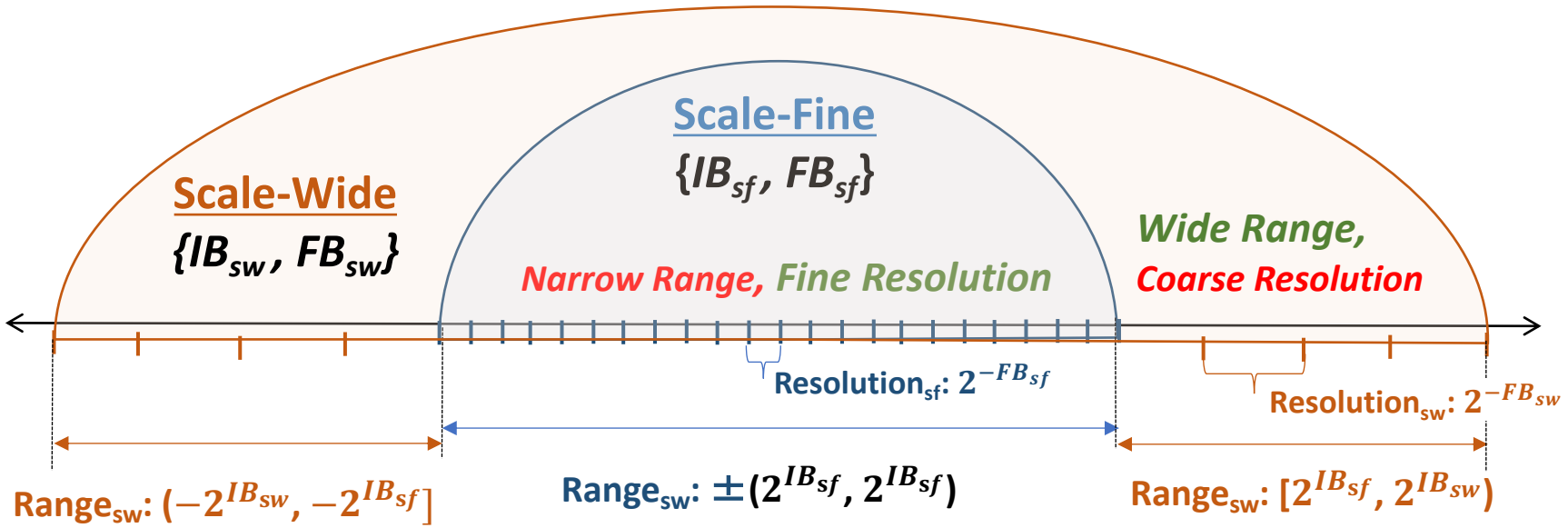
Key Idea: Use 2 scale factors to quantize long-tailed datastructures

- **Both small and large numbers use same bit-width, but different integer and fraction bits**
  - **Scale-Fine $\{IB_{sf}, FB_{sf}\}$ → Uses more fractional bits to minimize quantization error in small numbers**
  - **Scale-Wide $\{IB_{sw}, FB_{sw}\}$ → Uses more integer bits to avoid saturation of large numbers**

$$IB_{sf} + FB_{sf} = IB_{sw} + FB_{sw}$$

  - During quantization, based on the magnitude of the element, scale-fine or scale-wide is appropriately used

- **Reduced quantization error enables use of smaller bit-widths with BiScaled FxP!**



BiScaled Dot Product: Illustration

- **Dot-product occupies >99% of computations in DNNs**
- **With BiScaled, the multiplier output needs to be shifted before accumulation if either operand uses scale-wide**
  - Infrequently exercised as very few elements use scale-wide

Superior accuracy with little overhead!

BiScaled Dot-product (BX.BY)

BX.BY = $\sum (bx_i . by_i$ << $\text{Shift}_i)$

$\text{Shift}_i = (\text{isSwx}_i + \text{isSwy}_i) * (FB_{sf} - FB_{sw})$

isSwx_i = 1 (if bx_i is scale-wide)

isSwy_i = 1 (if by_i is scale-wide)

Full precision

32 Bit FP

i	x_i	y_i
1	5.1	14.9
2	44.9	2.9

X.Y = 206.20

Fixed Point

5 Bit FxP

i	qx_i	qy_i
1	4	16
2	44	4

Format [IB=6, FB=-2]

QX.QY = 240

Error = +16.4%

6 Bit FxP

i	qx_i	qy_i
1	6	14
2	44	2

Format [IB=6,FB=-1]

QX.QY =172

Error = -16.6%

BiScaled FxP

5 Bit BiScaled

i	bx_i	by_i
1	5	16
2	44	3

Scale-fine

Scale-wide

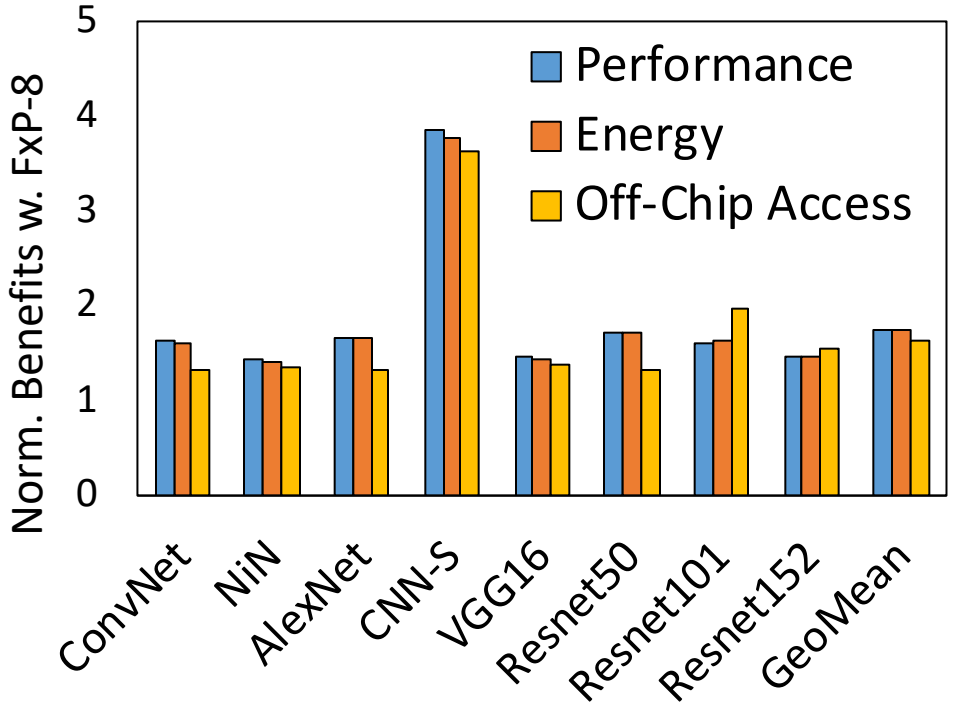
Format: **Scale-Fine** [IB_{sf} = 4, FB_{sf} = 0]
Scale-Wide [IB_{sw} = 6, FB_{sw}= -2]

BX.BY = 212

Error = +2.81%

BiScaled vs. FxP8: Performance and Accuracy Comparison

Network	FxP (8 bit)	BiScaled-FxP (6 bits)		
		Accuracy Increase	% Scale-wide elements	Eff. Bit width
ConvNet	75.27%	+0.05%	0.77%	6.11
AlexNet	54.44%	+0.46%	0.6%	6.10
VGG-16	66.43%	+0.13%	0.45%	6.08
CNN_S	57.52%	+0.63%	3.85%	6.32
NiN	53.15%	+0.50%	2.33%	6.22
ResNet50	69.91%	+0.55%	1.85%	6.18
ResNet101	72.51%	+0.03%	2.38%	6.22
ResNet152	73.31%	+0.10%	0.45%	6.08

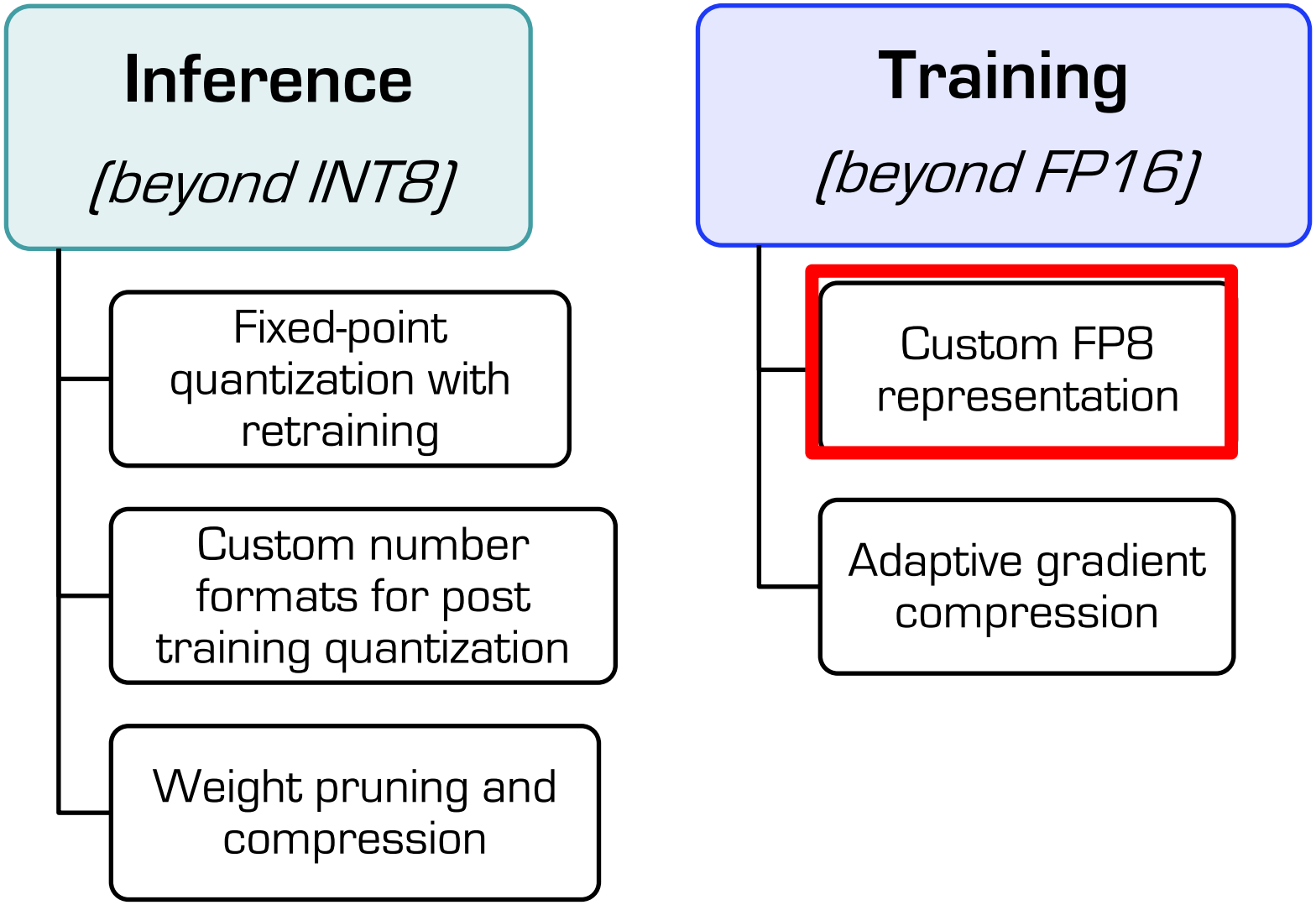


- **Without loss in accuracy, BiScaled quantization reduces computation bits from 8 to 6**
  - In many cases, BiScaled quantization achieves a small boost in classification accuracy
  - **1.1%** of elements were quantized with scale-wide
  - Effective bit-width is **6.16** including overheads for storing scale factors

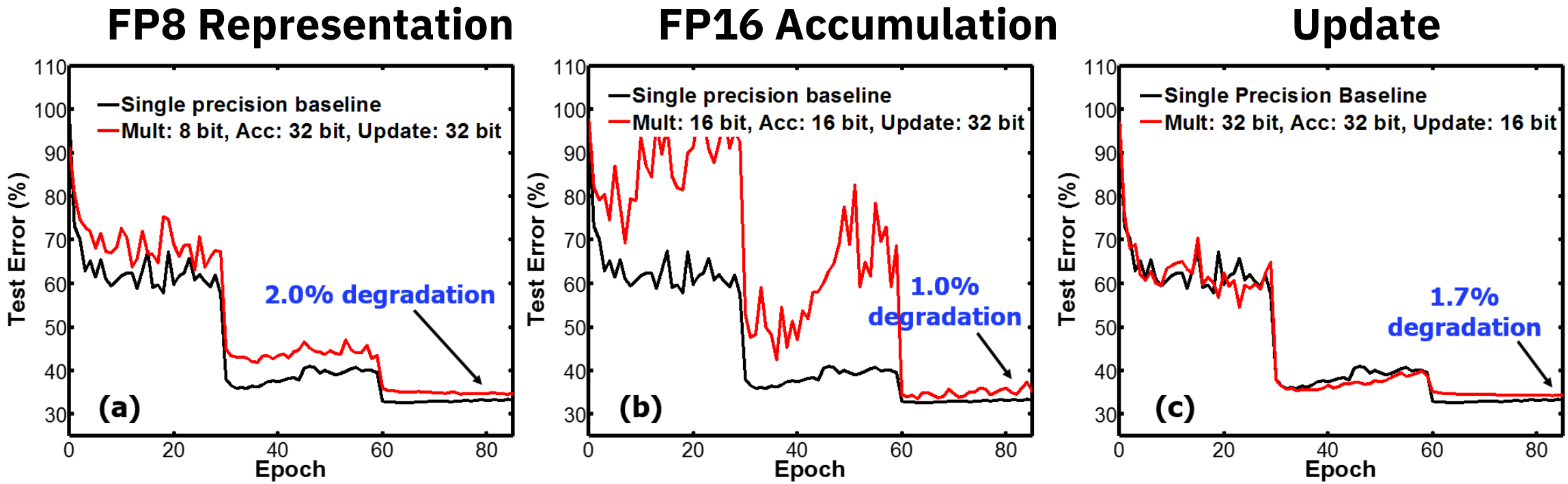
- **1.5X-3.8X improvement in performance/energy and 1.62X reduction in external memory accesses**

BiScaled-DNN: Quantizing Long-tailed Datastructures with Two Scale Factors for Deep Neural Networks, DAC 2019

Algorithmic Approximations for AI



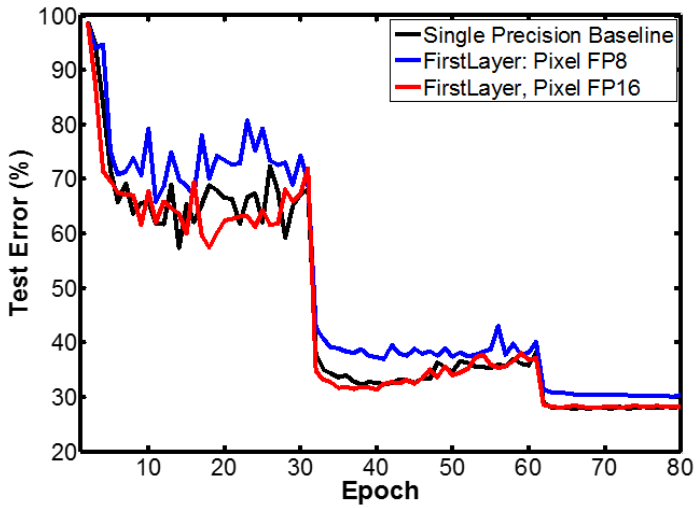
Training Challenges Beyond FP16

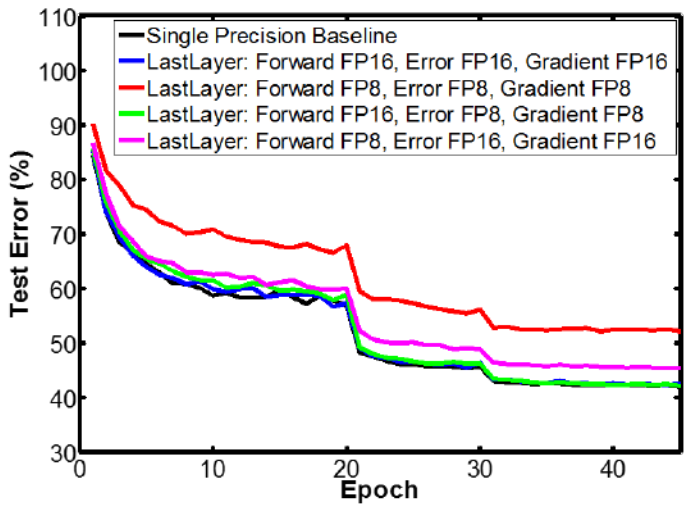


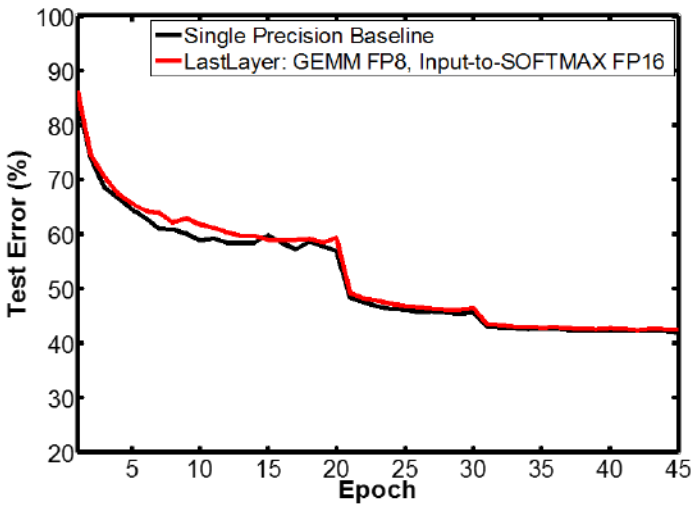
- State of the art Training systems use FP16 for data representations and FP32 for accumulations & weight updates.
- Challenge: To reduce these precisions down to FP8 for representation and FP16 for accumulation & weight updates.
- Goal: Increase training-performance by 4X over today's systems!

N. Wang et. al (presented at NeurIPS 2018)

FP8 Training : FP8 Data Representation







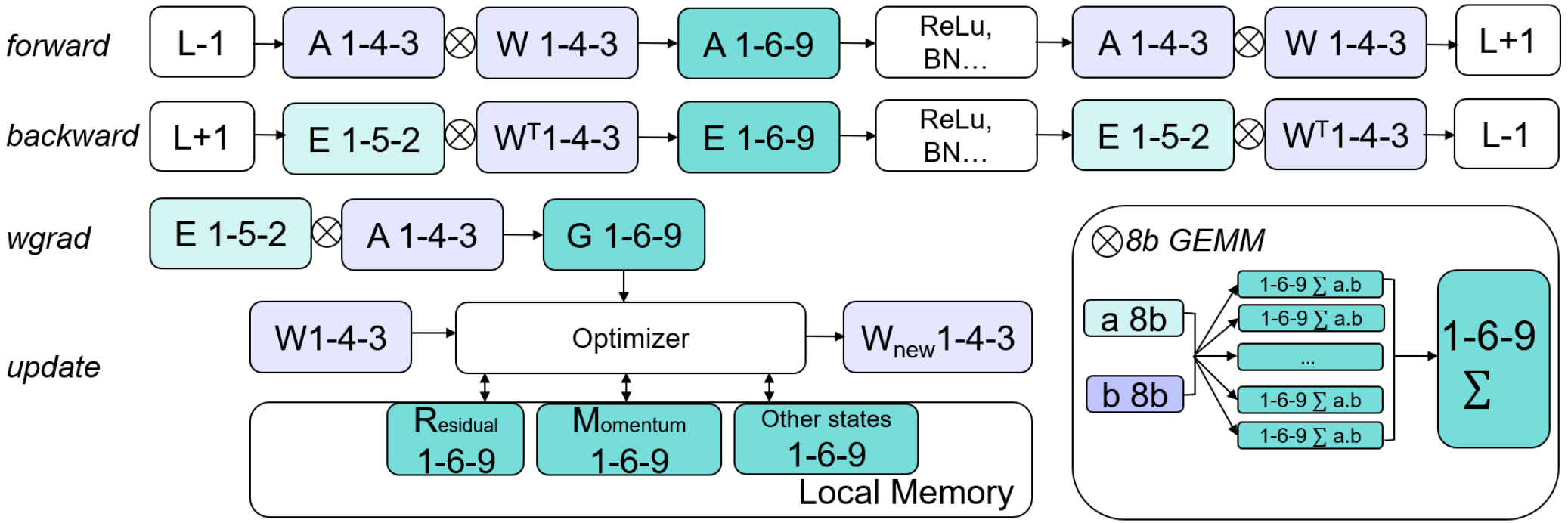
- First layer and last layers are very sensitive to lower-precision (FP8) – especially for image processing (keep them in FP16).

- Input to softmax is sensitive to low-precision : possible to keep last layer in FP8 but outputs need to be preserved in FP16.

N. Wang et. al (NeurIPS 2018), and X. Sun et. al (NeurIPS 2019)

FP8 Training : Hybrid FP8 Data Representation

- Multiple FP8 formats investigated
- FWD FP 1-4-3 and BWD FP 1-5-2 (sign-exponent-mantissa)
- FP 1-4-3 has an exponent bias of 4 to cover small numbers, [2^{-11}, 30]

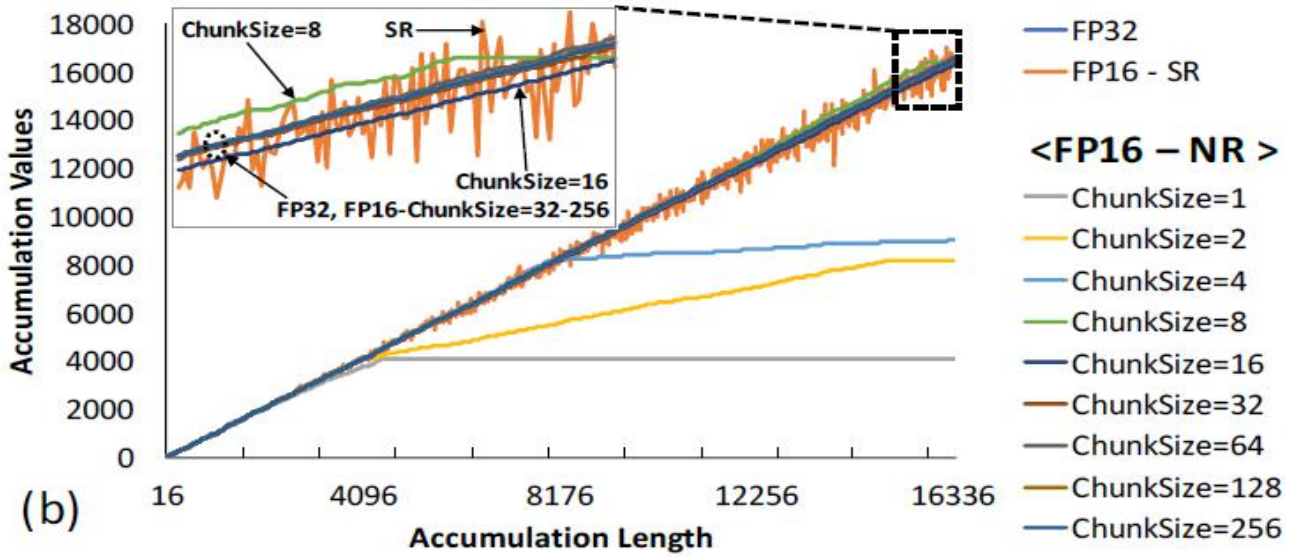


N. Wang et. al (NeurIPS 2018), and X. Sun et. al (NeurIPS 2019)

FP8 Training : FP16 Accumulations

```
Input: {x_n}_{n=1:N}, {y_n}_{n=1:N} (FP_mult),
Parameter: chunk size CL
Output: sum (FP_acc)
sum = 0.0; idx = 0; num_ch = N/CL
for n=1:num_ch {
   sum_ch = 0.0
   for i=1:CL {
      idx++
      tmp = x_idx · y_idx (in FP_mult)
      sum_ch += tmp (in FP_acc) }
   sum += sum_ch (in FP_acc) }
```

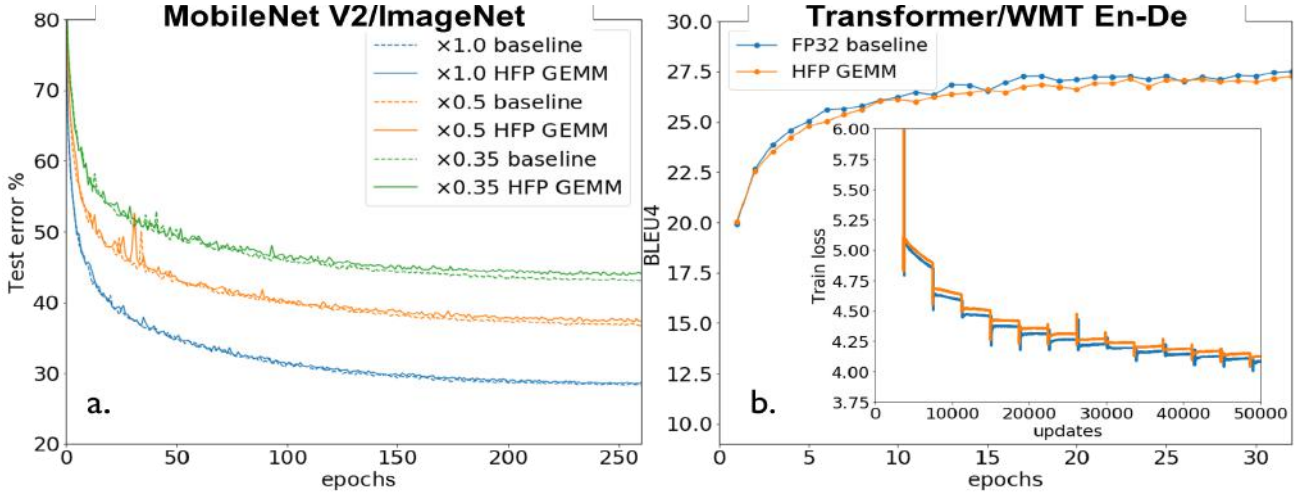
(a)



- **Identified low-precision swamping as the key challenge with FP16 accumulations (current state of the art is FP32 accumulations).**
  - Floating point addition involves right-shift of the smaller operands by the difference in exponents. In case of large-to-small number addition, small numbers maybe partially or completely truncated causing information loss.
- **Chunk-based accumulations (Hierarchical accumulations) fix this problem.**
  - Enables FP16 accumulation (may need hardware support)

N. Wang et. al (NeurIPS 2018), and X. Sun et. al (NeurIPS 2019)

FP8 Training Results

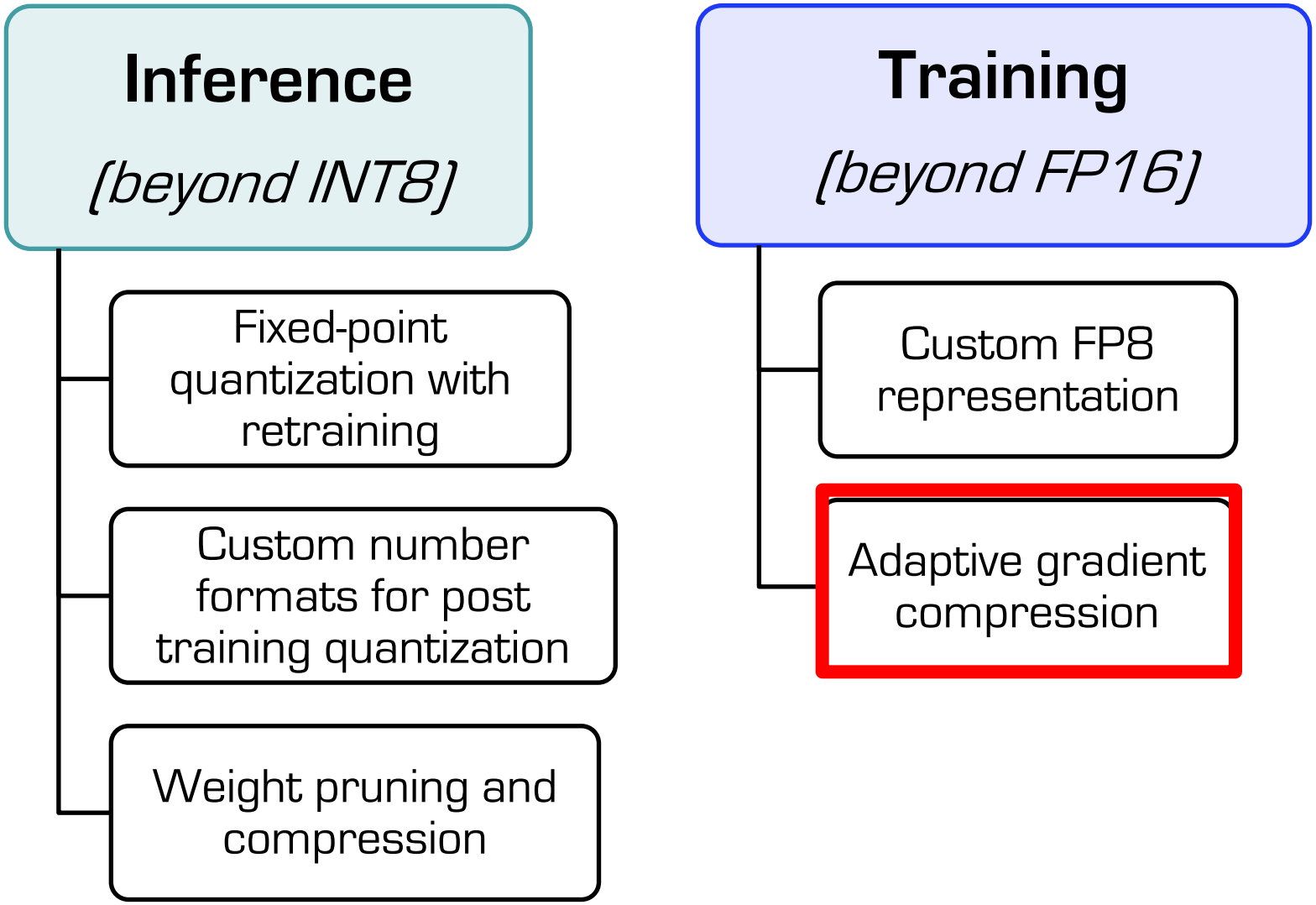


Model(Dataset) Accuracy or [other metrics]	Baseline(FP32)	HFP8 + Round-off update
AlexNet (ImageNet)	57.28	57.21
ResNet18 (ImageNet)	69.38	69.39
ResNet50 (ImageNet)	76.44	76.22
MobileNetV2 (ImageNet)	71.81	71.61
DenseNet121 (ImageNet)	74.76	74.65
2-LSTM (PennTreeBank)[Test ppl.]	83.66	83.86
Transformer-base (WMT14 En-De)[BLEU]	27.50	27.27
4-bidirectional-LSTM Speech (SWB300)[WER]	9.90	10.00
MaskRCNN(ResNet50) (COCO)[Box/Mask AP]	33.58/29.27	33.06/28.86
SSD-Lite(MobileNetV2) (VOC)[mAP]	68.79	68.72

- Combination of selective FP8 precisions, chunk-based accumulations and stochastic weight updates enable baseline accuracies with FP8 Training for a wide variety of neural networks.

N. Wang et. al (NeurIPS 2018), and X. Sun et. al (NeurIPS 2019)

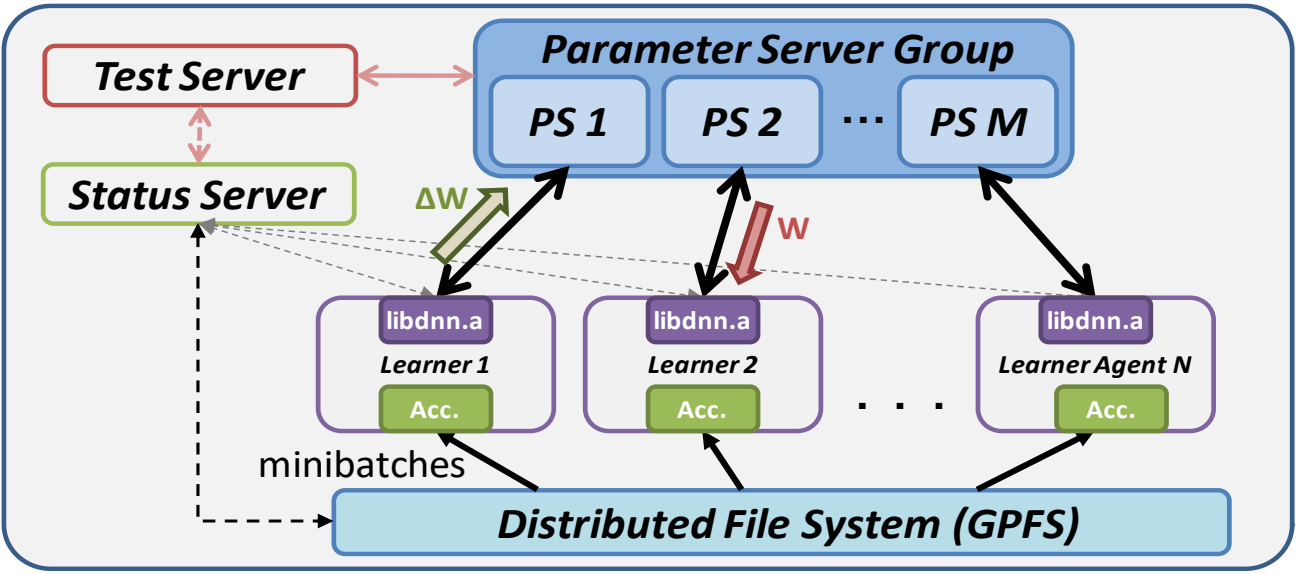
Algorithmic Approximations for AI



Gradient Compression for Distributed DNN Training

- **Data-parallel distributed training suffers large communication overhead**
  - Time for forward/backward computation reduced proportional to #Learners (=N)
  - But communication overhead for exchanging gradient of weight increases ➔ Bottleneck!

- **Gradient compression to overcome bottleneck**
  - Convert communication-bounded problem back into computation-bounded problem



Ex: **VGG-B data-parallel training** (over-simplified)
- Number of learners (N)
- VGG-B FLOPs (F) = 23 GFLOPs * 128 samples
- VGG-B model size (W) = 133 M * 4 Bytes
- PCIe-gen3 Bandwidth (B) = 12 GB/s * 80%
- Accelerator throughput (A) = 5 TFLOP/s
- Compute time: **Tcomp = (3*F / N) / A**
- Communication time: **Tcomm = (W / B) * N**

➔ **Tcomm > Tcomp when N >= 4 learners!**

70

AdaComp: Adaptive Residual Gradient Compression

- **Local selection**
  - Select locally best gradients (no sorting)
    → Selection spreads out across dimension
  - Better capture local activity of activation
    - Good for large mini-batch
- **Adaptive threshold**
  - Threshold based importance of gradient
    - Send RG[i] if RG[i] >= threshold = (RGmax – 2*dW[i])
  - When sending local-best residual gradient, also send "equally important" residual gradients
    - Robust to compress CONV/FC/LSTM

Algorithm 1 Computation Steps

```
learningNoUpdate();          ▷ Forward/Backward only
serializeGrad(); ▷ Collect grad of each layer as a vector
pack();                      ▷ AdaComp Compression for each layer
exchange();   ▷ Learner receives packed grads from others
unpack();          ▷ AdaComp Decompression for each layer
averageGradients();    ▷ Locally average to all learners
updateWeights();       ▷ Performed locally at each learner
```

Algorithm 2 Details of pack()

$G \leftarrow residue + dW$ ▷ dW is from `serializeGrad()`
$H \leftarrow G + dW$ ▷ H = Residue + 2*dW
Divide G into bins of size T;
for i $\leftarrow 1, length(G)/T$ **do** ▷ Over all bins
  Calculate $g_{max}(i)$; ▷ Identify the largest value in each bin
end for
for i $\leftarrow 1, length(G)/T$ **do** ▷ Over all bins
  for j $\leftarrow 1, T$ **do** ▷ Over all indices within each bin
    $index \leftarrow (i-1) * T + j$
    if H(index) $\geq g_{max}(i)$ **then** ▷ Compare to local max
      $Gq(index) \leftarrow Quantize(G(index))$
      send Gq(index) to other learners
      $residue(index) \leftarrow G(index) - Gq(index)$
    else
      $residue(index) \leftarrow G(index)$
    end if
  end for
end for

71

"AdaComp : Adaptive Residual Gradient Compression for Data-Parallel Distributed Training," Chen et al., AAAI 2018

Compression Performance

- **Experiments on various DNNs across variety of applications**
  - CNN: MNIST, CIFAR10, AlexNet, ResNet-18/50 for Image classification
  - LSTM: Char-RNN for character prediction (Shakespeare), FC: BN50 for Speech Recognition
  - Implemented using IBM Rudra DNN training platform
- **For all DNNs, achieved effective compression rate of 40x for Conv, 200x for FC**
  - Local threshold (L_T) = 50 for Conv, 500 for FC/LSTM, ternary quantization
  - i.e., 50 local best + equally good gradients are exchanged for each Conv layer
- **For all DNNs, no accuracy degradation**

Compression hyper parameters: convolution layer L_T: 50 and fully connected layer L_T: 500							
Model	MNIST-CNN	CIFAR10-CNN	AlexNet	ResNet18	ResNet50	BN50-DNN	LSTM
Dataset	MNIST	CIFAR10	ImageNet	ImageNet	ImageNet	BN50	Shakespeare
Mini-Batch size	100	128	256	256	256	256	10
Epoch	100	140	45	80	75	13	45
Baseline (top-1)	0.88%	18%	42.7%	32.41%	28.91%	59.8%	1.73%
Our method (top-1)	0.85% (8L)	18.4%(128L)	42.9%(8L)	32.87%(4L)	29.15%(4L)	59.8% (8L)	1.75% (8L)
Learner number	1,8	1,8,16,64,128	8	4	4	1,4,8	1,8

Summary

- **AI workloads have revolutionized application landscape**
  - Enabled new apps and services
  - Impose extreme computational challenges
- **Need to rethink computing stack to boost efficiency of AI workloads**
- **3 key approaches:**
  - **Hardware Accelerators**: Specialized computing systems for AI
  - **Custom Compilers**: Software stack to extract efficiency without sacrificing end-user productivity
  - **Approximate Computing**: Leverage resiliency to approximate computations to benefit efficiency
- **Many exciting opportunities for the future as workloads continue to grow and evolve!**