TU Graz
SCIENCE
PASSION
TECHNOLOGY

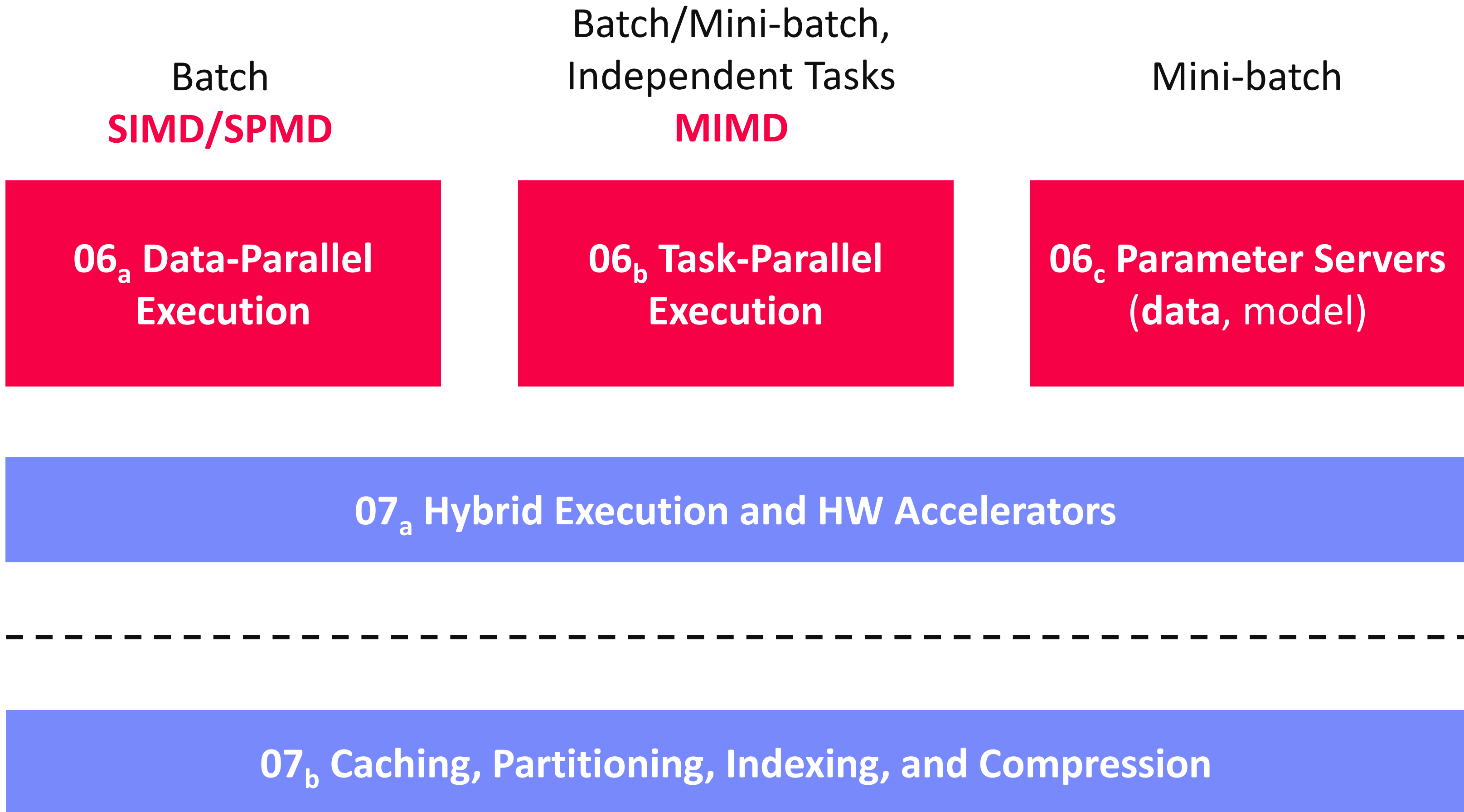
Architecture of ML Systems*
07 Hardware Accelerators and Data Access Methods

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMK endowed chair for Data Management

Last update: Aug 25, 2022

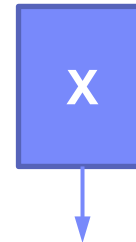
Categories of Execution Strategies



Agenda

- **GPUs in ML Systems**
- **FPGAs in ML Systems**
- **ASICs and other HW Accelerators**
- **Caching, Partitioning, and Indexing**
- **Lossy and Lossless Compression**

Iterative, I/O-bound ML algorithms ➔ **Data access crucial for performance**



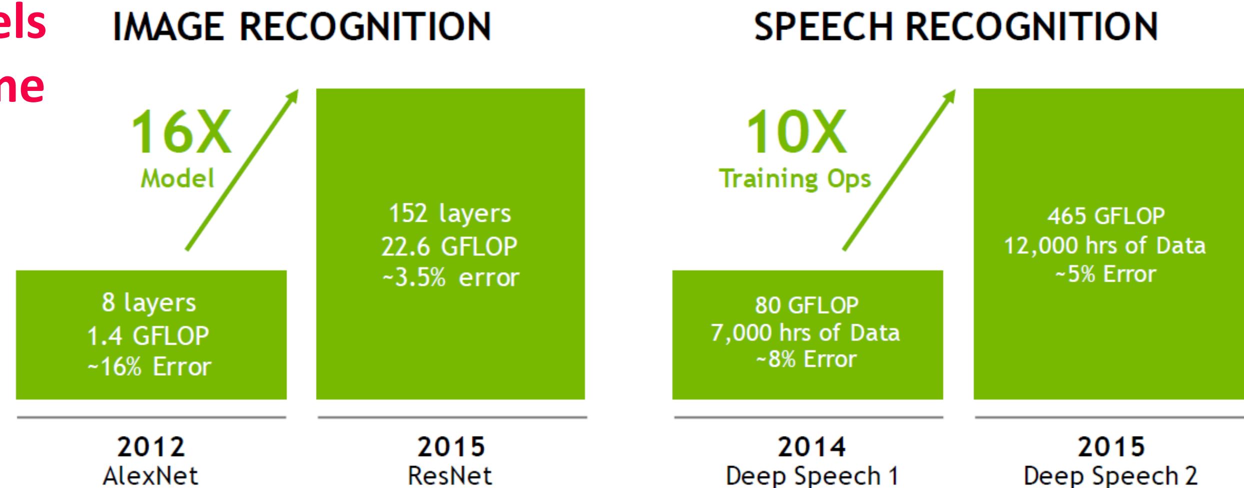

```
while(!converged) {
  … q = X %*% v …
}
```

Data **Weights**

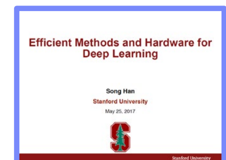
Graphics Processing Units (GPUs) in ML Systems

DNN Challenges

- **#1 Larger Models and Scoring Time**



- **#2 Training Time**
  - **ResNet18:** 10.76% error, 2.5 days training
  - **ResNet50:** 7.02% error, 5 days training
  - **ResNet101:** 6.21% error, 1 week training
  - **ResNet152:** 6.16% error, **1.5 weeks training**
- **#3 Energy Efficiency**



[Song Han: Efficient Methods and Hardware for Deep Learning, Stanford cs231n, 2017]

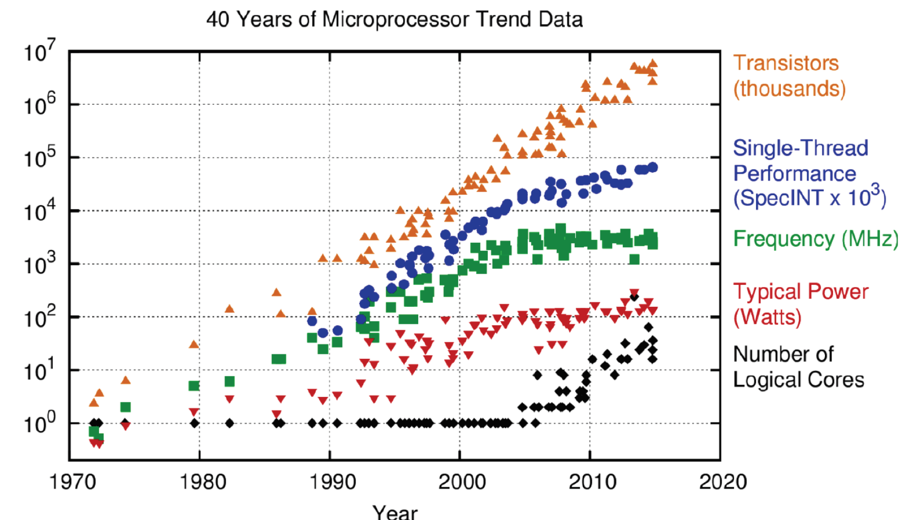
HW Challenges

[S. Markidis, E. Laure, N. Jansson, S. Rivas-Gomez and S. W. D. Chien: Moore's Law and Dennard Scaling]

- **#1 End of Dennard Scaling** (~2005)
  - **Law:** power stays proportional to the area of the transistor
  - Ignored leakage current / threshold voltage
    → **increasing power density S^2** (power wall, heat) → stagnating frequency

$P = \alpha CFV^2$ (**power density 1**)
(P .. Power, C .. Capacitance, F .. Frequency, V .. Voltage)

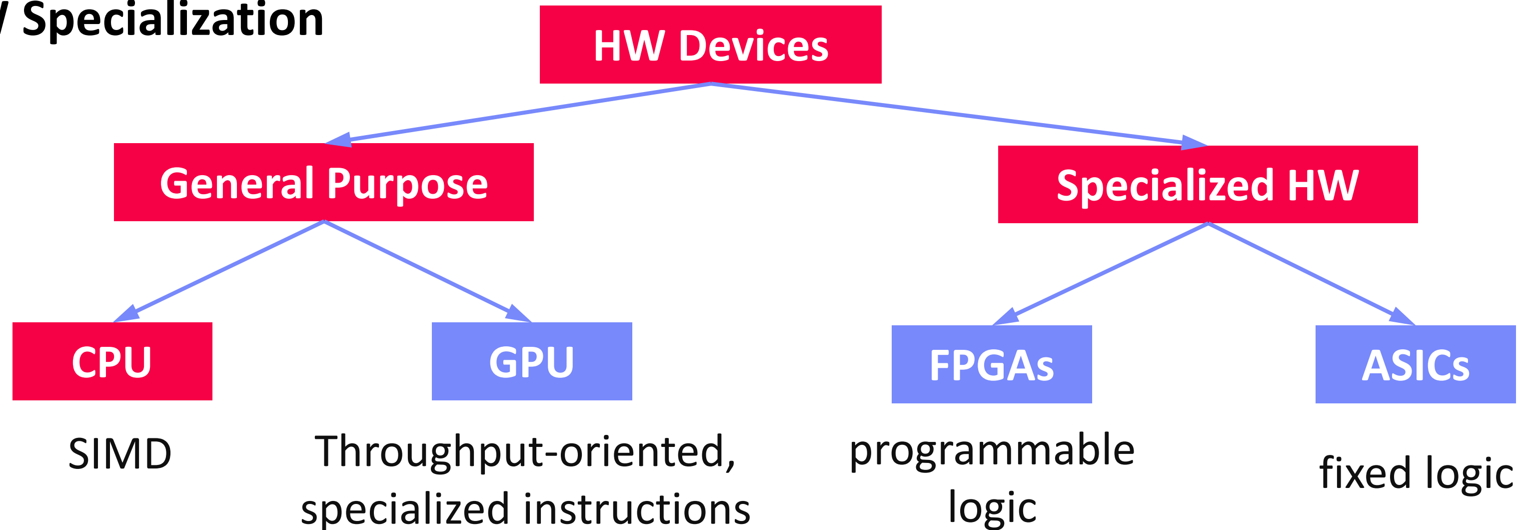
- **#2 End of Moore's Law** (~2010-20)
  - **Law:** #transistors/performance/ CPU frequency doubles every 18/24 months
  - Original: # transistors per chip doubles every two years **at constant costs**
  - Now increasing costs (10/7/5nm)



➔ **Consequences: Dark Silicon and Specialization**

Towards Specialized Hardware

- **HW Specialization**



- **Additional Specialization**
  - **Data Transfer & Types:** e.g., low-precision, quantization
  - **Sparsity Exploitation:** e.g., sparsification, exploit across ops, defer weight decompression just before instruction execution
  - **Near-Data Processing:** e.g., operations in main memory, storage class memory (SCM), secondary storage (e.g., SSDs), and tertiary storage (e.g., tapes)

Caching, Indexing and Compression

NVIDIA Volta V100 – Specifications

- **Tesla V100 NVLink**
  - FP64: **7.8 TFLOPs**, FP32: **15.7 TFLOPs**
  - DL FP16: **125 TFLOPs**
  - NVLink: 300GB/s
  - Device HBM: 32 GB (**900 GB/s**)
  - Power: 300 W

- **Tesla V100 PCIe**
  - FP64: 7 TFLOPs, FP32: 14 TFLOPs
  - DL FP16: 112 TFLOPs
  - PCIe: 32 GB/s
  - Device HBM: 16 GB (900 GB/s)
  - Power: **250 W**

[Credit: https://nvidia.com/de-de/data-center/tesla-v100/]

NVIDIA Volta V100 – Architecture

[NVIDIA Tesla V100 GPU Architecture, Whitepaper, **Aug 2017**]

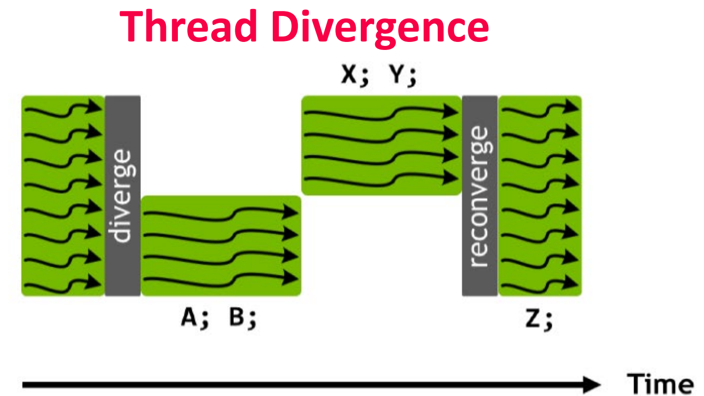
- **6 GPU Processing Clusters (GPCs)**
  - 7 Texture Processing Clusters (TPC)
  - 14 Streaming Multiprocessors (SM)

- **SM Architecture**
  - FP64 cores: 32
  - FP32 cores: 64
  - INT32 cores: 64
  - "Tensor cores": 8
  - Max warps /SM: 64
  - Threads/warp: 32

Single Instruction Multiple Threads (SIMT)

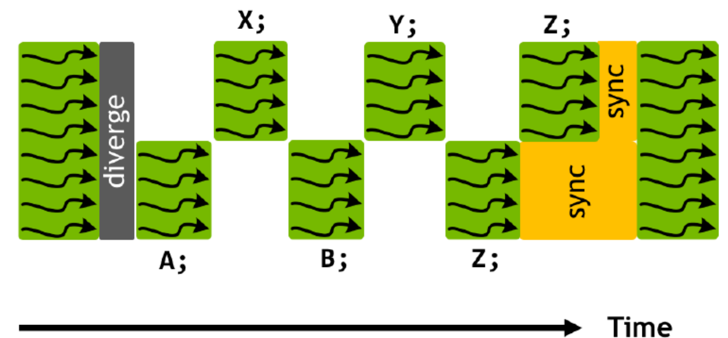
- **32 Threads grouped to warps and execute in SIMT model**
- **Pascal P100 Execution Model**
  - Warps use a single program counter + active mask

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



- **Volta V100 Execution Model**
  - Independent thread scheduling
  - Per-thread program counters and call stacks

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



  - New **`__syncwarp()`** primitive (if needed) + **convergence optimizer**

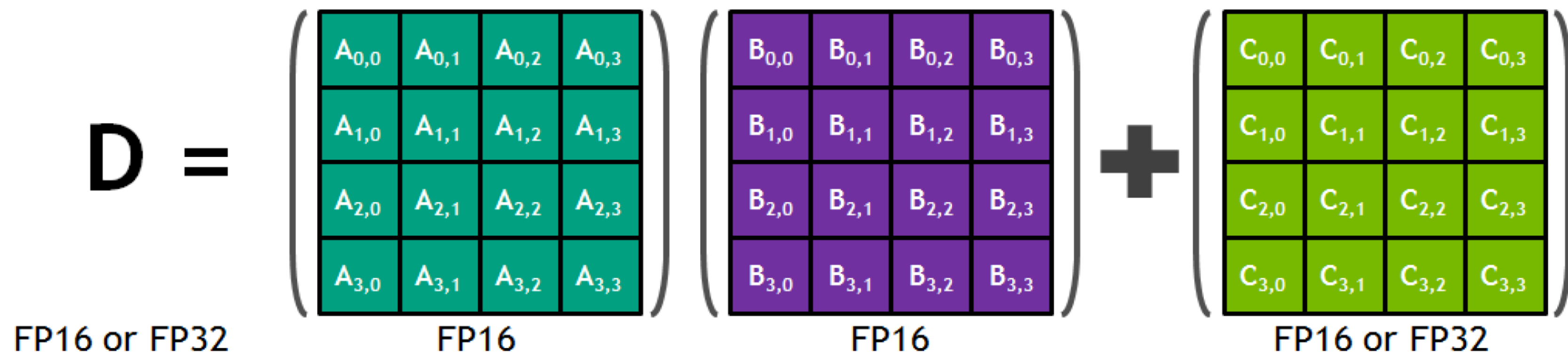
NVIDIA Volta V100 – Tensor Cores

- **"Tensor Core"**
  - Specialized instruction for **4x4 by 4x4 fused matrix multiply**
  - Two FP16 inputs and FP32 accumulator
  - Exposed as warp-level matrix operations w/ special load, mm, acc, and store

[Bill Dally: Hardware for Deep Learning. **SysML 2018**]

D = A %*% B + C

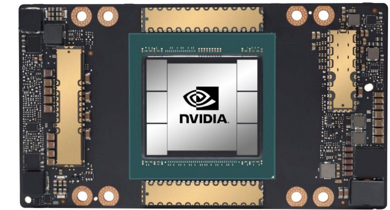
64 FMA operations



NVIDIA Ampere A100

[NVIDIA A100 Tensor Core GPU Architecture - UNPRECEDENTED ACCELERATION AT EVERY SCALE, Whitepaper, **Aug 2020**]

- **Specification**
  - 7nm, 8 GPC x 8 TPC * 2 SM = 128 SMs, 40GB HBM
  - FP64: 9.7 TFLOPs / FP64 TensorCore: 19.5 TFLOPs
  - FP32 19.5 TFLOPs, FP16: 78 TFLOPs, BF16: 39 TFLOPs
  - TF32 TensorCore 156 TFLOPs / 312 TFLOPs (sparse)
  - FP16 TensorCore 312 TFLOPs / 624 TFLOPs (sparse), INT8, INT4



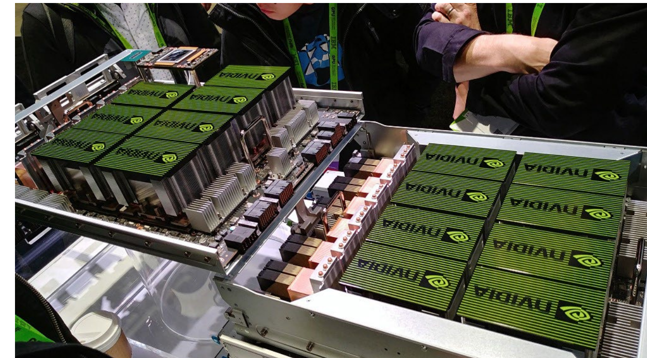
- **New Features**
  - New generation of “TensorCores” (FP64, **new data types:** TF32, BF16)
  - Fine-grained **sparsity exploitation**
  - Multi-instance **GPU (MIG) virtualization**: up to 7 virtual GPU instances
  - Link technologies: **NVLink 3** (25GB/s bidirectional) x 12 links = 600GB/s
  - **Submission of task graphs** (launch a workflow of kernels)

GPUs for DNN Training

- **GPUs for DNN Training** (2009)
  - Deep belief networks
  - Sparse coding

[Rajat Raina, Anand Madhavan, Andrew Y. Ng: Large-scale deep unsupervised learning using graphics processors. **ICML 2009**]

- **Multi-GPU Learning** (Now)
  - Exploit multiple GPUs with a mix of **data- and model-parallel parameter servers**
  - Dedicated ML systems for multi-GPU learning
  - Dedicated HW: e.g., NVIDIA DGX-1 (8xP100), **NVIDIA DGX-2 (16xV100, NVSwitch)**, NVIDIA DGX A100 (8x A100, NVSwitch, Mellanox)



- **New GPU Link Technologies (NVSwitch + NVLink 1.0 / 2.0 / 3.0)**

- **DNN Framework support**
  - All specialized DNN frameworks have very good support for GPU training
  - Most of them also support multi-GPU training

DNN Benchmarks

[**MLPerf** v0.6: https://mlperf.org/training-results-0-6/,
MLPerf v0.7: https://mlperf.org/training-results-0-7]

Closed Division Times

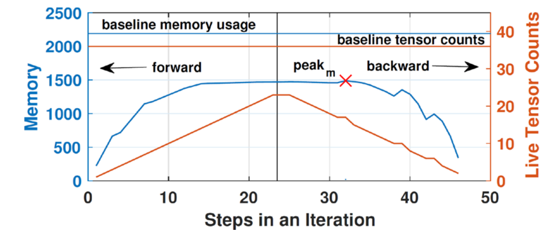
V0.6

#	Submitter	System	Processor	#	Accelerator	#	Software	Benchmark results (minutes)							Details	Code	Notes
								Image classification	Object detection, light-weight	Object detection, heavy-wt.	Translation, recurrent	Translation, non-recur.	Recommendation	Reinforcement Learning			
								ImageNet	COCO	COCO	WMT E-G	WMT E-G	MovieLens-20M	Go			
								ResNet-50 v1.5	SSD w/ ResNet-34	Mask-R-CNN	NMT	Transformer	NCF	Mini Go			
Available in cloud																	
0.6-1	Google	TPUv3.32			TPUv3	16	TensorFlow, TPU 1.14.1.dev	42.19	12.61	107.03	12.25	10.20	[1]		details	code	none
0.6-2	Google	TPUv3.128			TPUv3	64	TensorFlow, TPU 1.14.1.dev	11.22	3.89	57.46	4.62	3.85	[1]		details	code	none
0.6-3	Google	TPUv3.256			TPUv3	128	TensorFlow, TPU 1.14.1.dev	6.86	2.76	35.60	3.53	2.81	[1]		details	code	none
0.6-4	Google	TPUv3.512			TPUv3	256	TensorFlow, TPU 1.14.1.dev	3.85	1.79		2.51	1.58	[1]		details	code	none
0.6-5	Google	TPUv3.1024			TPUv3	512	TensorFlow, TPU 1.14.1.dev	2.27	1.34		2.11	1.05	[1]		details	code	none
0.6-6	Google	TPUv3.2048			TPUv3	1024	TensorFlow, TPU 1.14.1.dev	1.28	1.21			0.85	[1]		details	code	none
Available on-premise																	
0.6-7	Intel	32x 2S CLX 8260L	CLX 8260L	64			TensorFlow						[1]	14.43	details	code	none
0.6-8	NVIDIA	DGX-1			Tesla V100	8	MXNet, NGC19.05	115.22					[1]		details	code	none
0.6-9	NVIDIA	DGX-1			Tesla V100	8	PyTorch, NGC19.05		22.36	207.48	20.55	20.34	[1]		details	code	none
0.6-10	NVIDIA	DGX-1			Tesla V100	8	TensorFlow, NGC19.05						[1]	27.39	details	code	none
0.6-11	NVIDIA	3x DGX-1			Tesla V100	24	TensorFlow, NGC19.05						[1]	13.57	details	code	none
0.6-12	NVIDIA	24x DGX-1			Tesla V100	192	PyTorch, NGC19.05			22.03			[1]		details	code	none
0.6-13	NVIDIA	30x DGX-1			Tesla V100	240	PyTorch, NGC19.05		2.67				[1]		details	code	none
0.6-14	NVIDIA	48x DGX-1			Tesla V100	384	PyTorch, NGC19.05				1.99		[1]		details	code	none
0.6-15	NVIDIA	60x DGX-1			Tesla V100	480	PyTorch, NGC19.05					2.05	[1]		details	code	none
0.6-16	NVIDIA	130x DGX-1			Tesla V100	1040	MXNet, NGC19.05	1.69					[1]		details	code	none
0.6-17	NVIDIA	DGX-2			Tesla V100	16	MXNet, NGC19.05	57.87									
0.6-18	NVIDIA	DGX-2			Tesla V100	16	PyTorch, NGC19.05		12.21	101.00	10.94	11.04					
0.6-19	NVIDIA	DGX-2H			Tesla V100	16	MXNet, NGC19.05	52.74									
0.6-20	NVIDIA	DGX-2H			Tesla V100	16	PyTorch, NGC19.05		11.41	95.20	9.87	9.80					
0.6-21	NVIDIA	4x DGX-2H			Tesla V100	64	PyTorch, NGC19.05		4.78	32.72							
0.6-22	NVIDIA	10x DGX-2H			Tesla V100	160	PyTorch, NGC19.05					2.41					
0.6-23	NVIDIA	12x DGX-2H			Tesla V100	192	PyTorch, NGC19.05			18.47							
0.6-24	NVIDIA	15x DGX-2H			Tesla V100	240	PyTorch, NGC19.05		2.56								
0.6-25	NVIDIA	16x DGX-2H			Tesla V100	256	PyTorch, NGC19.05				2.12						
0.6-26	NVIDIA	24x DGX-2H			Tesla V100	384	PyTorch, NGC19.05				1.80						
0.6-27	NVIDIA	30x DGX-2H, 8 chips each			Tesla V100	240	PyTorch, NGC19.05		2.23								
0.6-28	NVIDIA	30x DGX-2H			Tesla V100	480	PyTorch, NGC19.05					1.59					
0.6-29	NVIDIA	32x DGX-2H			Tesla V100	512	MXNet, NGC19.05	2.59									
0.6-30	NVIDIA	96x DGX-2H			Tesla V100	1536	MXNet, NGC19.05	1.33									

96 x DGX-2H = 96 * 16 = 1536 V100 GPUs
➔ ~ 96 * $400K = **$35M – $40M**

[https://www.forbes.com/sites/tiriasresearch/2019/06/19/nvidia-offers-a-turnkey-supercomputer-the-dgx-superpod/#693400f43ee5]

Handling Memory Constraints



[Linnan Wang et al: Superneurons: dynamic GPU memory management for training deep neural networks. **PPOPP 2018**]

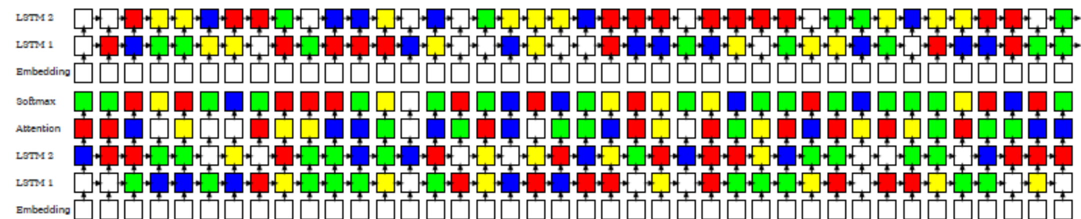
- **Problem: Limited Device Memory**
- **#1 Live Variable Analysis**
  - Remove intermediates ASAP
  - **Examples:** SystemML, TensorFlow, MXNet, Superneurons, MONeT
- **#2 GPU-CPU Eviction**
  - Evict variables from GPU to CPU memory under memory pressure
  - **Examples:** SystemML, Superneurons, GeePS, (TensorFlow)
- **#3 Recomputation**
  - Recompute inexpensive operations (e.g., activations of forward pass)
  - **Examples:** MXNet, Superneurons, MONet
- **#4 Reuse Allocations**
  - Reuse allocated matrices and tensors via free lists, but **fragmentation**
  - **Examples:** SystemML, Superneurons, MONet
- **#5 Physical Operator Selection**
  - Different tradeoffs of performance and size of intermediates (MONet)

Hybrid CPU/GPU Execution

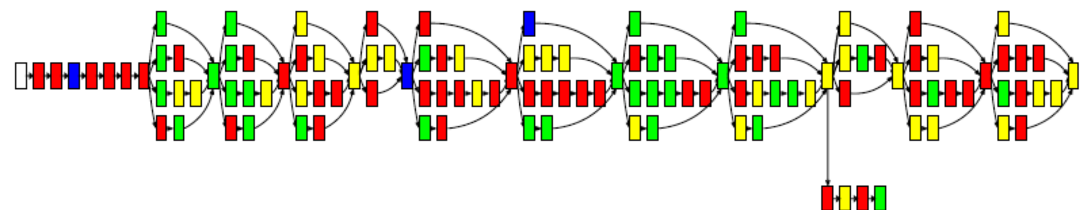
- **Manual Placement**
  - Most DNN frameworks allow manual placement of variables and operations on individual CPU/GPU devices
  - **Heuristics and intuition of human experts**
- **Automatic Placement**
  - Sequence-to-sequence model to predict which operations should run on which device
  - Examples:

[Azalia Mirhoseini et al: Device Placement Optimization with Reinforcement Learning. **ICML 2017**]

Neural MT graph



Inception V3



Sparsity in DNN

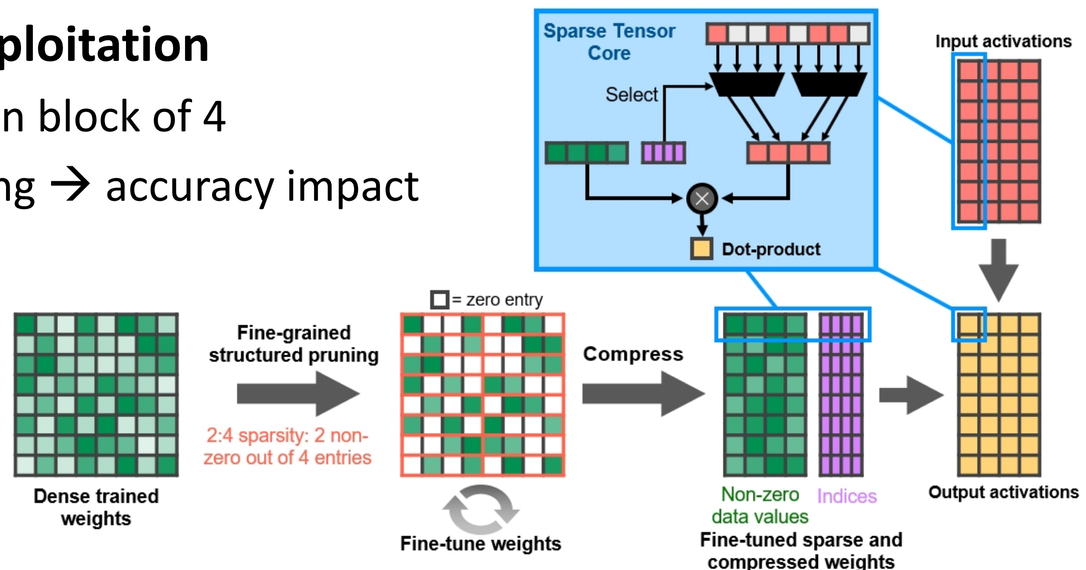
- **State-of-the-art**
  - **Very limited support of sparse tensors** in TensorFlow, PyTorch, etc
  - GPU operations for linear algebra (**cuSparse**), early support in ASICs
  - Problem: **Irregular structures of sparse matrices/tensors**

PYTORCH

- **Common Techniques**
  - #1: **Blocking/clustering** of rows/columns by number of non-zeros
  - #2: **Padding rows/columns** to common number of non-zeros

- **Example A100 Sparsity Exploitation**
  - Constraint: 2 non-zeros in block of 4
  - Structured valued pruning → accuracy impact
  - Regular access pattern

[NVIDIA A100 Tensor Core GPU Architecture, Whitepaper, **Aug 2020**]

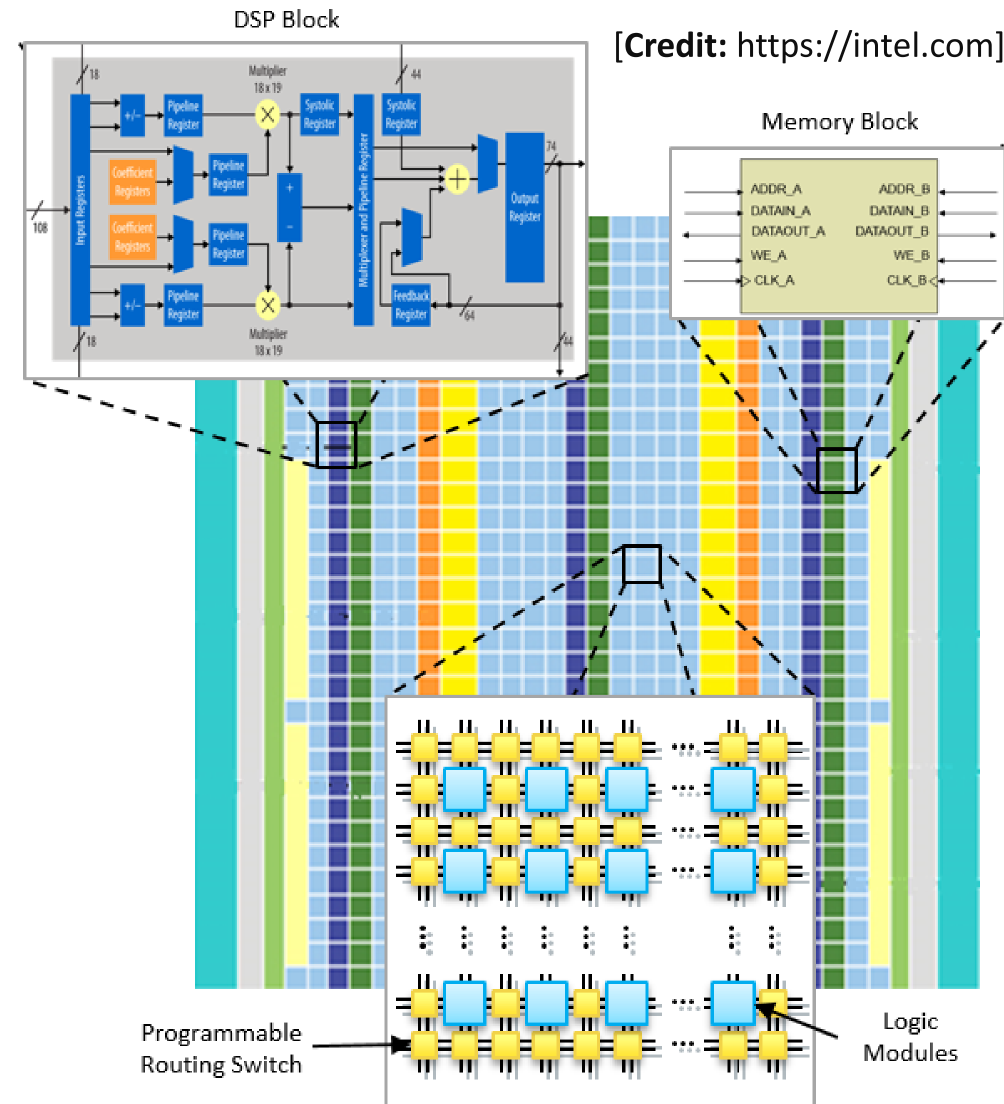


Field-Programmable Gate Arrays (FPGAs) in ML Systems

FPGA Overview

- **FPGA Definition**
  - Integrated circuit that allows **configuring custom hardware designs**
  - Reconfiguration in <1s
  - HW description language: e.g.., VHDL, Verilog

- **FPGA Components**
  - **#1 lookup table** (LUT) as logic gates
  - **#2 flip-flops** (registers)
  - **#3 interconnect network**
  - Additional memory and DSPs



[**Credit:** https://intel.com]

→ **Specialized neural networks and kernel implementations**

Example FPGA Characteristics

- **Intel (Altera) Stratix 10 SoC FPGA**
  - 64bit quad-core ARM
  - 10 TFLOPs FP32
  - 80GFLOPs/W
  - Other configurations w/ HBM2

- **Xilinx Virtex UltraSCALE+**
  - DSP: 21.2 TMACs
  - 64MB on-chip memory
  - 8GB HBM2 w/ 460GB/s

FPGAs in Microsoft's Data Centers

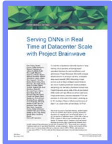
- **Microsoft Catapult**
  - Dual-socket Xeon w/ PCIe-attached FPGA
  - Pre-filtering neural networks, compression, and other workloads

[Adrian M. Caulfield et al.: A cloud-scale acceleration architecture. **MICRO 2016**]

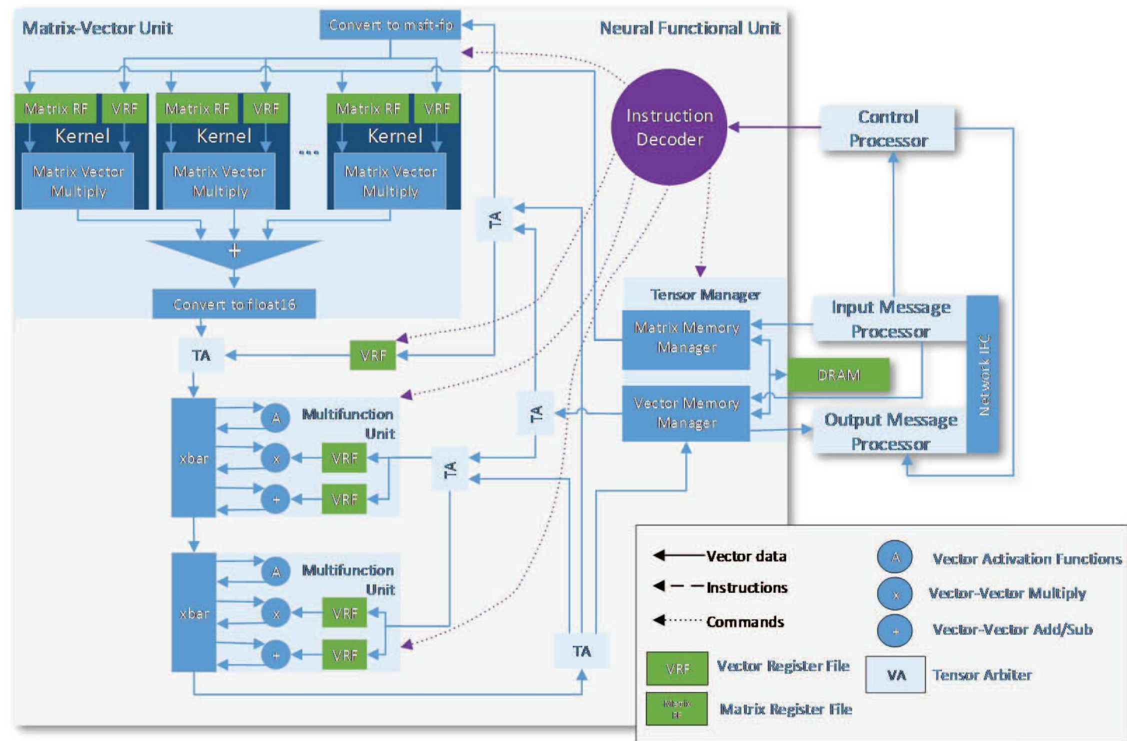
L2
L1
L1
TOR
TOR
TOR
TOR
Network switch (top of rack, cluster)
FPGA – switch link
FPGA acceleration board
NIC – FPGA link
2-socket CPU server
Datacenter hw acceleration plane
Deep neural networks
Expensive compression
Web search ranking
Bioinformatics
Web search ranking
Traditional sw (CPU) server plane
2-socket server blade
DRAM
DRAM
CPU
QPI
CPU
Gen3 x8
Gen3 2x8
NIC
Accelerator card
FPGA
DRAM
QSFP
QSFP
QSFP
40Gb/s
40Gb/s
TOR

FPGAs in Microsoft's Data Centers, cont.

[Eric S. Chung et al: Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. **IEEE Micro 2018**]



- **Microsoft Brainwave**
  - ML serving w/ low latency (e.g., Bing)
  - Intel Stratix 10 FPGA
  - Distributed **model parallelism**, precision-adaptable
  - Peak 39.5 TFLOPs



- **Brainwave NPU**
  - Neural processing unit
  - Dense matrix-vector multiplication

Example DM Cluster Node

2x Intel Xeon Gold 6238 (112 vcores, 7.7 TFLOP/s), 768 GB DDR4 RAM, 12x 2TB SSDs, NVIDIA **T4 GPU** (8.1 TFLOP/s, 16 GB), and Intel FPGA PAC D5005 (w/ Stratix **10SX FPGA**, 32 GB)

12+2 x SSD: 25TB

2 x Intel Xeon Gold 6238R

768 GB RAM

Intel PAC D5005 Stratix 10 FPGA

Broadcom 2x10G RDMA Ethernet

Adaptec 12G SAS

Nvidia Tesla T4

Application-Specific Integrated Circuit (ASICs) and other HW Accelerators

Overview ASICs

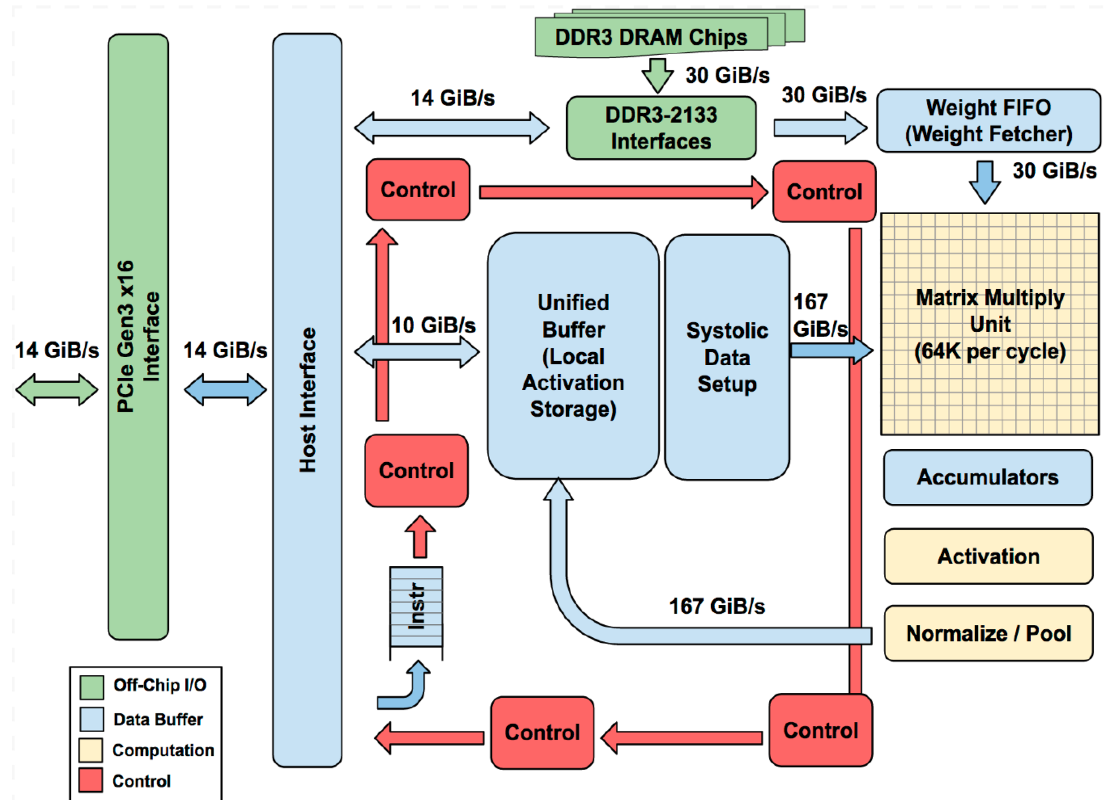
- **Motivation**
  - Additional improvements of performance, power/energy
  - → **Additional specialization via custom hardware**
- **#1 General ASIC DL Accelerators**
  - HW support for matrix multiply, convolution and activation functions
  - Examples: **Google TPU**, **NVIDIA DLA** (in NVIDIA Xavier SoC), **Intel Nervana NNP**
- **#2 Specialized ASIC Accelerators**
  - Custom instructions for specific domains such as computer vision
  - Example: (Cadence) **Tensilica Vision processor** (image processing)
- **#3 Other Accelerators/Technologies** (some skepticism)
  - a) **Neuromorphic computing / spiking neural networks** (e.g., SyNAPSE → IBM TrueNorth, HP memristor for computation storage)
  - b) **Analog computing** (especially for ultra-low precision/quantization)

Tensor Processing Unit (TPU v1)

- **Motivation**
  - Cost-effective ML scoring (no training)
  - Latency- and throughput-oriented
  - **Improve cost-performance over GPUs by 10x**

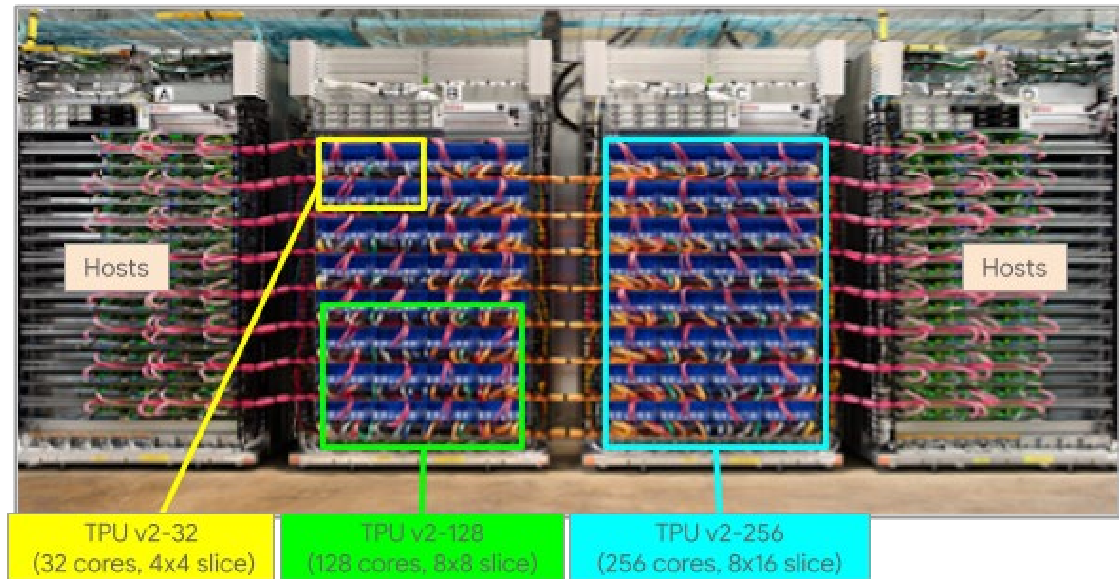
[Norman P. Jouppi et al: In-Datacenter Performance Analysis of a Tensor Processing Unit. **ISCA 2017**]

- **Architecture**
  - **256x256 8bit matrix multiply unit** (systolic array → **pipelining**)
  - **64K MAC per cycle** (92 TOPs at 8 bit)
  - 50% if one input 16bit
  - 25% if all inputs 16 bit



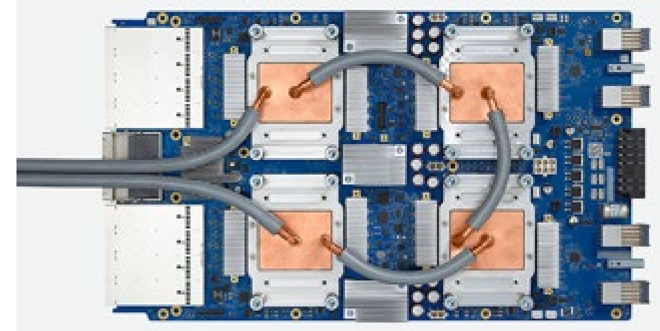
Tensor Processing Unit (TPU v2)

- **Motivation**
  - Cost effective ML training (not scoring) because edge device w/ custom inference but training in data centers
  - Unveiled at Google I/O 2017
  - Board w/ 4 TPU chips
  - Pod w/ 64 boards and custom high-speed network
  - Shelf w/ 2 boards or 1 processor

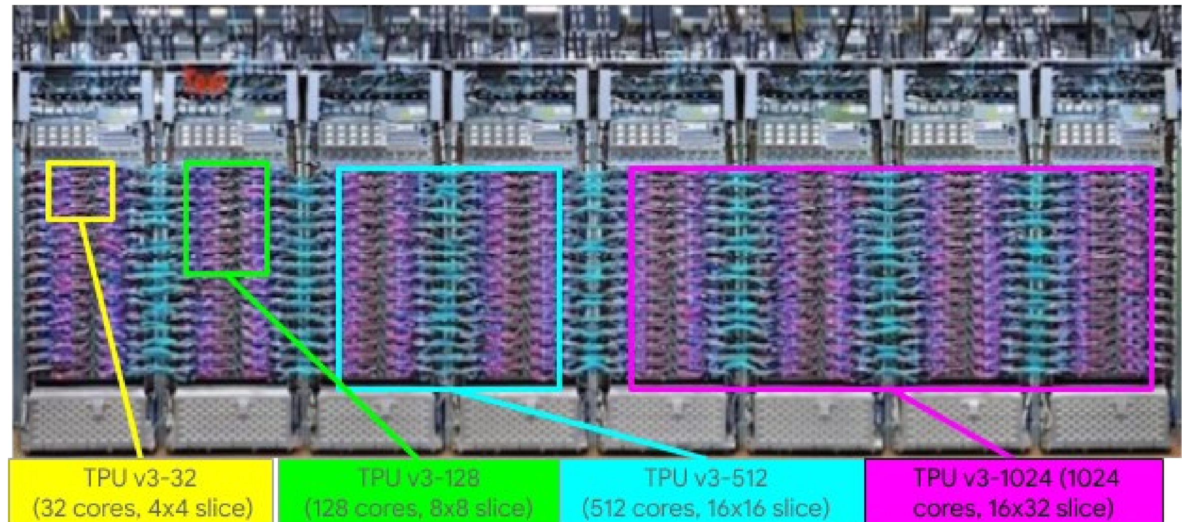


- **Cloud Offering (beta)**
  - Min 32 cores
  - Max 512 cores

Tensor Processing Unit (TPU v3)



- **Motivation**
  - Competitive cost-performance compared to state-of-the-art GPUs
  - Unveiled at **Google I/O 2018**
  - Added **liquid cooling**
  - Twice as many racks per pod, twice as many TPUs per rack
  - ➔ TPUv3 promoted as **8x higher performance** than TPUv2
- **Cloud Offering (beta)**
  - Min 32 cores
  - Max 2048 cores (~100PFLOPs)

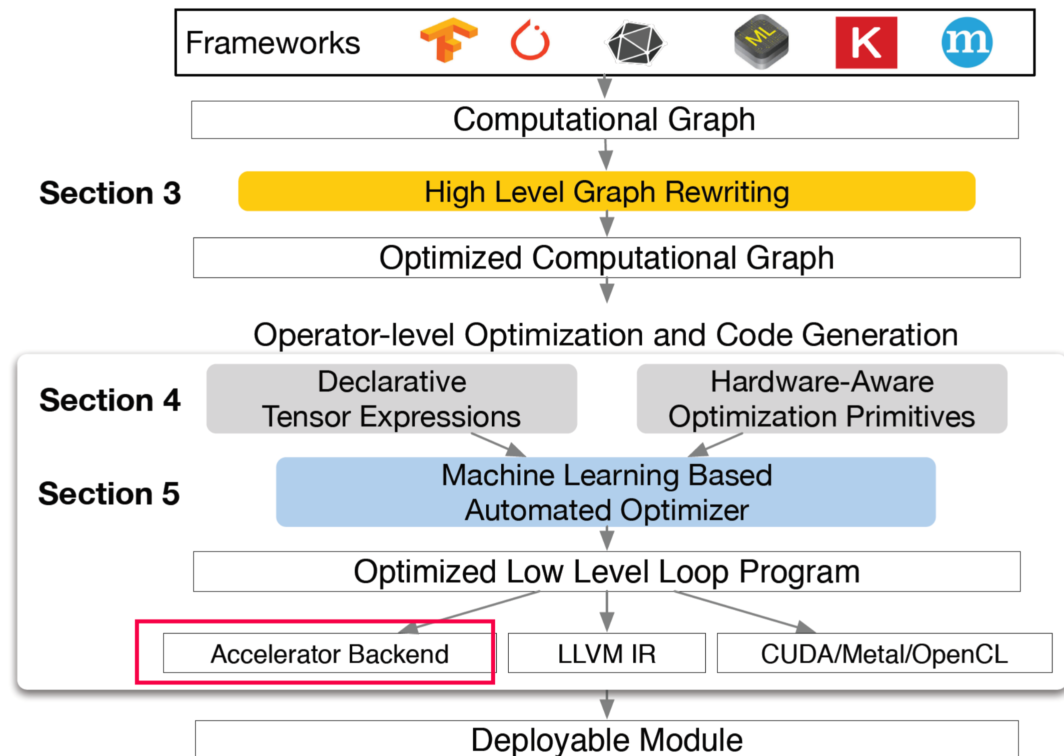


[TOP 500 Supercomputers:
Summit @ Oak Ridge NL ('18):
200.7 PFLOP/s (2.4M cores)]

Recap: Operator Fusion and Code Generation

- **TVM: Code Generation for HW Accelerators**
  - Graph- /operator-level optimizations for **embedded and HW accelerators**
  - **Lack of low-level instruction set!**
  - Schedule Primitives
    - Loop Transform
    - Thread Binding
    - Compute Locality
    - Tensorization
    - Latency Hiding

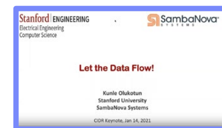
[Tianqi Chen et al: TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. **OSDI 2018**]



➔ Apache tvm

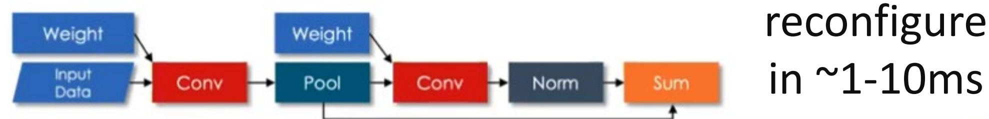
SambaNova

[Kunle Olukotun: Let the Data Flow!, **CIDR 2021**, https://www.youtube.com/watch?v=iHhHHBuk3W4, **SDSC 2020**, https://www.youtube.com/watch?v=E7se0KEa4BY]

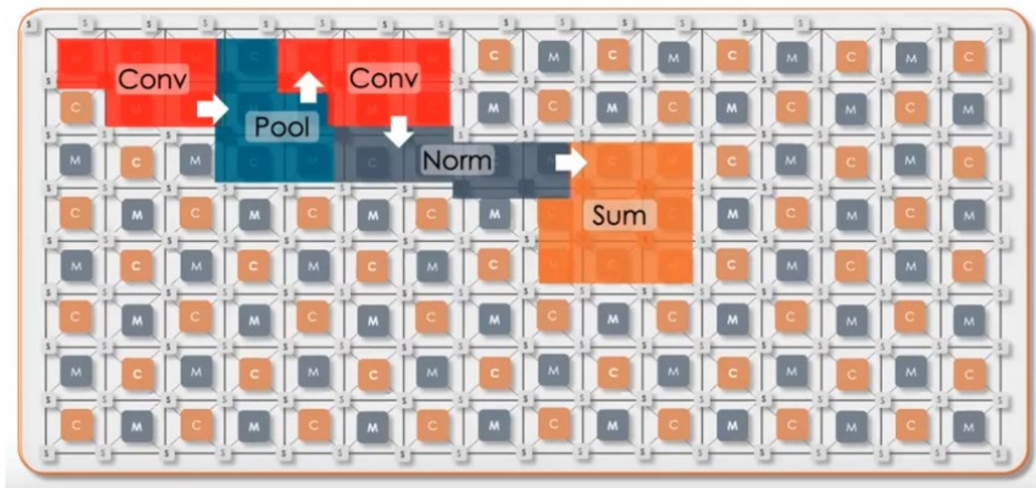


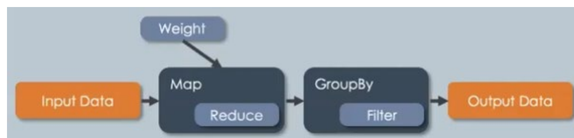
- **Overview**
  - Reconfigurable data flow architecture
  - Based on **hierarchical parallel patterns** (map, zip, reduce, flatMap, groupBy)
  - Reconfigurable Dataflow Unit (**RDU**), 100s of TFLOPs, 100s MB on chip

- **Mapping of Dataflow Computation**
  - DNN / ML
  - Graph processing
  - SQL query processing



reconfigure in ~1-10ms

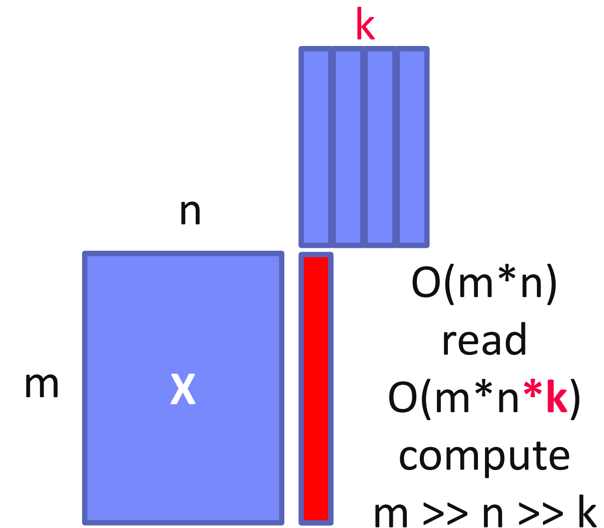




Caching, Partitioning, and Indexing

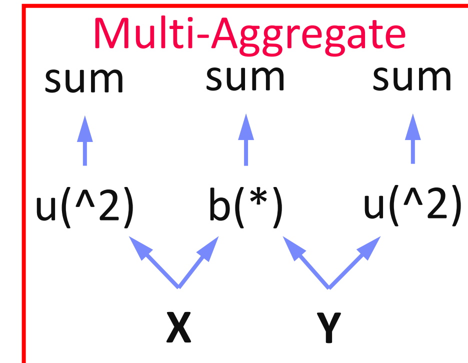
Scan Sharing

- **#1 Batching**
  - One-pass evaluation of multiple configurations
  - Use cases: EL, CV, feature selection, hyper parameter tuning, multi-user scoring
  - E.g.: **TUPAQ** [SoCC'16], **Columbus** [SIGMOD'14]



- **#2 Fused Operator DAGs**
  - Avoid unnecessary scans, (e.g., mmchain)
  - Avoid unnecessary writes / reads
  - Multi-aggregates, redundancy
  - E.g.: **SystemML codegen** [PVLDB'18]

```
a = sum(X^2)
b = sum(X*Y)
c = sum(Y^2)
```



- **#3 Runtime Piggybacking**
  - Merge concurrent data-parallel jobs
  - "Wait-Merge-Submit-Return"-loop
  - E.g.: **SystemML parfor** [PVLDB'14]

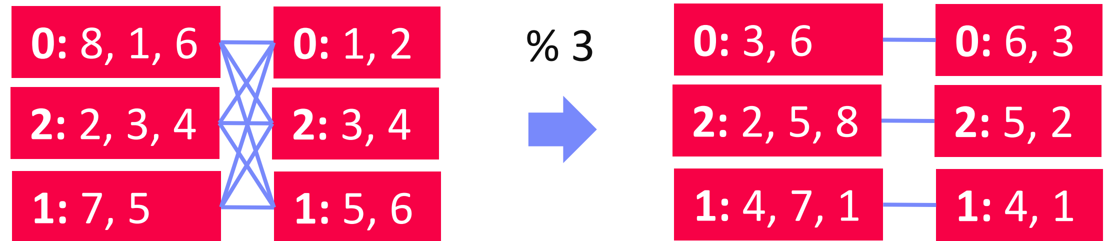
```
parfor( i in 1:numModels )
  while( !converged )
    q = X %*% v; ...
```

Distributed Partitioning

- **Spark RDD Partitioning**
  - Implicitly on every data shuffling
  - Explicitly via `R.repartition(n)`

Example Hash Partitioning:
For all (k,v) of R:
hash(k) % numPartitions → pid

- **Distributed Joins**
  - R3 = R1.join(R2)



- **Single-Key Lookups** `v = C.lookup(k)`
  - **Without partitioning:** scan all keys (reads/deserializes out-of-core data)
  - **With partitioning:** lookup partition, scan keys of partition

- **Multi-Key Lookups**
  - Without partitioning: scan all keys
  - With partitioning: lookup relevant partitions

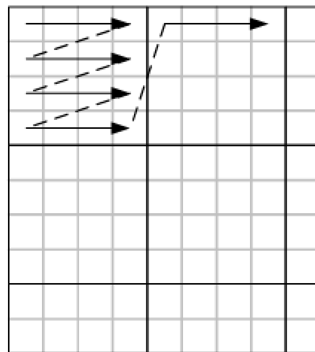
```
//build hashset of required partition ids
HashSet<Integer> flags = new HashSet<>();
for( MatrixIndexes key : filter )
   flags.add(partitioner.getPartition(key));

//create partition pruning rdd
ppRDD = PartitionPruningRDD.create(in.rdd(),
    new PartitionPruningFunction(flags));
```

Linearized Array B-Tree (LAB-Tree)

[Yi Zhang, Kamesh Munagala, Jun Yang: Storing Matrices on Disk: Theory and Practice Revisited. **PVLDB 2011**]

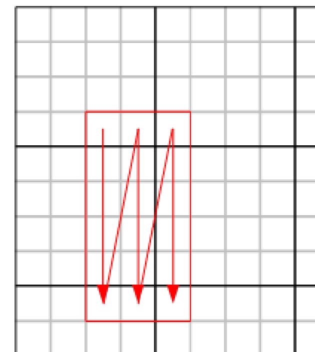
- **Basic Ideas**
  - B-tree over linearized array representation (e.g., row-/col-major, Z-order, UDF)
  - New leaf splitting strategies; dynamic leaf storage format (sparse and dense)
  - Various flushing policies for update batching (all, LRU, smallest page, largest page, largest page probabilistically, largest group)

#1 Example linearized storage order



matrix A:
4 x 4 blocking
row-major block order
row-major cell order

#2 Example linearized iterator order



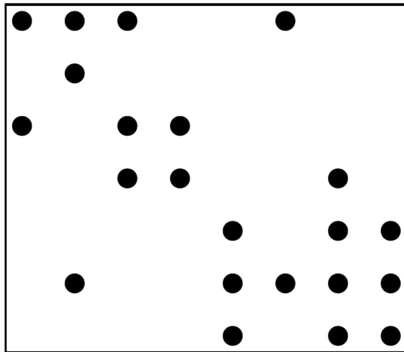
range query A[4:9,3:5]
with column-major
iterator order

Adaptive Tile (AT) Matrix

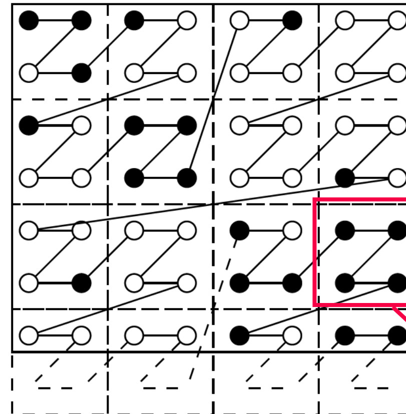
[David Kernert, Wolfgang Lehner, Frank Köhler: Topology-aware optimization of big sparse matrices and matrix multiplications on main-memory systems. **ICDE 2016**]

- **Basic Ideas**
  - Two-level blocking and NUMA-aware range partitioning (tiles, blocks)
  - Z-order linearization, and **recursive quad-tree partitioning** to find var-sized tiles (tile contains N blocks)

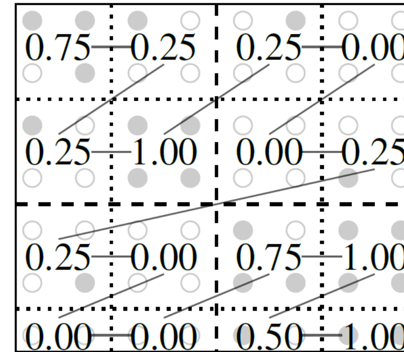
Input Matrix

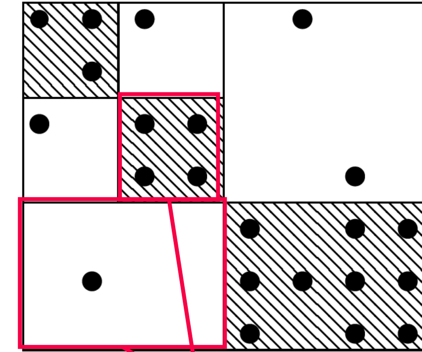


Z-ordering



Density Map
(see sparsity est.)





block

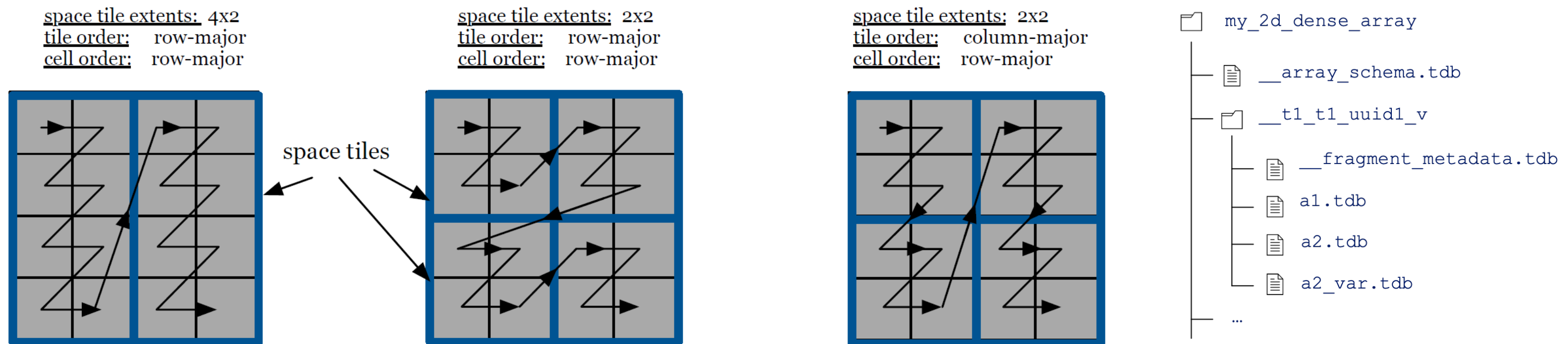
tiles

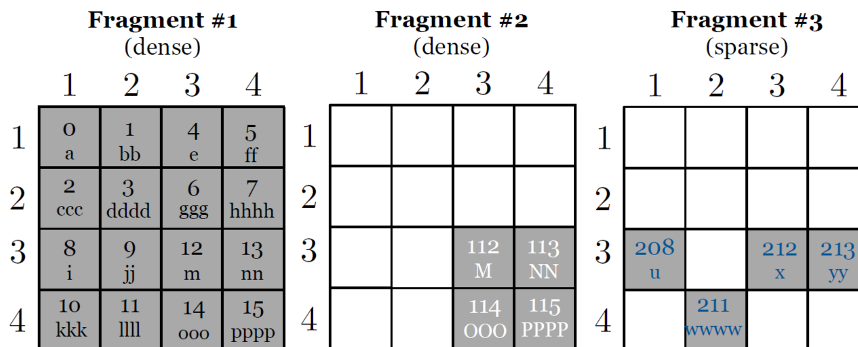
TileDB Storage Manager

[Stavros Papadopoulos, Kushal Datta, Samuel Madden, Timothy G. Mattson: The TileDB Array Data Storage Manager. **PVLDB 2016**]

https://docs.tiledb.com

- **Basic Ideas**
  - Storage manager for 2D arrays of different data types (incl. vector, 3D)
  - Two-level blocking (space/data tiles), update batching via fragments





Fragment #1 (dense)

	1	2	3	4
1	0 a	1 bb	4 e	5 ff
2	2 ccc	3 dddd	6 ggg	7 hhhh
3	8 i	9 jj	12 m	13 nn
4	10 kkk	11 llll	14 ooo	15 pppp

Fragment #2 (dense)

	1	2	3	4
1				
2				
3			112 M	113 NN
4			114 OOO	115 PPPP

Fragment #3 (sparse)

	1	2	3	4
1				
2				
3	208 u		212 x	213 yy
4		211 wwww		

Fragment #1
Fragment #2
Fragment #3

Collective logical array view

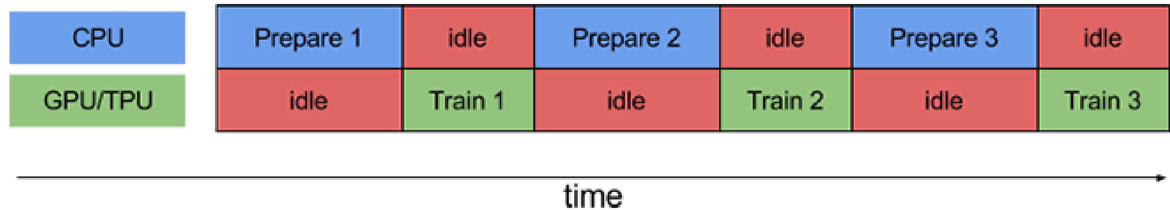
	1	2	3	4
1	0 a	1 bb	4 e	5 ff
2	2 ccc	3 dddd	6 ggg	7 hhhh
3	208 u	9 jj	212 x	213 yy
4	10 kkk	211 wwww	114 OOO	115 PPPP

Pipelining for Mini-batch Algorithms

- **Motivation**
  - Overlap data access and computation in mini-batch algorithms (e.g., DNN)

  ➔ **Simple pipelining of I/O and compute via queueing / prefetching**

- **Example TensorFlow**
  - **#1** Queueing and Threading



  - **#2 Dataset API Prefetching**

    [https://www.tensorflow.org/guide/performance/datasets]

```
dataset = dataset.batch(batch_size=32)
dataset = dataset.prefetch(buffer_size=1)
```

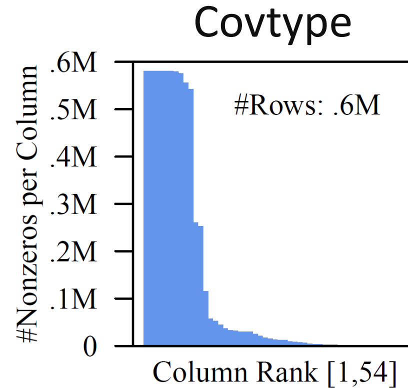
  - **#3** Reuse via **Data Echoing**

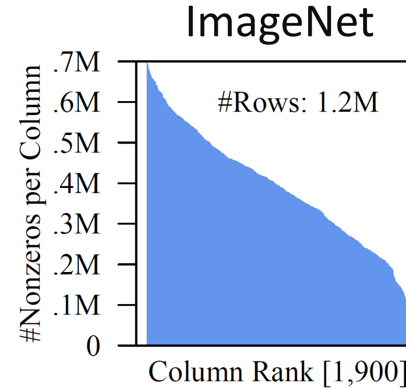
[https://ai.googleblog.com/2020/05/speeding-up-neural-network-training.html]

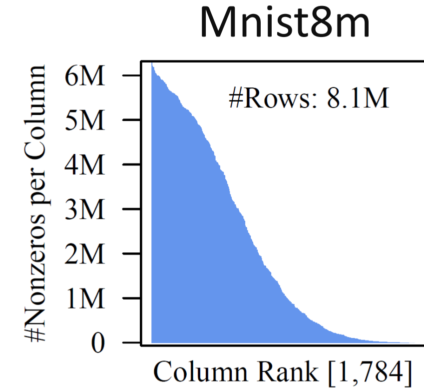
Lossy and Lossless Compression

Motivation: Data Characteristics

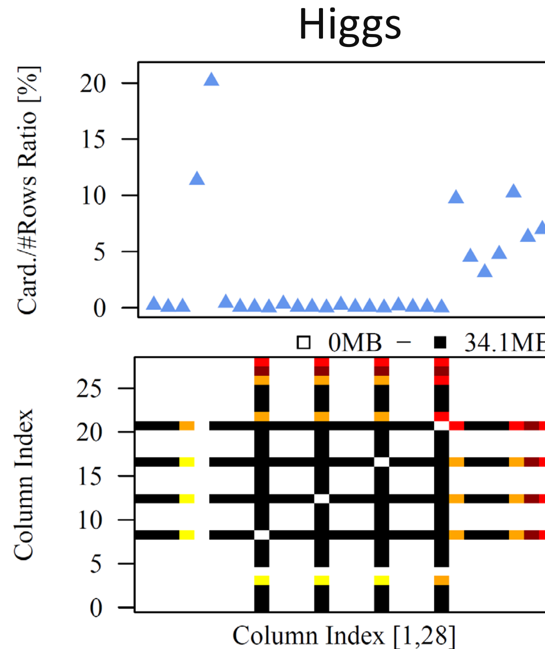
- **Tall and Skinny** (#rows >> #cols)
- **Non-Uniform Sparsity**

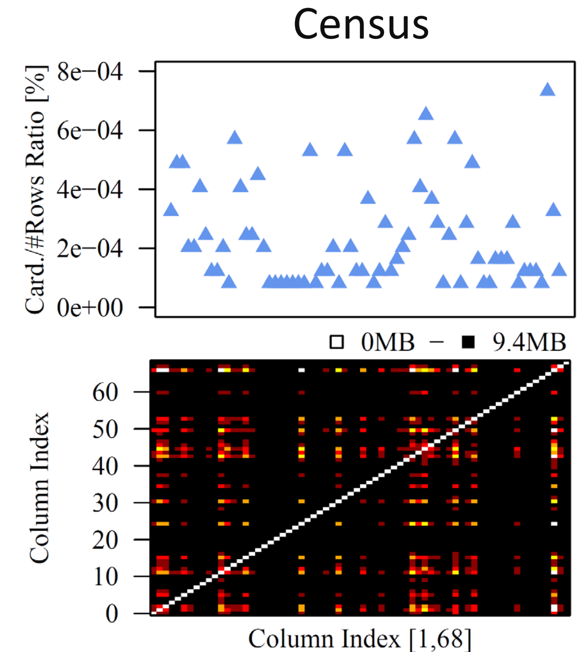






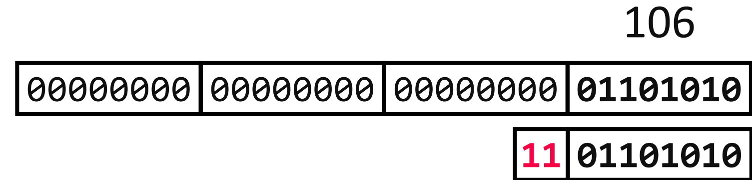
- **Small Column Cardinalities**
- **Small Val Range**
- **Column Correlations** (on census: 12.8x → 35.7x)



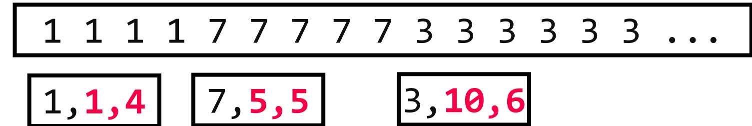


Recap: Database Compression Schemes

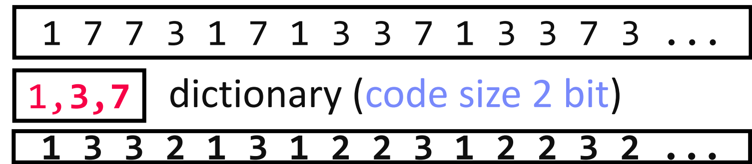
- **Null Suppression**
  - Compress integers by **omitting leading zero** bytes/bits (e.g., NS, gamma)



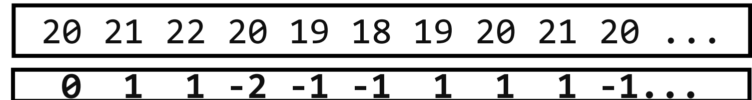
- **Run-Length Encoding**
  - Compress sequences of equal values by **runs** of (value, start, run length)



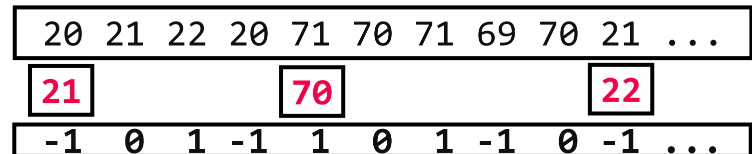
- **Dictionary Encoding**
  - Compress column w/ few distinct values as **pos in dictionary** (→ code size)



- **Delta Encoding**
  - Compress sequence w/ small changes by storing **deltas to previous value**

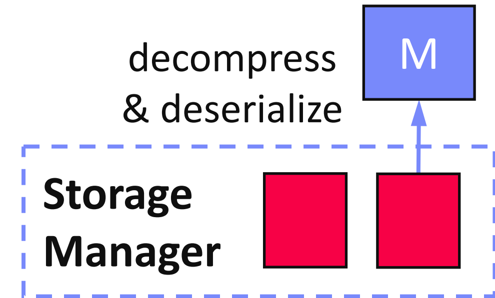


- **Frame-of-Reference Encoding**
  - Compress values by storing **delta to reference value** (outlier handling)

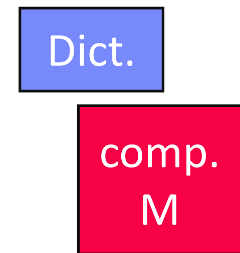


Overview Lossless Compression Techniques

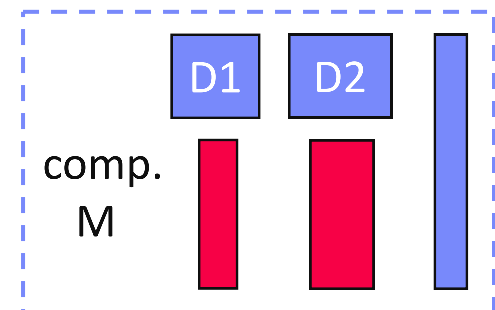
- **#1 Block-Level General-Purpose Compression**
  - Heavyweight or lightweight compression schemes
  - Decompress matrices block-wise for each operation
  - E.g.: Spark RDD compression (Snappy/LZ4), **SciDB** SM [SSDBM'11], **TileDB** SM [PVLDB'16], scientific formats **NetCDF**, **HDF5** at chunk granularity



- **#2 Block-Level Matrix Compression**
  - Compress matrix block with homogeneous encoding scheme
  - Perform LA ops over compressed representation
  - E.g.: **CSR-VI** (dict) [CF'08, TPDS'13], **cPLS** (grammar) [KDD'16], **TOC** (LZW w/ trie) [SIGMOD'19]



- **#3 Column-Group-Level Matrix Compression**
  - Compress column groups w/ heterogeneous schemes
  - Perform LA ops over compressed representation
  - E.g.: **SystemML CLA** (RLE, OLE, DDC, UC) [PVLDB'16]



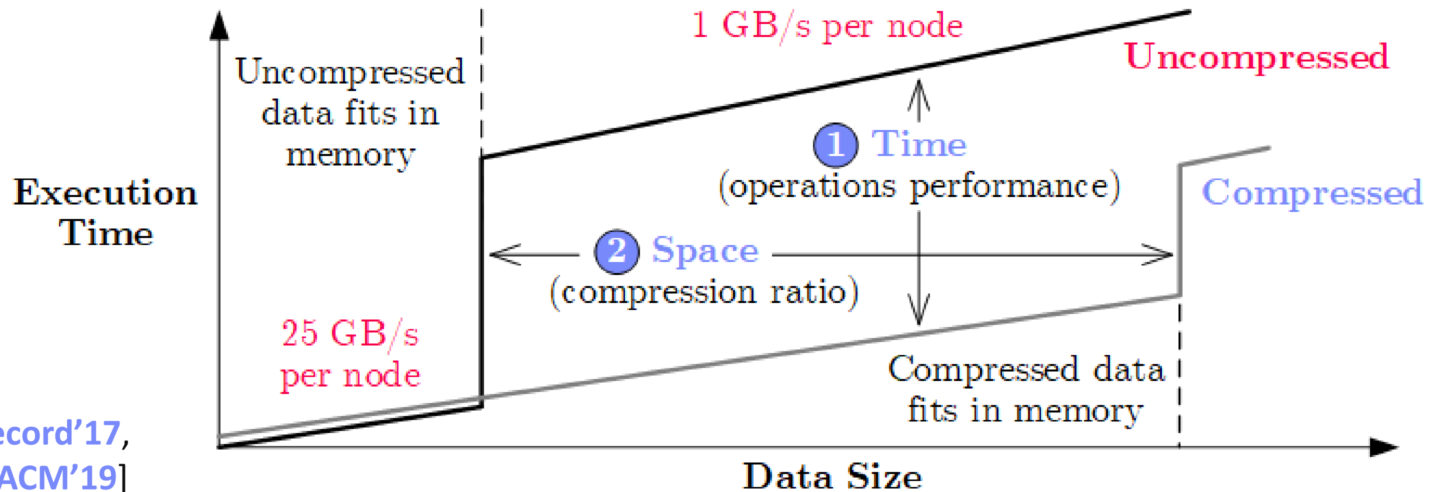
CLA: Compressed Linear Algebra

[Ahmed Elgohary et al: Compressed Linear Algebra for Large-Scale Machine Learning. **PVLDB 2016**]

- **Key Idea**
  - Use lightweight database compression techniques
  - Perform LA operations on compressed matrices
- **Goals of CLA**
  - Operations performance close to uncompressed
  - Good compression ratios

X

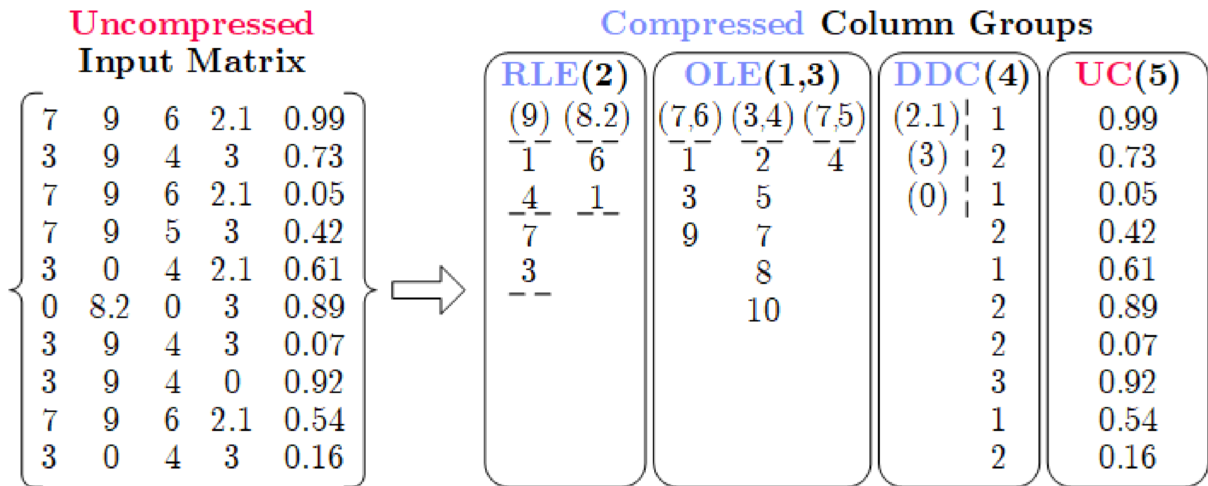
```
while(!converged) {
   … q = X %*% v …
}
```



[SIGMOD Record'17, VLDBJ'18, CACM'19]

CLA: Compressed Linear Algebra, cont.

- **Overview Compression Framework**
  - Column-wise matrix compression (values + compressed offsets / references)
  - **Column co-coding** (column groups, encoded as single unit)
  - **Heterogeneous column encoding formats** (w/ dedicated **physical encodings**)
- **Column Encoding Formats**
  - Offset-List (OLE)
  - Run-Length (RLE)
  - Dense Dictionary Coding (DDC)*
  - Uncompressed Columns (UC)



* DDC1/2 in VLDBJ'18

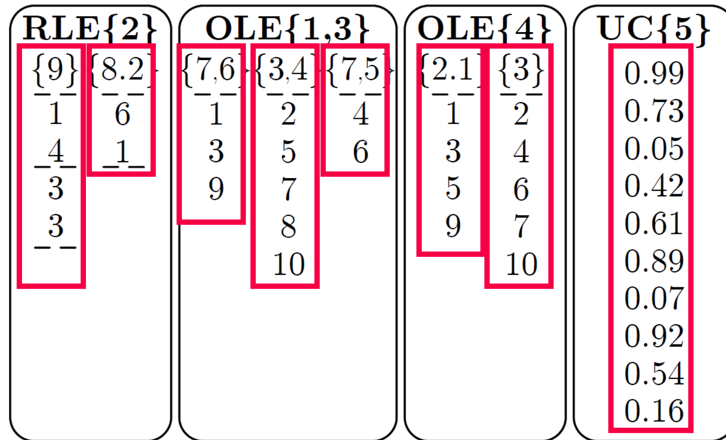
- **Automatic Compression Planning** (**sampling-based**)
  - Select column groups and formats per group (data dependent)

CLA: Compressed Linear Algebra, cont.

- **Matrix-Vector Multiplication**
  - Naïve: for each tuple, pre-aggregate values, add values at offsets to q

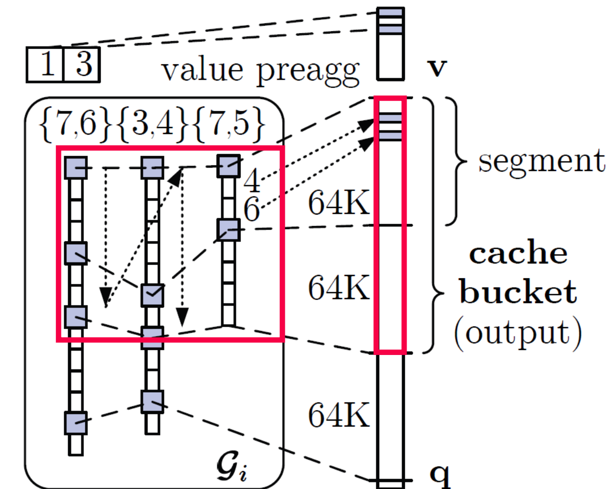
    Example: q = X v, with v = (7, 11, 1, 3, 2)

9*11=99 90.2 55 25 54 6.3 9



162.3
134.5
160.4
162.8
32.5
155
133.1
125.8
161.4
34.3

➔ cache unfriendly on output (q)



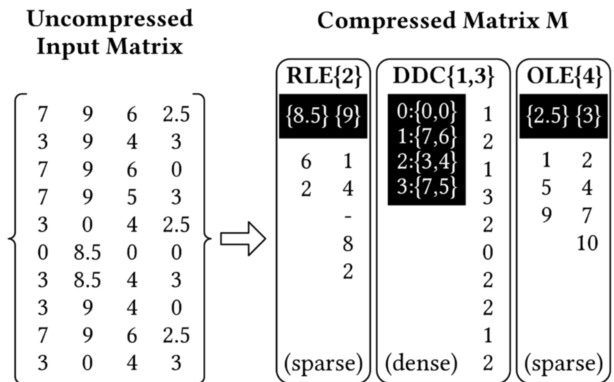
  - Cache-conscious: Horizontal, segment-aligned scans, maintain positions

- **Vector-Matrix Multiplication**
  - Naïve: cache-unfriendly on input (v)
  - Cache-conscious: again use horizontal, segment-aligned scans

Compressed Linear Algebra Extended

[SIGMOD 2023]

- **Lossless Matrix Compression**
  - Improved general applicability (compression time, new compression schemes, new kernels, intermediates, workload-aware)
  - Sparsity → Redundancy exploitation (data redundancy, structural redundancy)



- **Workload-aware Compression**
  - Workload summary → compression
  - Compression → execution planning

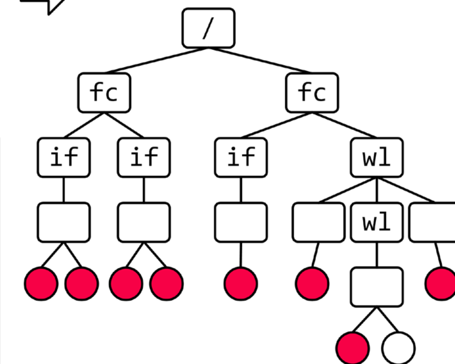
User Script:

```
X = read("data/X")
y = read("data/y")

X = scale(X,TRUE,TRUE)
w = l2svm(X,y,TRUE,
      1e-9,1e-3,100)

write(w,"data/wXy")
```

```
if(shift)
 X = X - colMeans(X)
if(scale)
 X = X / colSds(X)
```

Built-in Functions:

```
if(intercept)
 X = cbind(X,ones)
while(conto & i<maxi) {
 Xd = X %*% s
 while(conti) {
  out = 1-y*(Xw+sz*Xd)
  sz = sz - g/h; # ...
 }
 g_new = t(X) %*% (out*y)
}
```



Tuple-oriented Compression (TOC)

- **Motivation**
  - DNN and ML often trained with **mini-batch SGD**
  - Effective compression for small batches (#rows)

[Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F. Naughton, Jignesh M. Patel: Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent, **SIGMOD 2019**]

Original Table (A)

	1	2	3	4
R1	1.1	2	3	1.4
R2	1.1	2	3	0
R3	0	1.1	3	1.4
R4	1.1	2	0	0

Step1: Sparse Encoding

Sparse Encoded Table (B)

R1	1:1.1	2:2	3:3	4:1.4
R2	1:1.1	2:2	3:3	
R3	2:1.1	3:3	4:1.4	
R4	1:1.1	2:2		

Step2 :Logical Encoding

Prefix Tree (C)

Encoded Table (D)

R1	1	2	3	4
R2	6	3		
R3	5	8		
R4	6			

Column_index:value pairs in the first layer of the prefix tree (I)

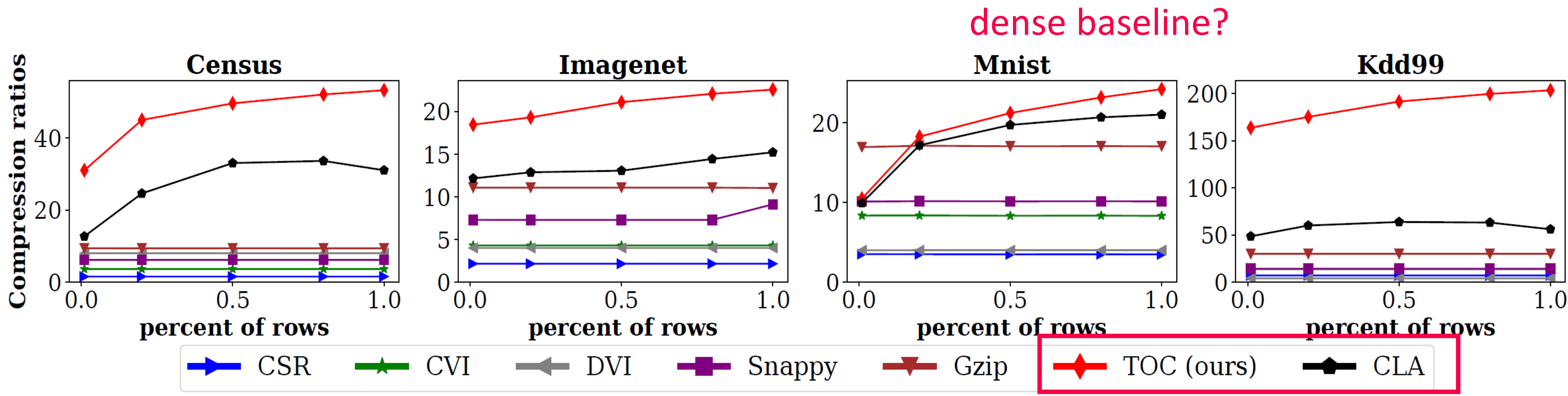
Step3 :Physical Encoding

Step3 :Physical Encoding

Tuple-oriented Compression (TOC), cont.

- **Example Compression Ratios**

[Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F. Naughton, Jignesh M. Patel: Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent, **SIGMOD 2019**]



Take-away: specialized lossless matrix compression
→ **reduce memory bandwidth requirements and #FLOPs**

Lossy Compression

- **Overview**
  - Extensively used in DNN (runtime vs accuracy) ➔ **data format + compute**
  - **Careful manual application** regarding data and model
  - **Note:** ML algorithms approximate by nature + noise generalization effect

- **Background Floating Point Numbers (IEEE 754)**
  - Sign s, Mantissa m, Exponent e: `value = s * m * 2^e` (simplified)

Precision	Sign	Mantissa	Exponent
Double (FP64)	1	52	11
Single (FP32)	1	23	8
Half (FP16)	1	10	5
Quarter (FP8)	1	3	4
Half-Quarter (FP4)	1	1	2

[bits]

Low and Ultra-low FP Precision

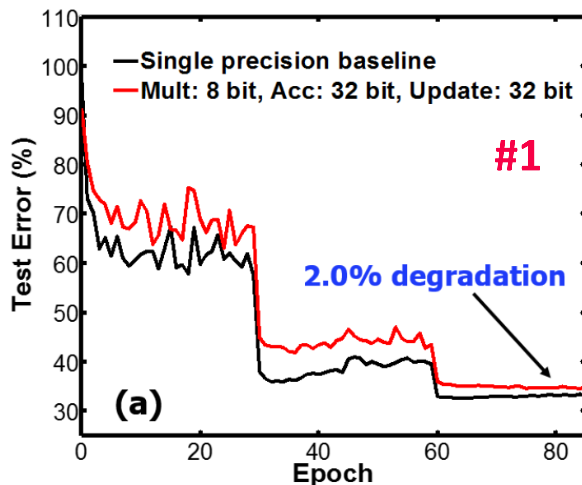
- **Model Training w/ low FP Precision**

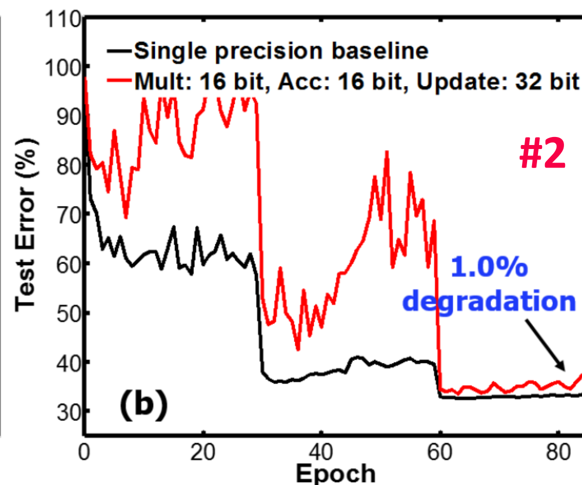
see 05 Execution Strategies, SIMD
→ speedup/reduced energy

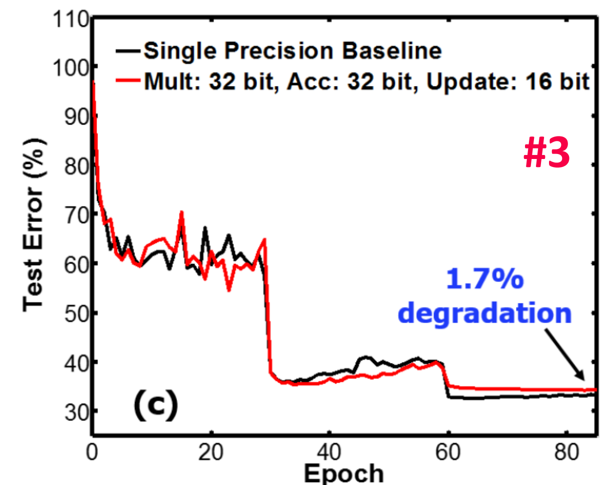
  - Trend: from **FP32/FP16** to **FP8**
  - **#1: Precision of intermediates** (weights, act, errors, grad) → loss in accuracy
  - **#2: Precision of accumulation** → impact on convergence (swamping s+L)
  - **#3: Precision of weight updates** → loss in accuracy

- **Example ResNet18 over ImageNet**

[Naigang Wang et al.: Training Deep Neural Networks with **8-bit** Floating Point Numbers. **NeurIPS 2018**]







Low and Ultra-low FP Precision, cont.

- **Numerical Stable Accumulation**
  - **#1 Sorting ASC + Summation**
  - **#2 Kahan Summation** w/ error independent of number of values n
  - **#3 Pairwise Summation** (divide & conquer)

[Yuanyuan Tian, Shirish Tatikonda, Berthold Reinwald: Scalable and Numerically Stable Descriptive Statistics in SystemML. **ICDE 2012**]

```
sumOld = sum;
sum = sum + (input + corr);
corr = (input + corr) – (sum – sumOld);

uak+: 5.000000005E17 //sum(seq(1,1e9))
ua+:  5.000000109721722E17
ua+:  5.000000262154688E17 //rev
```

- **#4 Chunk-based Accumulation**
  - Divide long dot products into smaller chunks
  - Hierarchy of partial sums → **FP16 accumulators**

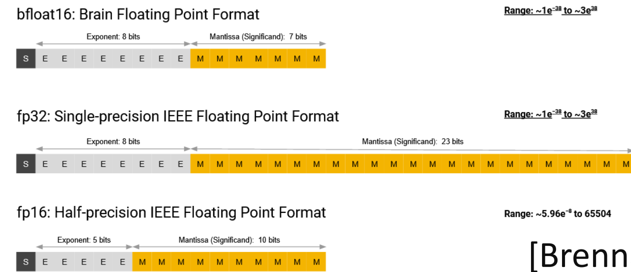
[N. Wang et al.: Training Deep Neural Networks with **8-bit** Floating Point Numbers. **NeurIPS 2018**]

- **#5 Stochastic Rounding**
  - Replace nearest w/ prob. rounding

$$Round(x) = \begin{cases} s \cdot 2^e \cdot (1 + \lfloor m \rfloor + \epsilon) & \text{with probability } \frac{m - \lfloor m \rfloor}{\epsilon}, \\ s \cdot 2^e \cdot (1 + \lfloor m \rfloor) & \text{with probability } 1 - \frac{m - \lfloor m \rfloor}{\epsilon}, \end{cases}$$

Low and Ultra-low FP Precision – New Datatypes

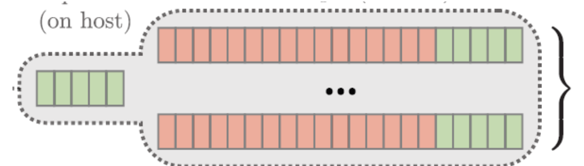
- **Google** `bfloat16`
  - "Brain" Float16 w/ range of FP32
  - Drop in replacement for FP32, no need for loss scaling



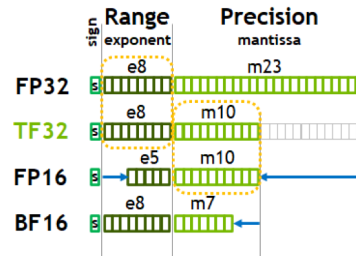
[Brennan Saeta: Training Performance A user's guide to converge faster, **TF Dev Summit 2018**]

- **Intel** `FlexPoint`
  - Blocks of values w/ shared exponent (N=16bit w/ M=5bit exponent)
  - Example: flex16+5

[Urs Köster et al.: Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. **NeurIPS 2017**]



- **NVIDIA** `TF32`
  - Range of FP32 w/ precision of FP16



[NVIDIA A100 Tensor Core GPU Architecture - UNPRECEDENTED ACCELERATION AT EVERY SCALE, Whitepaper, **Aug 2020**]

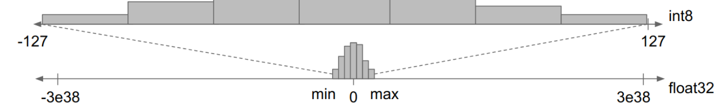
Fixed-Point Arithmetic

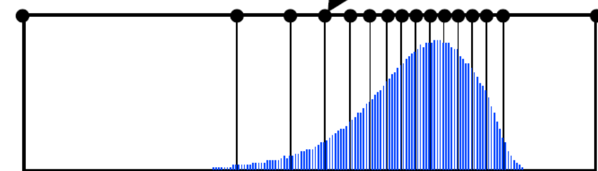
Recommended "Reading"
[Inside TensorFlow: Model Optimization Toolkit (Quantization and Pruning), **YouTube, 2020**]

- **Motivation**
  - Forward-pass for model scoring (inference) can be done in **UINT8** and below
  - **Static, dynamic, and learned quantization** schemes (**weights** and **inputs**)

- **Quantization** (reduce value domain)
  - **Split value domain into N buckets** such that $k = \log_2 N$ can encode the data
  - **a) Static Quantization** (e.g., min/max) per tensor or per tensor channel
  - **b) Learned Quantization** Schemes
    - Dynamic programming
    - Various heuristics
    - Example systems: **ZipML**, **SketchML**

[https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html]





[Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, Ce Zhang: ZipML: Training Linear Models with End-to-End Low Precision, and a Little Bit of Deep Learning. **ICML 2017**]

Other Lossy Techniques

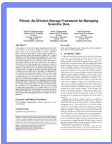
[https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html]

- **#1 Sparsification/Pruning** (reduce #non-zeros)
  - **Value clipping:** zero-out very small values below a threshold to reduce size of weights
  - **Training w/ target sparsity:** remove connections

Sparse Accuracy	NNZ
78.1% @ sp=1.0	27.1M
78.0% @ sp=0.5	13.6M
76.1% @ sp=0.25	6.8M
74.6% @ sp=0.125	3.3M

- **#2 Mantissa Truncation**
  - Truncate m of FP32 from 23bit to 16bit
  - E.g., **TensorFlow** (transfers), **PStore**

[Souvik Bhattacherjee et al: PStore: an efficient storage framework for managing scientific data. **SSDBM 2014**]



- **#3 Aggregated Data Representations**
  - a) Dim reduction (e.g., auto encoders)
  - b) No FK-PK joins in Factorized Learning (**foreign key** as lossy compressed rep)

[Amir Ilkhechi et al: DeepSqueeze: Deep Semantic Compression for Tabular Data, **SIGMOD 2020**]

[Arun Kumar et al: To Join or Not to Join?: Thinking Twice about Joins before Feature Selection. **SIGMOD 2016**]

- **#4 Sampling**
  - User specifies **approximation contract** for error (regression/classification) and scale
  - Min sample size for **max likelihood estimators**

[Yongjoo Park et al: BlinkML: Efficient Maximum Likelihood Estimation with Probabilistic Guarantees. **SIGMOD 2019**]

Summary and Q&A

- **GPUs in ML Systems**
- **FPGAs in ML Systems**
- **ASICs and other HW Accelerators**
- **Caching, Partitioning, and Indexing**
- **Lossy and Lossless Compression**

High Impact on Performance/Energy

→ **Different Levels of Hardware Specialization**
- General-purpose CPUs and GPUs
- FPGAs, DNN ASICs, and other technologies

Specialization w/o Abstraction is harmful

→ **Different Levels of Data Layout Specialization**
- Lossless caching, partitioning, indexing, compression
- Lossy compression, sparsification