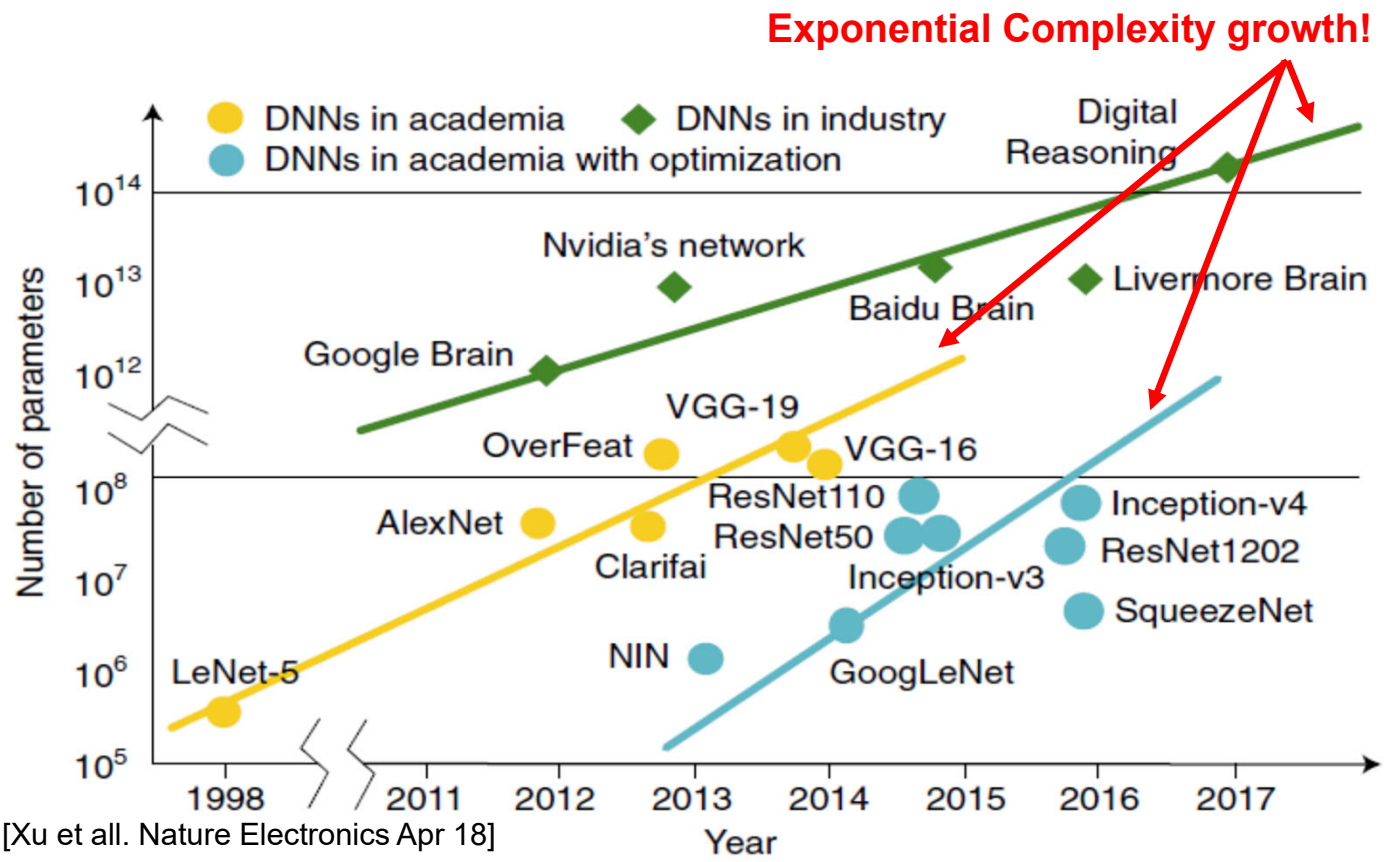
ETH zürich

NTX: An Energy-efficient Streaming Accelerator for Floating-point Generalized Reduction Workloads in 22nm FD-SOI

Fabian Schuiki[1], Michael Schaffner[1], Luca Benini[1,2]
ETH Zurich[1] and University of Bologna[2]

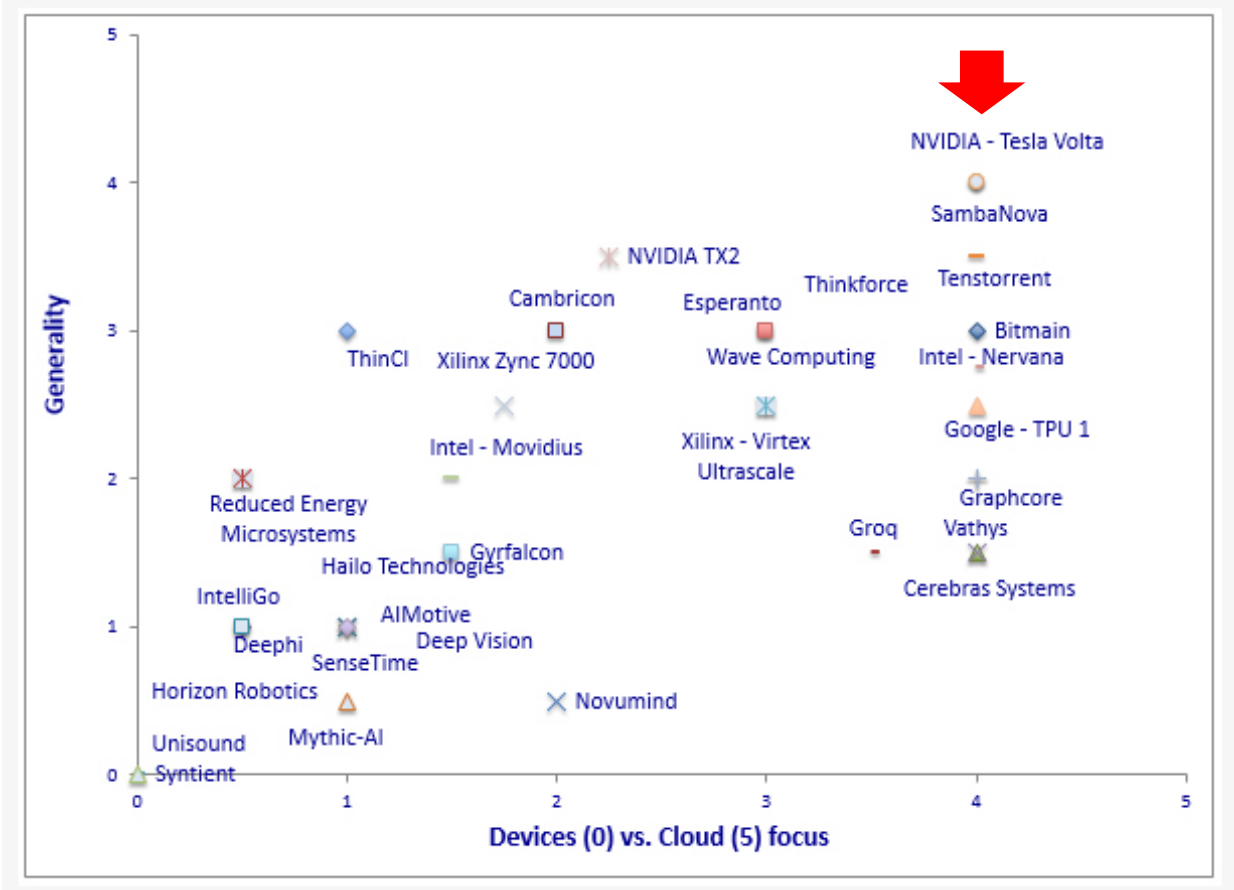
PRECOMP
Open Transprecision Computing

Machine Learning @ Cloud



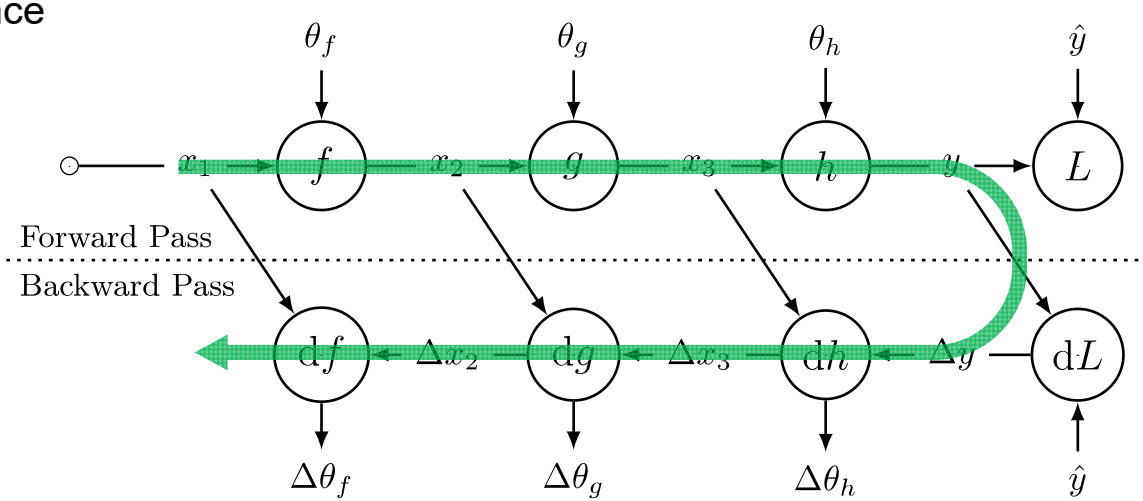
[Xu et all. Nature Electronics Apr 18]

ML: HW reinassance!



DNN training as a workload

- **Inference**
  - Well covered
  - Compute layer, only keep results, advance
- **Training**
  - **Long data dependency chain**
  - Intermediate results must be stored
  - Cannot fuse ReLU with convolution
  - Derivatives tricky (ReLU, Maxpool)
- **Offloading**
  - Convolution needs **6** loops
  - Accelerator must have high autonomy
  - Processor cores orchestrate training



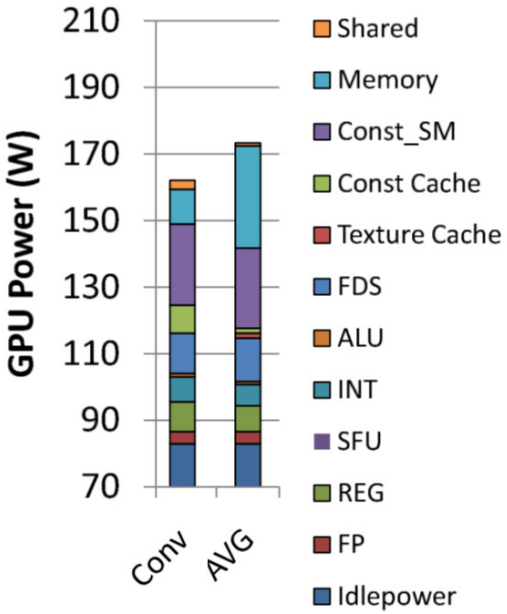
+ higher precision & dynamic range requirements!

Downside of GPUs for Training

- GPUs are the workhorse of data center DNN training
- They generally have a TDP of 200-300 W
- Peak compute reaches 15 Tflop/s these days

- Only **4.9%** of that power estimated to be spent in the FPUs [1]:
  - [1] reports 2.9%, but their kernels don't reach TDP/max perf.
  - In dubio pro Invidia: We scale power to assume modern GPUs can reach TDP at max perf. (making them more efficient).

- In practice GPU efficiency hard to estimate:
  - GPUs may not be able to reach max compute before hitting TDP



Graph extracted and cropped from [1].

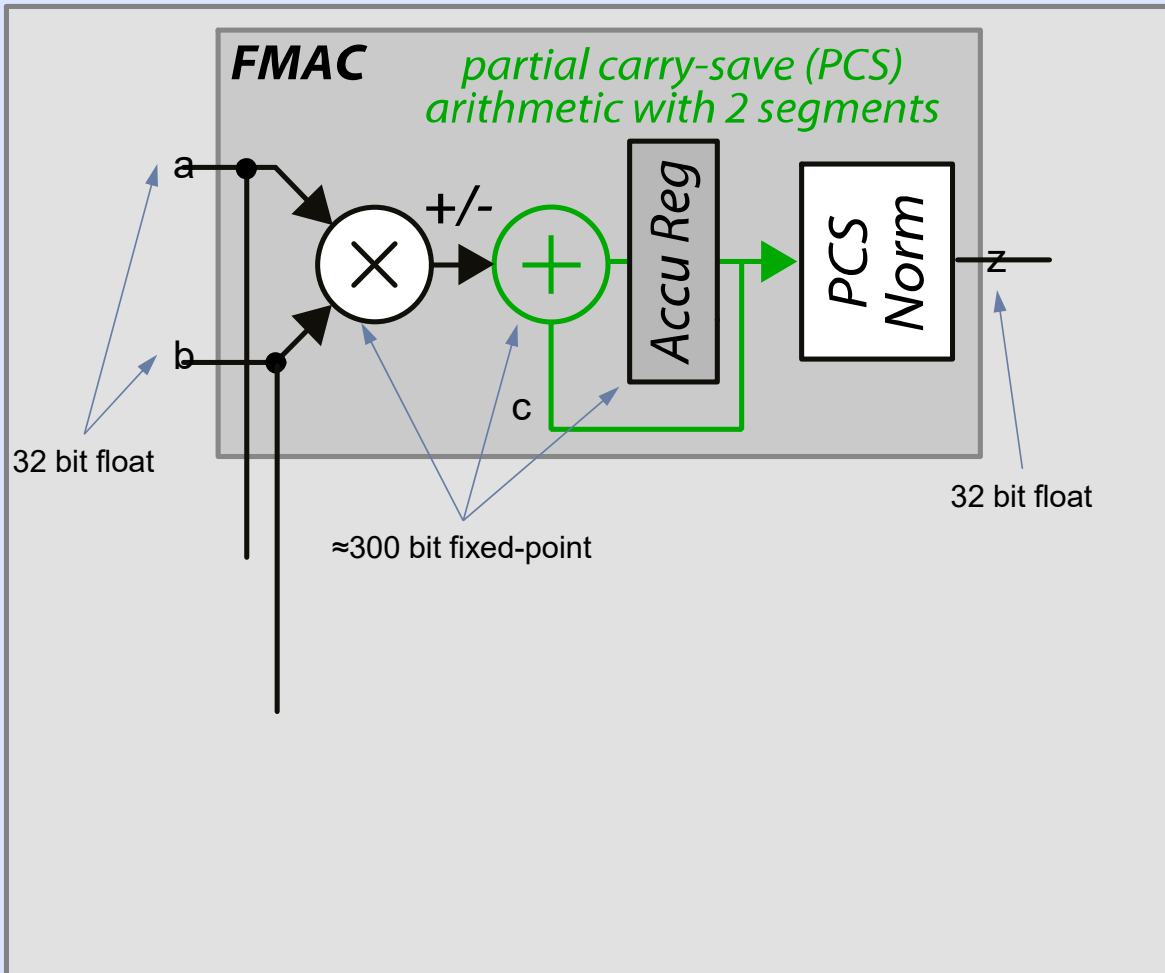
[1] S. Hong and H. Kim, "An integrated gpu power and performance model," in ACM SIGARCH Computer Architecture News, 2010.

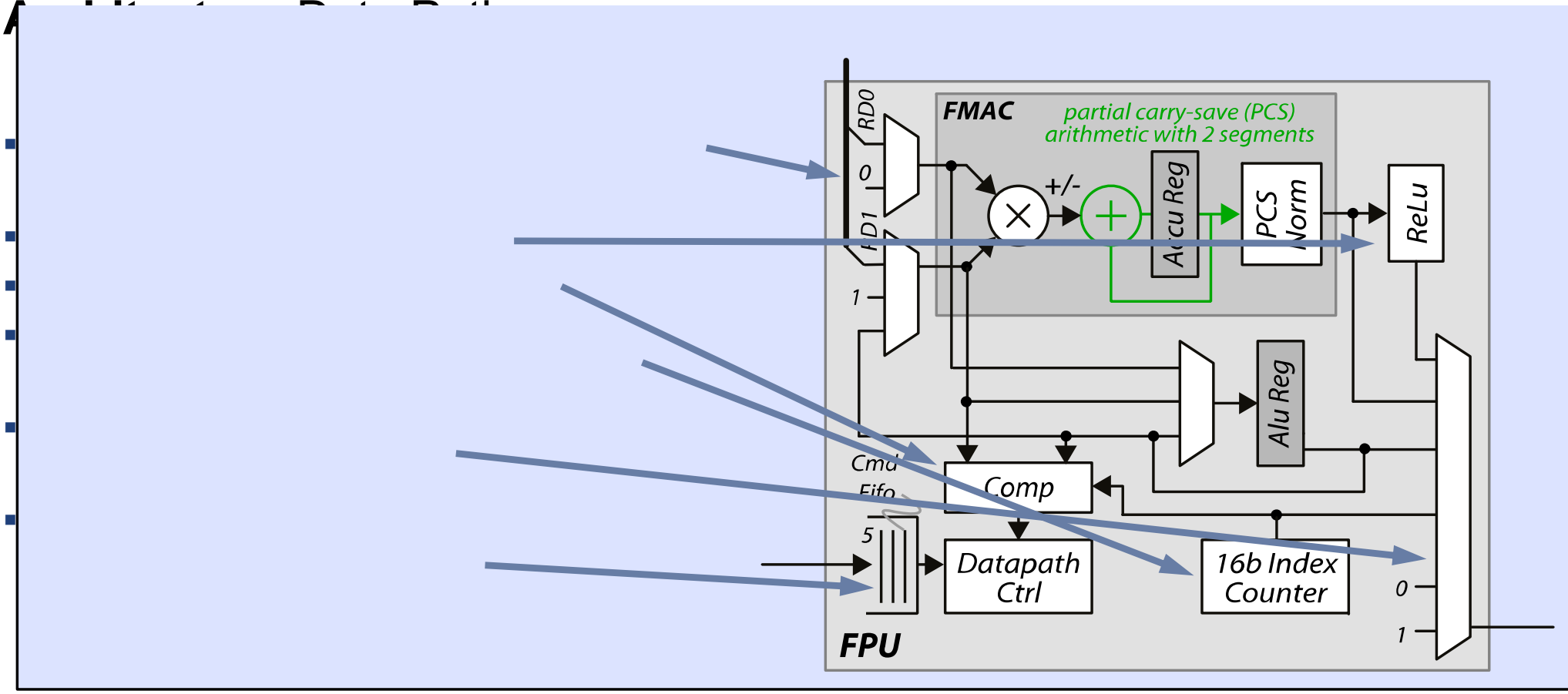
Network Training Accelerator (NTX)

- We propose NTX [1]
- Streaming floating-point co-processor
- Efficiently performs **float32 FMAC**
  - Fast multiply-accumulate, single cycle
- Address generation unit ensures **low control overhead**
  - 5 nested hardware loops
  - 3 address generators
- Many common C/C++ loop nests **map well** to this architecture
- **8 NTX** paired up per associated processor core
- Floating-point operation makes accelerator a drop-in replacement for GPUs for **training**.
- Based on lessons learned from Neurostream [2]

[1] Schuiki et al, “A Scalable Near-Memory Architecture for Training Deep Neural Networks on Large In-Memory Datasets,”, in IEEE TC 2019.
[2] Azarkhish et al, "Neurostream: Scalable and energy efficient deep learning with smart memory cubes,” in IEEE TPDS 2018.

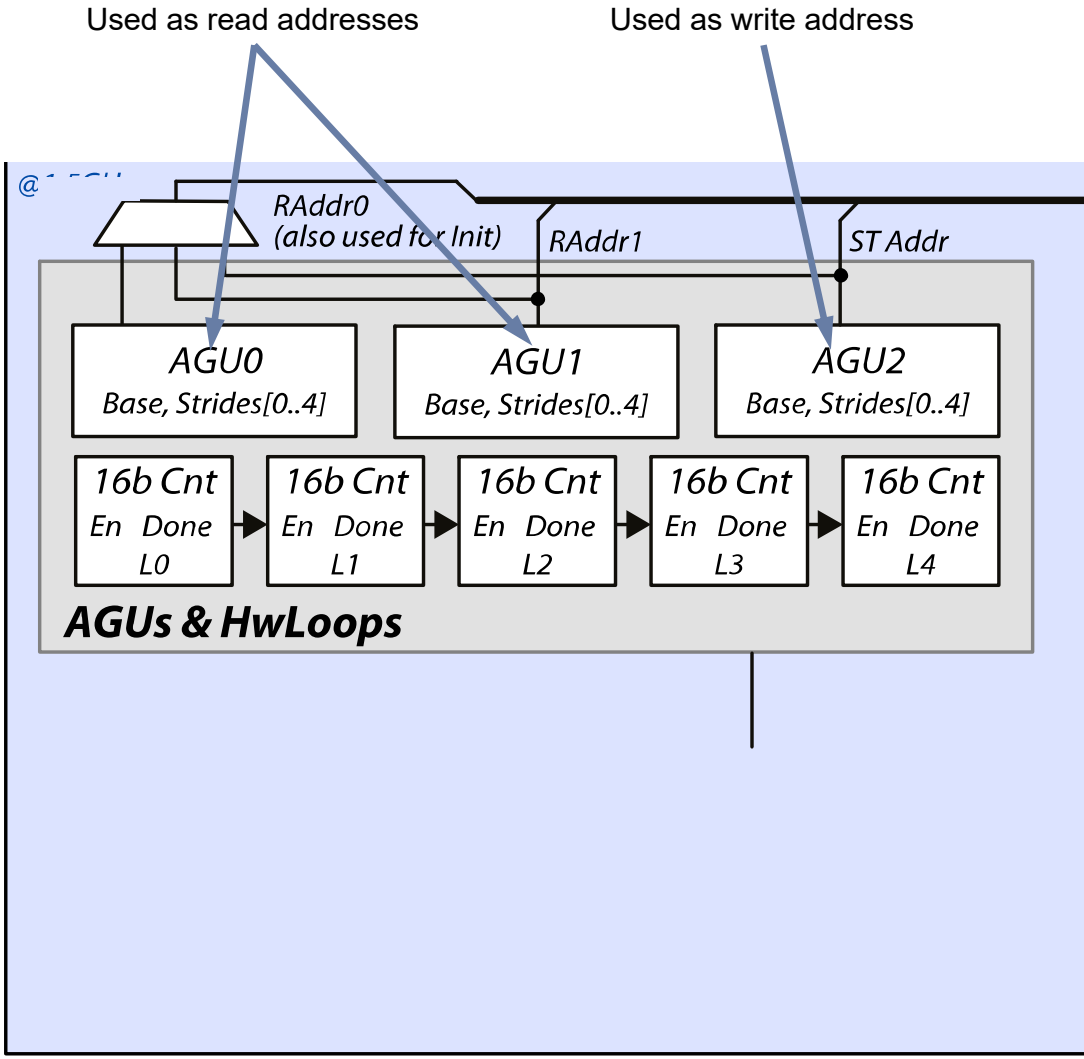
Architecture FMAC




FMAC
partial carry-save (PCS)
arithmetic with 2 segments
RD0
0
RD1
1
+/-
Accu Reg
PCS Norm
ReLu
Alu Reg
Cmd Fifo
5
Comp
Datapath Ctrl
16b Index Counter
0
1
FPU

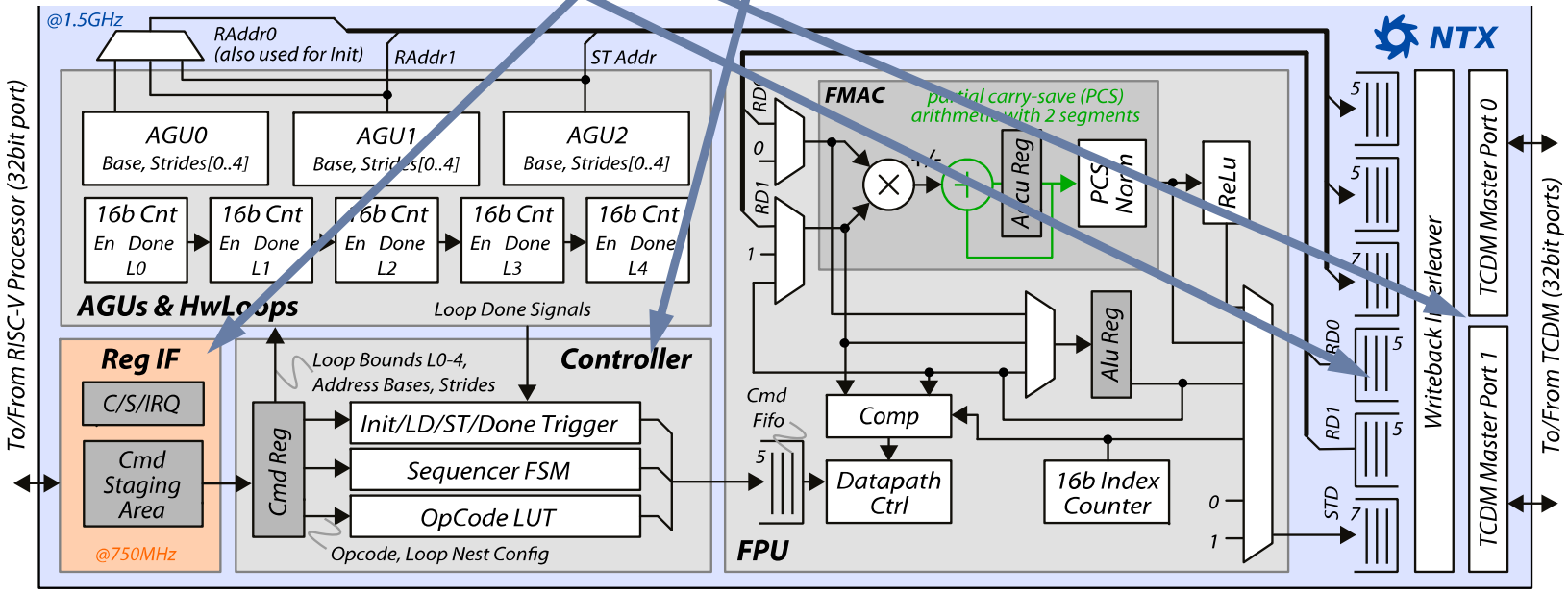
Architecture Address Generation

- 5 nested hardware loop counters
  - 16 bit counter register
  - Configurable number of iterations
  - Once last iteration reached:
    - Reset counter to 0
    - Enable next counter for one cycle
- 3 address generation units
  - 32 bit address register
  - Each has 5 configurable strides, one per loop
  - One stride added to register per cycle
  - Stride corresponds to the highest enabled loop
- Allows for complex address patterns



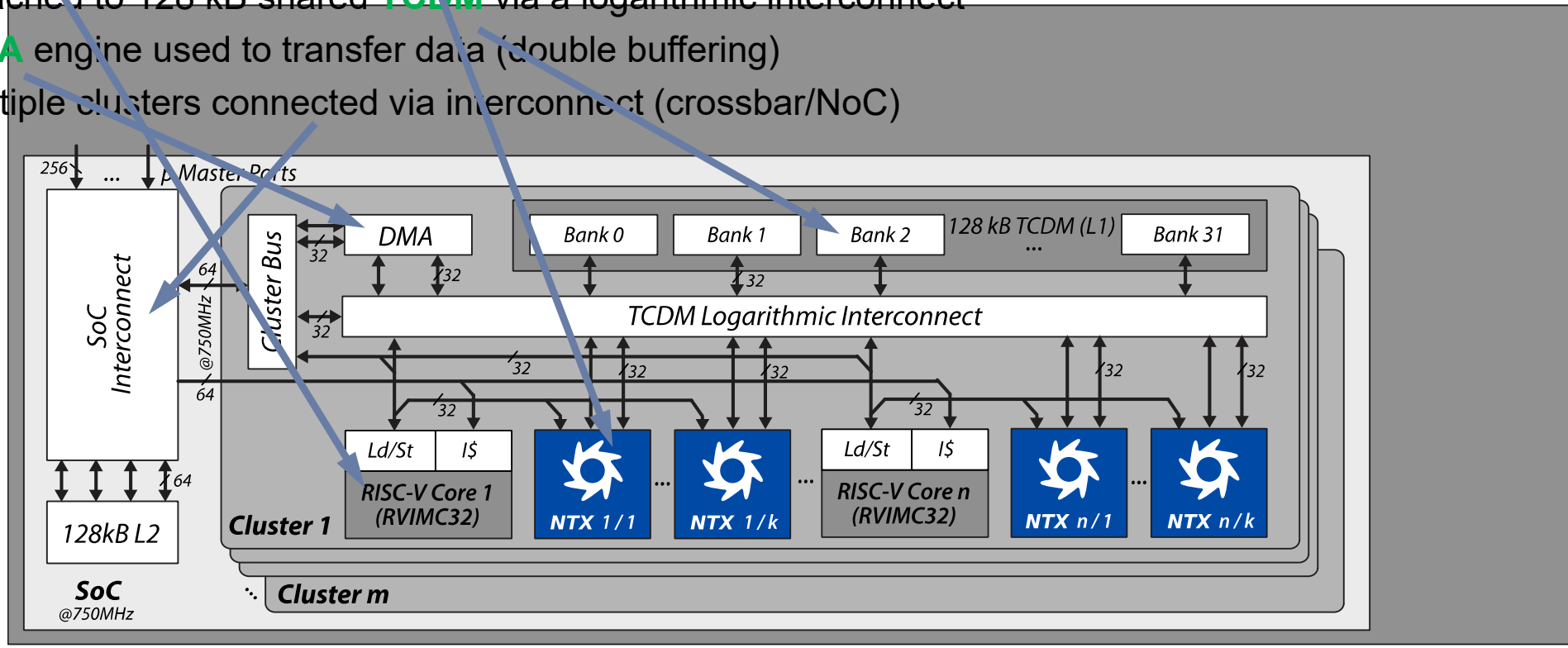
Architecture Coprocessor

- Processor configures operation via **memory-mapped registers**
- Controller issues AGU, HWL, and FPU micro-commands based on configuration
- Reads/writes data via **2 memory ports** (2 operand and 1 writeback streams)
- FIFOs help buffer data path and memory latencies



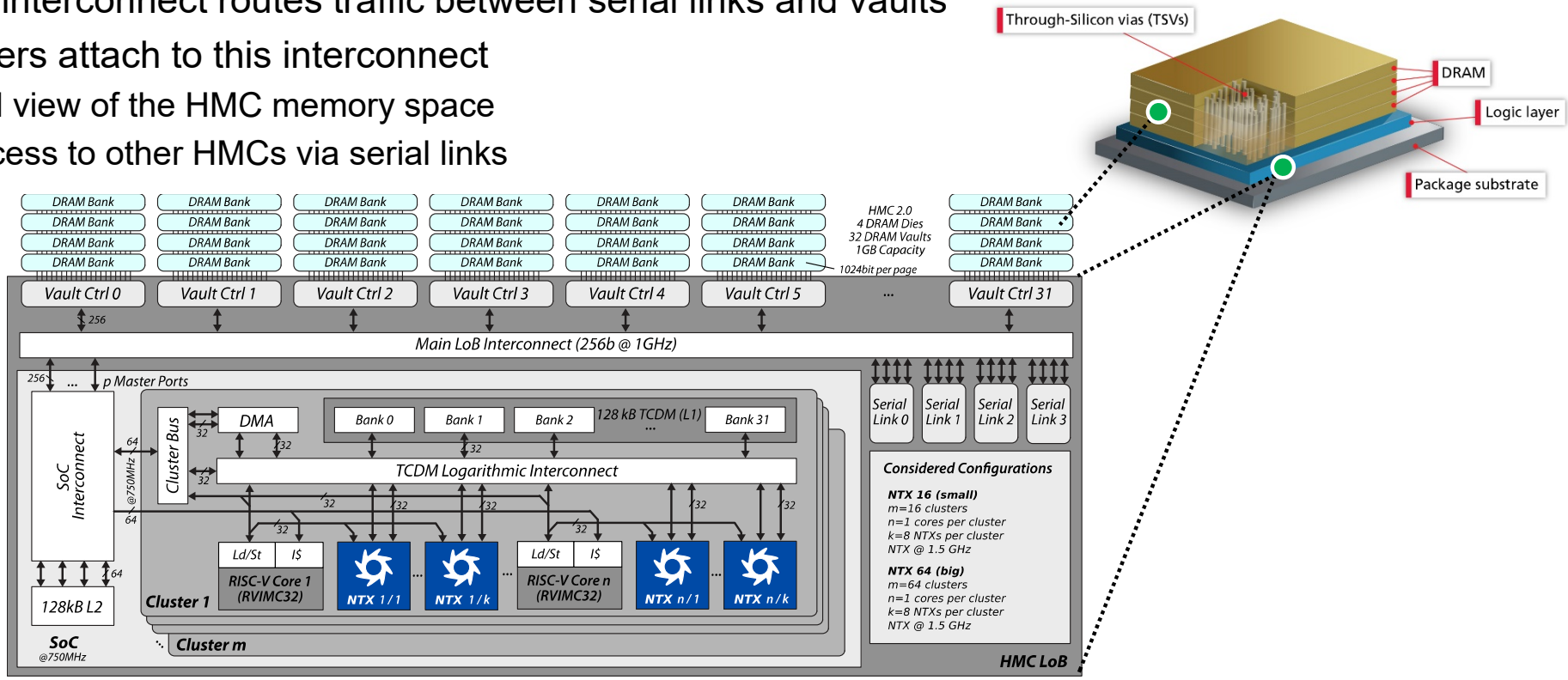
Architecture Processing Cluster

- **1 processor** core controls **8 NTX** coprocessors
- Attached to 128 kB shared **TCDM** via a logarithmic interconnect
- **DMA** engine used to transfer data (double buffering)
- Multiple clusters connected via interconnect (crossbar/NoC)



Architecture HMC Integration

- HMC is split into independent vaults (DRAM controllers)
- Main interconnect routes traffic between serial links and vaults
- Clusters attach to this interconnect
  - Full view of the HMC memory space
  - Access to other HMCs via serial links



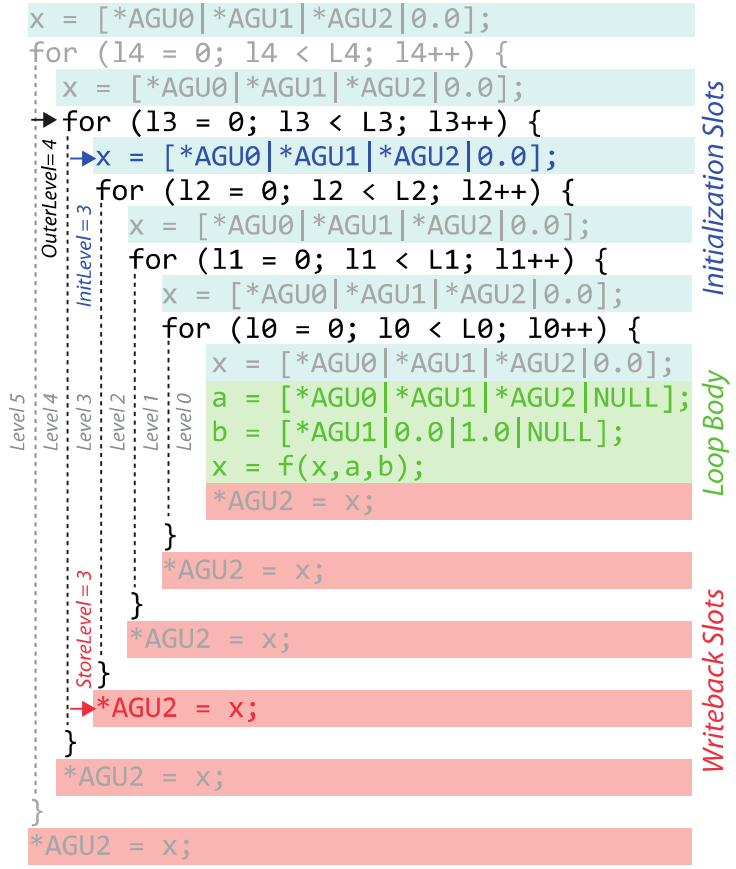
Programming Model Loops

- Up to 5 nested loops can be offloaded to NTX
  - Loops should describe a reduction for best performance
  - Covers convolutions, fully connected layers, and more
- Accumulator initialization and writeback is configurable
- For example a DNN convolution:

Perform outermost loop level on processor core.

```
for (int k = 0; k < K; ++k)
for (int n = 0; n < N; ++n)                    Level 4
for (int m = 0; m < M; ++m) {                  Level 3
    float a = b[k];                            Init Level = 3
    for (int d = 0; d < D; ++d)                Level 2
    for (int u = 0; u < U; ++u)                Level 1
    for (int v = 0; v < V; ++v) {              Level 0
        a += x[d][n+u][m+v] * w[k][d][u][v];
    }
    y[k][n][m] = a;                            Store Level = 3
}
```

NTX



Programming Model Tiling

- Cluster has limited memory (~128 kB)
- DNN data sets are usually multiple GB
- **Tile** input data into chunks that fit in 128 kB
- Use **double buffering** to hide latency while NTX are processing current chunk
  - Write back last iteration's result
  - Preload next iteration's input data
- NTX run independently; processor free to orchestrate data movement with the DMA

- Consider the tiled DNN convolution:

```
for (int tk = 0; tk < TK; ++tk)
for (int tn = 0; tn < TN; ++tn)
for (int tm = 0; tm < TM; ++tm) {
  // load tile inputs x, w, b with DMA
  for (int k = 0; k < K; ++k)
  for (int n = 0; n < N; ++n)
  for (int m = 0; m < M; ++m) {
    float a = b[k];
    for (int d = 0; d < D; ++d)
    for (int u = 0; u < U; ++u)
    for (int v = 0; v < V; ++v) {
      a += x[d][n+u][m+v] * w[k][d][u][v];
    }
    y[k][n][m] = a;
  }
  // store tile outputs y
}
```

Iterate over tiles of the input data

Iterate over pixels in the current tile

Perform convolution for current pixel

C++ API Example

Tiled convolution:

```
for (int tk = 0; tk < TK; ++tk)
for (int tn = 0; tn < TN; ++tn)
for (int tm = 0; tm < TM; ++tm) {
  load_tile(x, w, b);
  for (int k = 0; k < K; ++k)
  for (int n = 0; n < N; ++n)
  for (int m = 0; m < M; ++m) {
    float a = b[k];
    for (int d = 0; d < D; ++d)
    for (int u = 0; u < U; ++u)
    for (int v = 0; v < V; ++v) {
      a += x[d][n+u][m+v] * w[k][d][u][v];
    }
    y[k][n][m] = a;
  }
  store_tile(y);
}
```

Tiled convolution with NTX:

```
ntx_api ntx;
dma_api dma;
ntx.cfg_loops(5, {N,M,D,U,V}, ...);
for (int tk = 0; tk < TK; ++tk)
for (int tn = 0; tn < TN; ++tn)
for (int tm = 0; tm < TM; ++tm) {
  dma.start_read(x, w, b);
  for (int k = 0; k < K; ++k) {
    ntx.cfg_ptrs(x, &w[k], &y[k]);
    dma.wait_read();
    ntx.issue_cmd(ntx_api::MAC);
  }
  ntx.wait_ready();
  dma.start_write(y);
  swap_buffers();
}
```

Configure loop bounds once for the entire kernel

Start reading input data

Point NTX at the address of the input data

Wait for the input data to be loaded (overlaps with previous NTX computation)

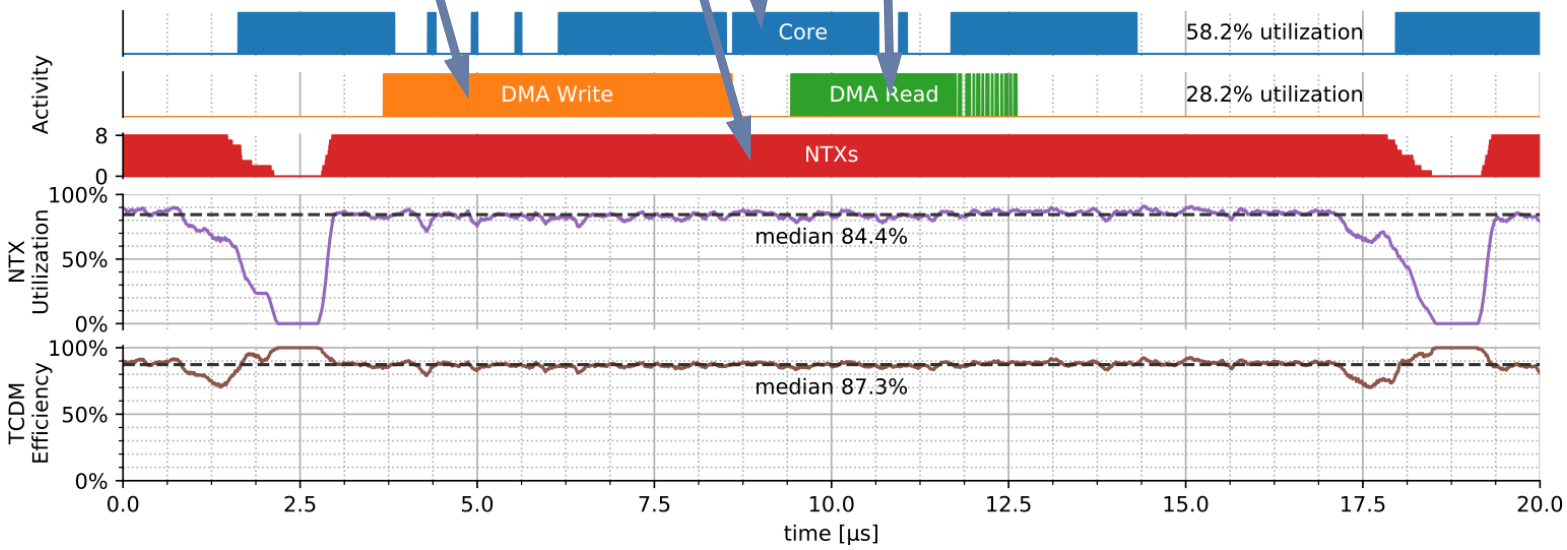
Start next computation

Wait for computation to complete

Start writing back output data

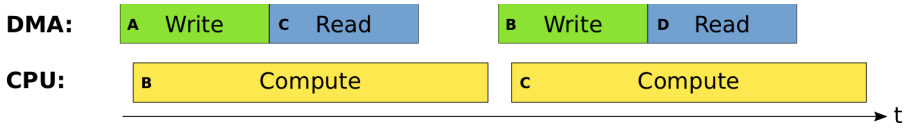
Execution Sample

- All 8 NTX perform the main computation
- DMA writes back results of last computation and reads inputs for next
- Processor core orchestrates operation, computes addresses, pads input data
  - NTXs require no control once started
  - DMA is capable of 2D transfers; core issues multiple small transfers for 3D/4D tensors



Stencil Examples

- Stencils map very well to NTX's loops
- Process input/output in tiles; overlap data movement and computation



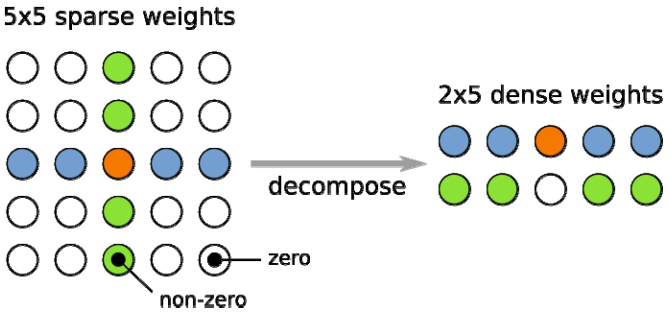
- Example: Dense Stencil in 2D
  - Similar to the convolution
  - Multiply pixel's neighbourhood with weights

- Example: Discrete Laplace Operator in 2D
  - Stencil has a star shape, i.e. it's sparse
  - Decompose into smaller, dense computations
  - Perform computation in "passes"

Dense Stencil in 2D:

```
for (int tn = 0; n < TN; ++tn)
for (int tm = 0; m < TM; ++tm) {
    dma.load(x, w);
    ntx.fmac(y, x, w);
    dma.store(y);
}
```

Discrete Laplace Operator in 2D:




```
for (int tn = 0; tn < TN; ++tn)
for (int tm = 0; tm < TM; ++tm) {
    dma.load(x, w);
    ntx.fmac(y, x, w[0], axis=0);
    ntx.fmac(y, x, w[1], axis=1);
    dma.store(y);
}
```

Implementation in 22nm FD-SOI

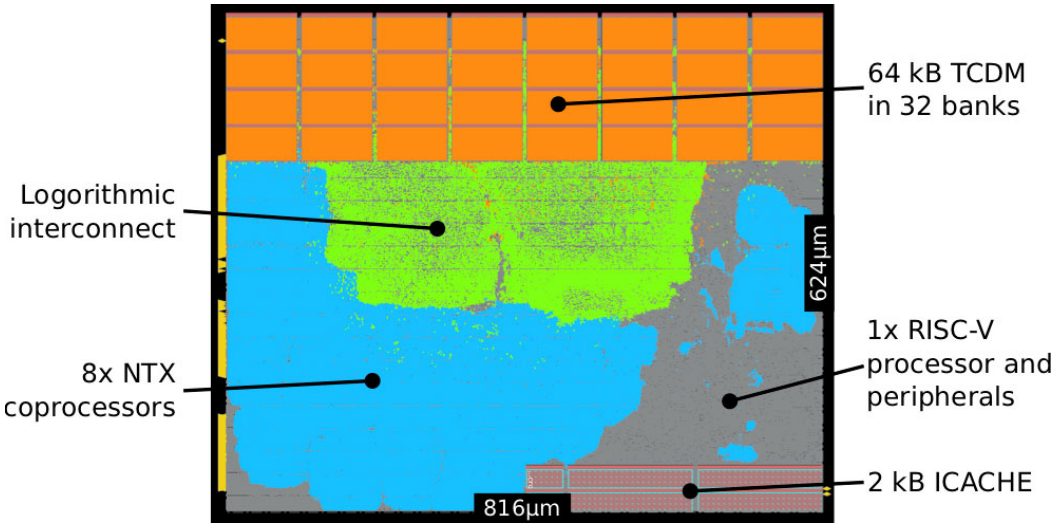
- We have taped out NTX in Globalfoundries' 22FDX technology
- Chips are back and being measured as we speak
- Paper presents post-layout estimates for a single cluster

- NTX runs at up to **1.25 GHz**
- Compute of **20 Gflop/s**
- Bandwidth of **5 GB/s**
- At **9.3 pJ/flop** and using only **0.51 mm2**

- Scale up by replicating cluster

Table I
FIGURES OF MERIT OF ONE NTX CLUSTER IMPLEMENTED IN 22FDX.

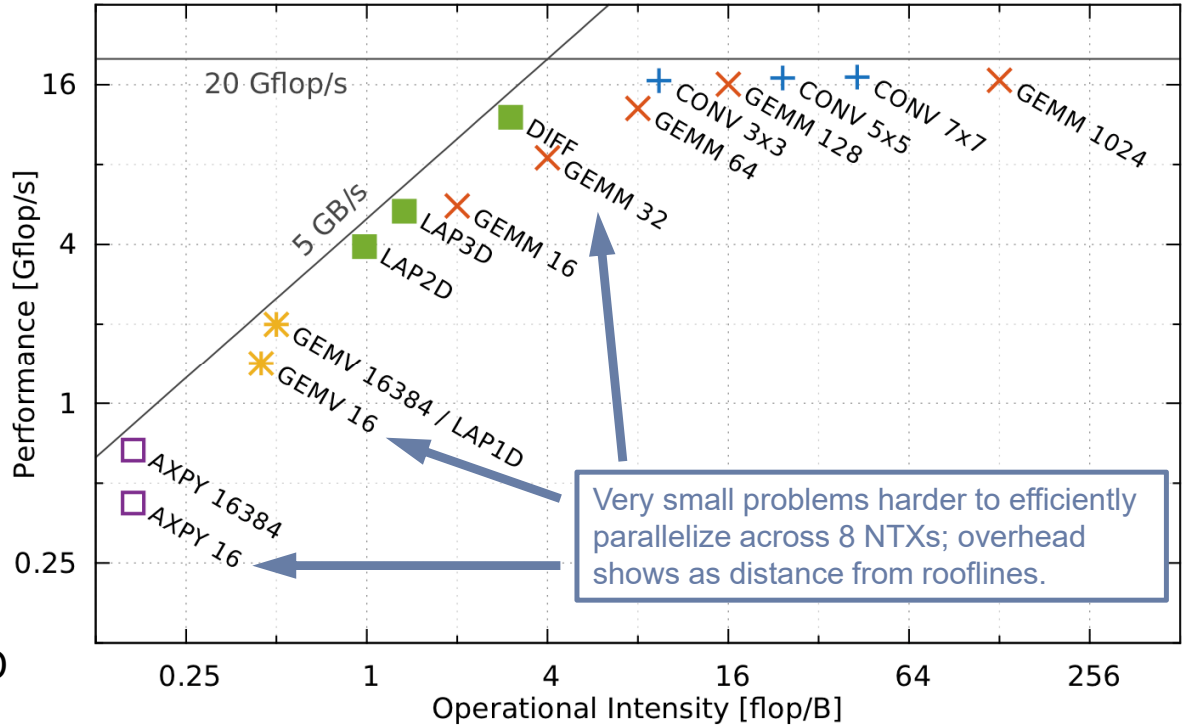
Processors:	1 RISC-V 8 NTX	**Memory:**	64 kB TCDM 2 kB ICACHE
Frequency:	1.25 GHz NTX 625 MHz Cluster	**Area:**	0.51 mm^2 59% density
Peak Perf.:	20 Gflop/s 5 GB/s	**Power:**	186 mW
		Efficiency:	108 Gflop/s W 9.3 pJ/flop



Roofline

- NTX achieves high utilization of available bandwidth and compute
- We investigate a range of different kernels:

- Linear Algebra
  - Mat-Mat product (GEMM)
  - Mat-Vec product (GEMV)
  - Vector sum (AXPY)
- Stencils
  - Discrete Laplace Operator in 1D/2D/3D
  - Diffusion
- Deep Learning



Von Neumann Bottleneck

- NTX helps alleviate the von Neumann bottleneck
  - No explicit load/store instructions
  - No explicit address calculation instructions
- Simple example: Dot product over 1024 elements
- With single RV32IF:
  - **5122** instructions executed
- With single NTX (plus RV32I):
  - **10** instructions executed
  - **1024 idle cycles** while NTX executes (can be used)
- NTX reduces instruction bandwidth by **512x**
  - Even more when using more nested loops
- NTX amortizes single instruction stream over 8 FPUs
  - Data/Inst. bandwidth ratio of **16** (worst case, usually higher)

Single RV32IF Core:

```
Setup      lp.setupi L0, 1024, 5
           flw       ft0, 0(a0)
           flw       ft1, 0(a1)
Hot Loop   fmadd     ft2, ft0, ft1, ft2
           addi      a0, a0, 4
           addi      a1, a1, 4
Writeback  fsw       ft2, 0(a2)
```

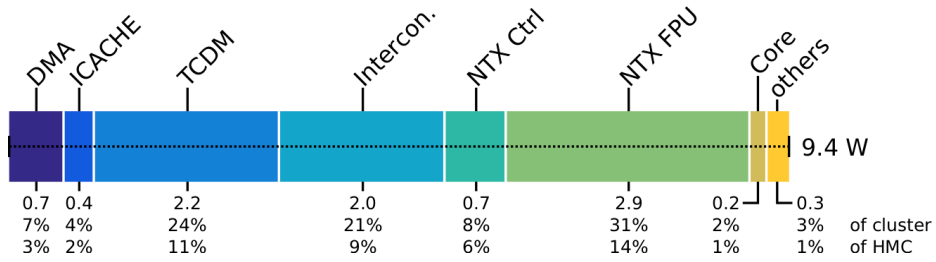
Single NTX:

```
           sw  a0, NTX_AGU0_PTR
           sw  a1, NTX_AGU1_PTR
           sw  a2, NTX_AGU2_PTR
           li  t1, 1024
           sw  t1, NTX_BOUND_L0
Setup      li  t1, 4
           sw  t1, NTX_AGU0_S0
           sw  t1, NTX_AGU1_S0
           li  t1, NTX_MAC_CMD
           sw  t1, NTX_CMD
Idle       wfi
```

Power Breakdown

- NTX dissipates significant fraction of power in its FPU (more is better):
  - **31%** of cluster
  - **14%** of entire HMC
  - Recall: GPU is around **4.8%** [1]
- Compared to NVIDIA Volta GPU [2]:
  - Register file in GPU holds registers and thread-local data
  - Each register read/write is an SRAM access
  - Register and data accesses compete for SRAM

1 Volta SM	8 NTX cl.
64 FPUs	64 FPUs
256 kB RF 128 kB L0 Cache	512 kB TCDM
32-2048 threads	8 threads



Volta Assembly

```
LDS  R2, [R0]
LDS  R3, [R1]
FFMA R4, R2, R3, R2
```

2 mem. acc. (“[…]”)
8 reg. acc.

= 10 SRAM hits total

NTX Pseudocode

```
FMAC accu, [AGU0], [AGU1]
```

2 mem. acc. (“[…]”)
0 reg. acc.
(+ addr. calc for free)

= 2 SRAM hits total

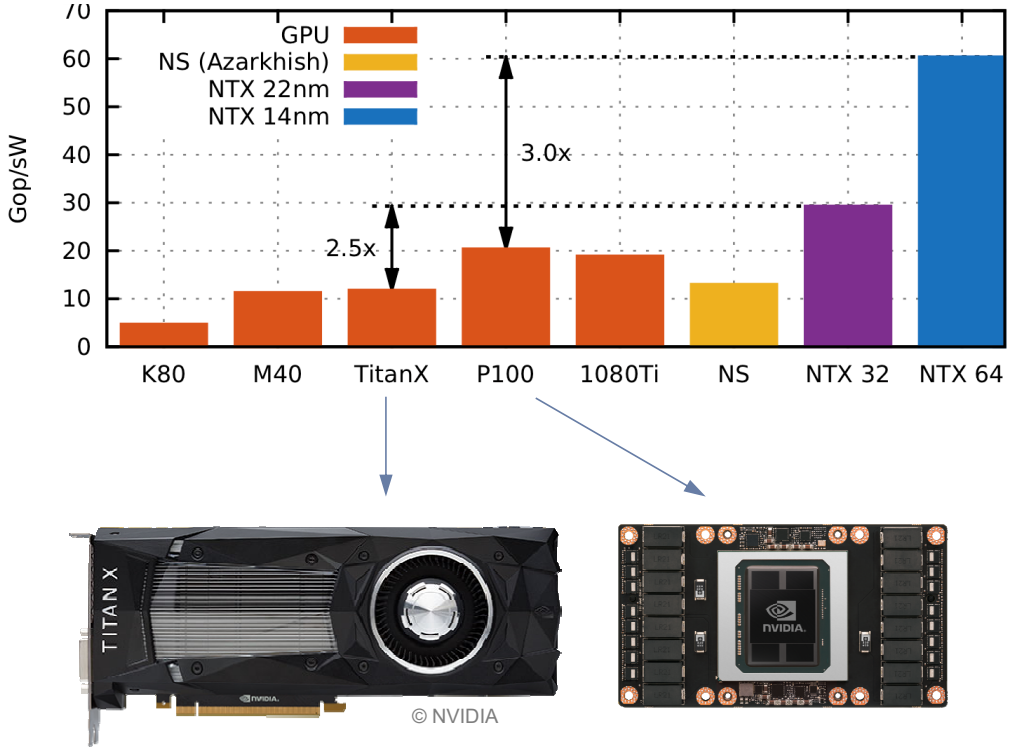
[1] S. Hong et al., “An integrated gpu power and performance model,” in ACM SIGARCH Computer Architecture News, 2010.
[2] Volta Architecture Whitepaper, NVIDIA

Results Energy Efficiency

- How much Gflop/s of compute do we get per W of power?
- Comparison of NTX against GPU in similar technology node

- 22 nm: **2.5x** more vs. Nvidia TitanX
- 14 nm: **3.0x** more vs. Nvidia Tesla P100

- A note on Nvidia V100:
  - Tensor cores operate on float16
  - Real float32 efficiency likely 30 Gflop/Ws
  - 12 nm NTX likely around 2x gain [1]



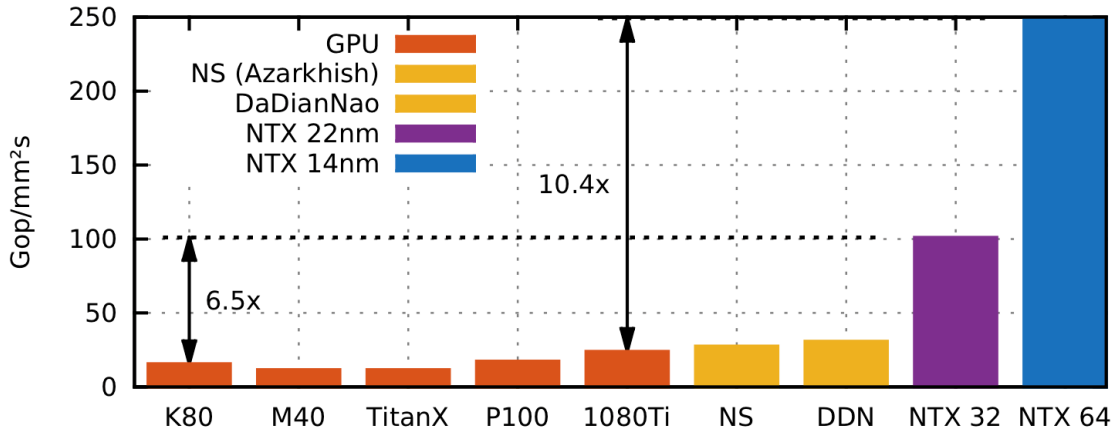
[1] O. Abdelkader et al, "The Impact of FinFET Technology Scaling on Critical Path Performance under Process Variations," at ICEAC, 2015.

Results Deployed Silicon

- How much Gflop/s of compute do we get per mm^2 of silicon area?
- Comparison of NTX against GPU in similar technology node

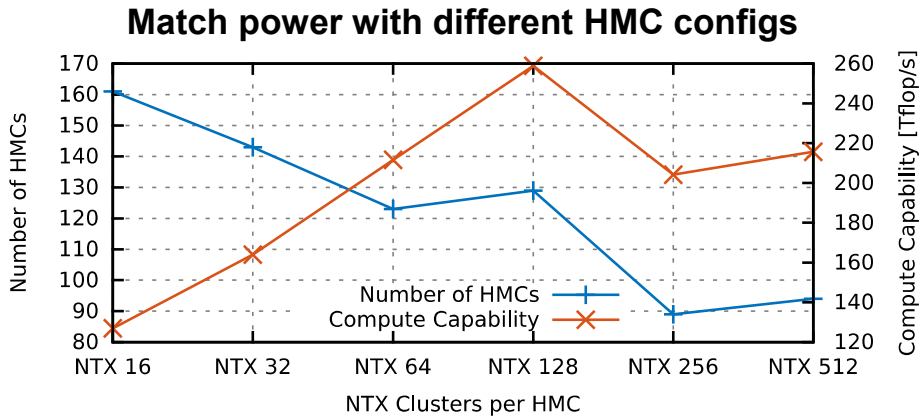
- 22nm: **6.5x** more vs. Nvidia K80
- 14nm: **10.4x** more vs. Nvidia 1080Ti

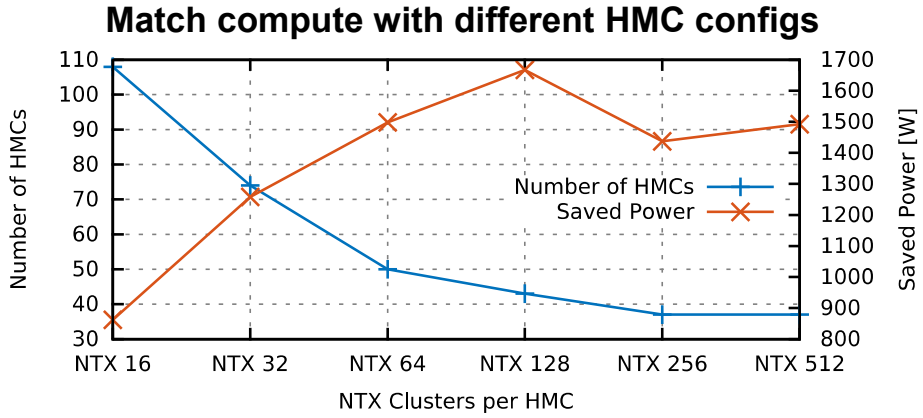
- GPU dies are huge (>500 mm^2)
- NTX fits easily into HMC
- Silicon in HMC manufactured anyway, but is unused; virtually zero additional cost



Results Data Center

- Match an Nvidia **DGX-1** with HMCs [1]
  - Two Intel Xeon CPUs, eight Tesla P100
  - 3.2 kW total, 2.4 kW due to GPU
  - 84.8 Tflop/s of compute
- Scenario 1: Match **3.2 kW** power envelope
  - **3.1x** increase in compute (258.9 Tflop/s)
  - 129 HMCs, 128 NTX clusters each
- Scenario 2: Match **84.8 Tflop/s** of compute
  - **2.1x** power reduction (1.53 kW)
  - 43 HMCs, 128 NTX clusters each
  - Energy bill: –$1808 per server and year





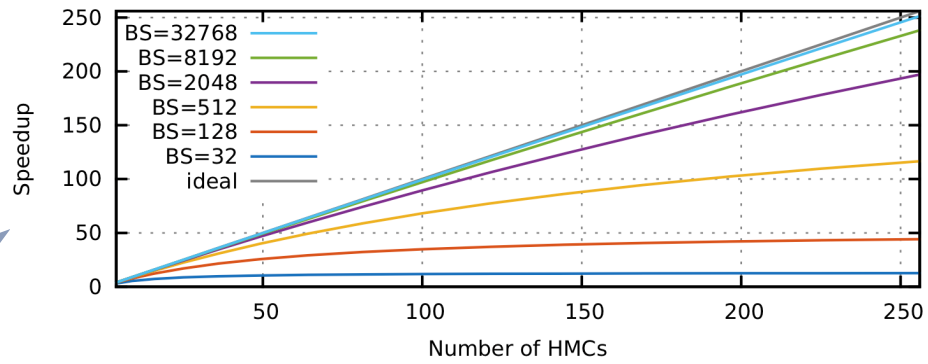
[1] F. Schuiki et al, "Schuiki, Fabian, et al. "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," in *IEEE Transactions on Computers*, 2019.

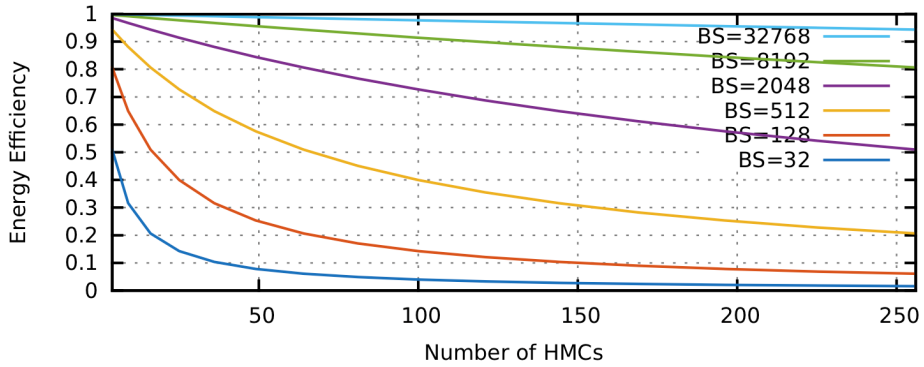
Results Scaling to Multiple HMCs

- Arrange HMCs in a mesh
- Communication with neighbors via serial links
- We investigate the scaling behaviour of DNN training over multiple HMCs
- For example 64 HMCs and batch sizes 8192:

- HMCs provide almost ideal speedup:
  - **62.8x** speedup
  - **98%** parallel efficiency

- HMCs lose little energy to communication:
  - **94.3%** energy efficiency





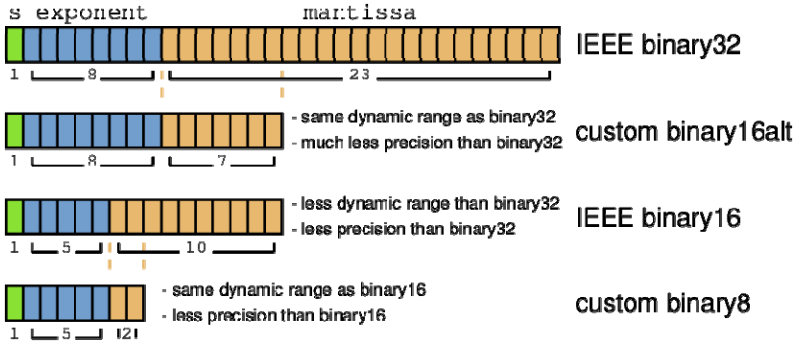
[1] F. Schuiki et al, "Schuiki, Fabian, et al. "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," in *IEEE Transactions on Computers*, 2019.

Future Work

- Address Generator Extension
  - NTX's address generator likely applicable to more kernels
  - FFTs, linear algebra decompositions/factorizations
  - Searches? Sorting? Graphs?

- Transprecision Computing
  - Save precious DRAM bandwidth
  - Custom number formats
    - Use float8, float16
    - Logarithmic numbers?
  - On-the-fly data type conversion in DMA

PRECOMP
Open Transprecision Computing



- Automated Mapping of Kernels
  - Starting from Compute Graph, e.g. TensorFlow

Thank you!