Hardware Accelerators for Deep Neural Networks

Energy Efficiency of Hardware Acceleration

Olivier Sentieys

Univ. Rennes, Inria, IRISA

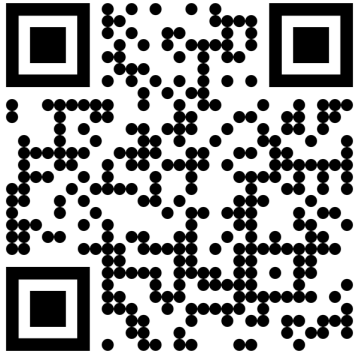
olivier.sentieys@irisa.fr

https://gitlab.inria.fr/sentieys/dnn_acc

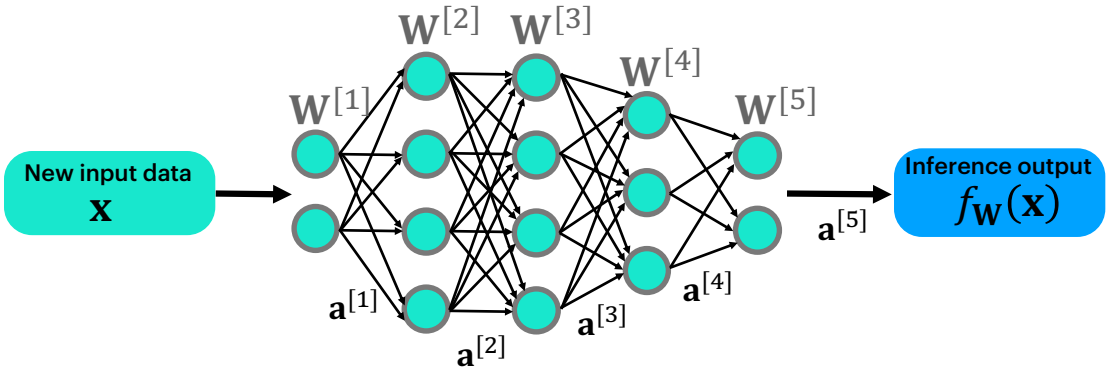
What will you learn in this course?

- Principle and design of DNN Accelerators
  - Energy efficiency of hardware accelerators
  - Speeding up the GEMM kernel
  - Designing hardware accelerators for GEMM/CONV
  - Available accelerators for DNNs
  - Computing at the right precision

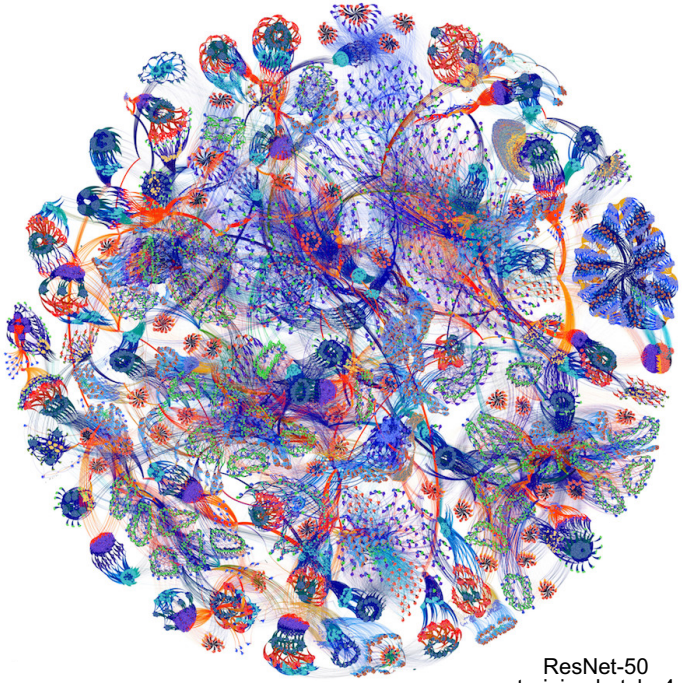
https://gitlab.inria.fr/sentieys/dnn_acc



Complexity Issues of Deep Neural Networks



- Two main tasks
  - training - determine set of network parameters to solve a *task* (minimize a loss on a *training set*)
  - inference - given an input, compute (forward propagate) using the trained network

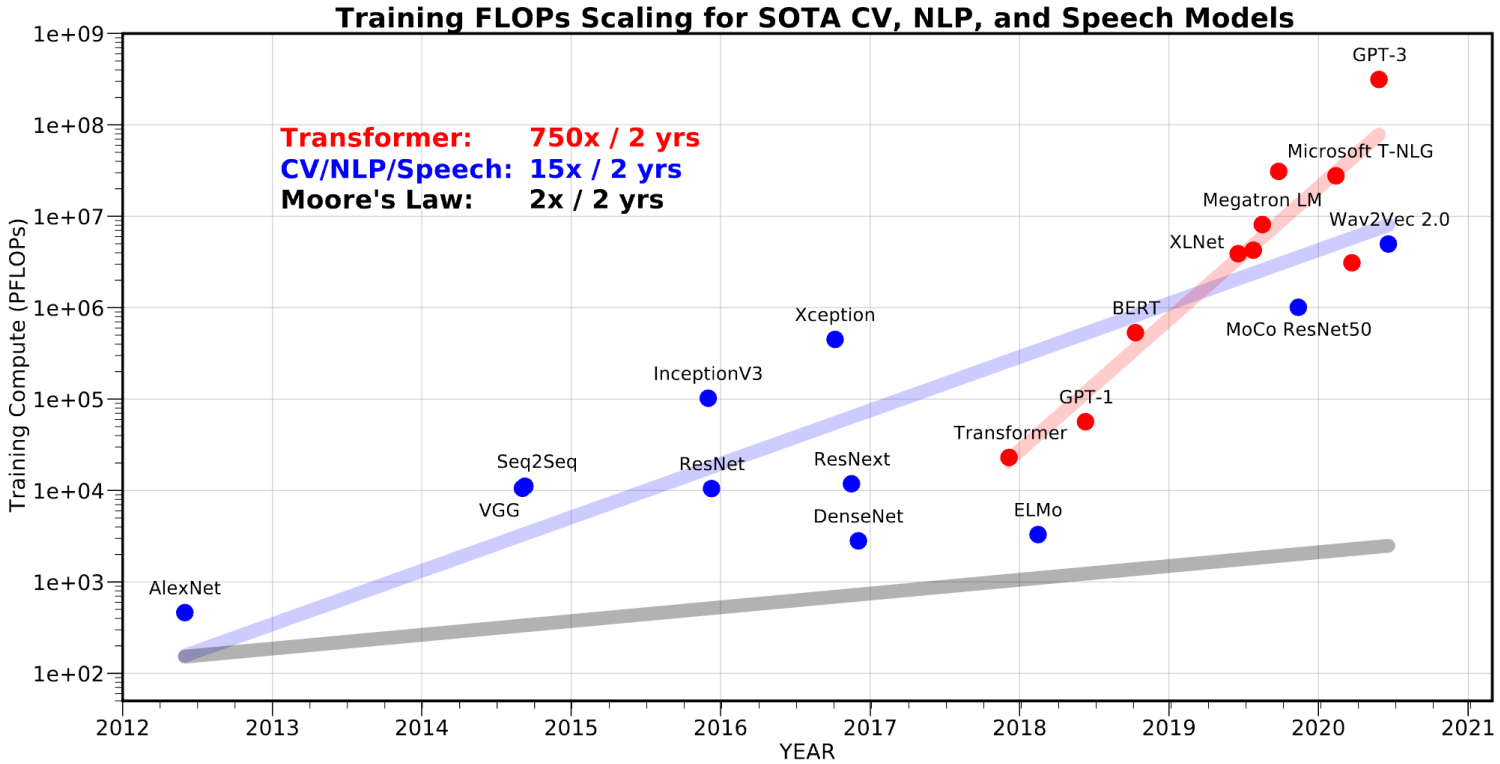


ResNet-50
training batch=4

Poplar® graph

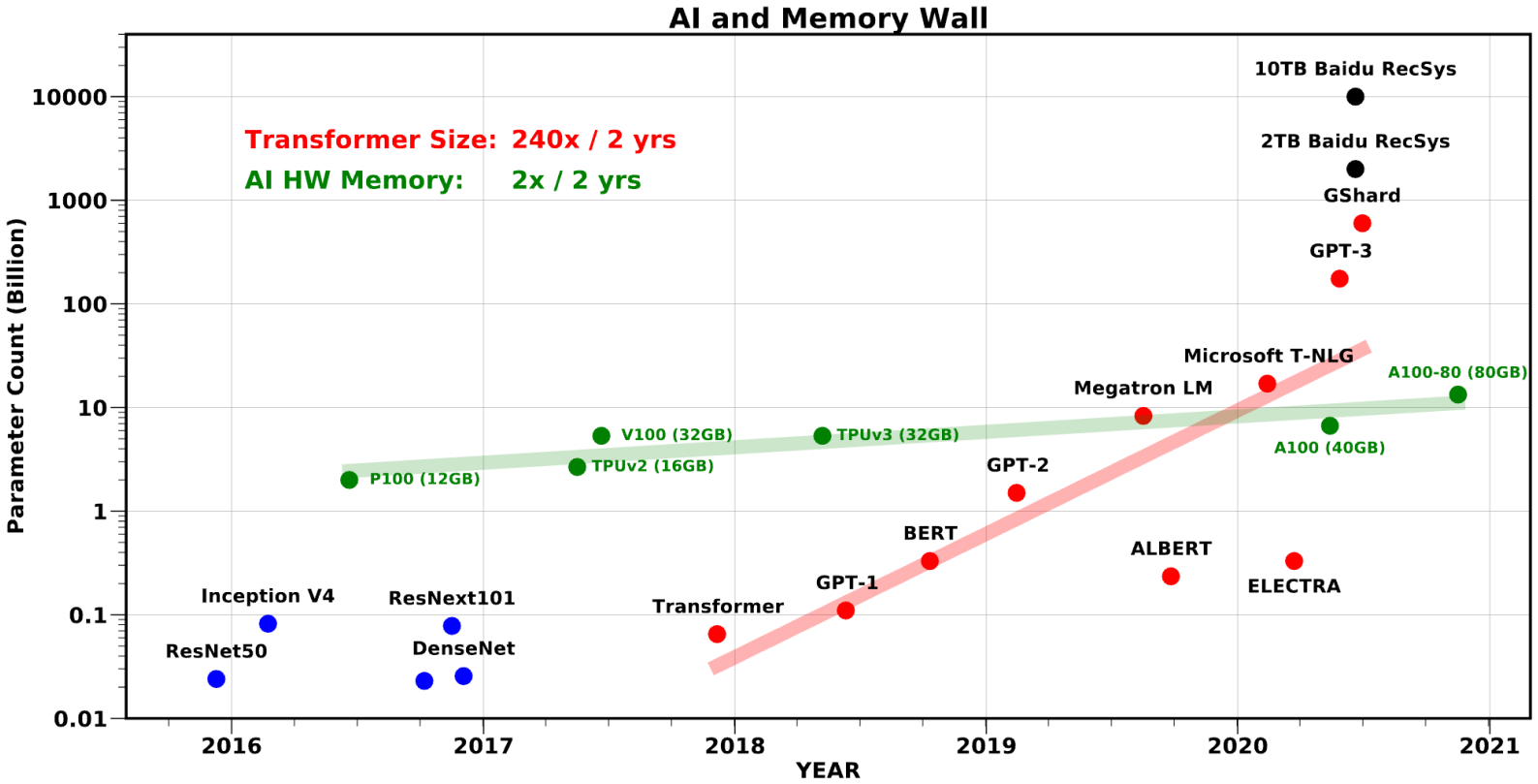
Computing Demand of AI

- is higher than what computer architectures can bring

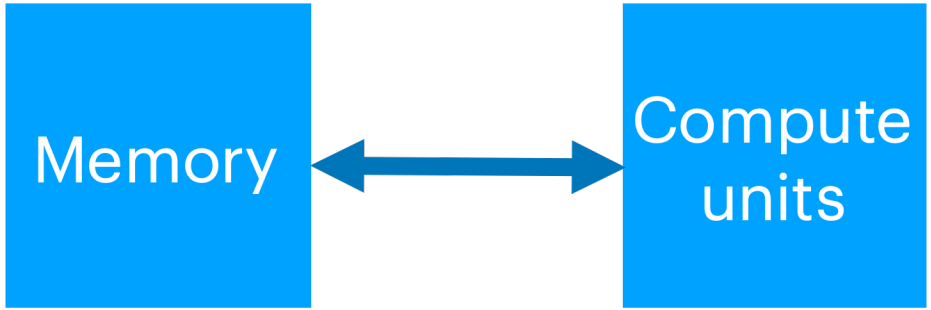


Evolution of the Number of Parameters

- is much higher than available (on-chip) memory capacity



Memory Bottleneck



Data movement

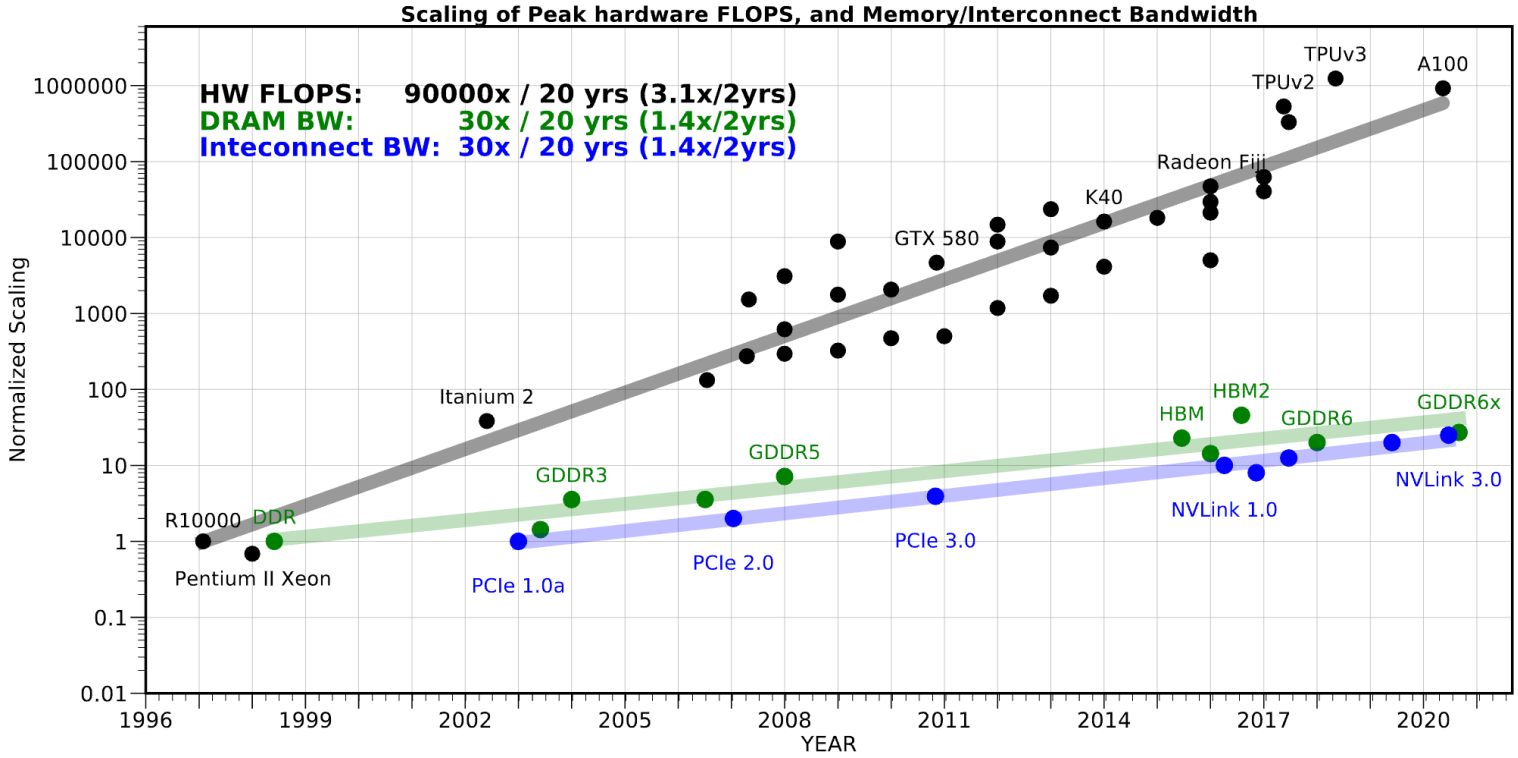
- move input data & model from memory to compute units
- send partial results back to memory

Computations

- vector/matrix manipulations
- done on CPU, GPU, or custom accelerators (e.g., FPGA, ASIC)

Evolution of Bandwidth

- is much slower than FLOPS



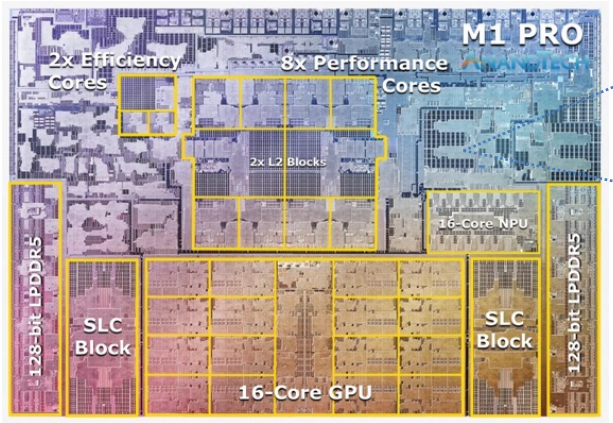
On the Computer Architecture Side

The Hardware Lottery

Sarah Hooker, The Hardware Lottery, Communications of The ACM, 2021

Silicon Technology Evolution

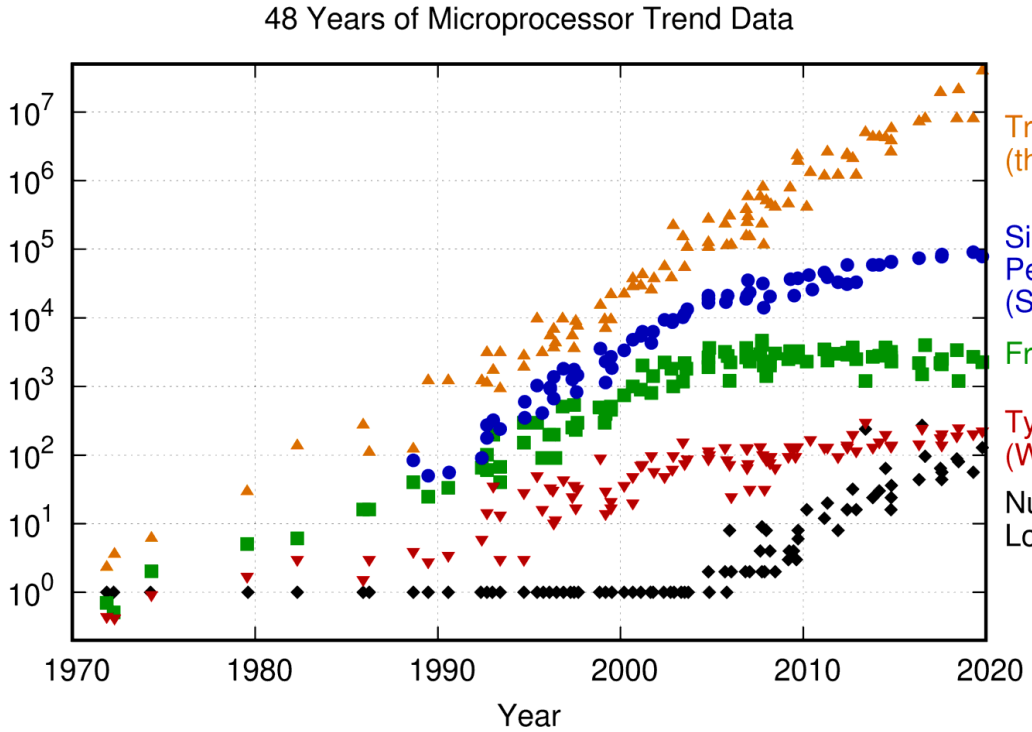
- Now several billions or transistors!
  - Apple M1: 33 B.Tr, 5nm, 2.5cm2



HPE Frontier
1.6 Exa FLOPS

- Amazing compute progress
  - 12 orders of magnitude performance i
  - A supercomputer in every body's pocket

The Many Walls of Computer Architecture



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

More transistors

Saturating perf.

Memory Wall

Microarchitecture

Power Wall

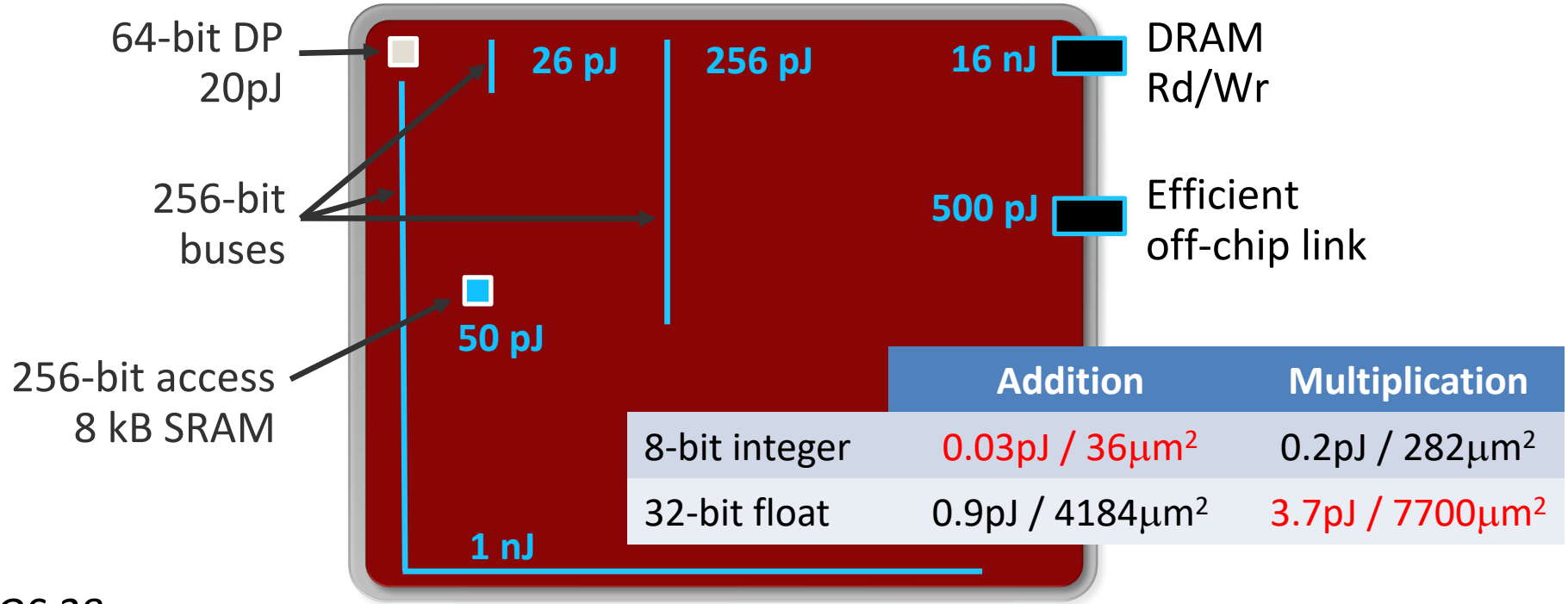
limit of ~25W/cm^2

Increasing Complexity

Dark Silicon

Energy Cost in a Chip

- Fetching operands costs more than computing



	Addition	Multiplication
8-bit integer	0.03pJ / 36μm²	0.2pJ / 282μm²
32-bit float	0.9pJ / 4184μm²	3.7pJ / 7700μm²

CMOS 28nm

[after B. Dally, NVIDIA & Stanford]

Energy Efficiency

$$Power = \underbrace{\frac{Operations}{Second}}_{\text{Performance}} \times \underbrace{\frac{Joules}{Operation}}_{\text{Energy Efficiency}}$$

Performance: e.g., Tera op/s (TOPS)

Energy Efficiency: e.g., TOPS/Watt

- Power budget is fixed
- How to increase energy efficiency while maintaining performance?

Improving Energy Efficiency

- Technology?
  - What can advanced technology nodes bring?
  - Dark Silicon Era
- Accelerate
  - Energy advantages of specialized hardware
- Approximate
  - Playing with precision and number representations to reduce energy

- Key message of this course: specialized hardware that computes at the right (lowest) precision

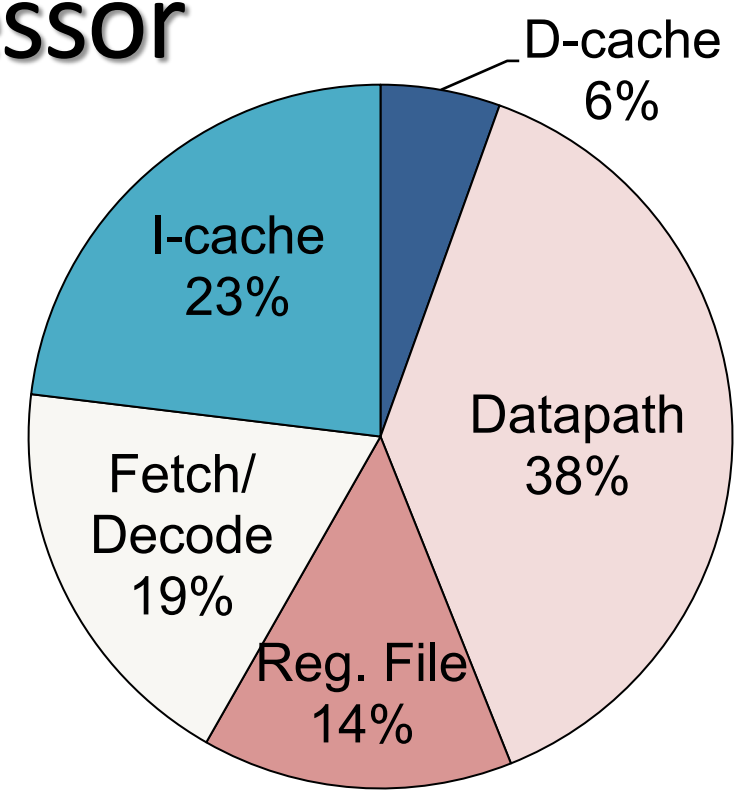
Outline

- Part I: the clear need for specialized hardware
  - Energy efficiency of hardware accelerators
- Part II: accelerating GEMM
  - DNN kernels with a focus on convolution
  - Speeding up the GEMM kernel
  - Designing hardware accelerators for GEMM/CONV
  - Available accelerators for DNNs
- Part III: computing at the right precision
  - A bit of arithmetic
  - Inference and training with low precision

Energy Cost in a Processor

- Operations
  - 32-bit addition: 0.05pJ
  - 16-bit multiply: 0.25pJ
  - 64-bit FPU: 20pJ/op

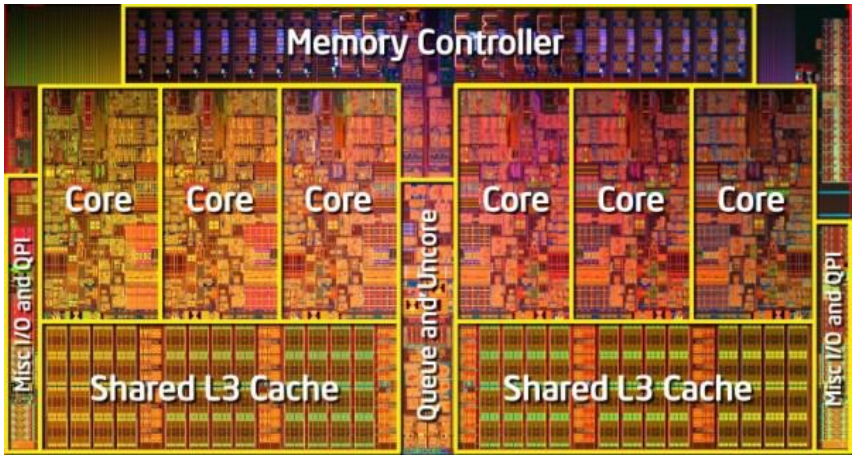
- Instructions
  - fetch, decode, read two operands from RF, execute, write back



General Purpose Processor
91 pJ/instruction

Achieving Higher Performance

- Pushing clock frequency…
- Branch/value prediction
- Cache memory
- In-core parallelism
  - Superscalar
  - Out of order execution
  - VLIW+good compilers



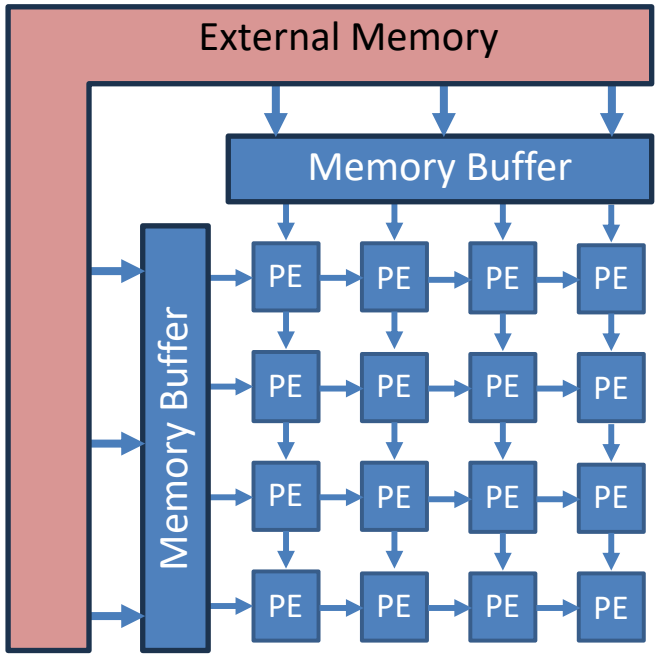
- Multiple cores on a single chip

Pushing for Hardware Acceleration!

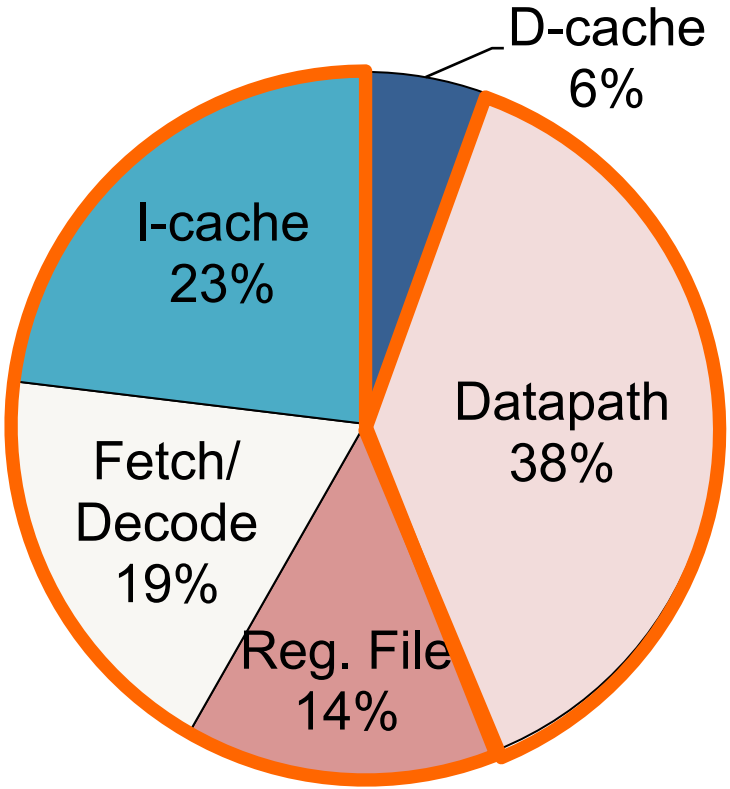
What is a Hardware Accelerator?

- Specialized hardware for a given set of kernels
- With limiting programmability
- Computes just right!

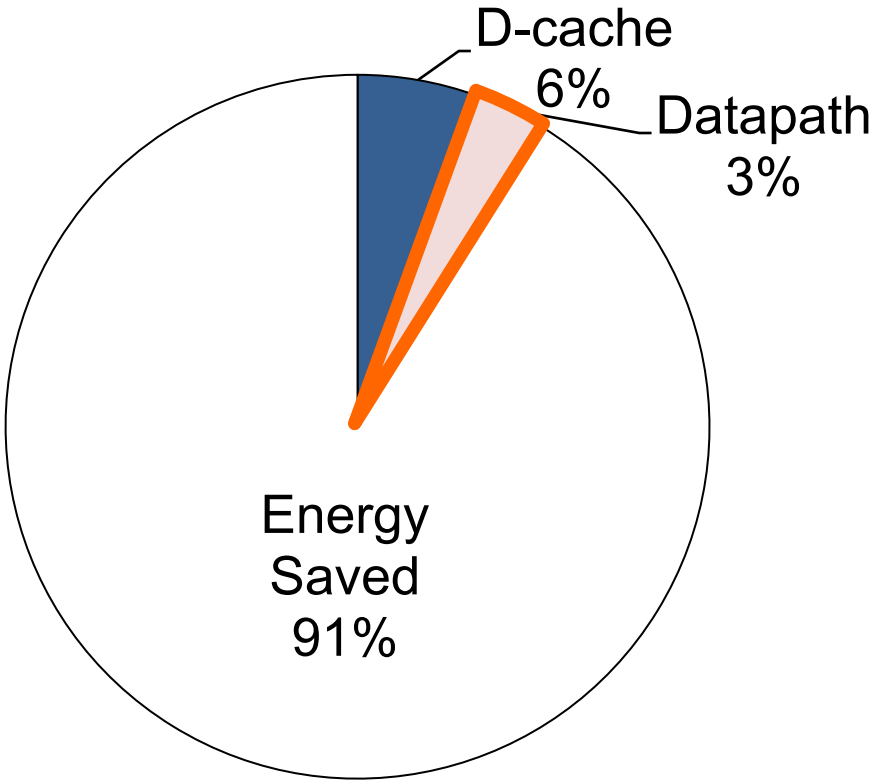
- e.g., Matrix Multiplication



Energy Savings in Specialized HW



GPP: 91 pJ/instr.



Specialized Core: 8 pJ/instr.

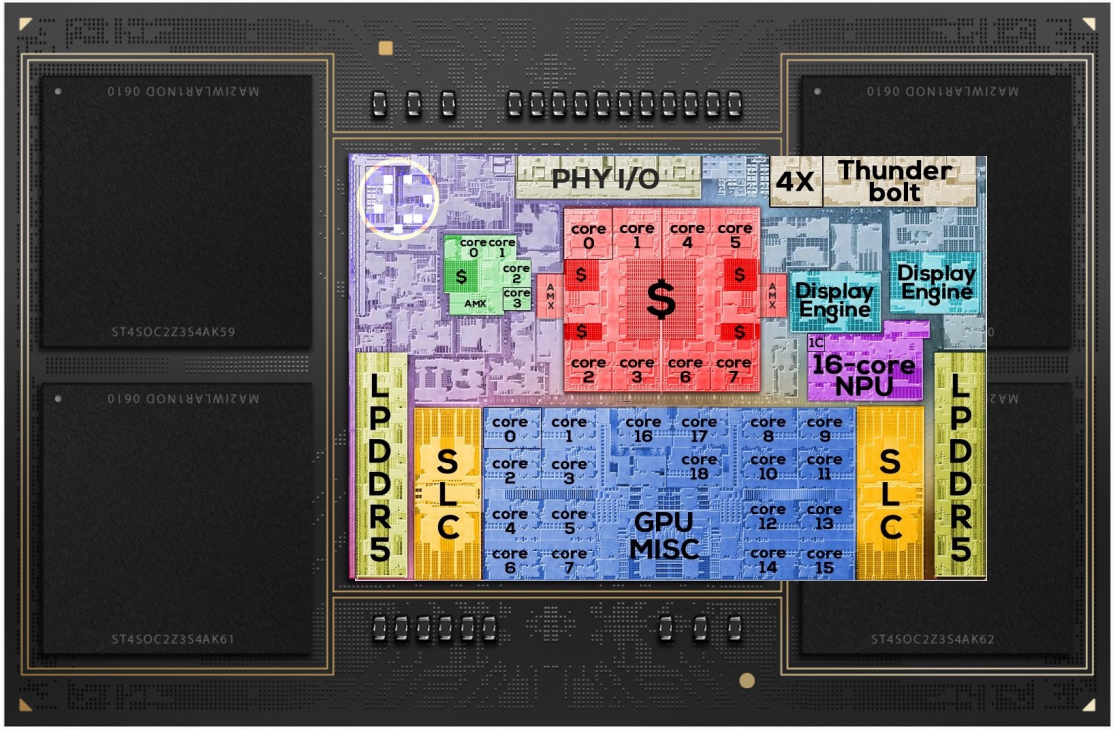
An example: Bitcoin Mining

Type	Model	Mhash/s	Mhash/J	Power (W)
GPP	Intel Xeon X5355 (dual)	22.76	0.09	120
GPP	ARMCortex-A9	0.57	1.14	1.5
GPP	Intel Core i7 3930k	66.6	0.51	130
GPU	AMD 7970x3	2050	2.41	850
GPU	Nvidia GTX460	158	0.66	240
ASIC	AntMiner S1	**180.000**	**500**	360
ASIC	AntMiner S5	1.155.000	**1957**	590
FPGA	Bitcoin Dominator X5000	100	**14.7**	6.8
FPGA	Butterflylabs Mini Rig	25.200	**20.16**	1250

Apple Silicon M2 Max

- 5 nm (TSMC 2G), 40 billion transistors
- 8 performance cores
  - 38 Int MOPS, 56 MFLOPS
  - NEON vector processor
- 4 power-efficiency cores
- Unified memory (32-96 GB LPDDR5-6400) next to the (400 GB/s bandwidth) for GPU&CPU
- 89W (peak CPU+GPU), CPU 36W (peak)

- **High-performance media engine**
- **16 TOPS Neural Engine**
  - **10,000 times the GPU speed for ML tasks**
  - **"power-efficient" (but no reported power figures)**



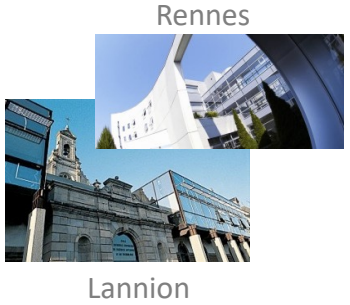
Accelerators for ML

CPU	GPU	FPGA	TPU
Threads SIMD	Massive Threads SIMD HBM	LUTs DSP BRAM	MM Unit BRAM

Key Takeaways

- Energy efficiency requires deeply specialized hardware
  - which also may come with pain from the programmer/designer
- Basic tasks of DNNs are easy to accelerate
  - this course is mainly focused on matrix multiplication
- Number representations and precisions are key techniques
  - also memory access since execution is often memory-bound

TARAN Team at a Glance



Domain-Specific Computers
in the post Moore's law era

Accelerators

Approximate Computing

Resilient Computing

Embracing Technologies

- ~40 people, Rennes and Lannion campuses
- Our focus: hardware specialization and acceleration
  - Energy Efficiency of hardware accelerators
  - Domain-specific architectures, languages and compilers
- Automatically create hardware that is resilient, secured, and computes just right
  - From Sensors to Clouds

Inria

UMR IRISA

TARAN

Outline

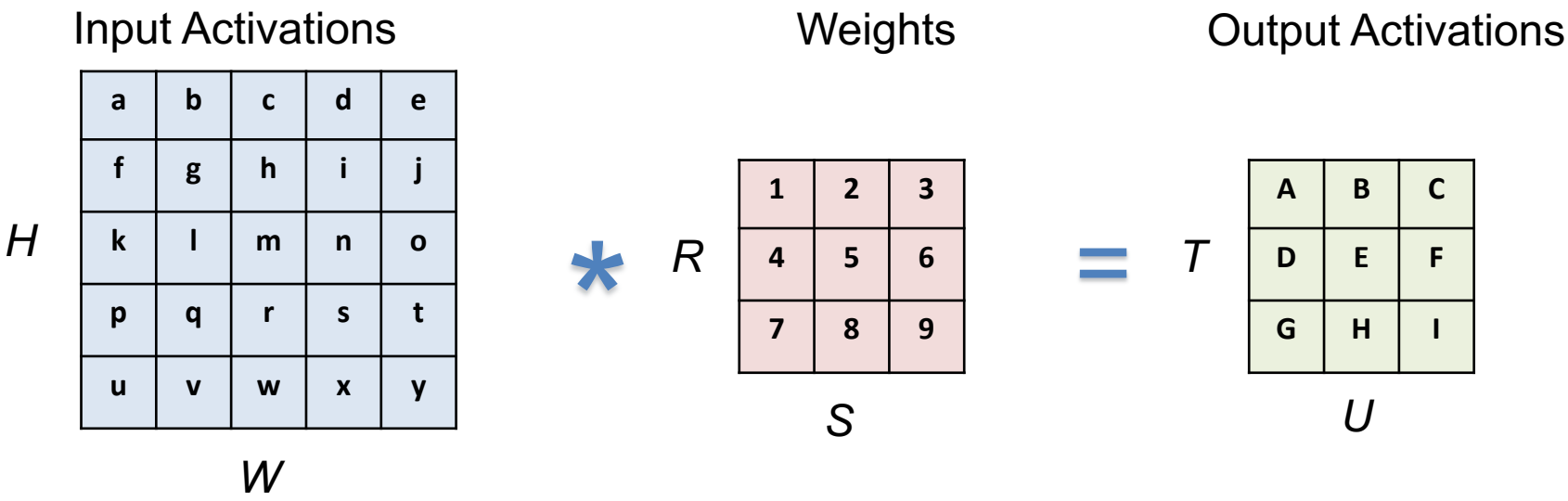
- Part I: the clear need for specialized hardware
  - Energy efficiency of hardware accelerators
- Part II: accelerating GEMM
  - DNN kernels with a focus on convolution
  - Speeding up the GEMM kernel
  - Designing hardware accelerators for GEMM/CONV
  - Available accelerators for DNNs
- Part III: computing at the right precision
  - A bit of arithmetic
  - Inference and training with low precision

Speeding Up GEMM

Efficient Processing of Matrix Multplication

Focus on Convolution Neural Networks (CNN)

2D Convolution

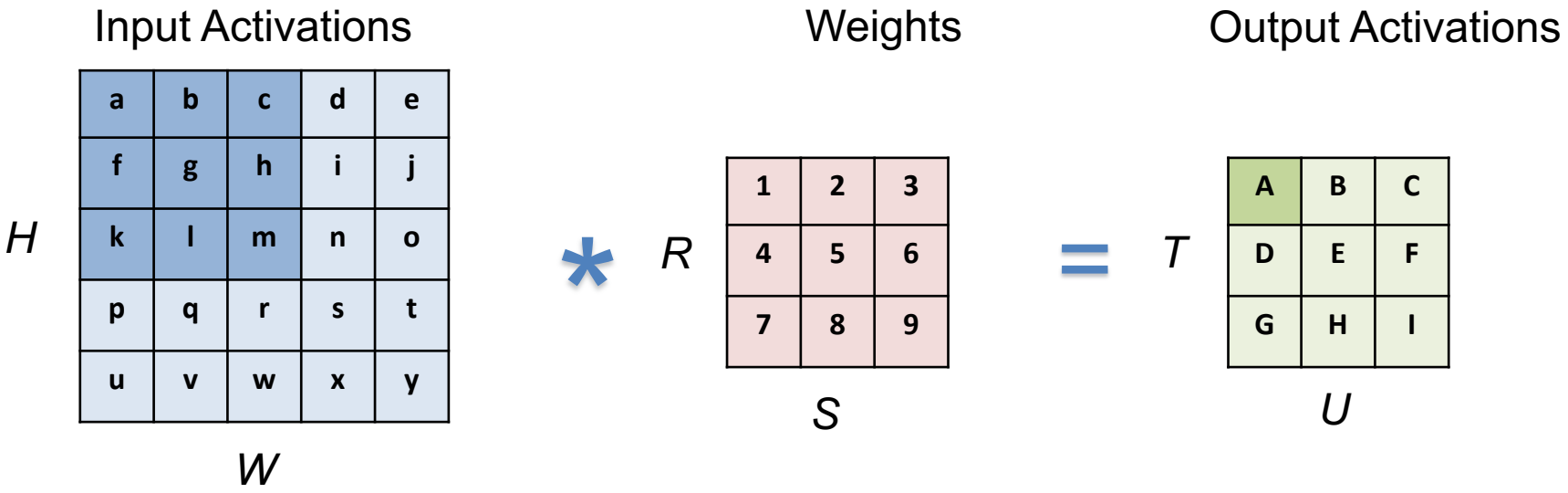


H: Height of Input Activation
W: Width of Input Activation
R: Height of Weight
S: Width of Weight
T: Height of Output Activation
U: Width of Output Activation

Example is with:

- *stride=1*
  # of rows/columns traversed per step
- *padding=0*
  # of zero rows/columns added

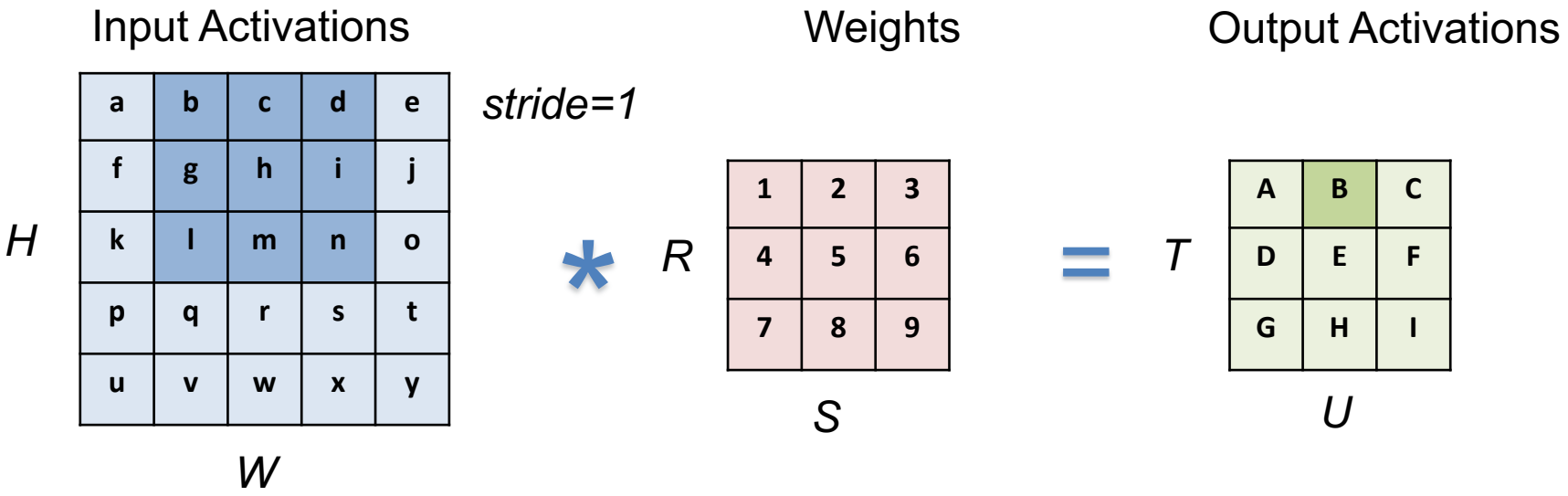
2D Convolution



$$A = a\times1 + b\times2 + c\times3 + f\times4 + g\times5 + h\times6 + k\times7 + l\times8 + m\times9$$

(R×S) Multiply and Accumulate (MAC) operations

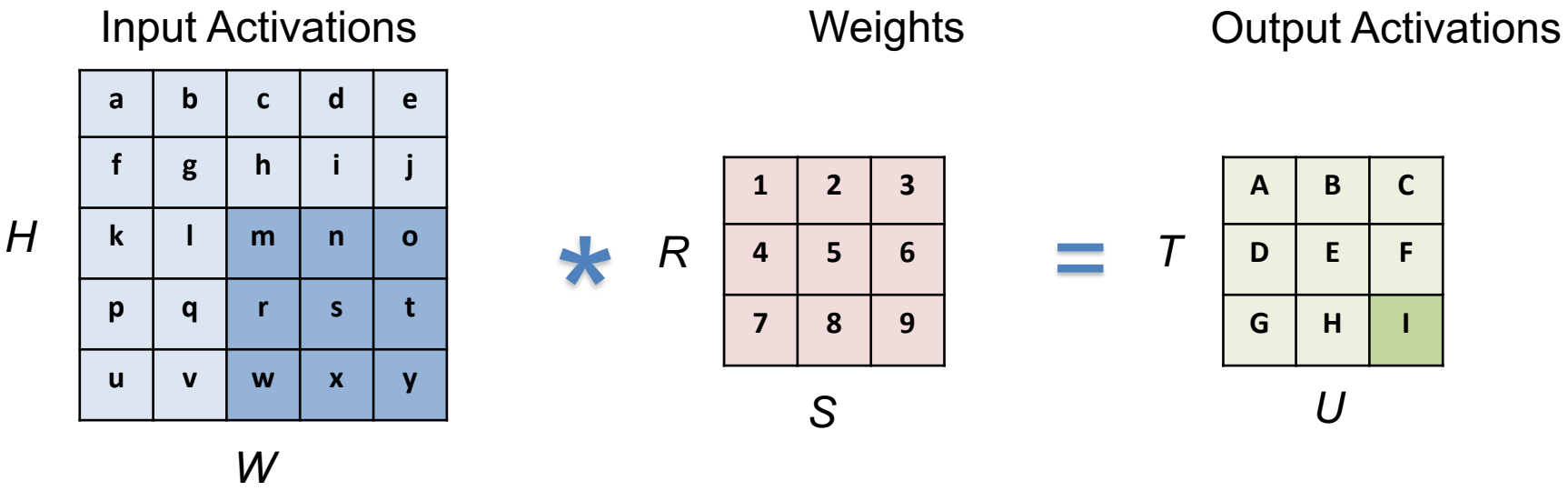
2D Convolution



$$B = b\times1 + c\times2 + d\times3 + g\times4 + h\times5 + i\times6 + l\times7 + m\times8 + n\times9$$

(R×S) Multiply and Accumulate (MAC) operations

2D Convolution



(T×U × R×S) MAC operations in total

A lot of potential data reuse for memory accesses

$$T = \frac{H - R}{stride} + 1$$

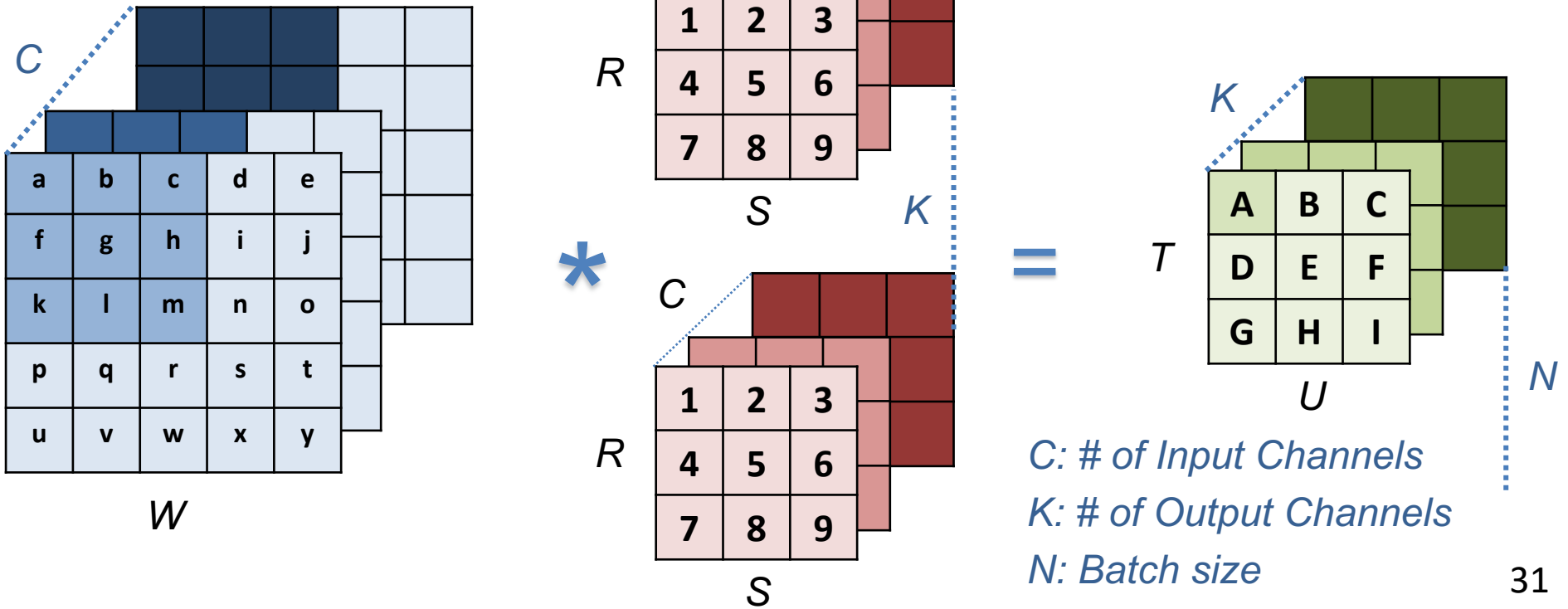
$$U = \frac{W - S}{stride} + 1$$

3D Convolution

Input Activations

Weights

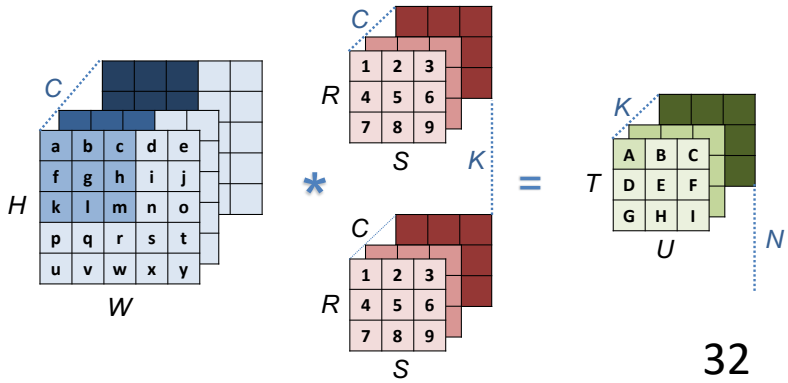
Output Activations



Convolution Loop Nest

```
for (n=0; n<N; n++) {                          // for each Batch
  for (k=0; k<K; k++) {                  // for each Output Channel
    for (t=0; t<T; t++) {                      // OA Height
      for (u=0; u<U; u++) {                    // OA Width
        OA[n][k][t][u]= 0;
        for (r=0; r<R; r++) {                  // W Height
          for (s=0; s<S; s++) {        // W Width
            for (c=0; c<C; c++) {  // for each Input Channel
              h = t * stride - pad + r;
              w = u * stride - pad + s;
              OA[n][k][t][u] += IA[n][c][h][w] * W[k][c][r][s];
            }
          }
        }
        Activation(OA[n][k][t][u]);
      }
    }
  }
}
```

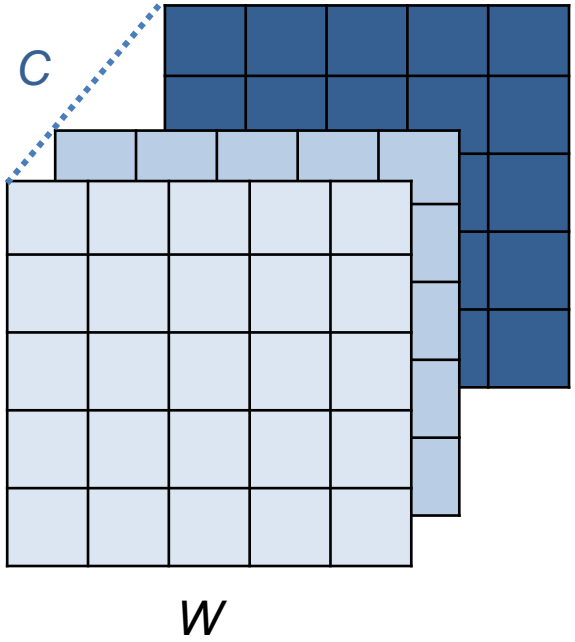
CONV kernel



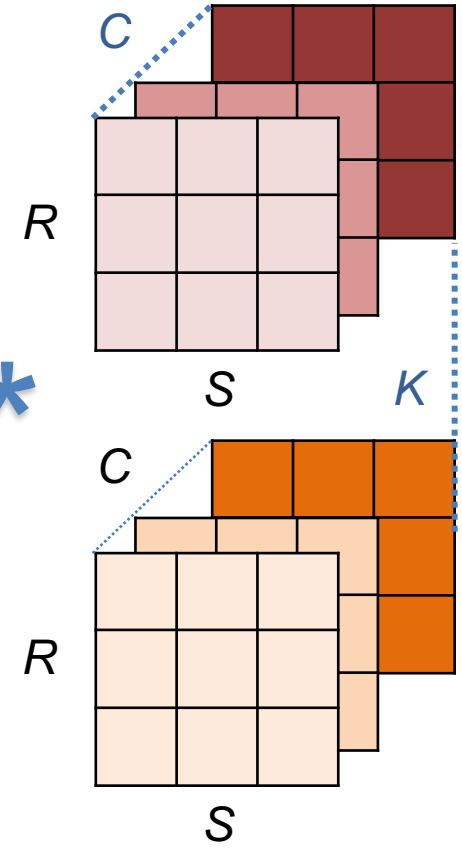
Opportunities for Data Reuse

Convolution Kernel

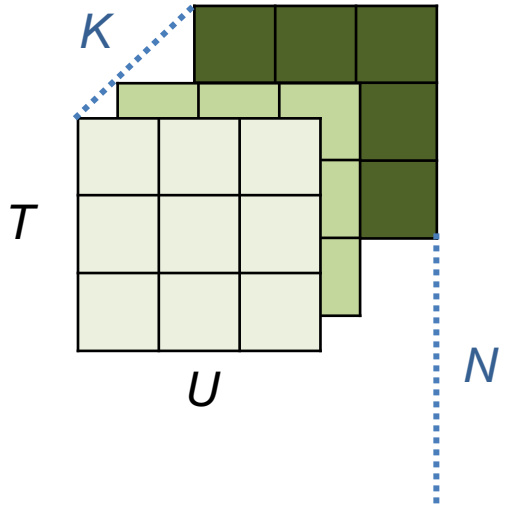
Input Activations
(feature maps)



Weights



Output Activations

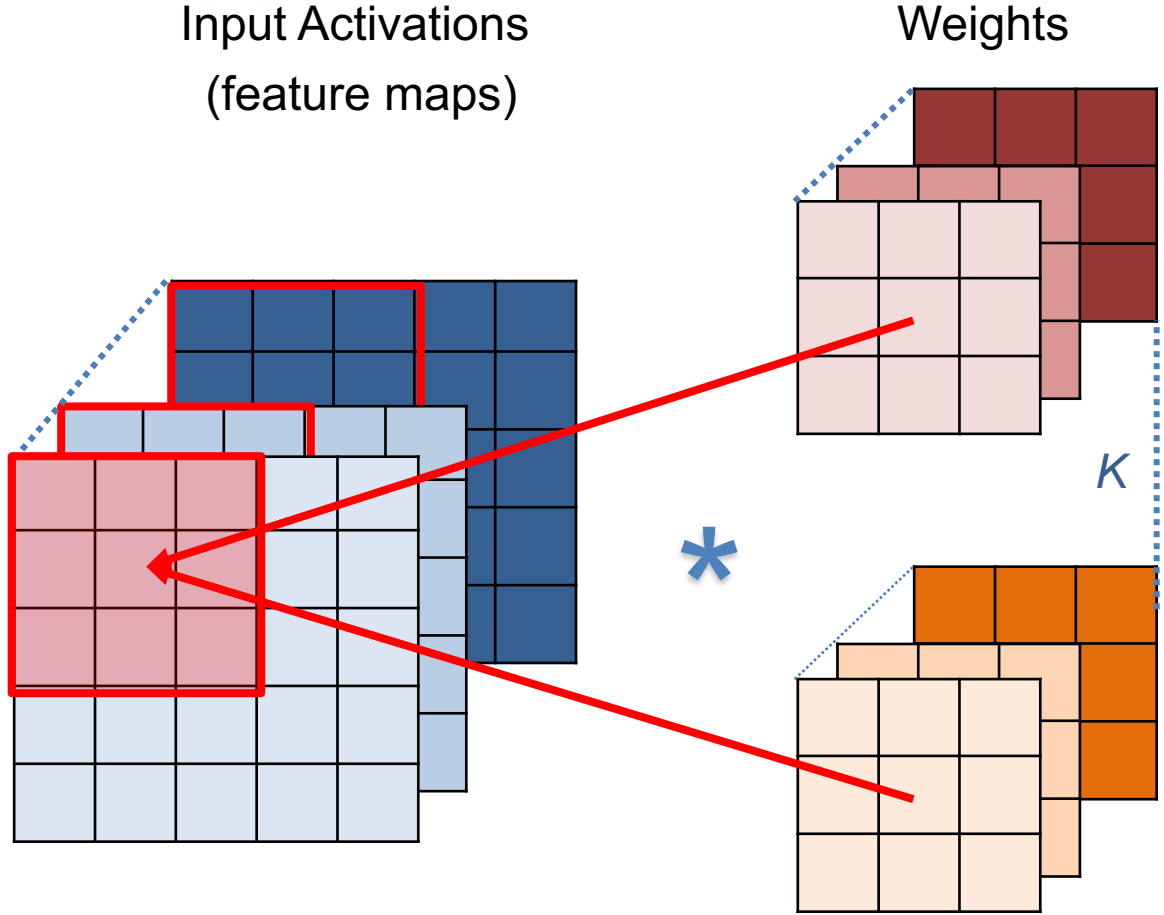


C: # of Input Channels

K: # of Output Channels

N: Batch size

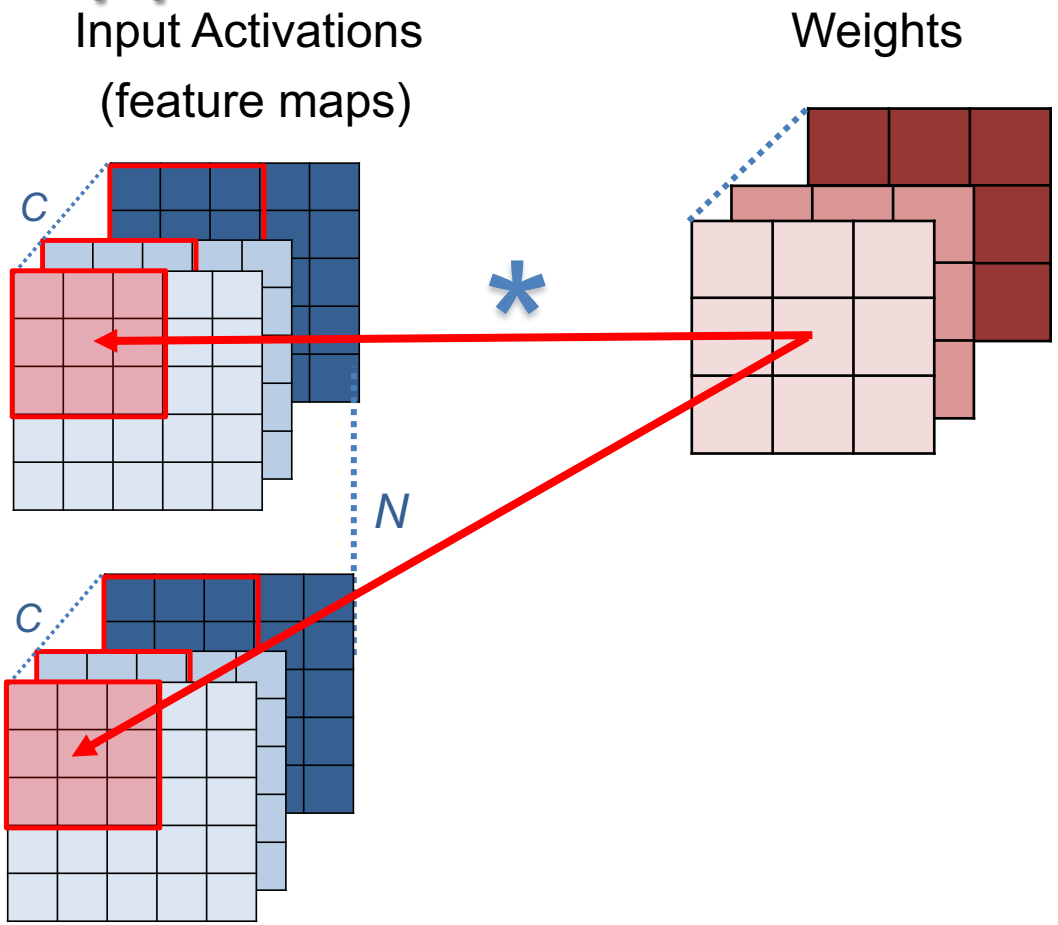
Opportunities for Data Reuse



- **Input reuse**
  - different filters are applied to the same input
  - each input is reused *K* times

C: # of Input Channels
K: # of Output Channels
N: Batch size

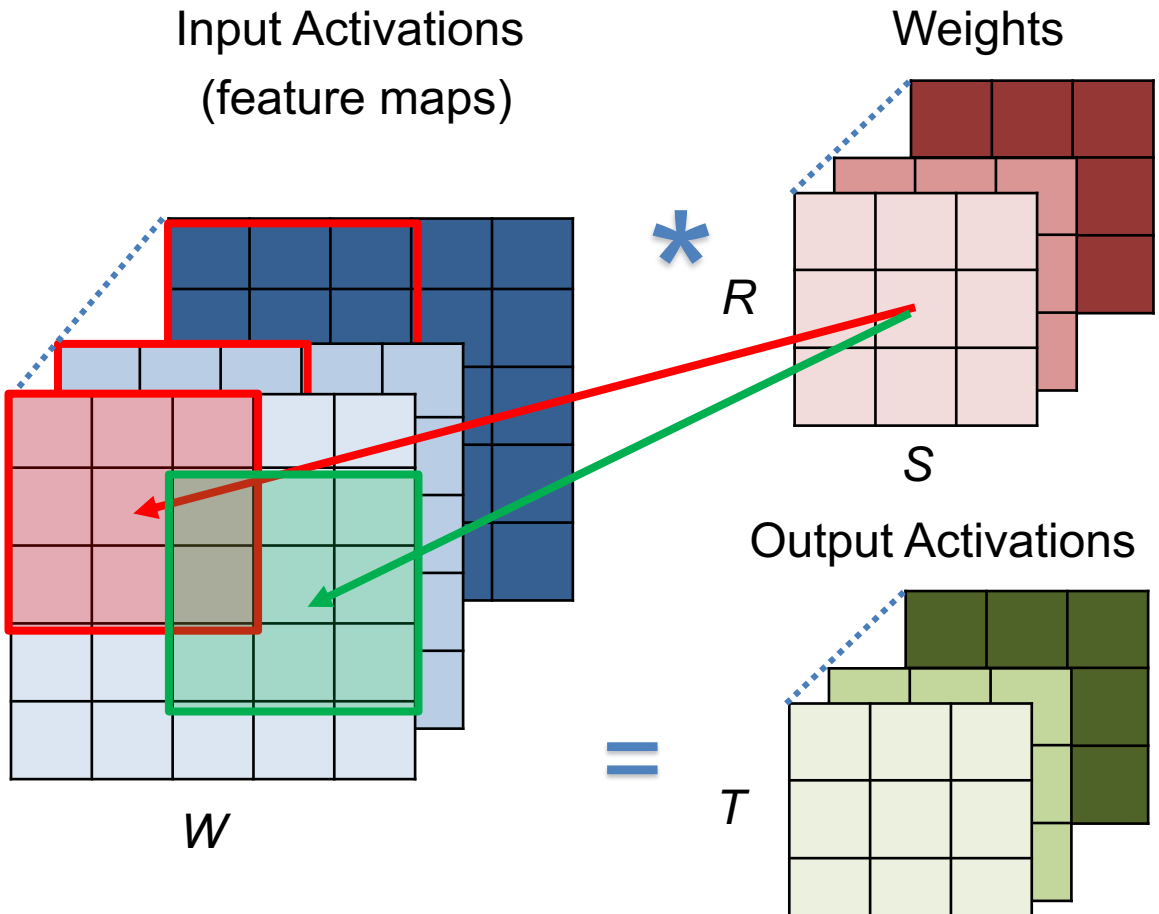
Opportunities for Data Reuse



- **Filter (weight) reuse**
  - when processing a batch of size *N*, all inputs are applied to the same filter
  - each filter weight is reused *N* times

C: # of Input Channels
K: # of Output Channels
N: Batch size

Opportunities for Data Reuse



- **Conv. reuse**
  - filters slide across different positions of the same input
  - each weight is reused ≈T.U times
  - each input is reused ≈R.S times

C: # of Input Channels
K: # of Output Channels
N: Batch size

Other Kernels

- Fully-Connected Layer
  - H=W=R=S=T=U=1
- Depth-Wise Convolution
  - K=1
- Pooling Layer
  - [MAX, AVG], pooling stride and kernel size
- BatchNorm Layer
  - provides zero-mean, unit-variance activations
- Activations
  - ReLU, L-ReLU, sigmoid, tanh, clipping

Speeding Up the GEMM Kernel

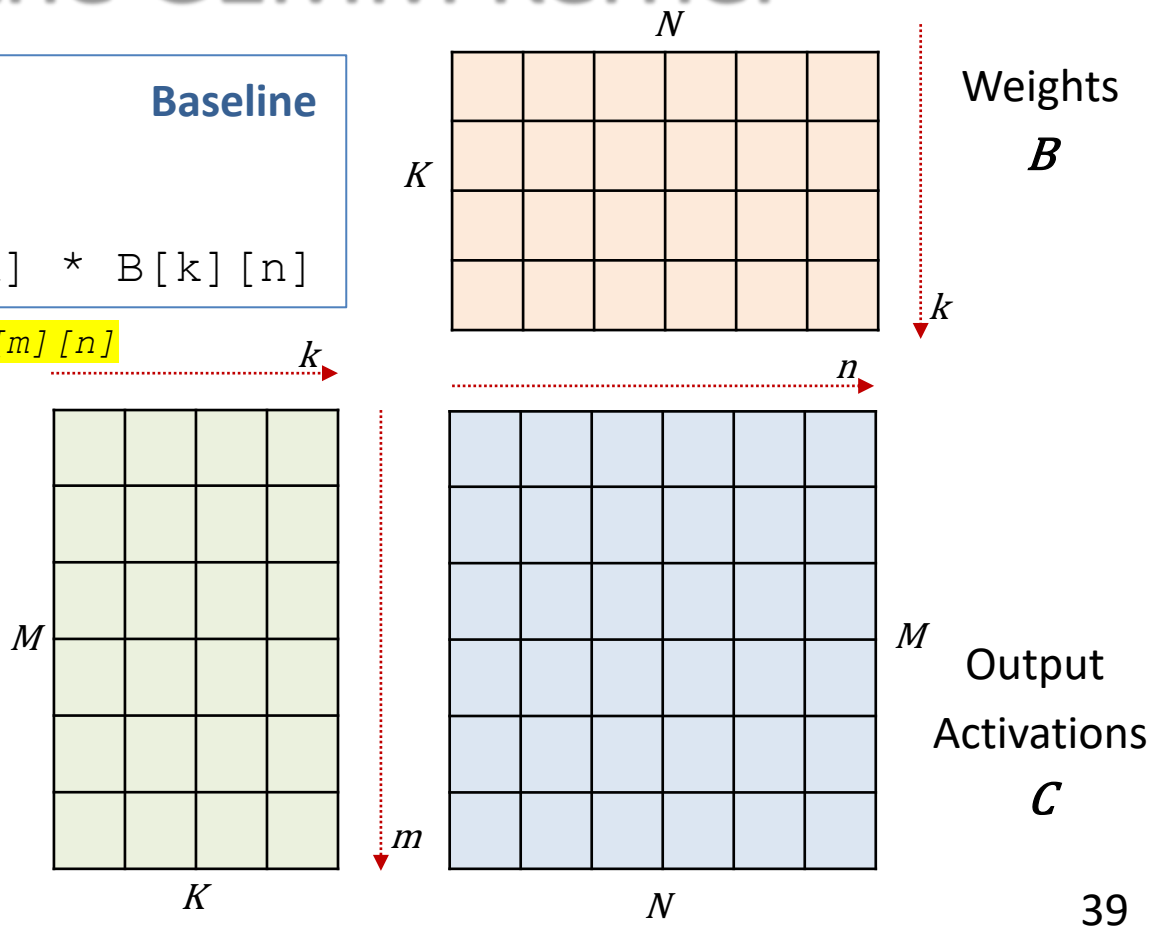
```
for (m=0; m<M; m++)
  for (n=0; n<N; n++)
    for (k=0; k<K; k++)
      C[m][n] += A[m][k] * B[k][n]
```

Baseline

note: a register should be used instead of `C[m][n]`

note: `C[m][n]` *should be intialized to 0*

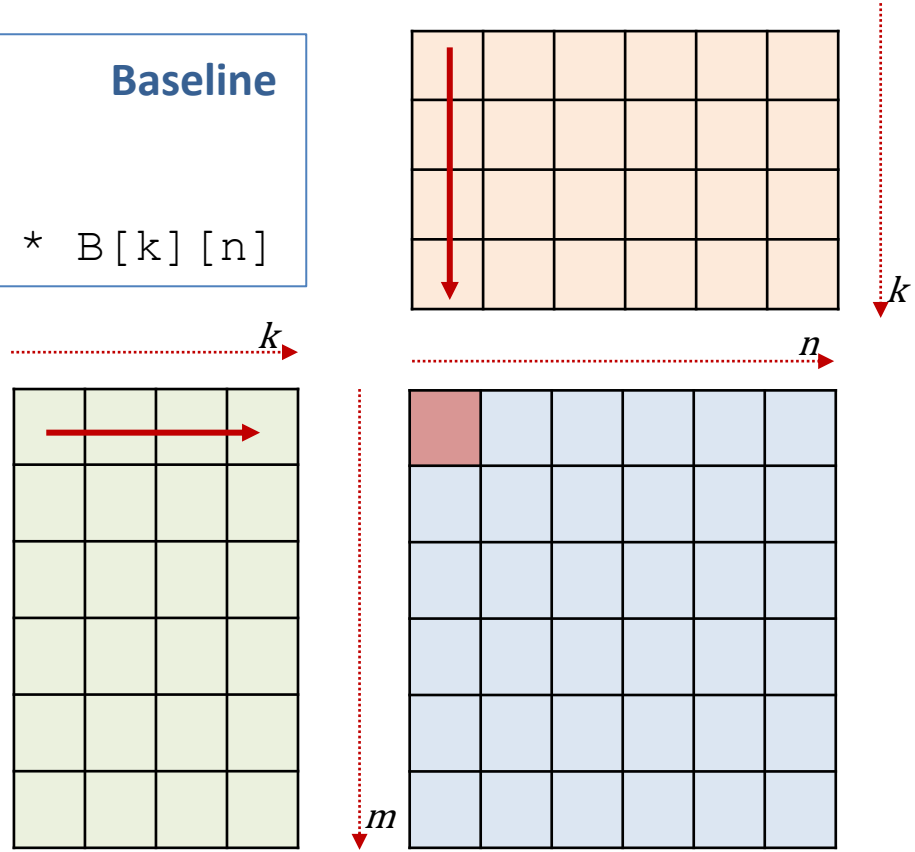
$$C_{M\times N} = A_{M\times K} \times B_{K\times N}$$



Speeding Up the GEMM Kernel

Baseline

```
m=0
  n=0
    for (k=0; k<K; k++)
      C[m][n] += A[m][k] * B[k][n]
```

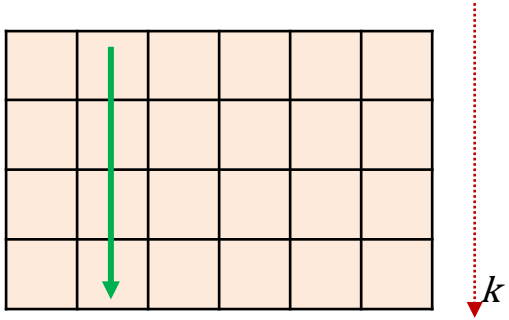


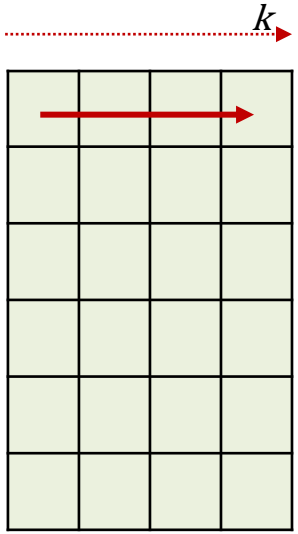
Traversal Order

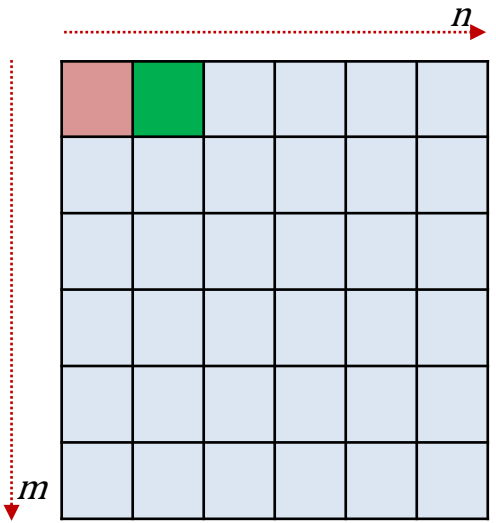
Speeding Up the GEMM Kernel

Baseline

```
m=0
  n=1
    for (k=0; k<K; k++)
      C[m][n] += A[m][k] * B[k][n]
```





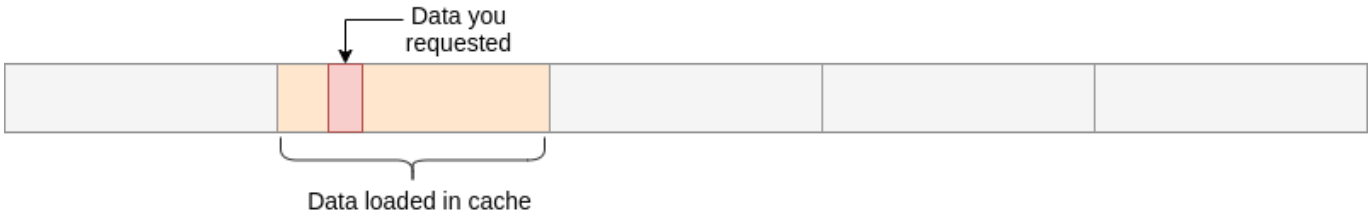


- **Assuming the storage is row-major, what happens in the cache memory?**
- **For which matrix?**

Traversal Order

Caching

- CPU caches are orders of magnitude faster, but much smaller, so using them correctly is critical
  - Automatically managed by the CPU.
  - Every time we fetch data from the main memory, the CPU automatically loads it and its neighboring memory into the cache, hoping to utilize locality of reference.

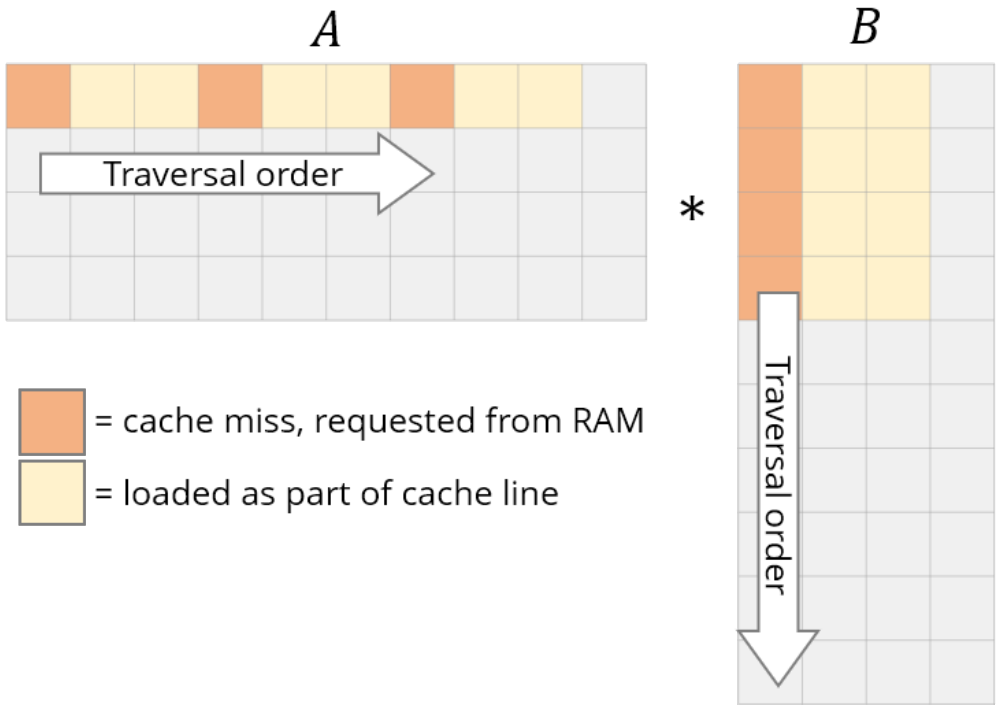


Caching

- In our case:
  - once we access A[m, k], the next element in the row, A[m, k+1] is already cached
  - but we get a cache miss for each data from matrix B fetched B[k, n]

```
for (m=0; m<M; m++)
  for (n=0; n<N; n++)
    for (k=0; k<K; k++)
      C[m][n] += A[m][k] * B[k][n]
```

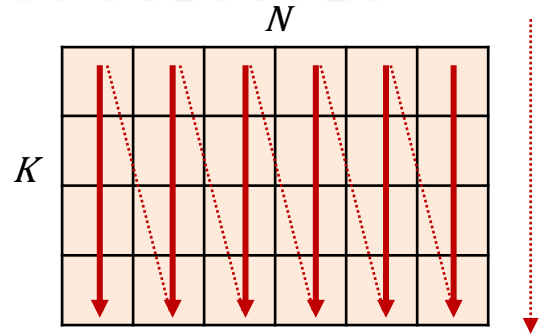
Baseline

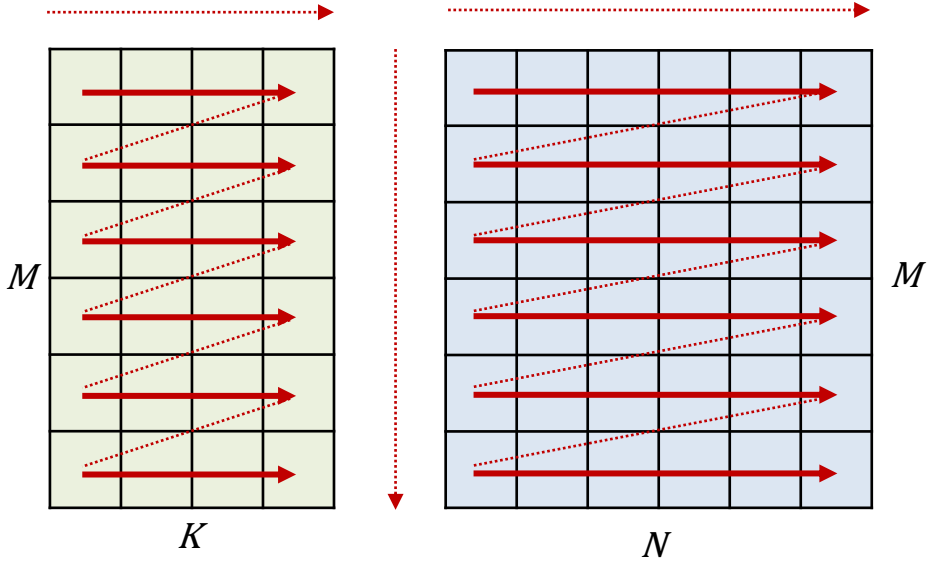


Speeding Up the GEMM Kernel

```
for (m=0; m<M; m++)
  for (n=0; n<N; n++)
    for (k=0; k<K; k++)
      C[m][n] += A[m][k] * B[k][n]
```

Baseline





Traversal Order

Speeding Up the GEMM Kernel

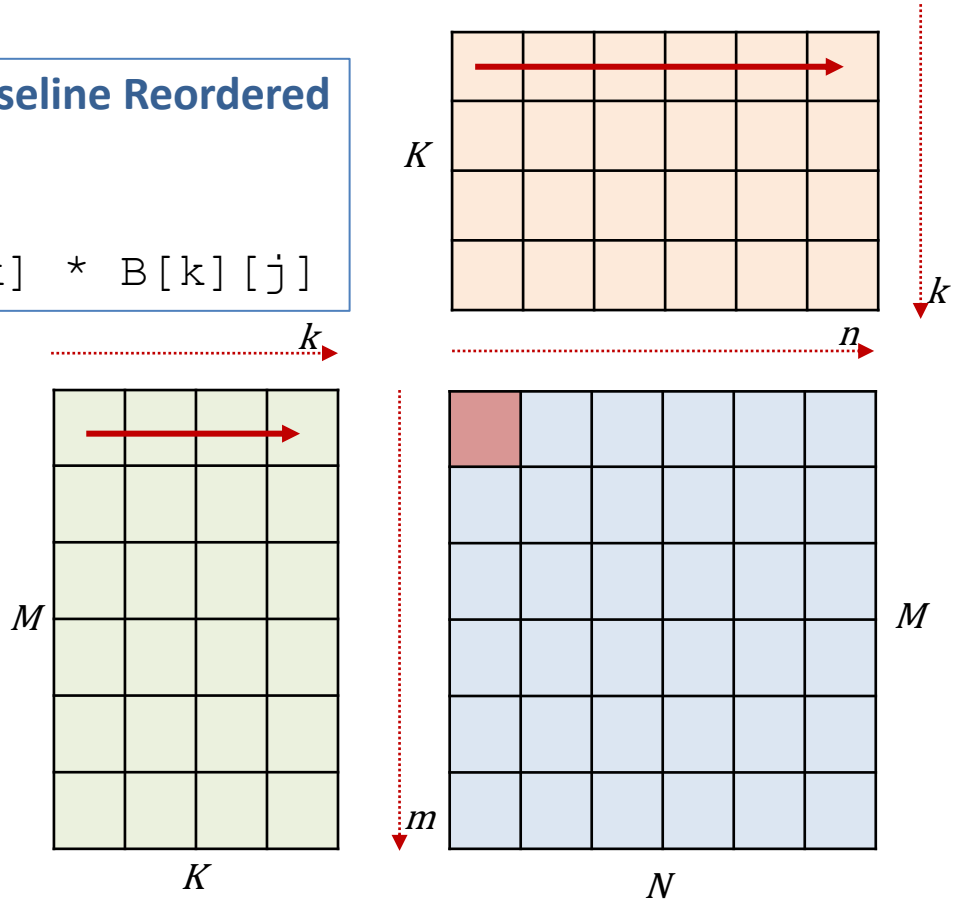
```
for (m=0; m<M; m++)
  for (k=0; k<K; k++)
    for (n=0; n<N; n++)
      C[i][j] += A[i][k] * B[k][j]
```

Baseline Reordered

- Reordering the loops from `m,n,k` to `m,k,n`
  - Improve data locality (better cache usage)

Speeding Up the GEMM Kernel

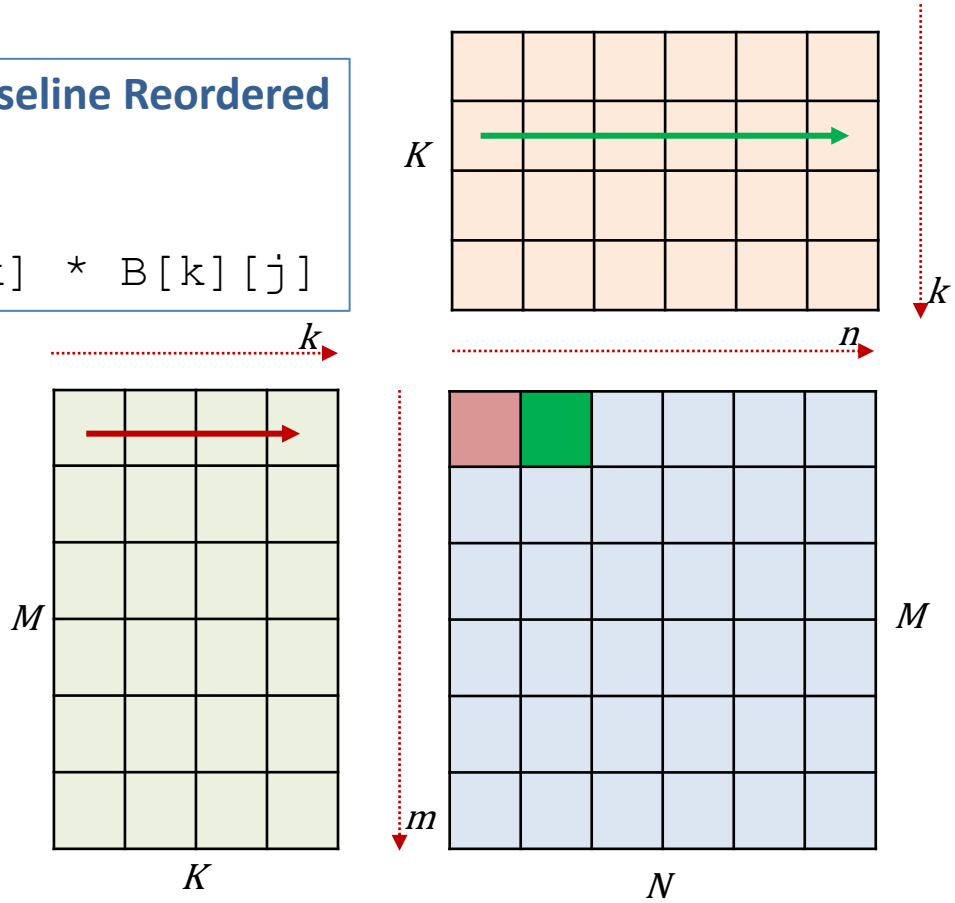
```
m=0                                    Baseline Reordered
   k=0
      for (n=0; n<N; n++)
         C[i][j] += A[i][k] * B[k][j]
```



Traversal Order

Speeding Up the GEMM Kernel

```
m=0                                          Baseline Reordered
  k=1
    for (n=0; n<N; n++)
      C[i][j] += A[i][k] * B[k][j]
```



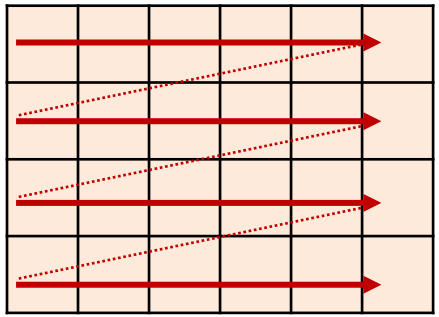
- **Assuming the storage is row-major, what happens in the cache memory?**
- **Why does this loop reordering result in better cache usage?**

Traversal Order

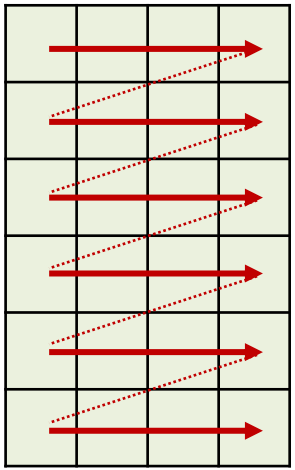
Speeding Up the GEMM Kernel

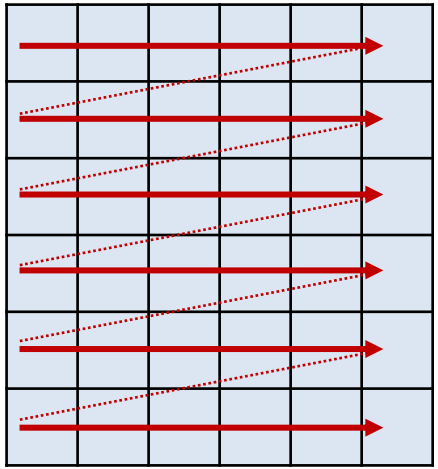
```
for (m=0; m<M; m++)
  for (k=0; k<K; k++)
    for (n=0; n<N; n++)
      C[i][j] += A[i][k] * B[k][j]
```

Baseline Reordered



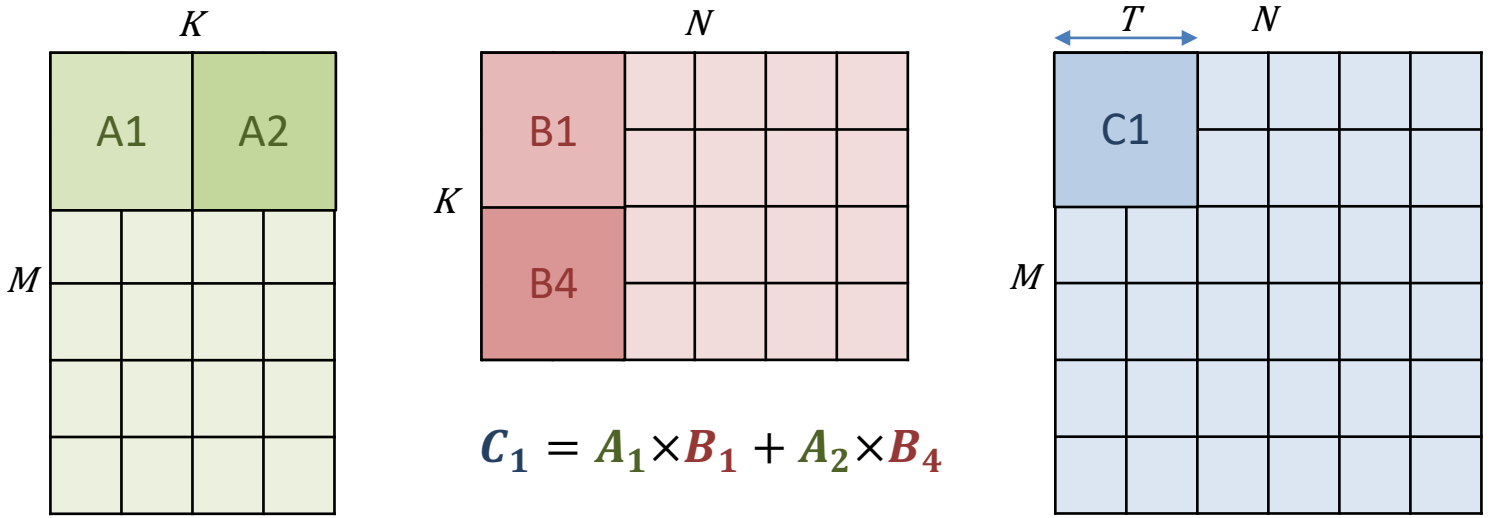
- Reordering the loops from `m, n, k` to `m, k, n`
  - Improve data locality (better cache usage)





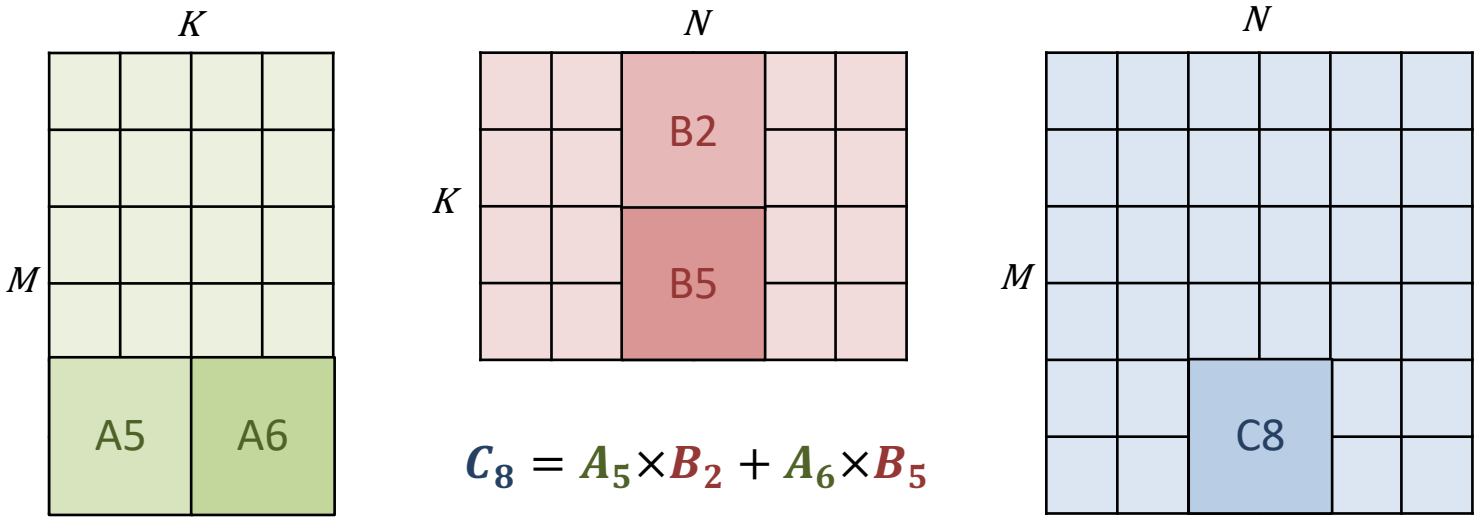
Speeding Up the GEMM Kernel

- Tiling
  - Looping on smaller submatrices (tiles of size $T \times T$)
    - small enough to fit in the cache



Speeding Up the GEMM Kernel

- Tiling
  - Looping on smaller submatrices (tiles of size $T \times T$)
    - small enough to fit in the cache



Speeding Up the GEMM Kernel

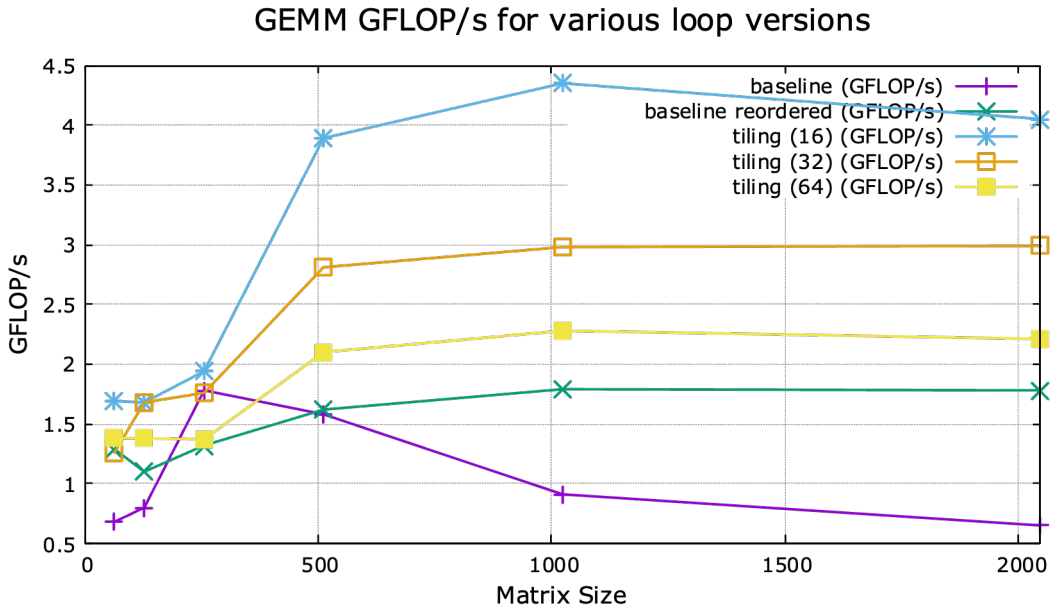
- Tiling
  - Looping on smaller submatrices (tiles of size $T \times T$)

```
for (m=0; m<M/T; m++)                    // Tile row index
  for (n=0; n<N/T; n++)                  // Tile column index
    for (k=0; k<K/T; k++)                // Tile inner index

      for (mt=0; mt<T; mt++)             // Tile-level mult.
        for (nt=0; nt<T; nt++)
          for (kt=0; kt<T; kt++)

            C[m*T+mt][n*T+nt] += A[m*T+mt][k*T+kt]
                               * B[k*T+kt][n*T+nt]
```

note: for this code M, N and K must be divisible by T

Tiling

Speeding Up the GEMM Kernel



Results on CPU
(Apple Silicon M2 Pro)

- More of Tiling
  - Tile inner loops can be vectorized and unrolled
  - Tiles can run in parallel (multithreading)

Try it yourself

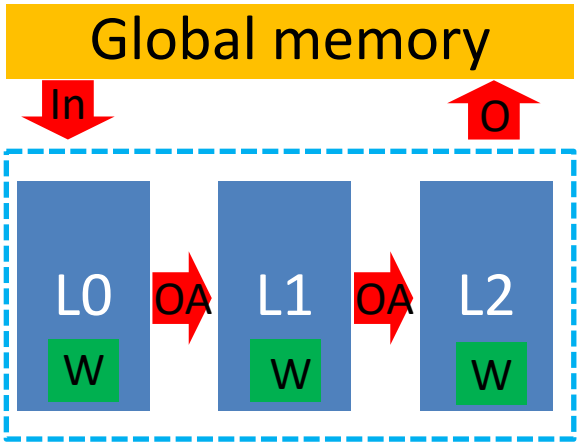
```
git clone https://gitlab.inria.fr/sentieys/dnn_acc.git
cd GEMM
# look at the C code: baseline.c, baseline_reordered.c, opti-l1.c,
opti-l2.c
make all
```

How to add SIMD and vectorization?

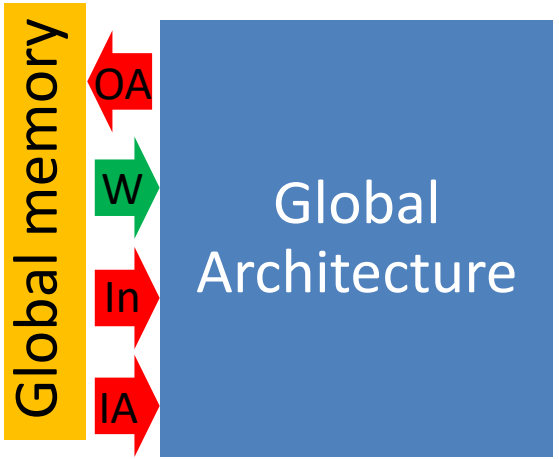
Designing Hardware Accelerators for GEMM/CONV

Building the Accelerator

Pipeline (Dataflow) Architecture



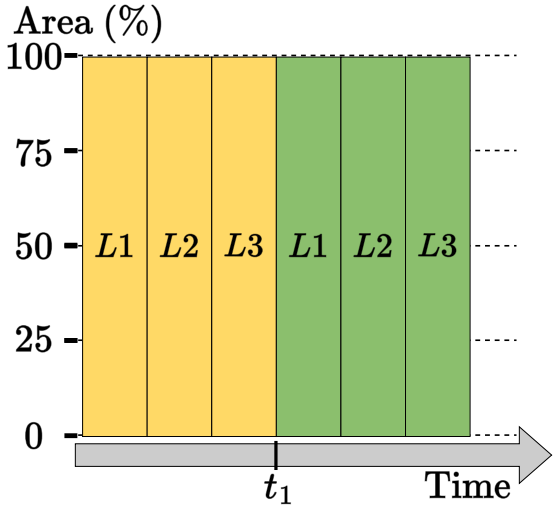
Li: Layer *i*

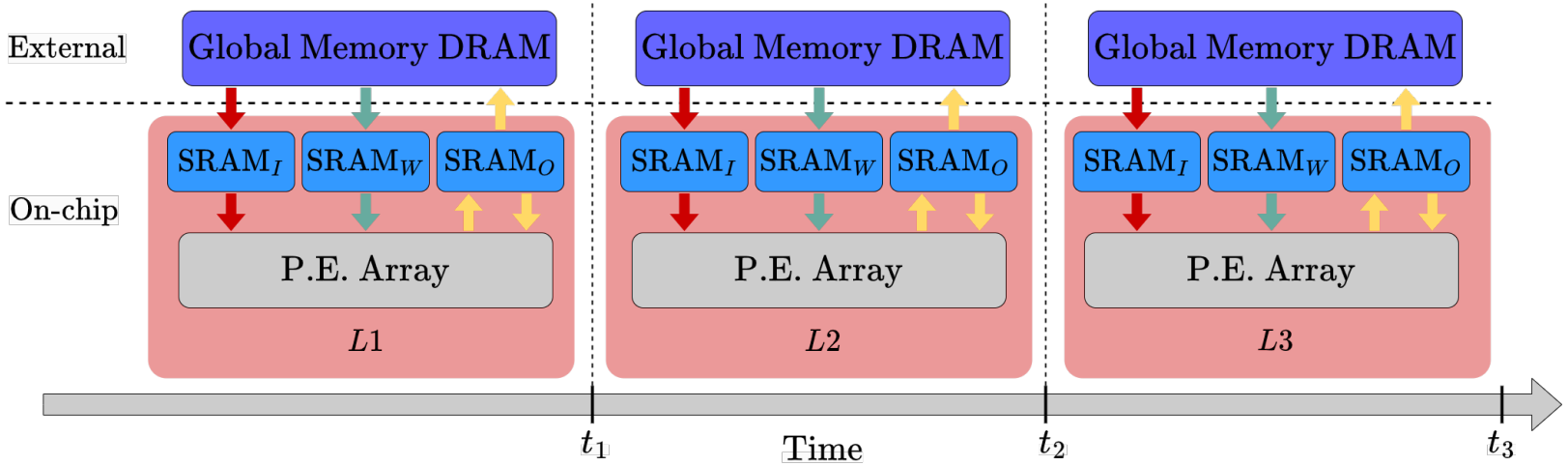


Sequential Architecture

Building the Accelerator

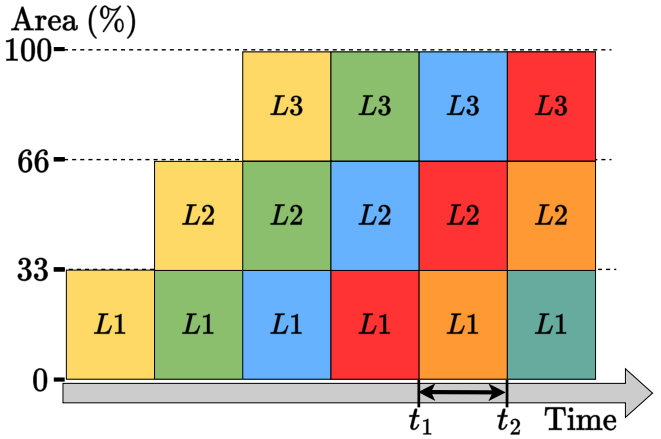
- Sequential Architecture

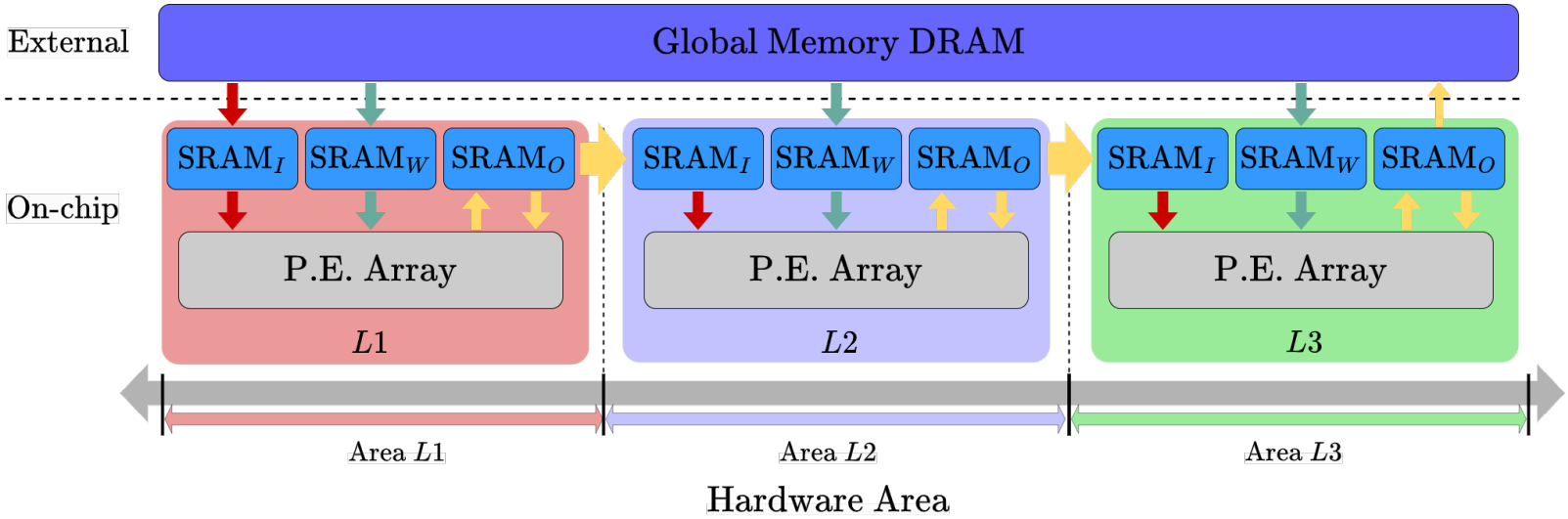




Building the Accelerator

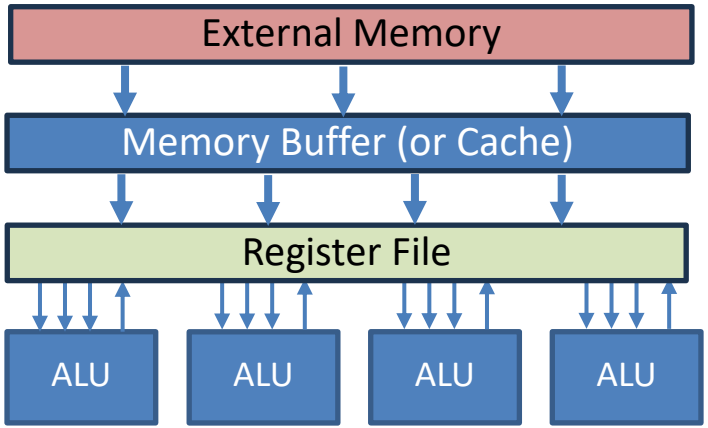
- Dataflow Architecture



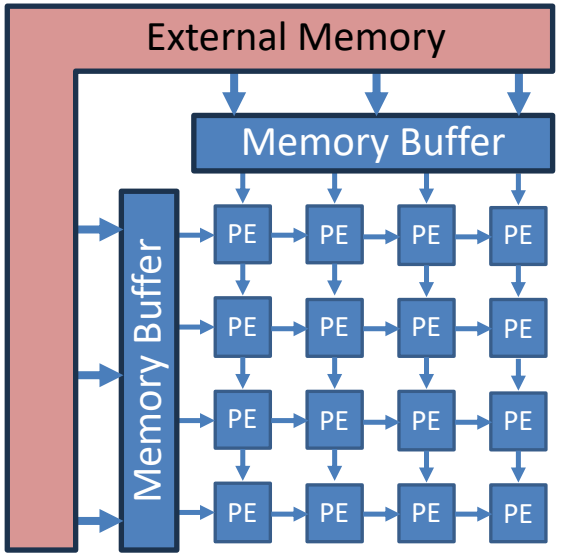


Exploiting Data Reuse in PE Array

- Temporal Architecture
  - SIMD (CPU), SIMT (GPU)
  - Classical Memory Hierarchy



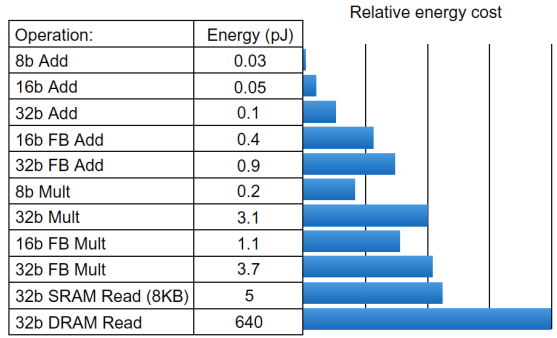
- Spatial Architecture
  - Dataflow accelerators

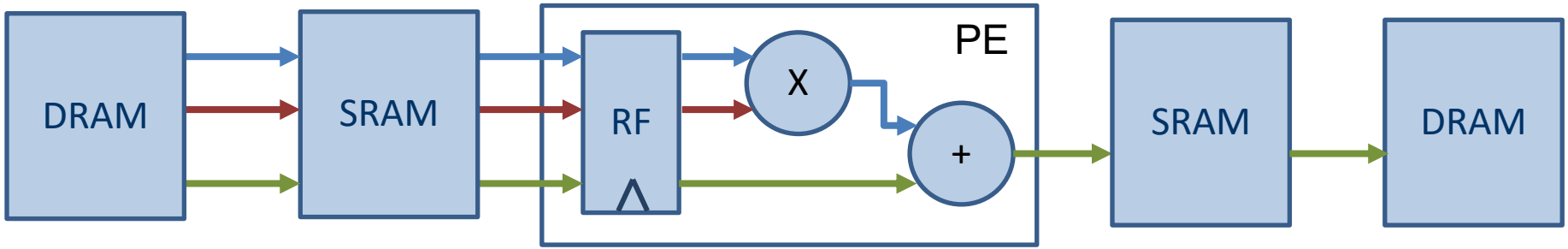


Exploiting Data Reuse

- Why reuse is important?
  - Relative energy costs
  - Memory access is the bottleneck

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FB Add	0.4
32b FB Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FB Mult	1.1
32b FB Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640

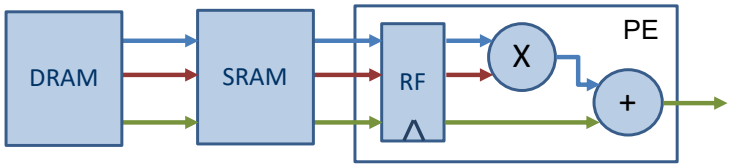




DRAM: **200x** (external) SRAM: **6x** (cache, buffer) PE: **2x** MAC: **1x** **Energy**

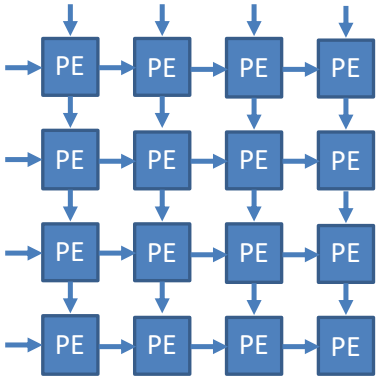
Exploiting Data Reuse

- Temporal Reuse
  - e.g., memory hierarchy
  - the same data is used more than once over time by the same PE

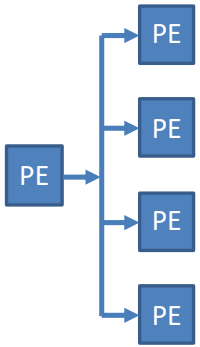


- Temporal and Spatial Reuse
  - Memory hierarchy and multiple PEs

- Spatial Architecture
  - e.g., systolic, multicast
  - the same data is used by more than one PE at different spatial locations of the hw



systolic

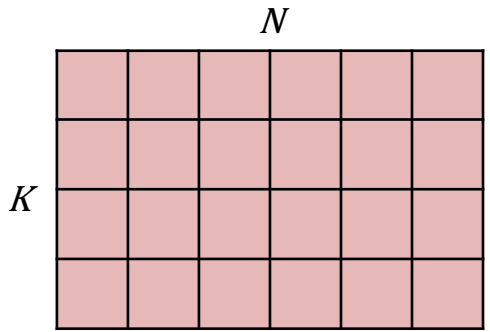


multicast

Back to the GEMM Kernel

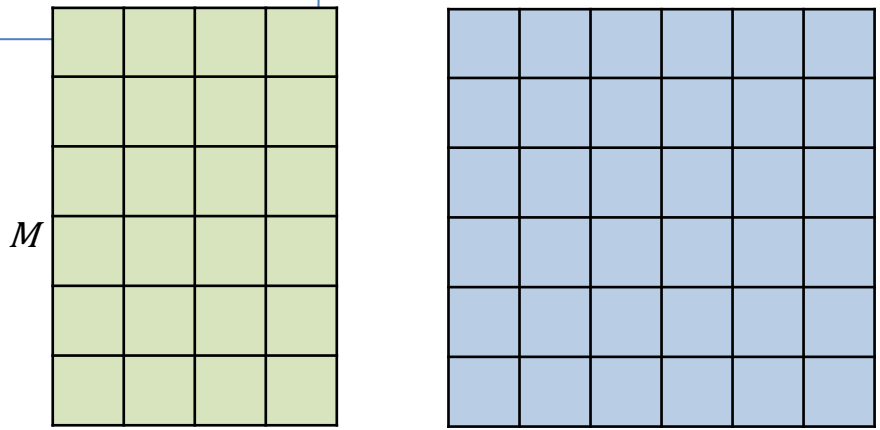
```
for (m=0; m<M; m++) {
  for (n=0; n<N; n++) {
    C[m][n] = 0;
    for (k=0; k<K; k++) {
      C[m][n] += A[m][k] * B[k][n];
    }
  }
}
```

$$C_{M\times N} = A_{M\times K} \times B_{K\times N}$$



Weights

B



Input Activations

A

Output Activations

C

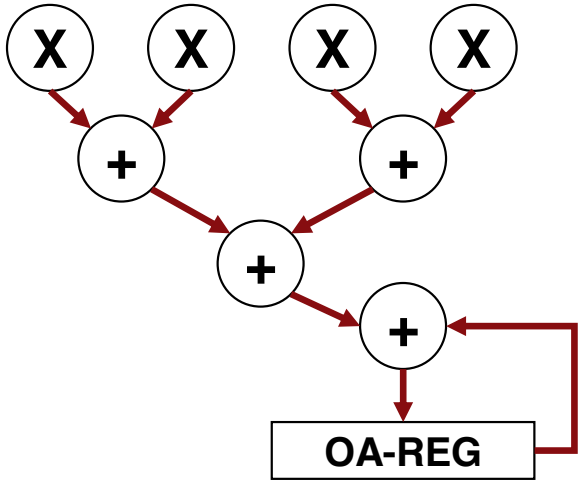
Accelerating GEMM

- Parallelizing most inner loop
  - (1) Adder tree
    - Typical width: 8-64
  - e.g. NVDLA, NVIDIA Tensor cores, FINN

```
for (m=0; m<M; m++) {
  for (n=0; n<N; n++) {
    C[m][n] = 0;
    parallel_for (k=0; k<K; k++) {
      C[m][n] += A[m][k] * B[k][n];
    }
  }
}
```

for: temporal execution order

parallel_for: parallel execution

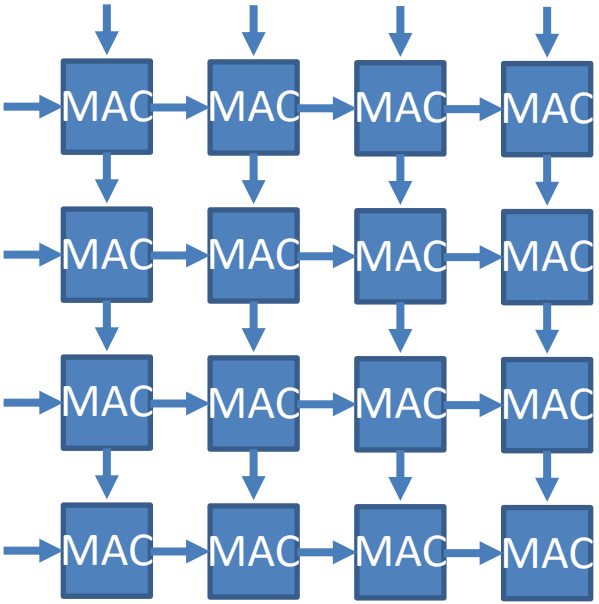


[Credit: Sophia Shao, Hardware for Machine Learning, Course@UC Berkeley]

Accelerating GEMM

- Parallelizing most inner loop
  - (2) Systolic Multiply-And-Accumulate (MAC)
    - Typical width: 8-256
  - e.g. Gemmini, Google TPU

```
for (m=0; m<M; m++) {
  for (n=0; n<N; n++) {
    C[m][n] = 0;
    parallel_for (k=0; k<K; k++) {
      C[m][n] += A[m][k] * B[k][n];
    }
  }
}
```



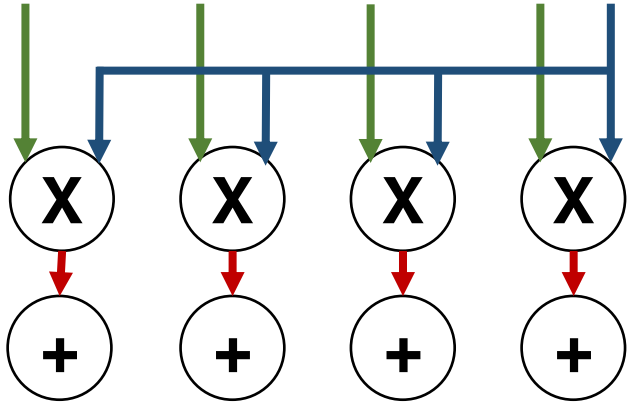
Accelerating GEMM

- Parallelizing second inner loop
  - (3) Multicasting a (sub)line of weights
    - Typical width: 8-16
  - e.g. NVDLA, NVIDIA Tensor cores

```
for (m=0; m<M; m++) {
  parallel_for (n=0; n<N; n++) {
    C[m][n] = 0;
    for (k=0; k<K; k++) {
      C[m][n] += A[m][k] * B[k][n];
    }
  }
}
```

for: temporal execution order

parallel_for: parallel execution

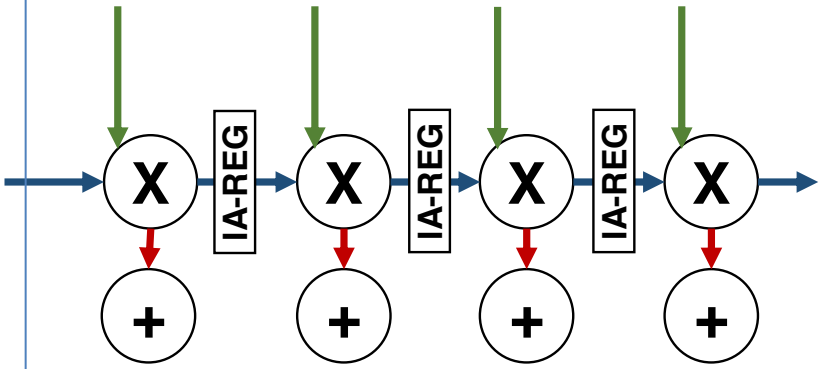


[Credit: Sophia Shao, Hardware for Machine Learning, Course@UC Berkeley]

Accelerating GEMM

- Parallelizing second inner loop
  - (4) Systolic multicast
    - Typical width: 8-256
  - e.g. Gemmini, Google TPU

```
for (m=0; m<M; m++) {
  parallel_for (n=0; n<N; n++) {
    C[m][n] = 0;
    for (k=0; k<K; k++) {
      C[m][n] += A[m][k] * B[k][n];
    }
  }
}
```

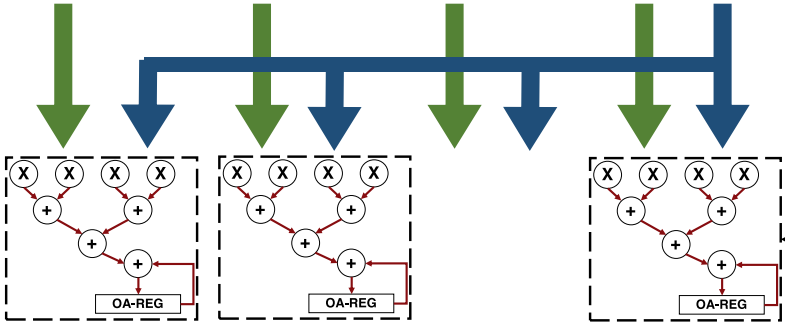


[Credit: Sophia Shao, Hardware for Machine Learning, Course@UC Berkeley]

Accelerating GEMM

- Parallelizing second and most inner loops
  - (1)+(3) Adder Tree + Multicast
  - e.g. NVDLA, FINN

```
for (m=0; m<M; m++) {
  parallel_for (n=0; n<N; n++) {
    C[m][n] = 0;
    parallel_for (k=0; k<K; k++) {
      C[m][n] += A[m][k] * B[k][n];
    }
  }
}
```

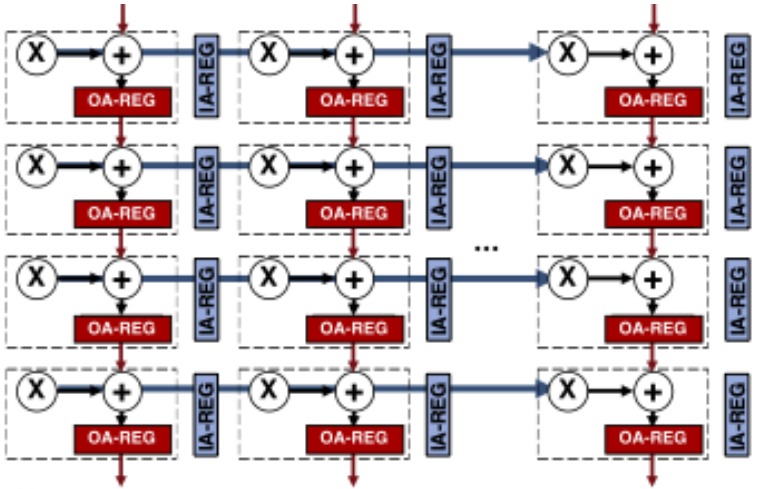


[Credit: Sophia Shao, Hardware for Machine Learning, Course@UC Berkeley]

Accelerating GEMM

- Parallelizing second and most inner loops
  - (2)+(4) Systolic MAC + Systolic Multicast
  - e.g. Gemmini, Google TPU

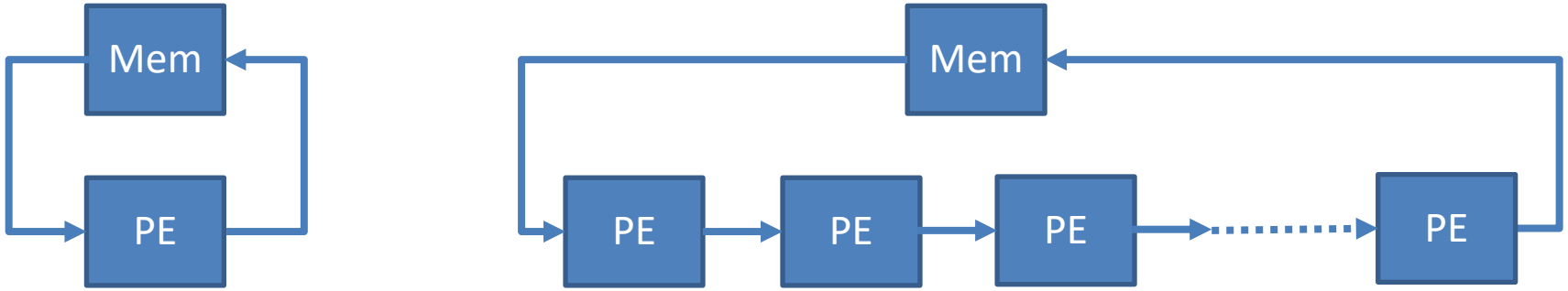
```
for (m=0; m<M; m++) {
  parallel_for (n=0; n<N; n++) {
    C[m][n] = 0;
    parallel_for (k=0; k<K; k++) {
      C[m][n] += A[m][k] * B[k][n];
    }
  }
}
```



[Credit: Sophia Shao, Hardware for Machine Learning, Course@UC Berkeley]

Systolic Arrays

- Replace single Processing Element (PE) with an array of regular PEs



- Orchestrate data flow for high throughput with less memory access than classical architectures
- Each PE may have (small) local instruction and data memory
- *Analogy with the heart ~~blood~~ → (many)cells → heart*

Systolic Arrays are New?

SYSTOLIC ARCHITECTURES FOR CONNECTED SPEECH RECOGNITION (*)

Francois CHAROT
Patrice FRISON
Patrice QUINTON

IRISA, Campus de Beaulieu,
35042 RENNES-CEDEX
FRANCE

July 1984

ABSTRACT

Systolic arrays for two connected speech recognition methods are presented. The first method is based on the dynamic time warping algorithm which is applied directly on acoustic feature patterns. The second method is the probabilistic matching algorithm which requires that the input sentence be preprocessed by a phonetic analyzer. It is shown that both methods may be implemented on either a two-dimensional or a linear systolic array. Advantages of each of these implementations are discussed. The architecture of a 12000 transistors programmable NMOS prototype IC which can be used as the basic processor of

(*) This work has been partly funded by a grant of the CNET in Lannion (French Telecommunication Research Center).

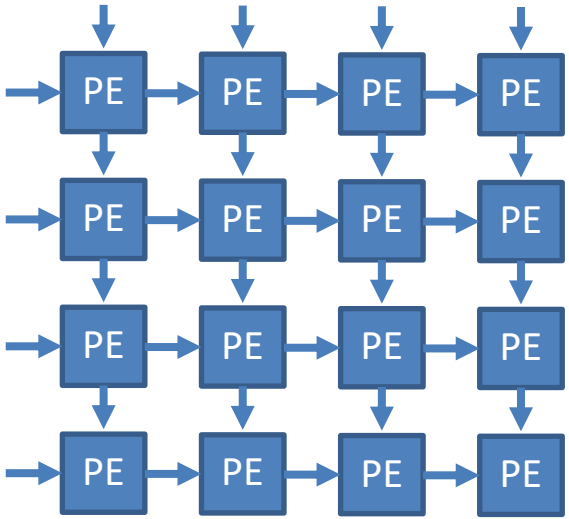
Systolic Array Matrix Multiplication

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,0}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,0}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,0}$	$b_{3,3}$

$$c_{i,j} = \sum_{k=0}^{K-1} a_{i,k} \times b_{k,j}$$

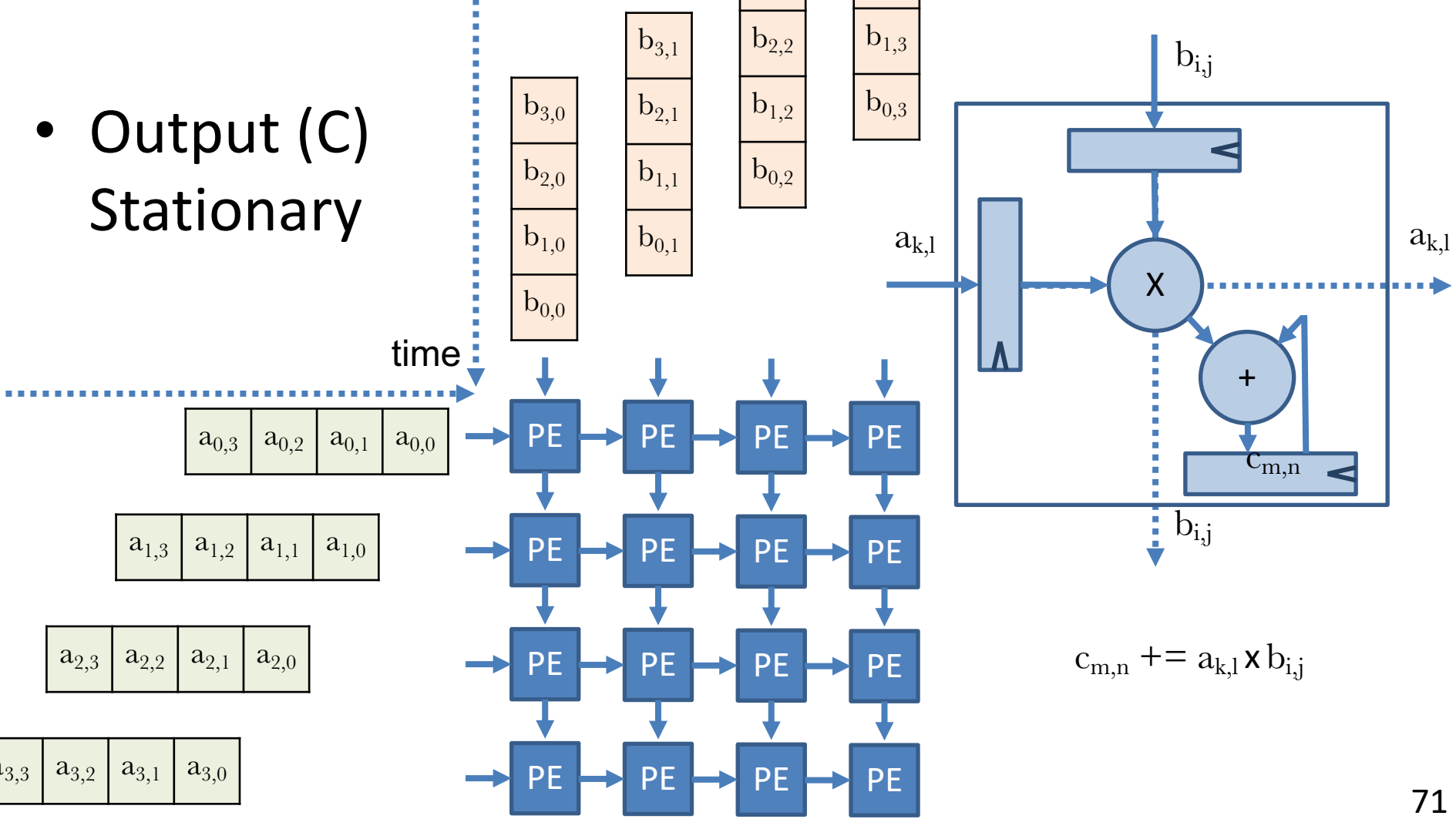
$$c_{i,j} = a_{i,0} \times b_{0,j} + a_{i,1} \times b_{1,j} + \ldots + a_{i,K-1} \times b_{K-1,j}$$

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,0}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,0}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,0}$	$a_{3,3}$

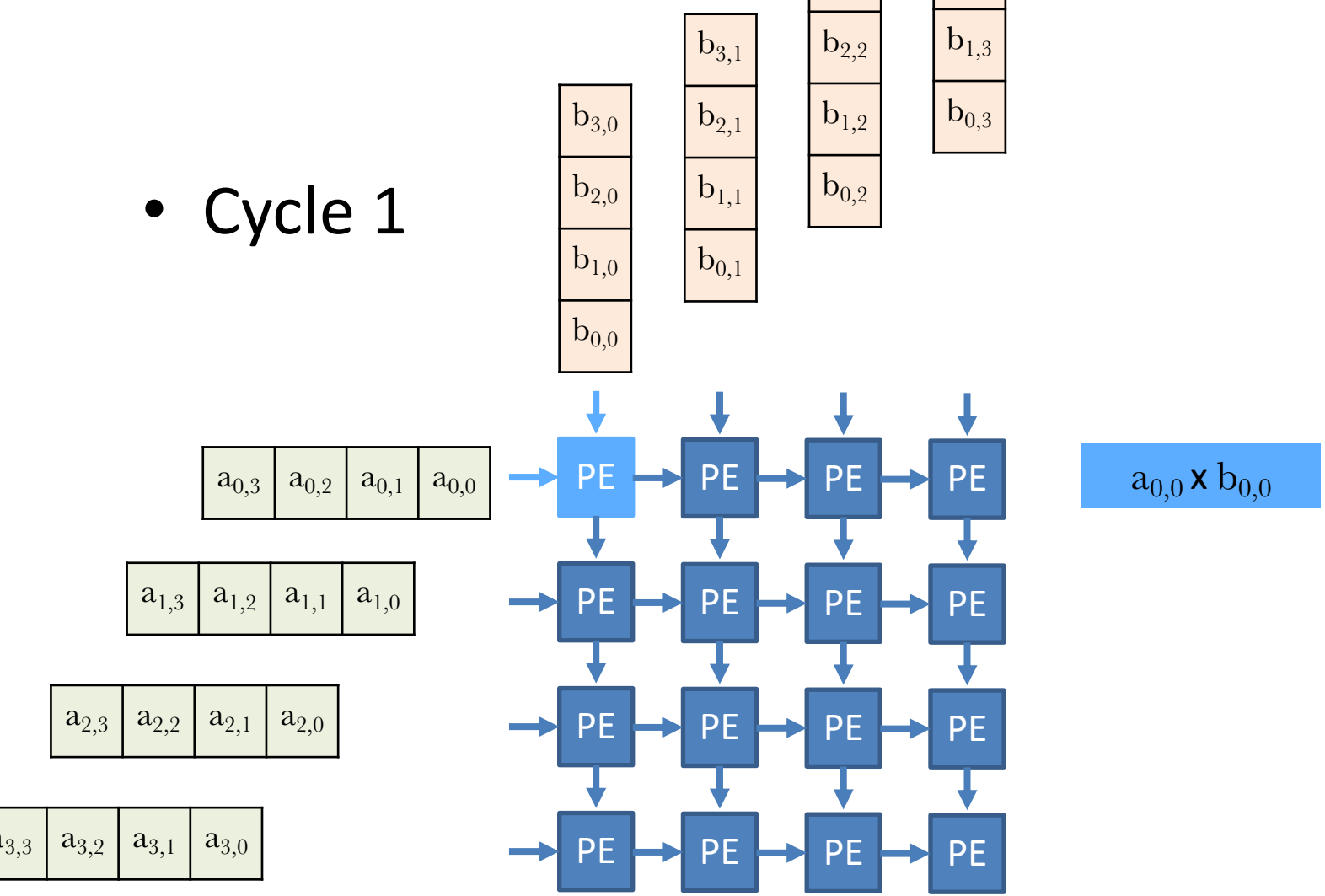


$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	$c_{0,3}$
$c_{1,0}$	$c_{1,1}$	$c_{1,0}$	$c_{1,3}$
$c_{2,0}$	$c_{2,1}$	$c_{2,0}$	$c_{2,3}$
$c_{3,0}$	$c_{3,1}$	$c_{3,0}$	$c_{3,3}$

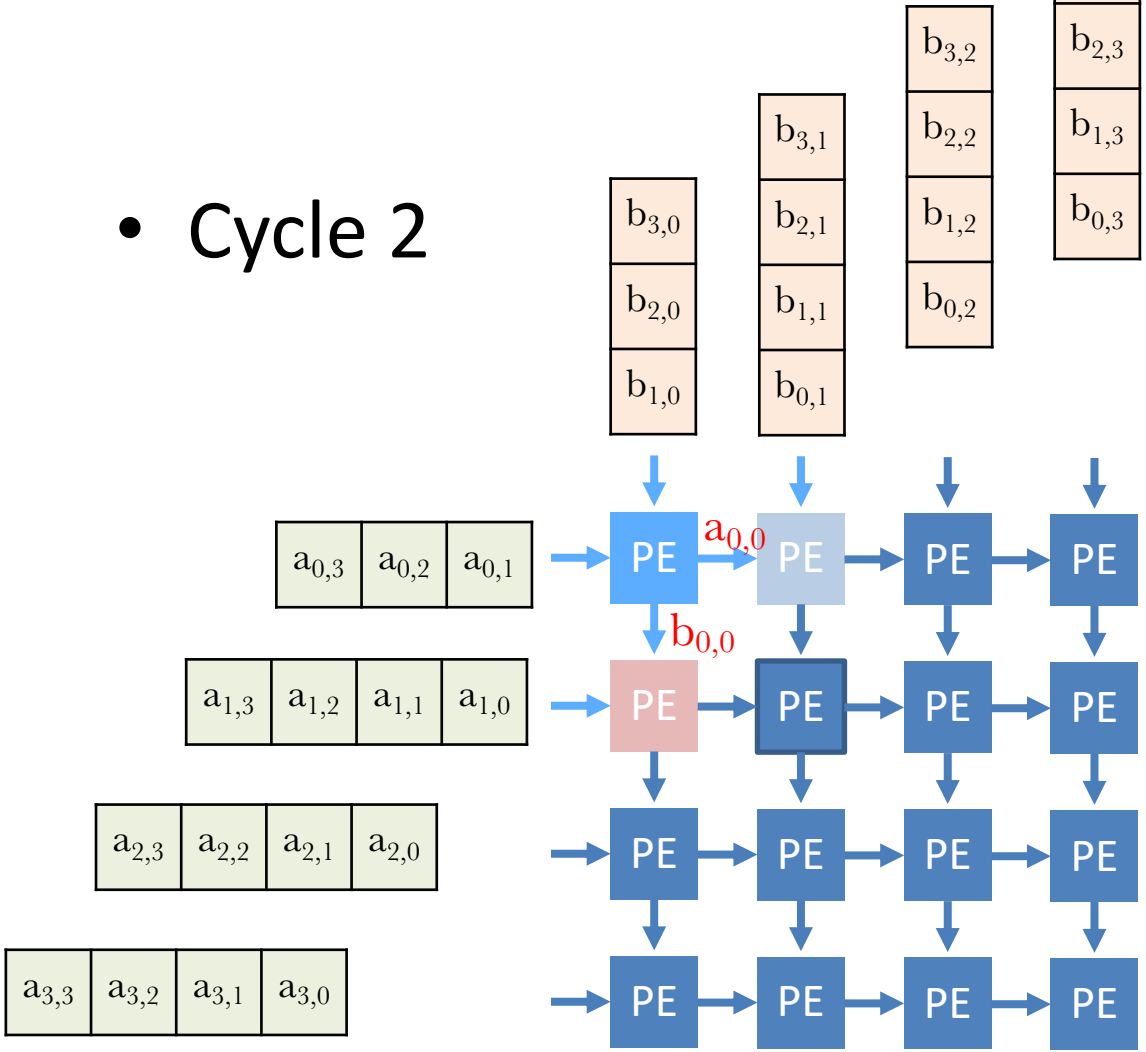
- Output (C) Stationary



- Cycle 1


b3,0
b2,0
b1,0
b0,0
b3,1
b2,1
b1,1
b0,1
b2,2
b1,2
b0,2
b1,3
b0,3
a0,3
a0,2
a0,1
a0,0
a1,3
a1,2
a1,1
a1,0
a2,3
a2,2
a2,1
a2,0
a3,3
a3,2
a3,1
a3,0
PE
a0,0 x b0,0

- Cycle 2

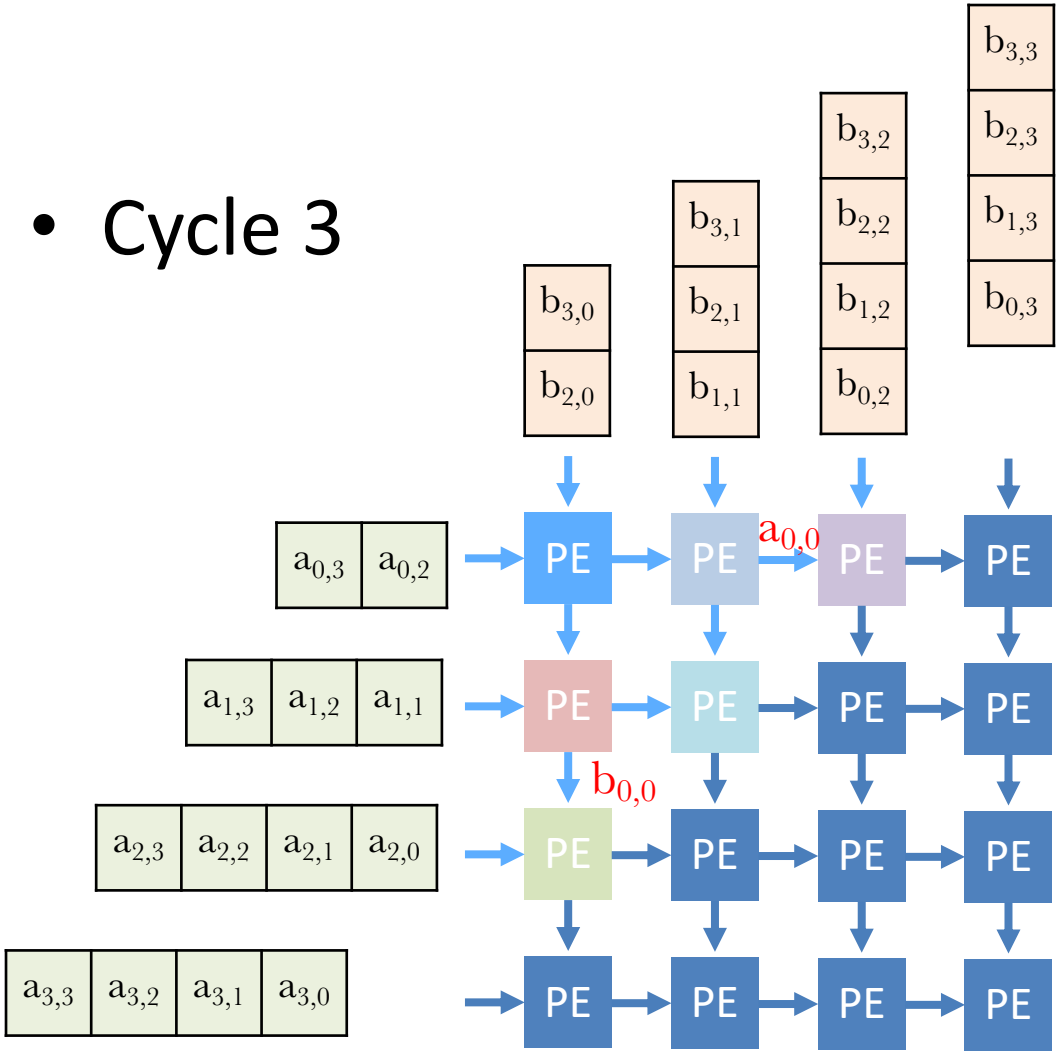


$a_{0,1} \times b_{1,0} + a_{0,0} \times b_{0,0}$

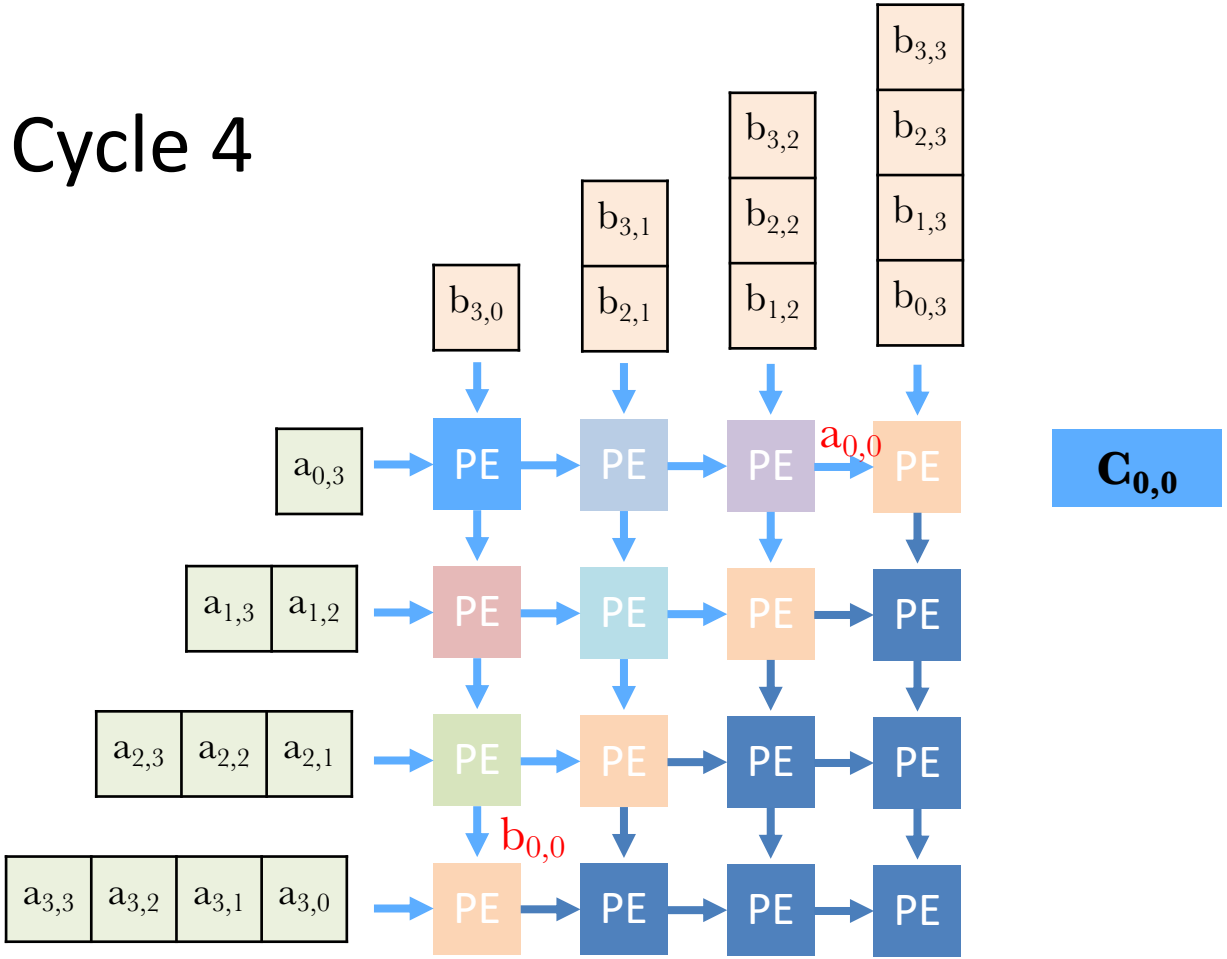
$a_{0,0} \times b_{0,1}$

$a_{1,0} \times b_{0,0}$

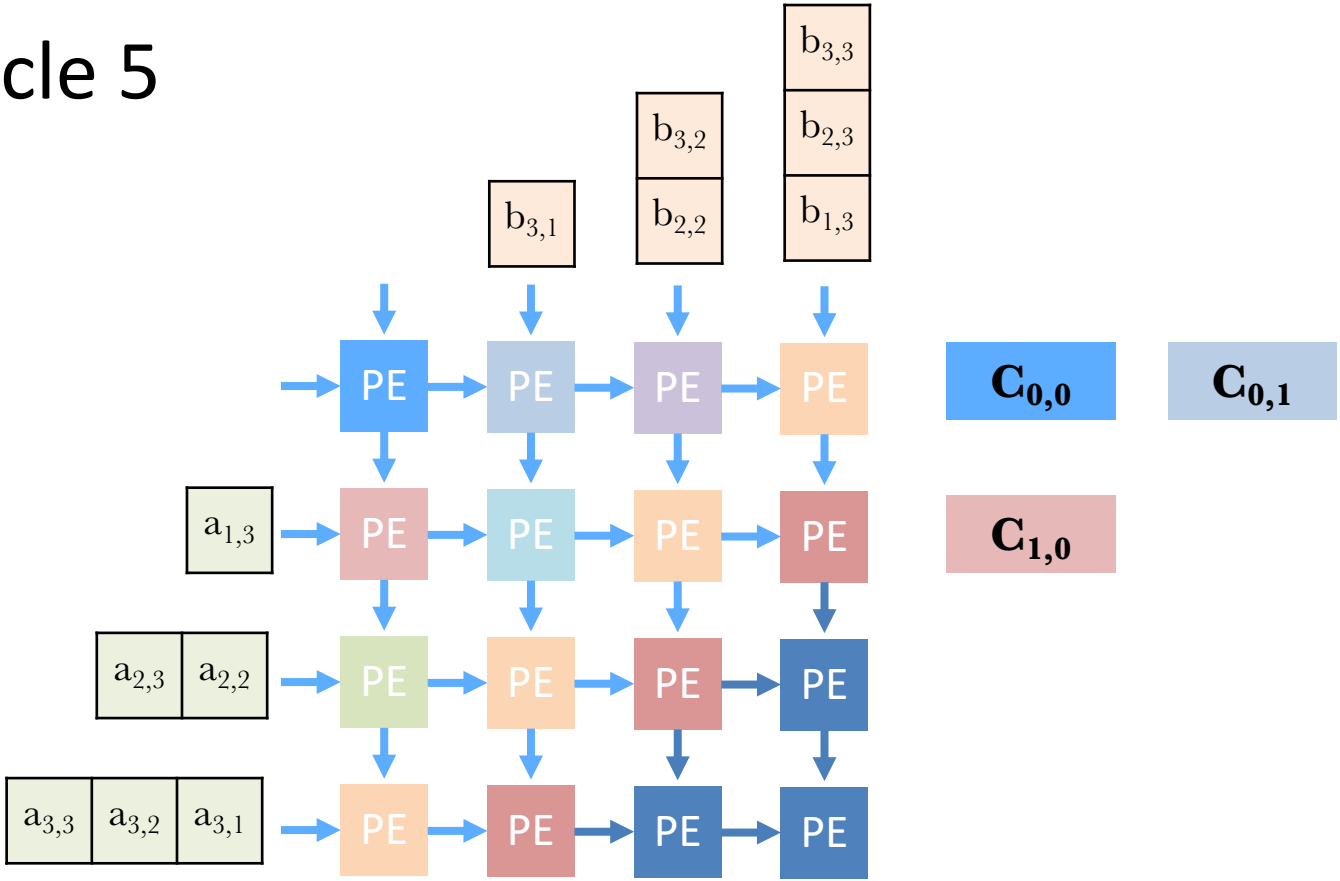
- Cycle 3


b3,3
b2,3
b1,3
b0,3
b3,2
b2,2
b1,2
b0,2
b3,1
b2,1
b1,1
b3,0
b2,0
a0,3
a0,2
a0,0
a1,3
a1,2
a1,1
b0,0
a2,3
a2,2
a2,1
a2,0
a3,3
a3,2
a3,1
a3,0
PE

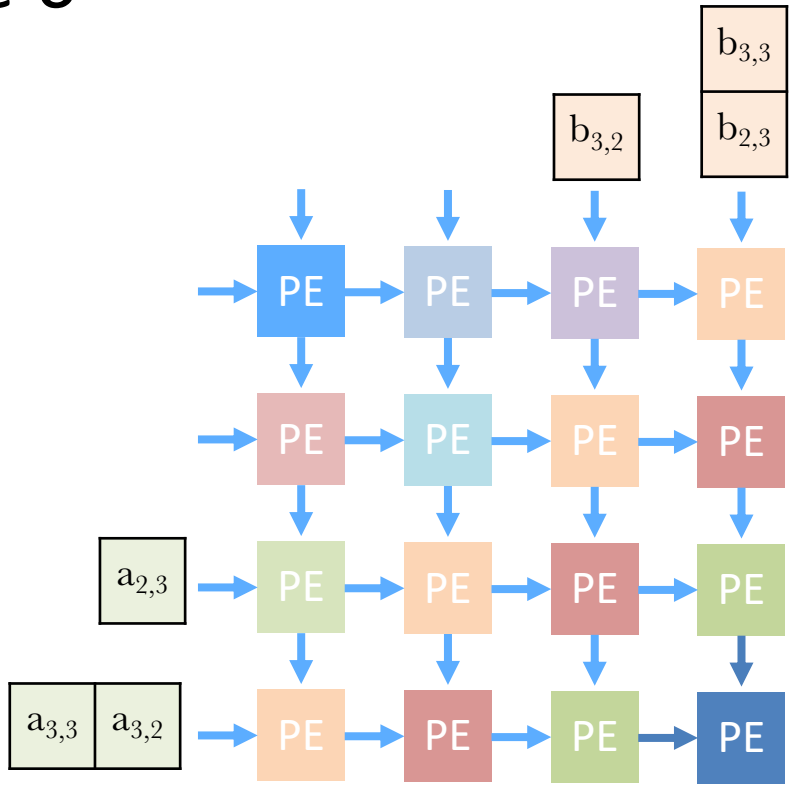
- Cycle 4


b3,3
b2,3
b1,3
b0,3
b3,2
b2,2
b1,2
b3,1
b2,1
b3,0
a0,3
a1,3
a1,2
a2,3
a2,2
a2,1
a3,3
a3,2
a3,1
a3,0
PE
a0,0
b0,0
C0,0

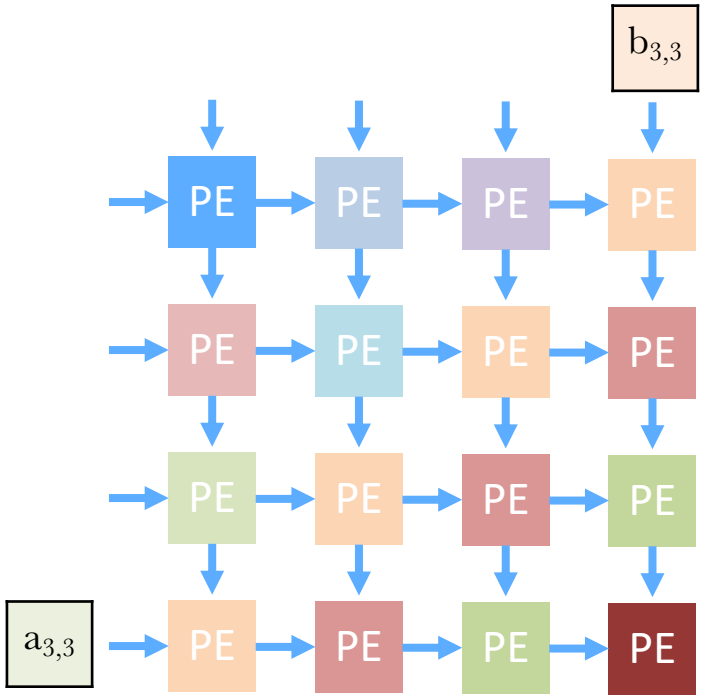
- Cycle 5



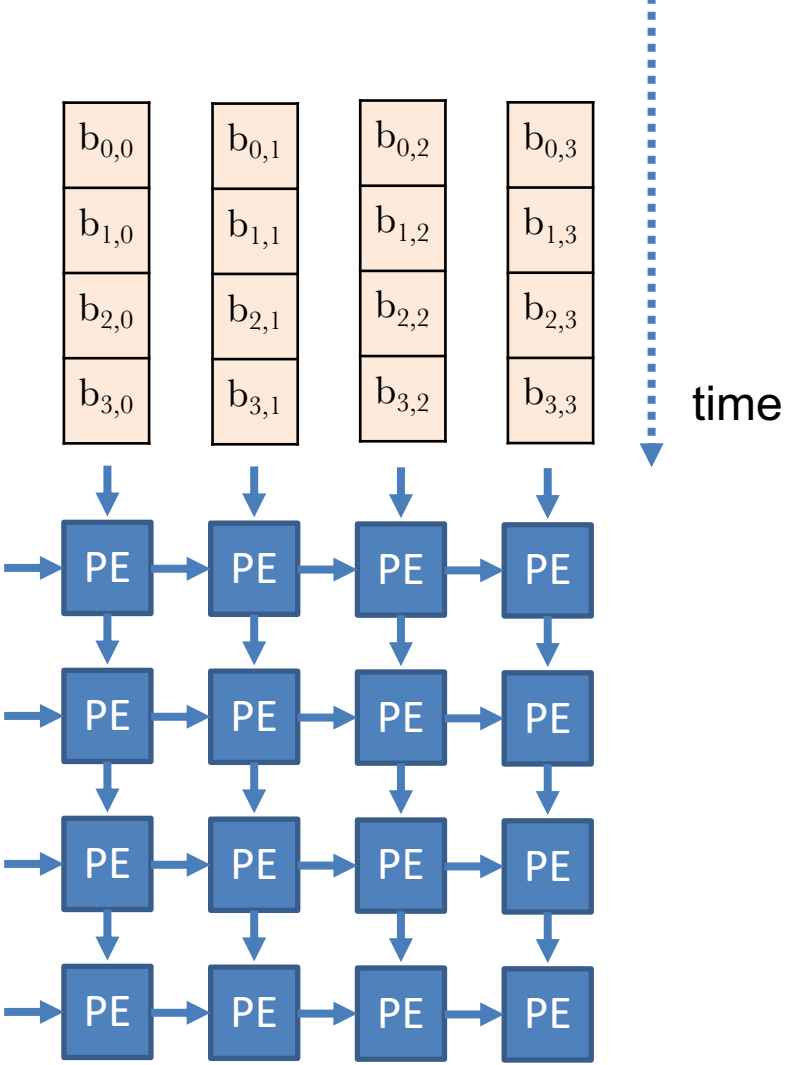
- Cycle 6


b3,3
b2,3
b3,2
PE
PE
PE
PE
PE
PE
PE
PE
a2,3
PE
PE
PE
PE
a3,3
a3,2
PE
PE
PE
PE

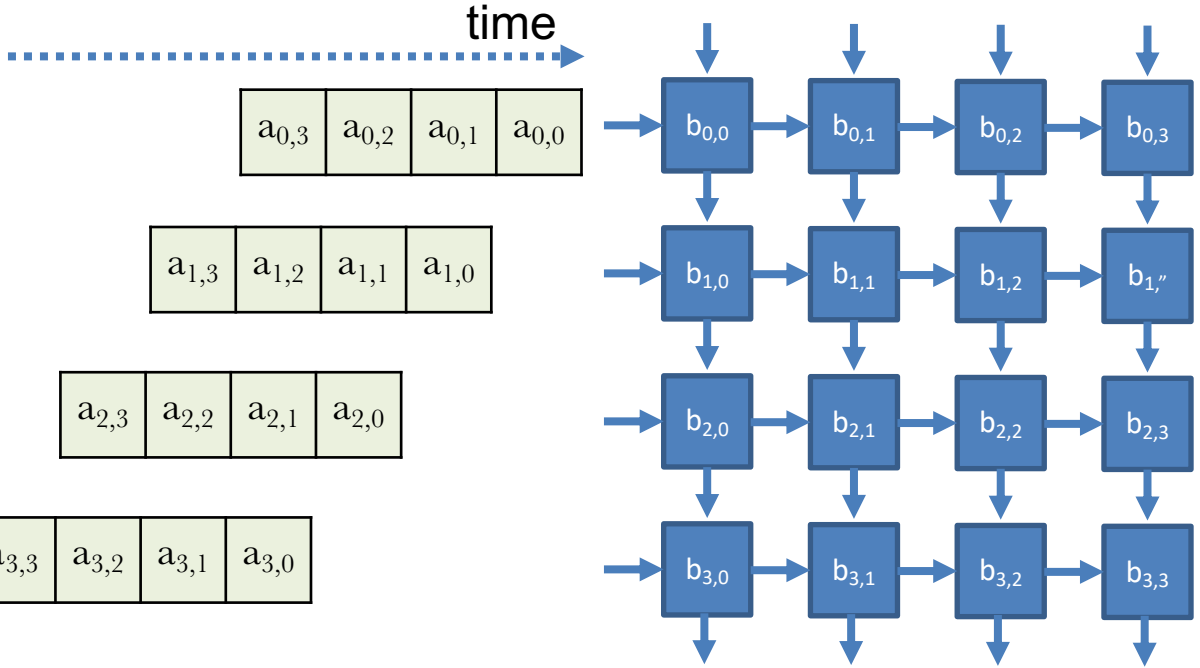
- Cycle 7: all PEs contain $C_{i,j}$ results
- $C_{i,j}$ values can be shifted to last column/row through the PEs at each cycle

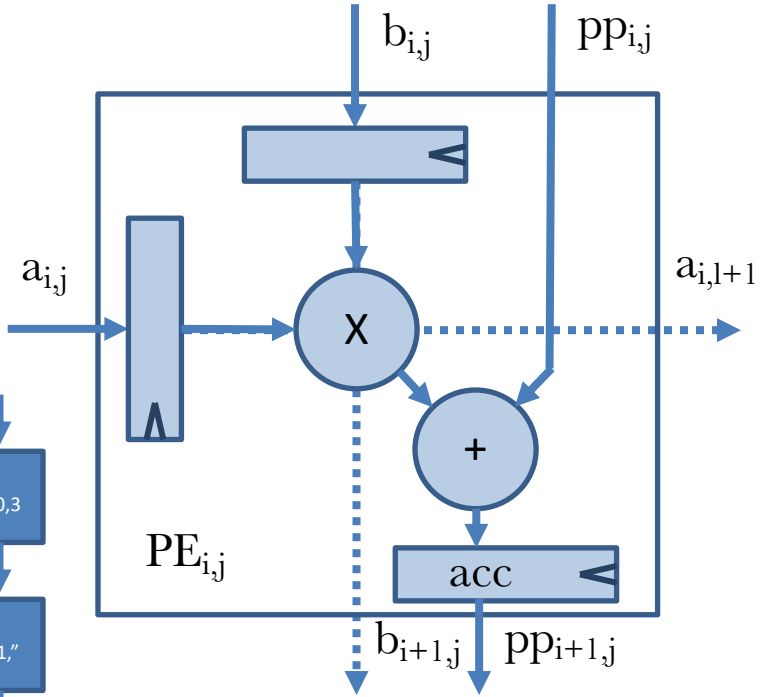


- Weight (B) Stationary
  - First phase: load weights



- Weight (B) Stationary
  - Second phase: matrix multiplication

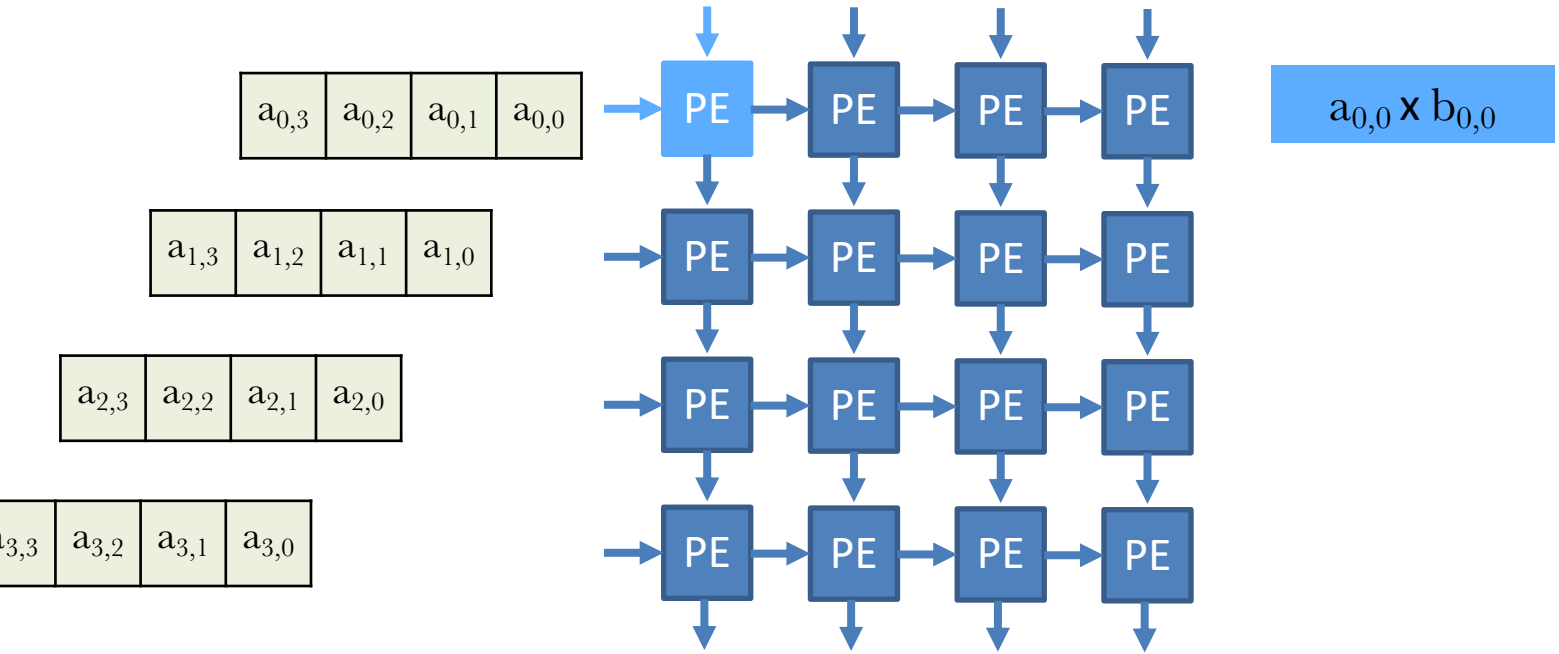




$$pp_{i+1,j} = a_{i,j+1} \times b_{i+1,j} + pp_{i,j}$$

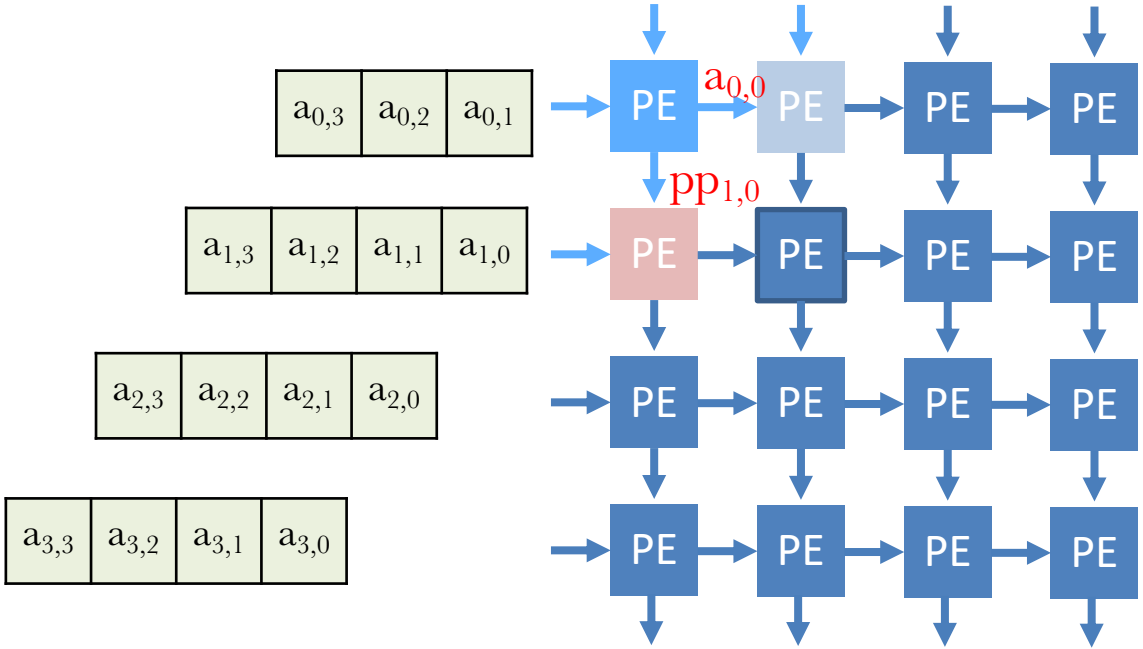
Systolic Array Matrix Multiplication (WS)

- Cycle 1



Systolic Array Matrix Multiplication (WS)

- Cycle 2



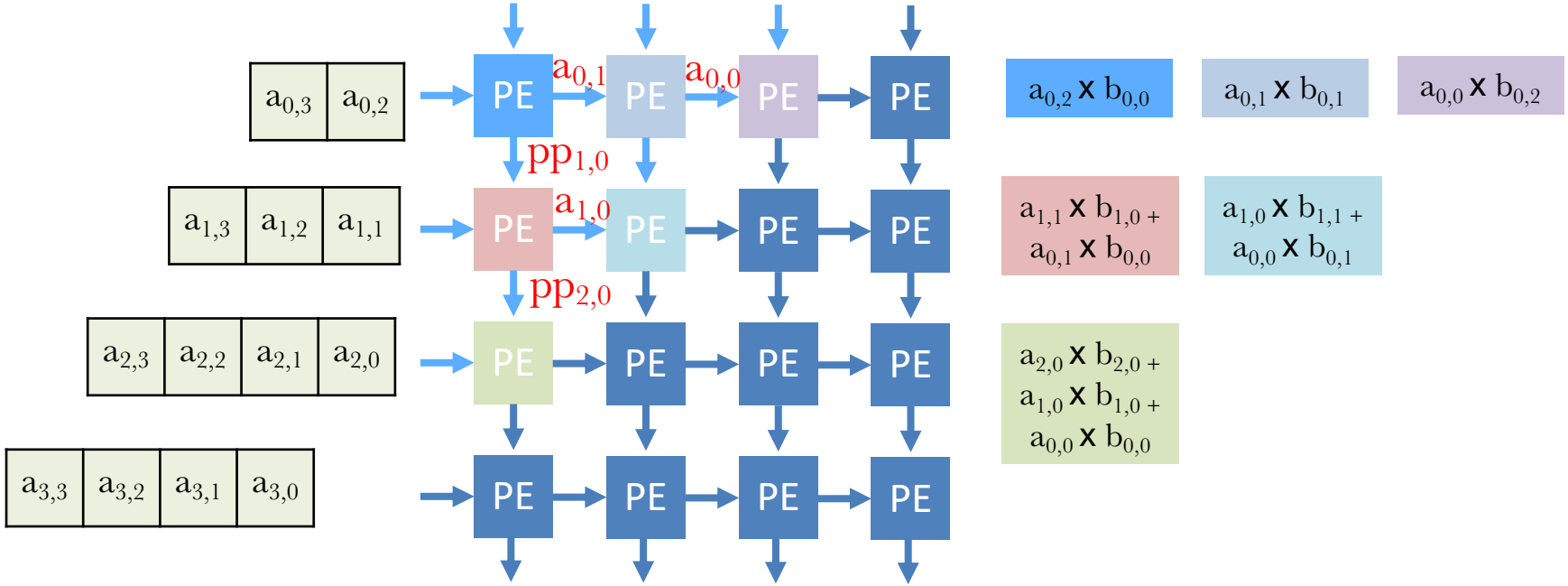
$a_{0,1} \times b_{0,0}$

$a_{0,0} \times b_{0,1}$

$a_{1,0} \times b_{1,0}$ + $a_{0,0} \times b_{0,0}$

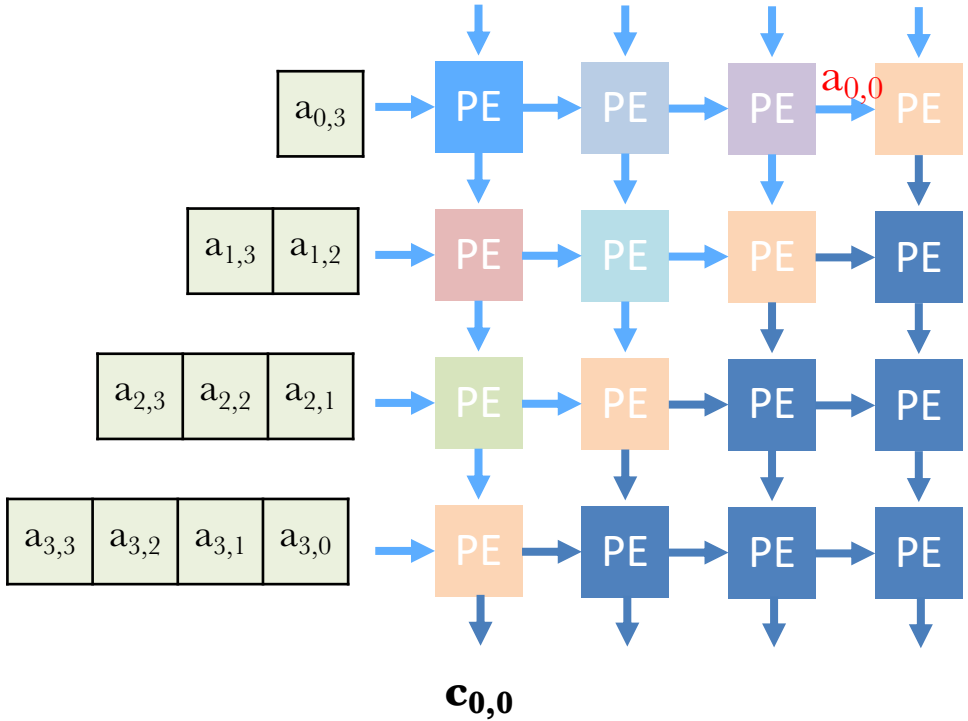
Systolic Array Matrix Multiplication (WS)

- Cycle 3



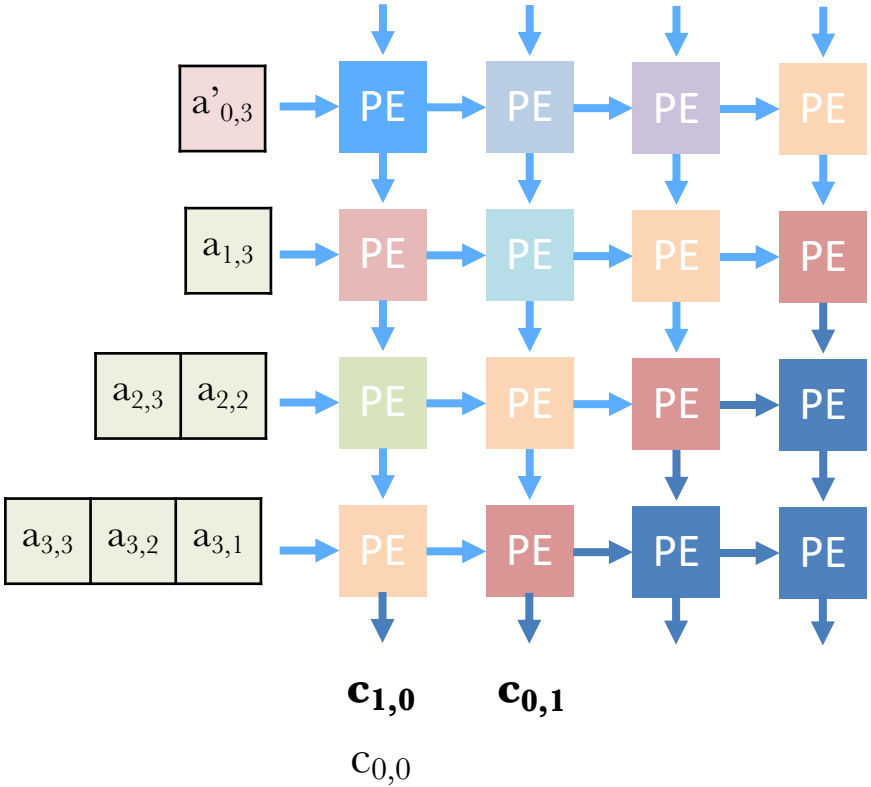
Systolic Array Matrix Multiplication (WS)

- Cycle 4

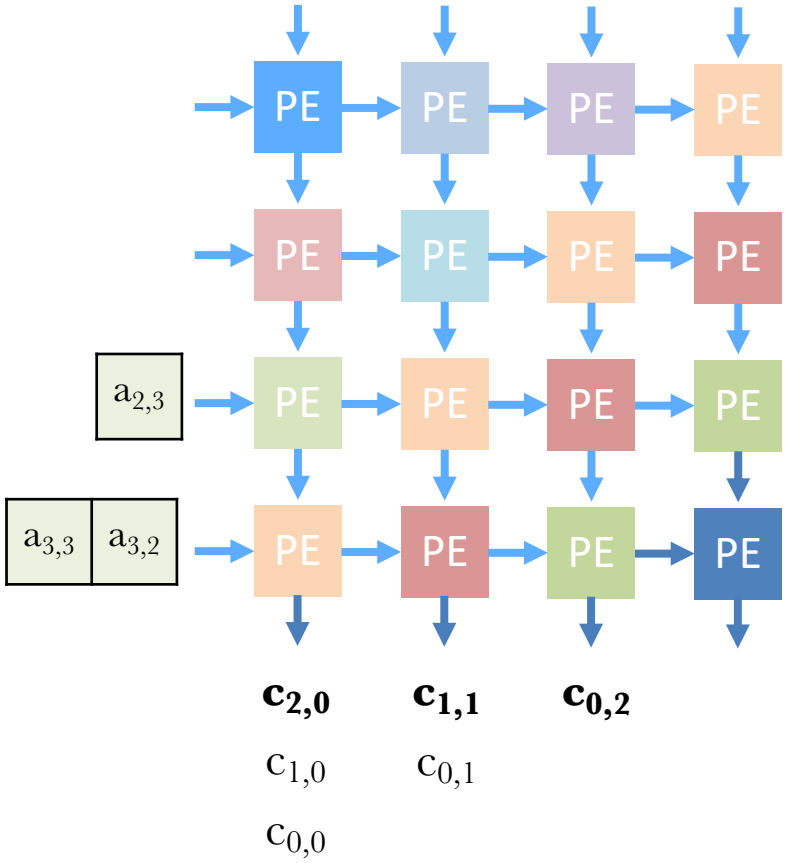


$c_{0,0} =$
$a_{3,0} \times b_{3,0} +$
$a_{2,0} \times b_{2,0} +$
$a_{1,0} \times b_{1,0} +$
$a_{0,0} \times b_{0,0}$

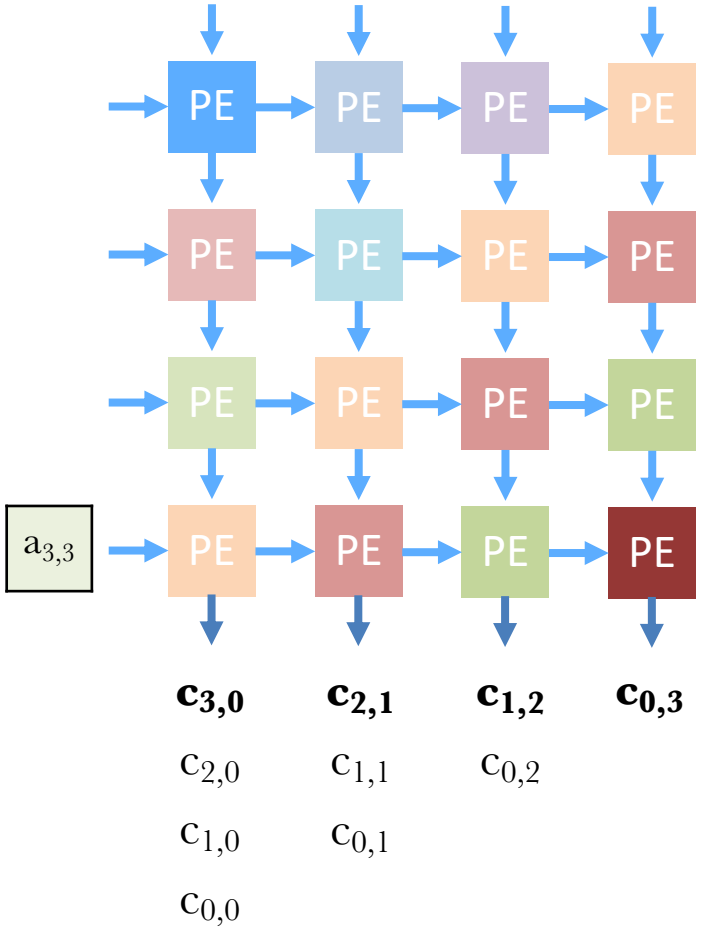
- Cycle 5
  - New inputs can start to be broadcasted



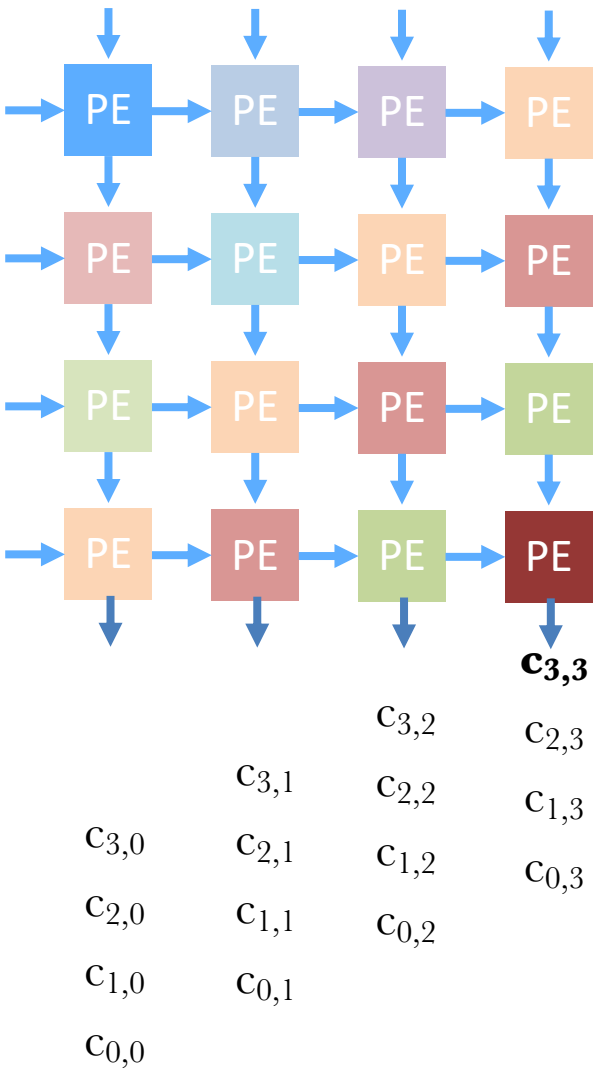
- Cycle 6


PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
a2,3
a3,3
a3,2
c2,0
c1,1
c0,2
c1,0
c0,1
c0,0

- Cycle 7


PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
PE
a3,3
c3,0
c2,1
c1,2
c0,3
c2,0
c1,1
c0,2
c1,0
c0,1
c0,0

- Cycle 10: all $C_{i,j}$ results have been outputted



Outline

- Part I: the clear need for specialized hardware
  - Energy efficiency of hardware accelerators
- Part II: accelerating GEMM
  - DNN kernels with a focus on convolution
  - Speeding up the GEMM kernel
  - Designing hardware accelerators for GEMM/CONV
  - Available accelerators for DNNs
- Part III: computing at the right precision
  - A bit of arithmetic
  - Inference and training with low precision

Accelerators for ML


CPU	GPU	FPGA	TPU
Threads SIMD	Massive Threads SIMD HBM	LUTs DSP BRAM	MM Unit BRAM

- Accelerators: GPU, TPU
- Open-source accelerators: NVDLA, Gemmini
- FPGA: overlays, dataflow (e.g., FINN)

GPU

- NVIDIA Volta GV100 (2017, 14nm)

21B transistors
815 mm²

80 SM
5120 CUDA Cores
640 Tensor Cores

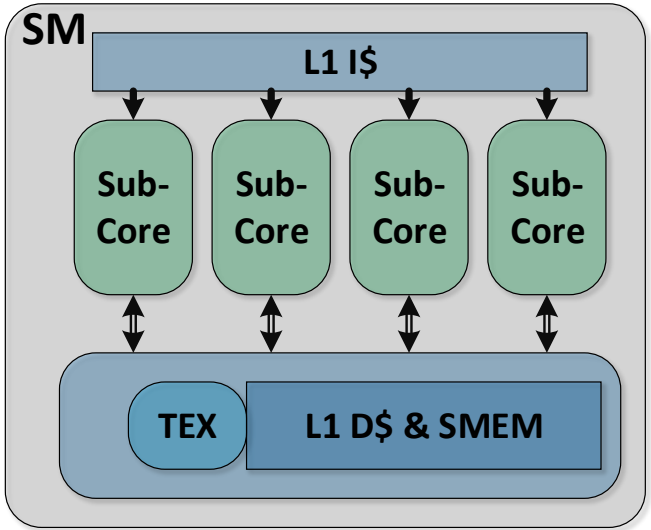
16 GB HBM2
900 GB/s HBM2
300 GB/s NVLink

*full GV100 chip contains 84 SMs

- NVIDIA Hopper GH100 (2022, 4nm)
  - 144 SMs

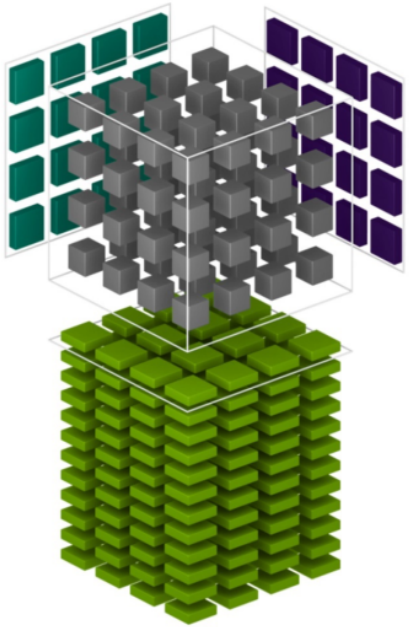
GPU

- Volta GV100: SM's details



NVIDIA Tensor Cores

- Mixed-Precision Matrix Math
  - 4x4 matrices operations in one cycle



Tensor Core Throughputs

Multiply-Accumulates per clock per SM

(multiply by 2x for ops counts)

	FP32	FP16	INT8	INT4	INT1
Volta	64	512			
Turing	64	512	1,024	2,048	8,192

NVIDIA GPU

- Highest Peak Performance

Peak performance (TFLOPS)							
Device	Year	fp64	fp32	tfloat32	fp16	bfloat16	fp8
P100	2016	5	9	-	19	-	-
V100	2017-2019	8	16	-	125	-	-
A100	2020-2021	19	19	156	312	312	-
H100	2022	48	48	400	800	800	1600

- A new chip in less than two years now…

GPU

- GPU have high control overhead

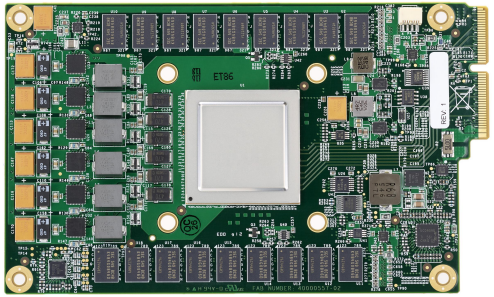
Operation	Energy**	Overhead*
Half-precision Fused Multiply-Add HFMA	1.5pJ	2000%
Half-precision 4-way Dot Product HDP4A	6.0pJ	500%
H.-p. Matrix Multiply and Acc. HMMA	110pJ	27%

- But also have very efficient tensor cores

- Last generation: Hopper GH100 GPU
  - 144 SMs with 128 FP32 cores, 64 FP64 cores, 64 INT32 cores, and four Tensor Cores per SM
  - Support for FP8 format
  - 4nm, 700 Watts...

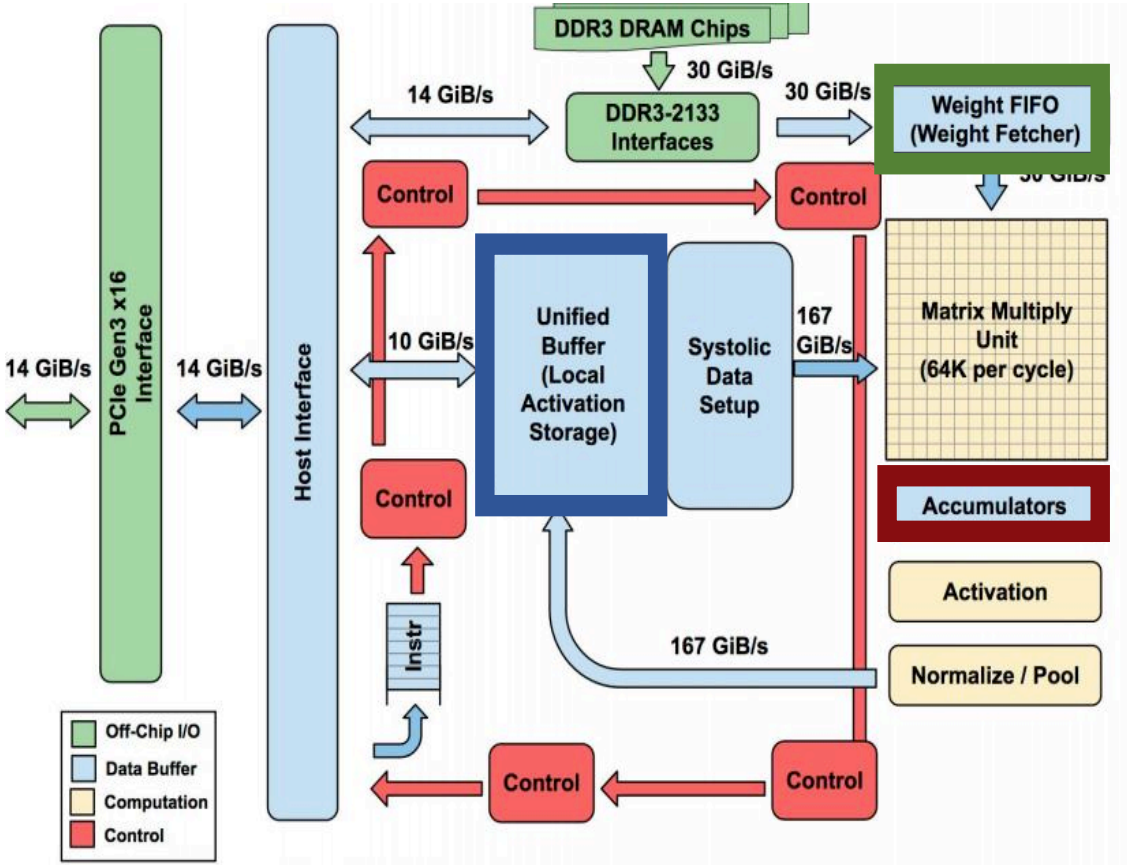
TPU: Tensor Processing Unit (Google)

- TPUv1
  - 2016, 28nm, 700MHz
  - 8GB DDR3, 28MB on-chip mem
  - 75W, 23 TOPS
- TPUv3
  - 2018, 16nm, 940 MHz
  - 32GB HBM, 32 MB
  - 450W, 92 TOPS



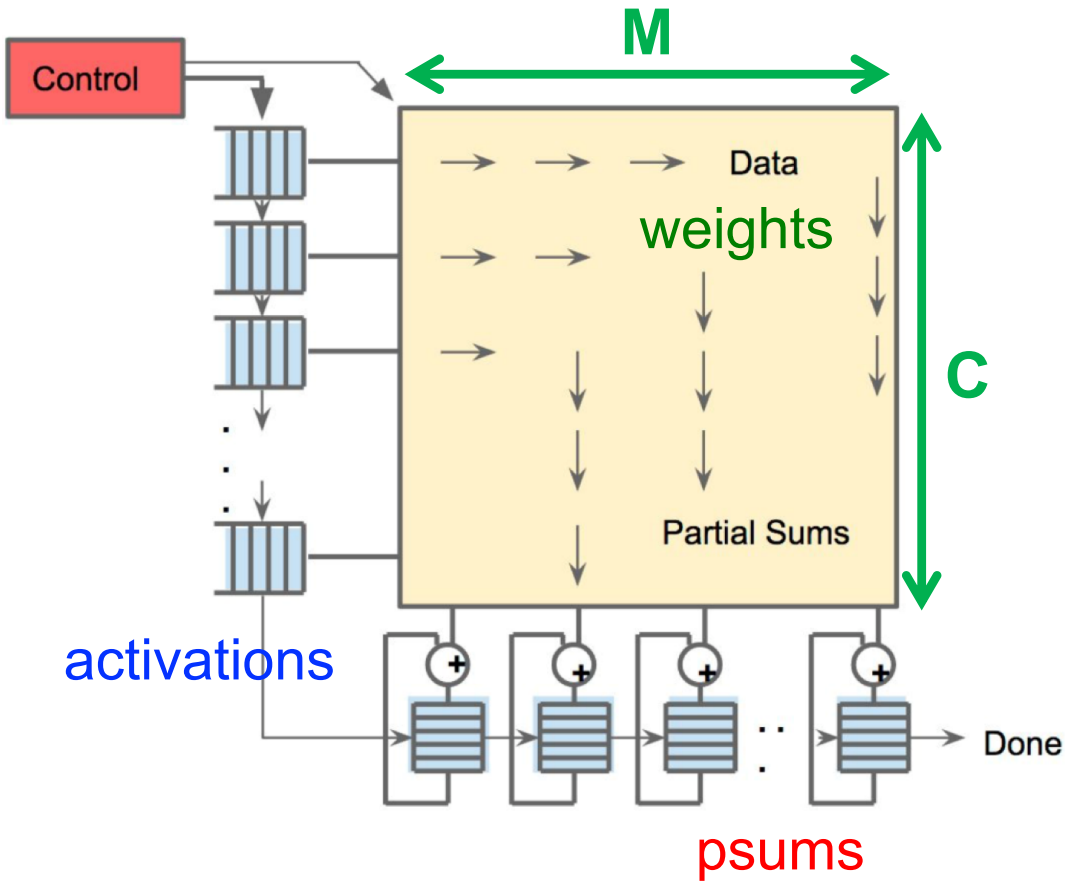
TPU

- Coarse-grained matrix multiply and data read/write instructions
- Compute intensive
  - 64K MACs per cycle
- Memory intensive
  - 4 MB of on-chip Accumulator Memory
  - 24 MB of on-chip Unified Buffer (activation memory)
  - Two 2133MHz DDR3 DRAM, 8 GiB of off-chip (weight DRAM)
- >25X as many MACs vs GPU



TPU

- Matrix Multiply Unit: 256x256 (65,536) 8-bit MAC as a systolic array
  - Peak: 92 TOPS
    - 2x65,536x700MHz
- Datapath:
  - Parallel-K: systolic accumulation
  - Parallel-N: systolic multicast
- Memory
  - Custom systolic registers
  - Dedicated accumulation and weight buffers
  - Double-buffered, weight-stationary dataflow



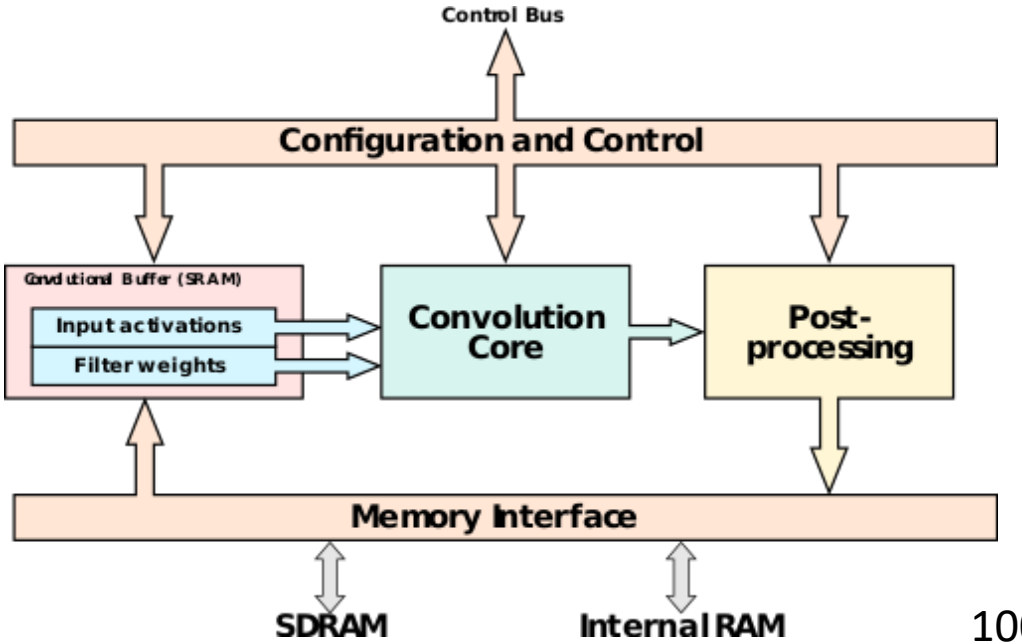
Accelerators for ML


CPU	GPU	FPGA	TPU
Threads SIMD	Massive Threads SIMD HBM	LUTs DSP BRAM	MM Unit BRAM

- Accelerators: GPU, TPU
- Open-source accelerators: NVDLA, Gemmini
- FPGA: overlays, dataflow (e.g., FINN)

NVDLA

- NVIDIA Deep Learning Accelerator
  - 8-16 bit datapath, weight compression
  - Int8, Int16, FP16
  - large-small config.
  - open-source
    - system Verilog

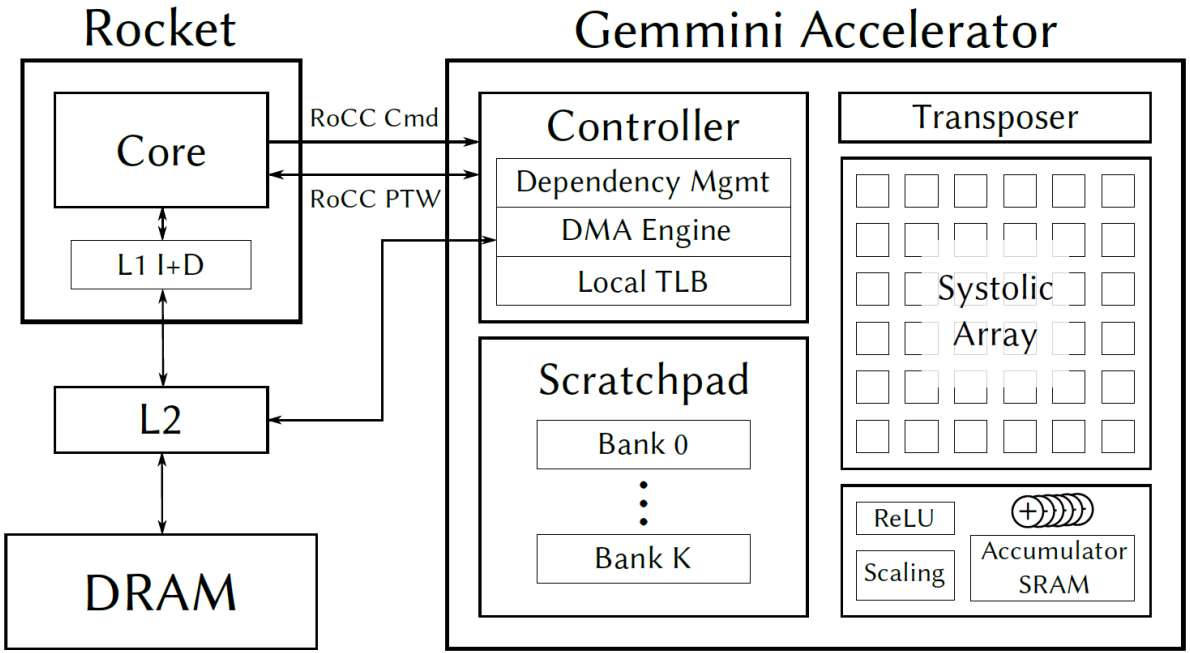


Gemmini

[Gemmini, DAC 2021]

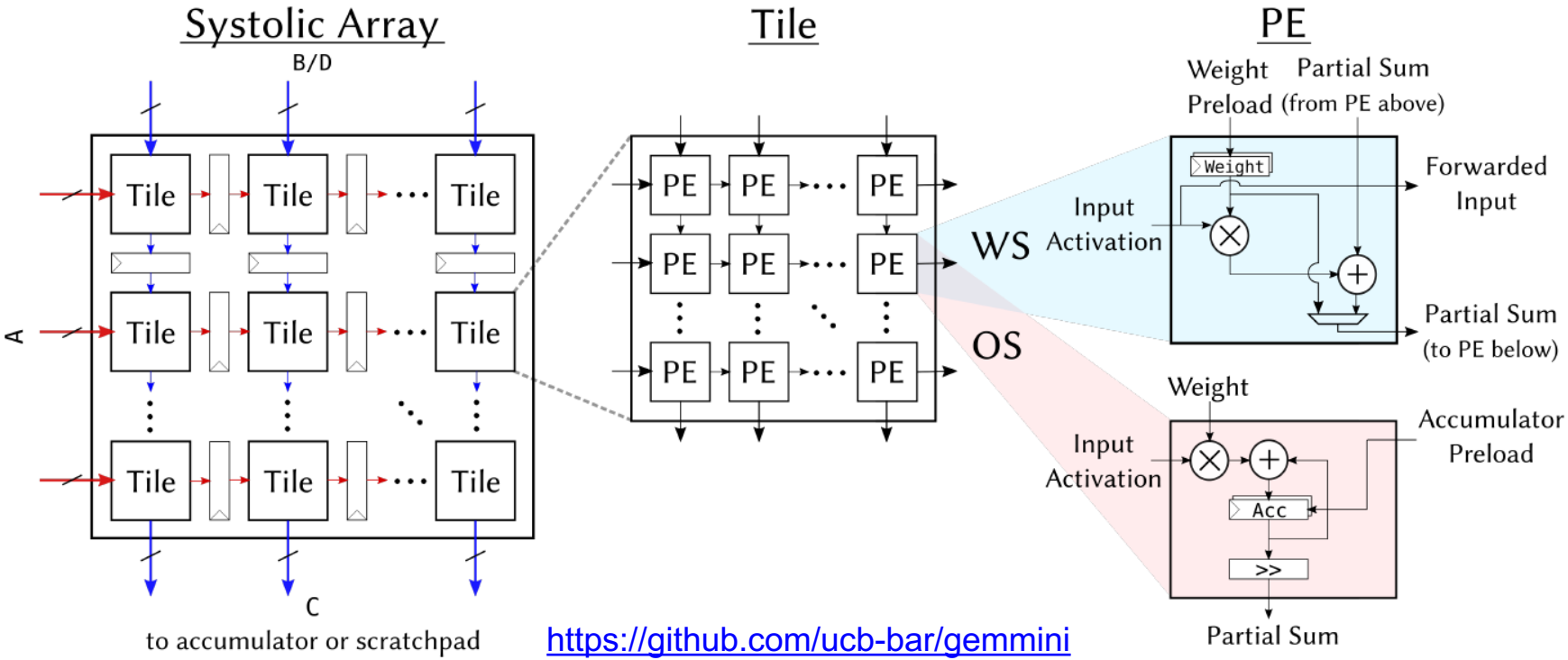
https://github.com/ucb-bar/gemmini

- UC Berkeley
  - open-source (Chisel), systolic array



Gemmini

- Weight-Stationary or Output-Stationary dataflow



https://github.com/ucb-bar/gemmini

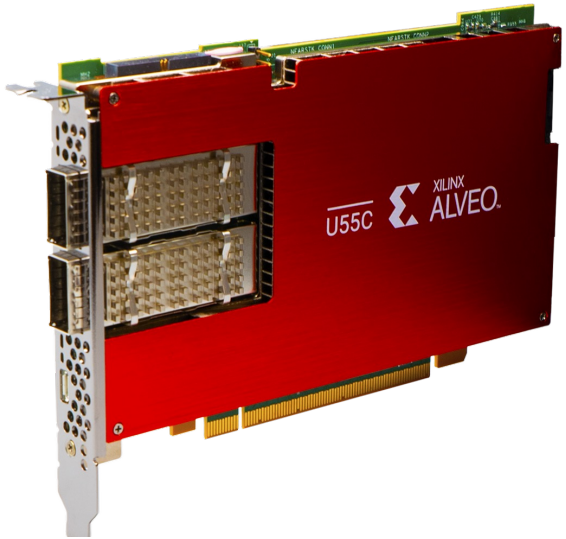
Accelerators for ML

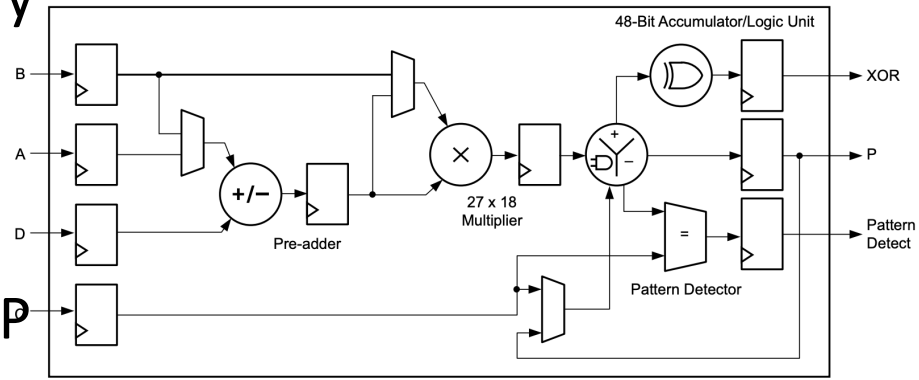

CPU	GPU	FPGA	TPU
Threads SIMD	Massive Threads SIMD HBM	LUTs DSP BRAM	MM Unit BRAM

- Accelerators: GPU, TPU
- Open-source accelerators: NVDLA, Gemmini
- FPGA: overlays, dataflow (e.g., FINN)

FPGA

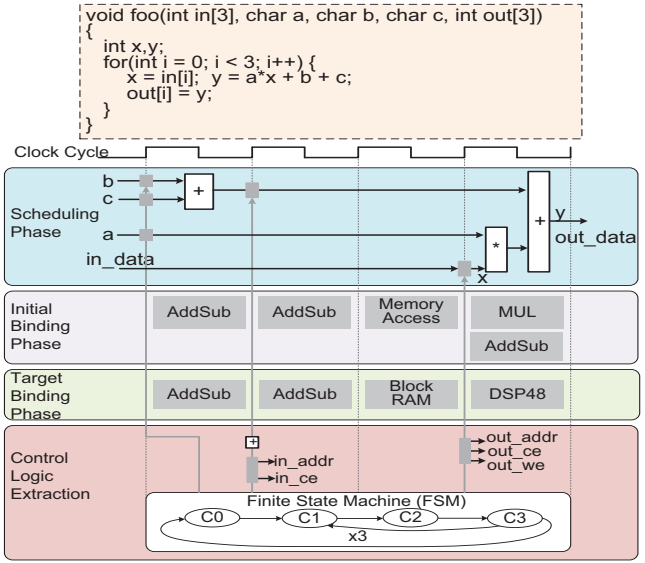
- Xilinx Alveo U55C card
  - PCIe® Gen3x16 or dual Gen4x8
- Ethernet 2 x 100 Gb/s
- XCU55 UltraScale+ FPGA
  - 16 GB High-Bandwidth Memory (HBM2), 460GB/s bandwidth
  - 1.3M CLBs
  - 270+70M BRAM
  - 9K DSP blocks, 4 INT8 MAC/DSP
  - 28 TOPS INT8 (peak, 800MHz)

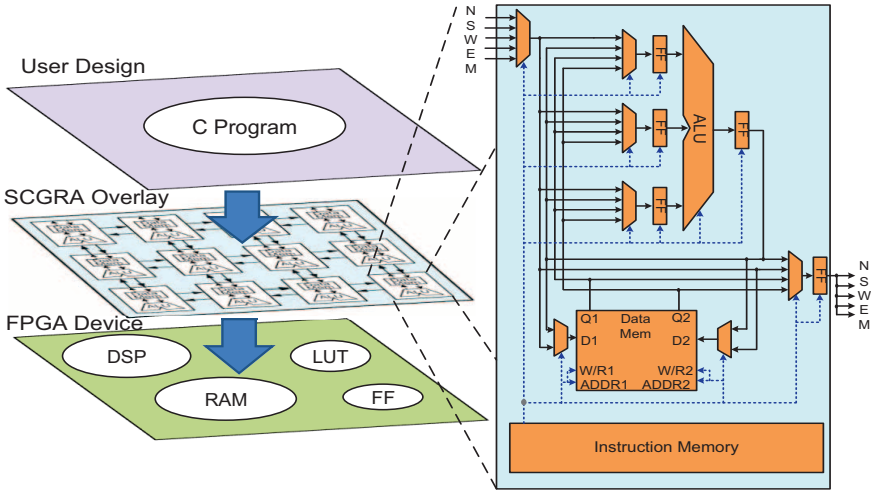




HLS vs. Overlays

- HLS synthesizes C-like high-level design and performs code transformations and synthesis optimizations
- FPGA overlay is a coarse-grained design abstraction layer over fine-grained FPGA resources



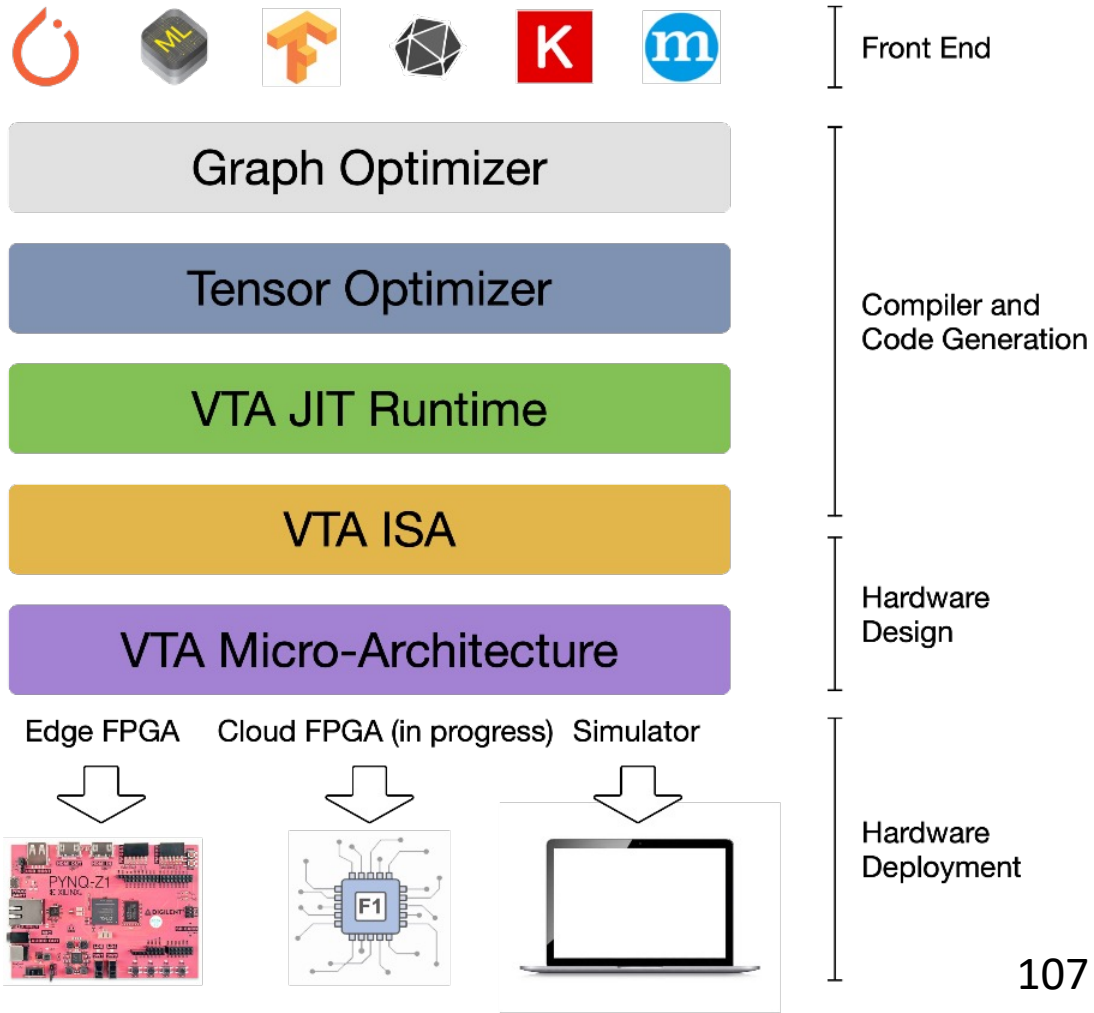


HLS vs. Overlays

- HLS synthesizes C-like high-level design and performs code transformations and synthesis optimizations
  - FINN (Xilinx): SIMD + parallel Matrix Vector Activate Unit
  - HLS4ML: "true" HLS style + reuse
  - HLS Libraries: e.g., Vitis Accelerated Blas Library, GEMM_HLS, HLS_LIB, AC_ML
- FPGA overlay is a coarse-grained design abstraction layer over fine-grained FPGA resources
  - Xilinx Deep Learning Processing Unit (DPU) + Vitis AI software stack + runtime (XRT)
  - VTA+TVM

TVM+VTA

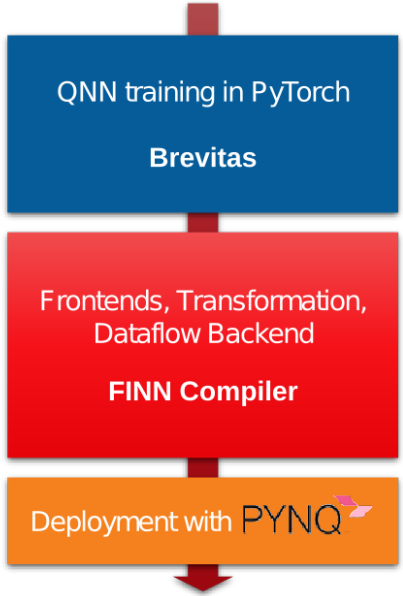
- End-to-end hardware-software DL system stack
  - hardware design
  - sequential architecture
  - drivers, JIT runtime
  - optimizing compiler stack based on TVM



FINN

- Experimental framework from Xilinx Research Labs
  - Provides an HLS library of standard NN layers
  - Design space exploration of QNN accelerators on FPGAs
  - Generates HLS code that supports a wide range of precisions

- Inference only, focused on Quantized NNs
  - Pipelined dataflow architecture
  - AXI controlled dataflow structure for low latency



Outline

- Part I: the clear need for specialized hardware
  - Energy efficiency of hardware accelerators
- Part II: accelerating GEMM
  - DNN kernels with a focus on convolution
  - Speeding up the GEMM kernel
  - Designing hardware accelerators for GEMM/CONV
  - Available accelerators for DNNs
- Part III: computing at the right precision
  - Number representation and precision
  - Inference and training with low precision

Computing just right!

(with the right accuracy and precision)

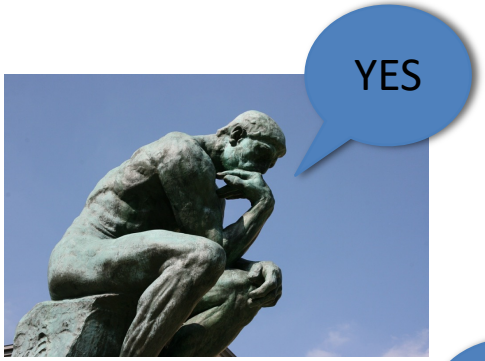
Number Representations and Precision

- Energy, delay, and area vary a lot between numeric formats and word-length

	Addition	Multiplication
8-bit integer	0.03pJ / 36μm^2	0.2pJ / 282μm^2
32-bit float	0.9pJ / 4184μm^2	3.7pJ / 7700μm^2

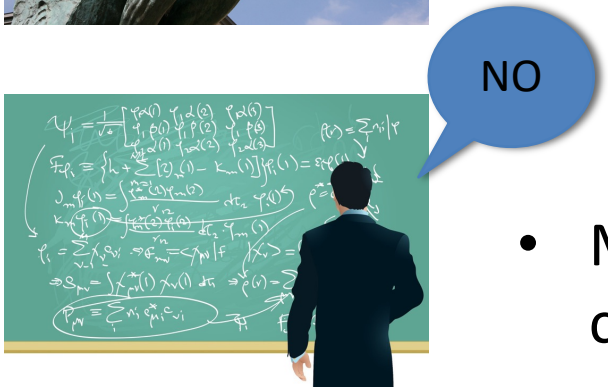
Humans Approximate…. But Computers Do Not!

$$\frac{923}{21} > 1.75$$



- No need for complex computation

$$\frac{923}{21} > 45$$



- More accurate computation is required

- High-precision computations often lead to inefficiency

Resilience of ANN?

Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoatnt tihng is taht the frist and lsat ltteer be at the rghit pclae. And we spnet hlaf our lfie larennig how to splel wrods. Amzanig, no! [O. Temam, ISCA10]

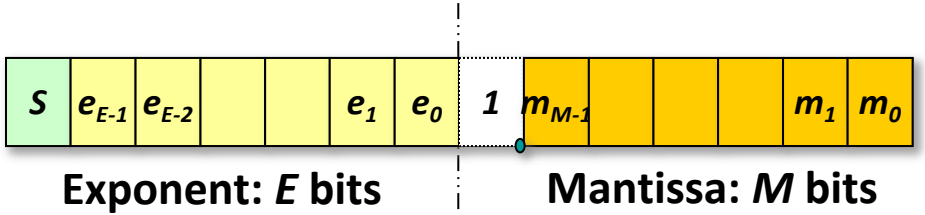
- Our biological neurons are tolerant to computing errors and noisy inputs
- Quantization of parameters and computations provides benefits in throughput, energy, storage

Number Representations

- Floating-Point (FlP)

$$x = (-1)^s \times m \times 2^{e-127}$$

s: sign, *m*: mantissa, *e*: exponent



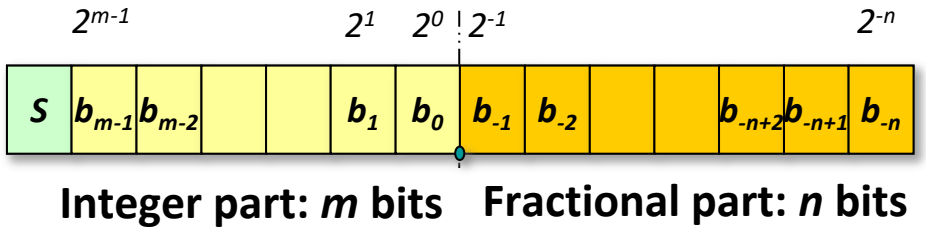
  - Easy to use
  - High dynamic range
  - Hardware cost and power

- Fixed-Point (FxP)

$$x = p \times K$$

p: integer, $K=2^{-n}$: fixed scale factor

  - Integer arithmetic
  - Efficient operators
    - Speed, power, cost
  - Harder to use...



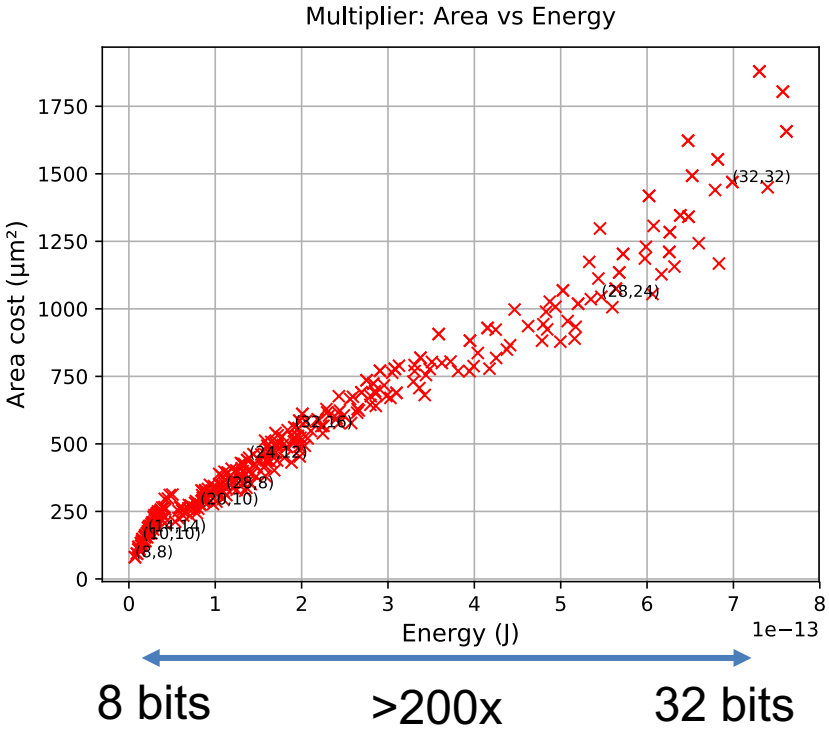
What can be Customized?

- Floating-Point (FlP)
- Precision
  - Exponent (*E*) and Mantissa (*M*) bit-width
  - *e* and *m* both impact accuracy
- Play with exponent bias
- Sub-normal numbers or not?
- 0, ∞, NaN?
- Rounding modes
  - stochastic, to nearest, truncation, to 0/∞

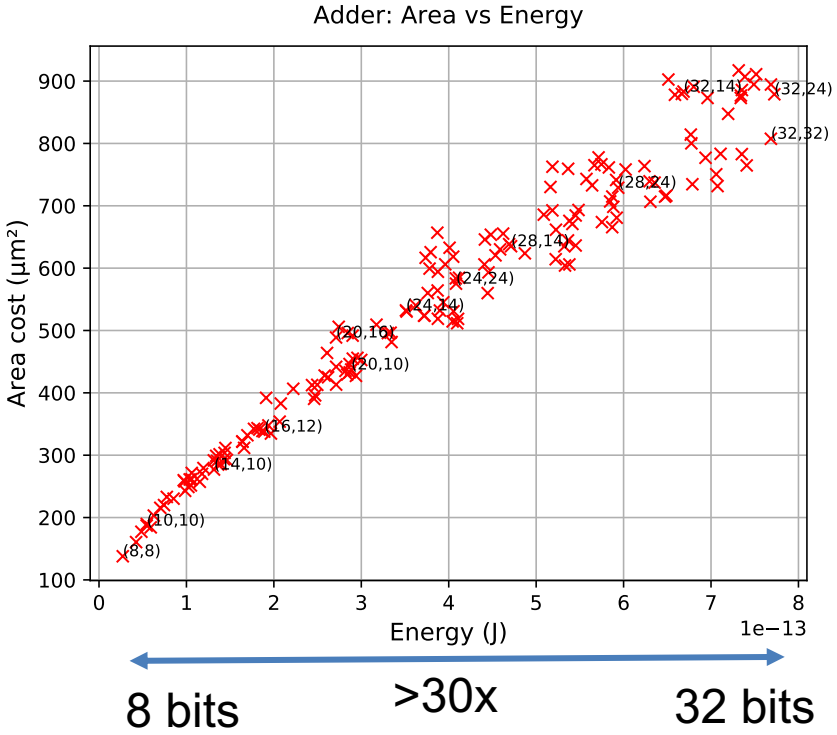
- Fixed-Point (FxP)
- Precision
  - Integer (*m*) and fractional (*n*) bit-width
  - *n* impacts accuracy
  - *m* impacts dynamic range
  - Wordlength (*W=m+n+1*)
- Rounding modes
  - stochastic, to nearest, truncation, to 0/∞
- Saturation modes
  - wrap, max/min

Energy Gains of Low Precision

- Multiplier (float)

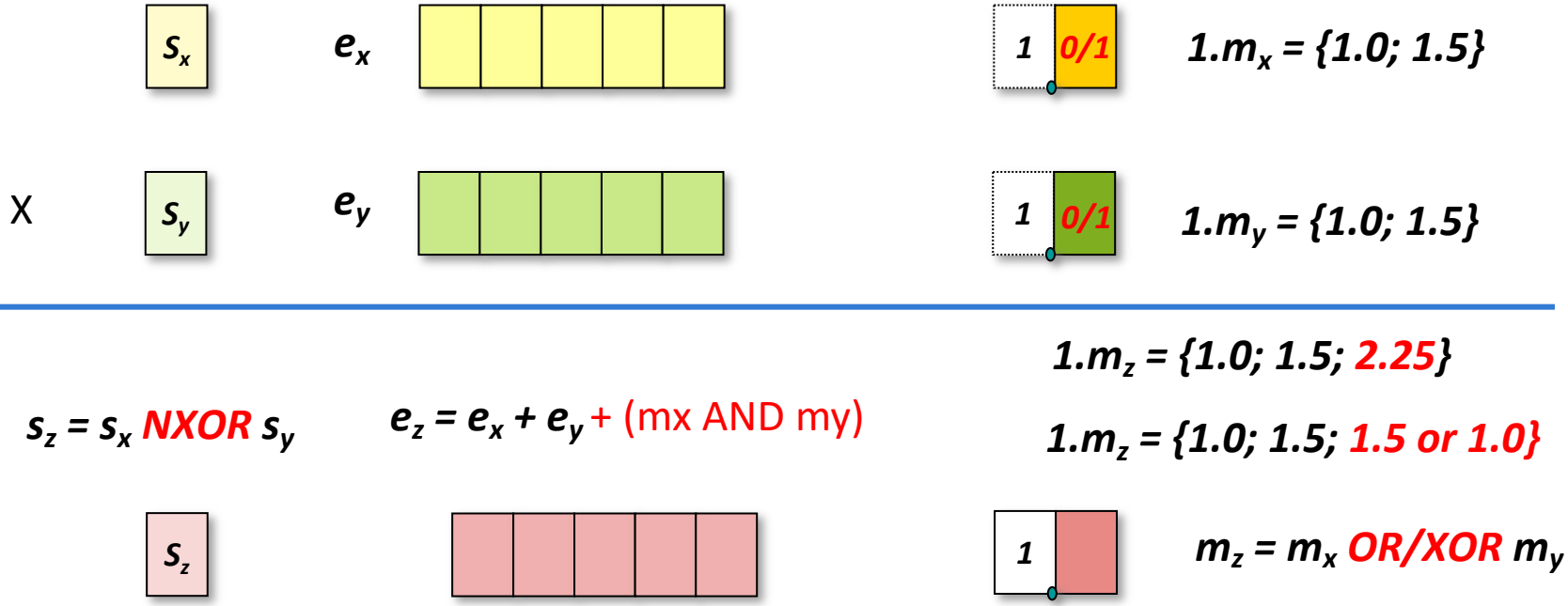


- Adder (float)



Very Low-Precision *float* Multiplication

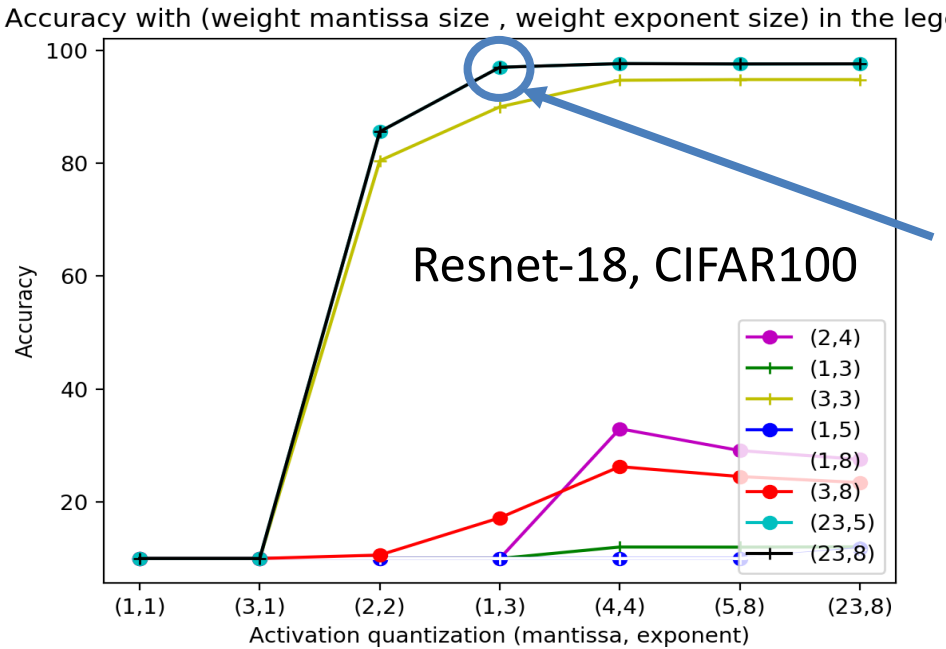
- Example: 7 bits (7,5,1)



How does this apply to DNNs?

Low-Precision Inference

- Not only Weights, but also Activations, Per-Layer Quantization, etc.



4-bit activations and
10-bit weights keeps
accuracy near (98.4%)
32-bit float reference

Even Worse for Training…

- Carbon footprint of DNN training

Analyzing the carbon footprint of current natural-language processing models shows an alarming trend: ***training one huge model for machine translation emits the same amount of CO2 as five cars in their lifetimes (fuel included)***

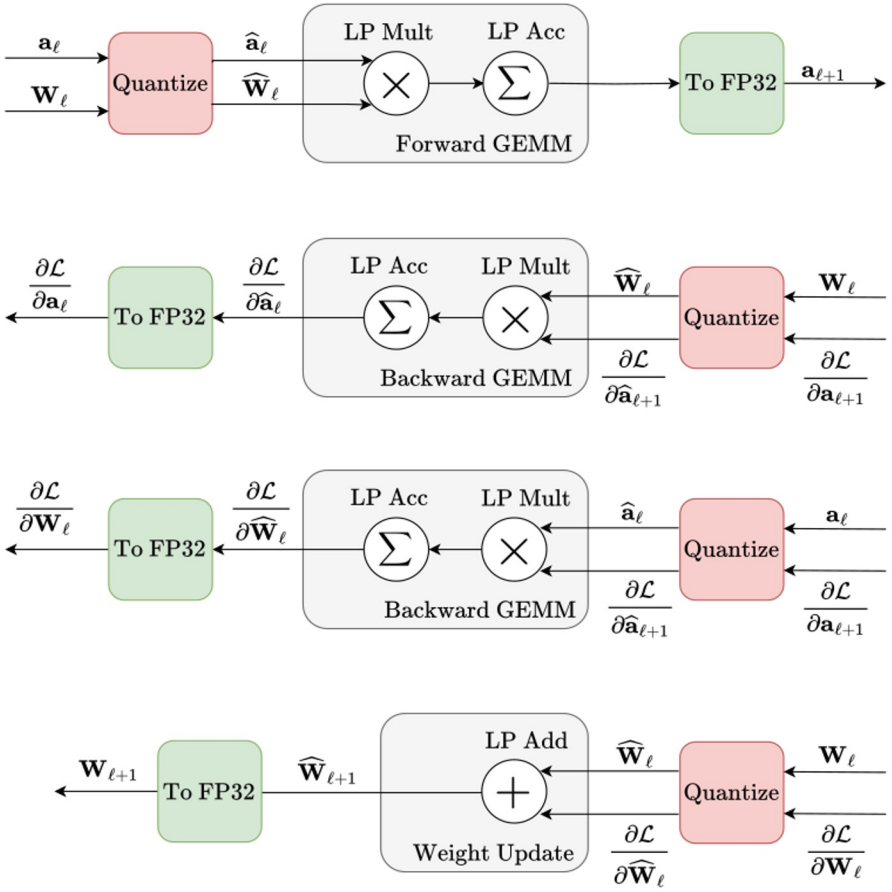
[Strubell *et al.*, ACL 2019]

- Many more operations than inference
- More pressure on memory access
- Much more difficult to accelerate

Need for a Significant Reduction of the Carbon Footprint of Neural Network Training Hardware

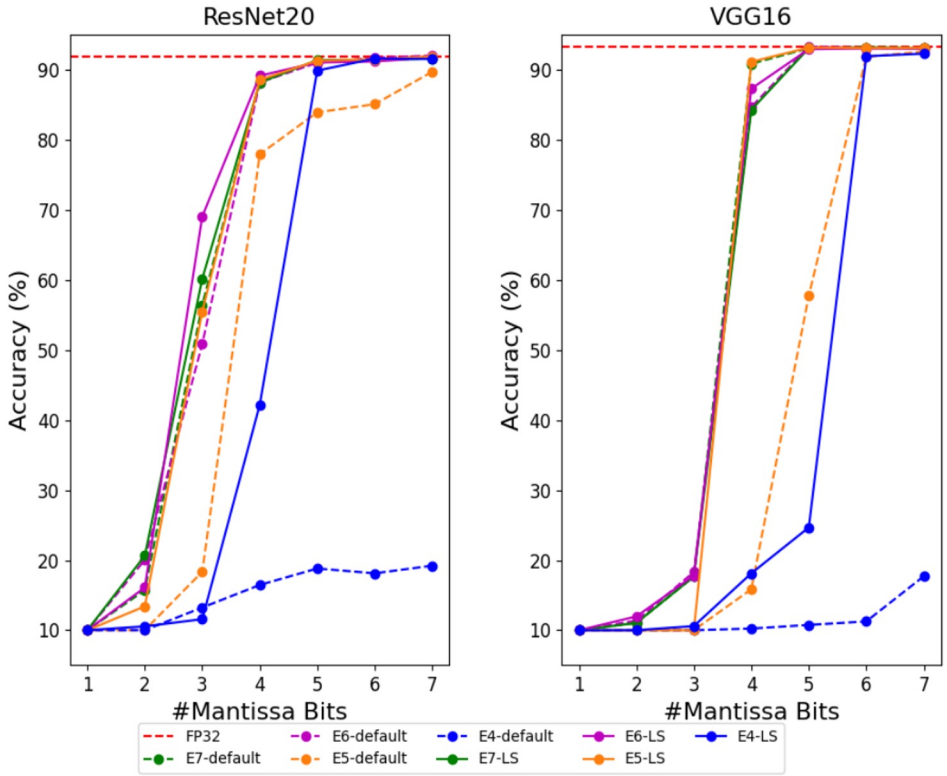
Mixed Precision DNN Training

- GEMMs and weight updates are performed in custom precision
- GEMMs are performed on GPU or FPGA kernels
- Different arithmetic configurations for FWD, BWD and WU operations



Mixed Precision DNN Training

- We can reach baseline accuracy using lower precision
- Too low of a precision could lead to divergence
- A Performance-Precision equilibrium must be found

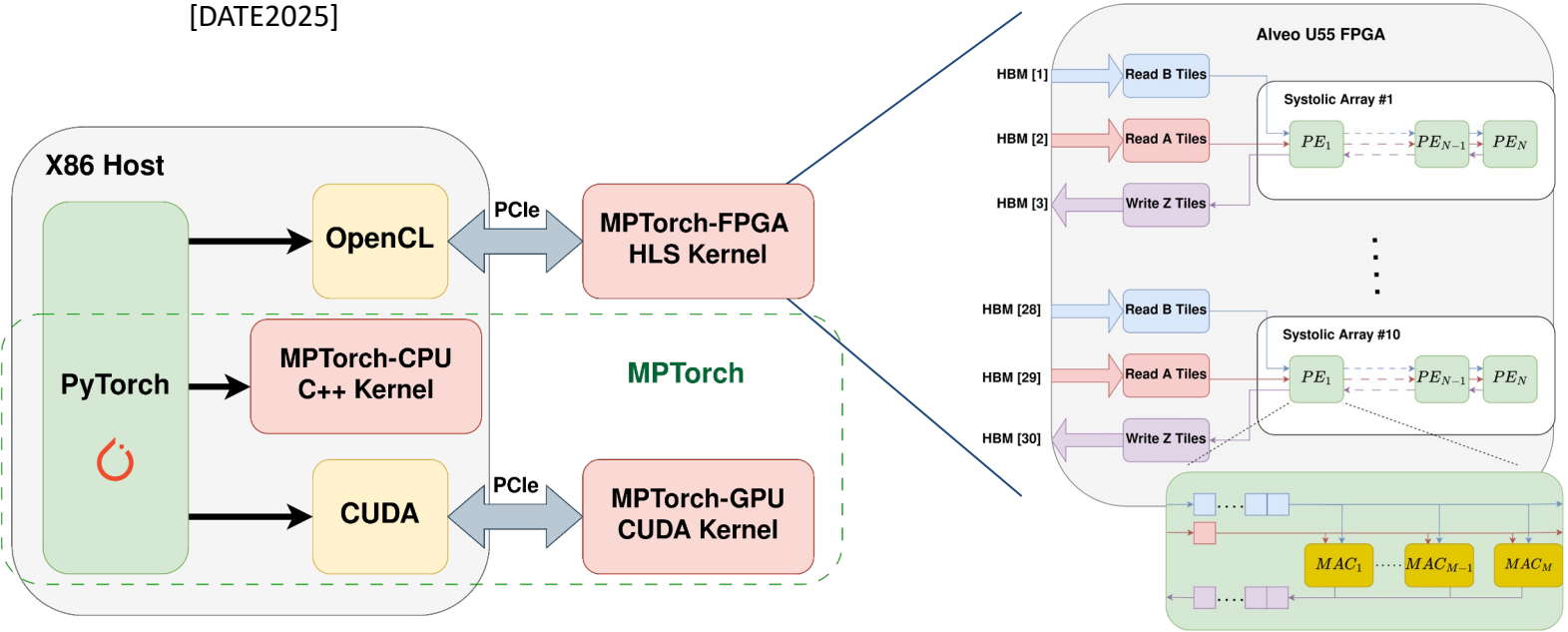


[FPT2022]

Putting It All Together

- MPtorch-FPGA: a Custom Mixed Precision Framework for FPGA-based DNN Training

[DATE2025]



Systolic Array: *C* cores of *N* PEs of *M* MAC units

Results

- Optimize <N,M,C,F> configuration for a given DNN model
- Explore model accuracy for different arithmetic configurations
  - FP8 multiplier, FP12 (E6M5) with Stochastic Rounding
  - with less than 2% accuracy loss w.r.t FP32

FP8/FP12 SR

FP32 baseline

Multiplier	Accumulator	LeNet5 †	ResNet20 ‡	VGG16 ‡	ResNet50 Δ
	E6M5-RZ	97.10	10.00	10.00	10.00
	E6M5-RO	98.00	10.00	10.00	10.00
E5M2-NR	E6M5-RN	98.61	10.00	10.00	10.00
	E6M5-SR	99.00	90.55	88.99	80.88
	E5M10-RN	99.05	91.24	89.81	82.97
E8M23-RN	E8M23-RN	99.18	91.91	90.67	82.92
FXP4, 4-RN		99.06	10.00	10.00	10.00
FXP4, 4-SR		99.14	10.00	10.00	10.00
FXP4, 4-RZ	FXP8, 8	98.85	10.00	10.00	10.00
FXP4, 4-RO		10.00	10.00	10.00	10.00

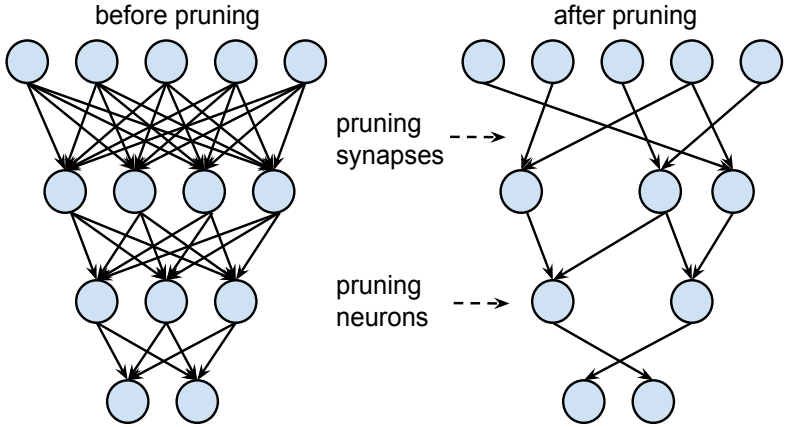
Datasets: †MNIST, ‡CIFAR10, ΔImagewoof

What's next?

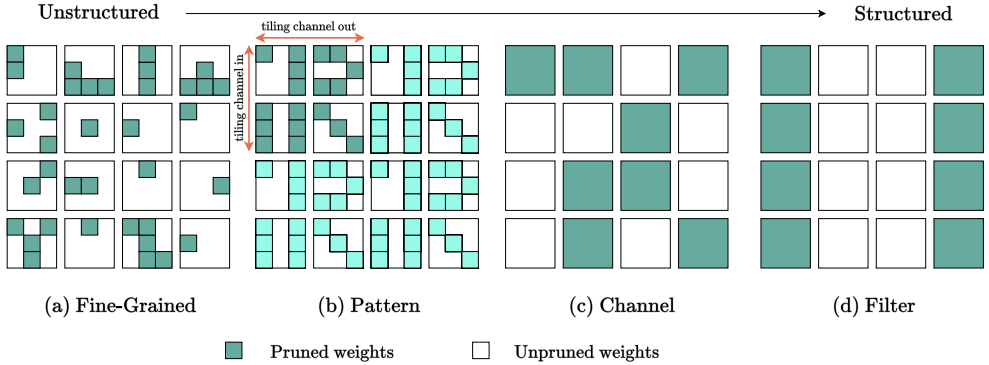
- Efficiency of hardware specialization
  - Domain-specific architectures and languages
- Computing just right
  - @design-time or @run- time
- Hardware-aware optimizations are mandatory
  - Deep knowledge of the hardware is required to propose energy-efficient DNN models

Pruning

a	b	c	d	e
f	0	0	i	0
0	l	m	0	o
p	0	r	0	t
u	v	0	x	y



- Does pruning always translate into energy savings?
- Solution: structured pruning



Backup Slides

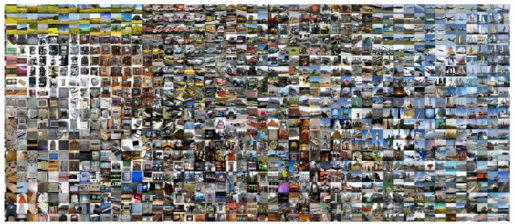
Key Metrics

- Accuracy
  - Evaluate using the appropriate DNN model and dataset
- Programmability
  - Support multiple applications
- Throughput / Latency
  - GOPS, frame rate, delay
- Energy / Power
  - Energy per operation and memory access
- DRAM Bandwidth
- Area Cost (memory size, # of cores)

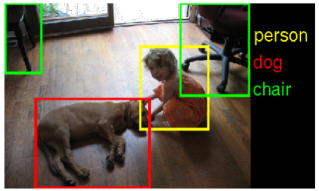
- Number of FLOPs, MACS, and Weights are not a good proxy for energy and latency

MNIST

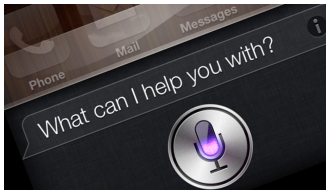
IMAGENET

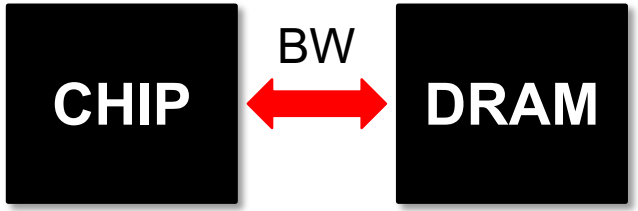


Computer Vision



Speech Recognition





[Eyeriss tutorial]

NVDLA

- Small configuration

INT8 MACs (# instances)	Conv. Buffer (KB)	Area (mm2)	Memory BW (GB/s)	ResNet50		
				Perf (frames/s)	Power (mW)	Power Eff. (DL TOPS/W)
2048	512	3.3	20	269	388	5.4
1024	256	1.8	15	153	185	6.3
512	256	1.4	10	93	107	6.8
256	256	1.0	5	46	64	5.6
128	256	0.84	2	20	41	3.8
64	128	0.55	1	7.3	28	2.0

https://en.wikichip.org/wiki/nvidia/microarchitectures/nvdla

TSMC 16 nm process at 1 GHz

NVDLA

- Large configuration

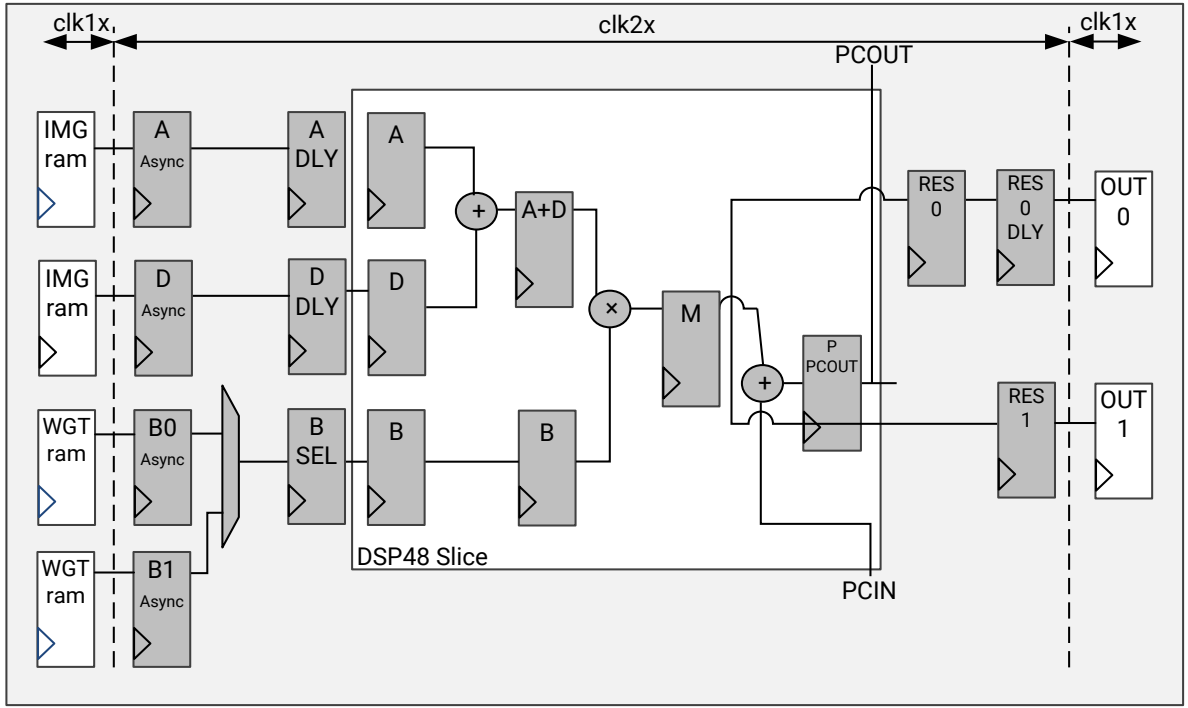
Configuration		Data Type	Internal RAM Size	ResNet50		
INT16/FP16	512 MACs			Perf (frames/s)	Power (mW)	Power Eff. (DL TOPS/W)
INT8	1024 MACs					
Conv Buffer	256 KB	INT8	none	165	267	4.8
Area	2.4 mm2	FP16	none	59	276	1.6
DRAM BW	15 GB/s	INT8	2M	230	348	5.1
TCM R/W BW	25/25 GB/s	FP16	2M	115	475	1.9

https://en.wikichip.org/wiki/nvidia/microarchitectures/nvdla

TSMC 16 nm process at 1 GHz

FPGA: DSP

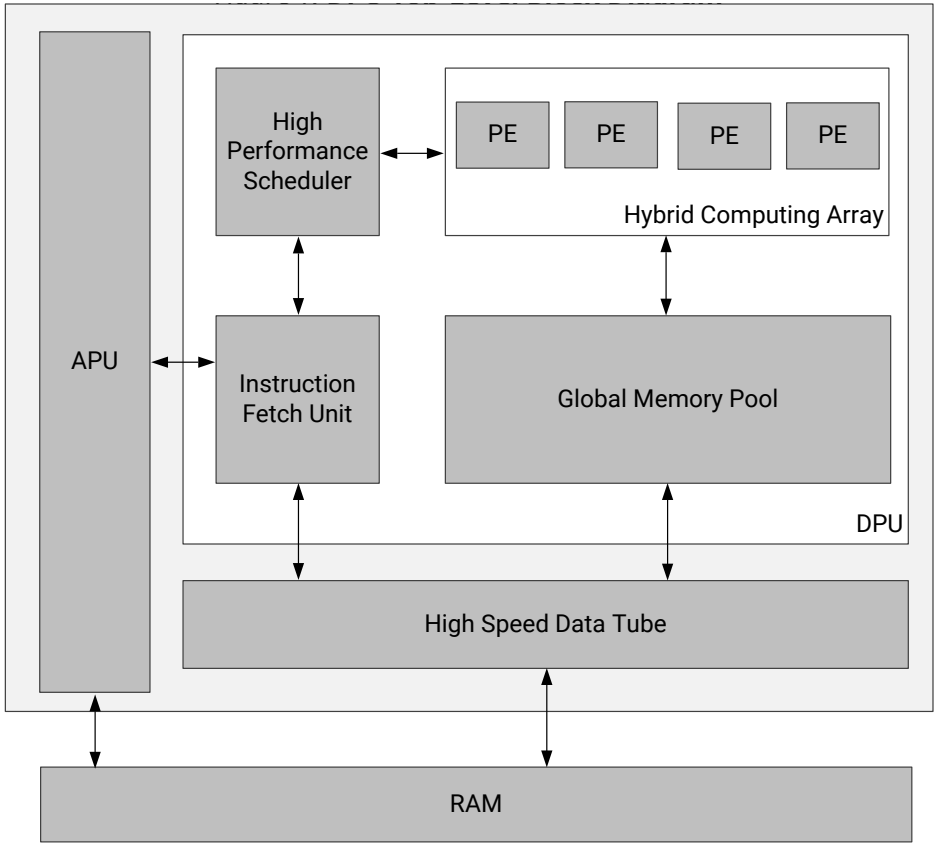
- DSP Double Data Rate (DDR) technique



Deep Learning Processing Unit (DPU)

XILINX

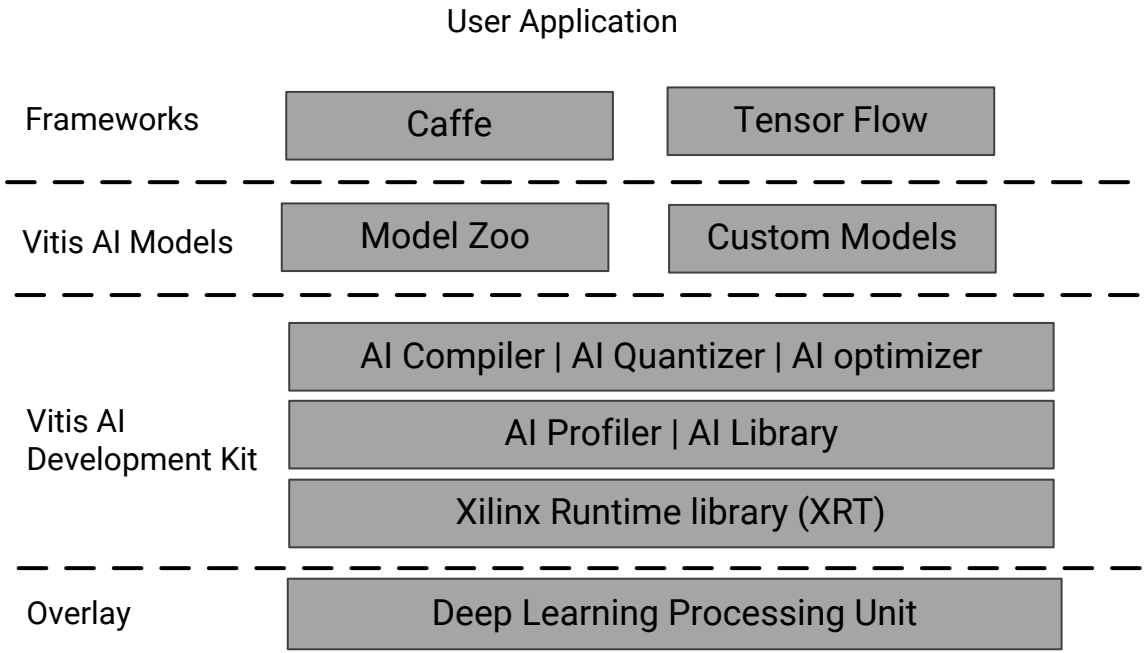
- DPU: programmable engine optimized for DNNs
- Includes
  - high-performance scheduler
  - hybrid computing array module
  - instruction fetch unit module
  - global memory pool module
- Uses a specialized instruction set
- Sequential architecture model



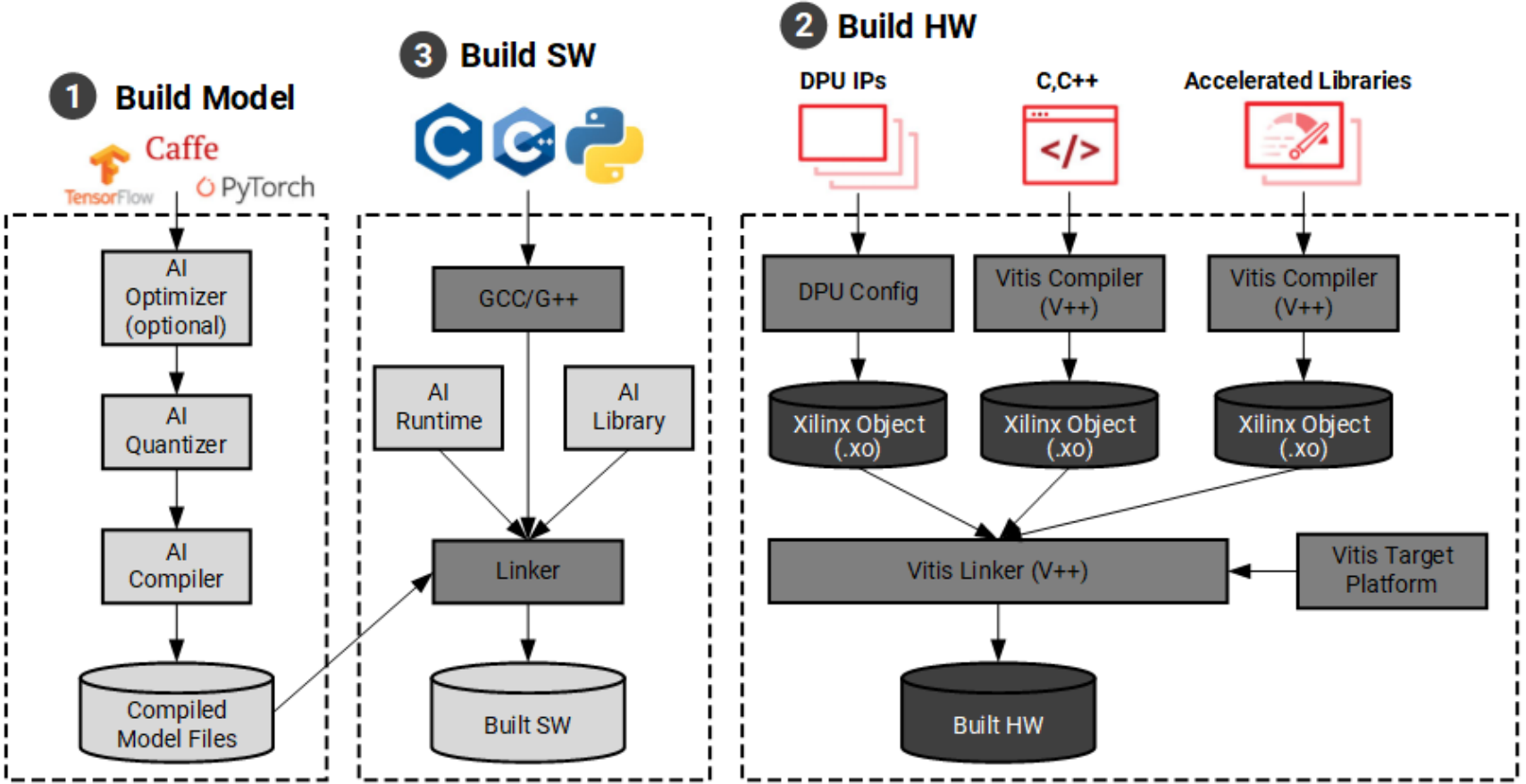
APU - Application Processing Unit

Vitis AI Stack

- Vitis AI
  - AI Compiler
  - AI Quantizer
  - AI Optimizer
  - AI Profiler
  - AI Library
  - Xilinx Runtime Library (XRT)

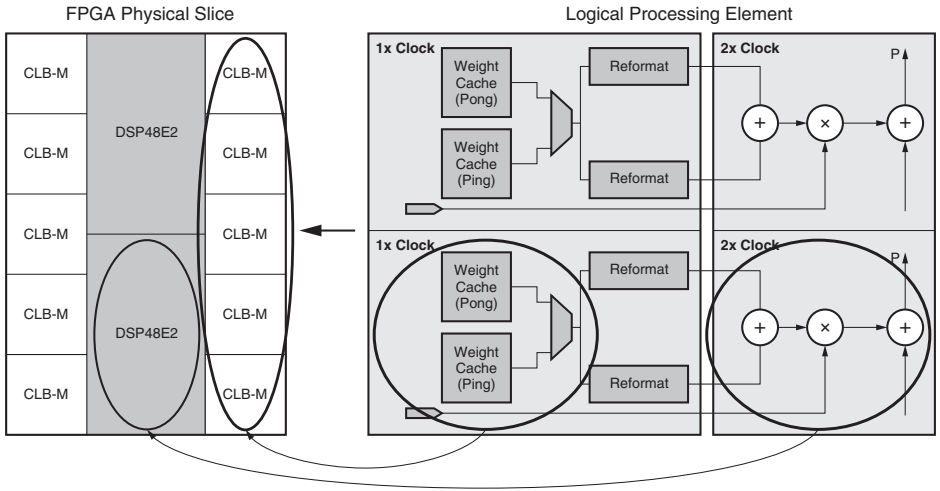


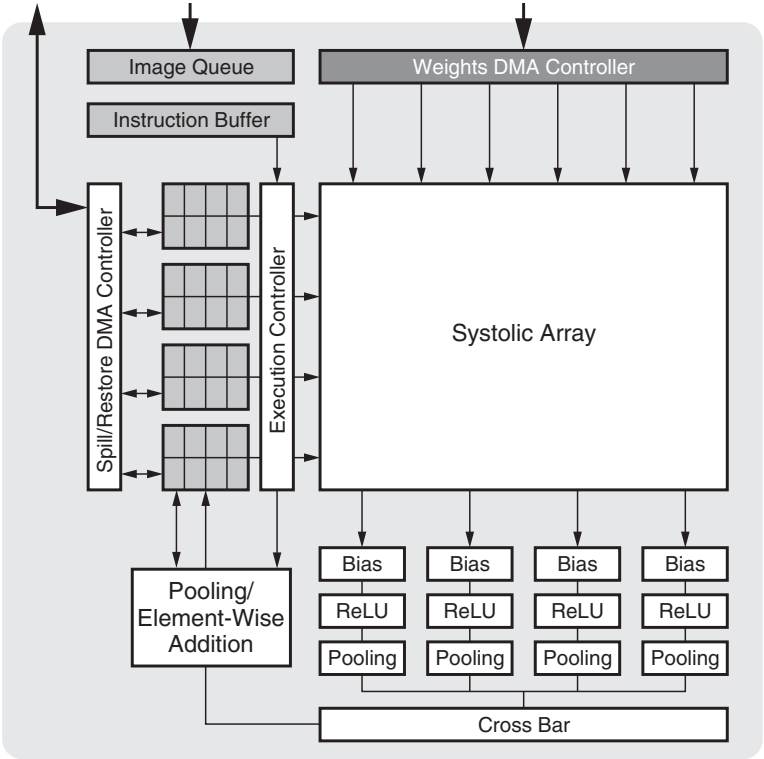
Vitis AI Stack



xDNN processing engine

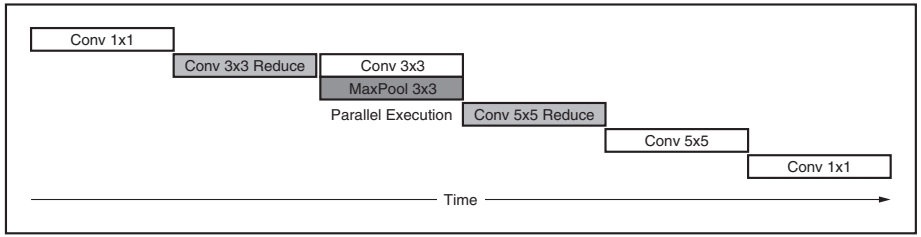
- instruction memory
- execution controller
- element-wise processing units
- systolic array

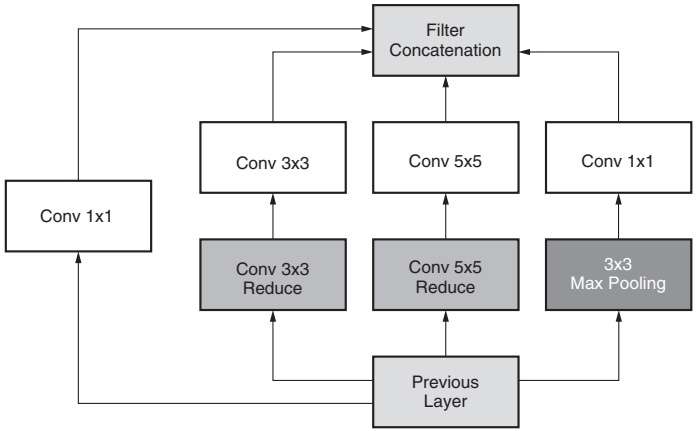




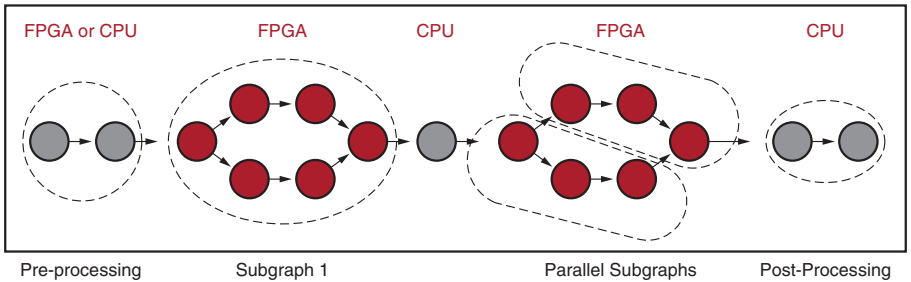
xDNN processing engine

- Command-Level Parallel Execution
  - Scheduling



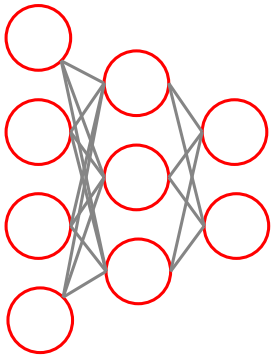


- Hardware-Assisted Image Tiling
- Custom Network Support through Heterogeneous Execution



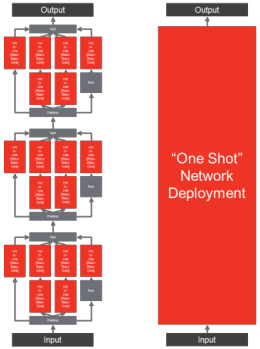
xfDNN Inference Toolbox

Graph Compiler



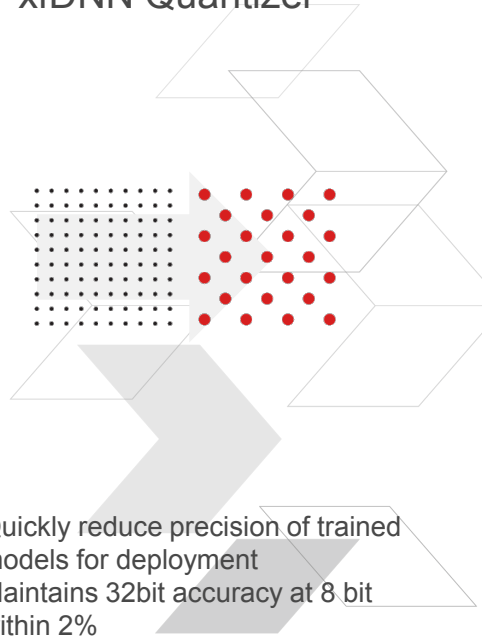
- Python tools to quickly compile networks from common Frameworks – Caffe, MxNet and Tensorflow

Network Optimization



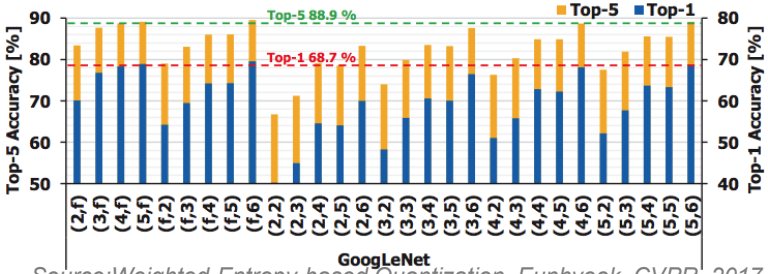
- Automatic network optimizations for lower latency by fusing layers and buffering on-chip memory

xfDNN Quantizer



- Quickly reduce precision of trained models for deployment
- Maintains 32bit accuracy at 8 bit within 2%

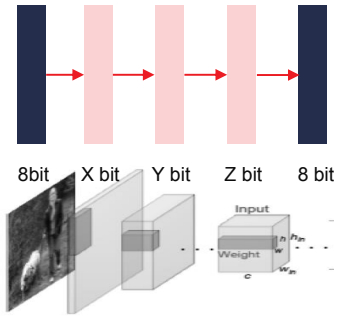
Mixed Low Precision

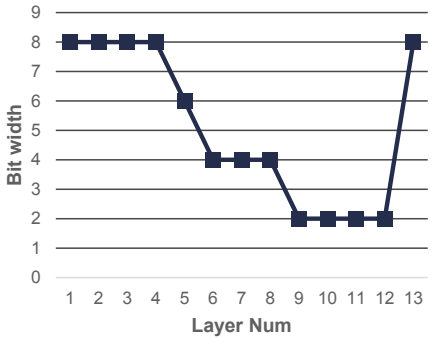


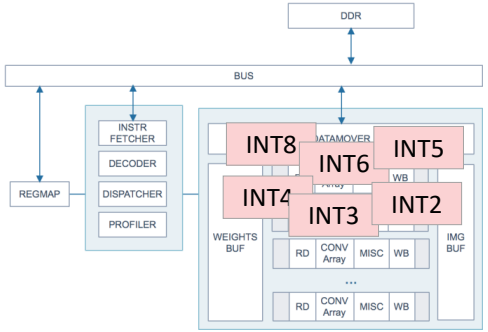
Source:Weighted-Entropy-based Quantization, Eunhyeok, CVPR, 2017-

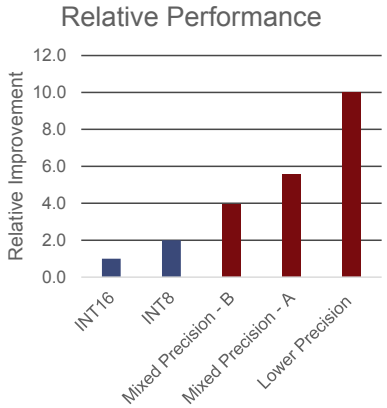
Fixed low-precision quantization already showed competitive results.

Next generation: **Variable** precision of activation/weights among layers









*accuracy drop less than 1%

BW	2	3	4	5	6	7	8
wgt	0	3	4	6	0	0	3
act	0	0	0	2	5	10	5

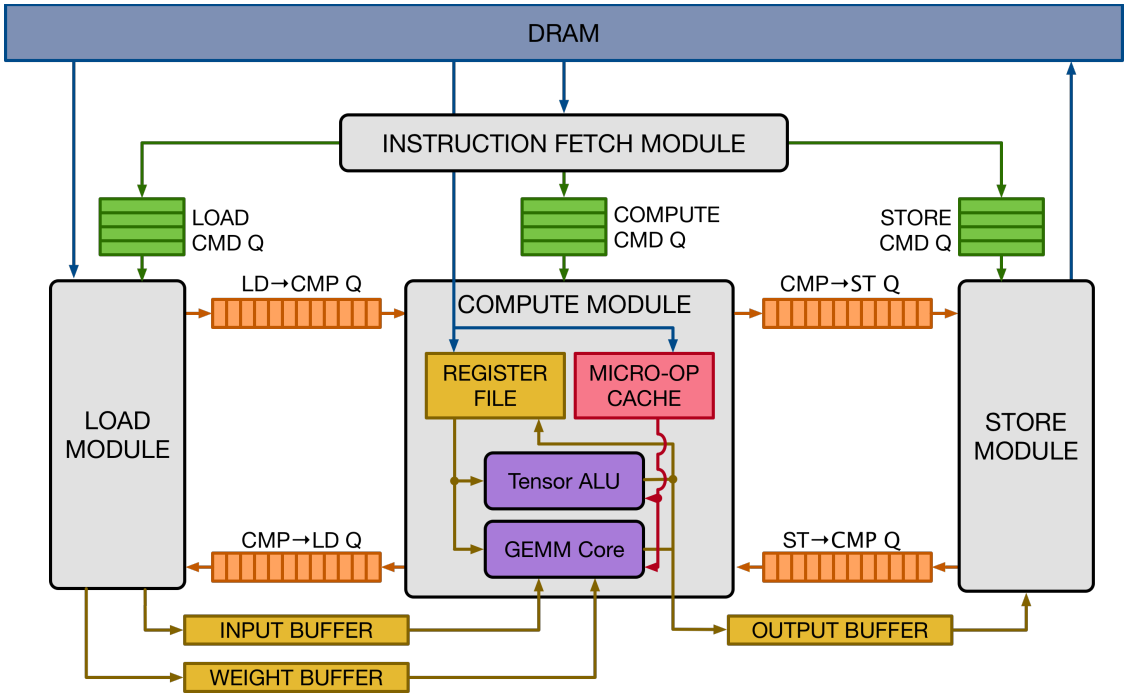
BW	2	3	4	5	6	7	8
wgt	0	0	3	22	17	10	2
act	0	0	0	16	41	13	3

BW	2	3	4	5	6	7	8
wgt	0	0	0	15	84	38	13
act	0	0	0	0	6	84	99

Preliminary experiments on popular networks. (vgg-16,resNet-50,inception-v4)

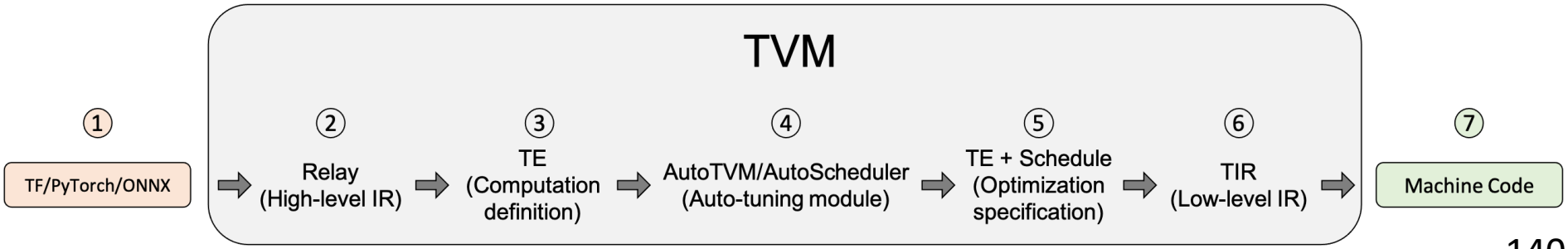
VTA: Versatile Tensor Accelerator

- VTA: an open, generic, and customizable DL accelerator
  - Sequential architecture model
- Complete TVM-based compiler stack



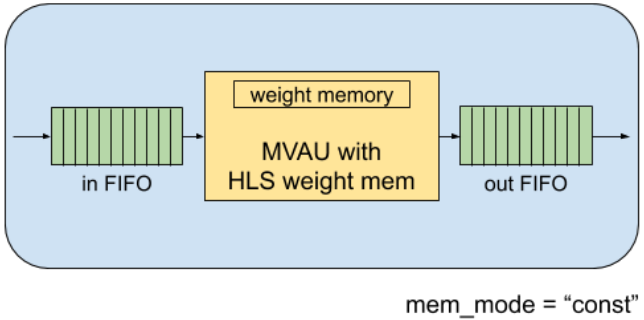
TVM: Open Deep Learning Compiler Stack

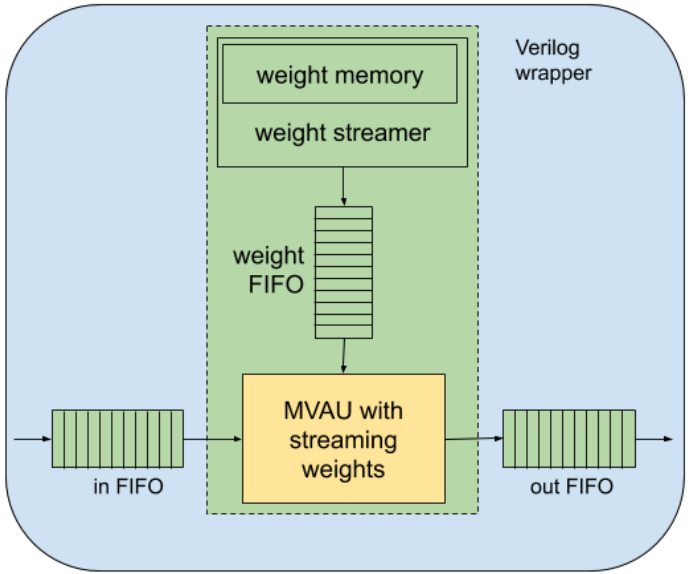
- Apache TVM is an open source machine learning (ML) compiler framework for CPUs, GPUs, and accelerators
  - optimize and run computations efficiently on any hardware backend



FINN Matrix Vector Activate Unit

- Two versions: block and streaming





- Computation structure
  - PEs based on SIMD units
  - #PE and #SIMD/PE are configurable

HLS components
Verilog components

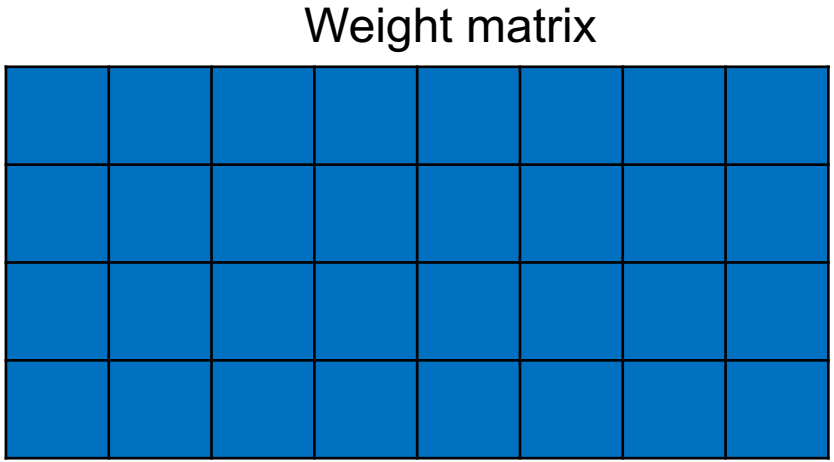
FINN Matrix Vector Activate Unit

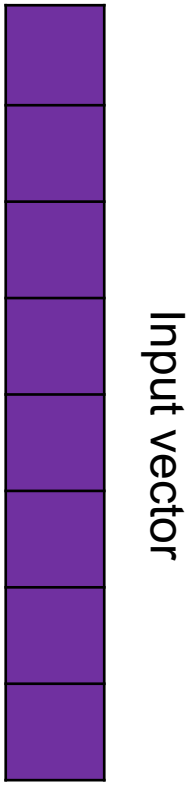
PE = 4
SIMD = 4

2 clock cycles

PE = 4
SIMD = 2

4 clock cycles



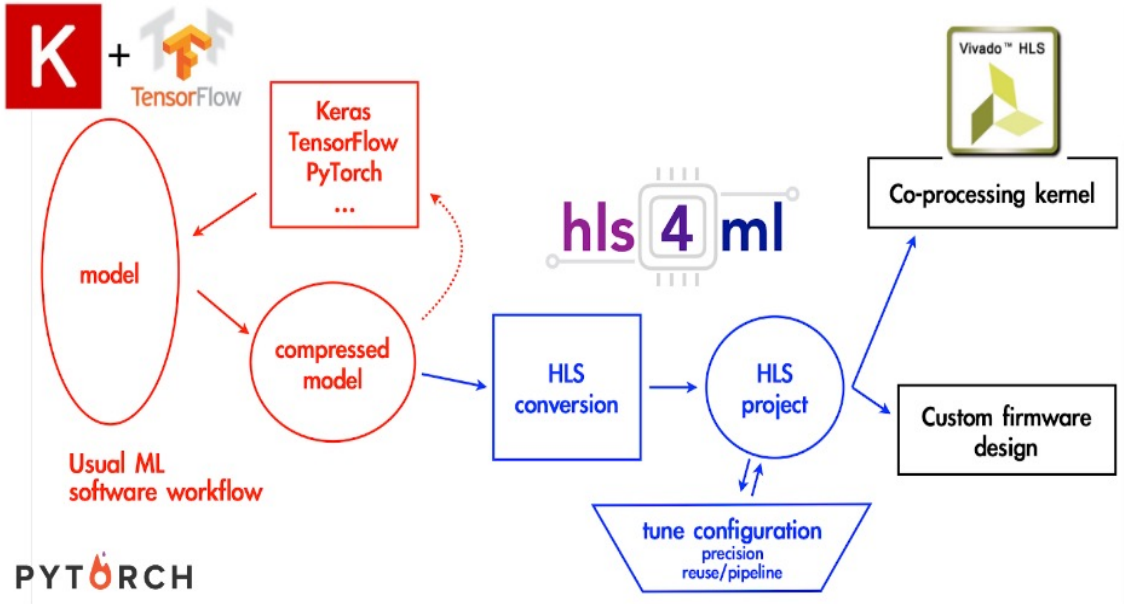


HLS sources: MVAU, MAC

HLS4ML

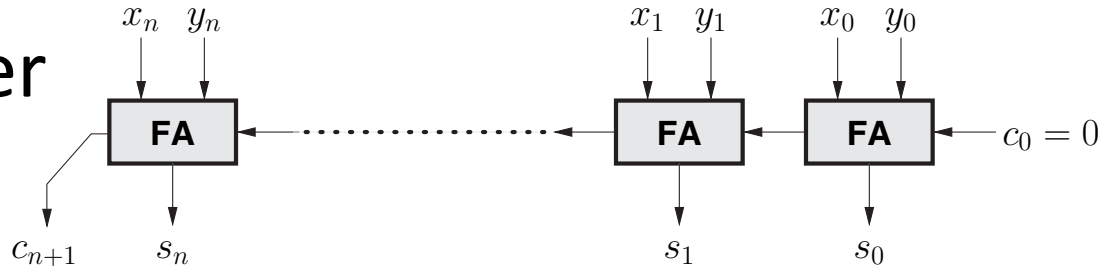
- **hls4ml** converts DNN models into a full HLS project
- Supports many kinds of models (MLP, CNN, RNN) and different formats (Keras/Tensorflow, Pytorch, ONNX)

– *Lacks good optimization (resource usage), models needs to be quantized aggressively*
– *Limited board, FPGA, recent HLS compiler support*



Integer Addition

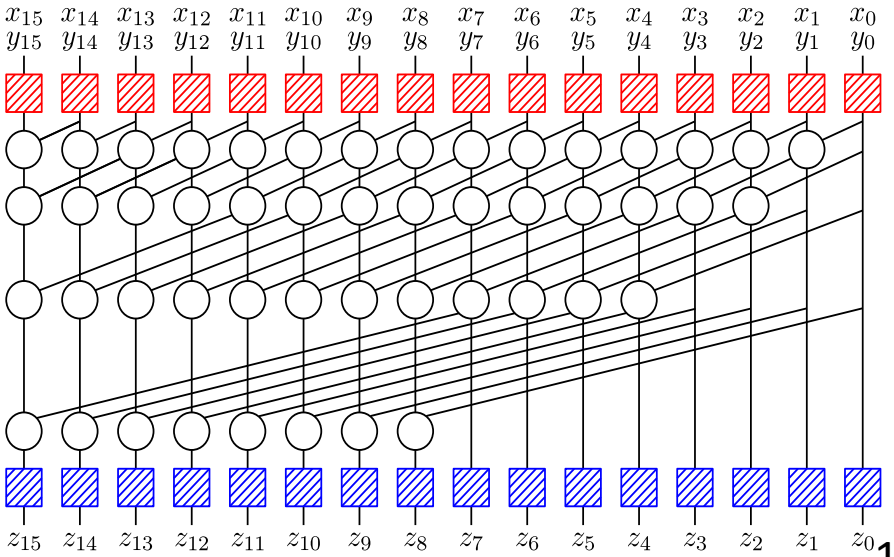
- Ripple-Carry Adder



delay: $\mathcal{O}(N)$

cost: $\mathcal{O}(N)$

- Parallel prefix adders



delay: $\mathcal{O}(\log N)$

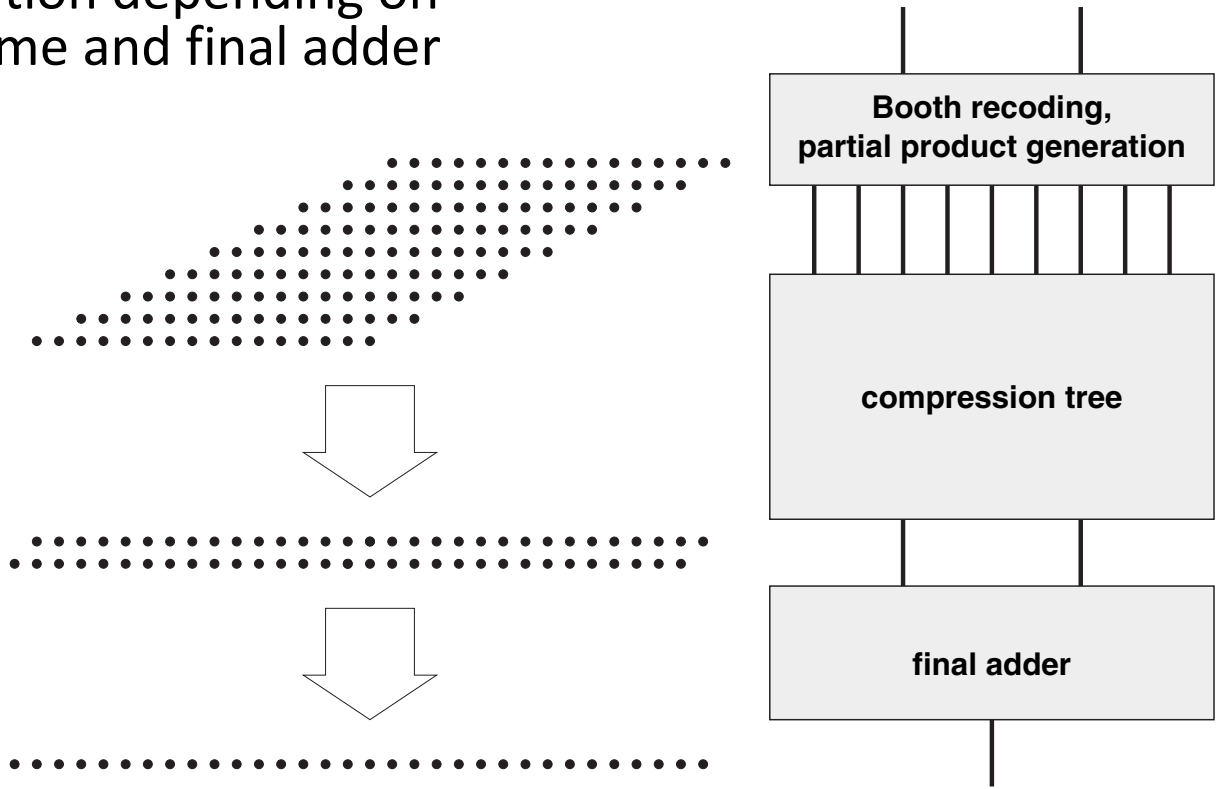
cost: $\mathcal{O}(N \log N)$

Integer Multiplication

- Many implementation depending on compression scheme and final adder

delay: $\mathcal{O}(\log N)$

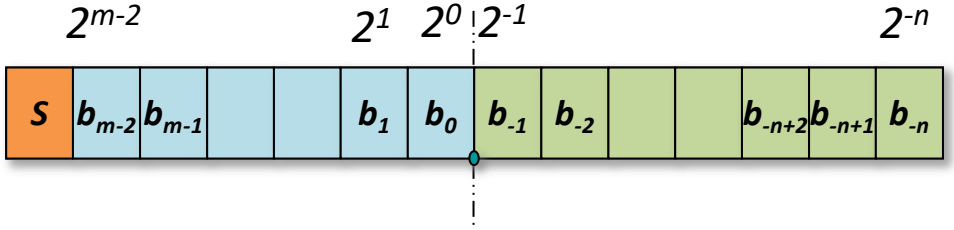
cost: $\mathcal{O}(N^2)$



J.-M. Muller et al., Handbook of floating-point arithmetic, Springer, 2009.

Fixed-Point Representation

- **Representation**
  (w,m,n)
  w=m+n



- **Examples**

  $97 = 01100001_2$ in (8,2,6)

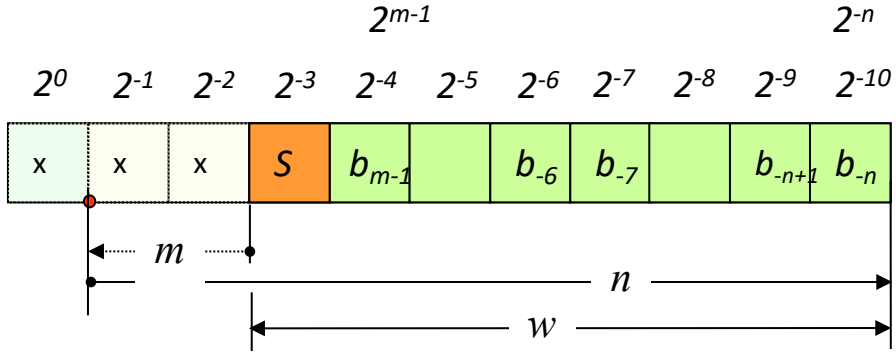
  $1+1/2+1/64 = 1.515625$

  $97 = 01100001_2$ in (8,-2,10)

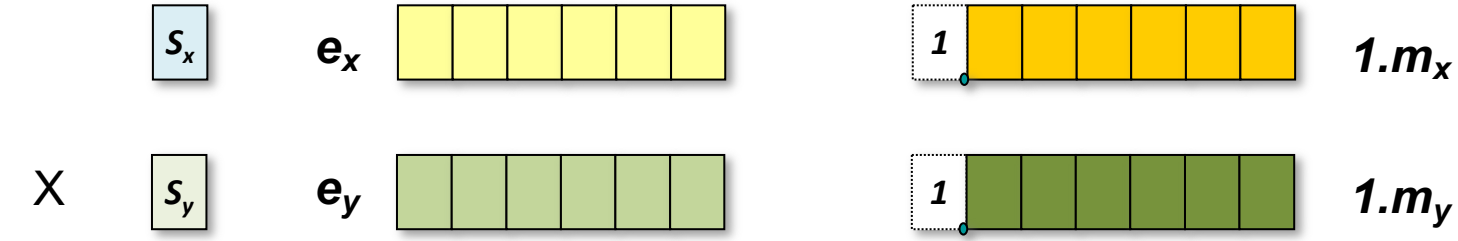
  $2^{-4}+2^{-5}+2^{-10}$

  0.0947265625

Arithmetic Rules (scaling)

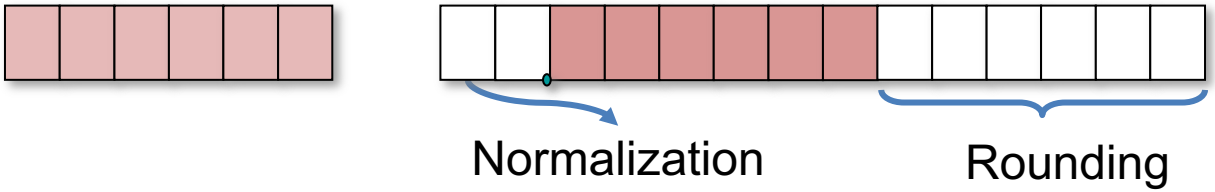


Floating-Point Multiplication



$e_z = e_x + e_y$

$1.m_z$



Normalization

Rounding

$(1.m_x \times 1.m_y) \times 2^{e_x+e_y}$

(shift m_z, add 1 to e_z)

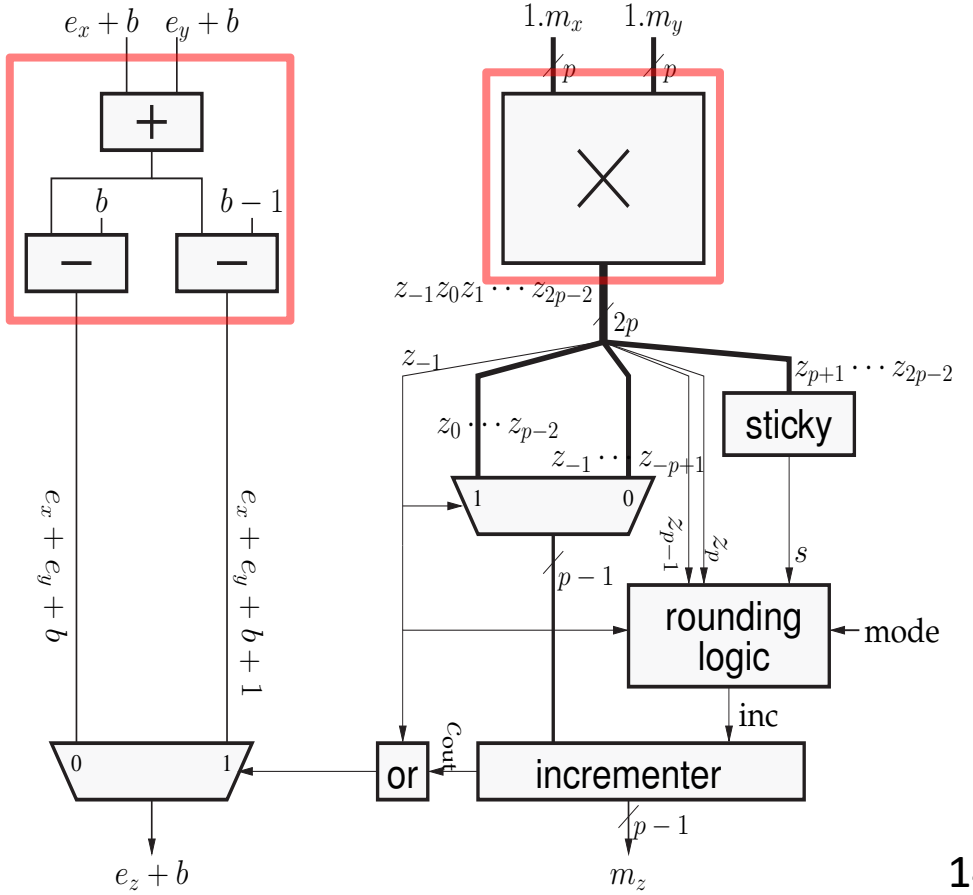
$1 \le 1.m_x, 1.m_y < 2$

$1 \le 1.m_x \times 1.m_y < 4$

Floating-Point Multiplication

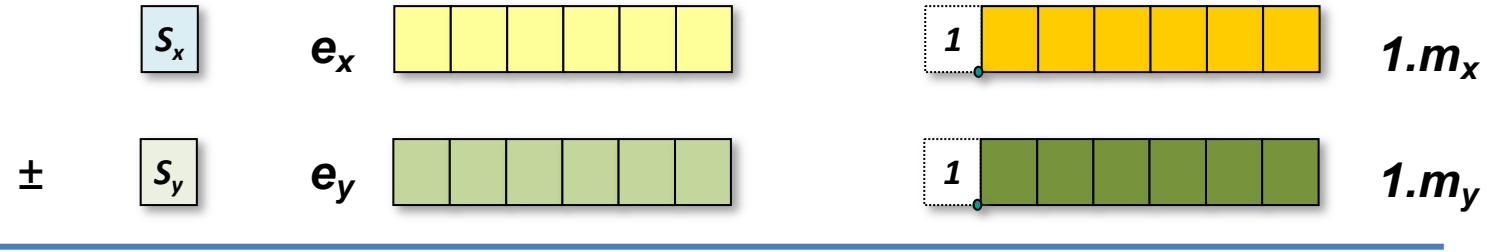
- Representation *(W,E,M)*
  - Exponent *e* on *E* bits
  - Mantissa *m on M* bits

- Floating-point hardware is doing the job for you!
  - FlP operators are therefore more complex than FxP



[J.-M. Muller et al., Handbook of Floating-point arithmetic, 2009]

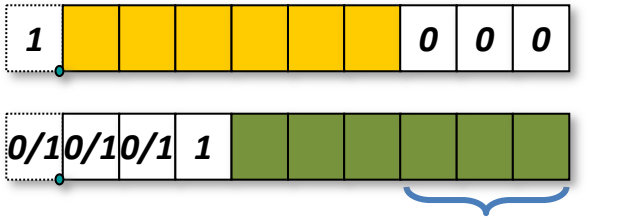
Floating-Point Addition



- *swap to have $e_y \leq e_x$*
- *determine if effective subtraction*
- *calculate $e_x - e_y$*
- *shift mantissa m_y accordingly*

$e_x - e_y \geq 2$

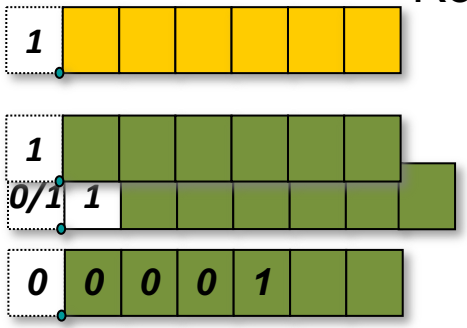
(far path)



- *cancellation may occur only if $e_x - e_y < 2$*
- *normalization (LZC/shift)*
- *rounding*

$e_x - e_y < 2$

(close path)

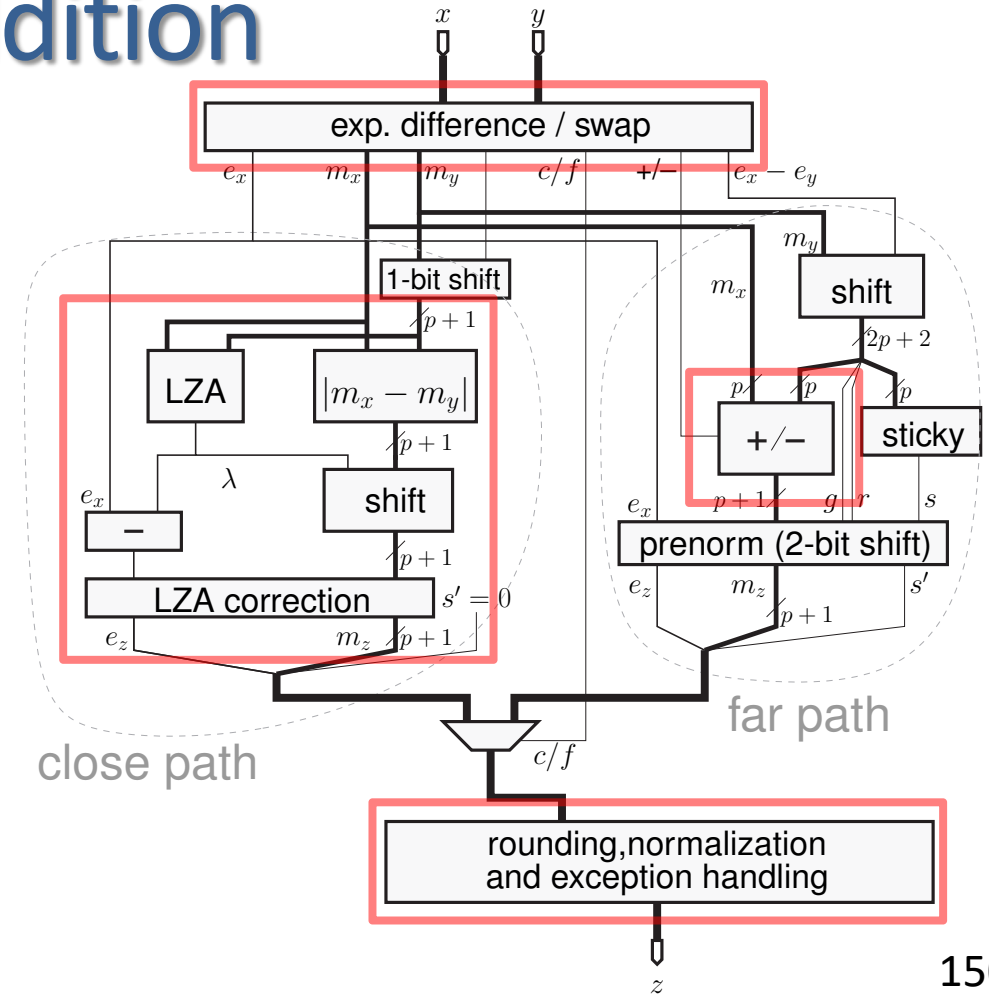


Floating-Point Addition

- Representation *(W,E,M)*
  - Exponent *e* on *E* bits
  - Mantissa *m on M* bits

$$s_z = s_x \text{ XOR } s_y$$

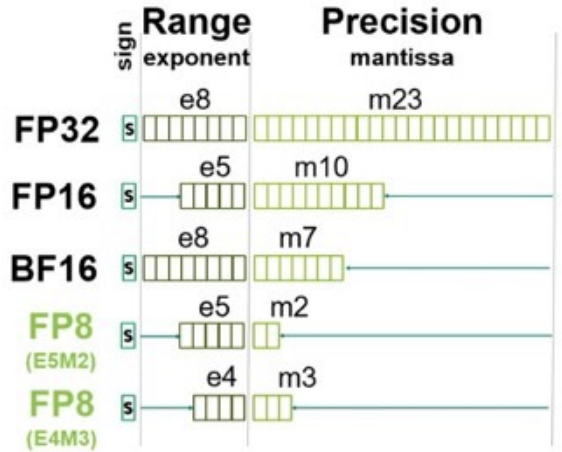
$$x + y = (-1)^{s_x} \cdot \left(|x| + (-1)^{s_z} \cdot |y| \right)$$



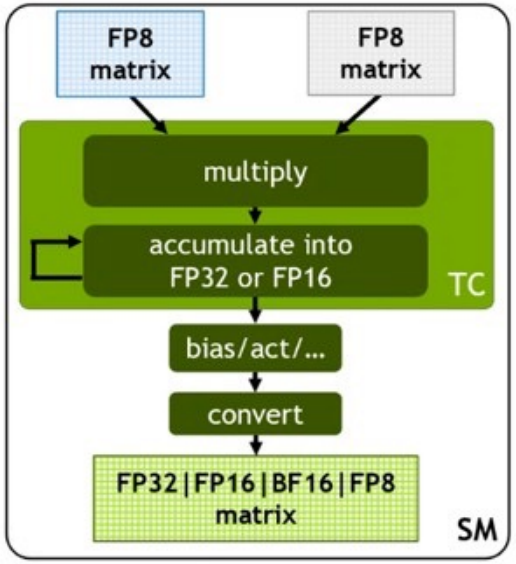
[J.-M. Muller et al., Handbook of Floating-point arithmetic, 2009]

Arithmetic Support in Latest Chips?

- Hopper GH100 GPU from Nvidia
  - FP8 support in tensor cores provides up to 4x speedup



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types