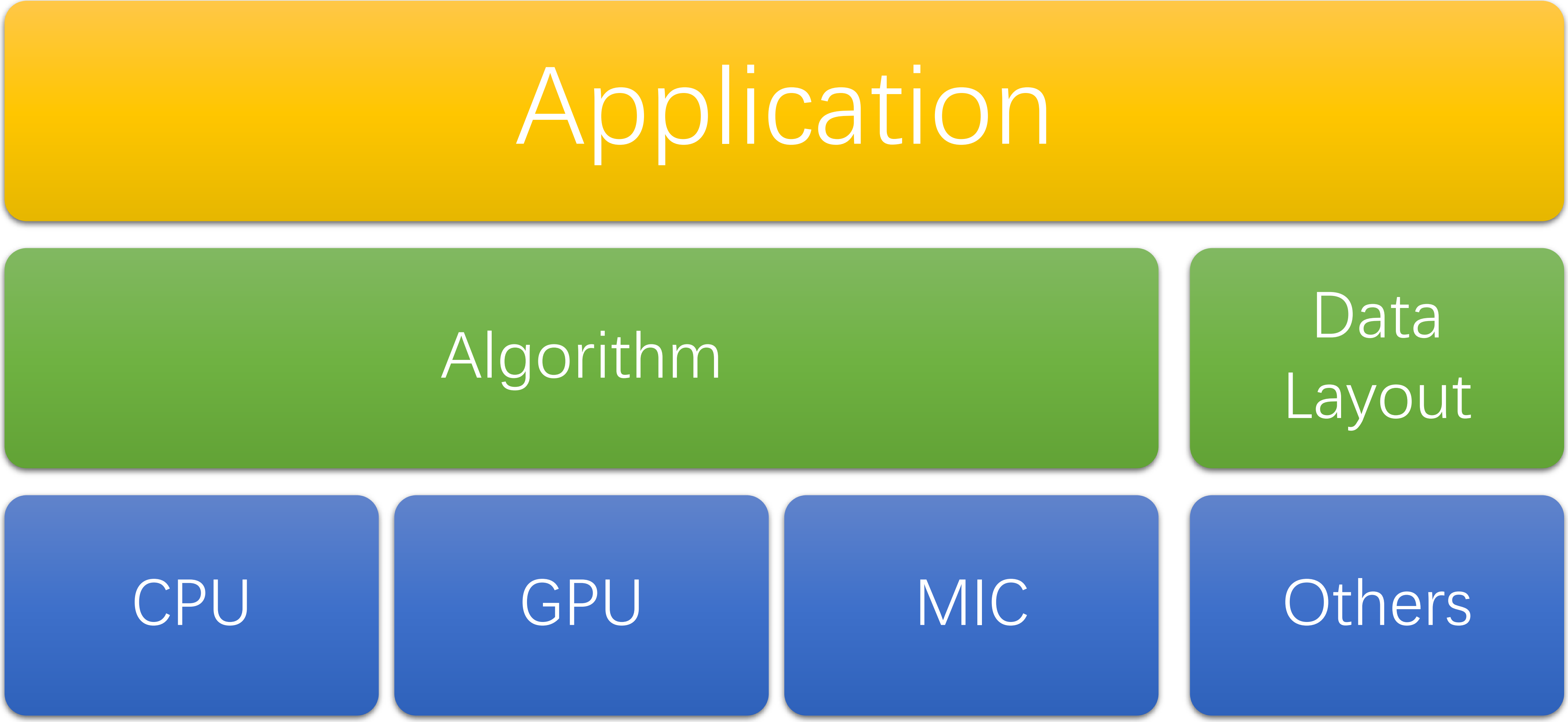
Deep Learning on Modern Architectures

Keren Zhou

4/17/2017

HPC Software Stack



Application

Algorithm

Data Layout

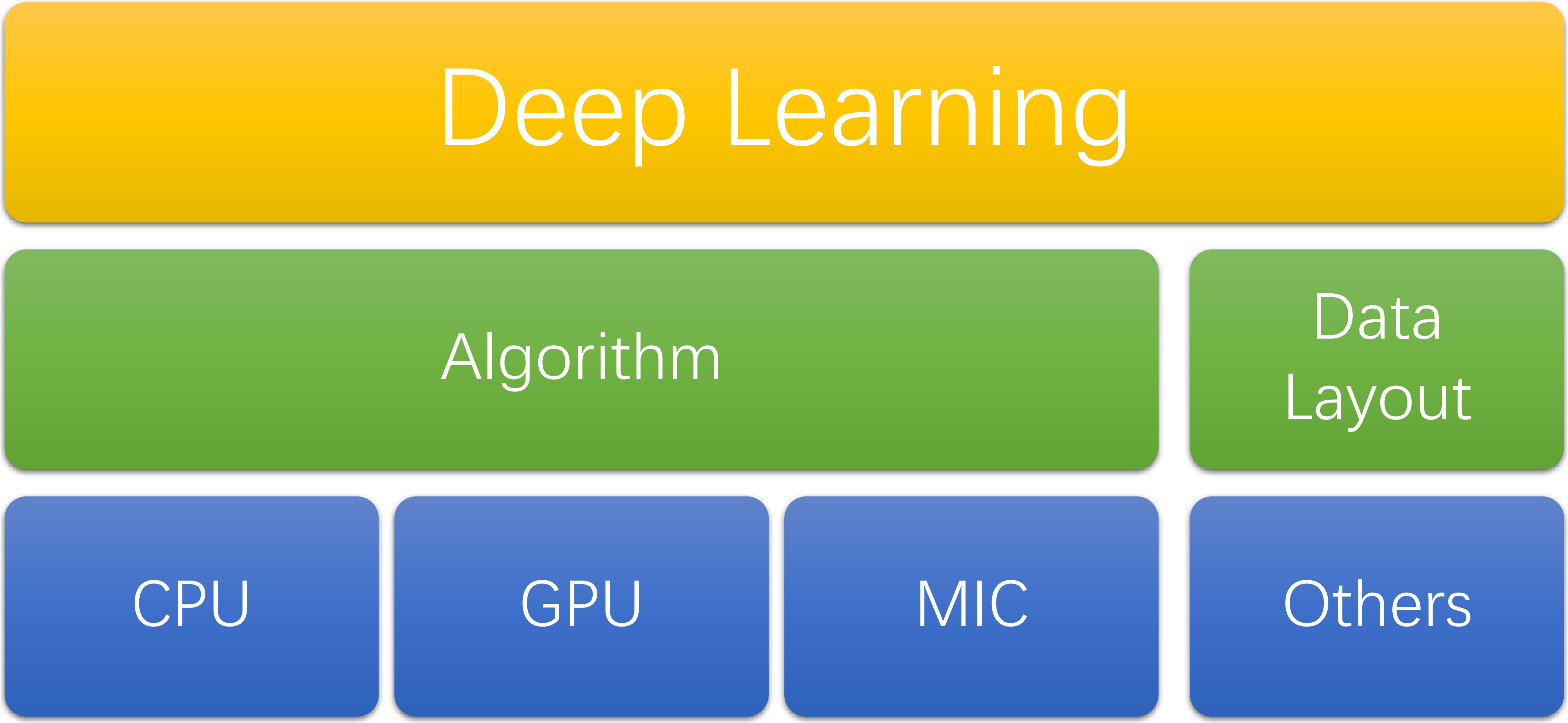
CPU

GPU

MIC

Others

HPC Software Stack



Deep Learning

Algorithm

Data Layout

CPU

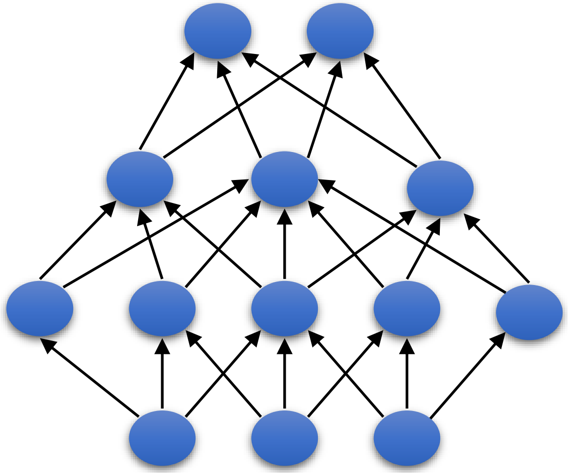
GPU

MIC

Others

Deep Learning

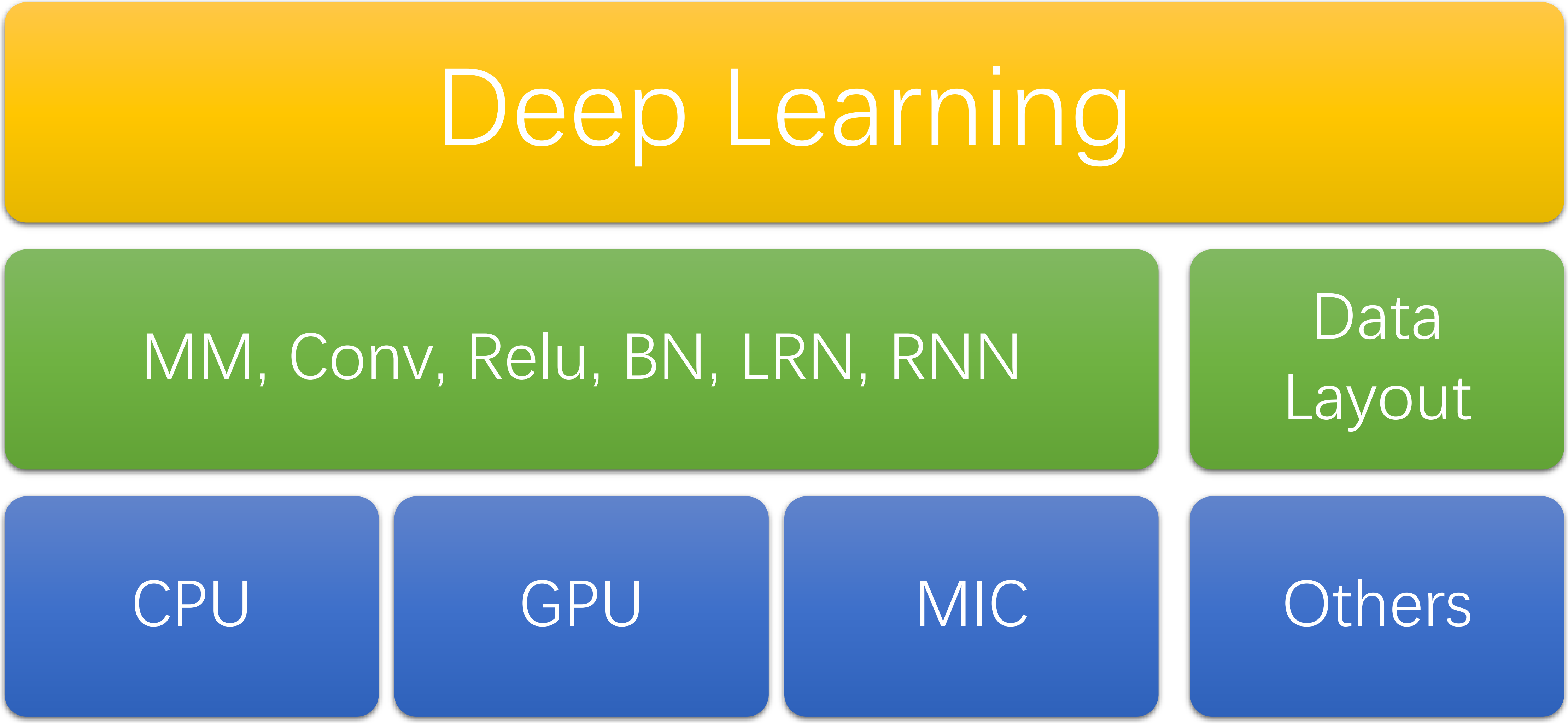
- Neural network with more than one non-linear hidden layers—*Hinton*
- *Three research areas:*
  - *Brain->model   (Neuro Science)*
  - *Model->parameters (Machine Learning)*
  - *Parameters->codes (HPC)*



```
Layers:
{
type: Conv
name: Conv1
stride: [str_h, str_w]
padding: 0
fshape: [K, C, R, S]
}
```

```
Kerenls:
GEMM(A, B, C, alpha, beta)
Convolution(I, F, O)
RNN(H, W, O)
```

HPC Software Stack



Deep Learning

MM, Conv, Relu, BN, LRN, RNN

Data Layout

CPU

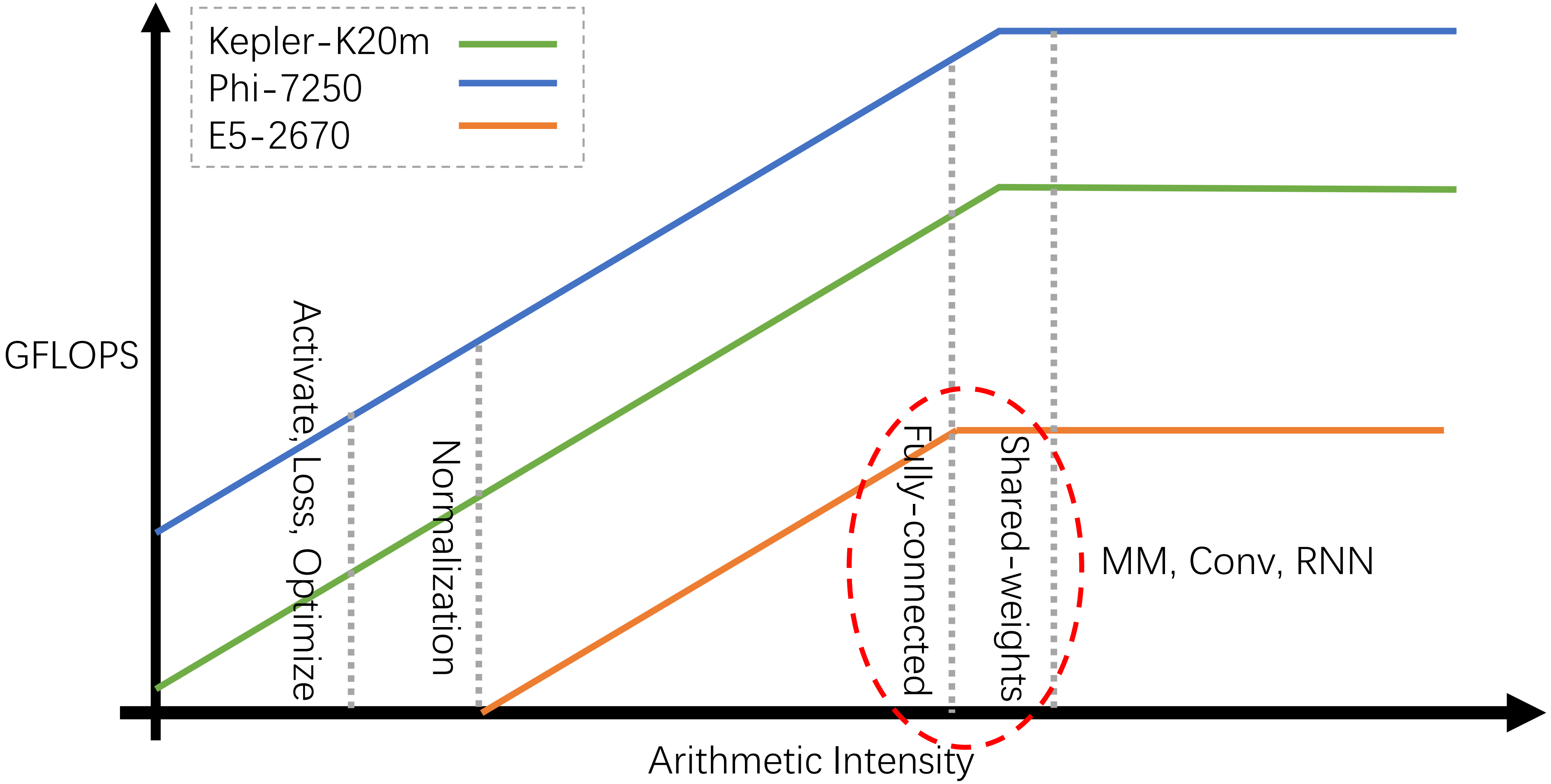
GPU

MIC

Others

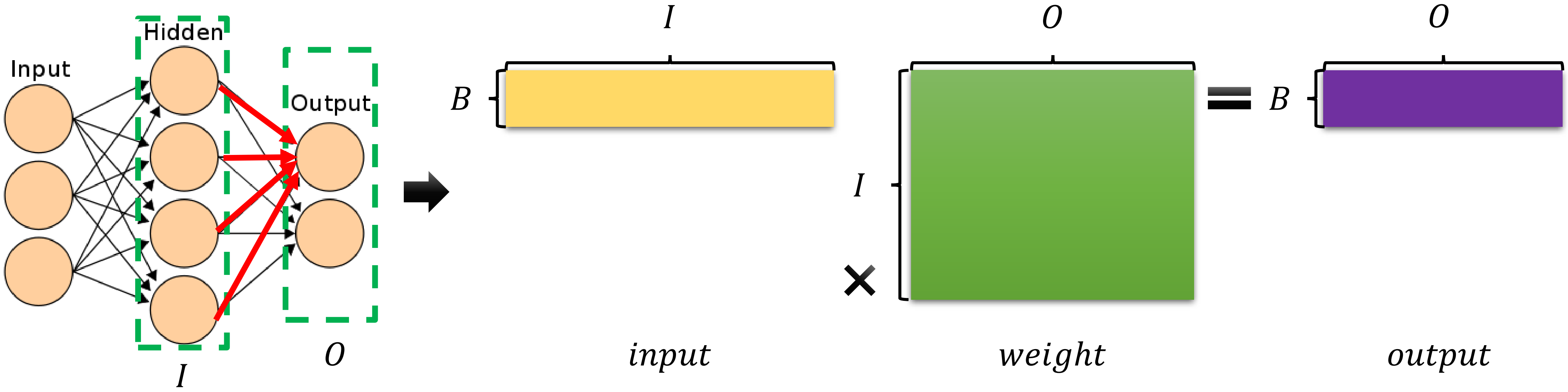
DNN Computations

- Arithmetic Intensity



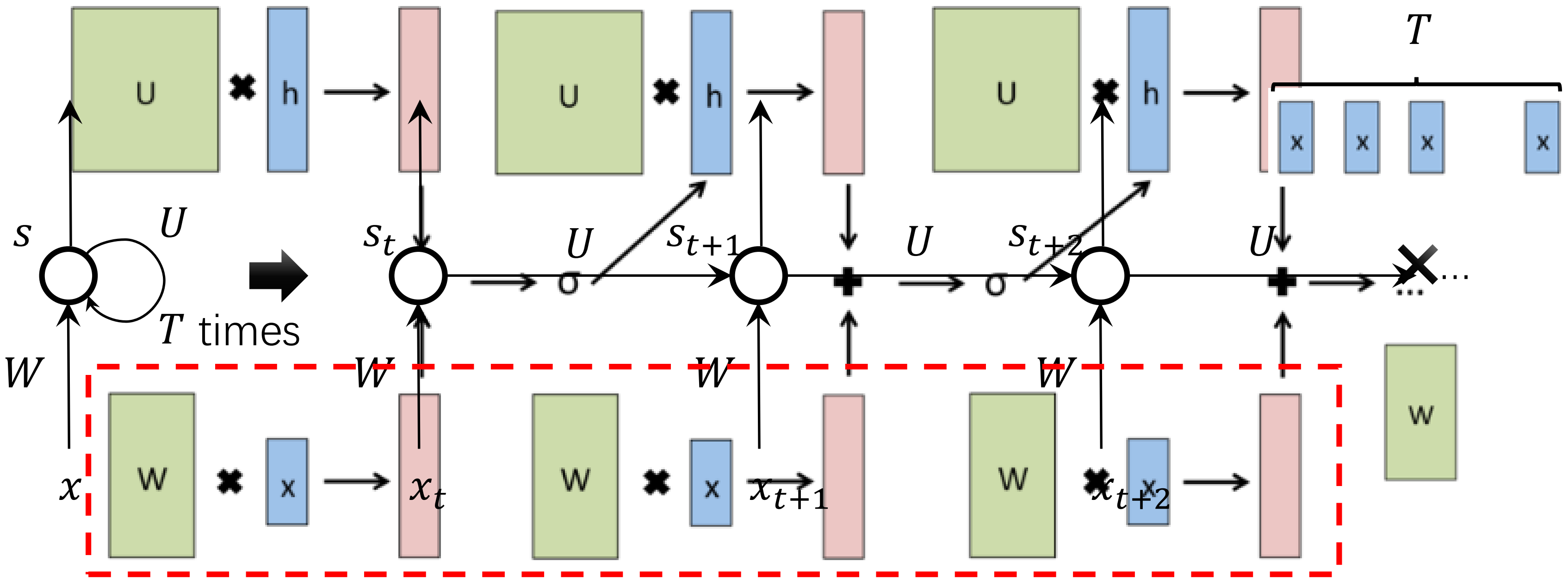
DNN Computations

- Matrix Multiplication is the core of Deep Learning
- Fully-connected layer
  - $output_j = \sum_{k=0} input_k \times weight_{kj}$
  - I: input size
  - O: output size
  - B: batch size



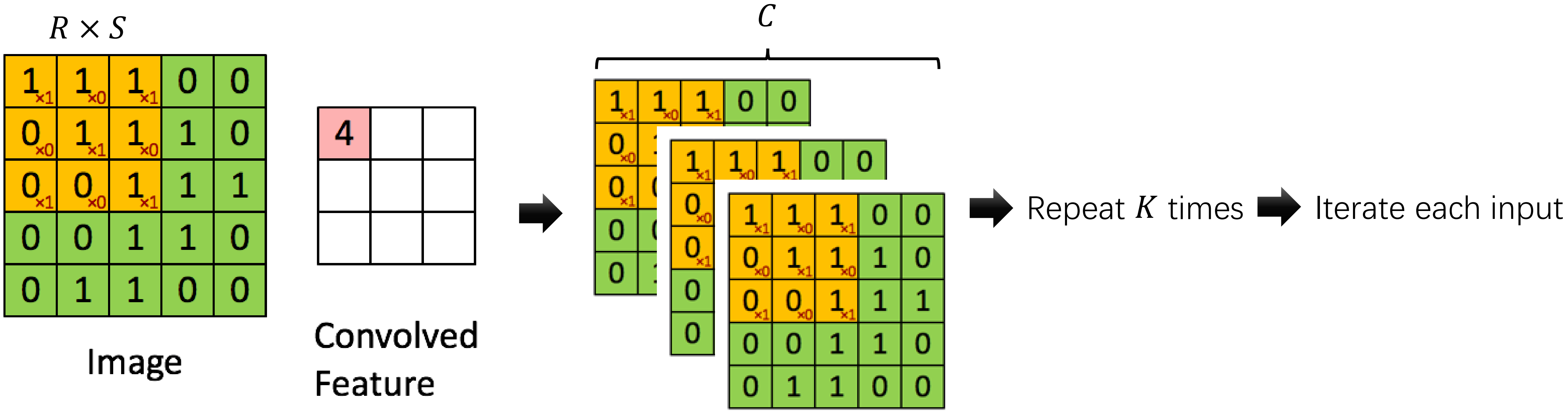
DNN Computations

- Recurrent Neural Network
  - $h^t = Wx^t + Uh^{t-1}$
  - x^t: time t's input
  - h^{t-1}: time t-1's hidden state



DNN Computations

- Convolution
  - $O_{bk} = \sum_{crs=0} I_{b[crs]} \times F_{[crs]k}$
  - R, S: filter height and width
  - C: input channels
  - K: output channels



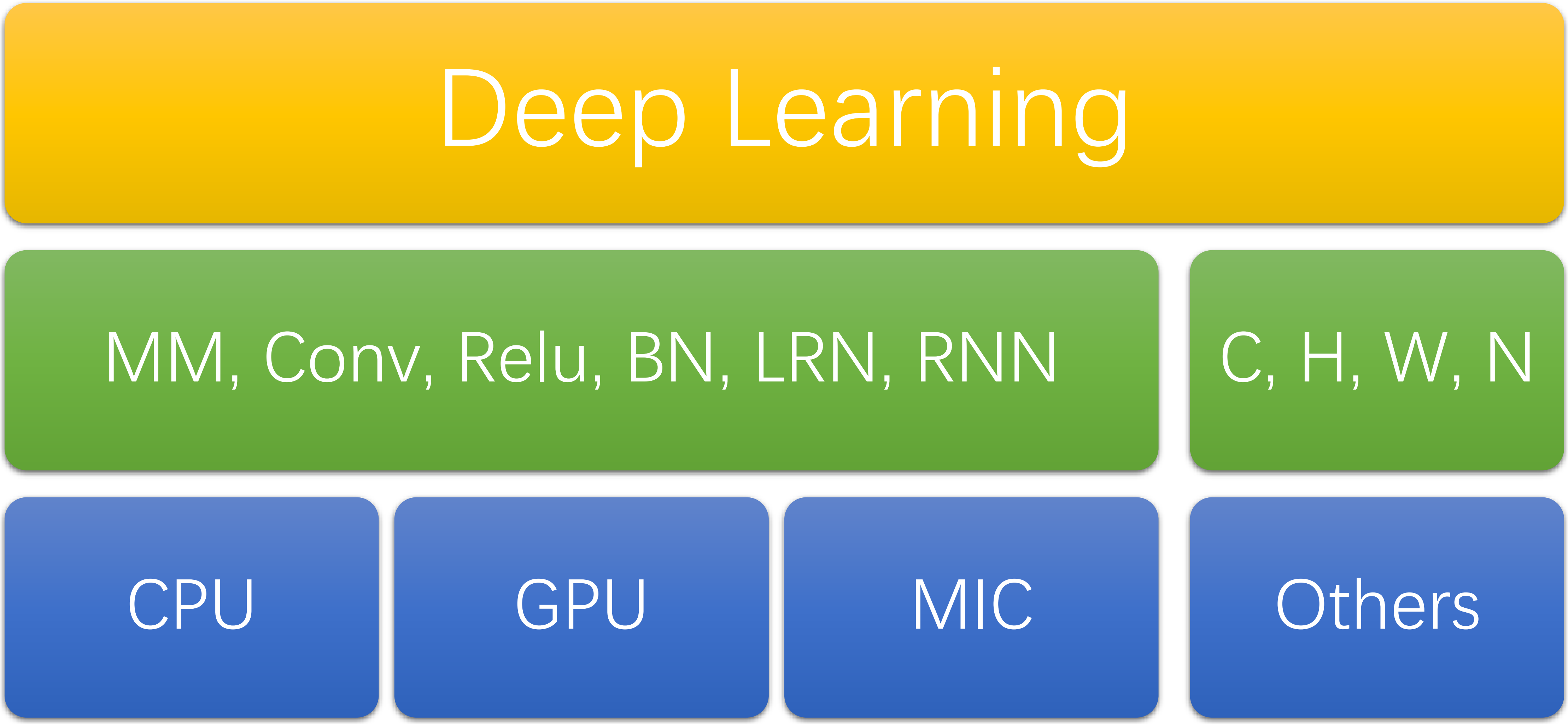
$1_{\times1}$	$1_{\times0}$	$1_{\times1}$	0	0
$0_{\times0}$	$1_{\times1}$	$1_{\times0}$	1	0
$0_{\times1}$	$0_{\times0}$	$1_{\times1}$	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

HPC Software Stack



Deep Learning

MM, Conv, Relu, BN, LRN, RNN

C, H, W, N

CPU

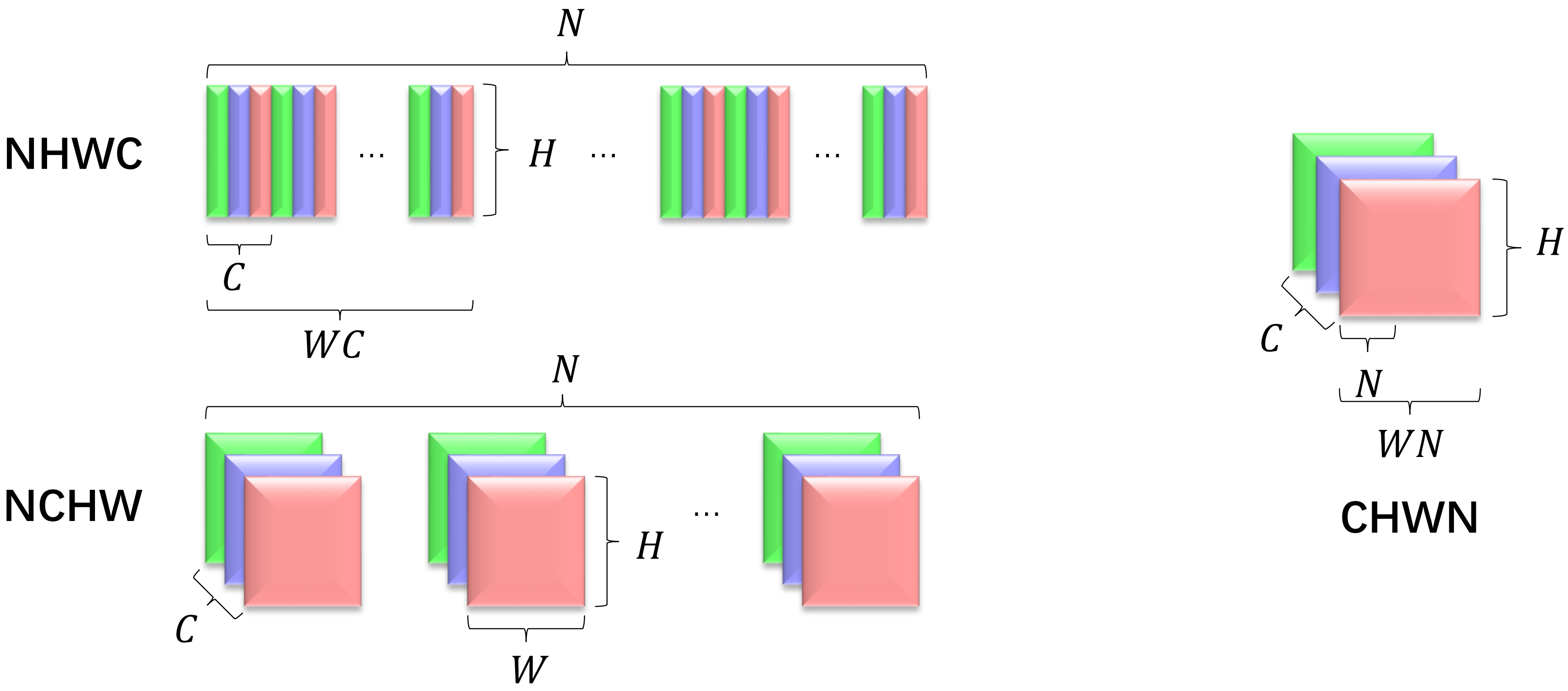
GPU

MIC

Others

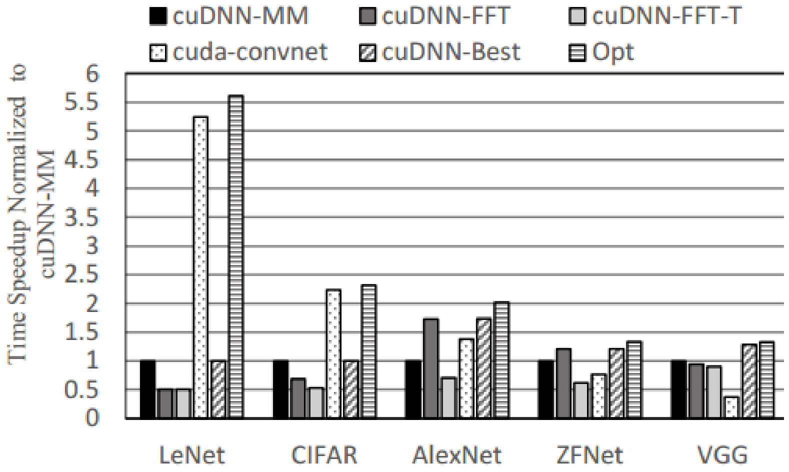
Data Layout

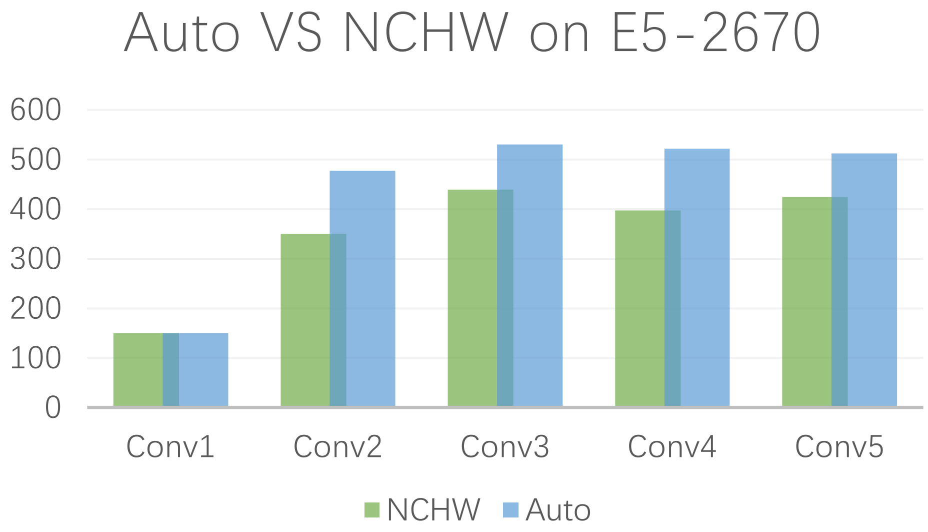
- N: Batch size, C: Channel size, H: Height, W: Width
  - Different data layout has different performance because of contiguous memory accesses
  - Single data layout cannot achieve the highest performance



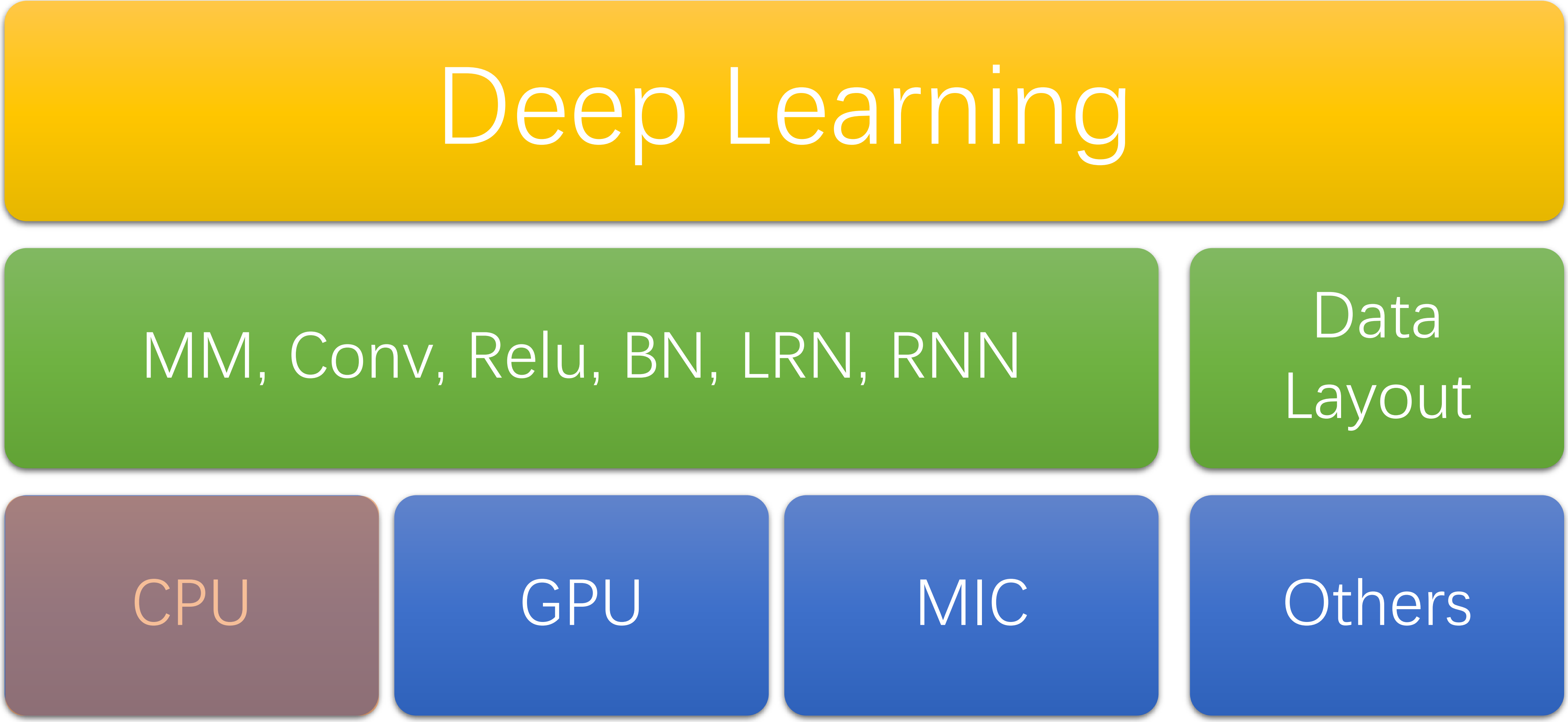
Data Layout

- Solutions
  - cuDNN
    - Different implementations for each data layout
  - MKL
    - Transformation before execution
  - Data transformation on GPU [*Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs*]
  - Blitz: search the best format on each layer





HPC Software Stack



Deep Learning

MM, Conv, Relu, BN, LRN, RNN

Data Layout

CPU

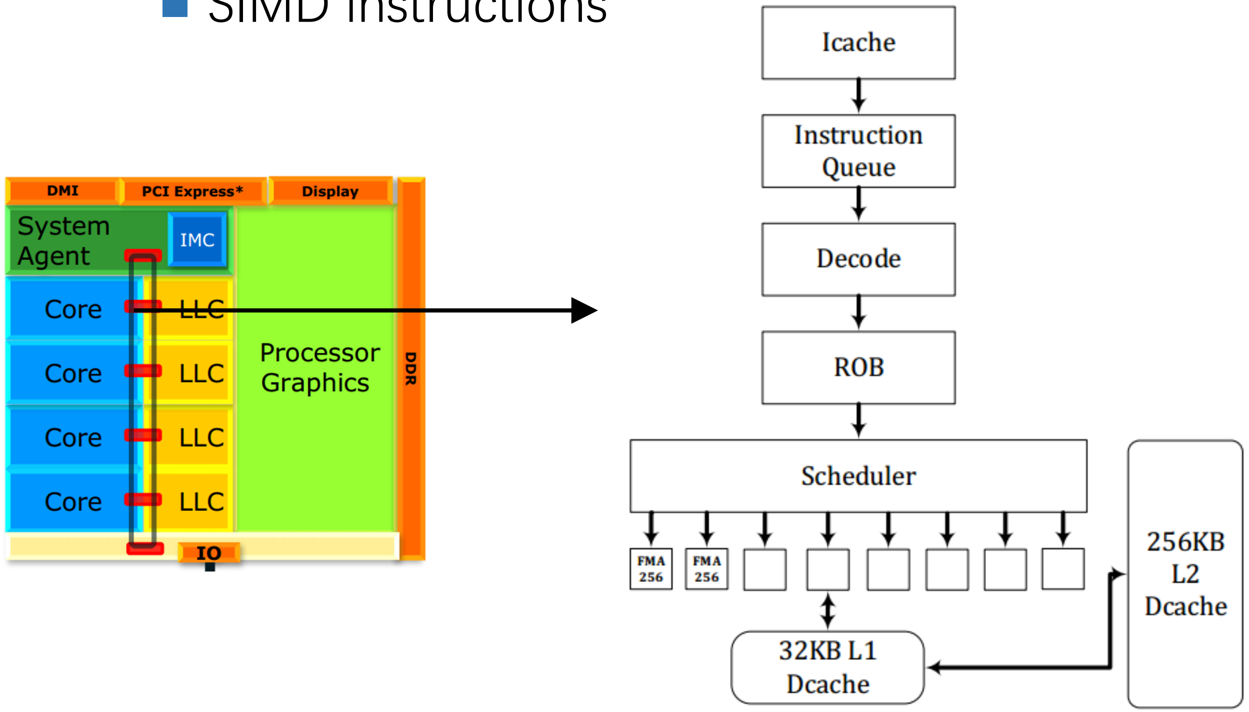
GPU

MIC

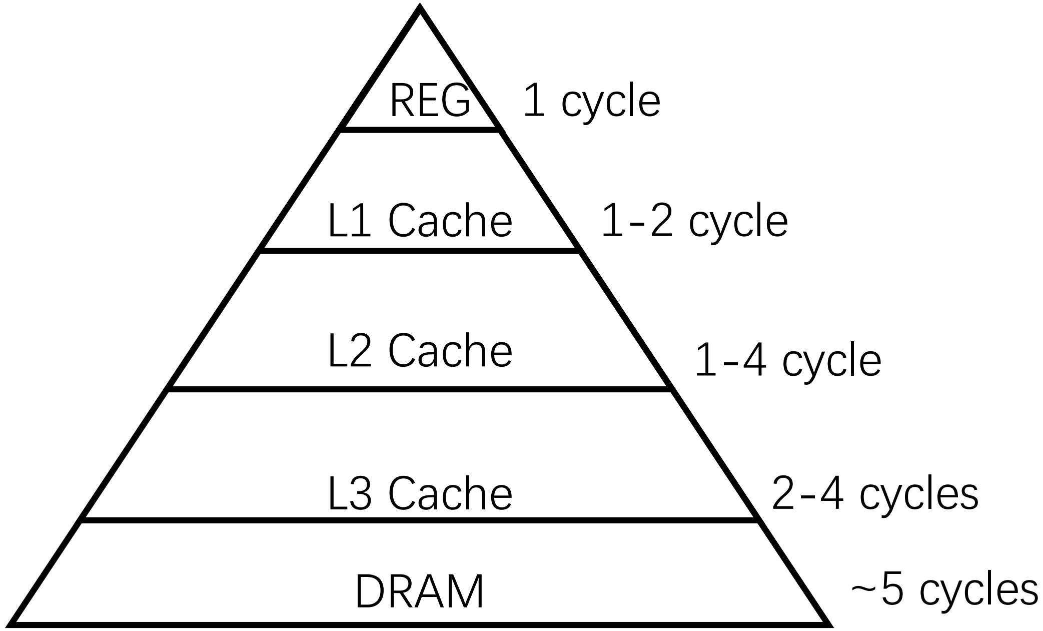
Others

CPU

- CPU Architecture
  - Cache Hierarchy
  - Multiple issue ports
  - SIMD instructions



Haswell



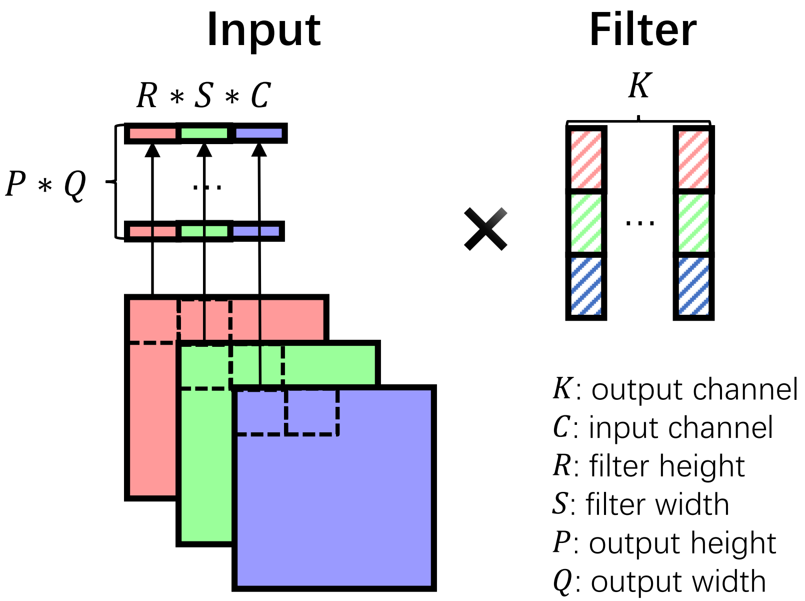
CPU

2014 ← **Caffe**

IM2COL+GEMM

Idea: Reduce my problem to one already solved

[https://github.com/Yangqing/caffe/wiki/Convolution-in-Caffe:-a-memo]



K: output channel
C: input channel
R: filter height
S: filter width
P: output height
Q: output width

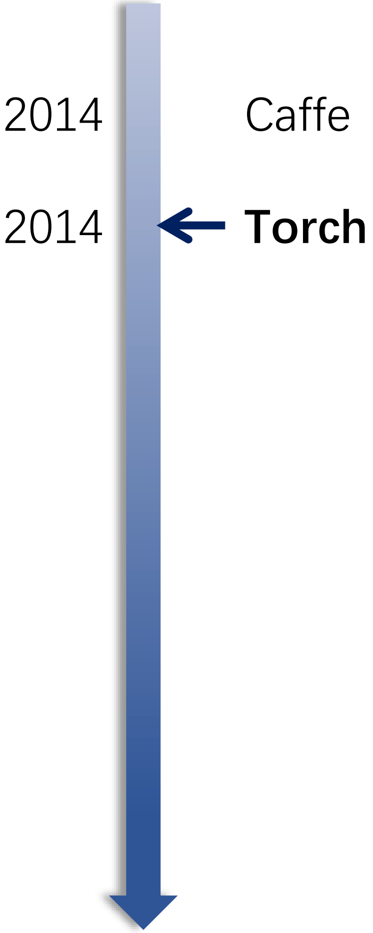
Pros:

- Easy to implement
- Cross-platform
- Relatively High-performance

Cons:

- Not extreme performance
- Inefficient on GPU
- Large memory consumption

CPU

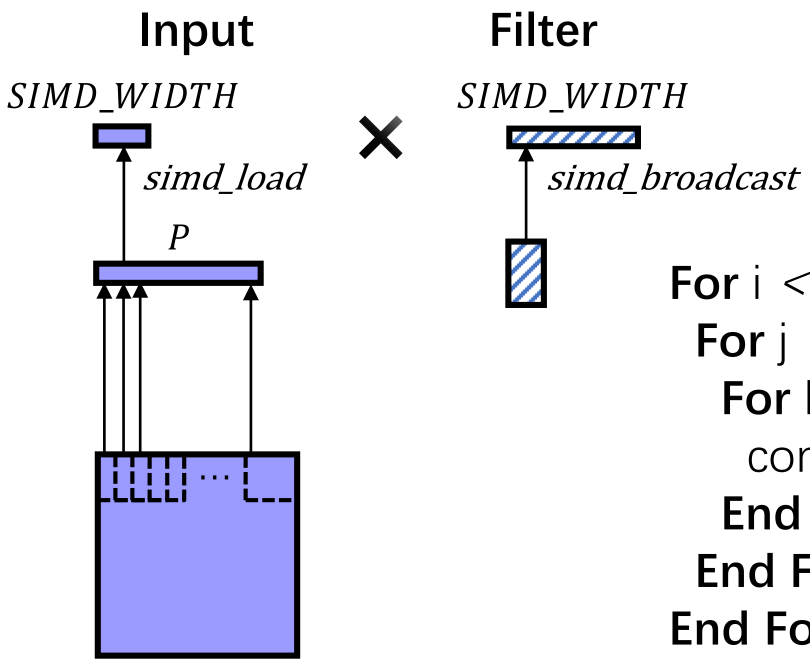


2014 Caffe

2014 **Torch**

Direct Conv+SIMD

Idea: SIMD instruction for each channel



Pros:

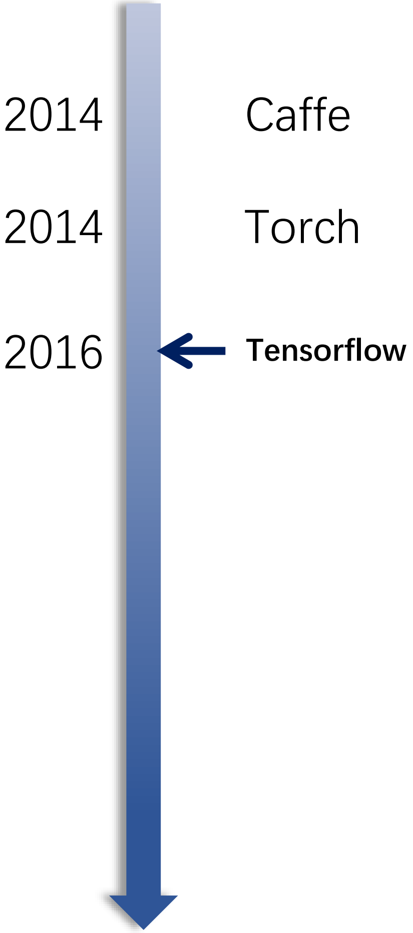
- Without extra memory

Cons:

- Without blocking
- Not cross platform
- Not applicable for stride > 1

```
For i < batch_size Do
  For j < input_channel_size Do
    For k < output_channel_size Do
      convolve_simd(input[i][j], filter[j][k], output[i][k])
    End For
  End For
End For
```

CPU



2014 Caffe

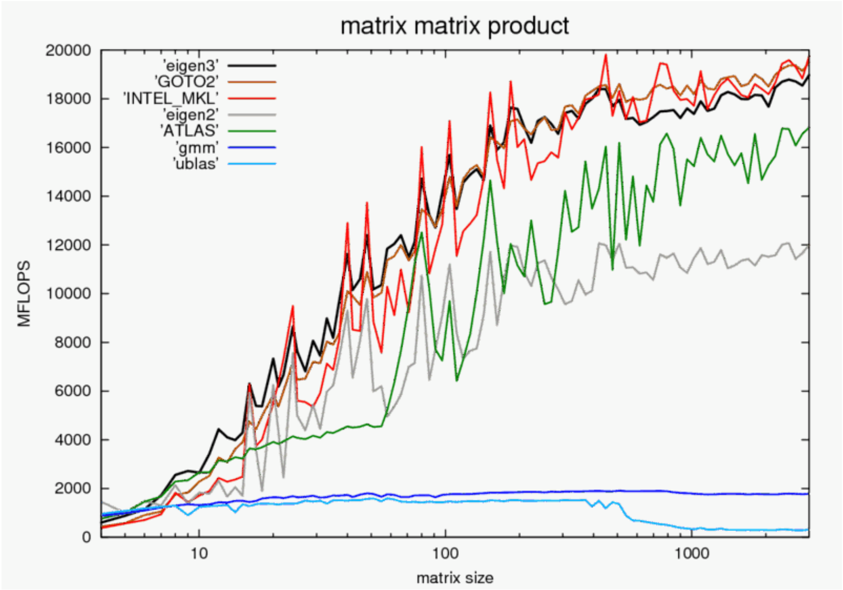
2014 Torch

2016 **Tensorflow**

Eigen *(C++ Template library for linear algebra)*

Idea: Use a mature and open-source product

[http://eigen.tuxfamily.org/index.php?title=Main_Page]



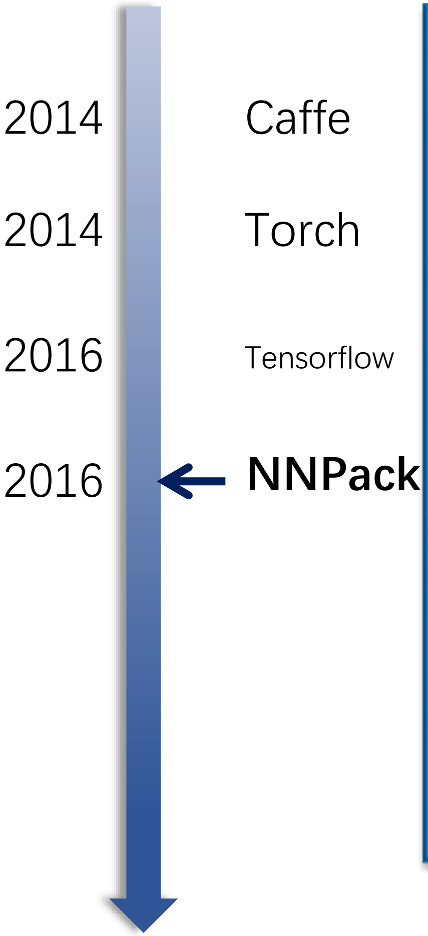
Pros:

- Uniform language interface
- Without extra development

Cons:

- Not extreme performance
- Unable for extension

CPU



2014 Caffe

2014 Torch

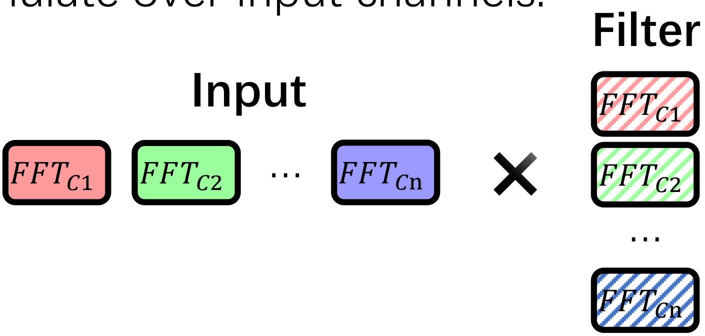
2016 Tensorflow

2016 ← **NNPack**

FFT+Winograd

Idea: Reduce algorithm complexity

1. Traditional FFT is for **single channel** and **stride one** convolution. It adopts **GEMM** to accumulate over input channels.



Library	Caffe	NNPACK
AlexNet:conv2	315 ms	**86 ms**
AlexNet:conv3	182 ms	**44 ms**
AlexNet:conv4	264 ms	**56 ms**
AlexNet:conv5	177 ms	**40 ms**

2. Optimize instructions for *Skylake* (**8 ports** and **4 out-of-order pipeline**)

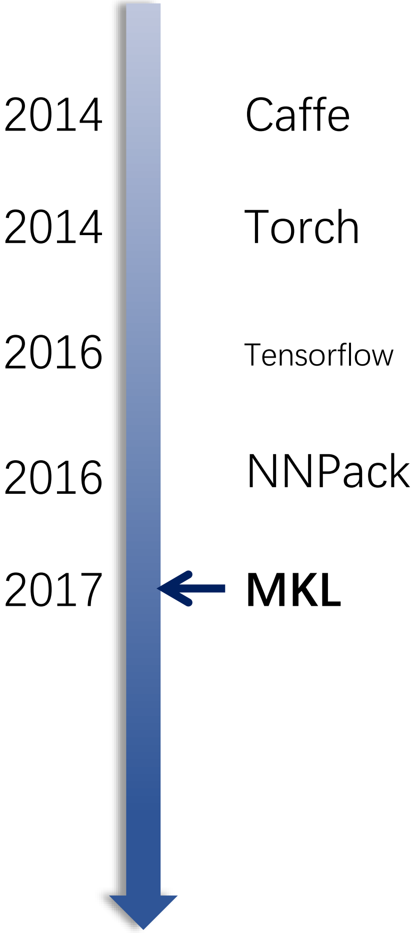
Pros:

- Four times speedup

Cons:

Not applicable for stride > 1

CPU

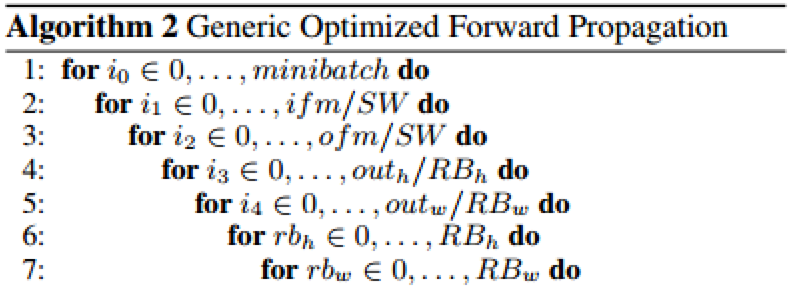


2014	Caffe
2014	Torch
2016	Tensorflow
2016	NNPack
2017	**MKL**

Blocking

Idea: Maximum Byte/Flops ratios. *[Distributed Deep Learning Using Synchronous Stochastic Gradient Descent]*

Original convolution consists of seven loops



Algorithm 2 Generic Optimized Forward Propagation

```
for $i_0 \in 0, \dots, minibatch$ do
  for $i_1 \in 0, \dots, ifm/SW$ do
    for $i_2 \in 0, \dots, ofm/SW$ do
      for $i_3 \in 0, \dots, out_h/RB_h$ do
        for $i_4 \in 0, \dots, out_w/RB_w$ do
          for $rb_h \in 0, \dots, RB_h$ do
            for $rb_w \in 0, \dots, RB_w$ do
```

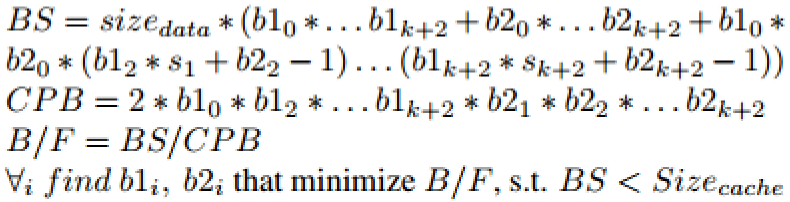
Pros:

- Extreme performance

Cons:

- Not open source
- Require format transformation

Blocking each loop is a constrained minimization problem:



$$BS = size_{data} * (b1_0 * \dots b1_{k+2} + b2_0 * \dots b2_{k+2} + b1_0 * b2_0 * (b1_2 * s_1 + b2_2 - 1) \dots (b1_{k+2} * s_{k+2} + b2_{k+2} - 1))$$

$$CPB = 2 * b1_0 * b1_2 * \dots b1_{k+2} * b2_1 * b2_2 * \dots b2_{k+2}$$

$$B/F = BS/CPB$$

$$\forall_i \ find\ b1_i,\ b2_i \text{ that minimize } B/F, \text{ s.t. } BS < Size_{cache}$$

70-90% GFLOPS!

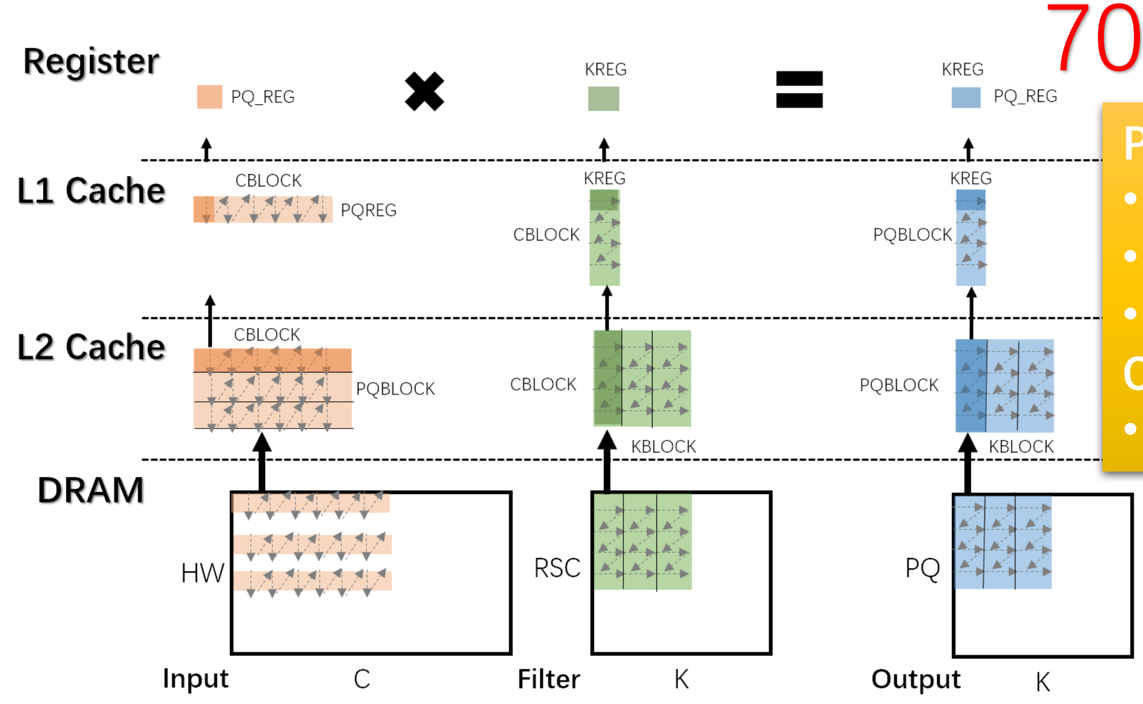
CPU

2014	Caffe
2014	Torch
2016	Tensorflow
2016	NNPack
2017	MKL
2017	← Blitz

Blocking+SIMD+Packing

Idea: Develop general guide lines for optimizing convolution.

70-80% GFLOPS!



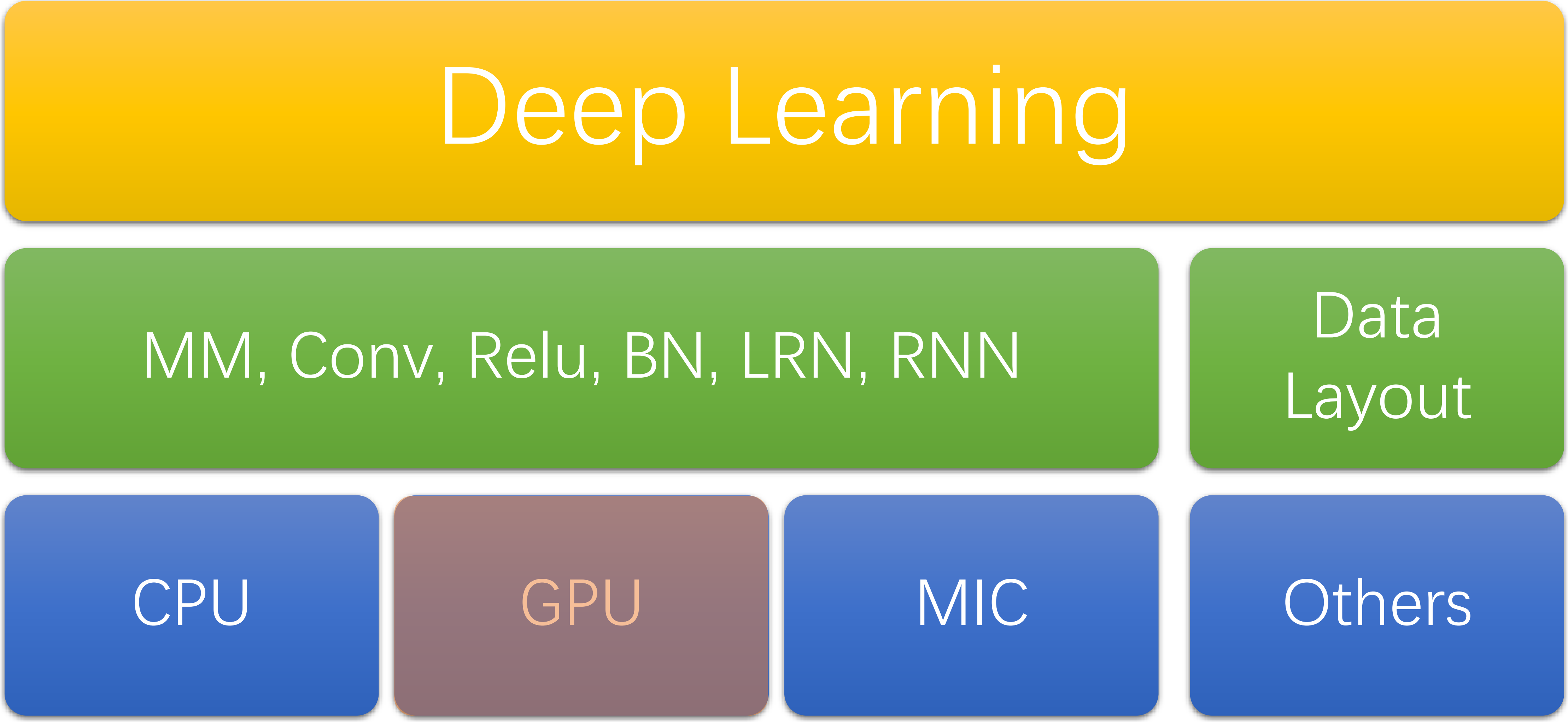
Pros:

- High performance
- Open source
- Easy tuning parameters

Cons:

- Only a single format

HPC Software Stack



Deep Learning

MM, Conv, Relu, BN, LRN, RNN

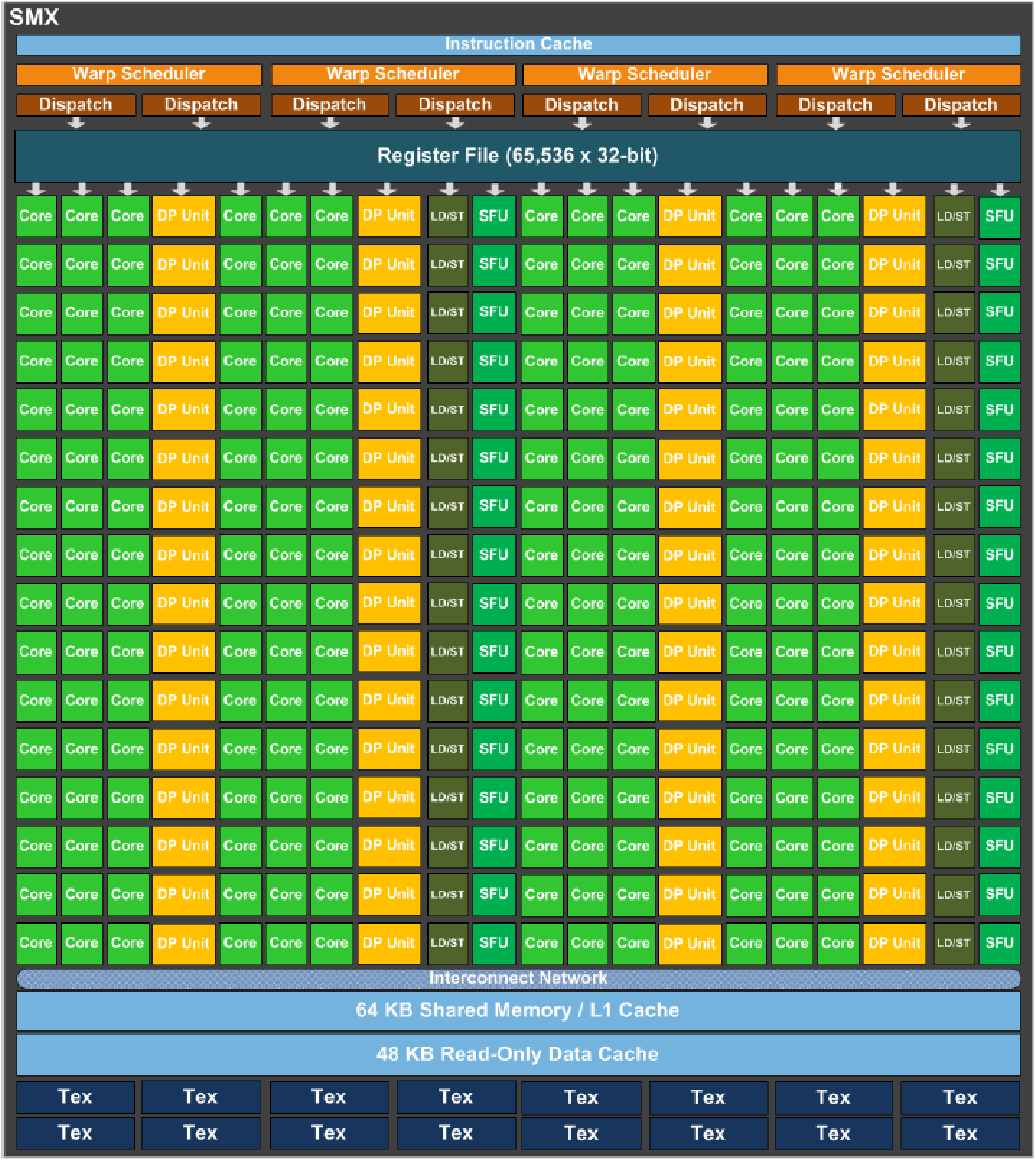
Data Layout

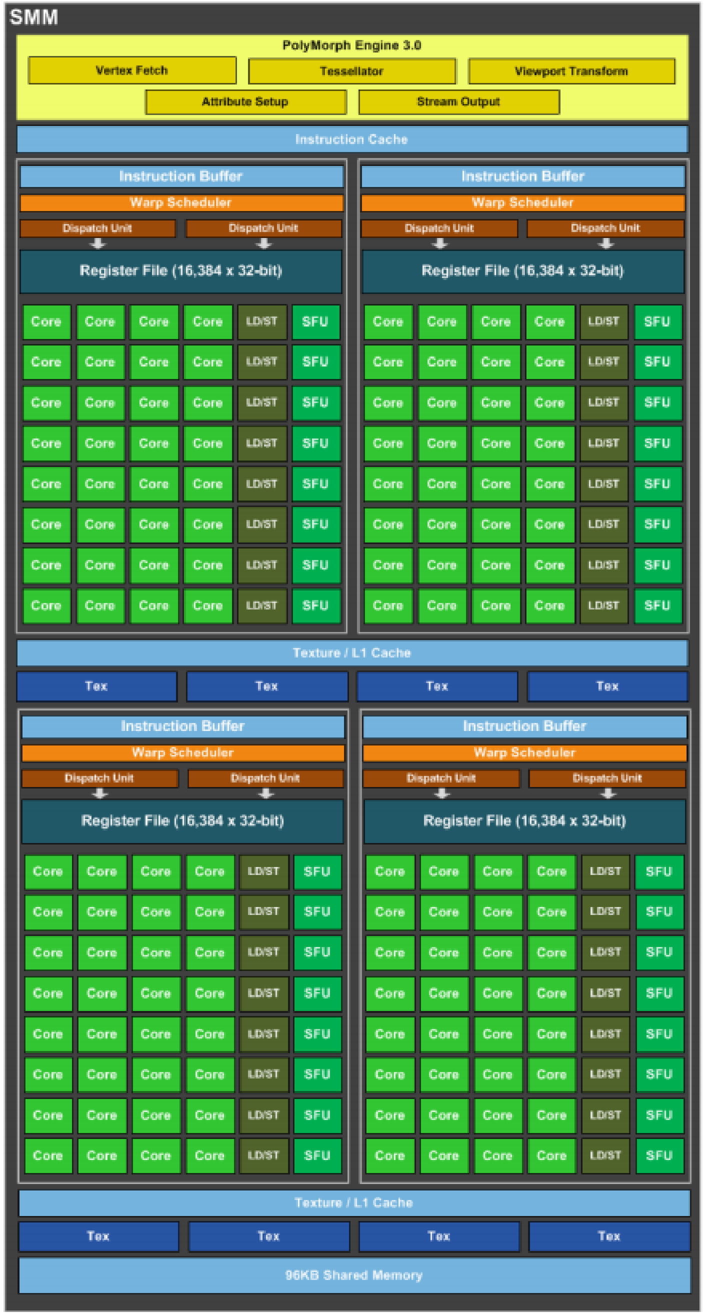
CPU

GPU

MIC

Others


SMX
Instruction Cache
Warp Scheduler
Warp Scheduler
Warp Scheduler
Warp Scheduler
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Register File (65,536 x 32-bit)
Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU Core Core Core DP Unit Core Core Core DP Unit LD/ST SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex Tex Tex Tex Tex Tex Tex Tex
Tex Tex Tex Tex Tex Tex Tex Tex


SMM
PolyMorph Engine 3.0
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Stream Output
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core Core Core Core LD/ST SFU
Texture / L1 Cache
Tex Tex Tex Tex
Instruction Buffer
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (16,384 x 32-bit)
Core Core Core Core LD/ST SFU
Texture / L1 Cache
Tex Tex Tex Tex
96KB Shared Memory

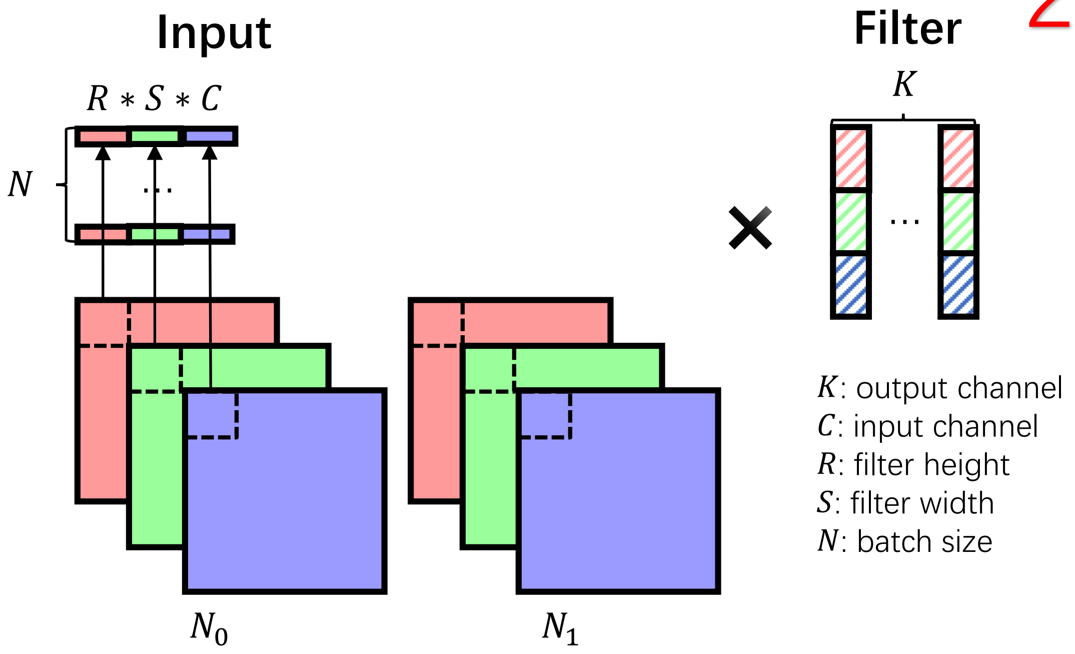
GPU

2013

cudaConvnet

CHWN+GEMM

Idea: Training often use large batch size (N) such that $N > P \times Q$.



K: output channel
C: input channel
R: filter height
S: filter width
N: batch size

20-40% GFLOPS!

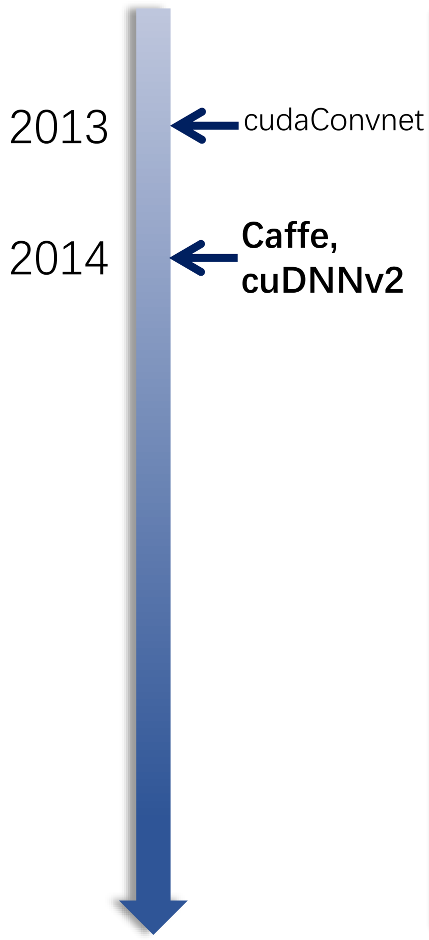
Pros:

- Contiguous memory
- Multi-GPU optimization

Cons:

- NVCC support

GPU

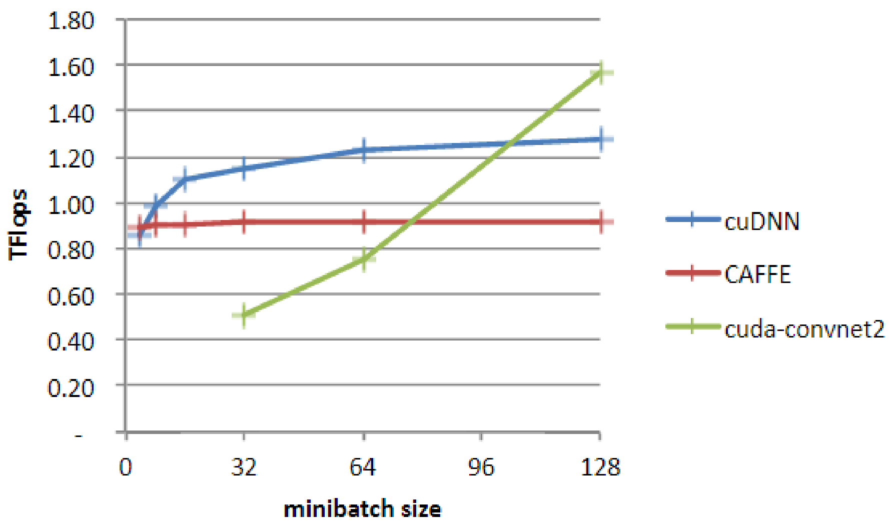


2013 cudaConvnet

2014 **Caffe, cuDNNv2**

CHWN+GEMM

Idea: Lowering, similar as we mentioned in Caffe CPU.



K40: 4.3T GLFOPS

Pros:

- Integrated interfaces
- Documents
- Same pattern as CPU

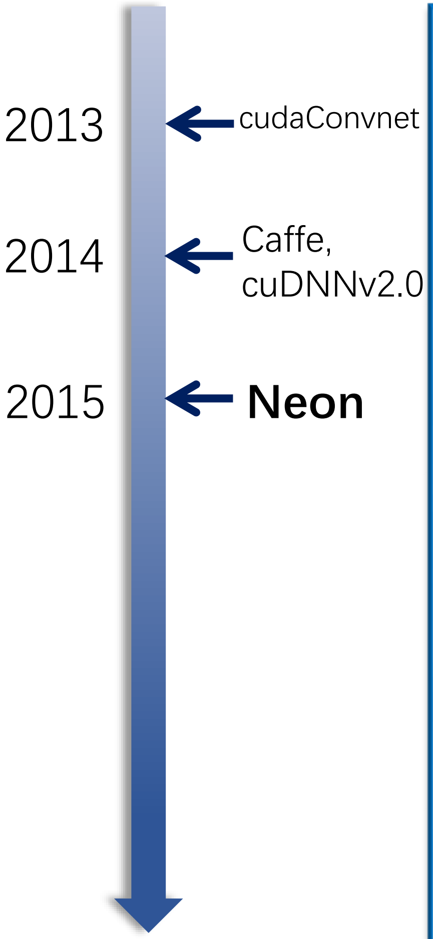
Cons:

- Not efficient
- Require large memory

im2col(input, workspace)
gemm(workspace, filter, output)

25% GFLOPS!

GPU



2013 cudaConvnet

2014 Caffe, cuDNNv2.0

2015 **Neon**

CHWN+Assembly

Idea: based on cudaConvnet**,** NVCC is not efficient in transforming Cuda C to assembly instructions.

80-95% GFLOPS!

```
FFMA R21, R76.reuse, R67.reuse, R21;
FFMA R20, R77.reuse, R67, R20;
FFMA R22, R77.reuse, R65.reuse, R22;
```

Register Reuse

```
FFMA R23, R76.reuse, R65, R23;
FFMA R49, R76.reuse, R70.reuse, R49;
FFMA R48, R77.reuse, R70, R48;
```

```
{FFMA R16, R77.reuse, R66, R16;
@P0 STS.128 [R114+0x1000], R100;}
```

Dual Issue

Pros:

- Extreme performance
- Open source assembler

Cons:

- Only support CHWN
- Perform bad when N is small
- Fixed K, C, and N parameters

GPU

2013 cudaConvnet

2014 Caffe, cuDNNv2

2015 Neon

2016 **cuDNNv4** **TensorRT**

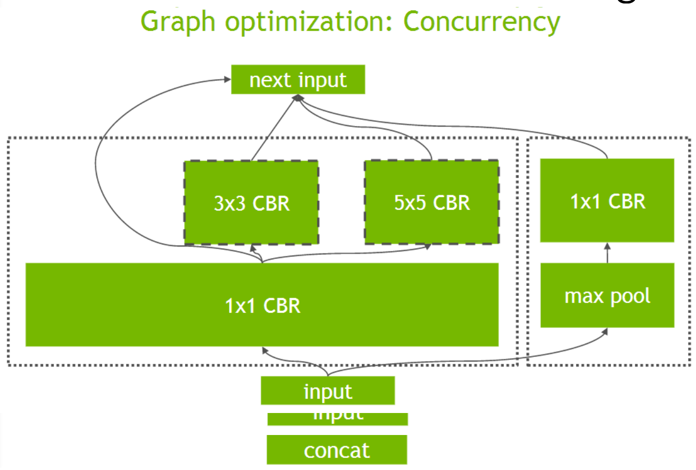
cuDNNv4: FFT+Multiple GEMM implementations

Idea: Using shared memory for blocking and FbFFT [*Fast Convolutional Nets With fbfft: A GPU Performance Evaluation*].

GEMM: 50-60% GFLOPS!

TensorRT: Graph+cuBLAS intrinsics+mixed precision

Idea: Parameters do not change in an inference engine



13% Speedup!

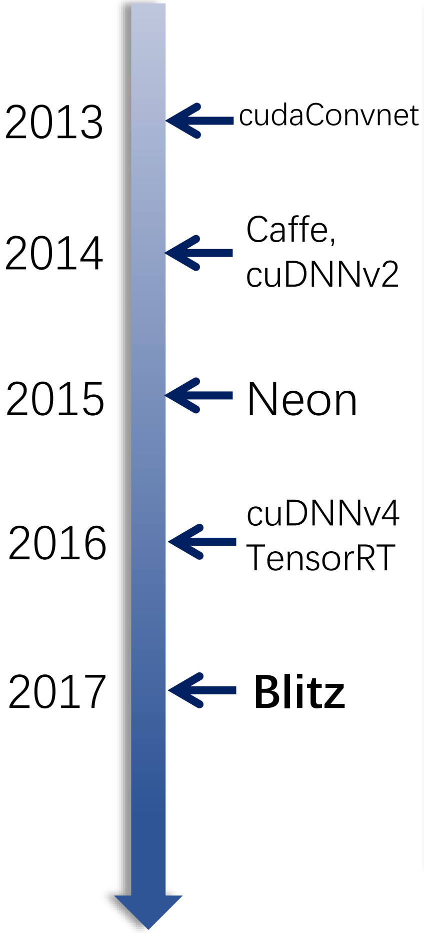
Pros:

- Efficient suitcase

Cons:

- Performance
- Hardware support for low precision

GPU



2013 cudaConvnet

2014 Caffe, cuDNNv2

2015 Neon

2016 cuDNNv4 TensorRT

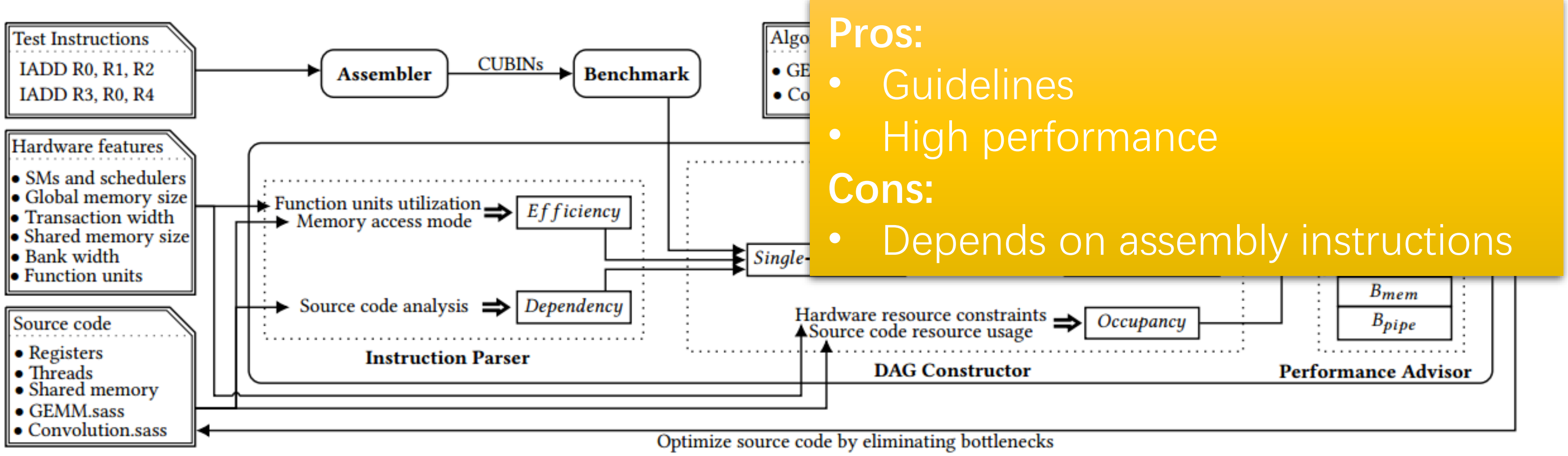
2017 **Blitz**

Assembly Performance Modeling

Idea: Performance modeling based on C or PTX is not accurate.

- Four potential bottlenecks: B_{ilp}, B_{comp}, B_{mem}, B_{pipe}
- Optimizing GEMM and convolution

84% GFLOPS!



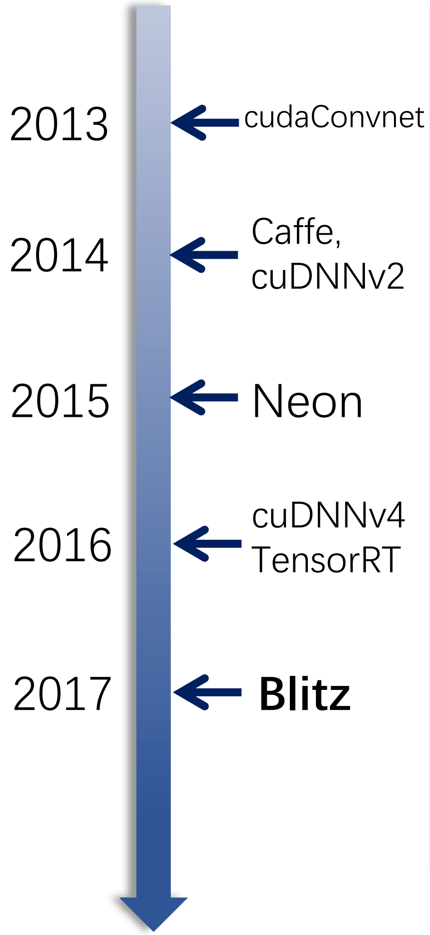
Pros:

- Guidelines
- High performance

Cons:

- Depends on assembly instructions

GPU



2013 cudaConvnet

2014 Caffe, cuDNNv2

2015 Neon

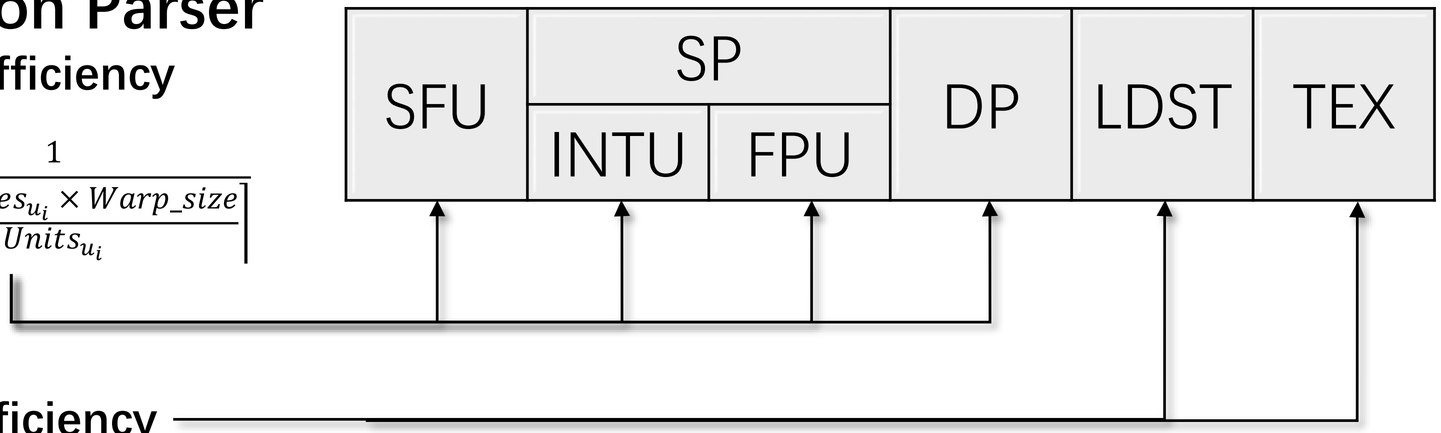
2016 cuDNNv4 TensorRT

2017 **Blitz**

Instruction Parser

Compute Efficiency

$$E_{u_i} = \frac{1}{\left\lceil \frac{Dispatches_{u_i} \times Warp\_size}{Units_{u_i}} \right\rceil}$$



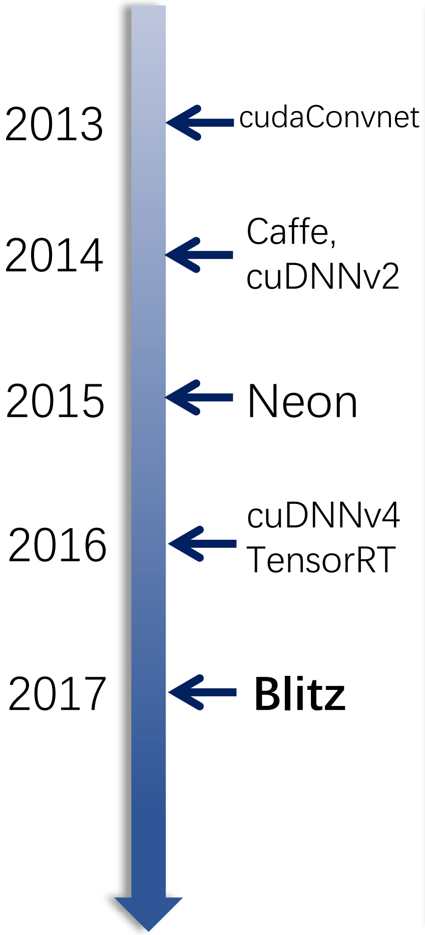
Memory Efficiency

$$E_{shared} = \frac{1}{N_i \times \left\lceil \frac{Dispatches_{ldst} \times Warp\_size}{Units_{ldst}} \right\rceil \times \left\lceil \frac{Ins\_width_i}{Bank\_width} \right\rceil}$$

$$E_{global} = \frac{1}{N_i \times \left\lceil \frac{Dispatches_{ldst} \times Warp\_size}{Units_{ldst}} \right\rceil}$$

$Dispatches_{u_i}$: Dispatches for units u_i
$Warp\_size$: Number of threads in a warp
$Units_{u_i}$: Number of units u_i
Ins_width_i: Instruction memory width
$Bank\_width$: Shared memory bank width
N_i: Number of memory transactions

GPU



2013 ← cudaConvnet

2014 ← Caffe, cuDNNv2

2015 ← Neon

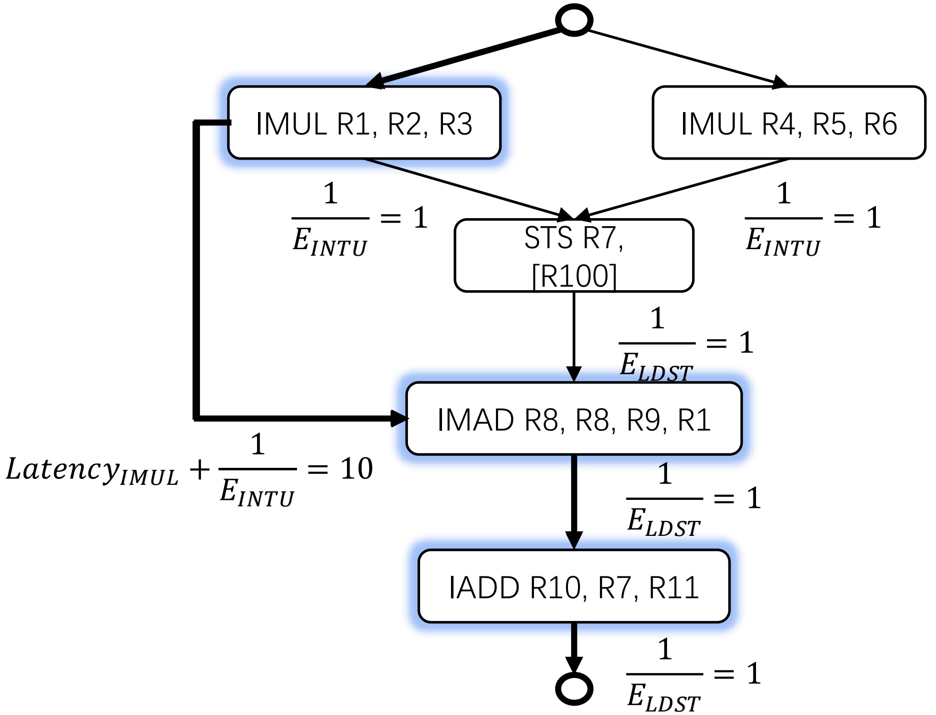
2016 ← cuDNNv4 TensorRT

2017 ← **Blitz**

DAG Constructor

Idea: Construct a DAG to simulate instruction execution

```
--:-:-:-:01 IMUL R1, R2, R3
--:-:-:-:00 IMUL R4, R5, R6
--:-:-:-:01 STS R7, [R100]
--:-:-:-:01 IMAD R8, R8, R9, R1
--:-:-:-:01 IADD R10, R7, R11
```



IMUL R1, R2, R3

IMUL R4, R5, R6

$\frac{1}{E_{INTU}} = 1$

$\frac{1}{E_{INTU}} = 1$

STS R7, [R100]

$\frac{1}{E_{LDST}} = 1$

IMAD R8, R8, R9, R1

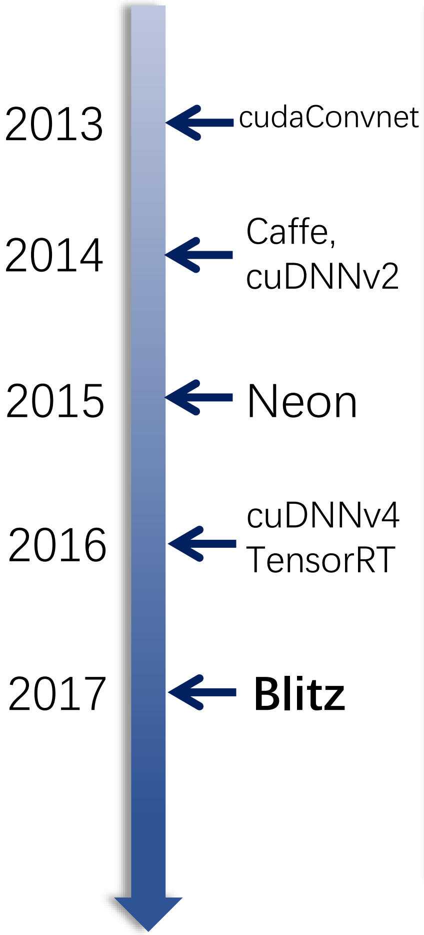
$Latency_{IMUL} + \frac{1}{E_{INTU}} = 10$

$\frac{1}{E_{LDST}} = 1$

IADD R10, R7, R11

$\frac{1}{E_{LDST}} = 1$

GPU



2013 cudaConvnet

2014 Caffe, cuDNNv2

2015 Neon

2016 cuDNNv4 TensorRT

2017 **Blitz**

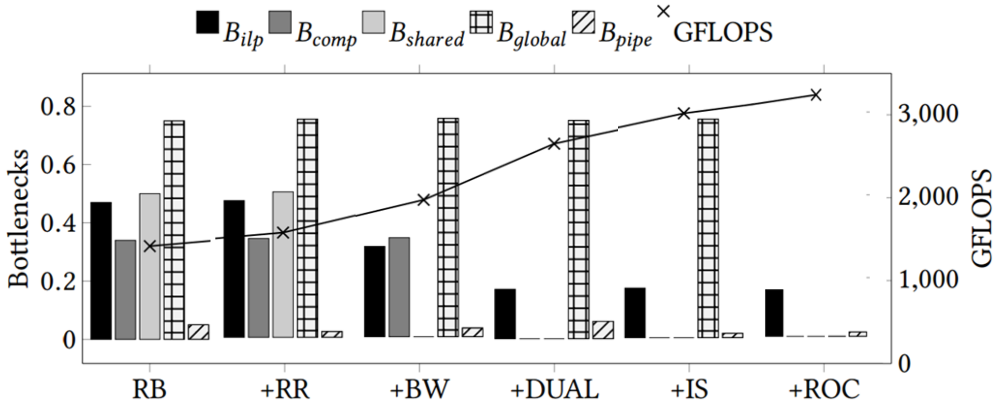
Performance Advisor

B_{ilp} ： Instruction level parallel bottleneck, optimized by issuing instructions simultaneously. （+DUAL)

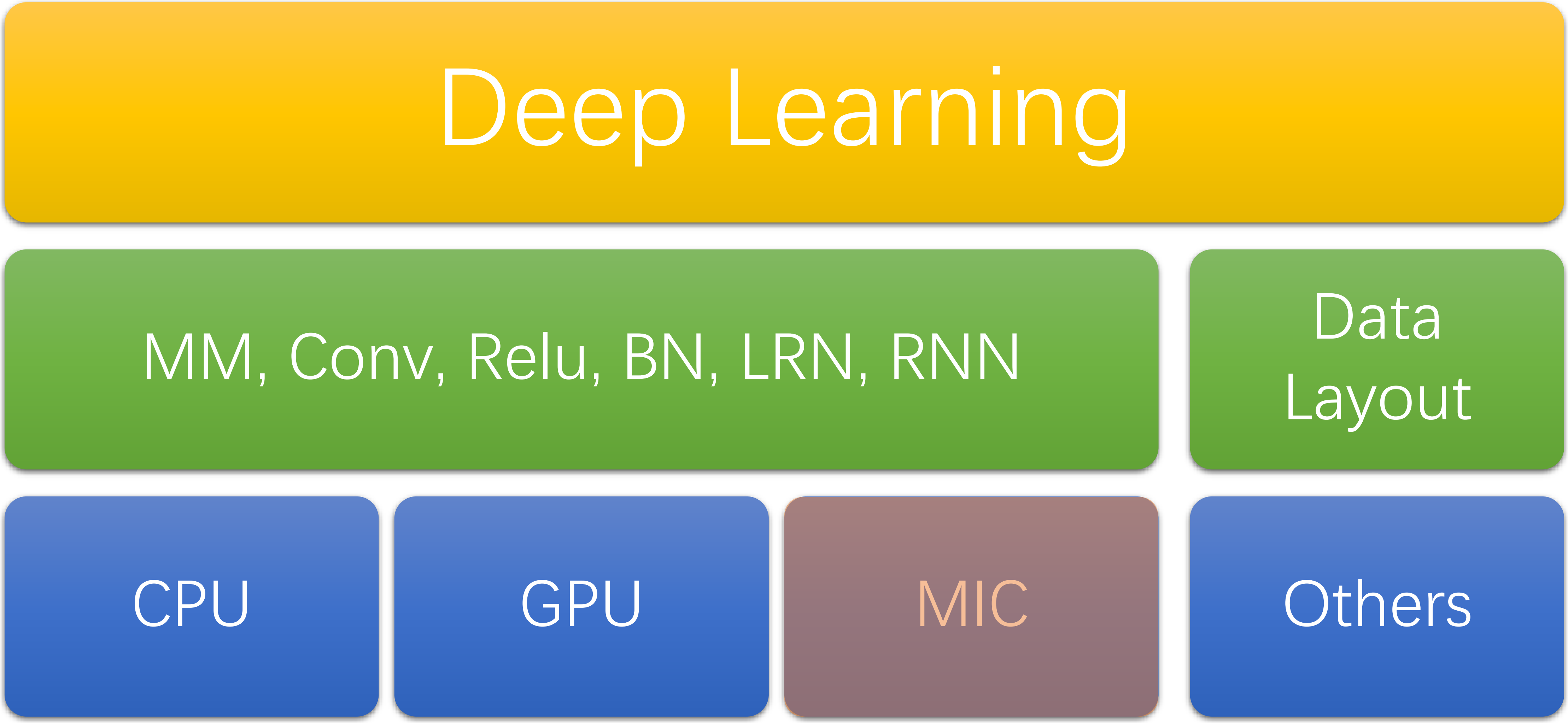
B_{comp} ： Compute resource usage bottleneck, optimized by using more compute units. (+DUAL)

B_{mem} ： Memory access bottleneck, optimized by high bandwidth instruction (+BW) and read-only-cache (+ROC).

B_{pipe} ： Instruction pipeline bottleneck, optimized by instruction scheduling (+IS) and register reuse (RR).



HPC Software Stack



Deep Learning

MM, Conv, Relu, BN, LRN, RNN

Data Layout

CPU

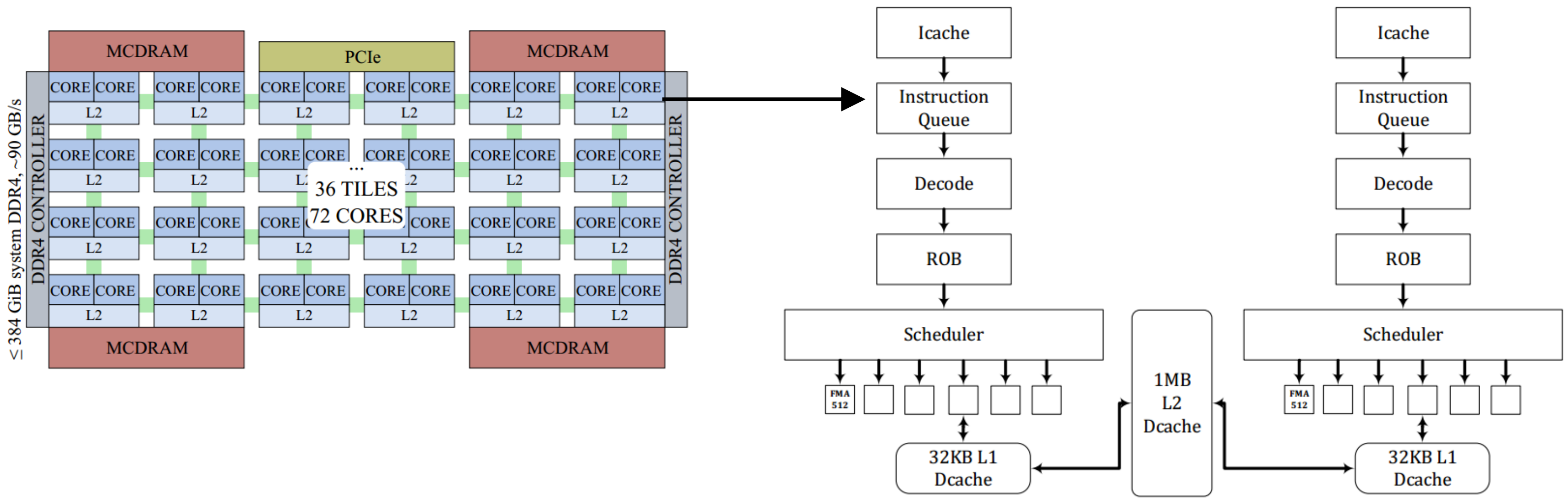
GPU

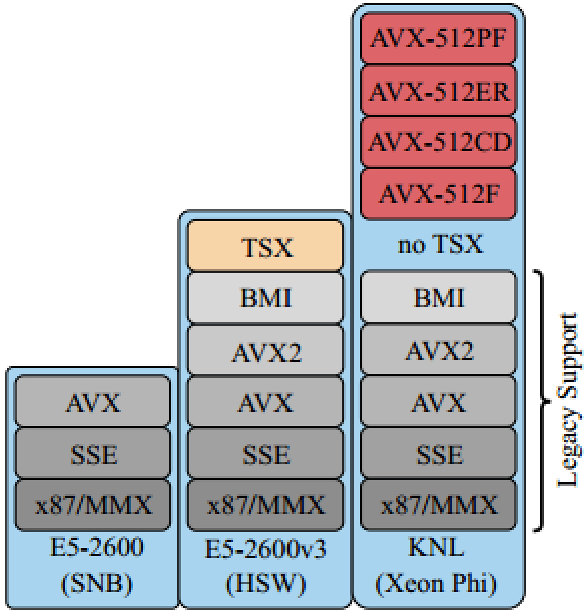
MIC

Others

MIC

- MIC Architecture
  - Cache Hierarchy
  - Instruction Scheduling
  - SIMD Instructions

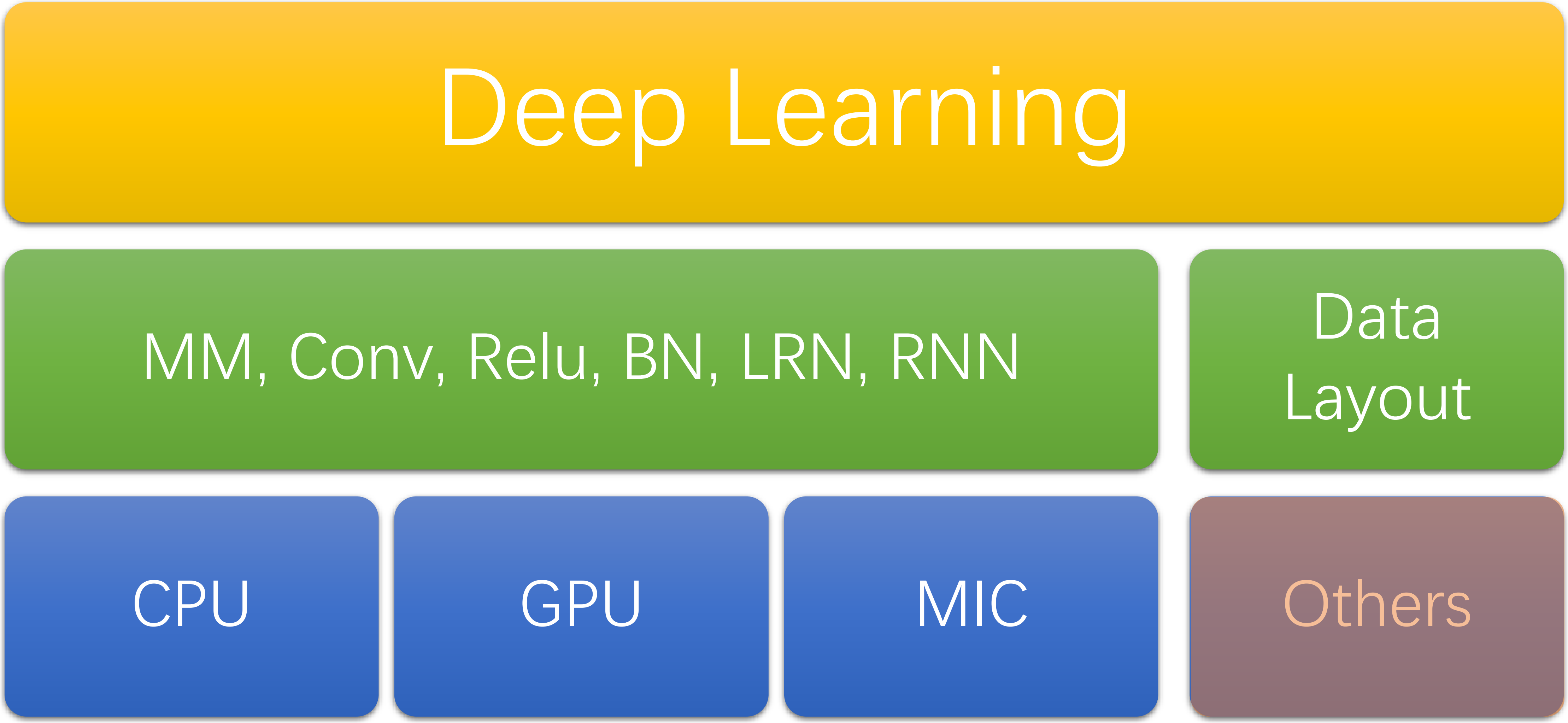




MIC

- Caffe
  - IM2COL+GEMM
  - 7% GFLOPS
- Libxsmm
  - JIT
  - 30%-80% GFLOPS
- MKL
  - Hand written ASMs
  - 80%GFLOPS
- Blitz
  - Cache blocking
  - 40% GFLOPS

HPC Software Stack



Deep Learning

MM, Conv, Relu, BN, LRN, RNN

Data Layout

CPU

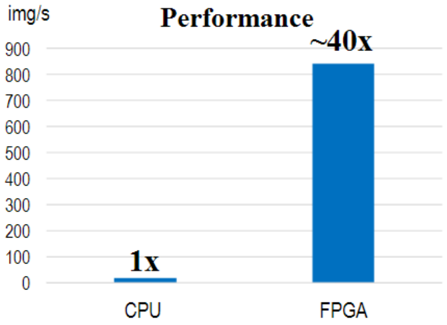
GPU

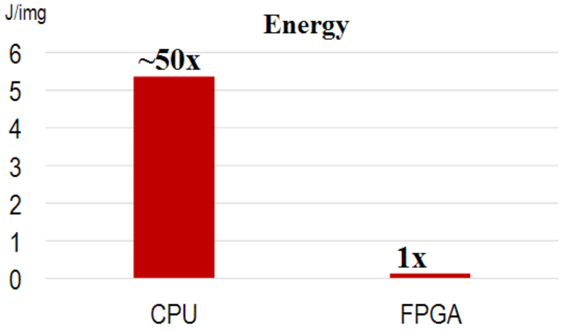
MIC

Others

Devices Accelerators

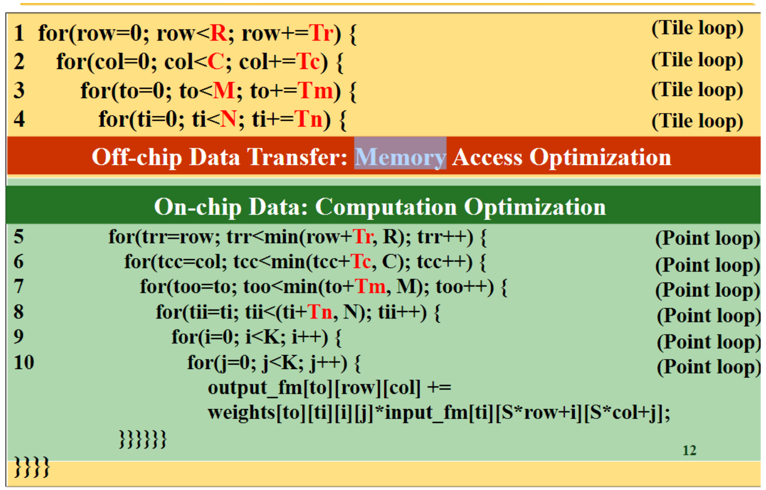
- FPGA
  - Configurable hardware
  - Both compute and energy efficient
- TPU, Cambrian





	CPU (Caffe+ATLAS)	FPGA
Device	E7-4807	VC709
Power	100 W	90W
Precision	float	~~Float~~ fixed-point
Frequency	1.87 GHz	~~50MHz~~ 200MHz
Process	32nm	28nm

24



```
for(row=0; row<R; row+=Tr) {                              (Tile loop)
  for(col=0; col<C; col+=Tc) {                            (Tile loop)
    for(to=0; to<M; to+=Tm) {                             (Tile loop)
      for(ti=0; ti<N; ti+=Tn) {                           (Tile loop)
          Off-chip Data Transfer: Memory Access Optimization

          On-chip Data: Computation Optimization
        for(trr=row; trr<min(row+Tr, R); trr++) {         (Point loop)
          for(tcc=col; tcc<min(tcc+Tc, C); tcc++) {       (Point loop)
            for(too=to; too<min(to+Tm, M); too++) {       (Point loop)
              for(tii=ti; tii<(ti+Tn, N); tii++) {        (Point loop)
                for(i=0; i<K; i++) {                      (Point loop)
                  for(j=0; j<K; j++) {                    (Point loop)
                    output_fm[to][row][col] +=
                    weights[to][ti][i][j]*input_fm[ti][S*row+i][S*col+j];
        }}}}}}
}}}}
```

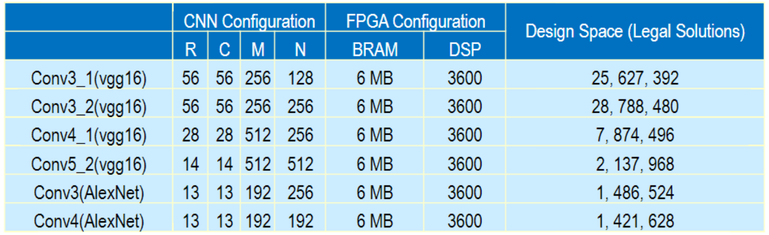
12

Constraints on CNN configuration

$$\begin{cases} 0 < Tm < M \\ 0 < Tn < N \\ 0 < Tr < R \\ 0 < Tc < C \end{cases}$$

Constraints on FPGA resource

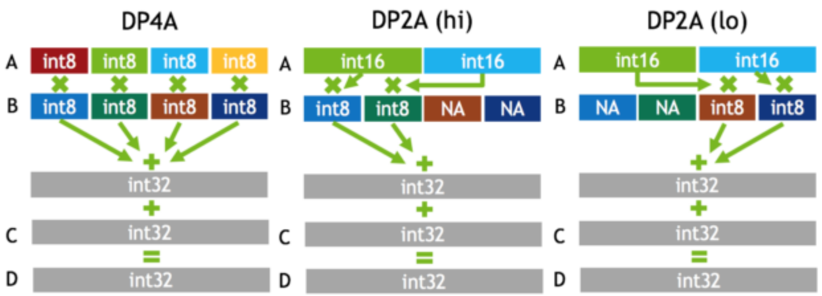
$$\begin{cases} 0 < Tm*Tn < \text{DSP resource} \\ 0 < Tn*Tr*Tc + Tm*Tr*Tc + Tm*Tn*K*K < \text{BRAM resource} \end{cases}$$



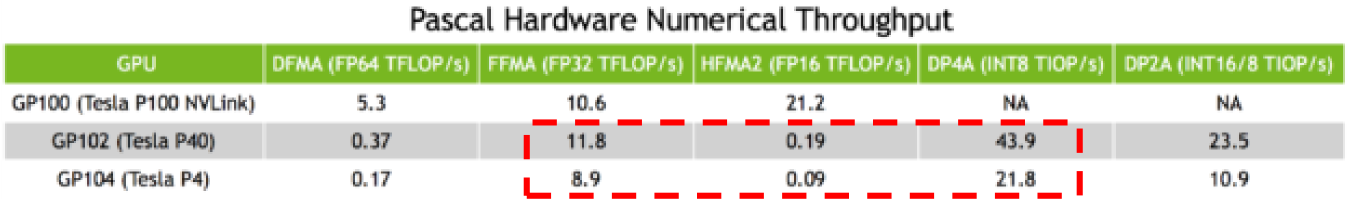
	CNN Configuration				FPGA Configuration		Design Space (Legal Solutions)
	R	C	M	N	BRAM	DSP	
Conv3_1(vgg16)	56	56	256	128	6 MB	3600	25, 627, 392
Conv3_2(vgg16)	56	56	256	256	6 MB	3600	28, 788, 480
Conv4_1(vgg16)	28	28	512	256	6 MB	3600	7, 874, 496
Conv5_2(vgg16)	14	14	512	512	6 MB	3600	2, 137, 968
Conv3(AlexNet)	13	13	192	256	6 MB	3600	1, 486, 524
Conv4(AlexNet)	13	13	192	192	6 MB	3600	1, 421, 628

Devices Accelerators

- Mixed Precision
- Using less bits for **energy, memory, and speed efficiency.**
- Energy efficiency
  - Fewer memory loads, save memory bandwidth
- Memory efficiency
  - Less parameter storage
- Speed efficiency
  - Operate more numbers in a cycle with available **architectural support**.



GP102



Pascal Hardware Numerical Throughput

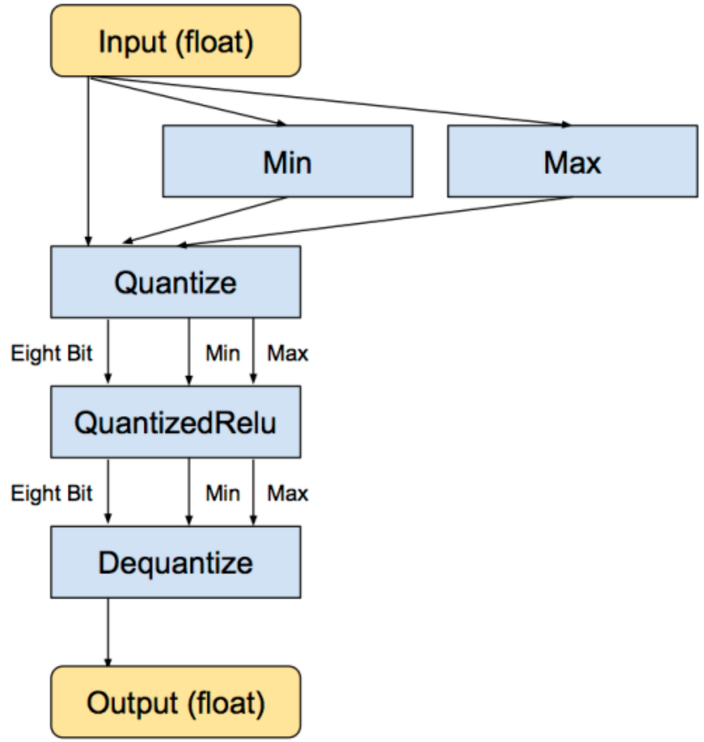
GPU	DFMA (FP64 TFLOP/s)	FFMA (FP32 TFLOP/s)	HFMA2 (FP16 TFLOP/s)	DP4A (INT8 TIOP/s)	DP2A (INT16/8 TIOP/s)
GP100 (Tesla P100 NVLink)	5.3	10.6	21.2	NA	NA
GP102 (Tesla P40)	0.37	11.8	0.19	43.9	23.5
GP104 (Tesla P4)	0.17	8.9	0.09	21.8	10.9

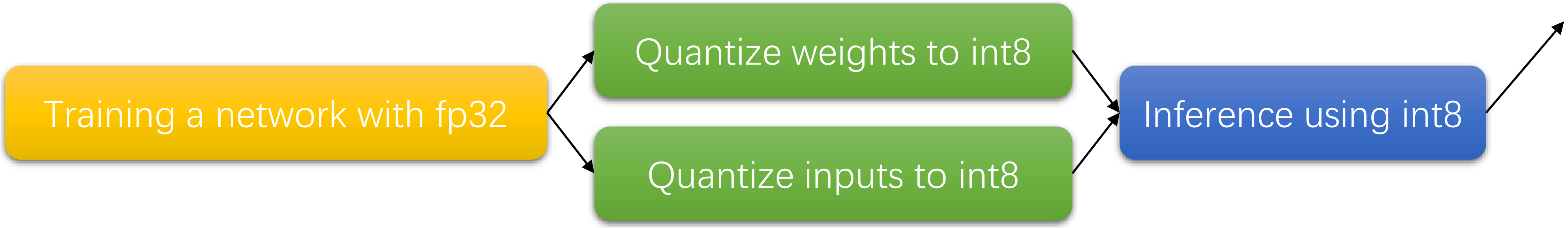
Devices Accelerators

- Mixed Precision
- 1-bit

$$w_b = \begin{cases} +1\ with\ probability\ p = \sigma(w), \\ -1\ with\ probability\ 1 - p \end{cases} \qquad w_b = \begin{cases} +1\ if\ w \geq 0, \\ -1\ otherwise \end{cases}$$

- 8-bit
  - CPU->short
  - GPU->int8




Training a network with fp32

Quantize weights to int8

Quantize inputs to int8

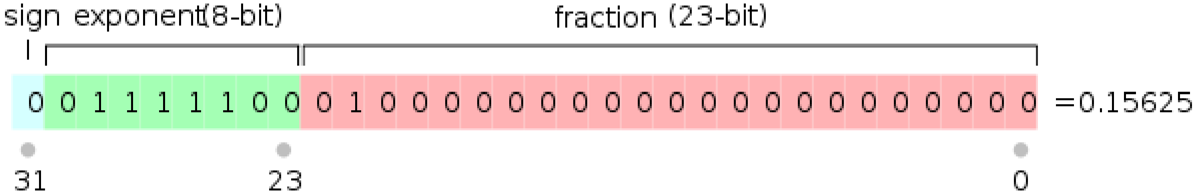
Inference using int8

Devices Accelerators

- Why could we quantize?
  - Constant parameters can be grouped into several ranges
  - E.g. Alexnet-2 $1e^{-2} - 1e^{-3}$
- Quantization Methods
- Clip

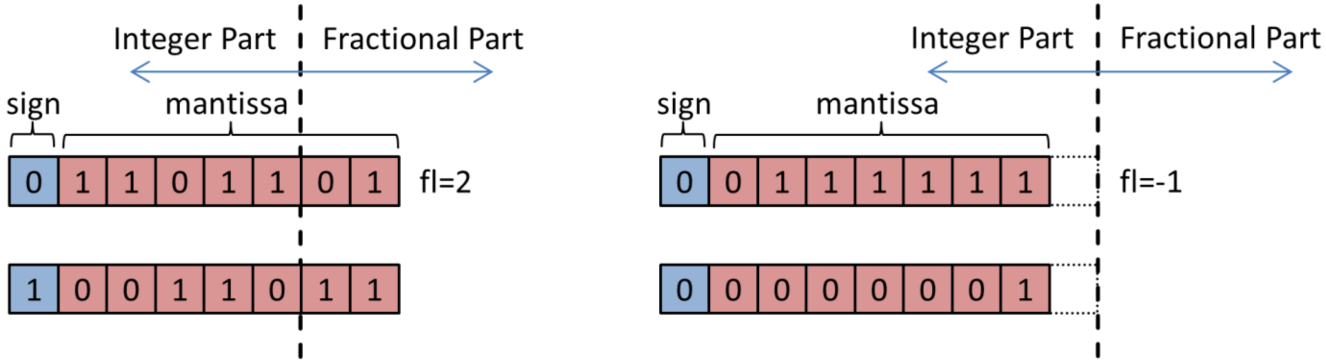
$$I = Clip(MaxV, MinV, round\_bit(F))$$

- Floating number

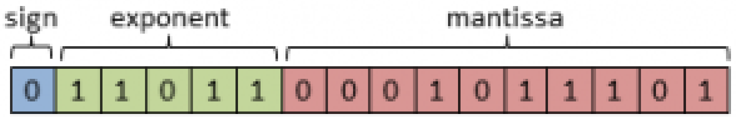


Devices Accelerators

- Dynamic Fixed Point
- $n = (-1)^s \cdot 2^{-fl} \cdot \sum_{i=0}^{B-2} 2^i \cdot x_i$



- Mini-float



- Multiplier-free

$$n = (-1)^s \cdot 2^{exp}$$

HPC Software Stack

Deep Learning

Co-design Hardwares for Conv, MM, Relu, BN, LRN, RNN

HPC Software Stack

Deep Learning-DSL

Co-design Hardwares for Conv, MM, Relu, BN, LRN, RNN

Suggestions

- Broader: ICLR, ICML, NIPS…
- Deeper: Intel and Nvidia Reference Guides

Thanks!