Part 3. AI/ML systems

3.1: AI/ML systems overview

Insop S., 7/30/2024, Ver 0.4

AI learning modules

Part 1 AI/ML foundation

1.1 AI/ML overview — Recorded on 7/23

Part 2 Generative AI

2.1: Generative AI/LLM overview — Recorded on 7/26

2.2: LLM applications — Will be scheduled

Part 3 AI/ML systems

3.1: AI/ML systems overview — We're here, scheduled on 7/30

3.2: Efficient AI/ML systems — Will be scheduled on 8/6

3. AI/ML systems

Contents

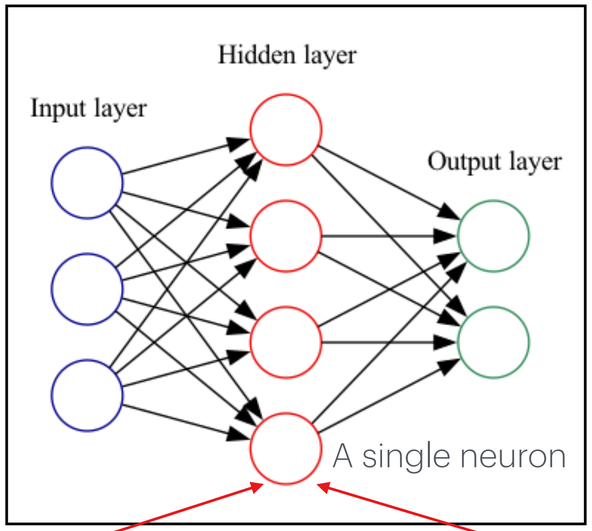
Review

From Part 1. AI/ML foundation

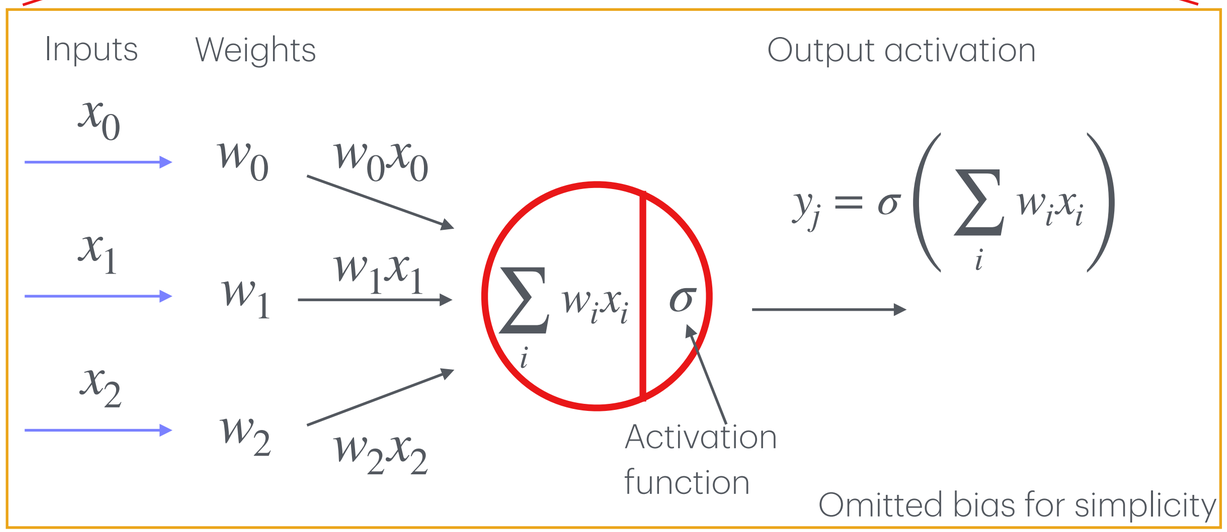
- Machine Learning (ML) learns how to solve problems through data and statistical methods without explicit programming
- Three key ingredients in ML:
  - *hypothesis* (model),
  - *loss* (defines the goodness of the model), and
  - *optimization* (how to find the parameters that minimize the loss).
- A neural network is a basic building block of deep learning and consists of fully connected layers with non-linear activation functions that can learn non-linear boundaries.

ℹ️ Check out the part 1 slide for references, books and neural network code

2 layers neural network with 1 hidden layer

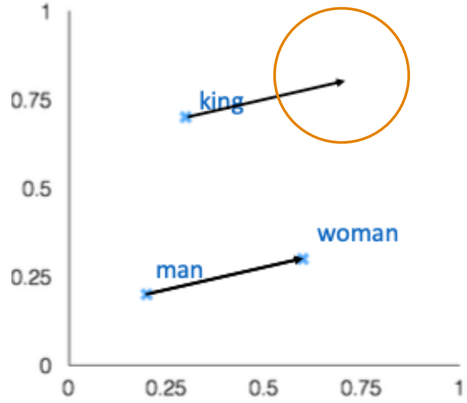


A single neuron internal view

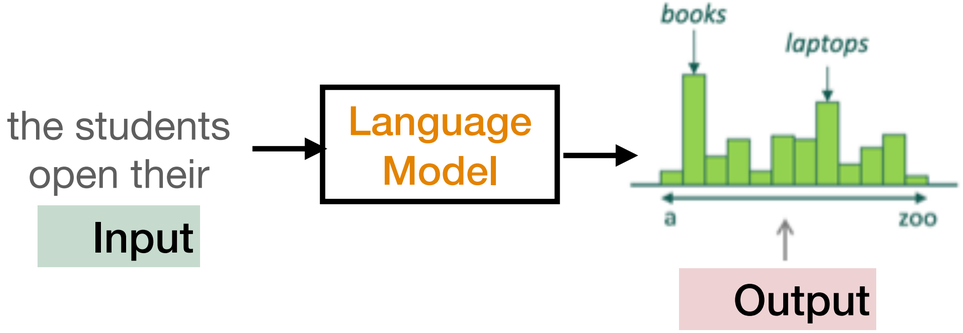


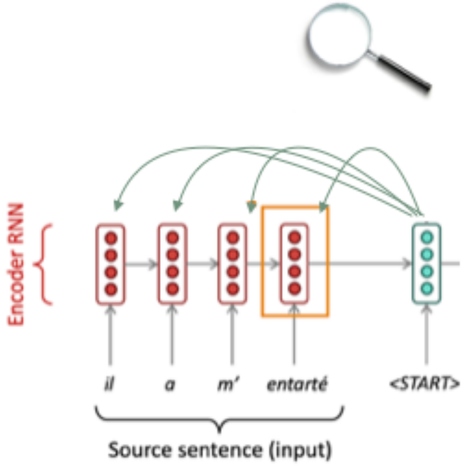
Review

From Part 2.1: Generative AI/LLM overview



- Word2vec: a model that trained to converts words to vector (word embeddings)
- LM (language model): predicts next word given input
- RNN: type of neural network that can process sequential data, which allows stateful computation since it uses hidden state from previous time. Any input sequence length can be processed
- Sequence to sequence model: consists of encoder block and decoder block and used for neural machine translation model
- Attention: a general method in neural network that solves information bottleneck, which directly computes the similarities with all input and focus on the specific part of the input

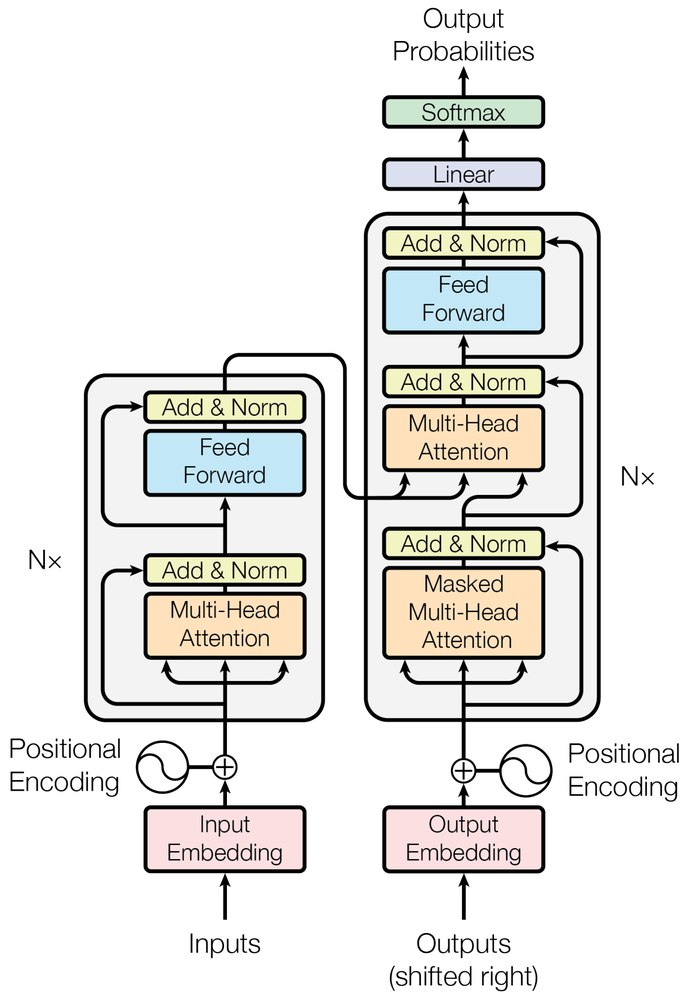




Review

From Part 2.1: Generative AI/LLM overview

- Transformer architecture/model: improves the issues with RNN, such as providing attention paths, enabling parallel computations.
- Transformer block: a core block for the Transformer architecture, which includes *multi-headed self attention* and *feedforward network*.
- Self-supervision: in language model training, it uses the input text to predict the next word without label.
- Pre-training: teaches the model how the world works in general by using self-supervision training
- Fine-tuning: customizes it to a specific task to easier to use.
- Generation (LM inference): the model continues its input (prompt) sequence.

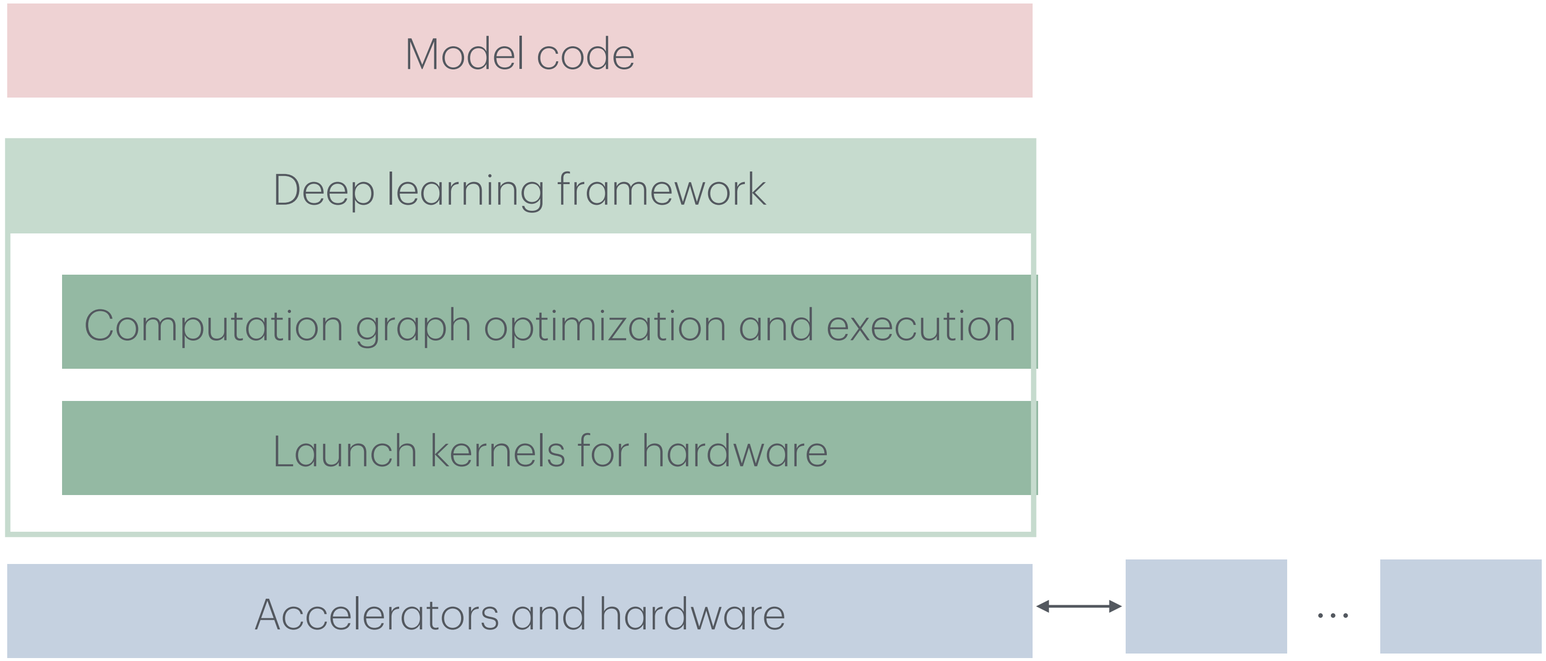


3.1: AI/ML systems overview

Motivation

Learn systems that enable building and running deep learning models

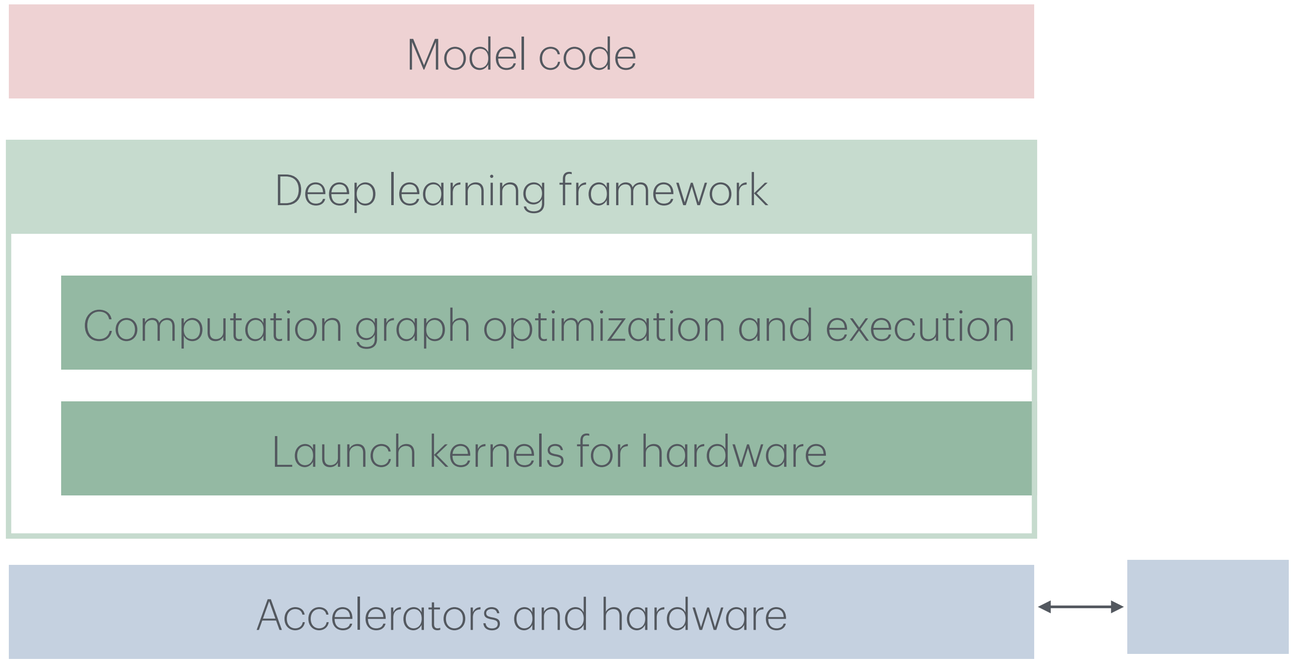
Deep learning systems



Connects multiple hw for large scale train/inference

Plan for today

- Deep learning software & framework
- Accelerator
- Large scale training



Connects multiple hw for
large scale train/inference

Deep learning SW & framework

Deep learning workload

- Define neural network model with weights

A simple code example for model training

```
for epoch in range(max_epochs):
    for data, target in training_data:
        optimizer.zero_grad()
        output = model(data)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
```

Ref. : link

Deep learning workload

- Define neural network model with weights
- compute the output from model

A simple code example for model training

```
for epoch in range(max_epochs):
    for data, target in training_data:
        optimizer.zero_grad()
        output = model(data)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
```

Ref. : link

Deep learning workload

- Define neural network model with weights
- compute the output from model
- Compute loss

A simple code example for model training

```
for epoch in range(max_epochs):
    for data, target in training_data:
        optimizer.zero_grad()
        output = model(data)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
```

Ref. : link

Deep learning workload

- Define neural network model with weights
- compute the output from model
- Compute loss
- Back propagation: compute derivatives using chain rule

A simple code example for model training

```
for epoch in range(max_epochs):
    for data, target in training_data:
        optimizer.zero_grad()
        output = model(data)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
```

Ref. : link

Deep learning workload

- Define neural network model with weights
- compute the output from model
- Compute loss
- Back propagation: compute derivatives using chain rule
- Update weights using computed gradients

A simple code example for model training

```
for epoch in range(max_epochs):
    for data, target in training_data:
        optimizer.zero_grad()
        output = model(data)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
```

Ref. : link

Deep learning framework

Deep learning framework: allows you to create models and run on hardware

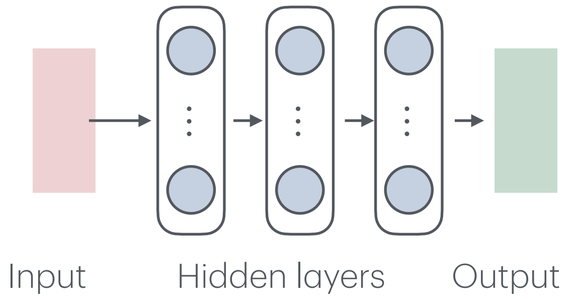
- TensorFlow (2015), *PyTorch (2016)*, JAX (2018) are widely used framework. They are provided as Python library and other language bindings, see comparison in here
- Auto differentiation: Given $y = f(x, w_0, w_1, \dots)$, these framework automatically compute $\frac{\partial y}{\partial w_0}, \frac{\partial y}{\partial w_1}$, ... using chain rule from calculus
- Hardware acceleration: support underlying hardware, such as CPU, GPU, TPU
- Distributed training support: allows to train over multiple servers and accelerators

PyTorch

PyTorch: widely used deep learning framework based on its easy of use and debug. Originally created by FB, and now linux foundation manages the project.

It dynamically creates computation graph, and uses the graph to run forward propagation to compute loss and backward propagation to compute derivatives.

- A simple neural network code (on the right)




```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        logits = self.linear_relu_stack(self.flatten(x))
        return logits

model = NeuralNetwork().to(device)

# inference code
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

Define hidden layers

Forward propagation

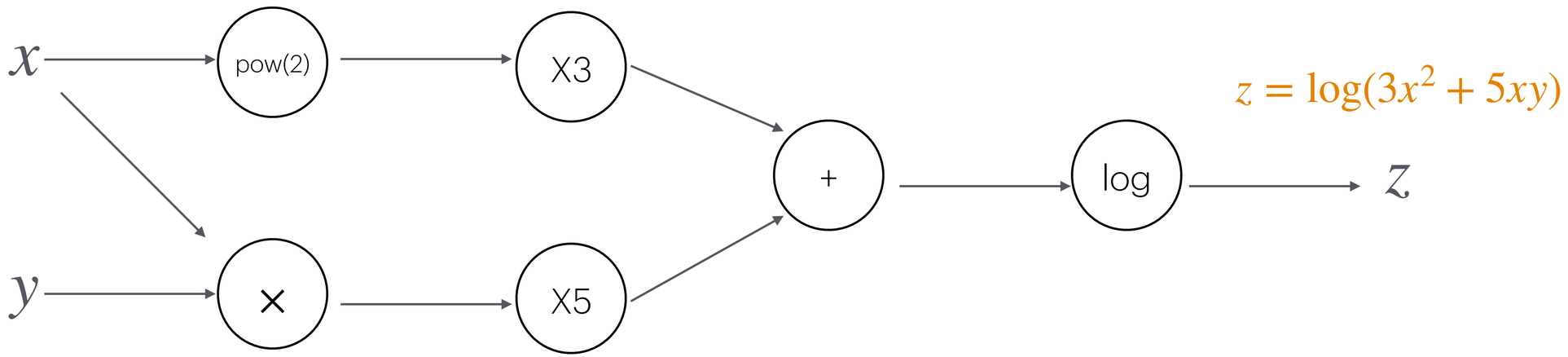
Computation graph

Form computation graph using an example

Computation graph is created when a model (neural network) is defined by DL framework.

Let's use a simple example to create a computation graph. Then check forward propagation and backward propagation.

From this simple example, $z = \log(3x^2 + 5xy)$, we want to compute z and derivative $\frac{\partial z}{\partial x}$ using computation graph



Ref. : link

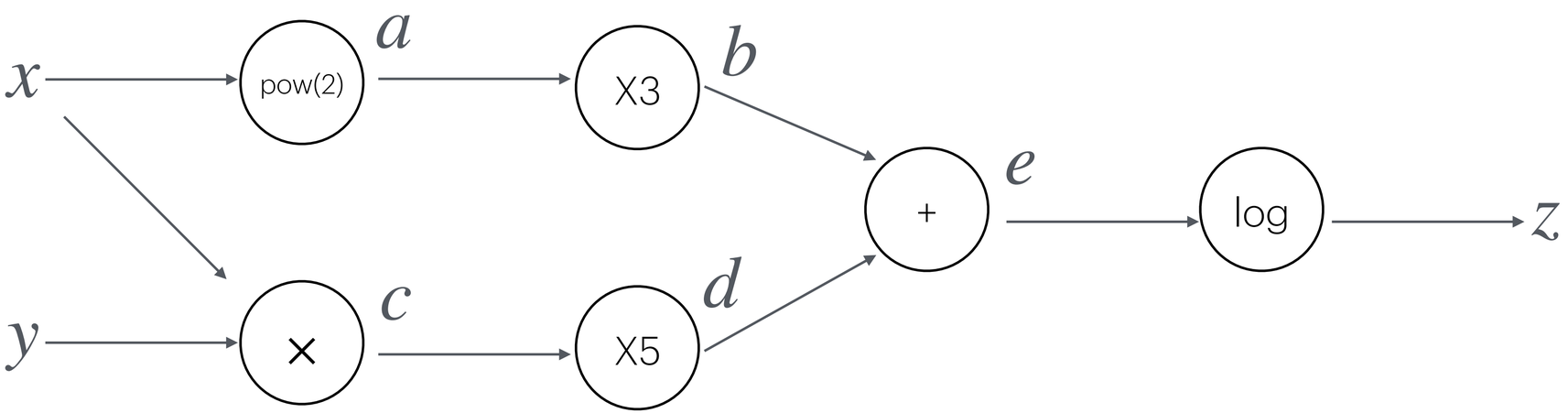
Computation graph

Apply chain rule

$$z = \log(3x^2 + 5xy)$$

Denote each computation from the left to right of the graph: ->

$$a = x^2$$
$$b = 3a$$
$$c = xy$$
$$d = 5c$$
$$e = b + d$$
$$z = log(e)$$



Ref. : link

Computation graph

Apply chain rule

$$z = \log(3x^2 + 5xy)$$

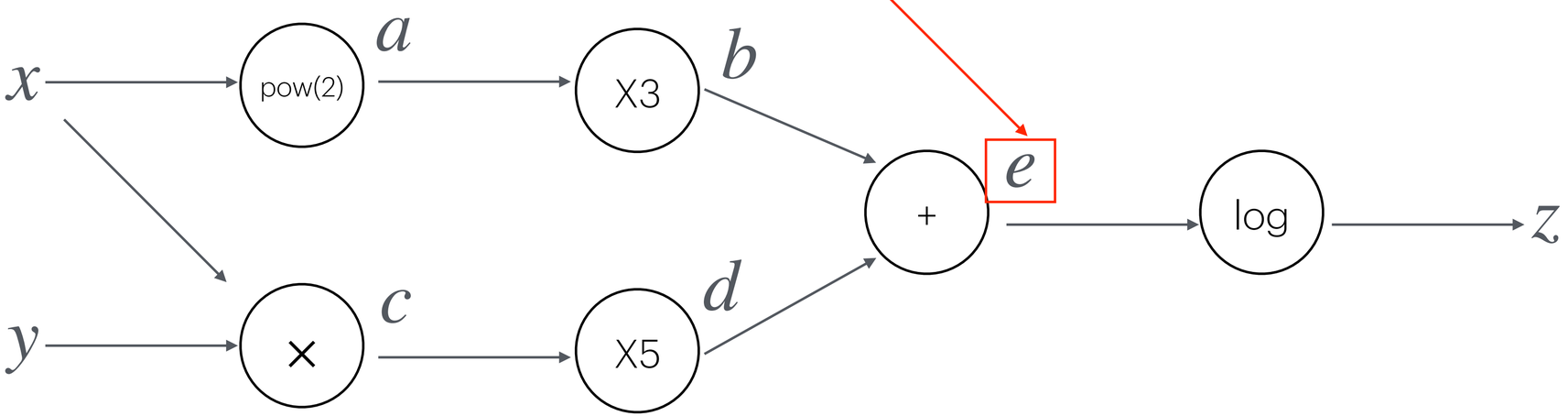
Denote each computation from the left to right of the graph: ->

$$a = x^2$$
$$b = 3a$$
$$c = xy$$
$$d = 5c$$
$$e = b + d$$
$$z = log(e)$$

Compute partial derivative from the right to left of the graph using chain rule: <-

$$\frac{\partial z}{\partial e} = \frac{1}{e}$$
$$\frac{\partial z}{\partial b} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial b} = \frac{\partial z}{\partial e}$$
$$\frac{\partial z}{\partial d} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d} = \frac{\partial z}{\partial e}$$
$$\frac{\partial z}{\partial a} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial b}\frac{\partial b}{\partial a} = \frac{\partial z}{\partial b} \cdot 3$$
$$\frac{\partial z}{\partial c} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c} = \frac{\partial z}{\partial d} \cdot 5$$
$$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial y} = \frac{\partial z}{\partial c} \cdot x$$
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial z}{\partial c}\frac{\partial c}{\partial x}$$
$$= \frac{\partial z}{\partial a}2x + \frac{\partial z}{\partial c}y$$

Note this is not exponential, just variable "e"



Ref. : link

Computation graph

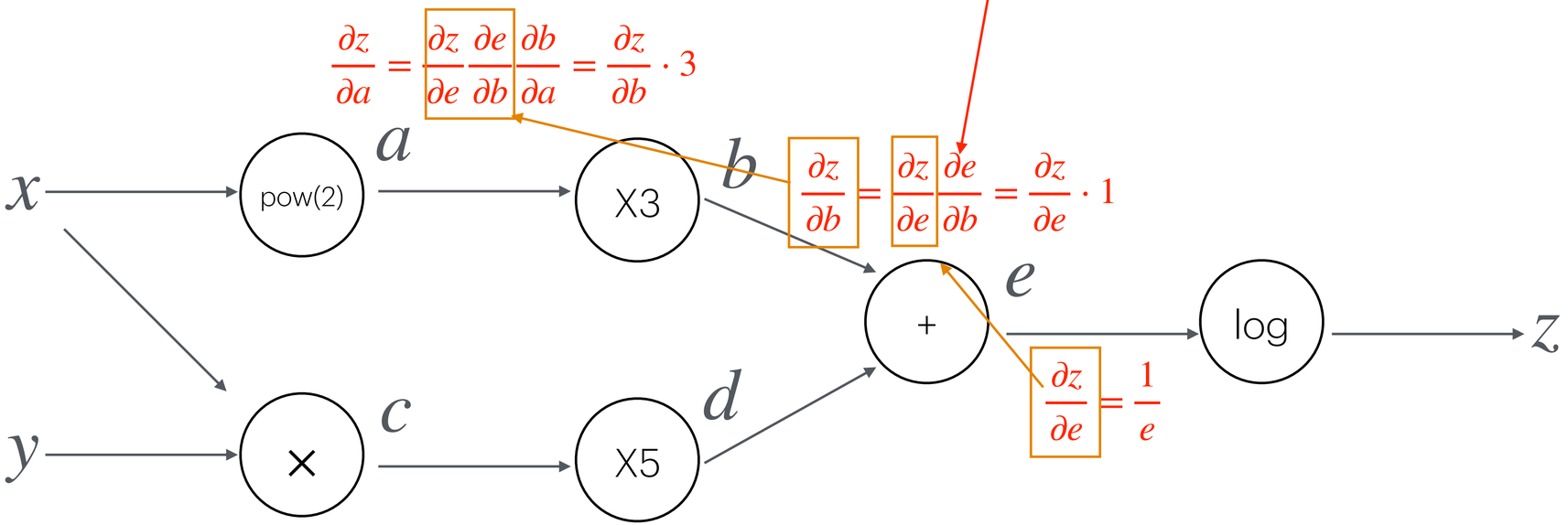
Apply chain rule

$$z = \log(3x^2 + 5xy)$$

Denote each computation from the left to right of the graph: ->

$$a = x^2$$
$$b = 3a$$
$$c = xy$$
$$d = 5c$$
$$e = b + d$$
$$z = log(e)$$

Compute partial derivative from the right to left of the graph using chain rule: <-



$$\frac{\partial z}{\partial e} = \frac{1}{e}$$

$$\frac{\partial z}{\partial b} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial b} = \frac{\partial z}{\partial e}$$

$$\frac{\partial z}{\partial d} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d} = \frac{\partial z}{\partial e}$$

$$\frac{\partial z}{\partial a} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial b}\frac{\partial b}{\partial a} = \frac{\partial z}{\partial b} \cdot 3$$

$$\frac{\partial z}{\partial c} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c} = \frac{\partial z}{\partial d} \cdot 5$$

$$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial y} = \frac{\partial z}{\partial c} \cdot x$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial z}{\partial c}\frac{\partial c}{\partial x}$$

$$= \frac{\partial z}{\partial a}2x + \frac{\partial z}{\partial c}y$$

Ref. : link

Computation graph

Use numerical example

$$z = \log(3x^2 + 5xy)$$

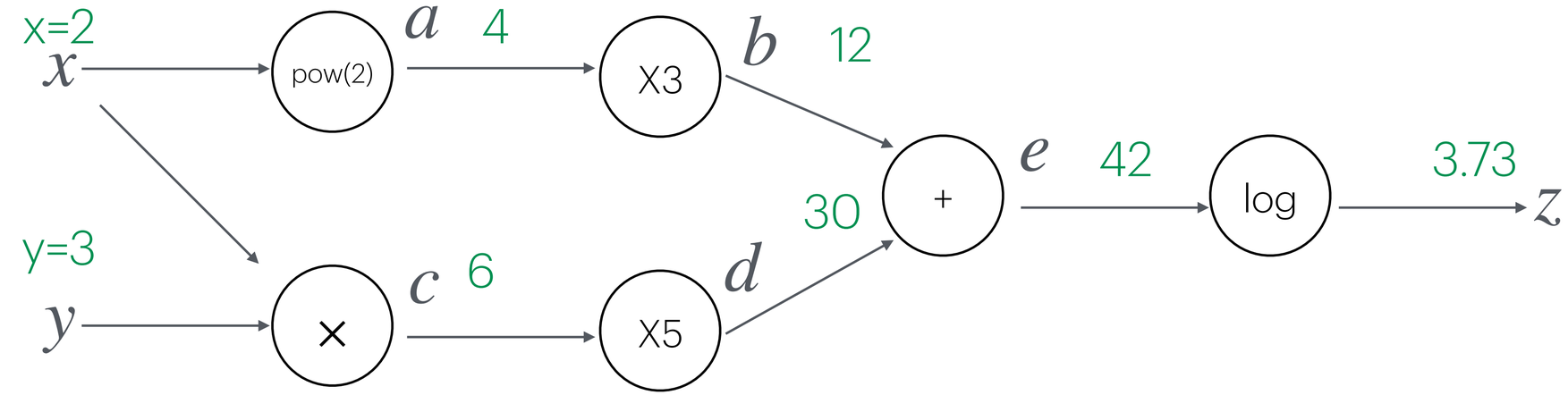
Denote each computation from the left to right of the graph: ->

$$a = x^2$$
$$b = 3a$$
$$c = xy$$
$$d = 5c$$
$$e = b + d$$
$$z = log(e)$$

Compute partial derivative from the right to left of the graph using chain rule: <-

$$\frac{\partial z}{\partial e} = \frac{1}{e}$$
$$\frac{\partial z}{\partial b} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial b} = \frac{\partial z}{\partial e}$$
$$\frac{\partial z}{\partial d} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d} = \frac{\partial z}{\partial e}$$
$$\frac{\partial z}{\partial a} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial b}\frac{\partial b}{\partial a} = \frac{\partial z}{\partial b} \cdot 3$$
$$\frac{\partial z}{\partial c} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c} = \frac{\partial z}{\partial d} \cdot 5$$
$$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial y} = \frac{\partial z}{\partial c} \cdot x$$
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial z}{\partial c}\frac{\partial c}{\partial x}$$
$$= \frac{\partial z}{\partial a}2x + \frac{\partial z}{\partial c}y$$

Use numerical example->



Ref. : link

Computation graph

Compute derivative

$$z = \log(3x^2 + 5xy)$$

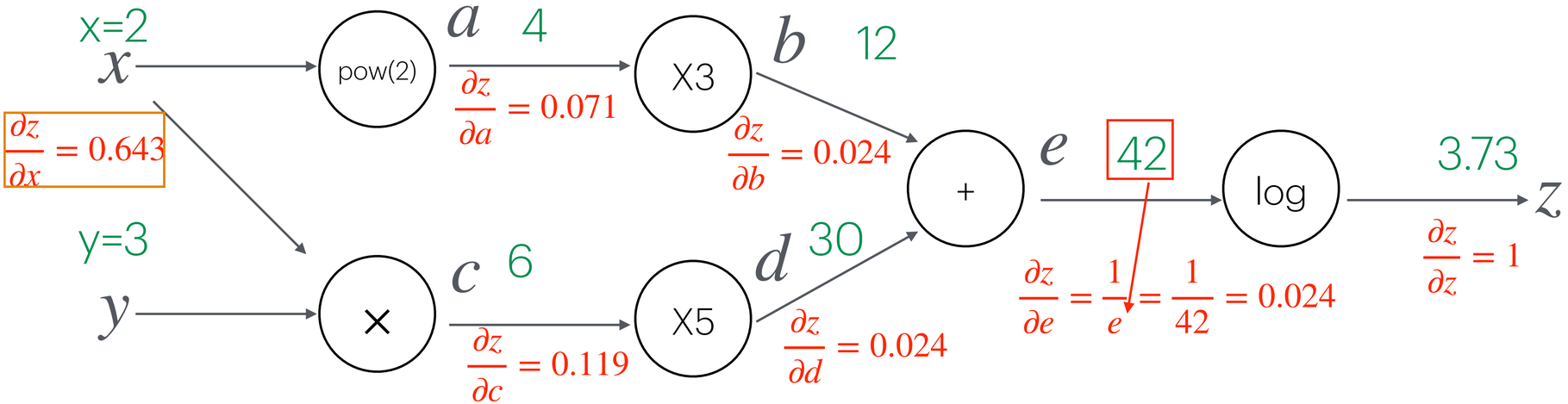
Denote each computation from the left to right of the graph: ->

$$a = x^2$$
$$b = 3a$$
$$c = xy$$
$$d = 5c$$
$$e = b + d$$
$$z = log(e)$$

Compute partial derivative from the right to left of the graph using chain rule: <-

$$\frac{\partial z}{\partial e} = \frac{1}{e}$$
$$\frac{\partial z}{\partial b} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial b} = \frac{\partial z}{\partial e}$$
$$\frac{\partial z}{\partial d} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d} = \frac{\partial z}{\partial e}$$
$$\frac{\partial z}{\partial a} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial b}\frac{\partial b}{\partial a} = \frac{\partial z}{\partial b} \cdot 3$$
$$\frac{\partial z}{\partial c} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c} = \frac{\partial z}{\partial d} \cdot 5$$
$$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial y} = \frac{\partial z}{\partial c} \cdot x$$
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial z}{\partial c}\frac{\partial c}{\partial x}$$
$$= \frac{\partial z}{\partial a}2x + \frac{\partial z}{\partial c}y$$

Use numerical example->



<— Numerical example for derivatives

Ref. : link

Computation graph

Pytorch example

$$z = \log(3x^2 + 5xy)$$

Denote each computation from the left to right of the graph: ->

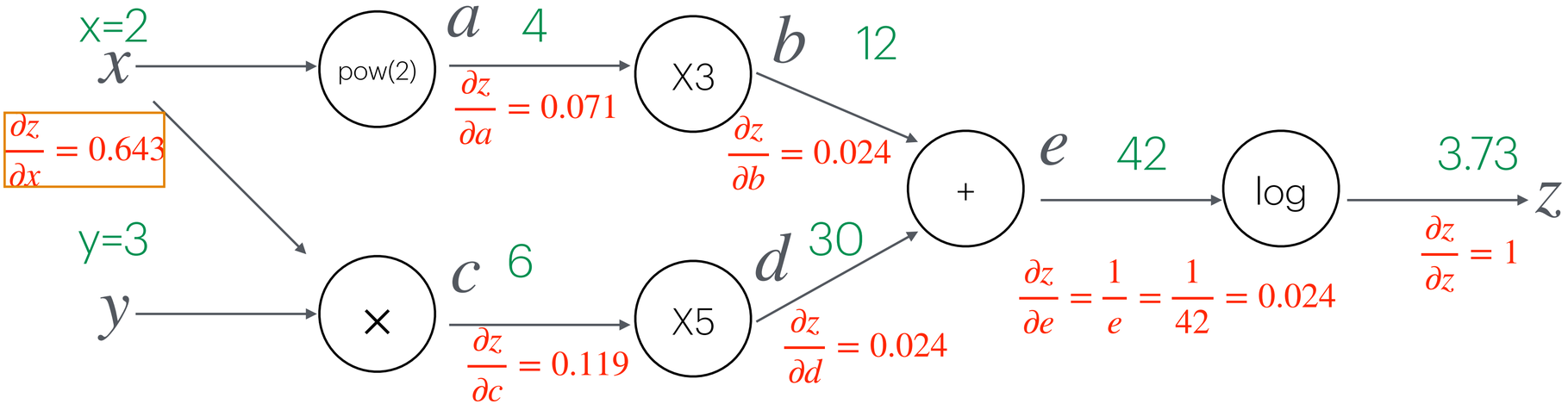
$$a = x^2$$
$$b = 3a$$
$$c = xy$$
$$d = 5c$$
$$e = b + d$$
$$z = log(e)$$

PyTorch code example

```
import torch
x = torch.tensor([2.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)
z = torch.log(3*x**2 + 5*x*y)
print("z = ", z)
z.backward()
print("x.grad = ", x.grad)
print("y.grad = ", y.grad)

# === output ====
z =  tensor([3.7377], grad_fn=<LogBackward0>)
x.grad =  tensor([0.6429])
y.grad =  tensor([0.2381])
```

Use numerical example->

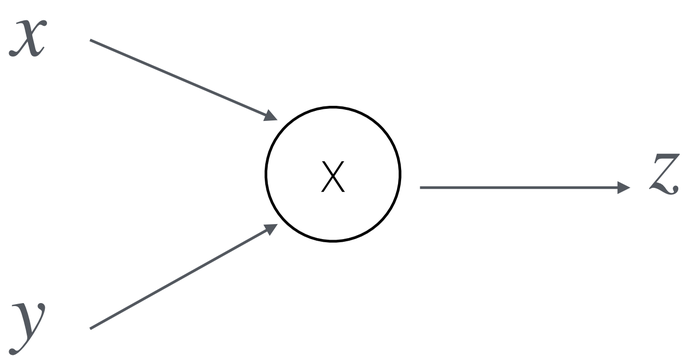


<-Numerical example for derivatives

Ref. : link

DL framework provides backward operators

DL framework provides operators with defined backward() operations.




```
class Multiply(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, y):
        ctx.save_for_backward(x, y)
        return x * y

    @staticmethod
    def backward(ctx, grad_z):
        x, y = ctx.saved_tensors
        grad_x = y * grad_z # dz/dx * dL/dz
        grad_y = x * grad_z # dz/dy * dL/dz
        return grad_x, grad_y
```

Ref.[8]

DL framework forward and backward

How DL framework runs a model:

- When computation graph is created then run forward in topologically sorted order for inputs, then run reverse topologically sorted order to compute derivatives for backward propagation

```
class ComputationalGraph(object):

    def forward(inputs):
        for node in self.graph.nodes_topologically_sorted():
            node.forward()
        return loss
```

Ref.[8]

DL framework forward and backward

When computation graph is created then run forward in topologically sorted order for inputs, then run reverse topologically sorted order to compute derivatives for backward propagation

```
class ComputationalGraph(object):

    def forward(inputs):
        for node in self.graph.nodes_topologically_sorted():
            node.forward()
        return loss

    def backward():
        for node in reversed(self.graph.nodes_topologically_sorted()):
            node.backward()
        return inputs_gradients
```

Ref.[8]

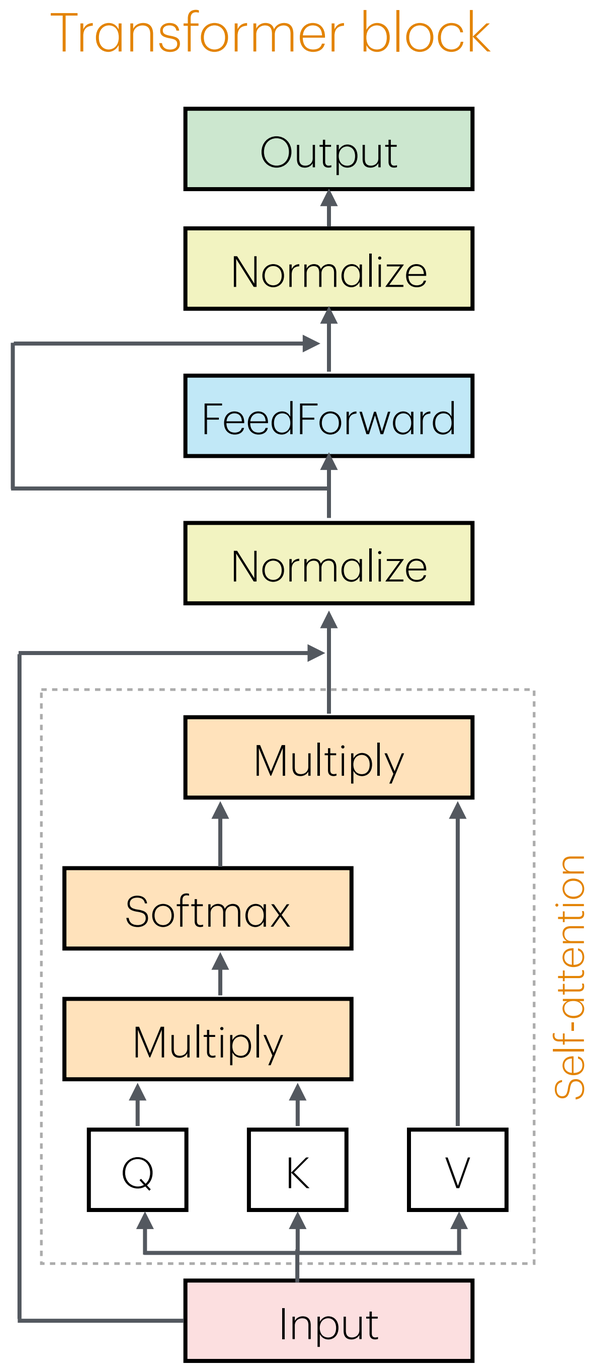
Code for llama3 Transformer block

Let's check the code for llama3 written in PyTorch before looking into software framework for deep learning

```
class TransformerBlock(nn.Module):
    def __init__(self, layer_id: int, args: ModelArgs):
        super().__init__()
        self.n_heads = args.n_heads
        self.dim = args.dim
        self.head_dim = args.dim // args.n_heads
        self.attention = Attention(args)
        self.feed_forward = FeedForward(...)
        self.layer_id = layer_id
        self.attention_norm = RMSNorm(args.dim, eps=args.norm_eps)
        self.ffn_norm = RMSNorm(args.dim, eps=args.norm_eps)

    def forward(self, x: torch.Tensor, start_pos: int,
        freqs_cis: torch.Tensor, mask: Optional[torch.Tensor],
    ):
        h = x + self.attention(self.attention_norm(x), start_pos, freqs_cis, mask)
        out = h + self.feed_forward(self.ffn_norm(h))
        return out
```

Code from link



Note: llama place layer normalization before input, pre-norm, the figure shown was from the original transformer and uses post-norm.

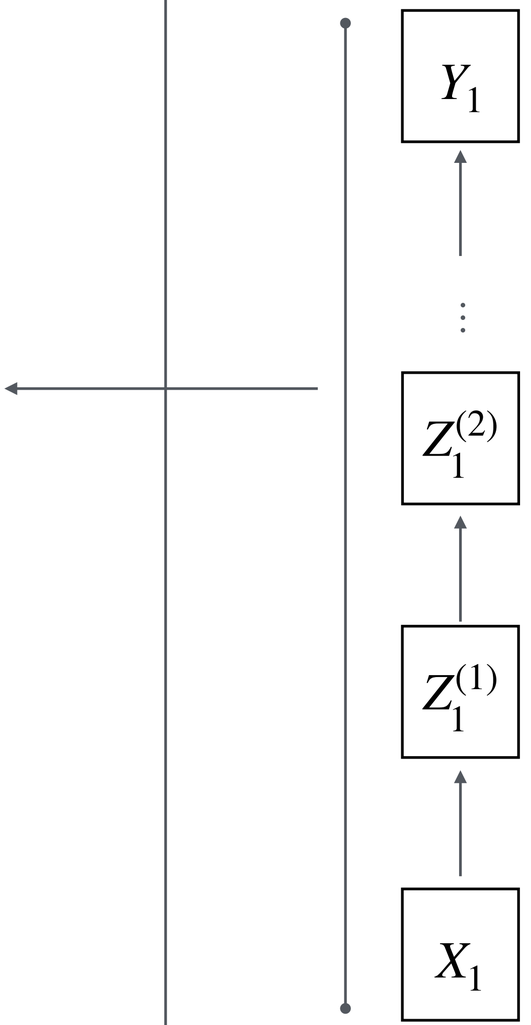
Code for Transformer (llama3) model

Let's check the code for llama3 written in PyTorch before looking into software framework for deep learning.

Let $Z_t^{(i+1)}$ as Transformer block,
$Z_t^{(i+1)} = \text{Transformer\_block}(Z_t^{(i)})$.
Llama3 model is consists of layers of Transformer blocks.

```
class Transformer(nn.Module):
    def __init__(self, params: ModelArgs):
        super().__init__()
        self.params = params
        self.vocab_size = params.vocab_size
        self.n_layers = params.n_layers
        self.tok_embeddings = VocabParallelEmbedding(...)
        self.layers = torch.nn.ModuleList()
        for layer_id in range(params.n_layers):
            self.layers.append(TransformerBlock(layer_id, params))
        self.norm = RMSNorm(params.dim, eps=params.norm_eps)
        self.output = ColumnParallelLinear(...)
        self.freqs_cis = precompute_freqs_cis(...)

    def forward(self, tokens: torch.Tensor, start_pos: int):
        _bsz, seqlen = tokens.shape
        h = self.tok_embeddings(tokens)
        self.freqs_cis = self.freqs_cis.to(h.device)
        freqs_cis = self.freqs_cis[start_pos : start_pos + seqlen]
        ...
        for layer in self.layers:
            h = layer(h, start_pos, freqs_cis, mask)
        h = self.norm(h)
        output = self.output(h).float()
        return output
```



Code from link

Summary: Key technical concepts

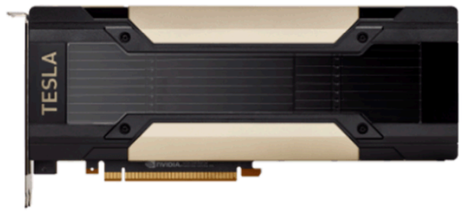
1. Deep learning framework supports:
    - auto differentiation,
    - hardware acceleration,
    - distributed training support
2. Deep learning framework: creates computation graph from model (neural network).
3. Auto differentiation is based on chain rule from calculus.

Deep learning HW

Deep learning hardware

There are different types of hardware chips for different devices and applications.
Now including PC as well.

In the following slide, we will cover GPU for its hardware and programming model

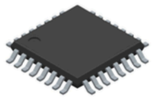


Cloud AI

GPUs/TPUs

Mobile AI

Smartphones



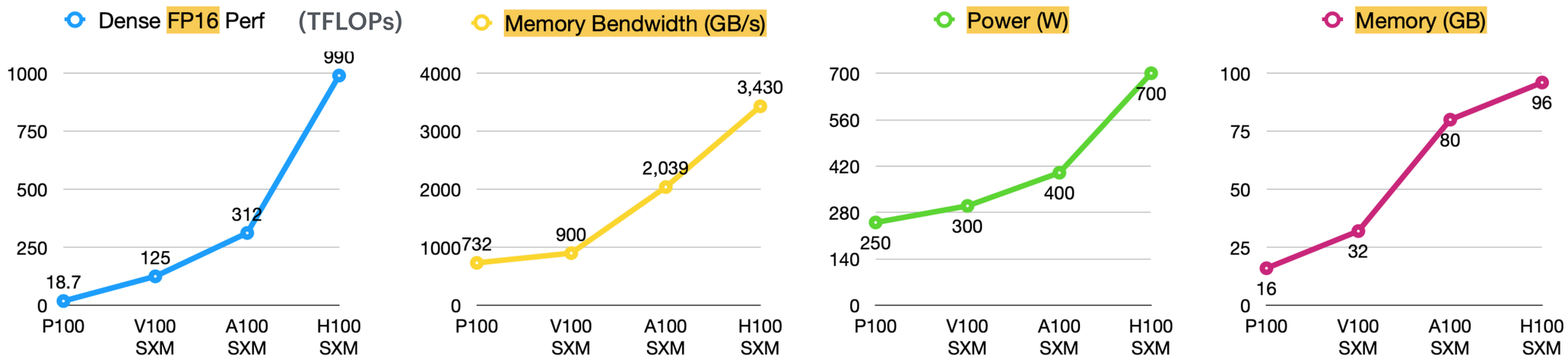
Tiny AI

IoT/Microcontrollers

Image source: [1]

Image source: link

GPUs over recent years



P100 (2016)
V100 (2017)
A100 (2020)
H100 (2022)

Image source: [1]

CPU vs. GPU

CPU: a small number of complex cores, the clock speed of each core is high, good for sequential tasks

GPU: a large number of small cores, the clock speed of each core his low, good for parallel tasks

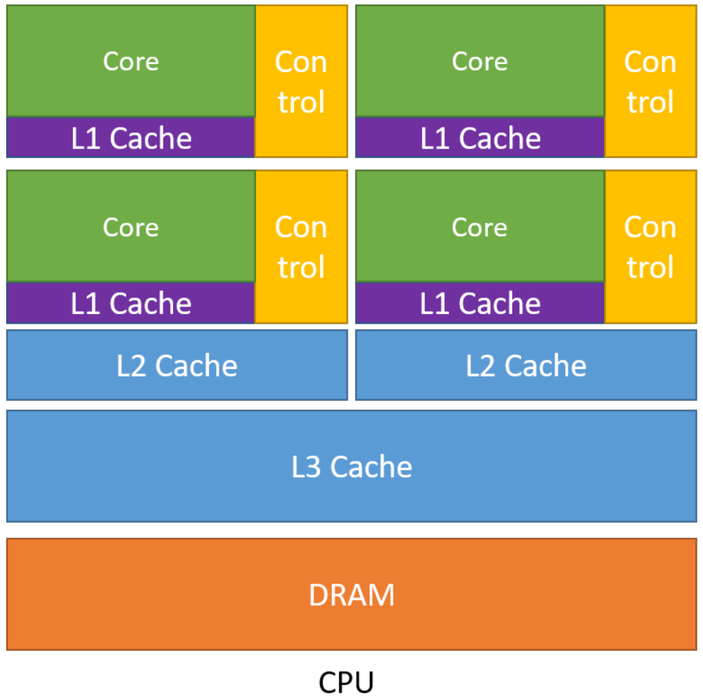
	Core	Clock speed	Memory	Price	Speed
CPU (intel core i7-7700)	4	4.2GHz	DDR4	$385	~540 GFLOPS FP32
GPU (nvidia V100)	5120	1.245 GHz	HBM2 16 GB	$9,478	12.7 TFLOPS FP32

Ref: [9]

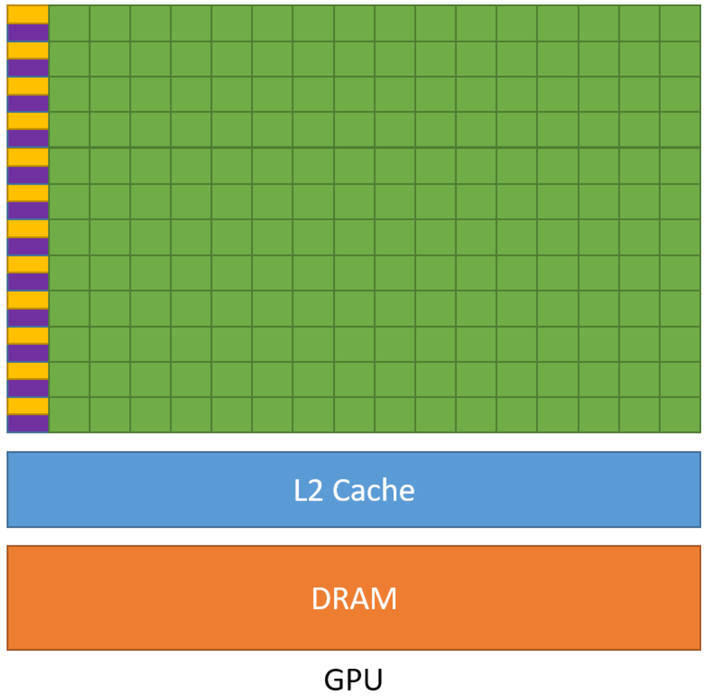
CPU vs. GPU

CPU is designed to excel at executing a sequence of operations, called a thread, as fast as possible and can execute a few tens of these threads in parallel.

GPU is designed to execute a few thousands of *threads* in parallel to achieve large throughput. With these threads, GPU can hide memory access latency with computation.



CPU



GPU

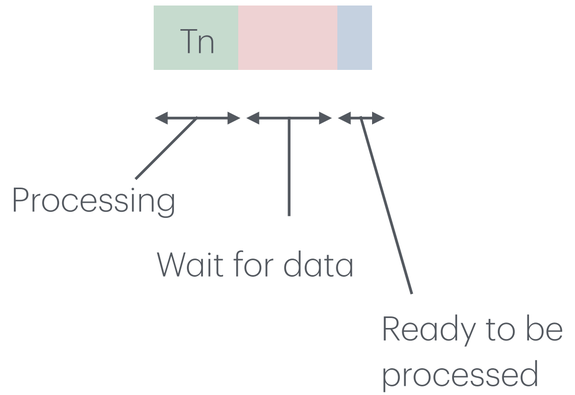
Ref: [5] Image source: [3]

Low latency vs High throughput

CPU minimizes latency within each thread

GPU *hides latency* with computation (several 100K+ threads)

CPU : low latency processor



Ref: link

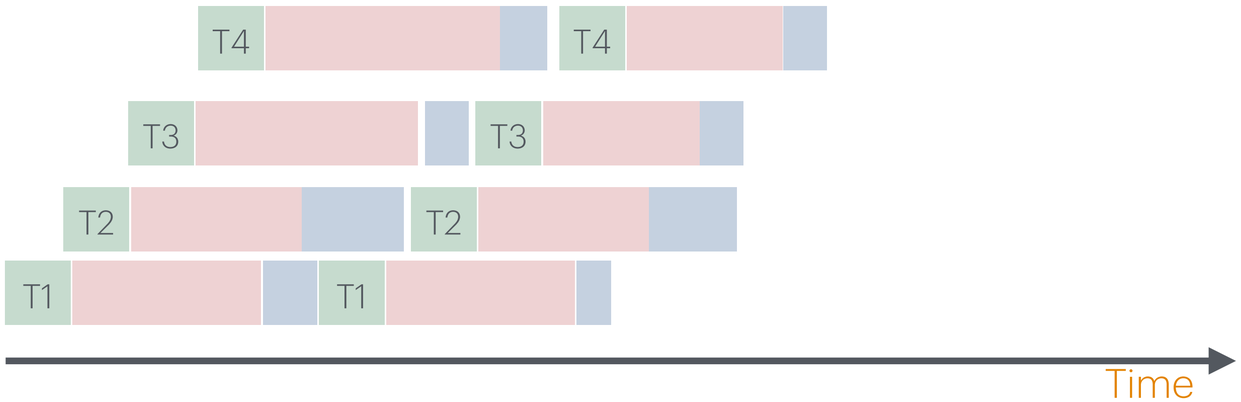
Low latency vs High throughput

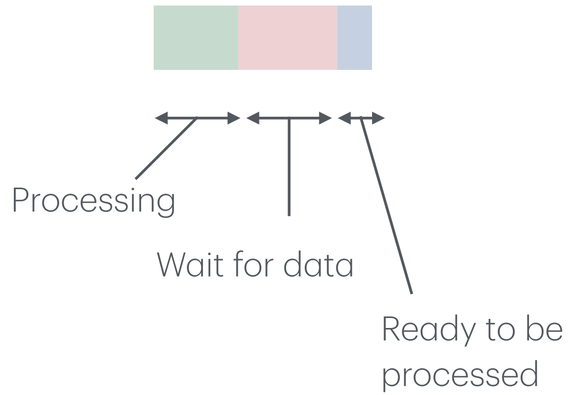
CPU minimizes latency within each thread

GPU *hides latency* with computation (several 100K+ threads)

CPU : low latency processor

GPU : High throughput processor

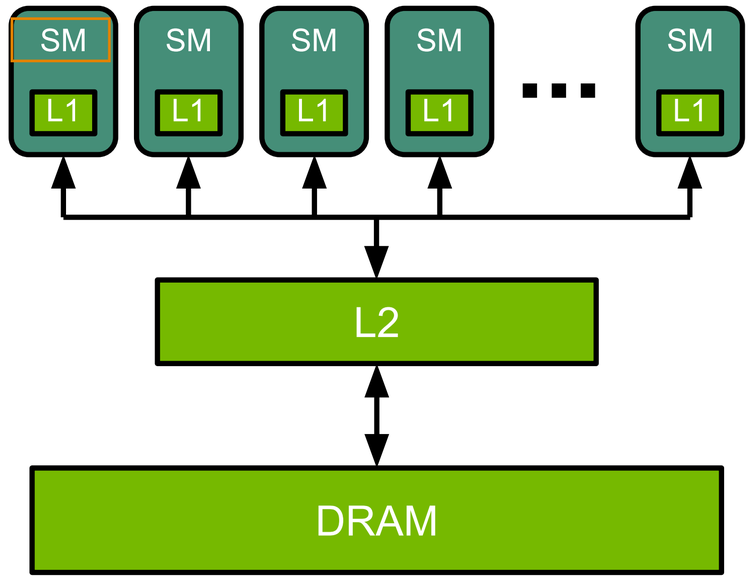




Ref: link

GPU terminology

- GPU consists of streaming multiple processors (*SMs*)
- SMs: unit capable of executing a thread blocks of a kernel
- Warp: group of 32 threads runs the same stream of instruction together on different data
- Kernel: software function that runs on the device, kernel may be divided into thread blocks. It specified by using **__global__**



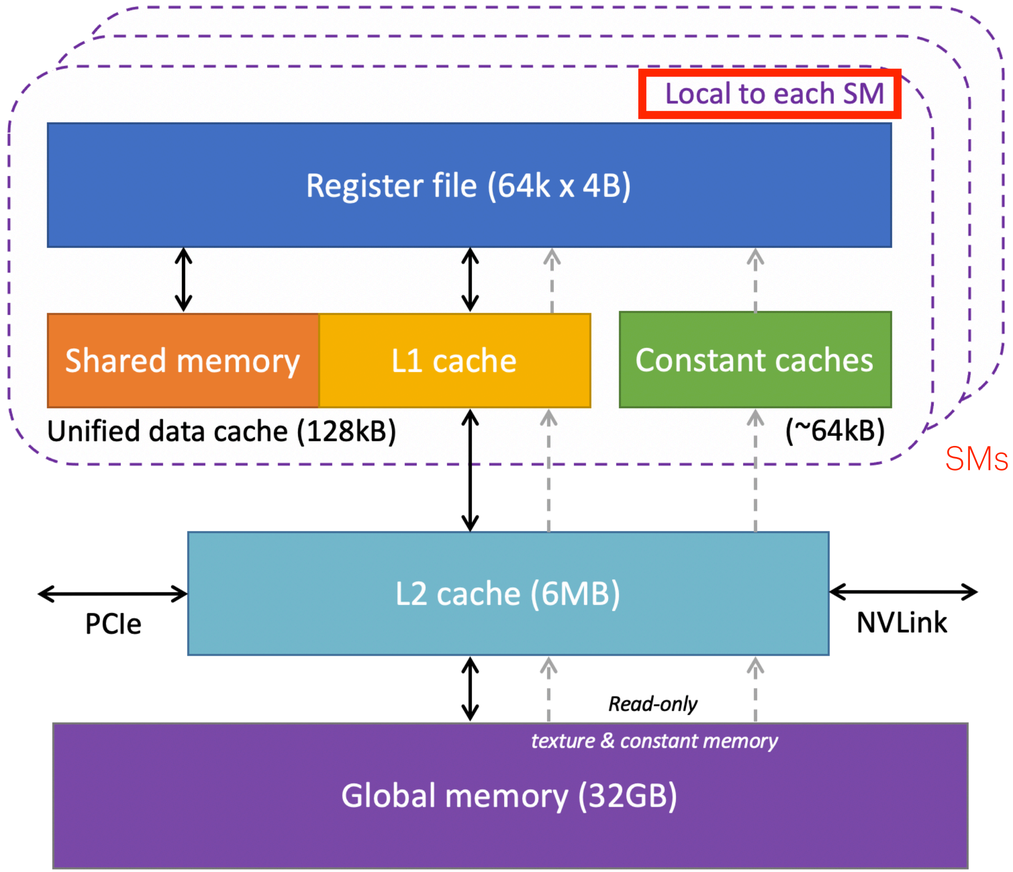
Ref. [10]. Image source: link

Memory hierarchy

Register file: memory (execution contexts) for threads

Shared memory: same physical memory as L1 cache, but all its data may be accessed by any thread in a thread block. It used for communication and share data between threads. It is controlled by programmers

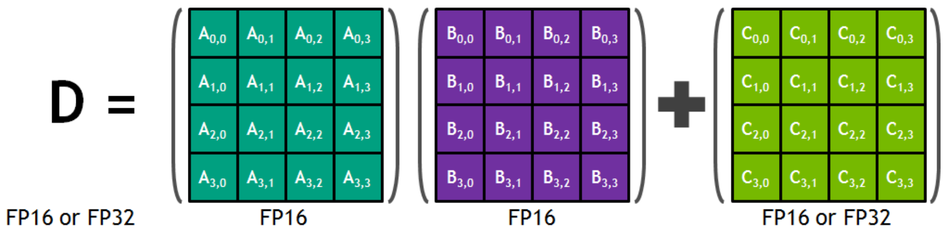
Global memory: main memory on device. For V100, HBM2 is used.



Ref. & Image from: [10]

TensorCore

$D = A \times B + C$ FMA (fused multiply and add)



The main operations in machine learning models are matrix multiplications and additions and parallel threads in GPU or dedicated mat mul block (TensorCore) are good at this operations.

TensorCore: dedicated unit for matrix multiplication.

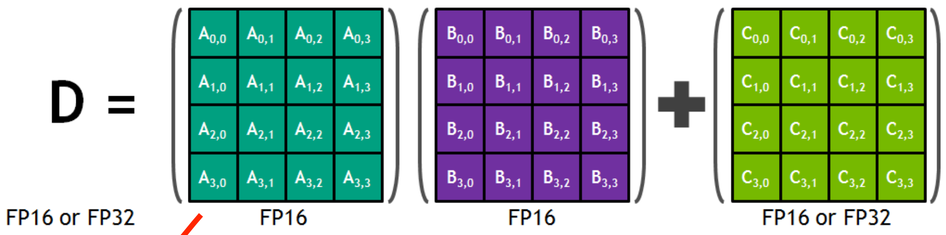
- A single TensorCore perform 64 FP16 *FMA (fused multiply and add)* per clock cycle
- 8 TensorCore per SM
- A single SM can perform 512 FMA (1024 FLOPs) per clock cycle

Figure on the right depicts 4x4 matmul and add with TensorCore.

Ref. and image from link

TensorCore

$D = A \times B + C$ FMA (fused multiply and add)

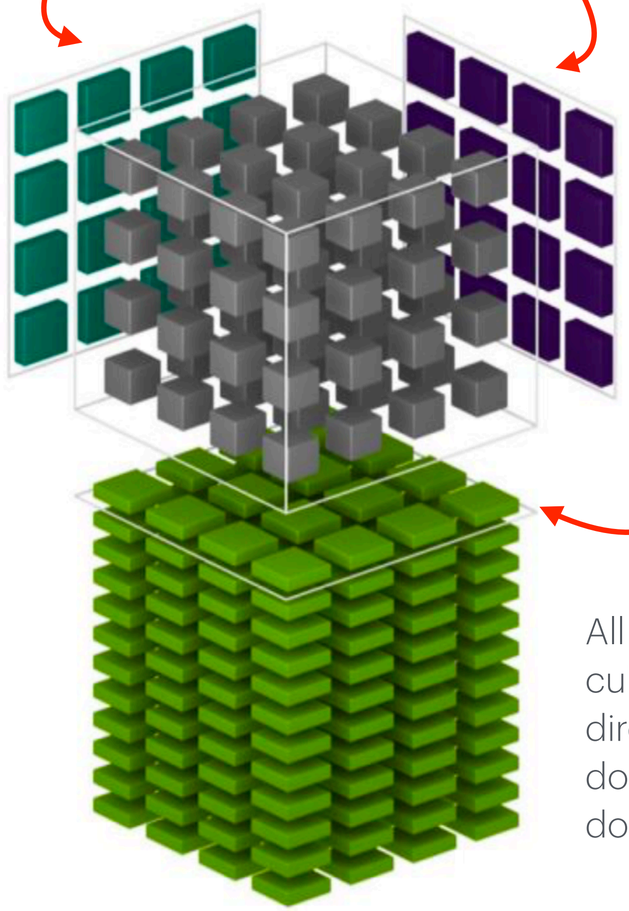


The main operations in machine learning models are matrix multiplications and additions and parallel threads in GPU or dedicated mat mul block (TensorCore) are good at this operations.

TensorCore: dedicated unit for matrix multiplication.

- A single TensorCore perform 64 FP16 FMA (fused multiply and add) per clock cycle
- 8 TensorCore per SM.
- A single SM can perform 512 FMA (1024 FLOPs) per clock cycle

Figure on the right depicts 4x4 matmul and add with TensorCore.



All 64 multiplications (gray cube) and additions in vertical directions of the gray cube are done. Then final additions are done with green cubes.

Ref. and image from link

SM Sub-core

V100 GPU

A SM (Streaming multiprocessor) consists of 4 sub-cores

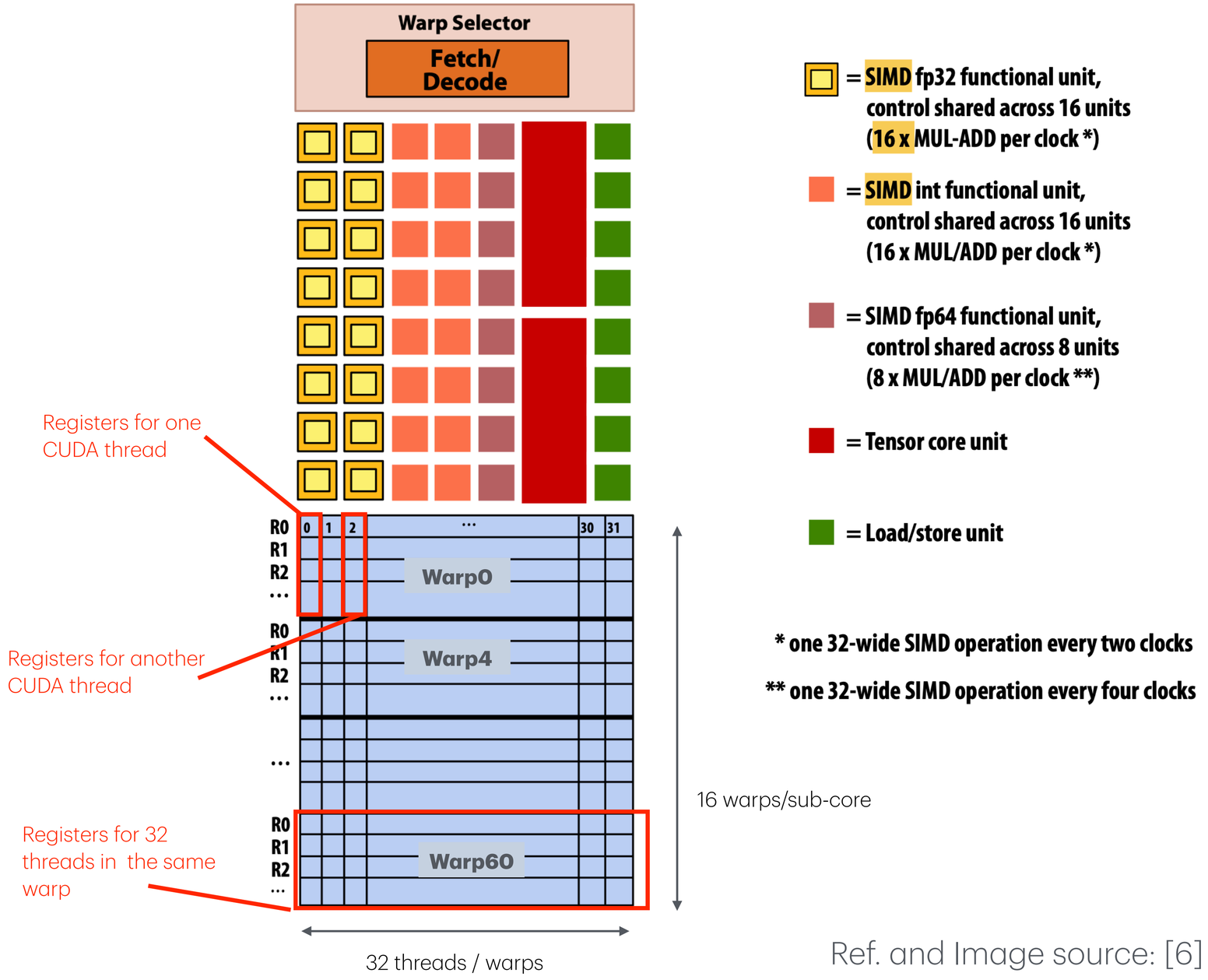
Each sub-core includes 16 warps. Each sub-core can schedule and interleave up to 16 warps

Warp: a group of 32 threads. So thread from 0-31 of a thread block map to the same warp.

For example a thread block with 256 threads is mapped to 8 warps, 256 (threads) / 32 (threads/warp)

Note: the number of threads per warp and warps per SM sub-core can be changed in the future GPU architecture.

SM sub-core: 4 sub-cores / 1 SM



Ref. and Image source: [6]

SM (streaming multiprocessor)

V100 GPU

SM consists of 4 SM sub-cores, 64 warps, and 2048 threads.

SM operation each clock:

- Each sub-core selects one runnable warp out of 16 warps
- Each sub-core runs next instruction for the CUDA threads in the warps

Threads in *a thread block* are schedule on the same SM for fast communication through shared memory.

Ref. and Image source: [6]

Geometry of GPU

V100 GPU

V100 GPU has 80 SMs per chip

1.245 GHz clock

12.7 TFLOPs FP32 (5,120 * 1.245 G * 2 FLOPs, mul and add as 2 FLOPs)

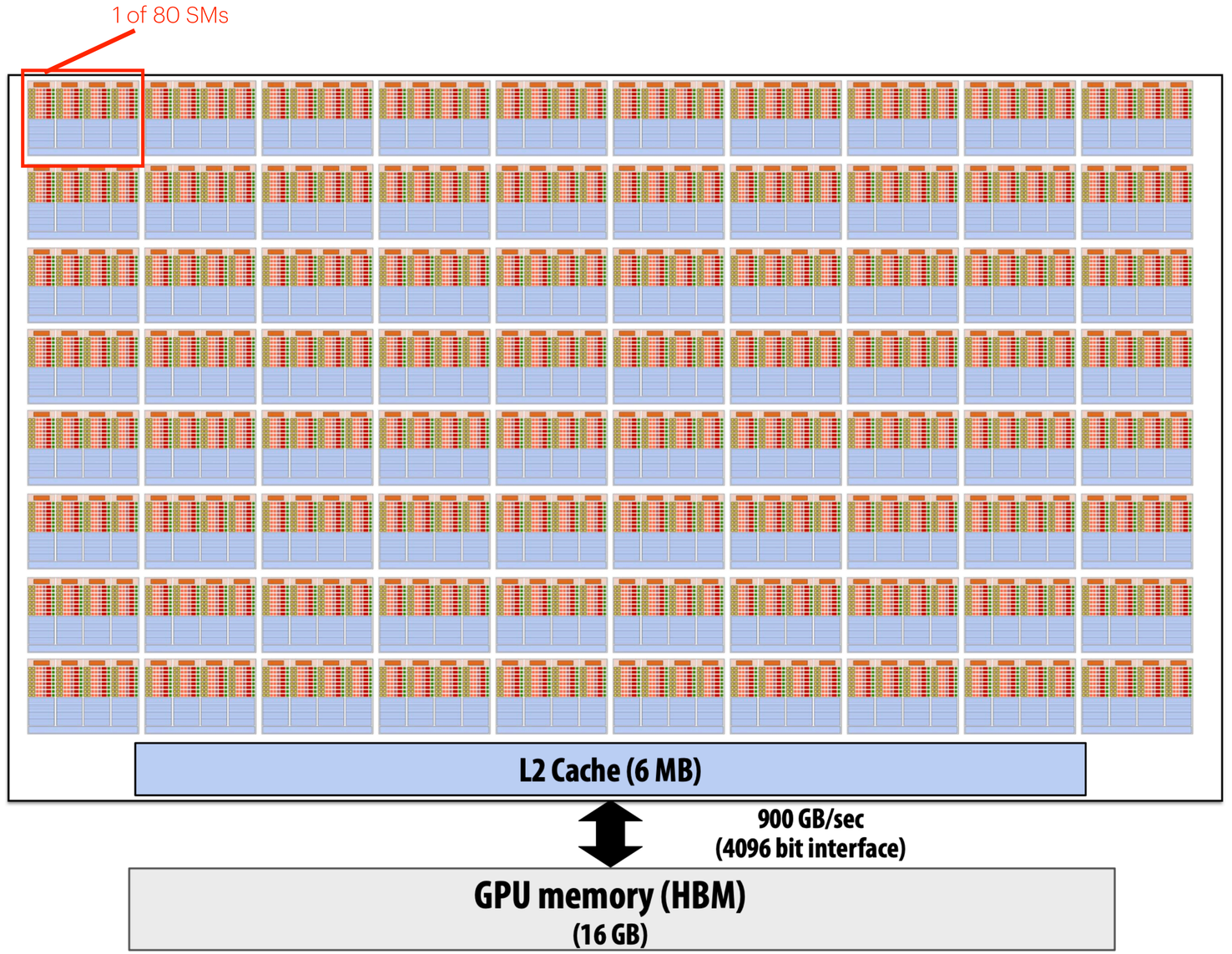
- 80x4x16 = 5,120 FP32 mul-add ALUs (80 SM, 4 sub-cores/SM, 16 ALUs/sub-core)

102 TFLOPs FP16 w/ TensorCore (640 * 64 ops/clock * 2 * 1.245 GHz)

- 640 TensorCores (80 SMs * 8 TensorCore/SM)

Up to 80 x 64 = 5,120 *interleaved* warps per chip (80 SM, 64 warps/SM)

80x64x32 = 163,840 threads/chip (80 SM, 64 warps/SM, 32 threads/SM)



Ref. and Image source: [6]

GPU programming abstraction

Key abstractions of CUDA programming model:

- Hierarchy of thread groups
- Shared memories
- Barrier synchronization

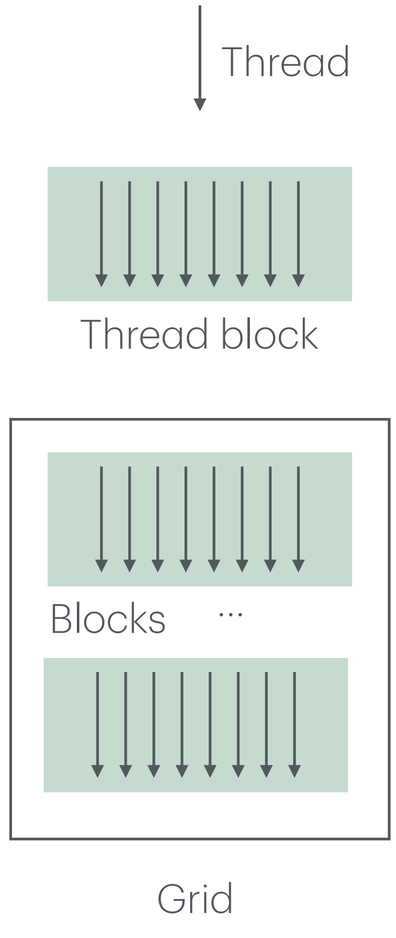
All threads runs the same code

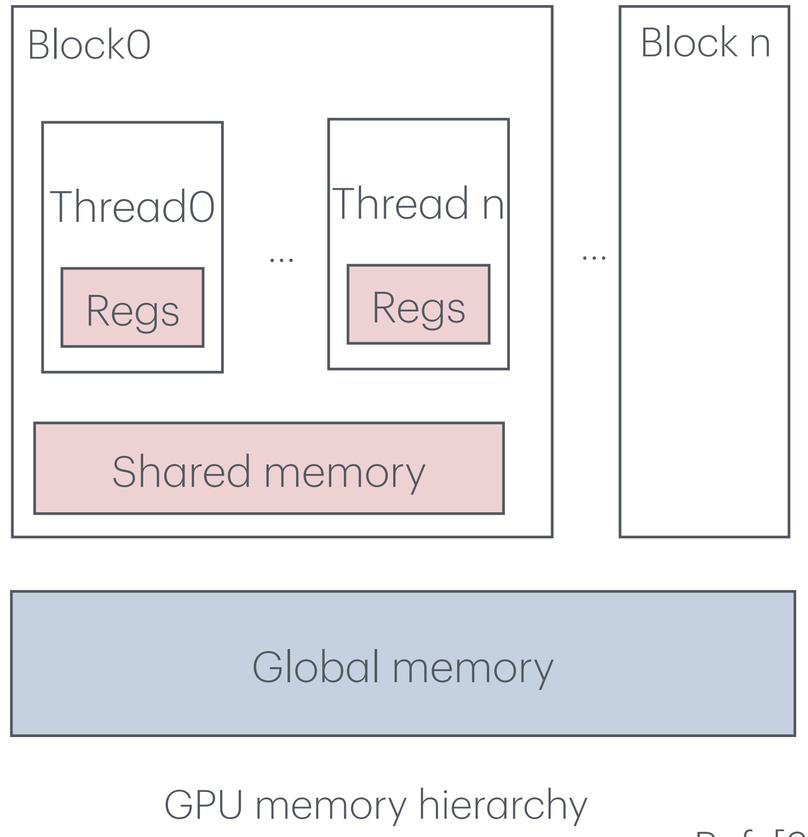
Thread block: group of threads.

- A thread block can have 1024 threads at most

Thread blocks are grouped into a launch grid

Thread within the same thread block have shared memory



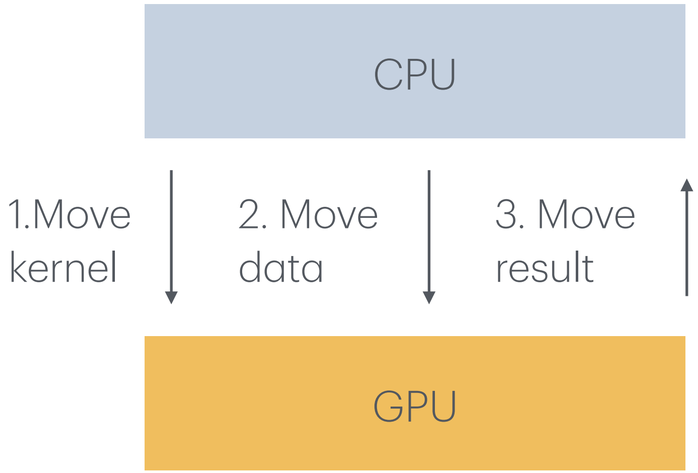


GPU memory hierarchy

Ref: [3]

GPU programming

1. CPU offload GPU kernels that will be run on multiple threads on the GPU
   - Kernel: program written in C/C++ for thread
2. CPU transfer data to GPU memory
3. CPU need to bring back data from GPU to CPU memory



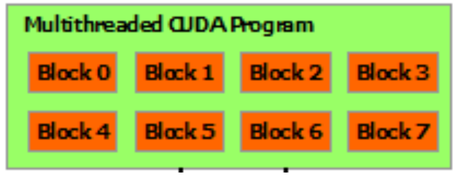
Thread scheduling

A program is partitioned into thread block. Each block of threads can be scheduled on any SM (streaming multiprocessor) in any order independently. No dependencies between blocks.

There is no concept of **num_cores**, threads in the code shown you.

A GPU with more SMs executes the program in less time.

A program can run on different GPUs without modification.



Ref: [6] Image source: [3]

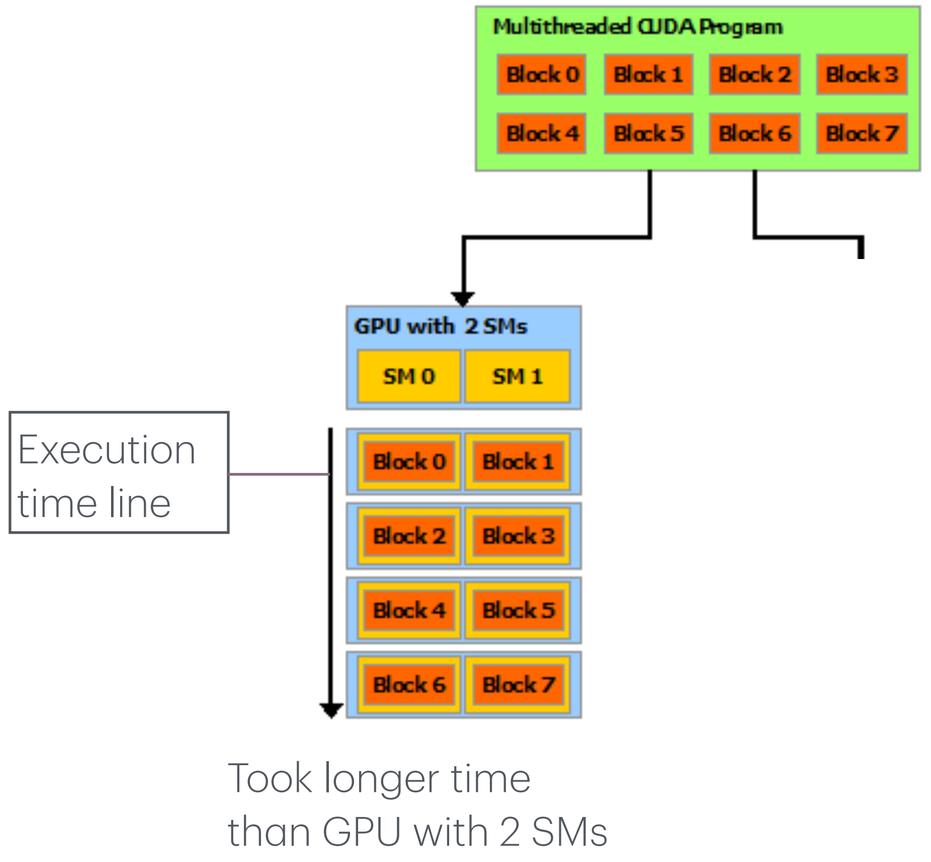
Thread scheduling

A program is partitioned into thread block. Each block of threads can be scheduled on any SM (streaming multiprocessor) in any order independently. No dependencies between blocks.

There is no concept of **num_cores**, threads in the code shown you.

A GPU with more SMs executes the program in less time.

A program can run on different GPUs without modification.



Took longer time than GPU with 2 SMs

Ref: [6] Image source: [3]

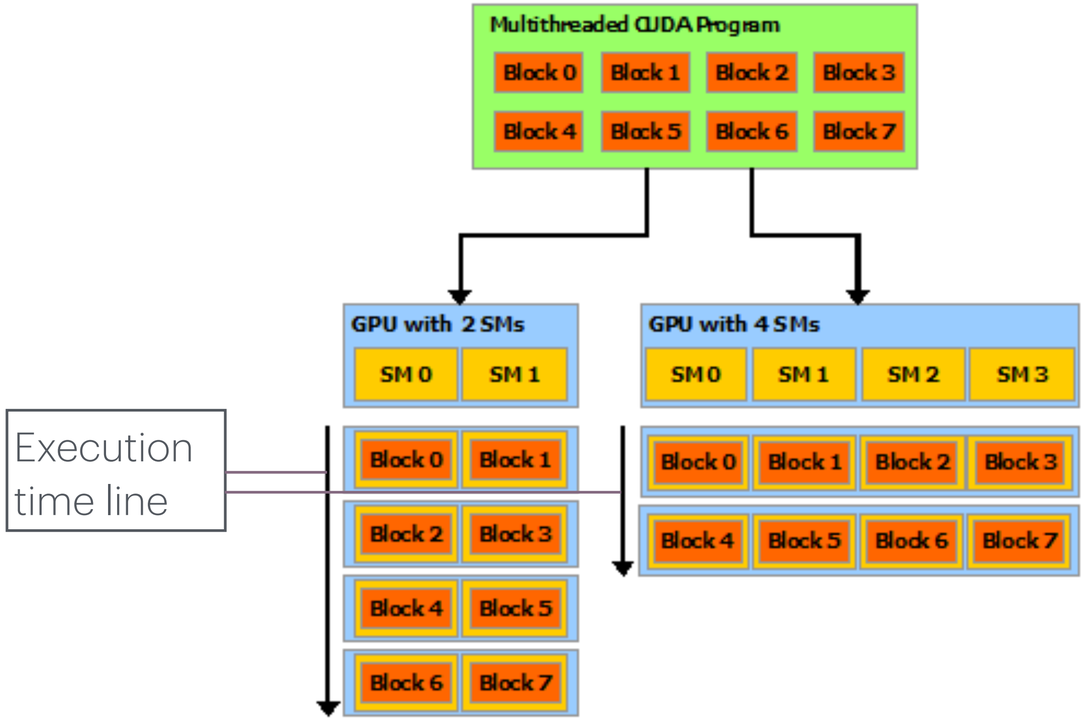
Thread scheduling

A program is partitioned into thread block. Each block of threads can be scheduled on any SM (streaming multiprocessor) in any order independently. No dependencies between blocks.

There is no concept of **num_cores**, threads in the code shown you.

A GPU with more SMs executes the program in less time.

A program can run on different GPUs without modification.



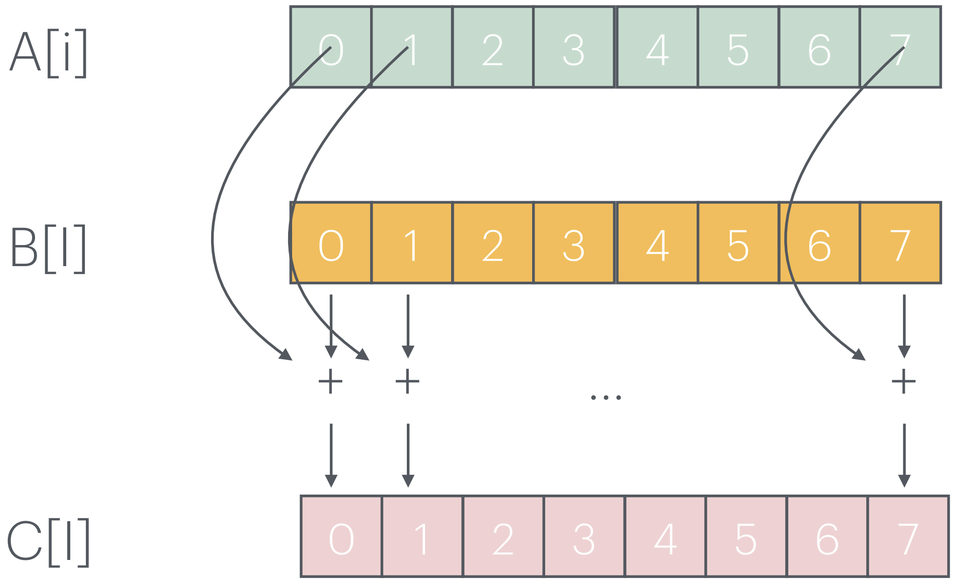
Took *longer* time than GPU with 2 SMs

Took *shorter* time than GPU with 4 SMs

Ref: [6] Image source: [3]

Example: vector addition in CUDA

Let's use an example of vector addition
to compare code for CPU and GPU




```
void VecAddCPU(float* A, float *B,
                        float* C, int n) {
  for (int i = 0; i < n; ++i) {
    C[i] = A[i] + B[i];
  }
}
```

Code for CPU

Ref: [3]

Example: vector addition in CUDA

Let's use an example of vector addition to compare code for CPU and GPU

For GPU, each thread do the addition

- Figure depicts a simple example of vector addition with dimension of n=8

```
void VecAddCPU(float* A, float *B,
                        float* C, int n) {
  for (int i = 0; i < n; ++i) {
    C[i] = A[i] + B[i];
  }
}
```

Code for CPU

n=8 (total vector dim, as example)
thread_per_block is 4,
blockDim.x = 4,
nblocks = 2

Ref: [3]

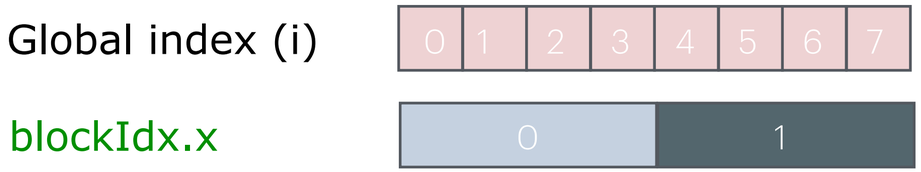
Example: vector addition in CUDA

Let's use an example of vector addition to compare code for CPU and GPU

For GPU, each thread do the addition

- Figure depicts a simple example of vector addition with dimension of n=8

n=8 (total vector dim, as example)
thread_per_block is 4,
blockDim.x = 4,
nblocks = 2




```
void VecAddCPU(float* A, float *B,
                        float* C, int n) {
  for (int i = 0; i < n; ++i) {
    C[i] = A[i] + B[i];
  }
}
```

Code for CPU

Ref: [3]

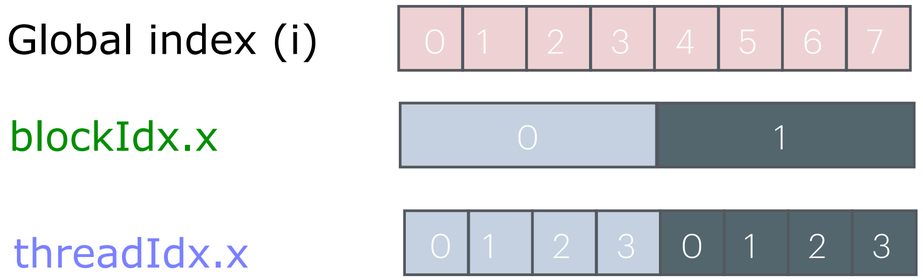
Example: vector addition in CUDA

Let's use an example of vector addition to compare code for CPU and GPU

For GPU, each thread do the addition

- Figure depicts a simple example of vector addition with dimension of n=8

n=8 (total vector dim, as example)
thread_per_block is 4,
blockDim.x = 4,
nblocks = 2




```
void VecAddCPU(float* A, float *B,
                        float* C, int n) {
  for (int i = 0; i < n; ++i) {
    C[i] = A[i] + B[i];
  }
}
```

Code for CPU

Ref: [3]

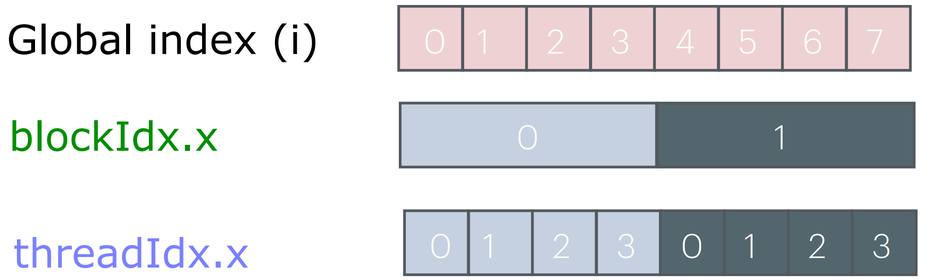
Example: vector addition in CUDA

Let's use an example of vector addition to compare code for CPU and GPU

For GPU, each thread do the addition

- Figure depicts a simple example of vector addition with dimension of n=8

n=8 (total vector dim, as example)
thread_per_block is 4,
blockDim.x = 4,
nblocks = 2




```
void VecAddCPU(float* A, float *B,
               float* C, int n) {
  for (int i = 0; i < n; ++i) {
    C[i] = A[i] + B[i];
  }
}
```

Code for CPU

```
__global__ void VecAddKernel(float* A, float *B,
                             float* C, int n) {
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

Compute global index

Ref: [3]

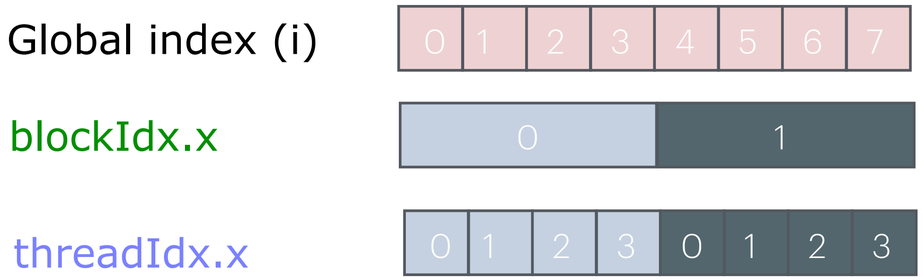
Example: vector addition in CUDA

Let's use an example of vector addition to compare code for CPU and GPU

For GPU, each thread do the addition

- Figure depicts a simple example of vector addition with dimension of n=8

n=8 (total vector dim, as example)
thread_per_block is 4,
blockDim.x = 4,
nblocks = 2




```
void VecAddCPU(float* A, float *B,
               float* C, int n) {
  for (int i = 0; i < n; ++i) {
    C[i] = A[i] + B[i];
  }
}
```

Code for CPU

```
__global__ void VecAddKernel(float* A, float *B,
                             float* C, int n) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < n) {
    C[i] = A[i] + B[i];
  }
}
```

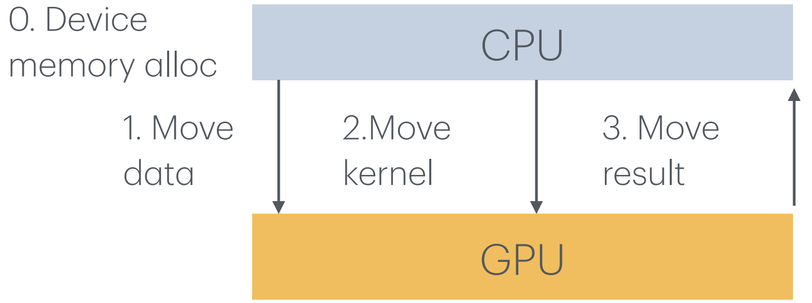
Compute global index

Each thread only run one addition

Code for GPU

Ref: [3]

Example of host code




```
void VecAddCUDA(float* Acpu, float *Bcpu, float* Ccpu, int n) {
 float *dA, *dB, *dC;
 cudaMalloc(&dA, n * sizeof(float));
 cudaMalloc(&dB, n * sizeof(float));
 cudaMalloc(&dC, n * sizeof(float));
 // copy memory from host to GPU
 cudaMemcpy(dA, Acpu, n * sizeof(float), cudaMemcpyHostToDevice);
 cudaMemcpy(dB, Bcpu, n * sizeof(float), cudaMemcpyHostToDevice);
 // launch vector addition kernel
 int threads_per_block = 512;
 int nblocks = (n + threads_per_block - 1) / threads_per_block;
 VecAddKernel<<<nblocks, thread_per_block>>>(dA, dB, dC, n);
 // copy memory from Gpu to host
 cudaMemcpy(Ccpu, dC, n * sizeof(float), cudaMemcpyDeviceToHost);

 cudaFree(dA); cudaFree(dB); cudaFree(dC);
}
```

GPU memory allocation

Copy memory to GPU

Kernel launch

Copy memory from GPU

Host code runs on CPU

CUDA code is consist of host code and device (GPU) code.

"Host" code runs serially on CPU and allocates device memory and bulk launch many CUDA threads.

CUDA device code runs on GPU, denoted with __global__

```
__global__ void VecAddKernel(float* A, float *B,
                             float* C, int n) {
 int i = blockDim.x * blockIdx.x + threadIdx.x;
 if (i < n) {
   C[i] = A[i] + B[i];
 }
}
```

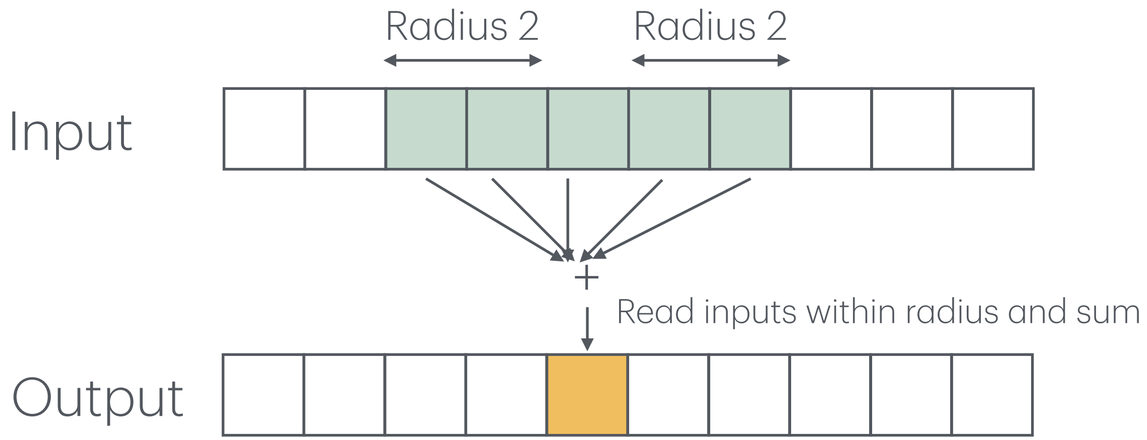
CUDA device code runs on GPU

Ref: [3]

Example: window sum

Computing the sum of a sliding window over vector

- Each output is the sum of input within a radius
- If radius is 2, each output is sum of 5 inputs

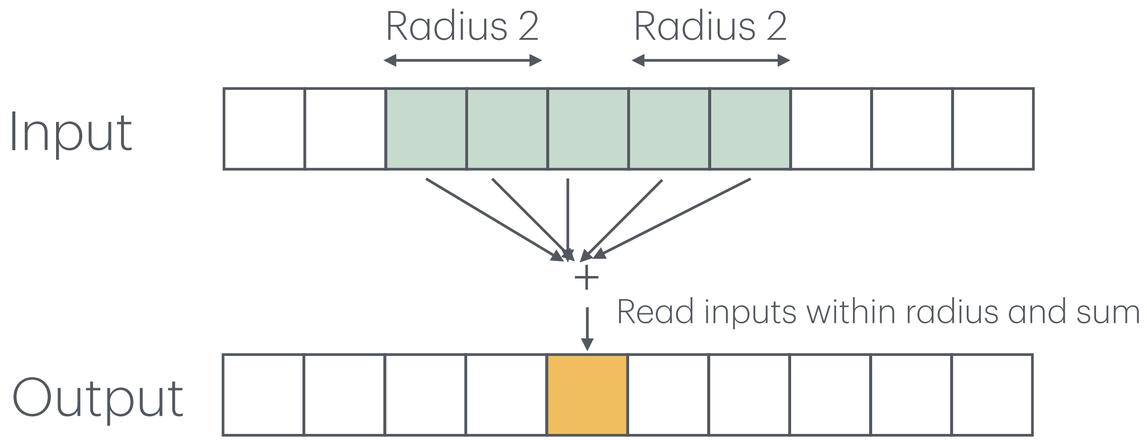


Ref: [2]

Example: window sum

Computing the sum of a sliding window over vector

- Each output is the sum of input within a radius
- If radius is 2, each output is sum of 5 inputs



Naive way code for window sum

```
#define RADIUS 2
__global__ void WindowSumSimpleKernel(
                        float* A, float *B, int n) {
 int out_idx = blockDim.x * blockIdx.x + threadIdx.x;
 if (out_idx < n) {
   float sum = 0;
   for (int dx = -RADIUS; dx <= RADIUS; ++dx) {
    sum += A[dx + out_idx + RADIUS];
   }
  B[out_idx] = sum;
}
```

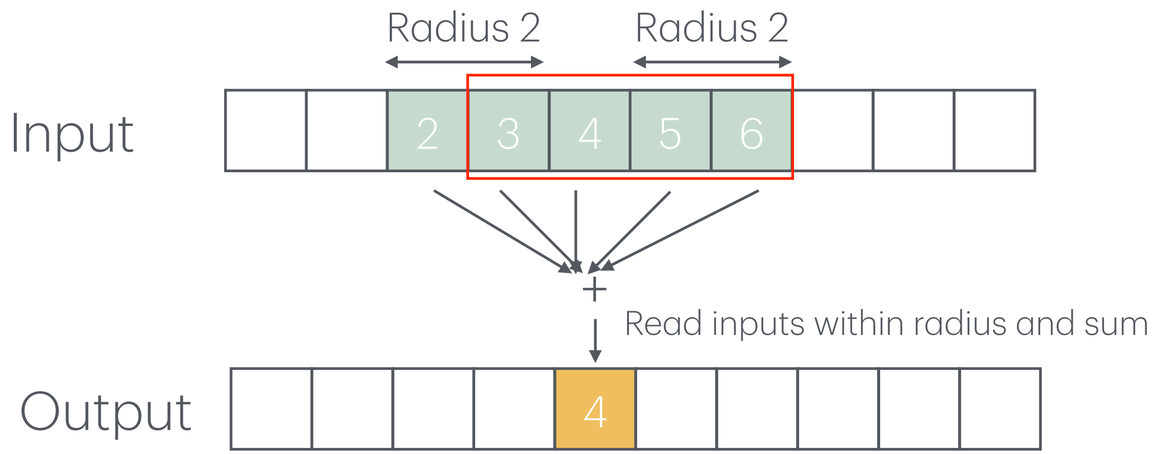
Ref: [2]

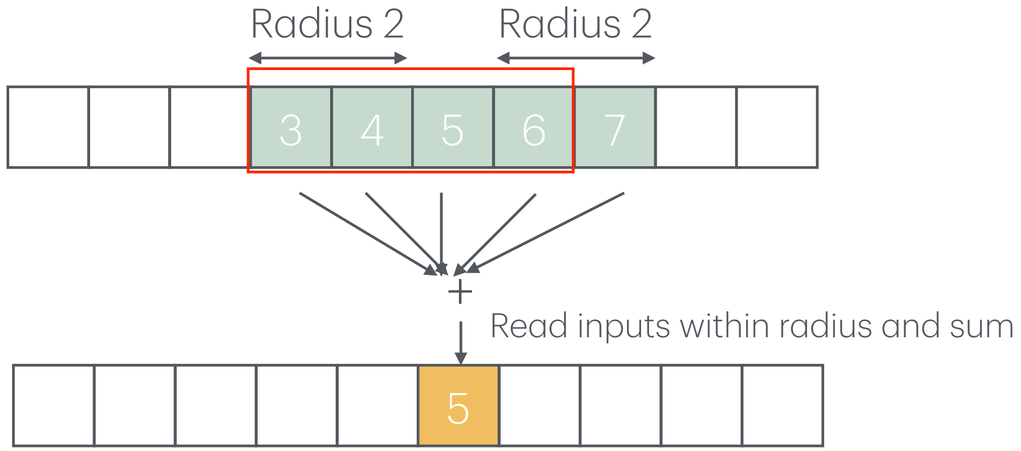
Example: window sum

Can we improve it?

For each input element, how many times it is read?

- Each input is read 5 times.
- Neighboring threads read most of the same input elements



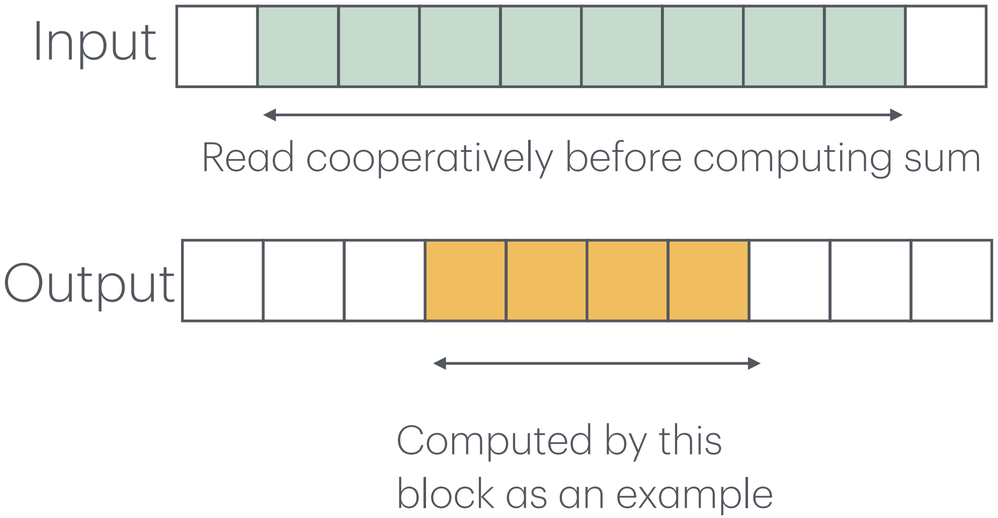


Ref: [2]

Example: window sum

Sharing data between threads within a block

A thread block cooperatively read input that are needed to shared memory before computing window sum



Ref: [2]

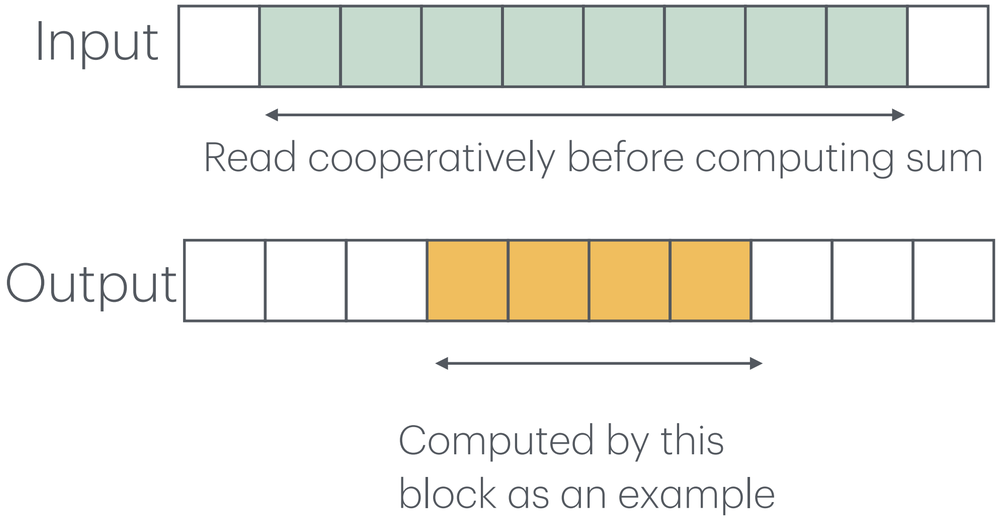
Example: window sum

Sharing data between threads within a block

```
__global__ void WindowSumSharedKernel(
                    float* A, float *B, int n) {
  __shared__ float temp[THREADS_PER_BLOCK + 2 * RADIUS];
```

Shared memory

A thread block cooperatively read input that are needed to shared memory before computing window sum

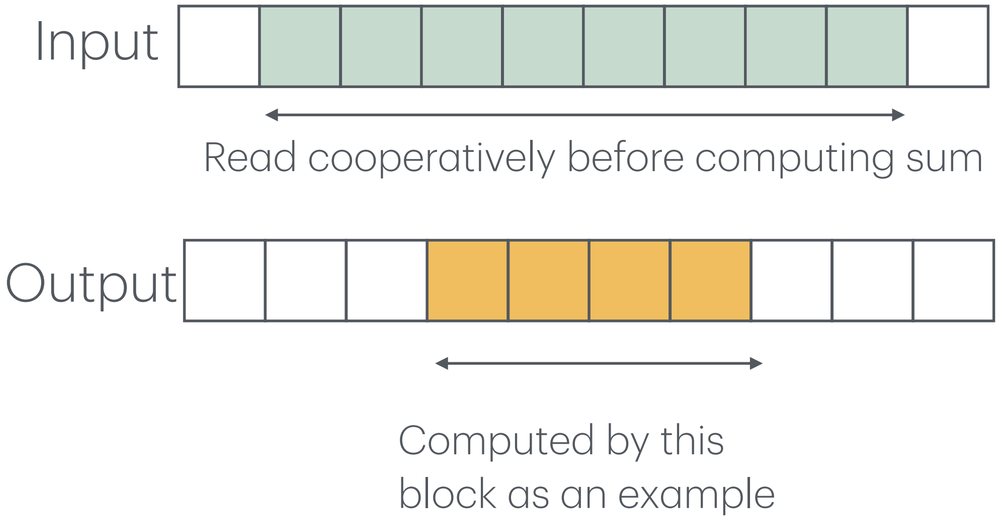


Ref: [2]

Example: window sum

Sharing data between threads within a block

A thread block cooperatively read input that are needed to shared memory before computing window sum




```
__global__ void WindowSumSharedKernel(
                    float* A, float *B, int n) {
  __shared__ float temp[THREADS_PER_BLOCK + 2 * RADIUS];
  int base = blockDim.x * blockIdx.x;
  int out_idx = base + threadIdx.x;
  if (base + threadIdx.x < n) {
    temp[threadIdx.x] = A[base + threadIdx.x];
  }
  if (threadIdx.x < 2 * RADIUS && base + THREADS_PER_BLOCK +
threadIdx.x < n) {
    temp[threadIdx.x + THREADS_PER_BLOCK] = A[base +
THREADS_PER_BLOCK + threadIdx.x];
  }
```

Shared memory

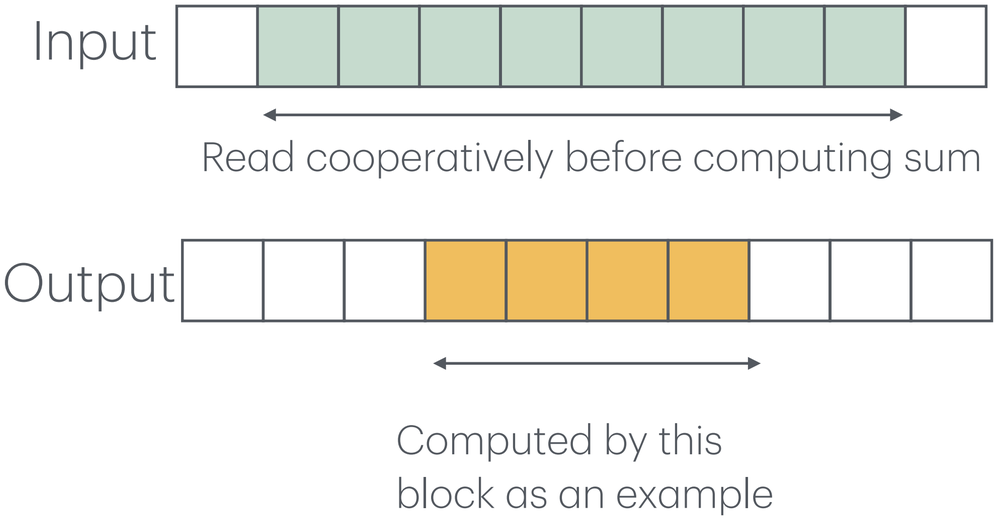
Read cooperatively

Ref: [2]

Example: window sum

Sharing data between threads within a block

A thread block cooperatively read input that are needed to shared memory before computing window sum




```
__global__ void WindowSumSharedKernel(
                    float* A, float *B, int n) {
  __shared__ float temp[THREADS_PER_BLOCK + 2 * RADIUS];
  int base = blockDim.x * blockIdx.x;
  int out_idx = base + threadIdx.x;
  if (base + threadIdx.x < n) {
    temp[threadIdx.x] = A[base + threadIdx.x];
  }
  if (threadIdx.x < 2 * RADIUS && base + THREADS_PER_BLOCK +
threadIdx.x < n) {
    temp[threadIdx.x + THREADS_PER_BLOCK] = A[base +
THREADS_PER_BLOCK + threadIdx.x];
  }
  __syncthreads();
```

Shared memory

Read cooperatively

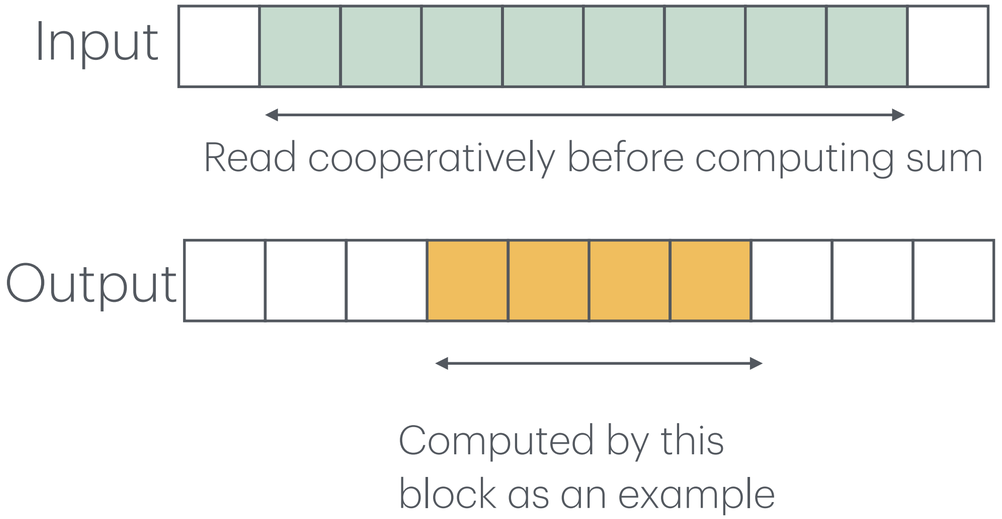
Synchronize threads

Ref: [2]

Example: window sum

Sharing data between threads within a block

A thread block cooperatively read input that are needed to shared memory before computing window sum




```
__global__ void WindowSumSharedKernel(
                    float* A, float *B, int n) {
  __shared__ float temp[THREADS_PER_BLOCK + 2 * RADIUS];
  int base = blockDim.x * blockIdx.x;
  int out_idx = base + threadIdx.x;
  if (base + threadIdx.x < n) {
    temp[threadIdx.x] = A[base + threadIdx.x];
  }
  if (threadIdx.x < 2 * RADIUS && base + THREADS_PER_BLOCK +
threadIdx.x < n) {
    temp[threadIdx.x + THREADS_PER_BLOCK] = A[base +
THREADS_PER_BLOCK + threadIdx.x];
  }
  __syncthreads();
  if (out_idx < n) {
    float sum = 0;
    for (int dx = -RADIUS; dx <= RADIUS; ++dx)
      sum += temp[threadIdx.x + dx + RADIUS];
    B[out_idx] = sum;
  }
}
```

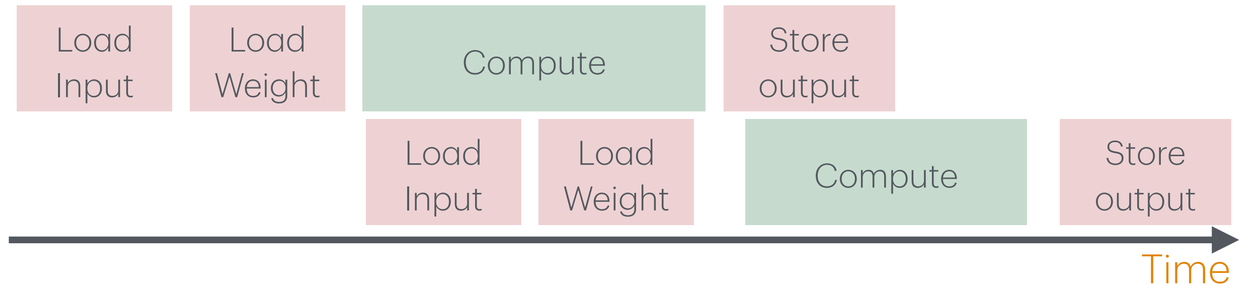
Shared memory

Read cooperatively

Synchronize threads

Ref: [2]

Compute vs. memory bound



When compute and memory I/O are overlapped, then total execution time is

$$\max(T_{mem}, T_{math}),$$

where T_{mem} is time spent accessing memory, T_{math} is time spent computation

The longer of the two limits the system performance.

- If T_{math} is longer, then the program is *compute bound* (math limited)
- If T_{mem} is longer, then the program is *memory bound* (I/O or memory or bandwidth limited)

Arithmetic intensity: how many operations we perform for each byte moved from memory

$$\text{arithemetic intensity} = \frac{\text{Total number of operations}}{\text{Number of bytes read from/written to memory}}$$

Ref: [5]

Roofline model

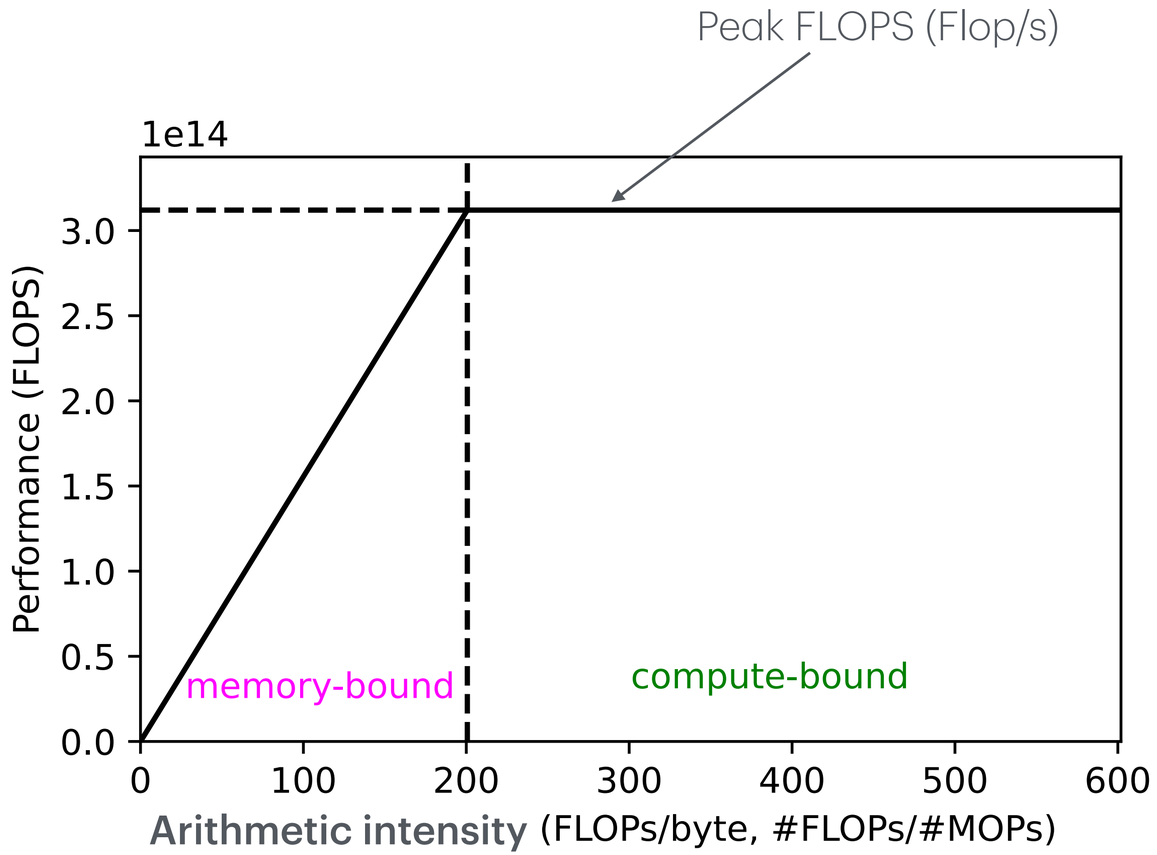
Roofline model provides a way to compare application performance against machine capabilities.

The X axis shows arithmetic intensity. For each floating-point operation, however many bytes moved.

The Y axis is the maximum achievable *peak* performance

If in memory-bound (bandwidth-bound) area, a program's peak performance is limited by memory I/O. Reducing the memory move will help and computation is free in this region.

If in compute-bound area, a program's peak performance is limited by hw's peak performance. Reducing computation will help in this region, such as using cached results.



$$\text{arithemetic intensity} = \frac{\text{Total number of operations}}{\text{Number of bytes read from/written to memory}}$$

Summary: Key technical concepts

1. GPU: includes many small cores that allows hide memory latency to produce high throughput

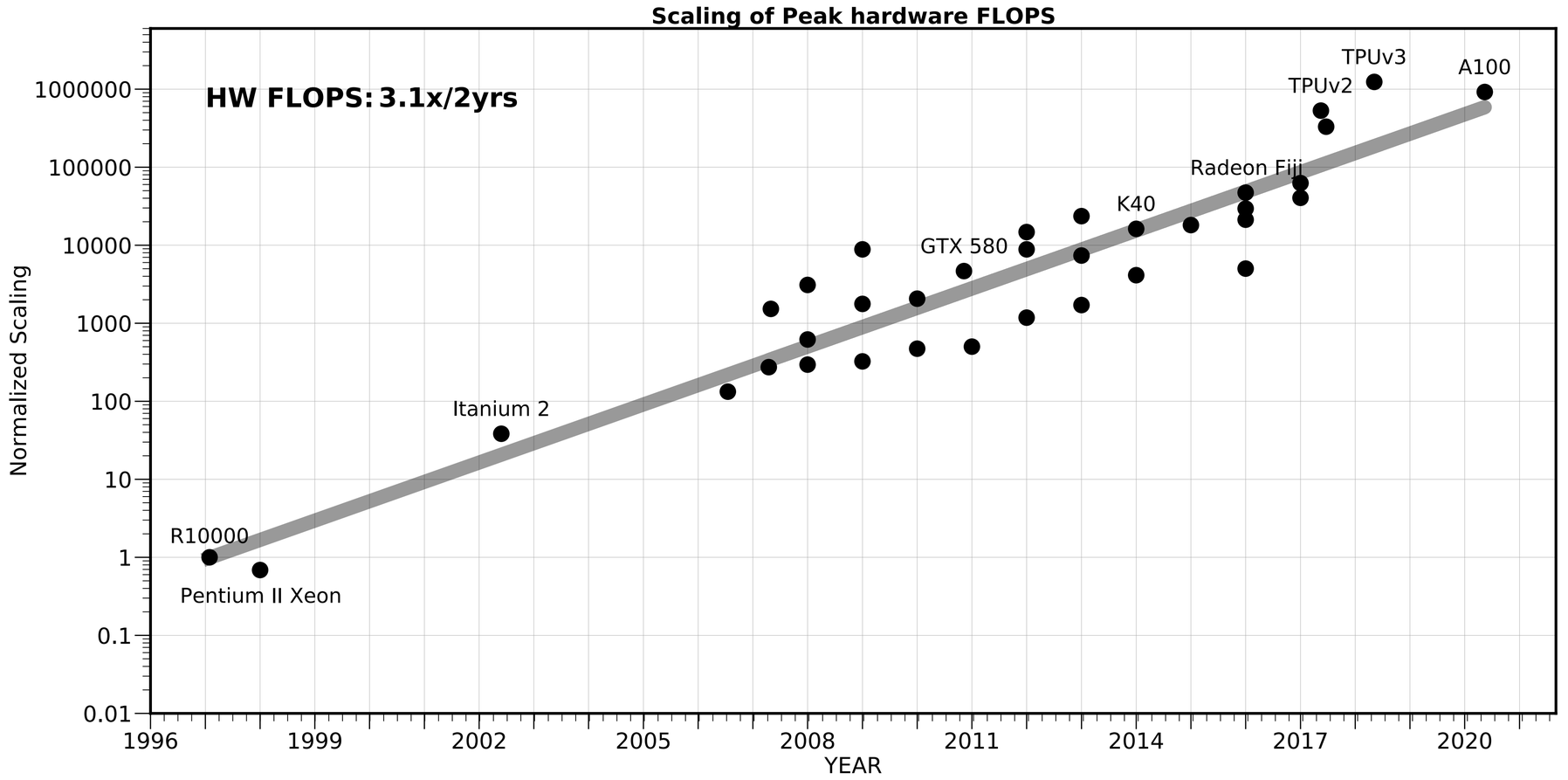
2. GPU programming model: threads are organized as thread blocks and blocks are organized as grid. Threads blocks will run the same instructions for different data.

3. Arithmetic intensity: for each floating point operation, how many bytes moved

4. Roofline model: provides a way to compare application performance against machine performance

Large model train/inference

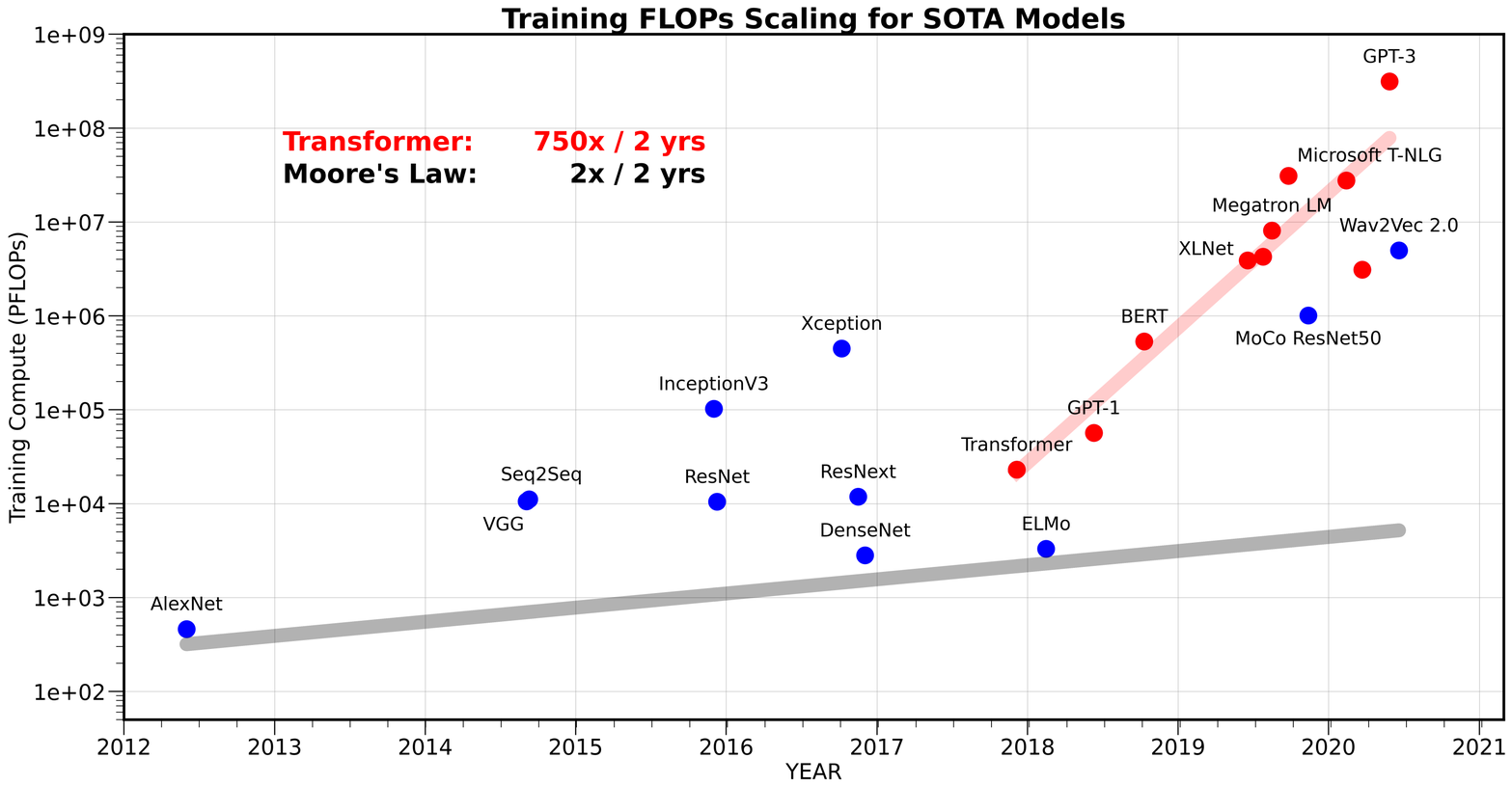
Increasing hardware FLOPS



Normalized scaling of peak hardware FLOPS over year

Image source: [4]

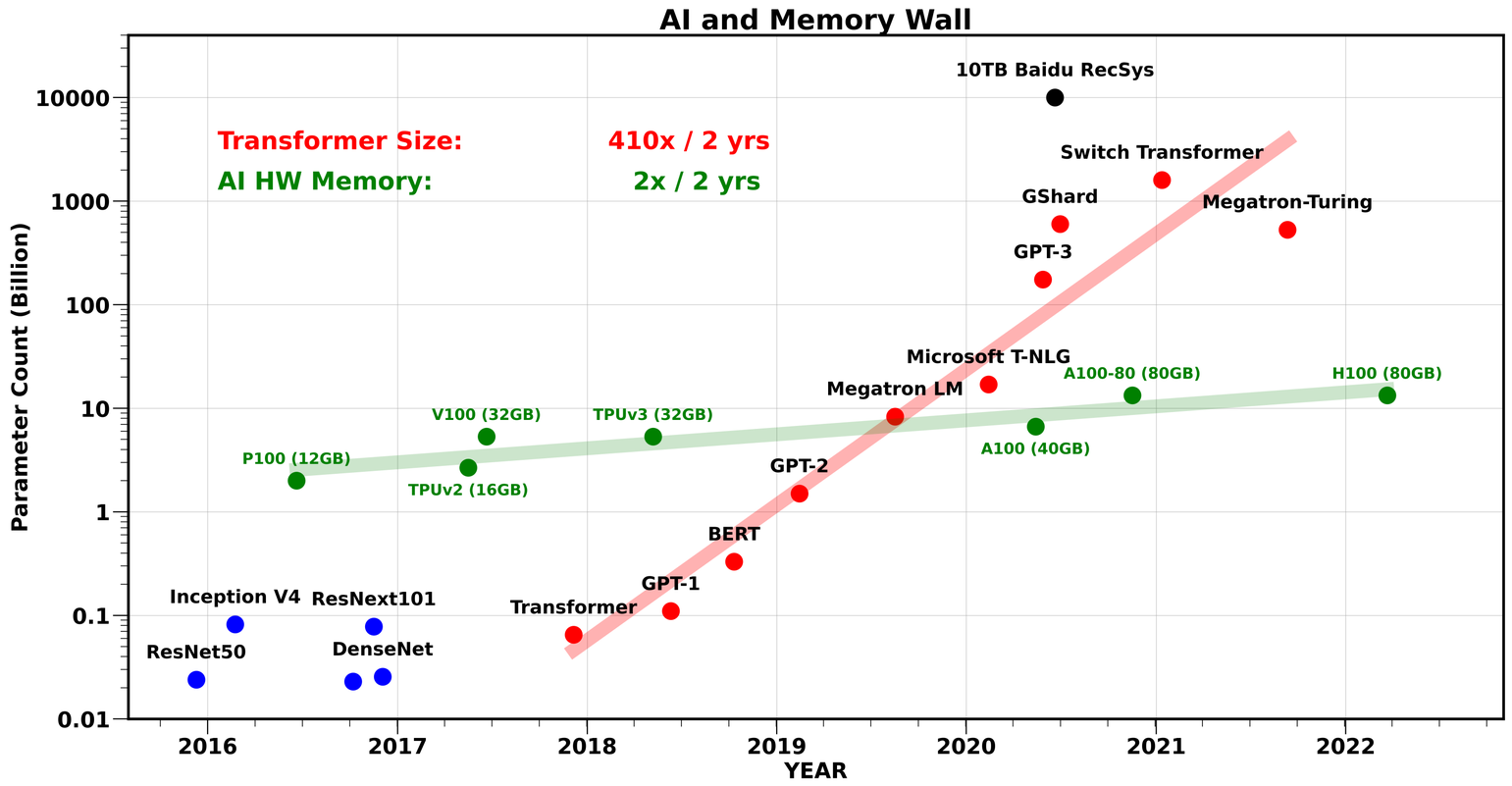
Increasing demand for compute



The amount of Training PFLOPs (Peta Floating Point Operations) need to train SOTA (state of the art) models in computer vision and natural language processing

Image source: [4]

Increasing model size and GPU memory



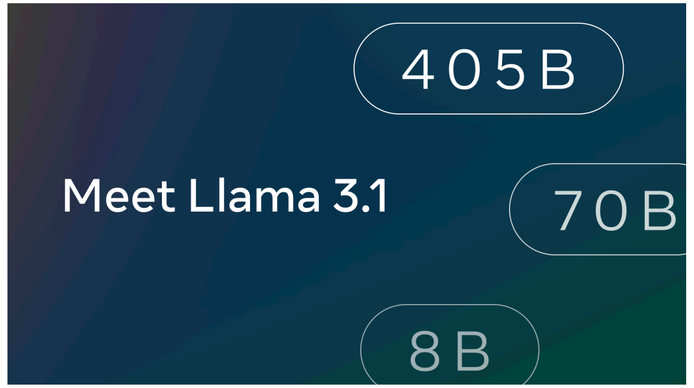
The increasing trend of model size growth requires more memory, 410x/2 yrs.

While the GPU memory increases 2x/2yrs

Scaling model size are bottlenecked by intra/inter-chip and communications across GPUs than compute.

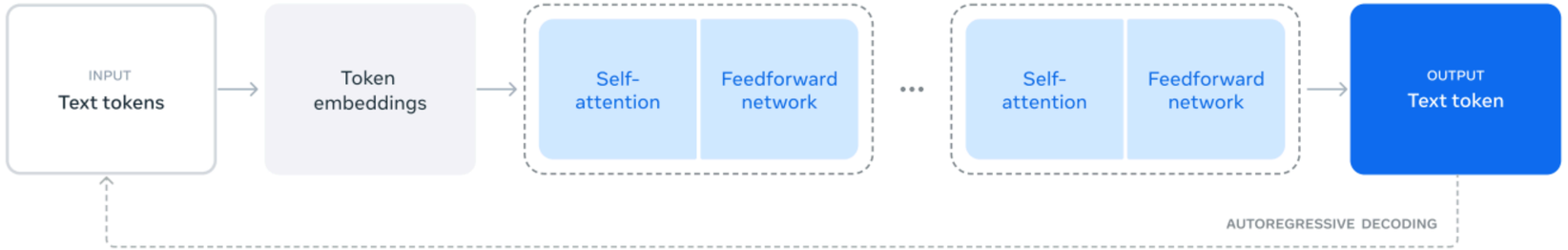
Image source: [4]

Llama 3 405b



Llama 3 405b model: recently released open weight LLM with 405 billions of parameters

- Simple model architecture, similar to GPT-2 with some improvements
- Parameters: 405b
- Data trained: 15 trillions of tokens
- Compute budget: 3.8 x 10^25 FLOPs
- Nvidia H100 GPUs used: 16,000
- Days trained: 54 days



	8B	70B	405B
Layers	32	80	126
Model Dimension	4,096	8192	16,384
FFN Dimension	6,144	12,288	20,480
Attention Heads	32	64	128
Key/Value Heads	8	8	8
Peak Learning Rate	3×10^{-4}	1.5×10^{-4}	8×10^{-5}
Activation Function	SwiGLU		
Vocabulary Size	128,000		
Positional Embeddings	RoPE ($\theta = 500,000$)		

Table 3 Overview of the key hyperparameters of Llama 3. We display settings for 8B, 70B, and 405B language models.

Ref. and image source: link

Distributed training

Distributed training: training model across multiple devices

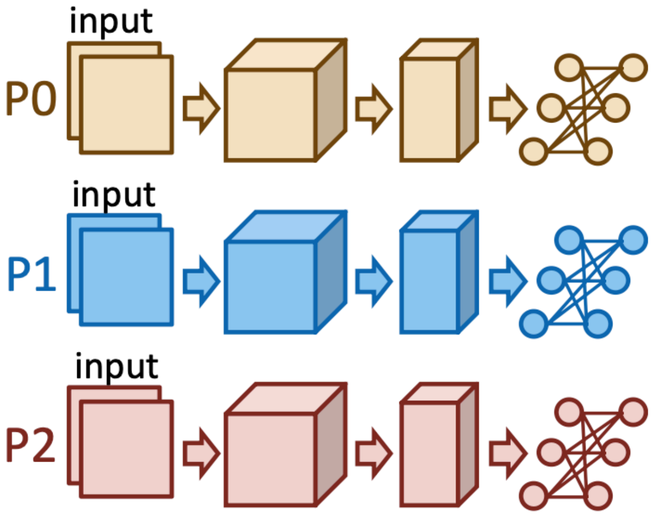
Why do we need distributed training?

- Need more memory: to train larger model we need to store weights, activations, gradients, optimization states.
- Need to train faster: to train faster by parallel computation

Ref: [7]

Large scale training

Data parallelism



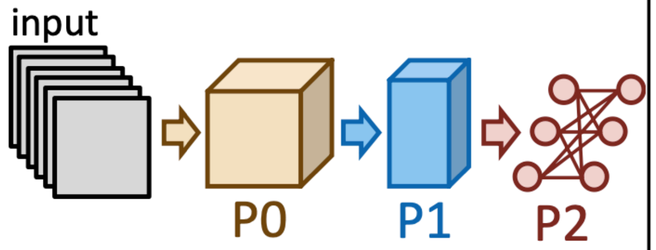
Pros:

a. Easy to realize

Cons:

a. Not work for large models
b. High allreduce overhead

Pipeline parallelism



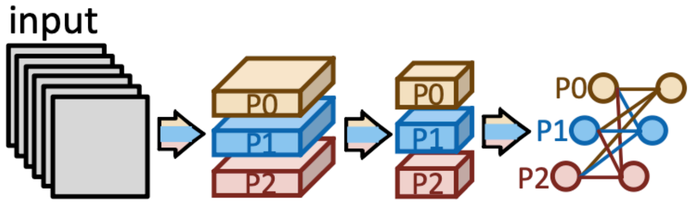
Pros:

a. Make large model training feasible
b. No collective, only P2P

Cons:

a. Bubbles in pipeline
b. Removing bubbles leads to stale weights

Tensor parallelism



Pros:

a. Make large model training feasible

Cons:

b. Communication for each operator (or each layer)

Note: P0, P1, P2 are processors (GPUs) that can execute model operators

Image source link

Data parallelism

Compute the entire morel on each processor (GPUs) P0, P1, P2, …

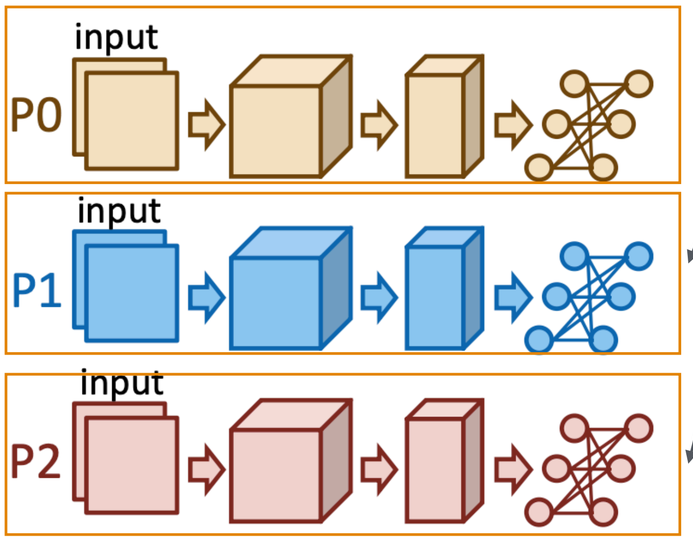
Distribute data batch evenly across each processor

Does not work for large model alone, it can be used with other parallelism methods (TP, PP).

At the end of backward propagation, communicate gradient update through allreduce collective communication

- Parameter server: high fan-in
- Ring allreduce: communication amount is evenly distributed. Fan-in is constant. Number of communication rounds is (# of processor - 1)
- Double binary tree all-reduce: double the fan-in size, but reduce the communication rounds with log(# of processor)

Full models run on different processors using different data batch



Pros:

a. Easy to realize

Cons:

a. Not work for large models
b. High allreduce overhead

Note: P0, P1, P2 are processors (GPUs) that can execute model operators

Image source link

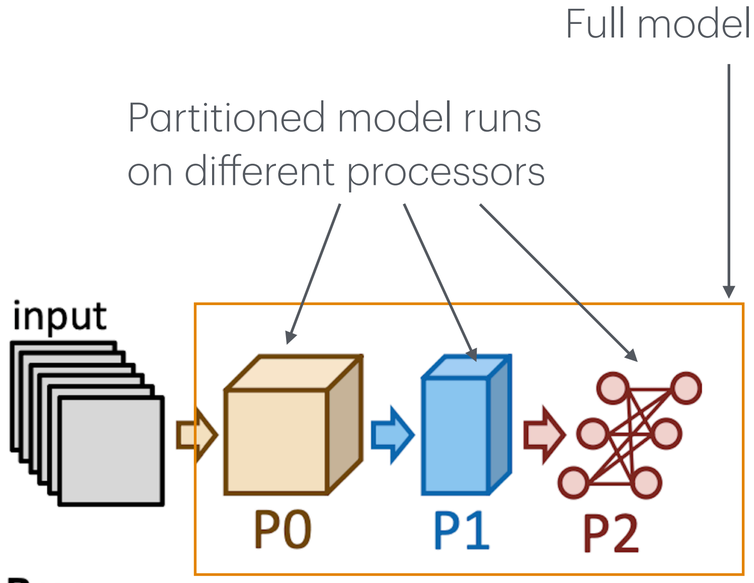
Pipeline parallelism

Partition model weights ‘vertically’ across model layers and run on different processors (GPUs), P0, P1, P2, …

Pipeline parallelism lowers hardware utilization due to idle time (bubble) between forward and backward pass

There are various approaches to reduce bubbles, such as GPipe, PipeDream

It requires only *point to point* communication, which is much cheaper than collective communications, so it is good for large scale training



Pros:

a. Make large model training feasible
b. No collective, only P2P

Cons:

a. Bubbles in pipeline
b. Removing bubbles leads to stale weights

Note: P0, P1, P2 are processors (GPUs) that can execute model operators

Ref. [7] and image source link

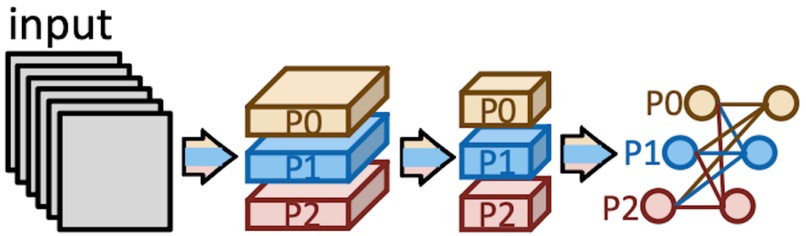
Tensor parallelism

Split individual layers and run across processors (GPUs), P0, P1, P2, …

Supports large model by breaking into smaller parts

Requires all gather communication for forward pass, and all reduce communication for backward pass

It is harder to define the split and implement



Pros:

a. Make large model training feasible

Cons:

b. Communication for each operator (or each layer)

Note: P0, P1, P2 are processors (GPUs) that can execute model operators

Ref. [7] and Image source link

Large scale training

Llama3 405b

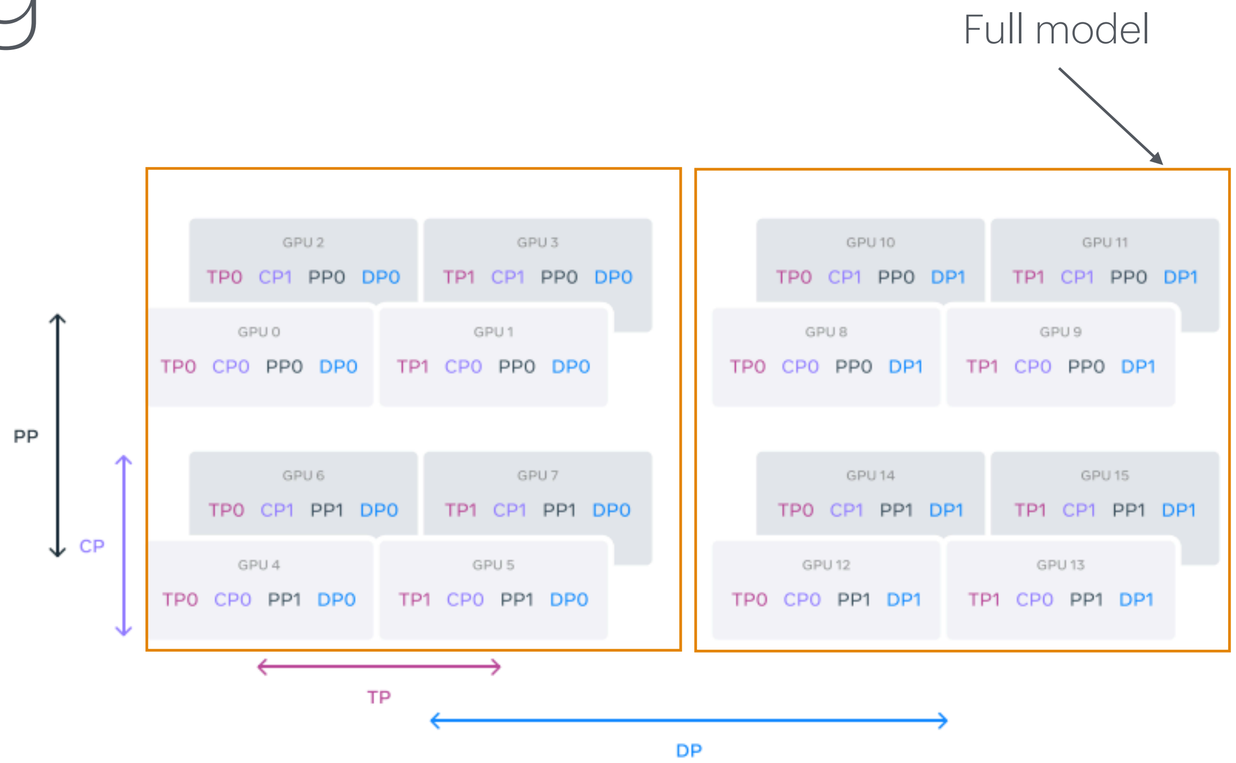
Large scale training uses multiple parallelisms.

Tensor parallelism (TP) requires the *highest network bandwidth and lower latency*, usually contained with the same server

Data parallelism (DP) may spread across a multi-hop network and can tolerate higher network latency up to tens of microseconds

For llama3 405b model, context parallelism (CP) is also used, which supports for long input sequences.

Llama 405b used 4D parallelism, TP, CP, PP, DP



Note: TP is Tensor parallelism, CP is Context parallelism, PP is Pipeline parallelism, and DP is Data parallelism.

GPUs	TP	CP	PP	DP	Seq. Len.	Batch size/DP	Tokens/Batch	TFLOPs/GPU	BF16 MFU
8,192	8	1	16	64	8,192	32	16M	430	43%
16,384	8	1	16	128	8,192	16	16M	400	41%
16,384	8	16	16	4	131,072	16	16M	380	38%

4D parallel configuration for different input sequence lengths

Ref. link

Summary: Key technical concepts

1. Data parallelism: put entire model on multiple processors and feed partitioned data to improve training time

2. Pipeline parallelism: partition model layer boundary and run on multiple processors. It only requires point-to-point communication, but need to put efforts to reduce bubbles.

3. Tensor parallelism: partition individual layers and run on multiple processors, requires highest network bandwidth and lowest latency

4. To train large scale model requires multiple parallelisms, such as 4D parallelism used by llama3 405b model

Parting thoughts, large scale models...

Model size grows fast and hardware growth rate is slower while demands of using models are growing...

There are various approaches to improve efficiency in deep learning systems.
Next module, we will review the challenges of serving LLM inference and selected approaches to improve efficiency:

- Model compression
- Algorithm optimization
- System optimization

References

1. MIT 6.5940: TinyML and Efficient Deep Learning Computing
2. Deep Learning Systems, Algorithms and Implementation
3. CUDA Programming guide
4. AI and Memory Wall
5. Systems for Machine Learning
6. PARALLEL COMPUTING
7. Machine Learning Systems (Spring 2022)
8. Deep Learning for Computer Vision
9. Accelerator Architectures for Machine Learning
10. Understanding GPU architecture