Accelerating Machine Learning through Software + Hardware

Tsung Tai Yeh

Computer Science Department of National Chiao Tung University, Taiwan

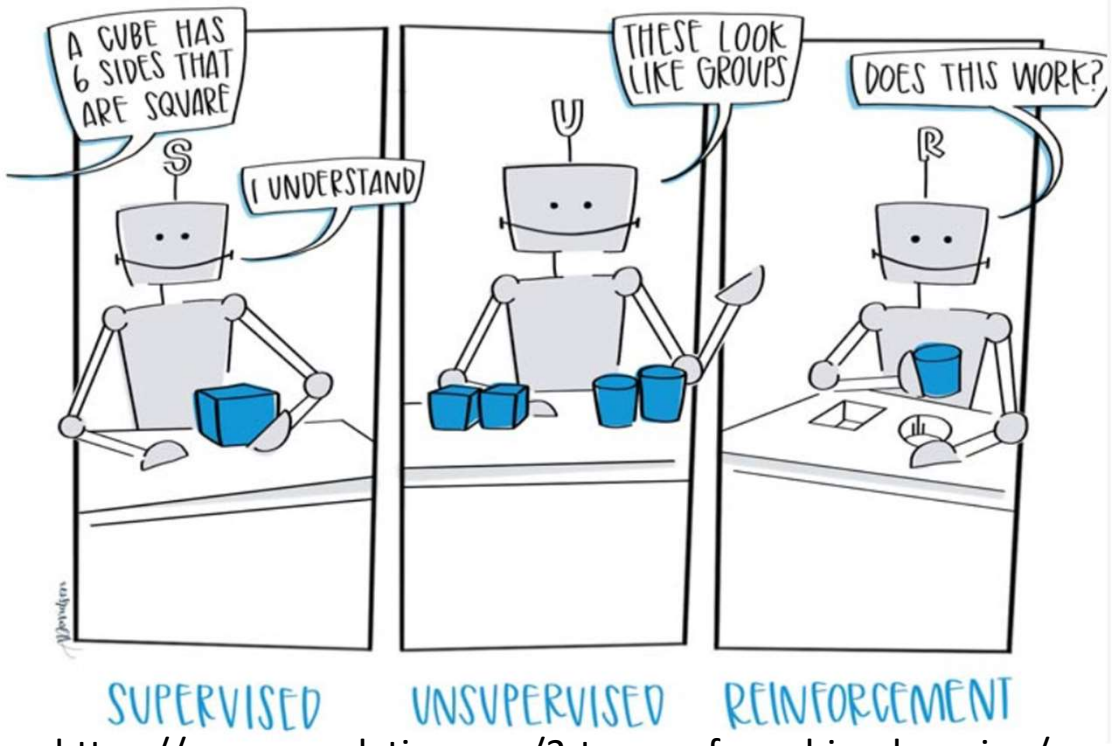
Overview

- Machine Learning & Deep Neural Network
- Golden Age of Microprocessor Design
- Software Optimization Techniques
- Domain Specific Accelerator
- My Research and my CAS Lab at NYCU

Machine Learning & Deep Neural Network

What is Machine Learning ?

- "Giving computers the ability to learn without being explicitly programmed" – Arthur Samuel, 1959s



https://www.ceralytics.com/3-types-of-machine-learning/

Supervised Learning

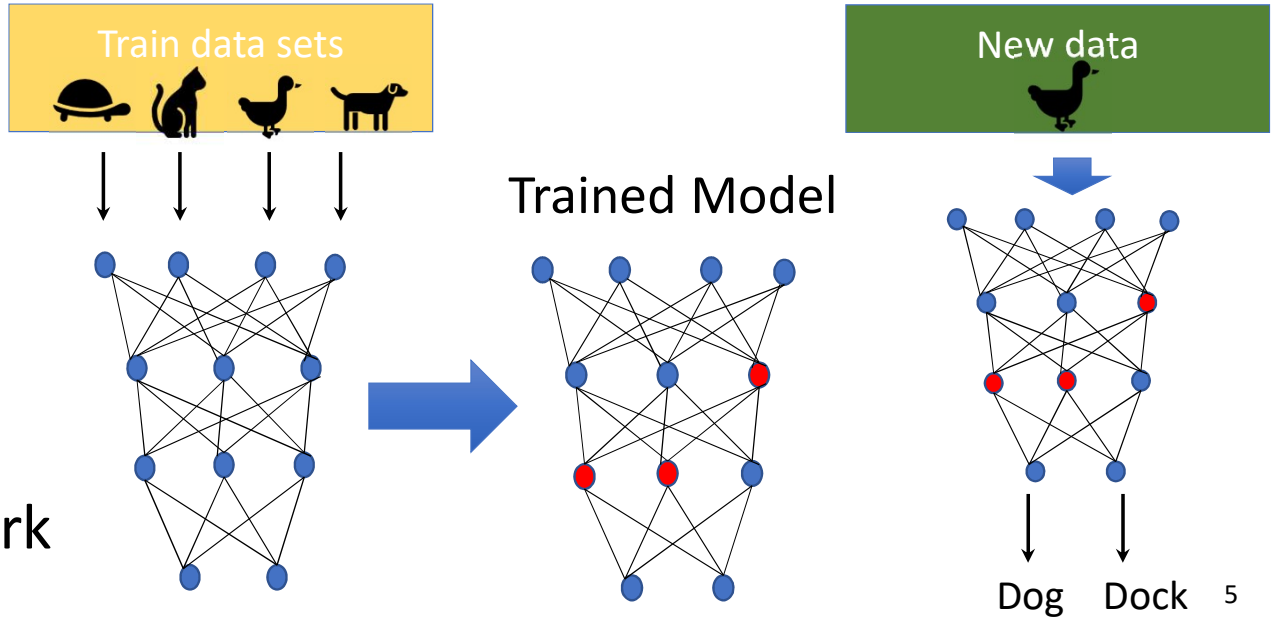
- **Data:**(x, y)

  The x is data, y is label

- **Goal:** Learn a function to map x from label data y

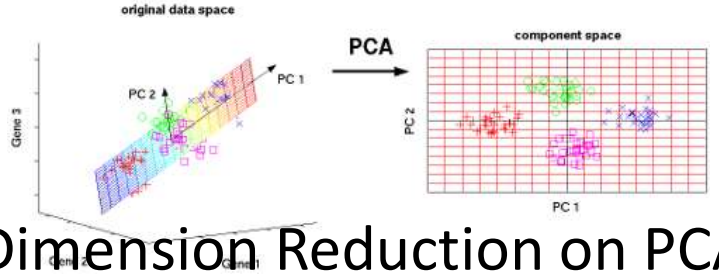
- **Examples:** Object detection, classification, image captioning etc..

- **Problems:**
  - Tedious labelling work



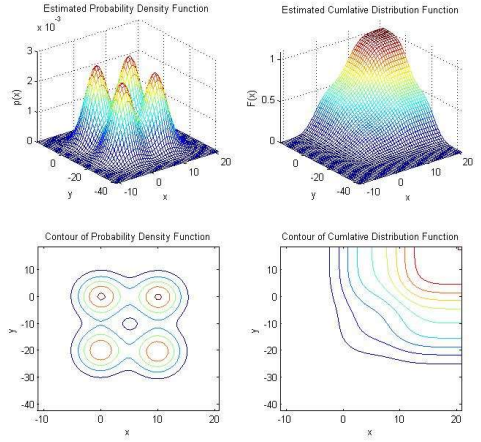
Unsupervised Learning

- **Data:** x, no labels !
- **Goal:** Learn underlying hidden structures of the data
- **Example:** Clustering, feature learning, density estimation, dimensionality reduction etc..
- **Problem:**
  - The curse of dimensionality



Dimension Reduction on PCA

http://www.nlpca.org/pca_principal_component_analysis.html

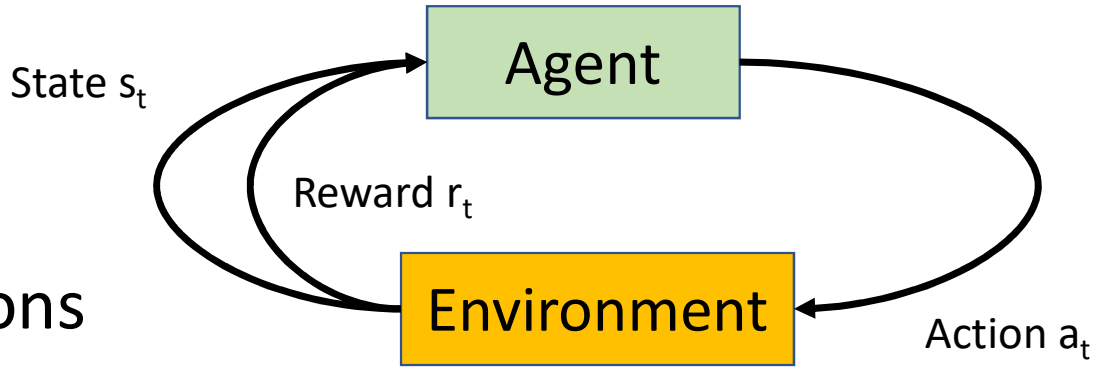


2-D Density Estimation

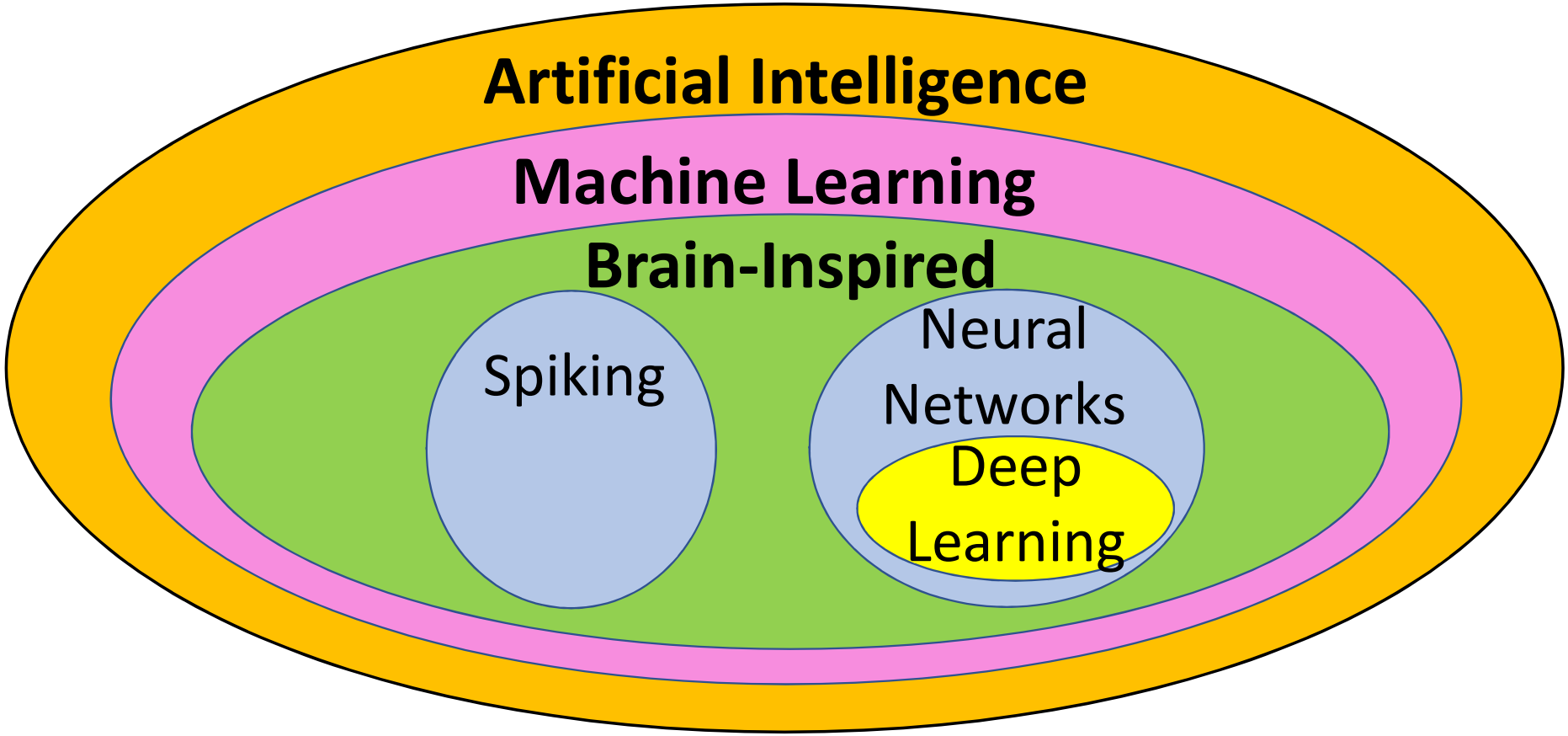
https://www.mathworks.com/matlabcentral/fileexchange/19280-bivariant-kernel-density-estimation-v2-1

Reinforcement Learning

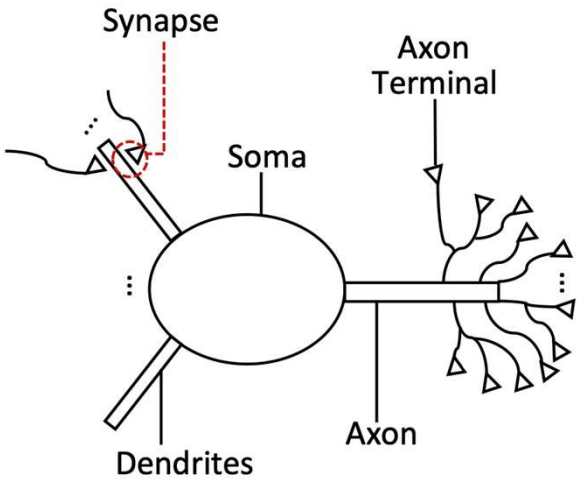
- An **agent** interacts with the **environment**
- Learning from the **reward** signals
- **Goal:** Learn how to take actions to maximize the reward
- **Examples:** Robots control, Deep mind AlphaGo, Atari Gaming
- **Problems:**
  - Reliability

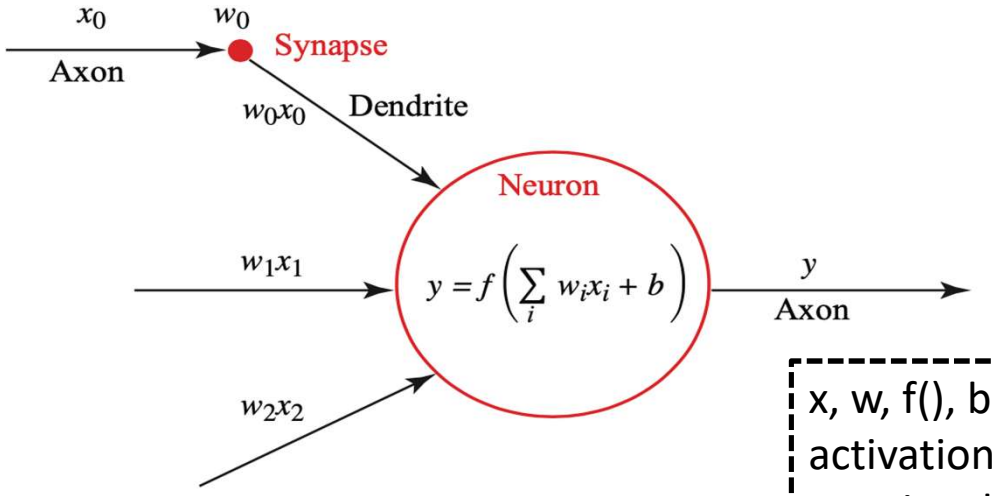


Deep Learning



How does the brain work ?





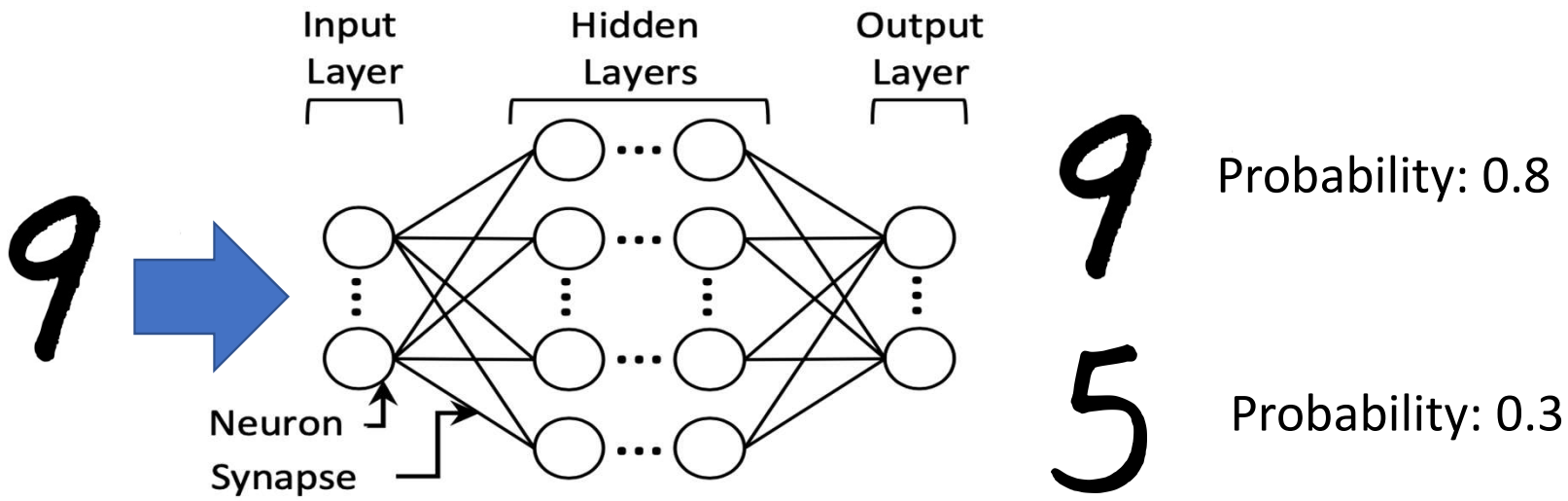
x, w, f(), b are activations(input/output signals), weights, nonlinear function, bias

- **Neurons** (86 B) (perception) are assembled into layers which are connected via synapses
- **Dendrites** receive inputs from upstream neurons via the synapses.
- Soma membrane fires inputs to an axon.
- **Axons** terminals transmits outputs to downstream neurons.

https://arxiv.org/pdf/2005.01467.pdf

Neural Networks (NN)

- **Hidden layer**: neurons in this layer are neither inputs nor outputs, extracting input features
- To encode the **intensities** of image pixels to the input neurons
- Picking the output values > 0.5 that indicates input image is 9

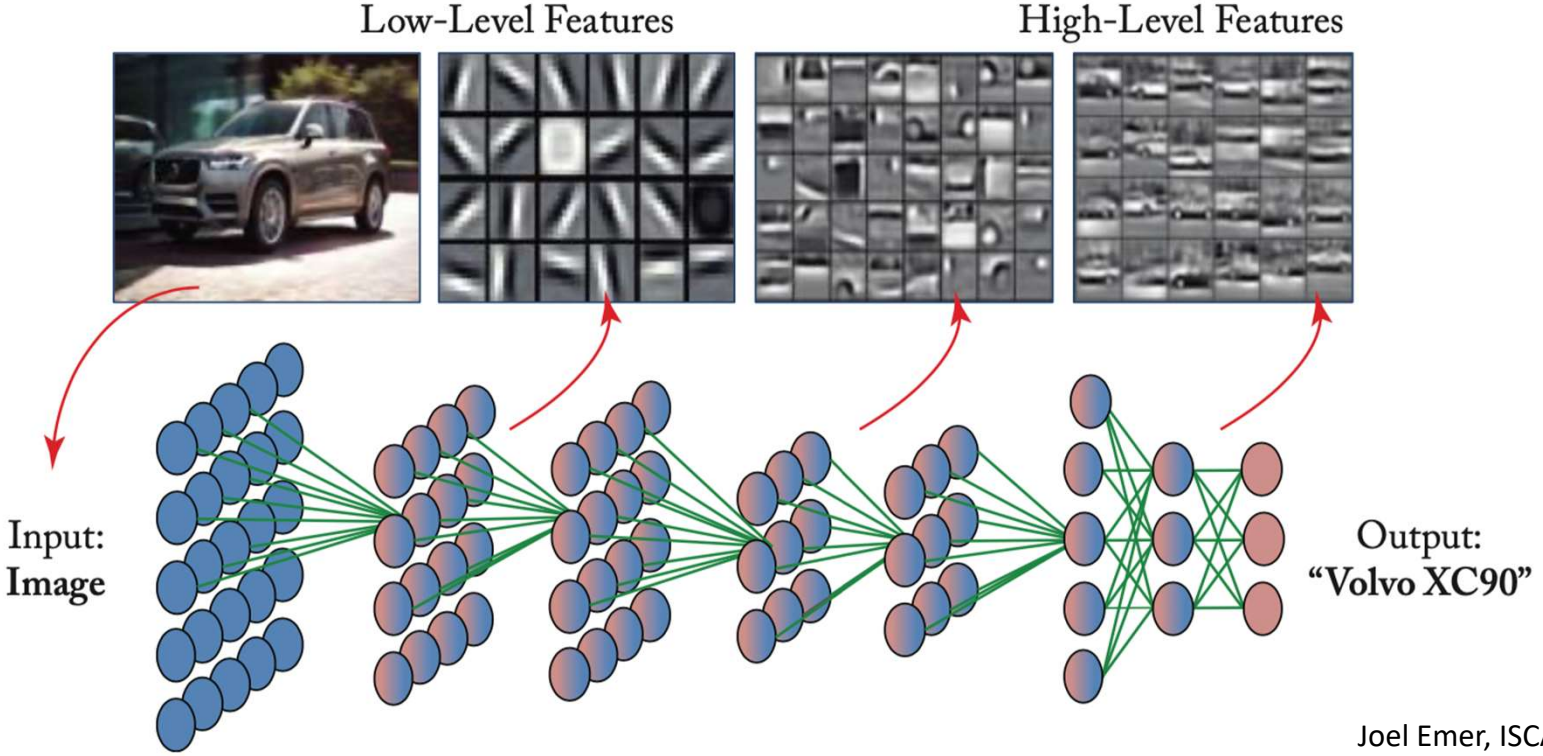


Multilayer perceptrons (MLPs)

https://arxiv.org/pdf/2005.01467.pdf

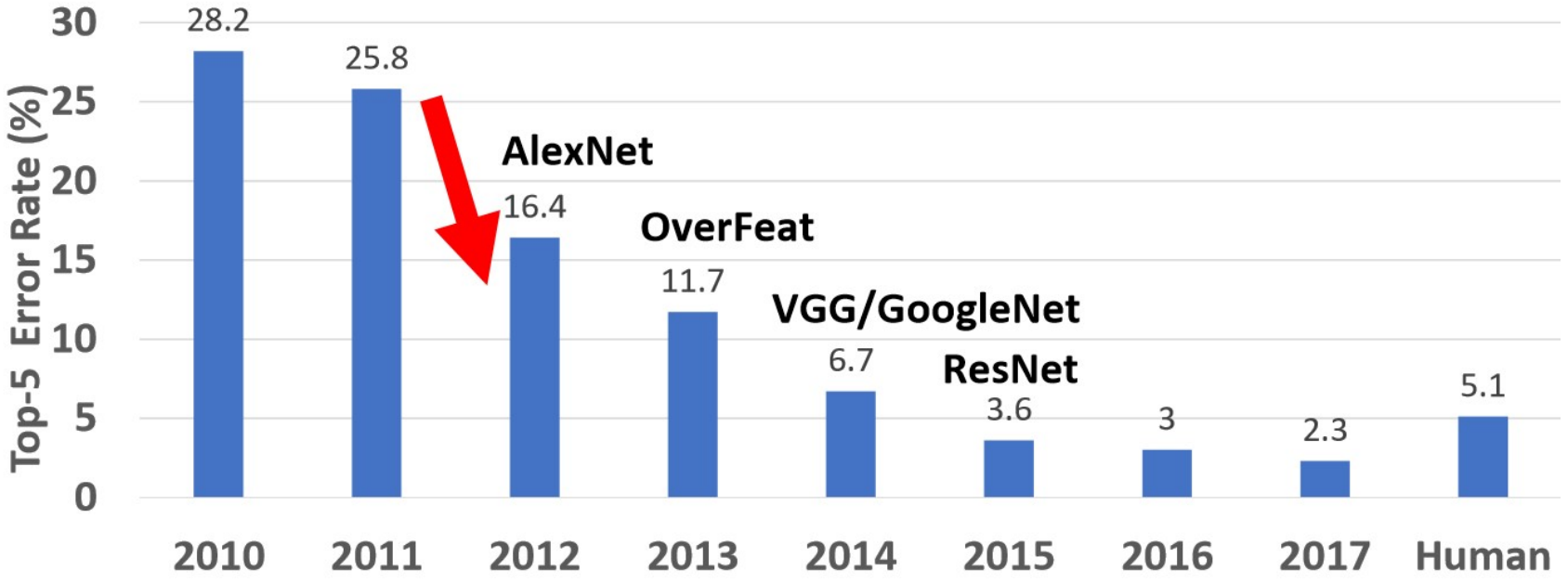
What is Deep Learning ?

- DNN has more than **3** layers (more than one hidden layer)
- DNNs can learn high-level features than shallow neural networks



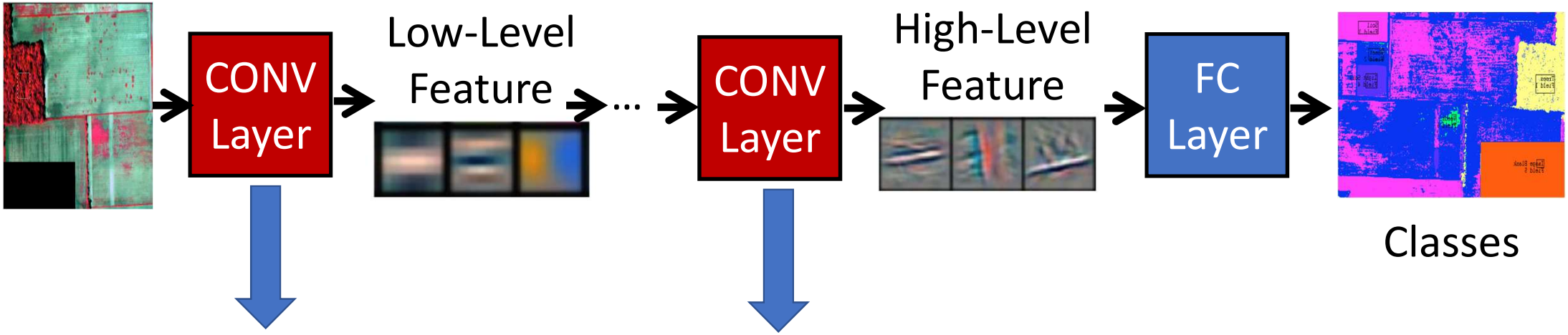
Why Deep Neural Network is popular?

- DNN model outperforms human-being on the ImageNet Challenge



https://arxiv.org/ftp/arxiv/papers/1911/1911.05289.pdf

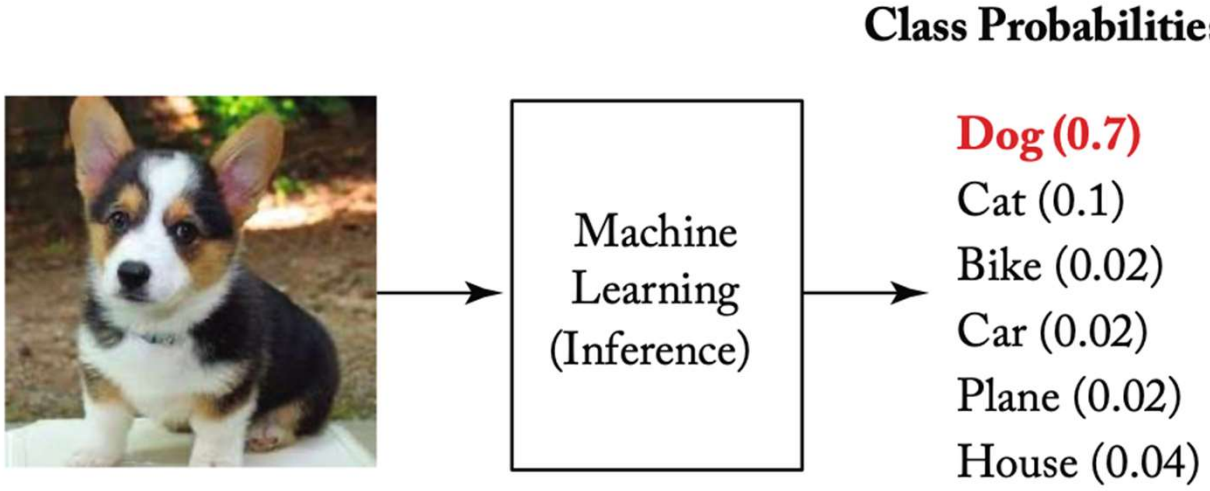
Convolutional (CONV) Layers



1. Convolutions mainly perform **vector-and-matrix multiplication**.
2. Convolutions takes more than **90%** of overall computation (critical path).
3. Optimization (software/hardware) for convolutions matters.

Training versus Inference

- **Training**: Determining the value of the **weights** in the network
  - Minimizing loss (L)
  - Loss (L): the gap between ideal correct probabilities and the probabilities computed by the DNN model
- **Inference**: Apply trained weights to determine output. Include only forward pass



No free lunch on DNN computation

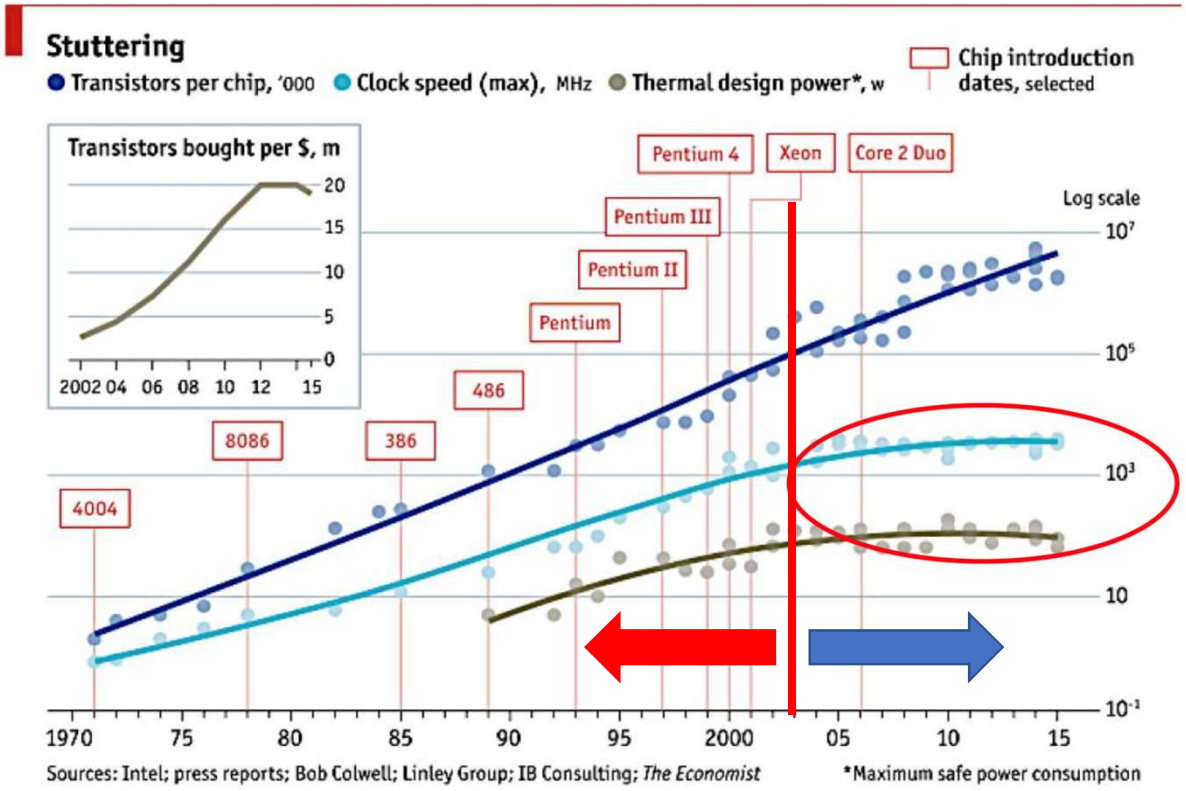
- AlexNet to AlphaGo Zero: A 300,000 x Increase in Compute

https://arxiv.org/ftp/arxiv/papers/1911/1911.05289.pdf

Challenges of Existing Computer Hardware

Moore's Law

- The number of transistors per chip **doubles** every 18-24 months
- The performance grows linearly with the transistor count (Before 2004)



A Golden Age in Microprocessor Design

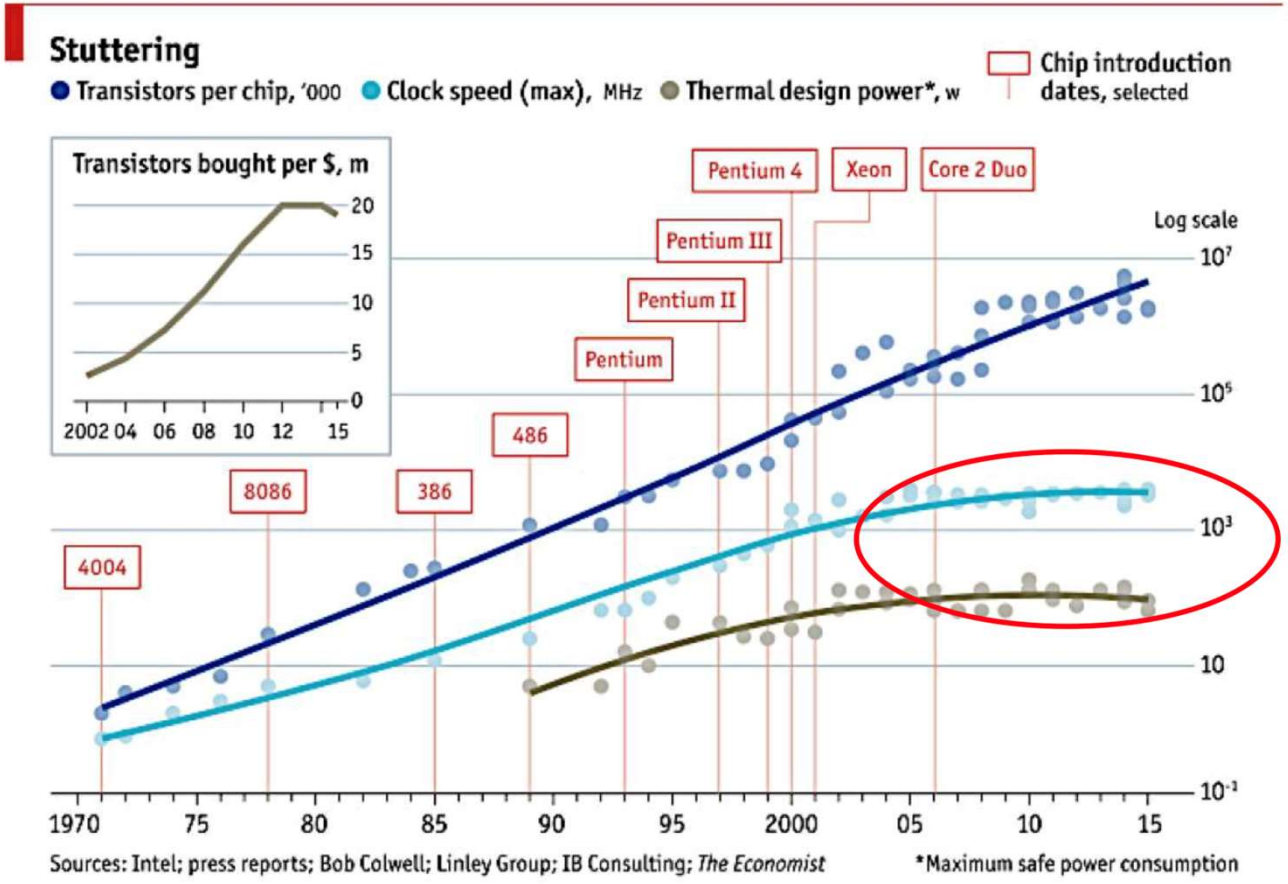
- A great leap in microprocessor speed ~10^3 X faster over 30 years
- Apple computer with similar prices from 1977 to 2004

Apple II	
Launch:	1977
Clock rate:	1MHz
Data path:	8 bits
Memory:	48 KB
Cost:	$1,395

Power Macintosh G4	
Launch:	2000
Clock rate:	400MHz
Data path:	32 bits
Memory:	64 MB
Cost:	$1,599

Power Macintosh G5	
Launch:	2004
Clock rate:	1.8GHz
Data path:	64 bits
Memory:	256 MB
Cost:	$1,499

Increasing transistors isn't getting efficient



General purpose processor is not getting faster and power-efficient because of **Slowdown of Moore's Law and Dennard Scaling**

Dennard Scaling

- As the size of the transistor becomes **small**
  - The voltage is reduced
  - Circuits can be operated at higher frequency at the same power

Related to transistor size

Power = alpha x CFV^2

alpha: percent time switched

C: capacitance

F: Frequency

V: Voltage

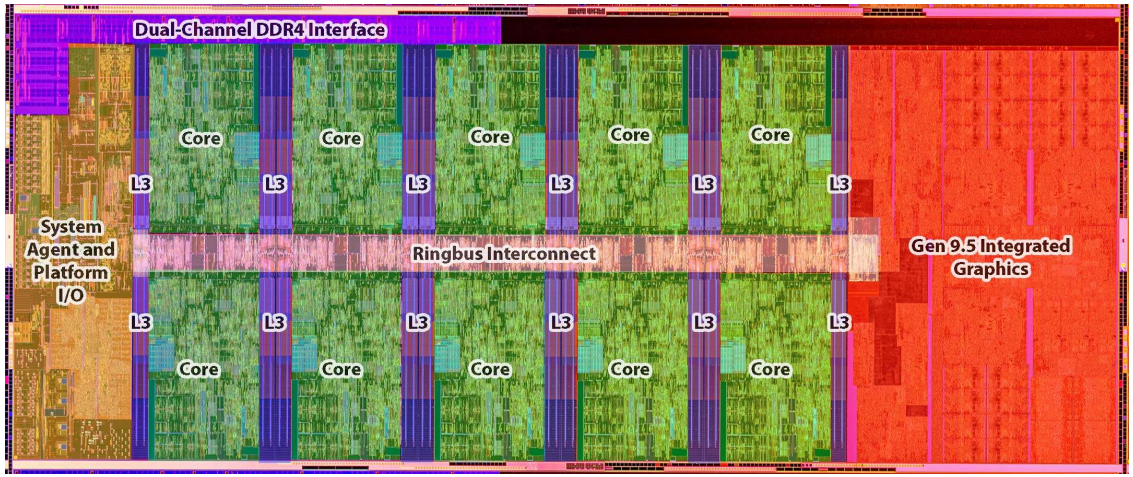
What's wrong on Dennard Scaling?

Dennard Scaling ignores "leakage current" ,"threshold voltage"

So, as transistors get small, power density increases !!

Vendor's Solution: Multi-core

- Intel Core i9 10900K (Comet Lake-S), 2020
  - 10 cores
  - 3.7 GHz
  - 20 MB L3 Cache
- In the new multi-core
  - Big multi-core processors
  - Complex cache hierarchies
  - Wide vector units
  - GPU
  - FPGA



Challenge: How to adapt the software to utilize this hardware efficiently ?

https://www.techpowerup.com/review/intel-core-i9-10900k/3.html

Software Optimization Techniques

Case Study: Matrix Multiplication

- Matrix Multiplication (MM) takes the majority of computation time in DNN apps
- Square-Matrix Multiplication

$$\begin{bmatrix} C_{11} & \dots & C_{1n} \\ \dots & \dots & \dots \\ C_{n1} & \dots & C_{nn} \end{bmatrix} = \begin{bmatrix} A_{11} & \dots & A_{1n} \\ \dots & \dots & \dots \\ A_{n1} & \dots & A_{nn} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & \dots & B_{1n} \\ \dots & \dots & \dots \\ B_{n1} & \dots & B_{nn} \end{bmatrix}$$

C A B

$$C_{ij} = \sum_{k=0}^{n} A_{ik} b_{jk}$$

Assuming for the simplicity $n = 2^k$

AWS Machine Specs

Feature	Specification
CPU	Intel Xeon E5-2666 v3
Clock frequency	2.9 GHz
Processor chips	2
Processing cores	9 per processor chip
Hyperthreading	2 ways
Float-point units	8 double-precision operations, including fused-multiply-add, per core per cycle

Peak = $(2.9 \times 10^9) \times 2$ (hyper-threading) x 9 (# of cores) x 2 (# of processors) x 8 (SIMD double precision) = 836 GFLOPS

Version 1: Nested Loops in Python

- Running time = **21042** seconds ≈ 6 hours
- $2n^3 = 2(2^{12})^3 = 2^{37}$ floating-point operations takes $2^{37}/21042 \approx$ **6.25** MFLOPS
- Python gets ≈ **0.00075%** of peak

```
import sys, random
from time import *
n = 4096
A = [[random.random() for row in xrange(n)] for col in xrange(n)]
B = [[random.random() for row in xrange(n)] for col in xrange(n)]
C = [[random.random() for row in xrange(n)] for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time ()
print '0.3f' % (end - start)
```

Version 2: Java

- Running time = **2,738** seconds ≈ **46** minutes
- About **8.8 X** faster than Python

```
import java.util.Random
Public class mm_java {

   static int n = 4096
   static double [][] A = new double[n][n]
   static double [][] B = new double[n][n]
   static double [][] C = new double[n][n]
   public static void main(string [] args )
   {
          …. // init A, B, C
          long start = System.nanoTime();
          for (int i  = 0; i < n; i++) {
              for (int j  = 0; j < n; j++) {
                   for (int k  = 0; k < n; k++) {
                        C[i][j] += A[i][k] * B[k][j]; }}}
           long stop = System.nanoTime();
           System.out.println(stop - start)
   }}
```

Version 3: C

- Running time = **1,156** seconds ≈ **19** minutes
- **2 X** faster than Java codes
- ≈ **18 X** faster than Python codes

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#define n 4096
double A[n][n], B[n][n], C[n][n];

int main(int argc, const char *argv[] ) {
        …. // init A, B, C
        struct timeval start, end;
        gettimeofday(&start, NULL);
        for (int i  = 0; i < n; i++) {
            for (int j  = 0; j < n; j++) {
                for (int k  = 0; k < n; k++) {
                    C[i][j] += A[i][k] * B[k][j]; }}}
         gettimeofday(&end, NULL);
   return 0;
}
```

Comparison Python, Java, C Implementation

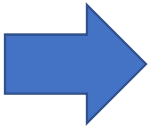
version	Implementation	Running time (sec.)	Absolute Speedup	GFLOPS	Percent of Peak
1	Python	21041.67	1.00	0.007	0.001
2	Java	2387.32	8.81	0.058	0.007
3	C	1155.77	18.2	0.119	0.014

- Why is Python codes so slow against C ?
  - Python is interpreted
  - C is compiled directly to machine code.
  - Java is compiled to byte-code, which is then interpreted and just-in-time (JIT) compiled to machine codes.

More Optimization – Loop Order

- We can change the order of the loops in MM program without affecting its correctness

```
for (int i  = 0; i < n; i++)
   for (int j  = 0; j < n; j++)
      for (int k  = 0; k < n; k++)
         C[i][j] += A[i][k] * B[k][j];
```



```
for (int i  = 0; i < n; i++)
   for (int k  = 0; k < n; k++)
      for (int j  = 0; j < n; j++)
         C[i][j] += A[i][k] * B[k][j];
```

Does loop order affect the performance of MM program ?

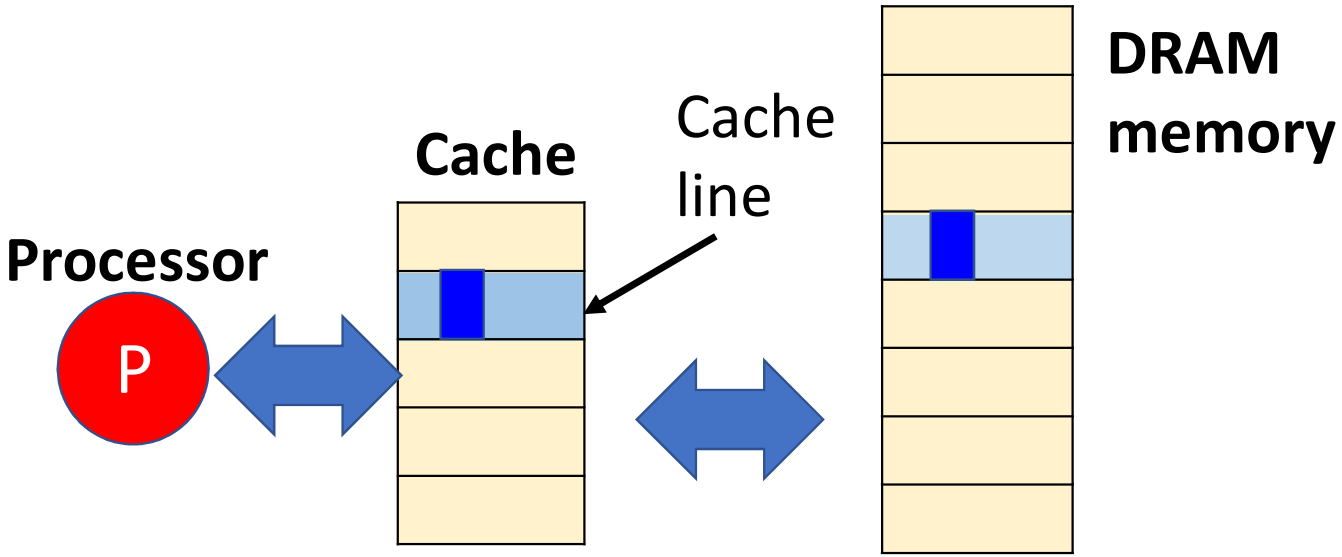
More Optimization – Loop Order

- Loop order affects running by a factor of **18** !!
- **Why different loop orders can make such a big change ?**

Loop Order (outer to inner)	Running time (sec.)
i, j, k	1155.77
i, k, j	177.68
J, i, k	1080.61
J, k, i	3056.63
k, i, j	179.21
k, j, i	3032.82

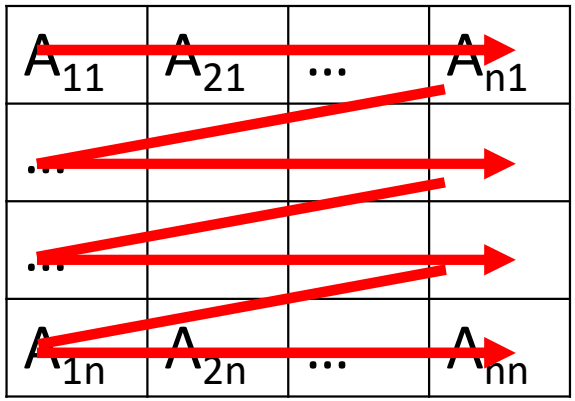
Hardware Caches

- The cache sits near the processor
- The cache reduces the memory access latency of reused data
  - **Cache hits** – accesses to data in cache – fast
  - **Cache misses** – accesses to data not in cache – slow
  - **Cache lines** – data is stored as contiguous blocks in cache



Memory Layout of Matrices

- In this matrix-multiplication code, matrices are laid out in memory in **row-major order**.



Matrix

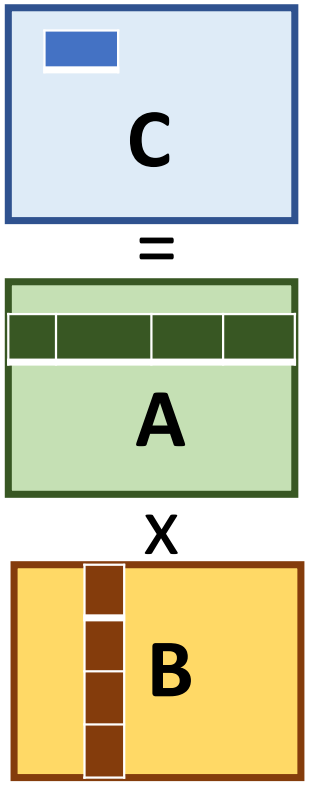
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6

Memory	Row 1	Row 2	Row 3	Row 4	Row 5

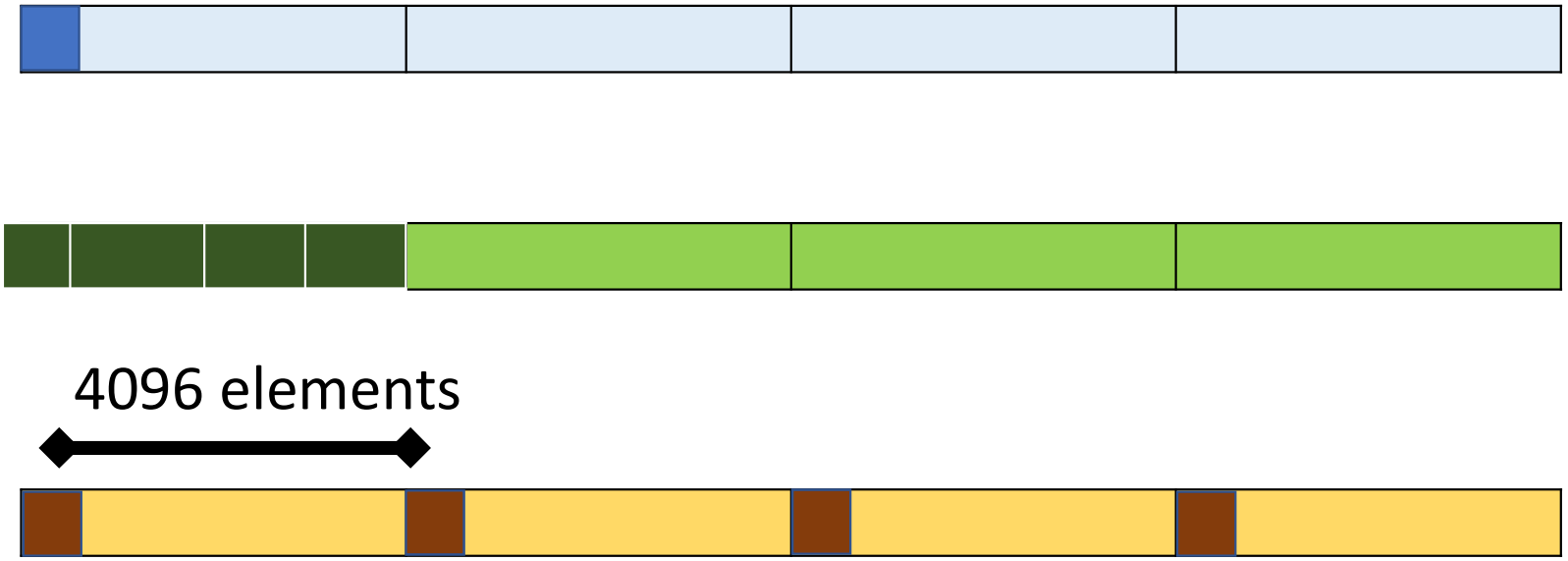
Access Pattern of Matrix Multiplication

Running time:
1155.77 sec.

```
for (int i  = 0; i < n; i++)
   for (int j  = 0; j < n; j++)
      for (int k  = 0; k < n; k++)
         C[i][j] += A[i][k] * B[k][j];
```



i = 0, j = 0, k = [0 … 4095]

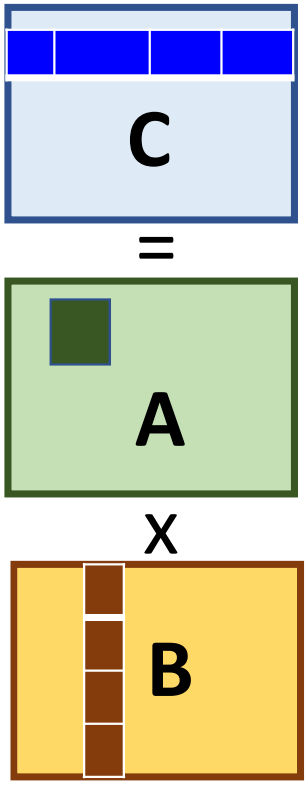


In-memory layout

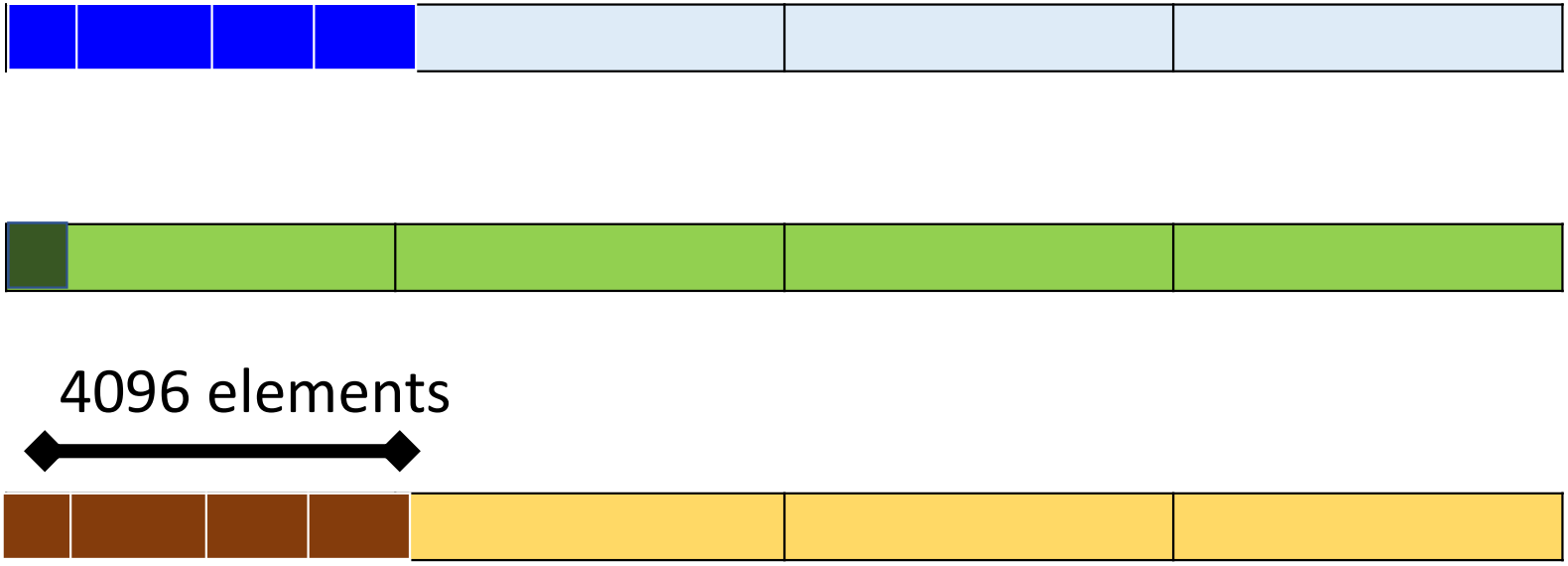
Memory Access Pattern for Order i, k, j

Running time:
177.68 sec.

```
for (int i  = 0; i < n; i++)
   for (int k  = 0; k < n; k++)
      for (int j  = 0; j < n; j++)
         C[i][j] += A[i][k] * B[k][j];
```



i = 0, k = 0, j = [0 … 4095]

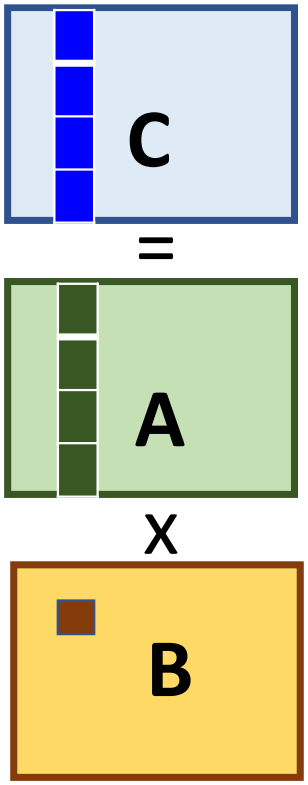


In-memory layout

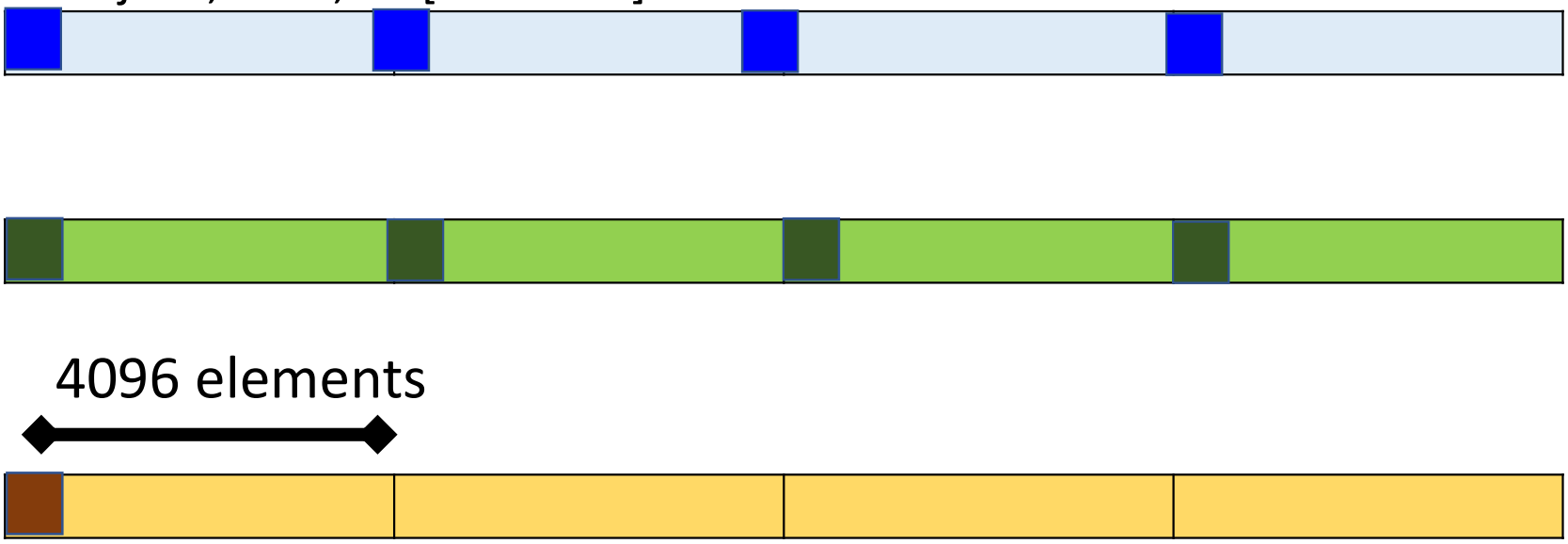
Memory Access Pattern for Order j, k, i

Running time:
3056.63 sec.

```
for (int j  = 0; j < n; j++)
   for (int k  = 0; k < n; k++)
      for (int i  = 0; i < n; i++)
         C[i][j] += A[i][k] * B[k][j];
```



j = 0, k = 0, i = [0 … 4095]



In-memory layout

Performance of Different Order

- The running time of MM is associated with cache miss rate

Loop Order (outer to inner)	Running time (sec.)	Last-level-cache miss rate
i, j, k	1155.77	7.7%
i, k, j	177.68	1.0%
J, i, k	1080.61	8.6%
J, k, i	3056.63	15.4%
k, i, j	179.21	1.0%
k, j, i	3032.82	15.4%

More optimization – Compiler Optimization

- Clang compiler includes a collection of optimization switches
- People can specify a switch to the compiler to perform code optimization

Opt. Level	Meaning	Time (sec.)
-O0	Do not optimize	177.54
-O1	Optimize	66.24
-O2	Optimize even more	54.63
-O3	Optimize yet more	55.58

Version 5: Optimization Flags

- We only reach 0.3% of the peak performance of the machine.
- How to squeeze more performance juice of MM ?

version	Implementation	Running time (sec.)	Absolute Speedup	GFLOPS	Percent of Peak
1	Python	21041.67	1.00	0.007	0.001
2	Java	2387.32	8.81	0.058	0.007
3	C	1155.77	18.2	0.119	0.014
4	Interchange loops	177.68	118.42	0.774	0.093
5	Optimization flag	54.63	385.17	2.516	0.301

Version 6: Parallel Loops on Multi-core

- Intel E5 has 9 cores per chip, and a AWS machine has 2 chips
- We can run MM on 18 parallel cores concurrently
- We can parallelize one of these 3 loops
- **OpenMP or Cilk** library can help us to run MM in parallel

```
Parallel_for (int i  = 0; i < n; i++)
   for (int j  = 0; j < n; j++)
      for (int k  = 0; k < n; k++)
         C[i][j] += A[i][k] * B[k][j];
```

Each iteration i is distributed to parallel cores

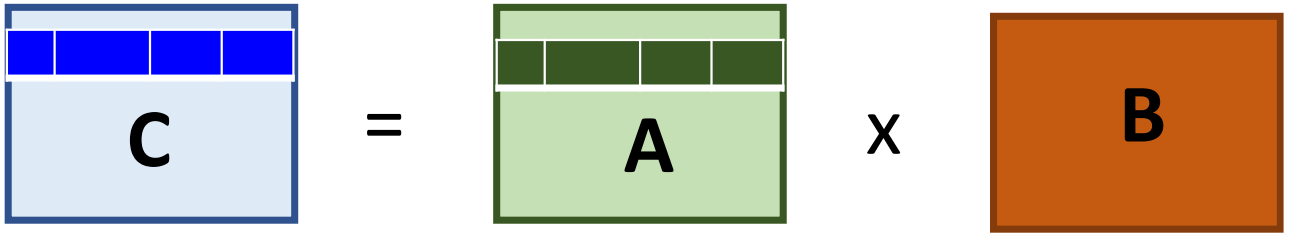
Version 6: Parallel Loops on Multi-core

- Using parallel loops achieves **~ 18 X** speed up on 18 cores
- **Disclaimer**: Not all code is so easy to parallelize effectively
- However, we still use 5% of peak performance so far !!

version	Implementation	Running time (sec.)	Absolute Speedup	GFLOPS	Percent of Peak
1	Python	21041.67	1.00	0.007	0.001
2	Java	2387.32	8.81	0.058	0.007
3	C	1155.77	18.2	0.119	0.014
4	Interchange loops	177.68	118.42	0.774	0.093
5	Optimization flag	54.63	385.17	2.516	0.301
6	Parallel loops	3.04	6921.6	45.21	5.408

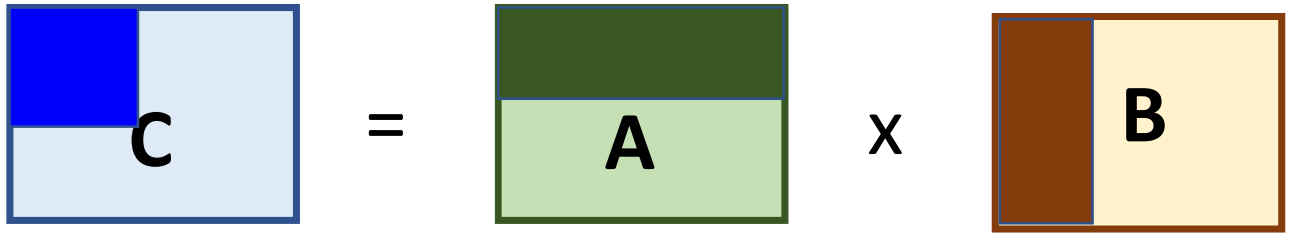
Version 7: Tiling on Cache

- Restructure the computation to reuse data in the cache as much as possible
- How many memory accesses of fully compute 1 row of C ?
  - **4096 x 1 = 4096** writes to C
  - **4096 x 1 = 4096** reads from A
  - **4096 x 4096** reads from B
  - **16,785,408** memory access in total



Version 7: Tiling on Cache

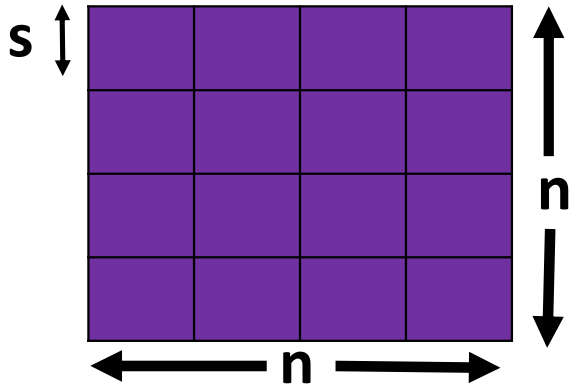
- Partition data in blocks which are reused in the cache
- How many memory accesses to compute 64 x 64 block of C ?
  - **64 x 64 = 4096** writes to C
  - **64 x 4096 = 262,144** reads from A
  - **4096 x 64 = 262,144** reads from B
  - **528,384** memory accesses in total



Tiled Matrix Multiplication

```
Parallel_for (int ih  = 0; ih < n; ih += s)
  Parallel_for (int jh  = 0; jh < n; jh += s)
    for (int kh  = 0; kh < n; kh += s)
      for (int il  = 0; il < s;  ++ il)
        for (int kl  = 0; kl < s;  ++ kl)
          for (int jl  = 0; jl < s; ++ jl)
            C[ih + il][jh + jl] += A[ih + il][kh + kl] * B[kh + kl][jh + jl];
```

The value of **s** (**the size of tiling block**) affect the performance.



Tiling block size	Running time (s)
4	6.74
8	2.76
16	2.49
32	1.74
64	2.33

Version 7: Tiling

version	Implementation	Running time (sec.)	Absolute Speedup	GFLOPS	Percent of Peak
1	Python	21041.67	1.00	0.007	0.001
2	Java	2387.32	8.81	0.058	0.007
3	C	1155.77	18.2	0.119	0.014
4	Interchange loops	177.68	118.42	0.774	0.093
5	Optimization flag	54.63	385.17	2.516	0.301
6	Parallel loops	3.04	6921.6	45.21	5.408
7	Tiling	1.74	12,092.9	72.782	9.184

Implementation	Cache reference (millions)	L1-D cache misses (millions)
Parallel loop	104,090	17,220
Tiling	64,690	11,777

The tiling reduces **38%** cache reference and **32%** cache misses

Further Optimization

version	Implementation	Running time (sec.)	Absolute Speedup	GFLOPS	Percent of Peak
1	Python	21041.67	1.00	0.007	0.001
2	Java	2387.32	8.81	0.058	0.007
3	C	1155.77	18.2	0.119	0.014
4	Interchange loops	177.68	118.42	0.774	0.093
5	Optimization flag	54.63	385.17	2.516	0.301
6	Parallel loops	3.04	6921.6	45.21	5.408
7	Tiling	1.74	12,092.9	72.782	9.184
8	Parallel divide-and-conquer	1.30	16,197	105.722	12.646
9	Vectorization	0.70	30272	196.341	23.486
10	AVX intrinsics	0.39	53292	352.408	41.677

We can gain **53,292 X** faster than naïve Python codes by software optimization !

What's Left ?

- Transistors not getting much better
- Power budget not getting much higher
- One inefficient processor/chip to N efficient processors/chip
- Only path left is **Domain Specific Architectures**
  - Just do a few tasks, but extremely well

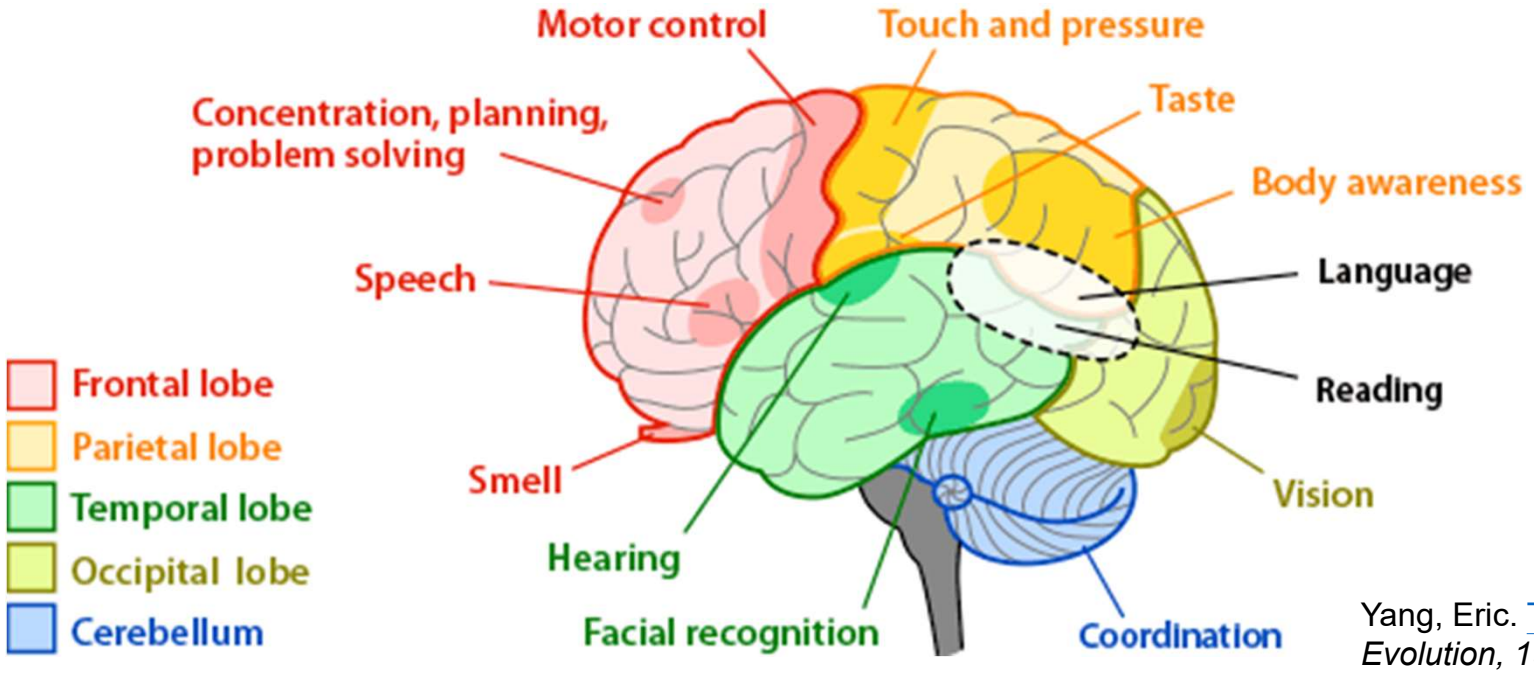
John Hennessy, "The Future of Microprocessors", 2017

Speed up Machine Learning through Domain-Specific Accelerator

Uncover Your Brain

2400 kcal/24 hr = 100 kcal/hr = 27.8 cal/sec = 116.38 J/s = 116 W
20% x 116 W = 23.3 W

- The human-being brain comprises different areas (accelerators)
- An adult brain only consumes about 23 W a day !! (Yang)

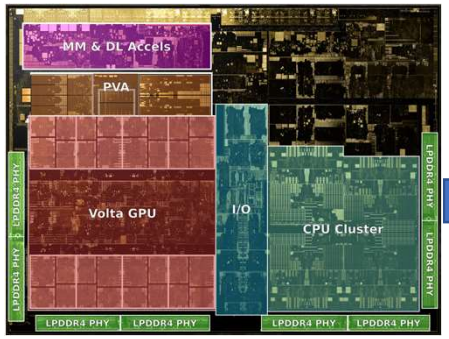


Yang, Eric. Think Dinner. *Mac Evolution, 1998*

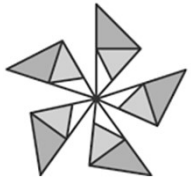
https://askabiologist.asu.edu/sites/default/files/resources/articles/nervous_journey/brain-regions-areas.gif

Learn from Human Being's Brain

- Designing "**Accelerators**" to boost up **Machine Learning**

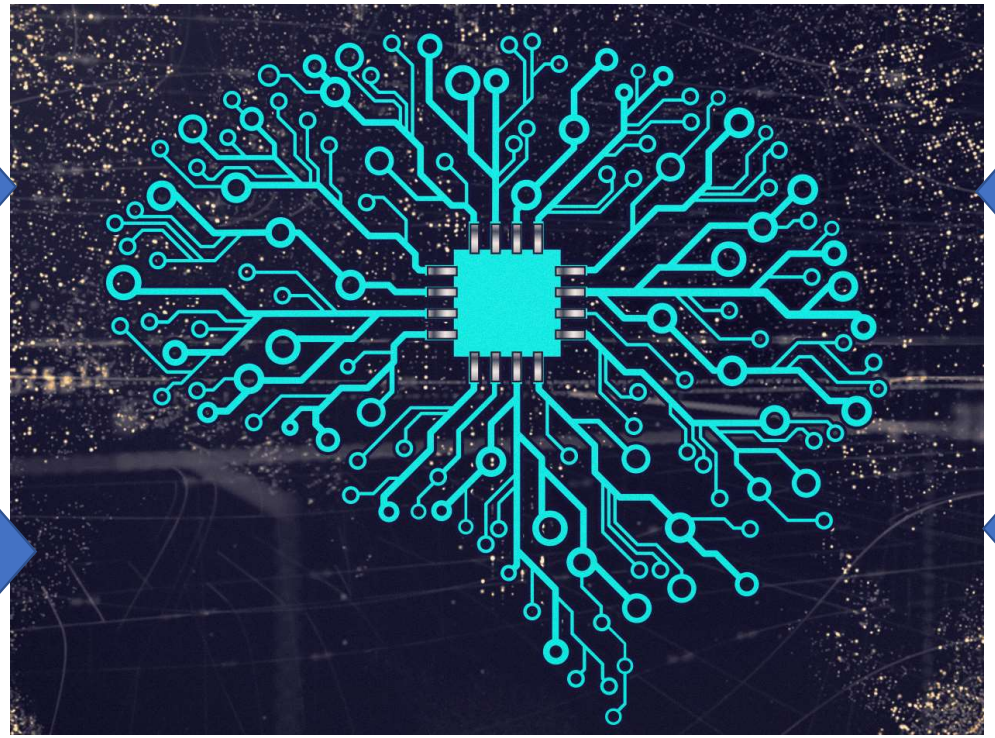


Micro-architecture



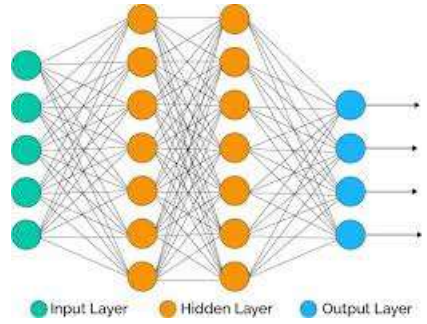
ONNX RUNTIME

System Software



tvm

Program/Language



Algorithm

https://upload.wikimedia.org/wikipedia/commons/2/27/Artificial_Neural_Network_with_Chip.png

Domain Specific Architecture (DSAs)

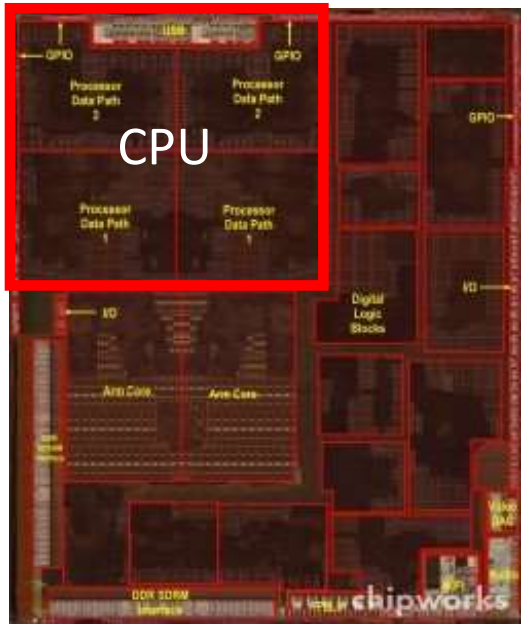
- Achieving higher performance by tailoring characteristics of domain applications to the architecture
  - Need domain-specific knowledge to work out good DSAs
  - Domain Specific Languages (DSLs) + DSAs (not strict ASIC)
  - Specialize to **a domain** of **many applications**
- Examples
  - GPU for computer 3D graphics, virtual reality
  - Neural processing unit (NPU) for machine learning
  - Visual processing unit (VPU) for image processing

Domain Specific Languages (DSL)

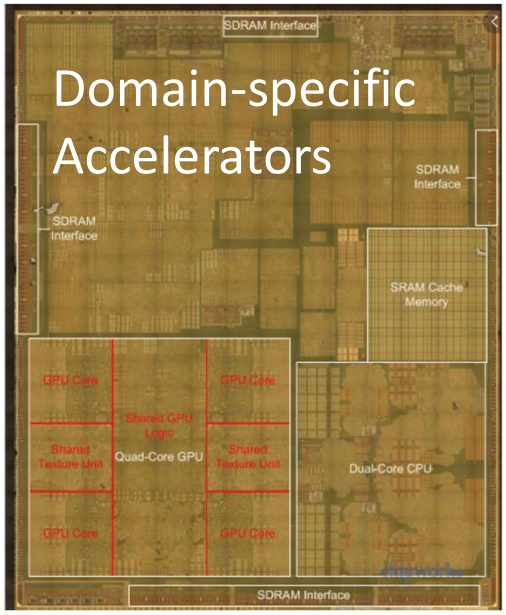
- DSLs target specific operations on a domain of applications
- Need vector, matrix or sparse matrix operations
- DSLs tailors for these operations
  - OpenGL, TensorFlow, Halide
- Compilers are important if DSLs are architecture-independent
  - Translate, schedule, map ISAs to right DSAs

Where is Domain-Specific Accelerators

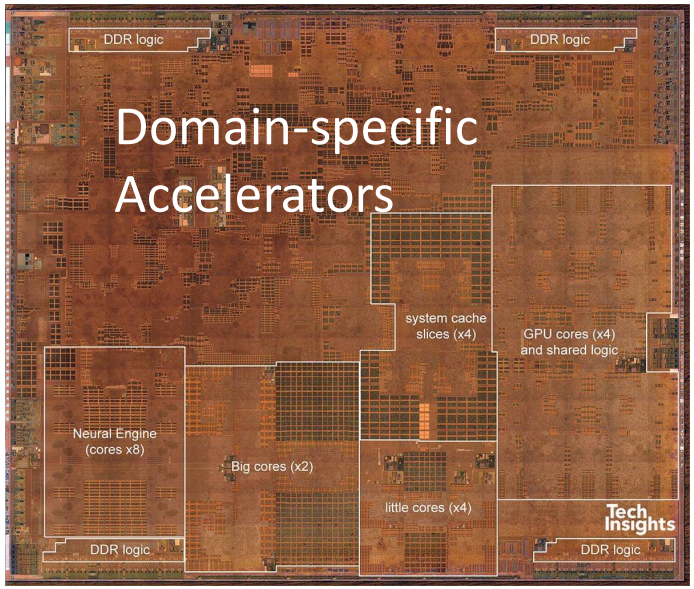
- Domain-Specific Accelerators are everywhere



2010 Apple A4
65 nm TSMC 53 mm^2
4 accelerators



2014 Apple A8
20 nm TSMC 89 mm^2
28 accelerators



2019 Apple A12
7 nm TSMC 83 mm^2
42 accelerators

https://edge.seas.harvard.edu/files/edge/files/alp.pdf

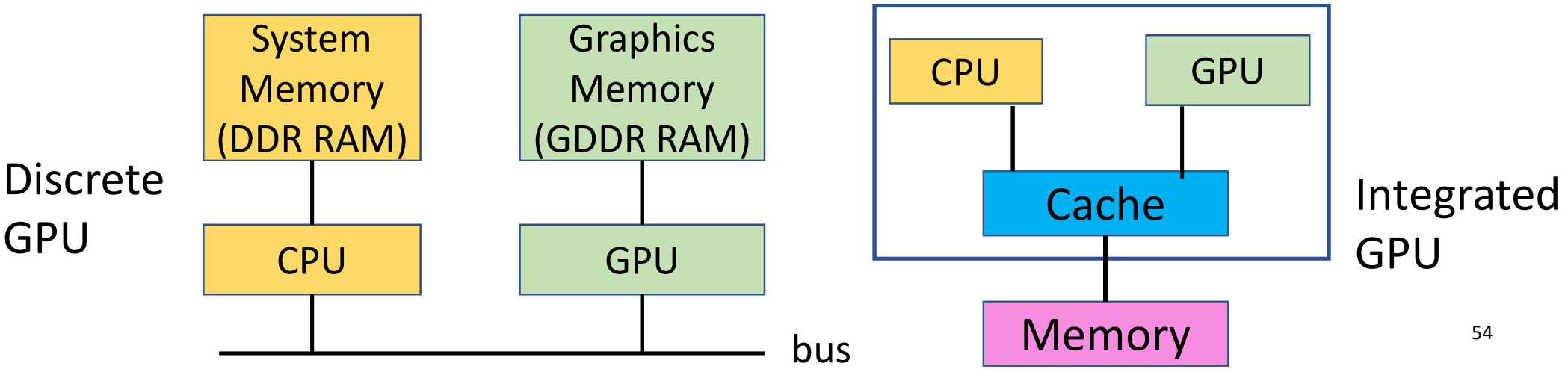
Why DSAs can win ?

- More effective parallelism for a specific domain
  - SIMD vs. MIMD
  - VLIW vs. Speculative, out-of-order
- More effective use of memory bandwidth
  - User controlled vs. caches
- Eliminate unneeded accuracy (Quantization)
  - Lower FP/INT data precision (32 bit integers -> 8 bit integers)
- Increase the hardware utilization
  - Reduce the idle time on pipelining and LD/ST

https://iscaconf.org/isca2018/docs/HennessyPattersonTuringLectureISCA4June2018.pdf

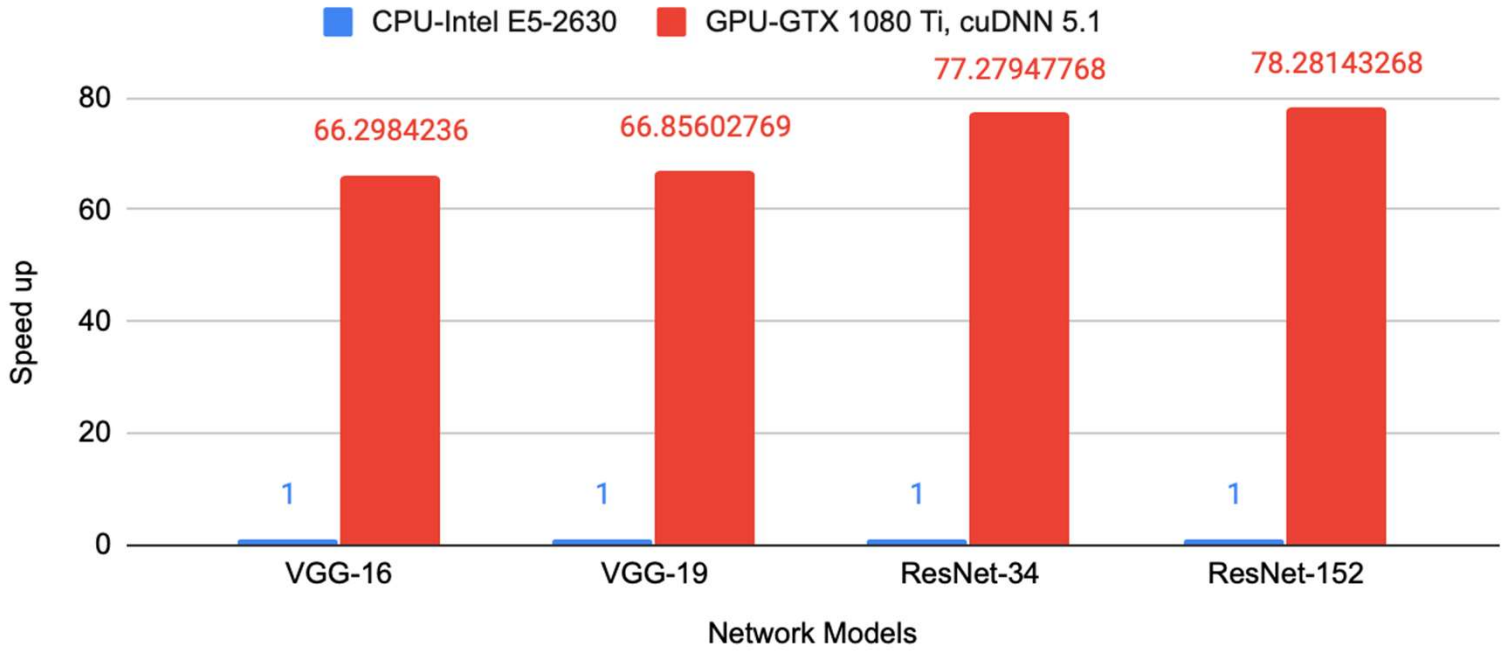
What is GPU?

- GPU = Graphics Processing Units
- Accelerate computer graphics rendering and rasterization
- Highly programmable (OpenGL, OpenCL, CUDA, HIP etc..)
- Why does GPU use GDDR memory?
  - DDR RAM -> low latency access, GDDR RAM -> high bandwidth



CPU vs GPU Training Time Comparison

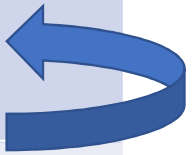
- Normalized Training time on CPU and GPU (CPU has 16 cores, 32 threads)
- Why the model training on GPUs is much faster than on the CPU?



https://github.com/jcjohnson/cnn-benchmarks

CPU vs GPU

	Cores	Clock Speed	Memory	Price	Throughput
CPU (Intel Core i7-7700k)	4	4.2 GHz	DDR4 RAM	$385	~540 GFLOPs F32
GPU (Nvidia RTX 3090 Ti)	10496	1.7 GHz	DDR6 24 GB	$1499	36 TFLOPs F32



6.67X

CPU: A **small** number of **complex** cores, the clock speed of each core is high, great for sequential tasks

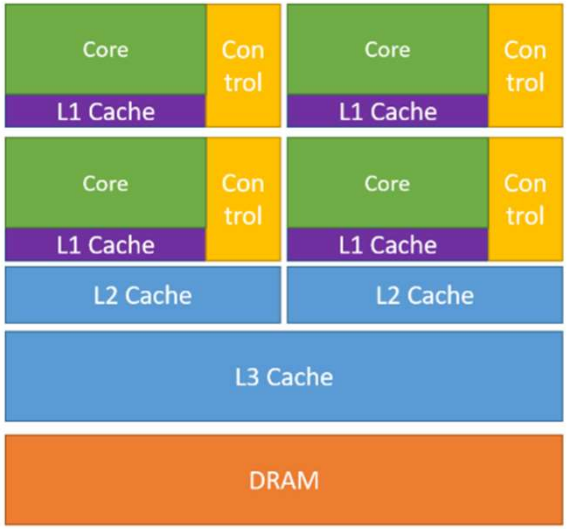
GPU: A **large** number of **simple** cores, the clock speed of each core is low, great for parallel tasks

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

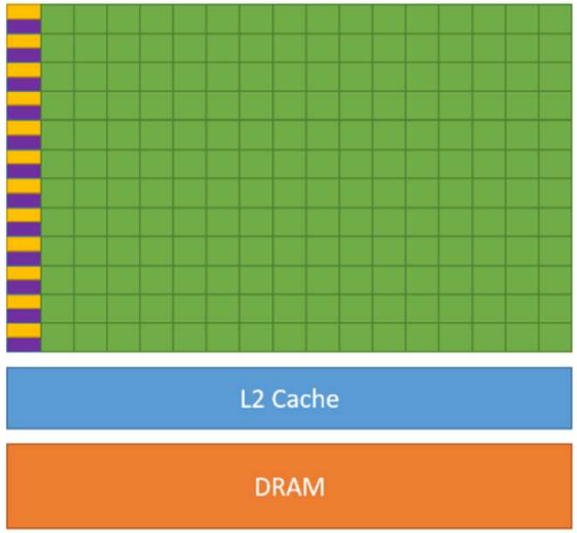
Why do we use GPU for computing ?

- What is difference between CPU and GPU?
  - GPU uses a large portion of silicon on the computation against CPU
  - GPU (2nJ/op) is more energy-efficient than CPU (200 pJ/op) at peak performance
  - Need to map applications on the GPU carefully (Programmers' duties)

CPU

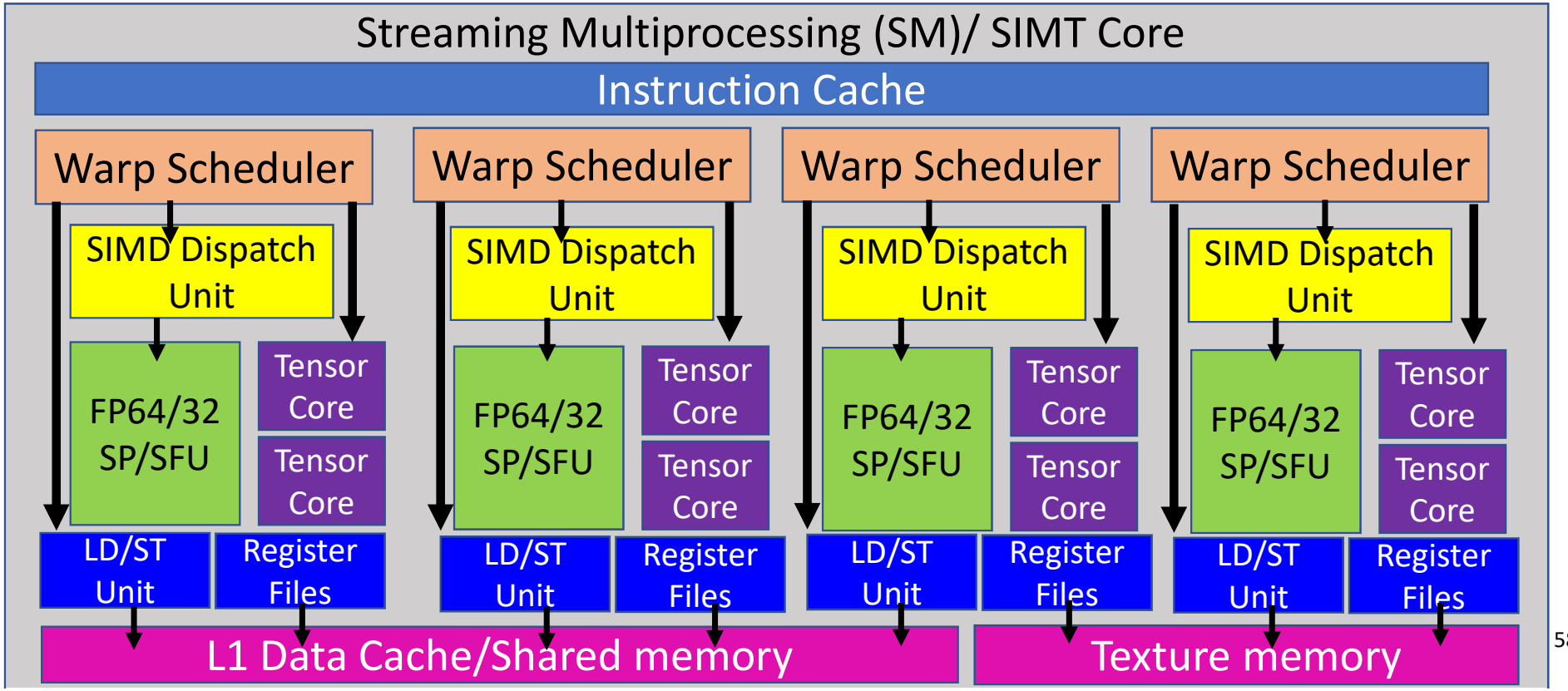


GPU



What is Tensor Core on GPU?

- Execute 4 x 4 x 4 matrix multiplication and addition in one cycle (D = A x B + C)



Zhu et.al., MICRO 2019

Why do we need Tensor Core on GPUs ?

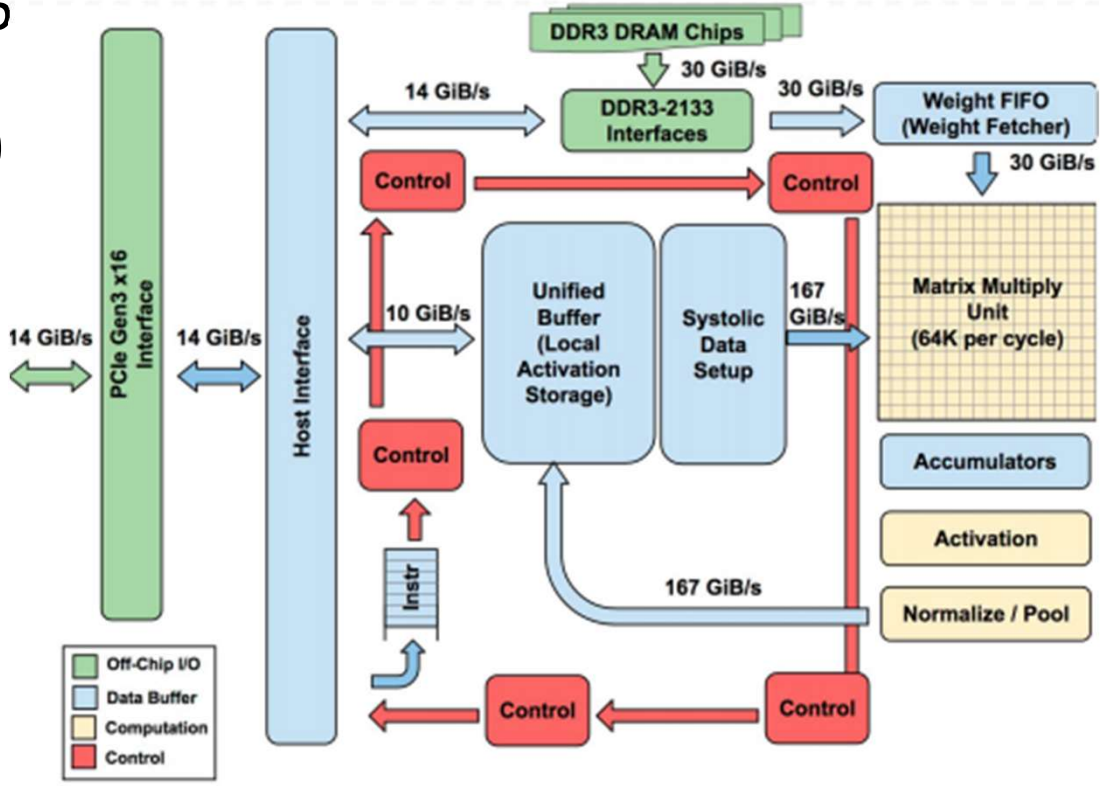
- Higher throughput for GEMM ?
  - A CUDA (SIMT) core offers 1 single precision multiply-and-accumulate operation per GPU cycle
  - Tensor core can multiply two 4 x 4 F16 matrices and add the multiplication product F32 matrix per GPU cycle
  - Tensor core can achieve **125 Tflops/s** vs **15.7 Tflops/s** for the single precision operation
  - Domain-specific Accelerator within the GPU

Story in Tensor Processing Unit (TPU)

- If people use DNN speech recognition service 3 mins per day
- Need to double Google's data center to meet this requirement
- Why not quickly a customized ASIC for inference ?
  - Need to **10 X** faster than GPUs
  - Must run existing apps developed for CPUs and GPUs
- Very short development time on TPU
  - Only take **15 months** for architecture and compiler invention, hardware design, build, test, deploy

Details in TPU v1

- **The Matrix Unit**: 64K (256 x 256 8 bit INT multiply-accumulate
- Peak: 92T ops = 65536 x 2 x 700 MHz clock rate
- 4 MiB of 32-bit **Accumulator** collects 16 bit products
- Hardware **activation logics**
- 2.4 MiB **on-chip Unified Buffer** (Intermediate results)
- 3.5 X as much on-chip memory vs GPU
- 8 GiB **off-chip weight DRAM**



Jouppi et. al, ISCA 2017

Performance Comparison

Processor	mm^2	Clock(MHz)	TDP (Watts)	Memory (GB/sec)	Peak TOPS/chip	
					8 b INT	32b FP
CPU: Haswell (18 core)	662	2300	145	51	2.6	1.3
GPU: Nvidia K80	561	560	150	160	--	2.8
TPU	<331	700	75	34	91.8	--

K80 and TPU in 28 nm process; Haswell fabbed in Intel 22nm process

Jouppi et. al, ISCA 2017

Why TPU can Win ?

- Large matrix multiply unit
- Substantial software-controlled on-chip memory
- Data Quantization (8-bit INT)
- Parallelism on the hardware instead of Thread-level parallelism on GPUs
- What else ?

My Research Work

Introducing Myself

"Hiring graduate and under-graduate students"

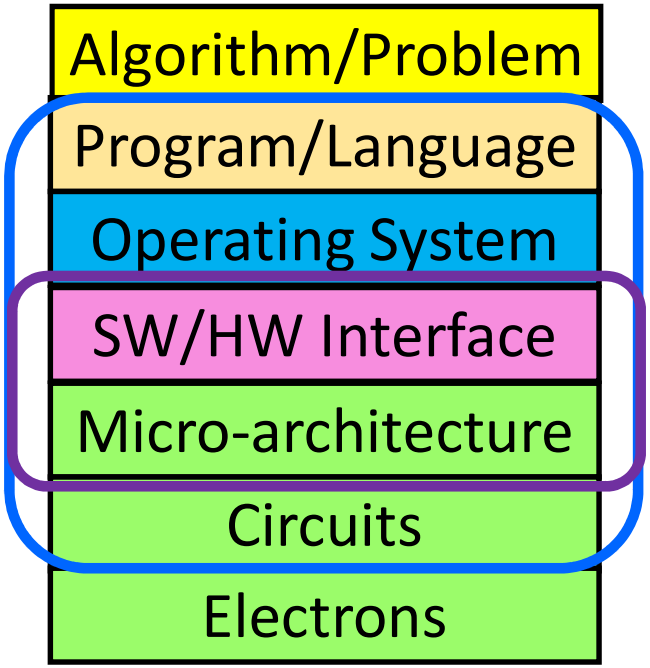
- Lecturer: Tsung Tai Yeh
  - E-mail: ttyeh@cs.nctu.edu.tw
  - Office: EC 707
  - Research topics:
    - Computer architecture
    - Computer systems
    - Memory and storage systems
    - Domain-specific accelerators (GPU, Neural Processing Units)

My Research Work

- Performance Engineering
  - How to accelerate your applications by using software + hardware techniques ?

Computer Architecture (Expanded view)

Hardware-Software Co-Design (Algorithms to Devices)



Computer Architecture (Narrow view)

Specialize on designs of SW/HW inference and Micro-architecture

Conclusion

- Speed up “machine learning”
- Need **Software Opt. + Accelerator**
- **Sky is the limit**
- May you have a beautiful mind to explore the beautiful future
- ttyeh@cs.nctu.edu.tw

Thank You!!
Q & A