IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

The distributed (and parallel) programming: it's all about scalability

Xingda Wei, Yubin Xia

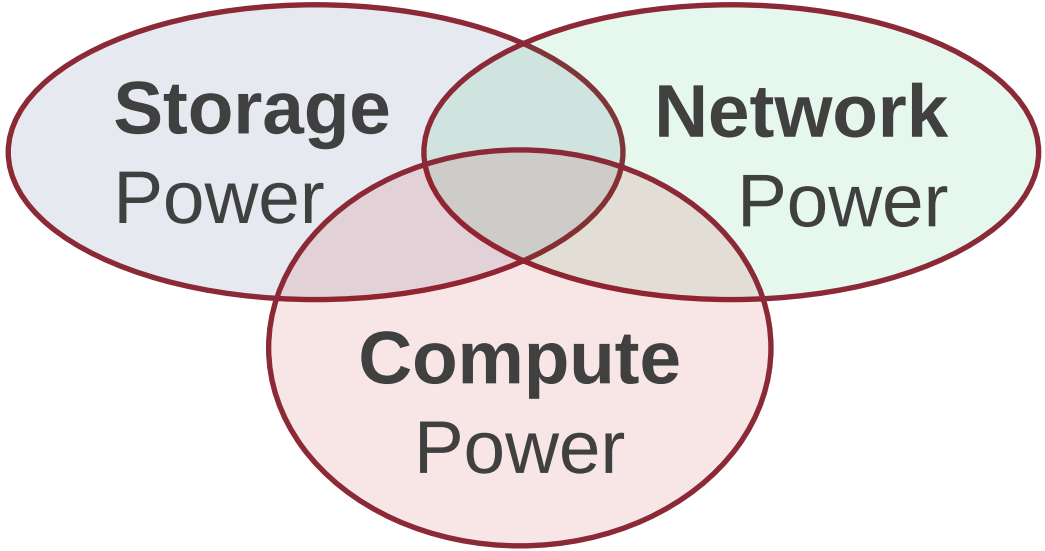
IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Credits: AI-sys Sp22, CMU15-418

Systems: storage, compute and network

Medical Imaging | Speech AI | Customer Service | Recommenders | Physics ML | Communications | Video Analytics | Logistics | Conversational AI | Robotics | Autonomous Vehicles | Cybersecurity



Review: Scalable web apps are everywhere: AI era

Essentially, also a web APP

- Pre-AI era: request handling = order an item
- AI era: request handling = use model to do an inference

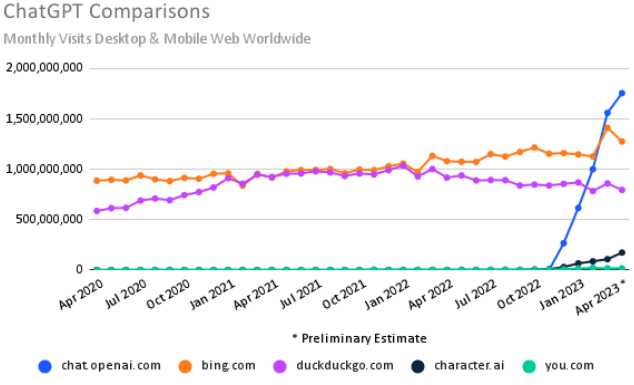
The same system requirements

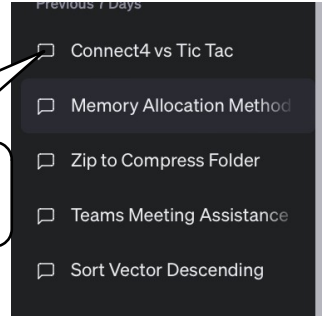
- High request rate
- Massive data: both for training & for serving history data

New requirements

- Huge computation power required
- **According to GPT (⏪), an inference of GPT ~= sorting 100,000,000 numbers**

ChatGPT





Per-user sessions

Question: given an AI workload, e.g., model training, how many computation does it need?

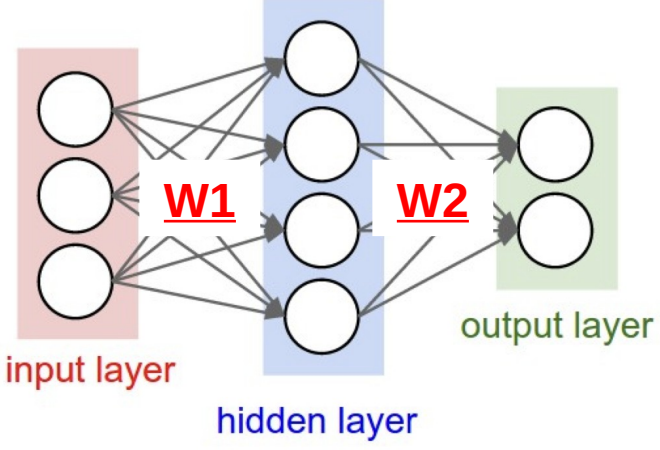
The scale of computation

Motivating example: the process of large AI models

- E.g., Current AI models are constructed via neural network (NN)

Question: given a simple NN network, how many computations are required to trained it?

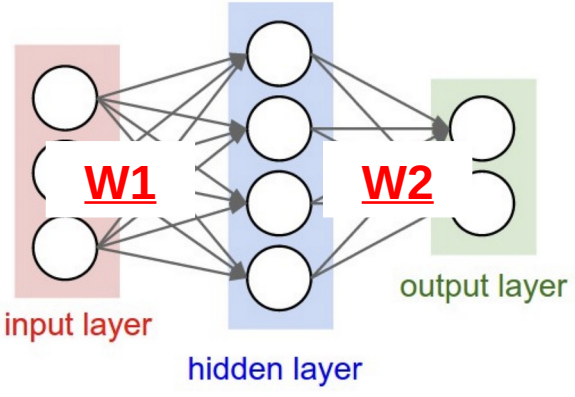
- Suppose the NN has **W (W1 + W2)** parameters



Motivating example: AI model training

Goal: , where y is the output

- E.g., in computer vision tasks, y is the category of a given image (x)



Where // simplified to one hidden layer

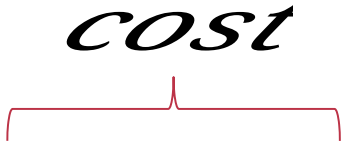
We compute W1 & W2 such that the NN can recognize a photo ⏪

- The process is typically called training

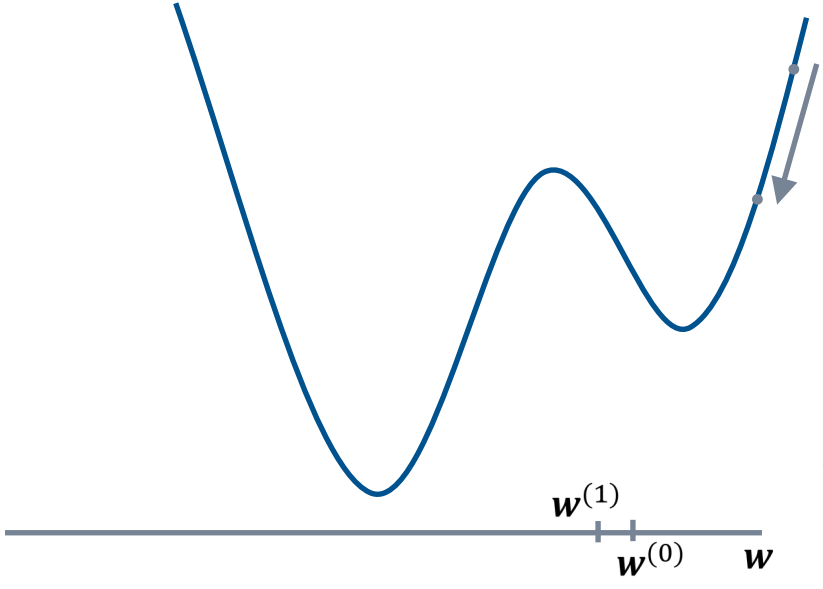
Supervised learning & gradient decent & its computation requirements

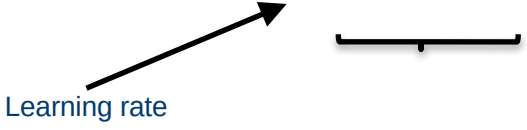
Supervised learning

- We have some known **n** as
- We have a cost function (L), e.g.,
- **Goal**: find W(W1 and W2) to minimize , i.e.,



How? Gradient decent

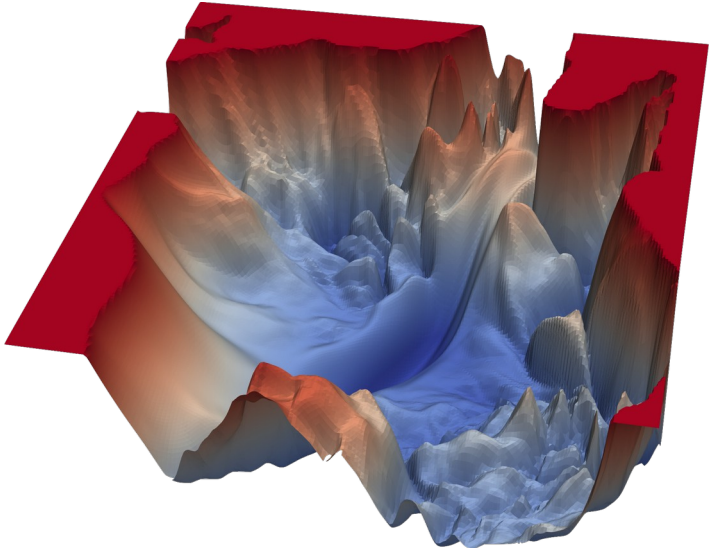




Two key elements:

- The computed gradient: the direction
- The learning rate: how big a step do we take?

Stochastic Gradient Descent (SGD)

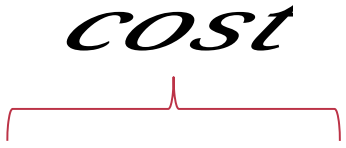


Two key elements:

- The computed gradient: the direction
- The learning rate: how big a step do we take?

Supervised learning & gradient decent & its computation requirements

Supervised learning



- We have some known **n** as
- We have a cost function (L), e.g.,
- Goal: find W(W1 and W2) to minimize , i.e.,

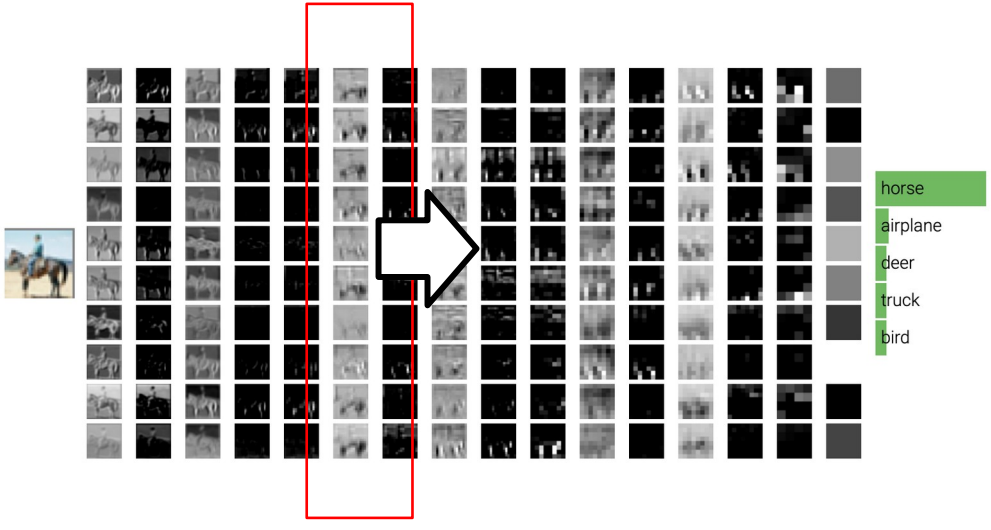
Stochastic Gradient decent

- Initialize W (W1 and W2) as some random inputs (should be non-zero ⏪)
- For a batch of update W as
- Update iteratively until find the (possible) local minimal of W

Back to our question: given a simple NN network, trained via supervised learning with SGD, how many computations are required ?

Calculation one: forward path (for loss function)

For each layer, calculate (: matrix multiplication)



Since model is calculated layer-by-layer, we just need to focus on the calculation of one layer

Calculation one: forward path (for loss function)

For each layer, calculate (: matrix multiplication)

- W:
- X:
- k: feature size; m : activation number; B: batch size of X

Approximate calculation

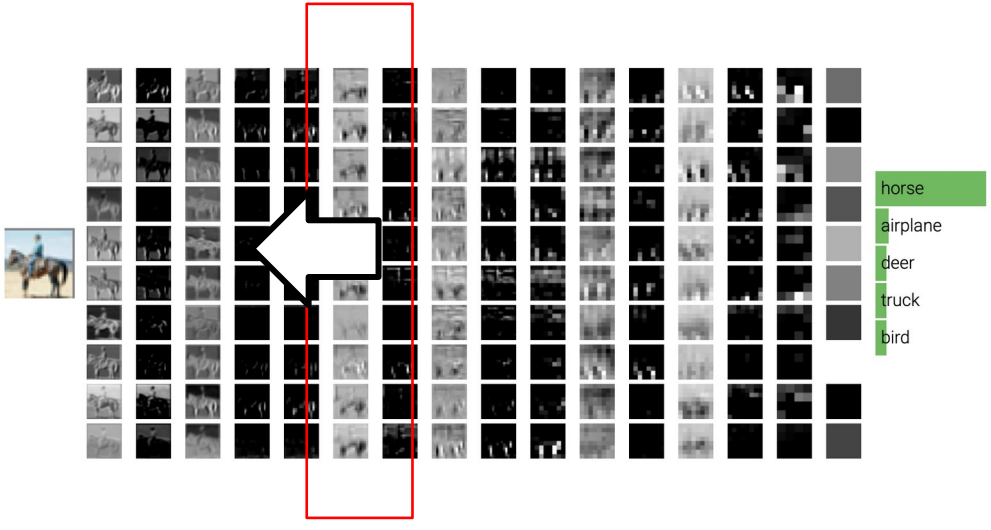
- $(2k-1) \times m \times B \sim= 2 \times \text{Size}(W) * B$

Considering all the layers

- 2 x Size(W0) * B + 2 * Size (W1) * B + … = 2 * B * # parameters

Calculation two: backward path (for dW & dX)

For each layer, calculate



Since model is calculated layer-by-layer, we just need to focus on the calculation of one layer

- Why dX? dX is used to calculate the dW of the previous layer

Calculation two: backward path (for dW & dX)

For each layer, calculate the dX and dW (: matrix multiplication)

- ()
- ()

dX calculation: (2m – 1) × k × B ~= 2 x Size(W) * B

dW calculation (the same): (2B – 1) × m ×k ~= 2 x Size(W) * B

Considering all the layers

- 2 * 2 x Size(W0) * B + 2 * 2 * Size (W1) * B + … = 4 * B * # parameters

Question: given an AI (model training) workload, how many computation does it need?

~= 6 * #parameters * Batch size

(Approximate) computation for NN like ChatGPT?

Known fact: GPT-4 has 1.76 trillion parameters [1]

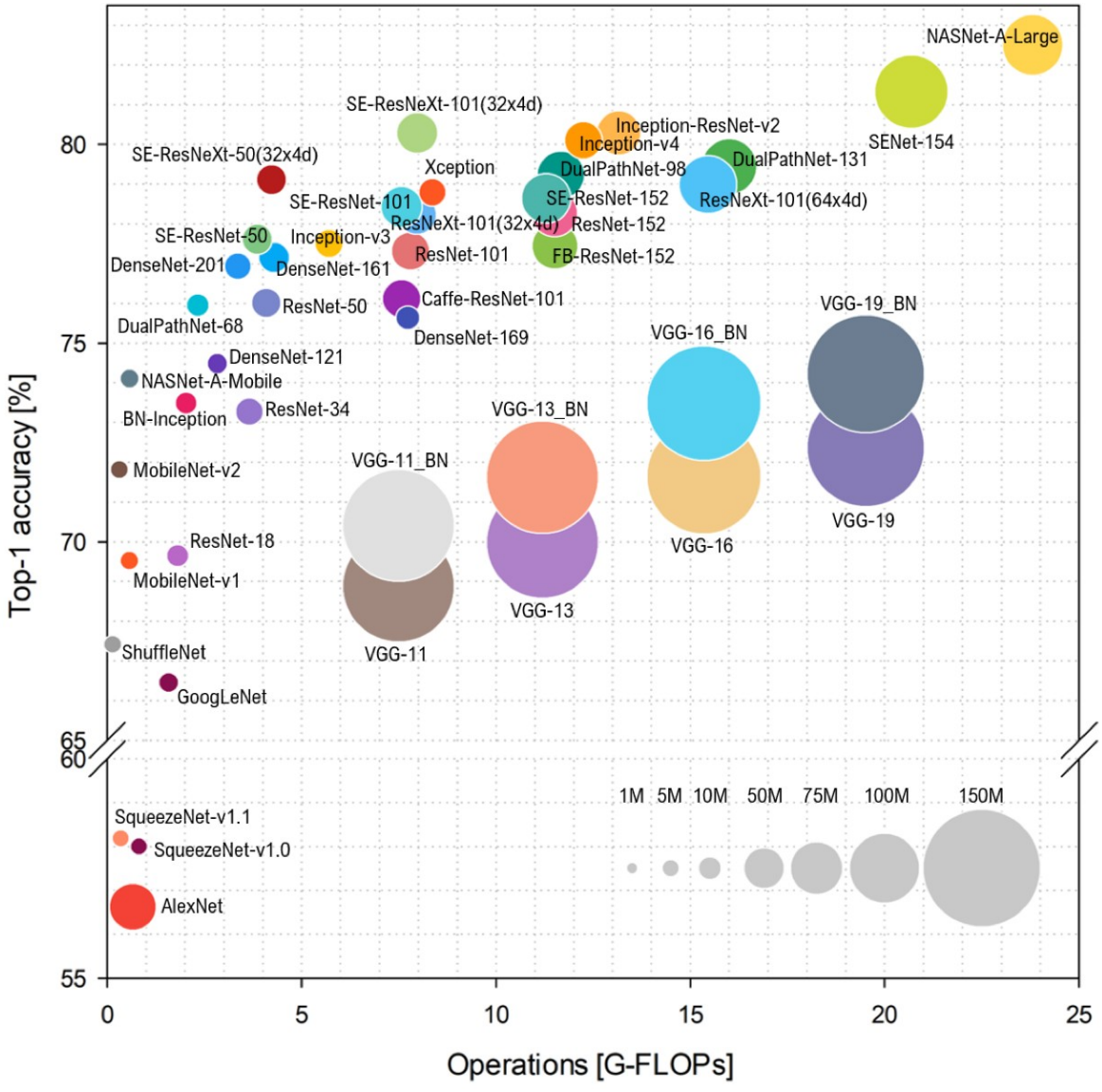
- This is 1,760,000,000,000 parameters
- So, this is 10,560,000,000,000 calculations for a single input of **a single iteration**!!

What are the computation capabilities of nowadays devices (e.g., A100)?

- 19.5 TFLOPS = 19,500,000,000,000 (FP32) float point per second
- Basically, it needs 30 seconds for an A100 GPU to finish an iteration **in the optimal case**

We need a powerful computation device for the AI!

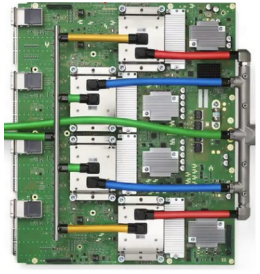
[1] https://the-decoder.com/gpt-4-has-a-trillion-parameters/


NASNet-A-Large
SE-ResNeXt-101(32x4d)
Inception-ResNet-v2
Inception-v4
SENet-154
SE-ResNeXt-50(32x4d)
Xception
DualPathNet-98
DualPathNet-131
SE-ResNet-101
SE-ResNet-152
ResNeXt-101(64x4d)
ResNeXt-101(32x4d)
ResNet-152
SE-ResNet-50
Inception-v3
ResNet-101
FB-ResNet-152
DenseNet-201
DenseNet-161
Caffe-ResNet-101
ResNet-50
VGG-19_BN
DualPathNet-68
DenseNet-169
VGG-16_BN
DenseNet-121
NASNet-A-Mobile
BN-Inception
ResNet-34
VGG-13_BN
MobileNet-v2
VGG-11_BN
VGG-19
ResNet-18
VGG-16
MobileNet-v1
VGG-13
ShuffleNet
VGG-11
GoogLeNet
1M 5M 10M 50M 75M 100M 150M
SqueezeNet-v1.1
SqueezeNet-v1.0
AlexNet
Top-1 accuracy [%]
80
75
70
65
60
55
0
5
10
15
20
25
Operations [G-FLOPs]

Devices available for computation

Spectrum of computation device available

Programmability ←



Intel i5-9600K, single core: 6.3GFLOPS Multiple cores: 37.7GFLOPS

Mate60 GPU 2.3 TFLOPs

Apple M2 GPU 3.6 TFLOPs

NVIDIA A100 GPU 19.5 TFLOPs

Google TPUv4 275 TFLOPS

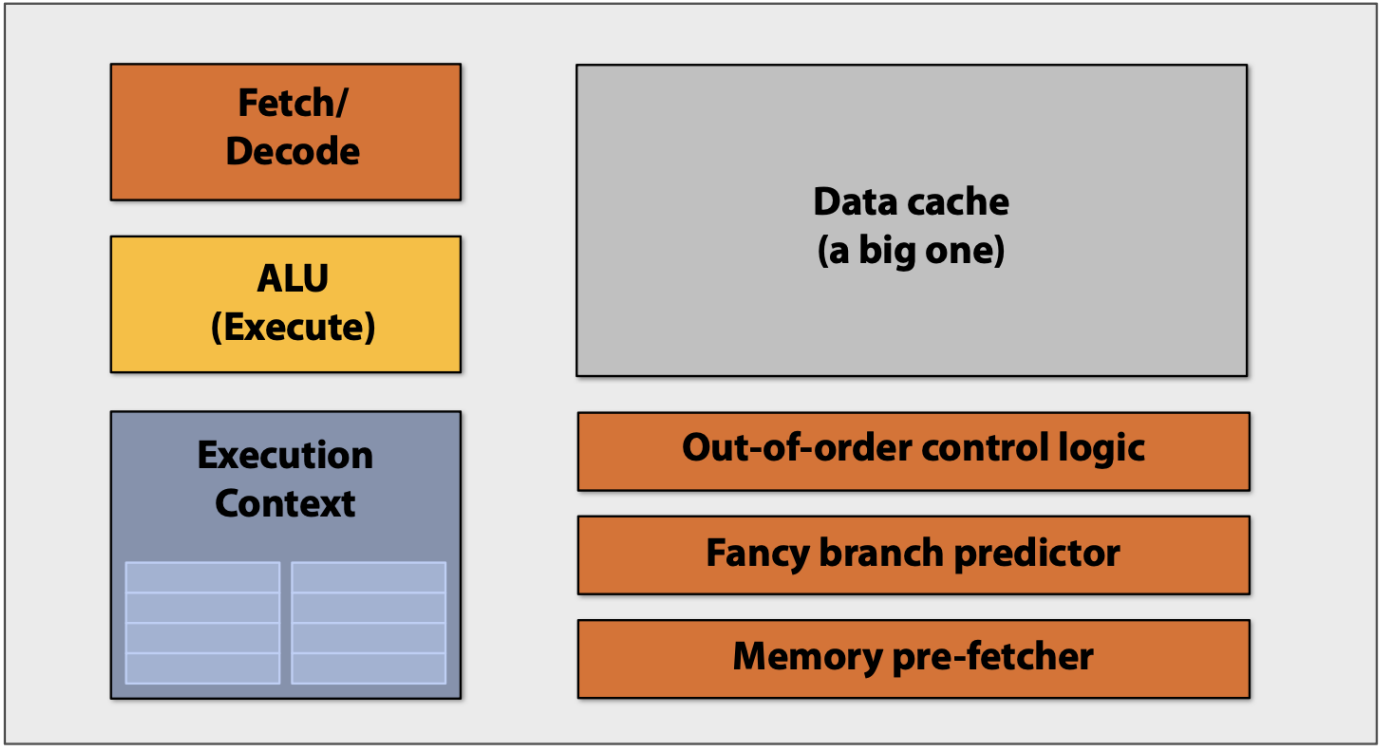
→ **Performance**

How to scale the computation capability?

Parallelism!

Case study: single chip device

Back to the old days: the single core system



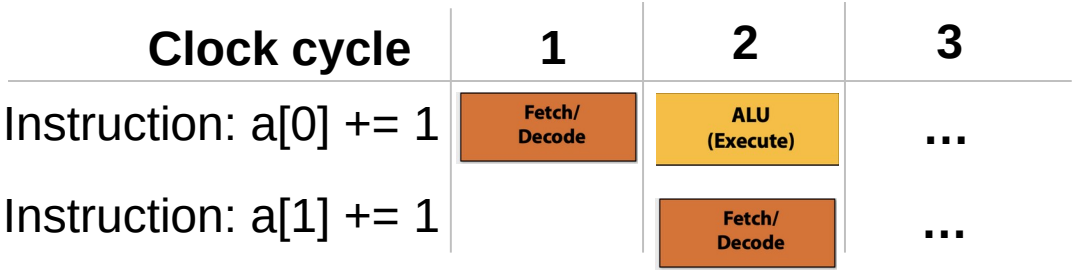
Review: the single core pipeline parallelism

Question

- Given a CPU core with known clock rate (e.g., 10MHz)
- How many FLOPS can it achieve?

Depends on the CPU architecture

- Suppose we have a perfect pipeline that has no bubble
- The FLOPS ~= the clock rate



The single-core machine also have parallelism

Parallelism methods

- Pipeline
- Instruction-level parallelism (a.k.a, super scalar)

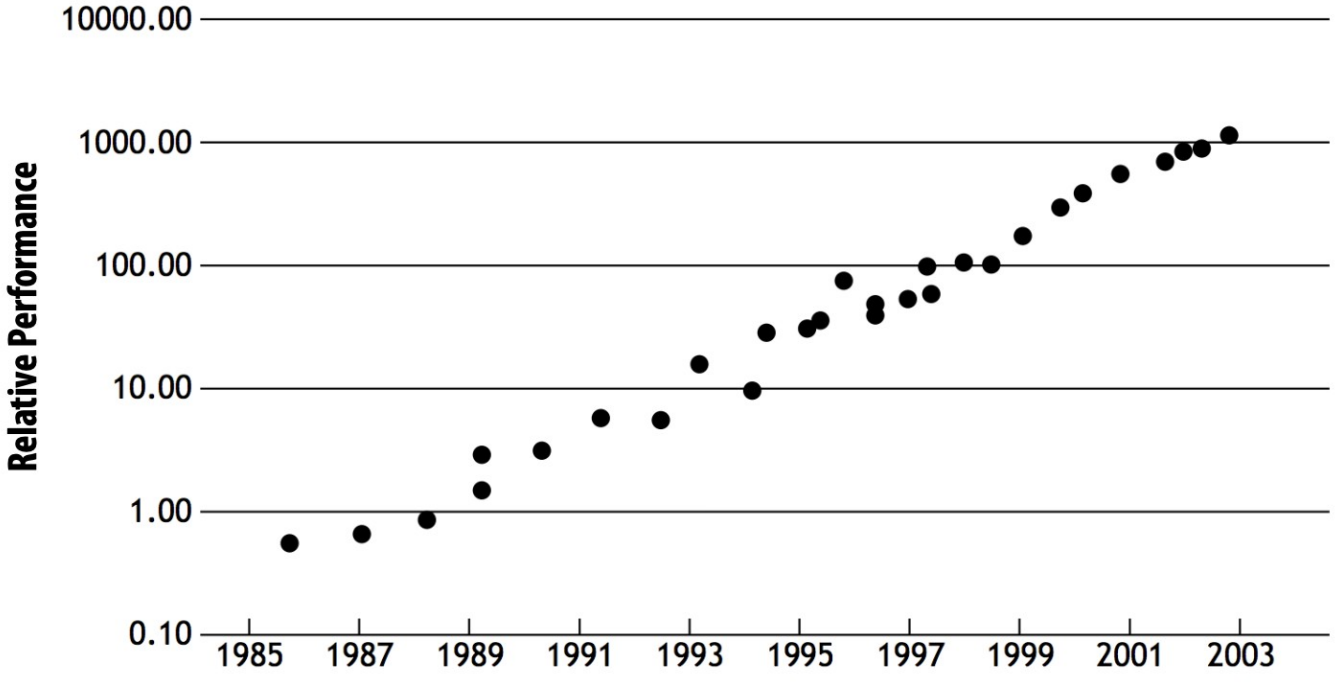
How does it scale?

- More efficient pipeline design & implementation, e.g., reduce bubble stall
- Fast clock rate, e.g., 10 MHz -> 3G Hz
- Exploiting wider instruction-level parallelism (ILP), e.g., issue 4 instructions / cycle

Optimal result: 12G Flops/sec

The pre-Moore law Era: single-core processor scales well

Single-core processor is growing exponentially faster

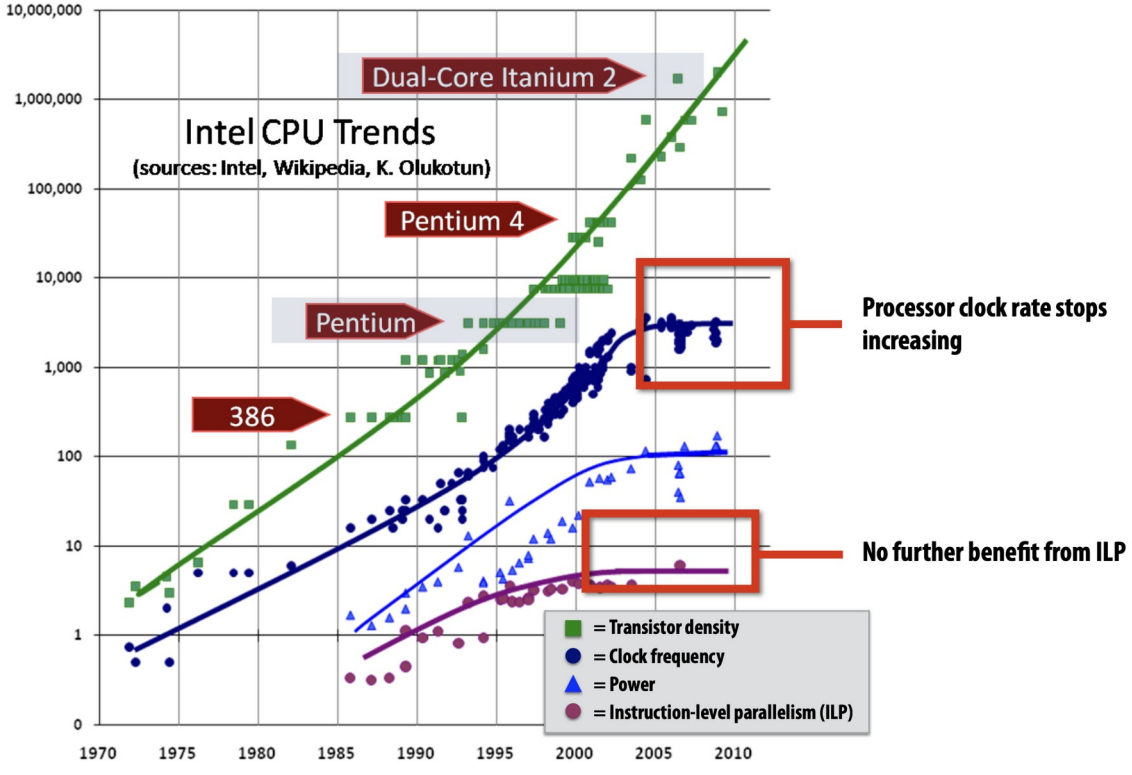


The single-core scale trend stalls quite a long time

TECHNOLOGY; Intel's Big Shift After Hitting Technical Wall

Then two weeks ago, Intel, the world's largest chip maker, publicly acknowledged that it had hit a "thermal wall" on its microprocessor line. As a result, the company is changing its product strategy and disbanding one of its most advanced design

ILP tapped out + end of frequency scaling



Source: The **Free Lunch** Is Over: A Fundamental Turn Toward Concurrency in Software

Speed up computation from a programmer

Question

- How to make your program run faster?

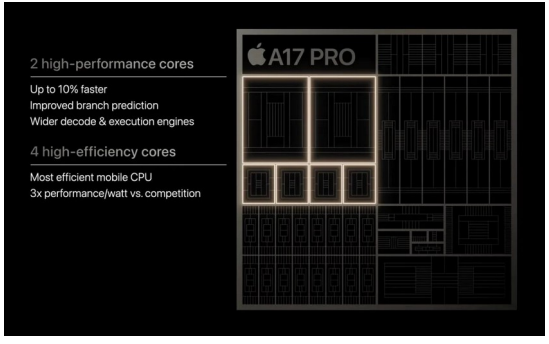
Before 2004:

- Just wait 6 months and buy a new computer!
- (Or, build a distributed system, but not really necessary, e.g., network is slow that time)

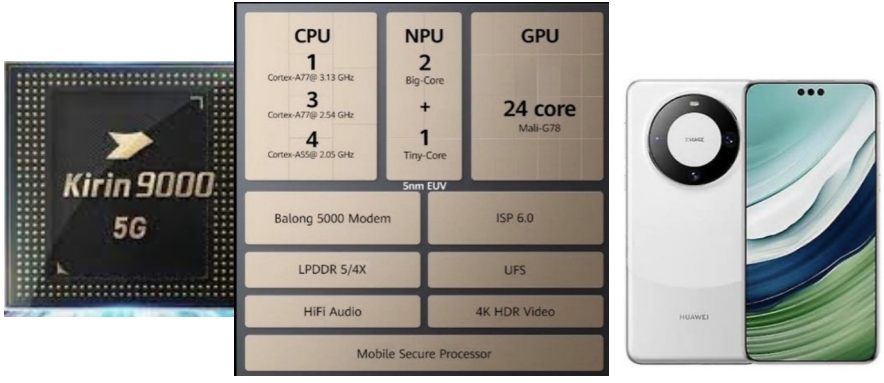
After 2004:

- You need to write a parallel program, either on a multi-core machine or on a cluster of machines

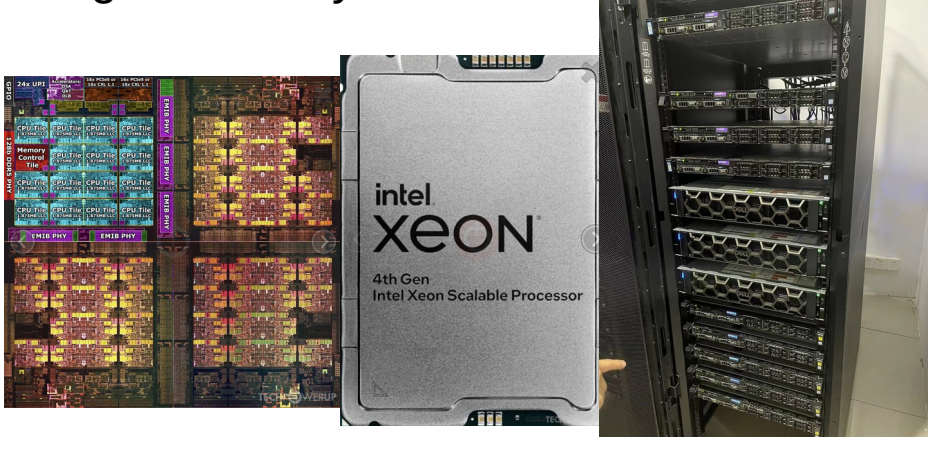
How to scale after 2004? “Glue” multi cores together



2 high performance cores +
2 high efficiency cores

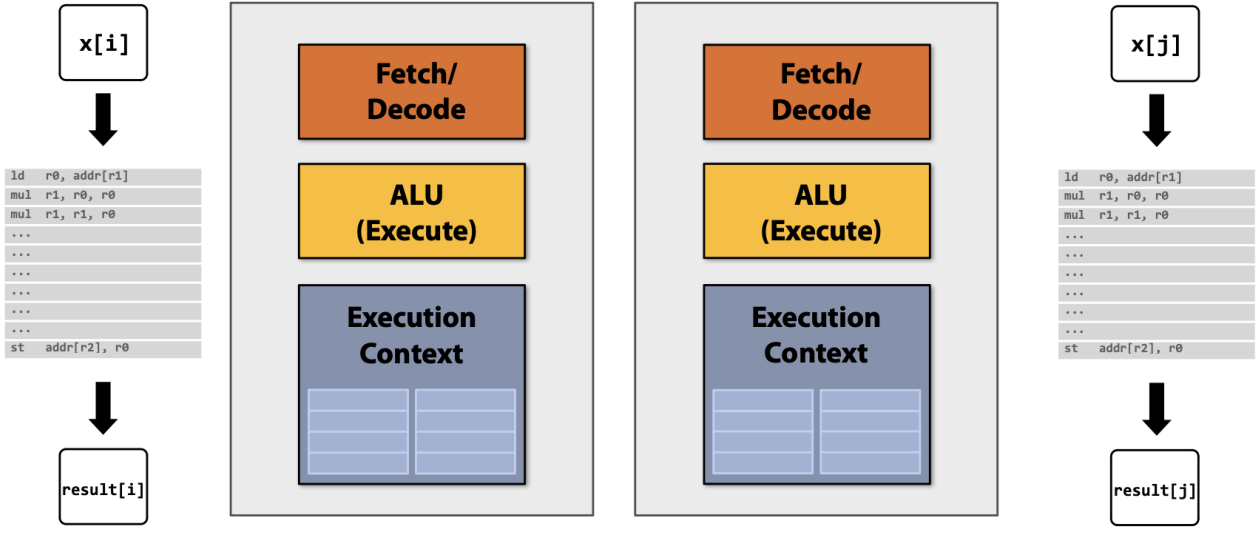


1 + 3 + 4 ARM cores



64 high-performance cores on a modern datacenter server at IPADS ⏪

Approach #1 of multi-core: add more physical cores



Add more cores to our system

- Instead of increasing the power of fetch/decode or the entire pipeline, add multiple cores together on a simple chip

Example program: a simple add program

Single-core version

```
int data[100000];

int sum = 0;

for i in 100000 :
    sum += data[i]

print(sum)
```

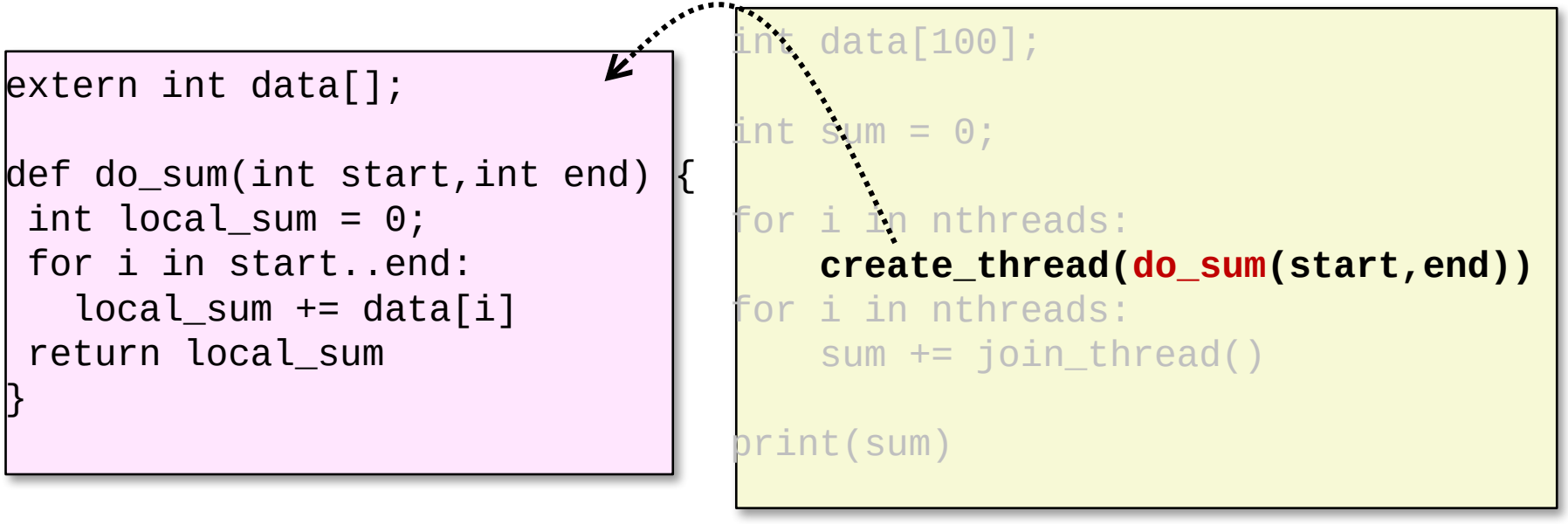
Multi-core version

```
int data[100000];

int sum = 0;

for i in nthreads:
    start = 100000 / nthreads * i;
    end = start + 100000 / nthreads;
    create_thread(do_sum(start, end))
for i in nthreads:
    sum += join_thread()

print(sum)
```

Exploiting physical cores with pthreads

How can a program to run on multiple cores ?

- The `pthread` library provides abstraction to create a thread on a multi-core CPU



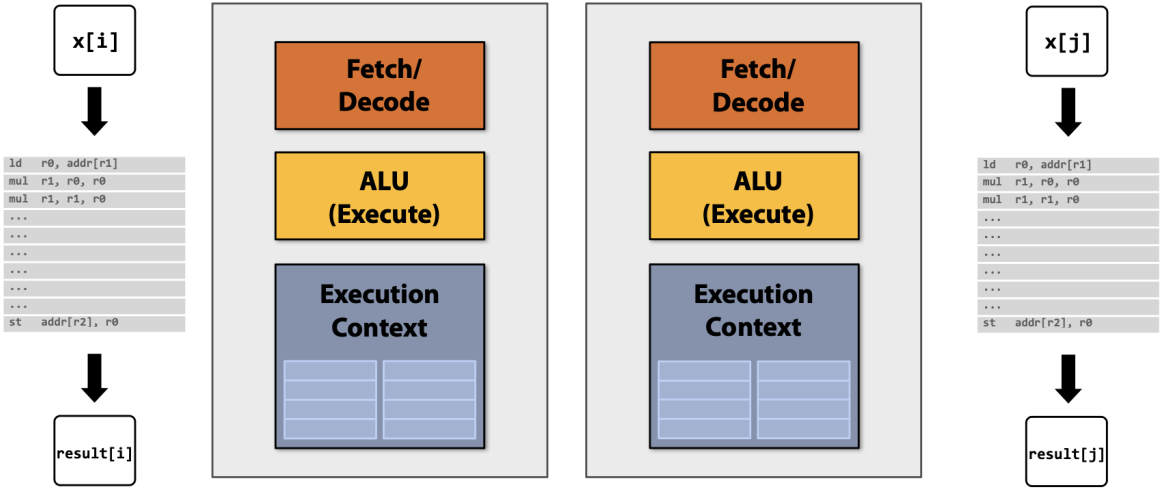
Problem: coherence under memory access hierarchy

Updates are first stored in the cache for better performance

- Without synchronization, different core's cache may have different values

The consistency problem described earlier happens in a multi-core setting

- E.g., if core 1 writes to addr0, and core 2 should see it later



A quick look at the cache coherence protocols

Cache coherence protocols

- Ensure different core have a consistent view of the cached value
- E.g., informally, if core 1 made an update to cache line L1, the L1's value will be visible to core 2 in a timely manner

Snoop-based cache (&directory-based) coherence protocol

- Each cache controller snoops all the memory reads/writes from the others
- Maybe implemented with a global directory (directory-based protocols)

Challenging to scale due to cache coherence protocols

Analysis of the previous protocols

- In snoop-based protocols, the #snoops increase linearly w/ the number of cores
- In directory-based protocols, the directory will become the bottleneck

Some multi-core CPUs do scale, but trade programmability

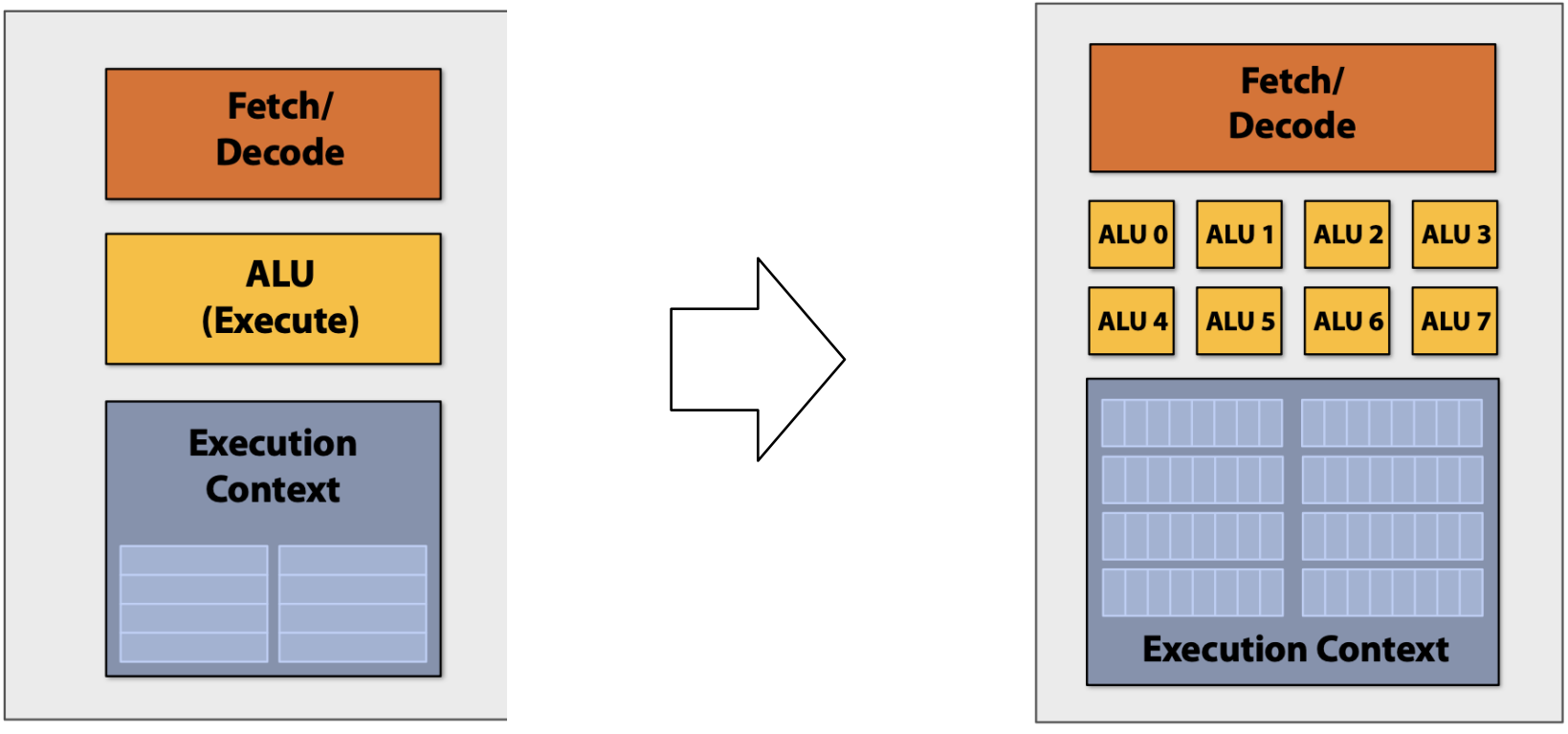
- E.g., ARM supports a relaxed consistency model, can support many cores; e.g., Huawei kunpeng 920 has 64 cores
- But, developers need to explicitly add barriers to ensure the last write has been seen by the others, e.g., fences

Modern CPU have a limited #cores on a single chip (20 – 100)

- Modern GPUs have many cores, but **has no cache coherence** at all

Approach #2 of multi-core: increase per-core density

More ALUs on a single core



Add ALUs to increase compute capability

Observation: data-parallel parallelism

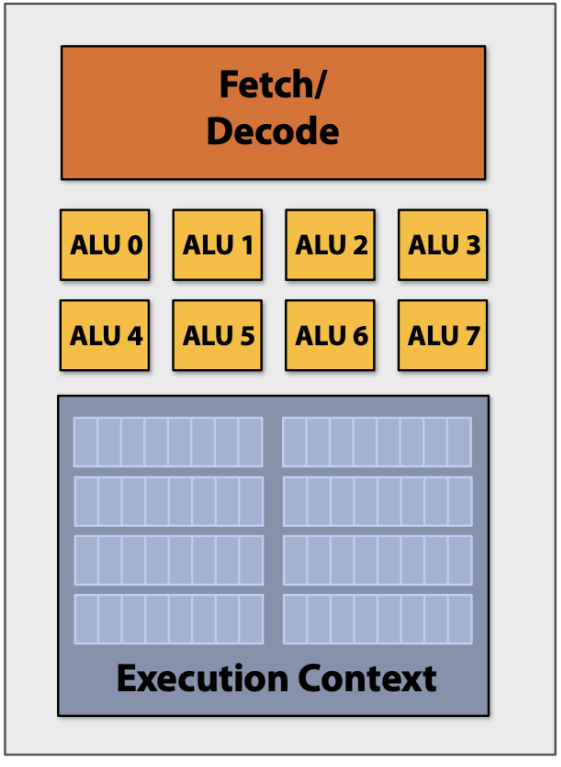
- Huge computation usually means operate on a huge amount of data
- Yet, the computation (instruction) on each of the data is similar

New ISA: SIMD processing

- Single instruction (e.g., read/add), multiple data
- Same instruction broadcast to multiple ALUs in parallel

Question: why not different instructions

- Cost of manage per-ALU instruction stream



SIMD processing with Intel’s AVX intrinsic

Example program: add two vectors

- Suppose the target machine has 256-bit SIMD width (32B = 8 x 4-bit float)
- The memory has too be aligned to enable SIMD calculation

```
int size = 1000000;
float* a = (float*)_mm_malloc(size * sizeof(float), 32);
float* b = (float*)_mm_malloc(size * sizeof(float), 32);
float* result = (float*)_mm_malloc(size * sizeof(float),
32);
```

SIMD processing with Intel’s AVX intrinsic

Non-SIMD version

```
for (int i = 0; i < size; ++i) {
   result[i] = a[i] + b[i];
}
```

SIMD processing with Intel’s AVX intrinsic

SIMD version

```
int numLanes = size / 8;

for (int i = 0; i < numLanes; ++i) {
   __m256 avx_a = _mm256_load_ps(&a[i * 8]);
   __m256 avx_b = _mm256_load_ps(&b[i * 8]);
   __m256 avx_result = _mm256_add_ps(avx_a, avx_b);
   _mm256_store_ps(&result[i * 8], avx_result);
}

int remainder = size % 8;
for (int i = size - remainder; i < size; ++i) {
   result[i] = a[i] + b[i];
}
```

SIMD vs. non-SIMD version

Machine setup: Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz

```
auto start_time_avx = std::chrono::high_resolution_clock::now();
addVectorsAVX(a, b, result, size);
auto end_time_avx = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_time_avx = end_time_avx -
start_time_avx;
std::cout << "AVX execution time: " << elapsed_time_avx.count() * 1000 <<
" ms" << std::endl;

auto start_time_non_avx = std::chrono::high_resolution_clock::now();
addVectors(a, b, result_non_avx, size);
auto end_time_non_avx = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed_time_non_avx = end_time_non_avx -
start_time_non_avx;
std::cout << "Non-AVX execution time: " << elapsed_time_non_avx.count() *
1000 << " ms" << std::endl;
```

AVX execution time: 2.54939 ms **// why not 8X faster?**
Non-AVX execution time: 5.67899 ms

Accessing memory (or other storage) & Roofline model

Memory stalls (Communication stalls)

A processor "stalls" when it cannot run the next instruction in an instruction stream because of a dependency on a previous instruction.

Accessing memory is a major source of stalls

Memory access times ~ 100's of cycles

- Memory "access time" is a measure of latency
- Possible optimization: prefetching

Prefetching isn't always possible if the memory read/write is the bottleneck

Terminology

Memory (storage) latency

- The amount of time for a memory request (e.g., load, store) from a processor to be serviced by the memory system
- E.g., 100 cycles (DRAM), 1us (Network in local datacenter)

Memory (storage) bandwidth

- The rate at which the memory system can provide data to a processor
- E.g., 20Gbps (Network)

Memory load speed

Time to load the data = Latency + Payload / bandwidth

- Note that it is the optimal case, e.g., latency and bandwidth can be affected by the others

Both factors matters

- For CPU load/store, the latency may be dominate
- For transfer large data, bandwidth will be the dominate factor

Real world problem

- What if I want to transfer 100PB data from shanghai to bejing?

Case study: AWS Snowmobile

工作原理

AWS Snowmobile 可将极大量数据迁移至 AWS。Snowmobile 是一个 45 英尺长的坚固的集装箱，由一台半挂卡车牵引，一次可以传输高达 100PB 的数据。

Thought experiment on load store

Example program: add two vectors

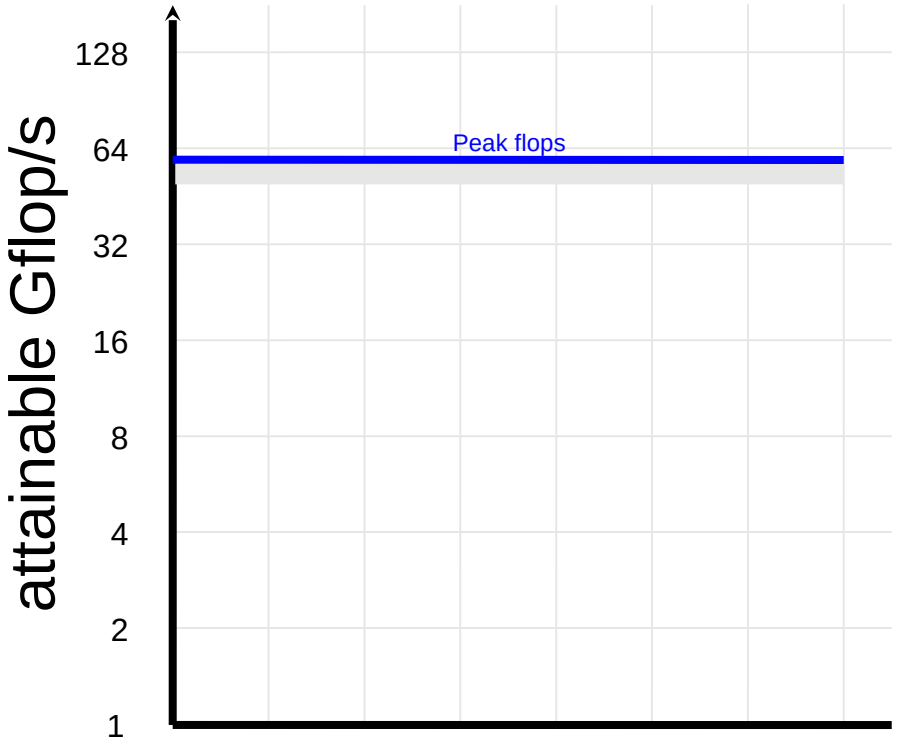
Assume vectors contain millions of elements

- Load input A[i + 8, 8]
- Load input B[i + 8, 8]
- Compute A[i, i + 8] + B[i, i + 8]
- Store output C[i, i + 8]

CPU specification (Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz)

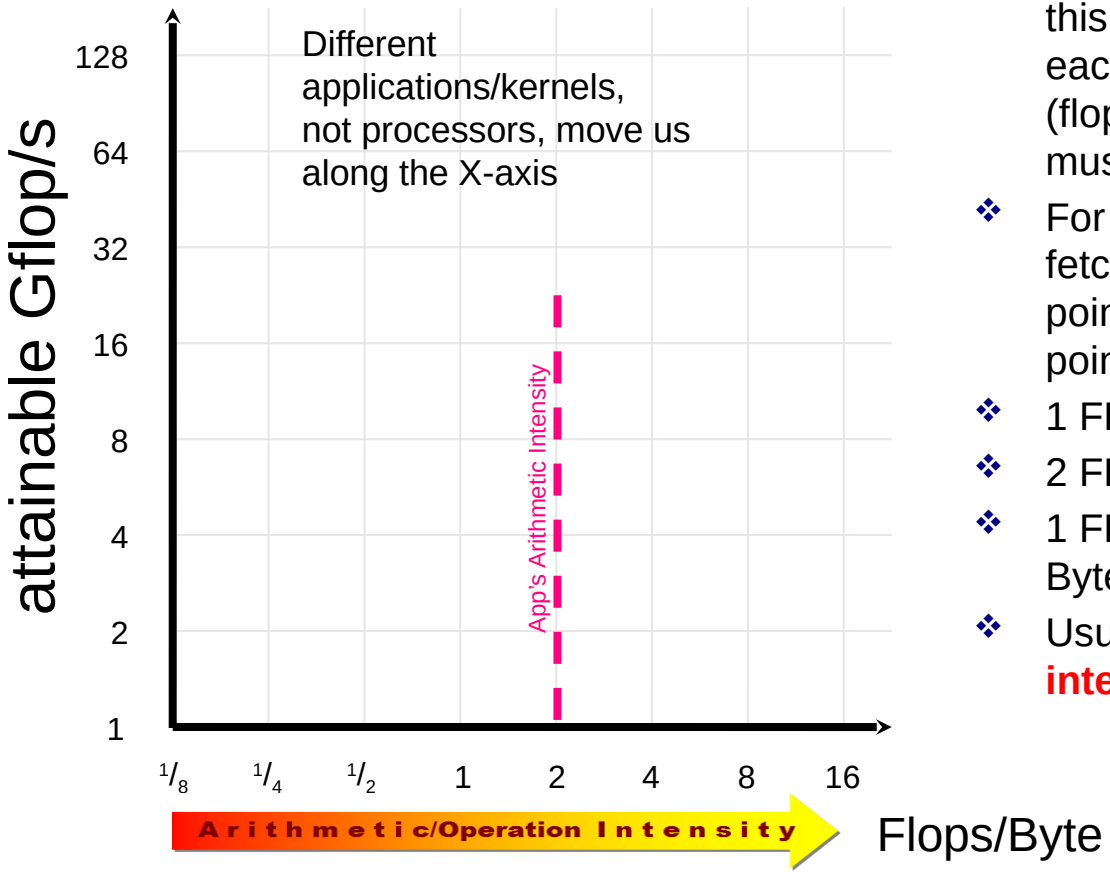
- 2.2GHz, assuming 8 addition per-cycle: ~= 211 GB/s (3 * 8 4B loads per cycle)
- Memory speed: 76.8 GB/s (measured), becomes the bottleneck!

Roofline Model: y-axis



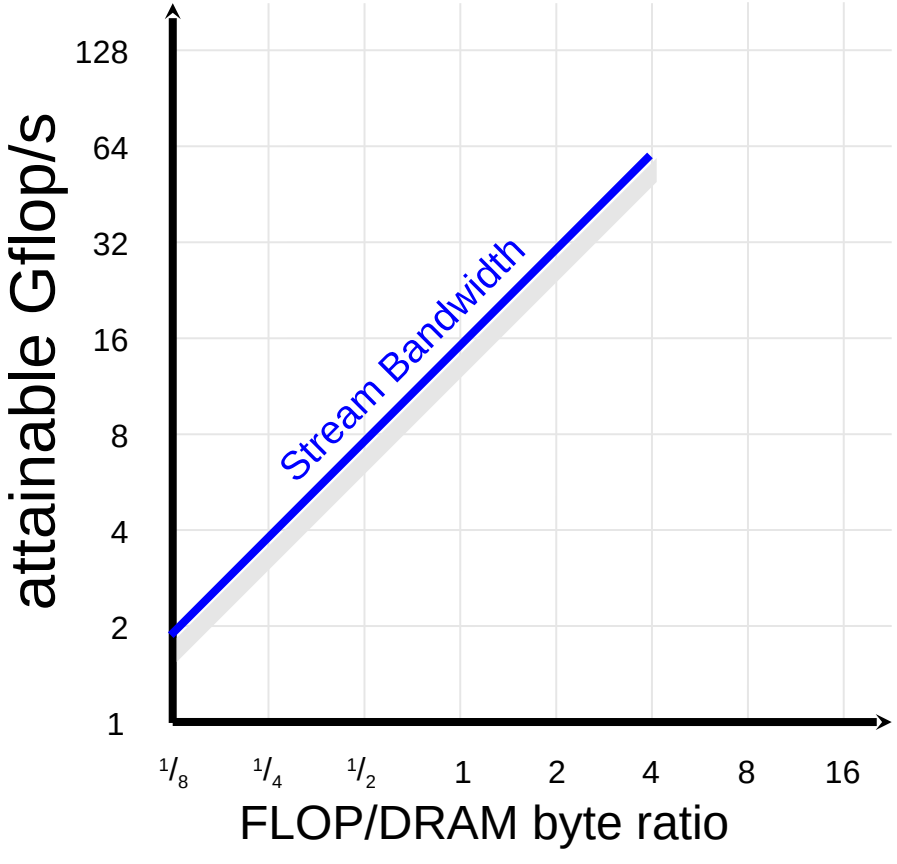
- The y-axis describes the attained performance (of the hardware)
- It's easy to add the "peak performance" as an upper bound

Roofline Model: x-axis



- The x-axis tells indicates for this particular application, for each floating-point operation (flop), how many bytes (B) must be fetched
- For example, if we have to fetch half precision floating-point numbers for each floating point operation, then:
- 1 FP16: 2 Bytes (16 bits)
- 2 FP16: 4 Bytes (32 bits)
- 1 FLOP requires 4 DRAM Bytes
- Usually called **operational intensity (OI)**

Bandwidth as Slope



Bandwidth is represented as a slope of

Peak Flops/ OI

It is a given by the system configuration/architecture

- Remember *m* =

Example

- m = y/x = 16
- y
- x =

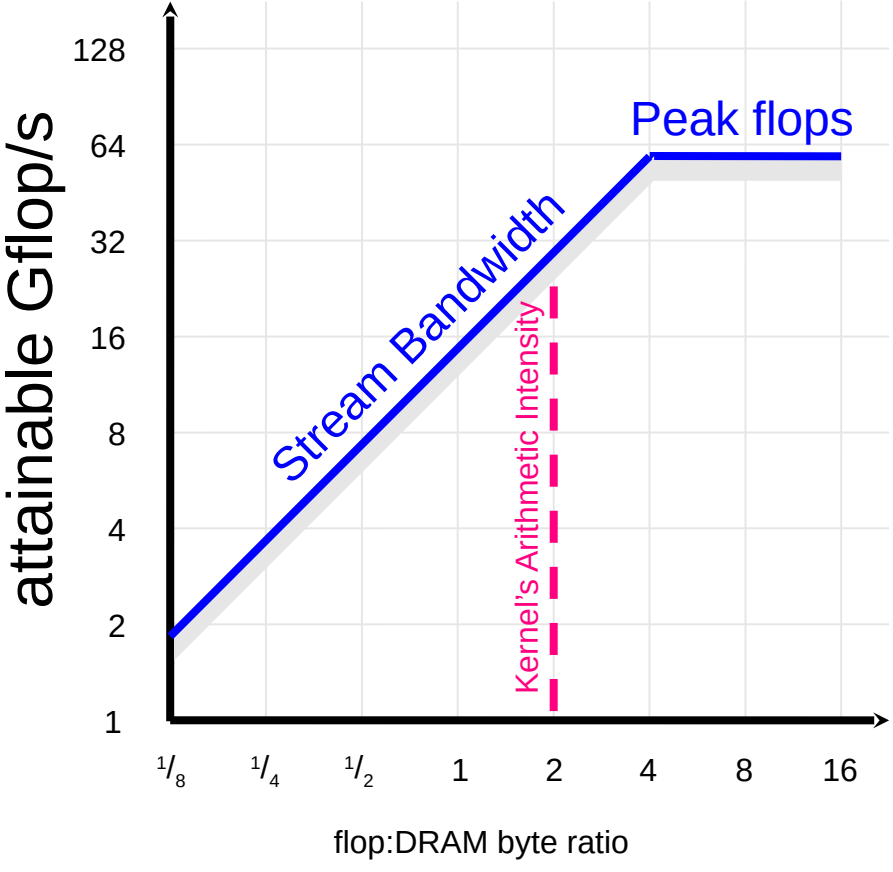
Put it all together: the roofline model

Given an application

- If we know how many FLOPs performed per-memory read, we can see whether it is computation bound or memory bound

Benefit

- Give hints on the optimization directions



Back to SIMD: Modern GPU also builds around SIMD

e.g., NVIDIA GPUs

- CPU: 8 SIMD ALUs per core
- GPU: 32 (and even more) SIMD ALUs per core

Question: why GPU can have such a large amount of ALUs?

- Trade other functionalities for general-purpose computing
- E.g., no branch predictor, no cache coherence

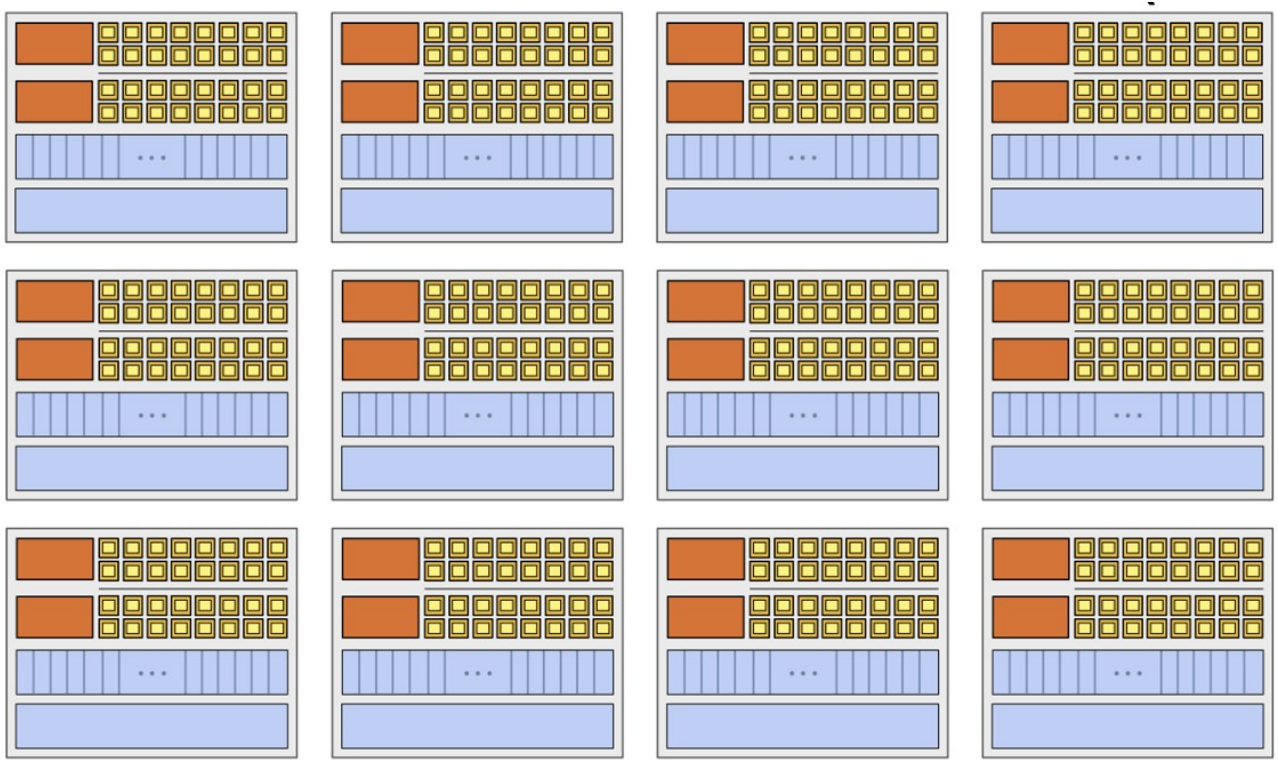
Put it all together:
Approach #1 + Approach #2 are typically used together

Example: Intel Core i9 (Coffee Lake)

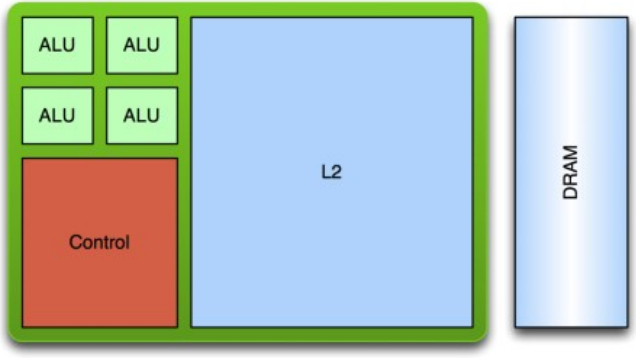
8 cores, 8 SIMD ALUs per core

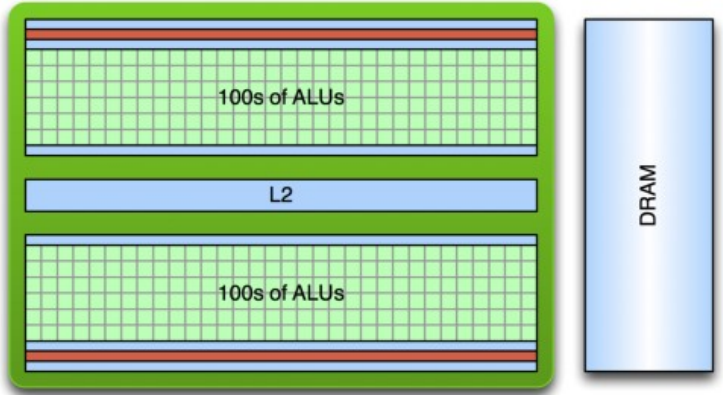
Example: NVIDIA GTX 480

15 cores, 32 SIMD ALUs per core



A quick comparison of CPU & GPU





CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution

GPU

- Optimized for data-parallel, throughput computation
- More transistors dedicated to computation
- Architecture tolerant of memory latency

SIMD alone is insufficient

SIMD instruction is low-level

- Only simple vector add, multiple, read/write, etc.

Given the same hardware area, deploy more for computation, less for control flow

- E.g., all ALUs doing a SIMD share **the same program counter**

However, programs are more high-level

- E.g., how to express conditional branch using SIMD instruction?
- Question: how can we execute conditional vector add on SIMDs?

Conditional execution under SIMD

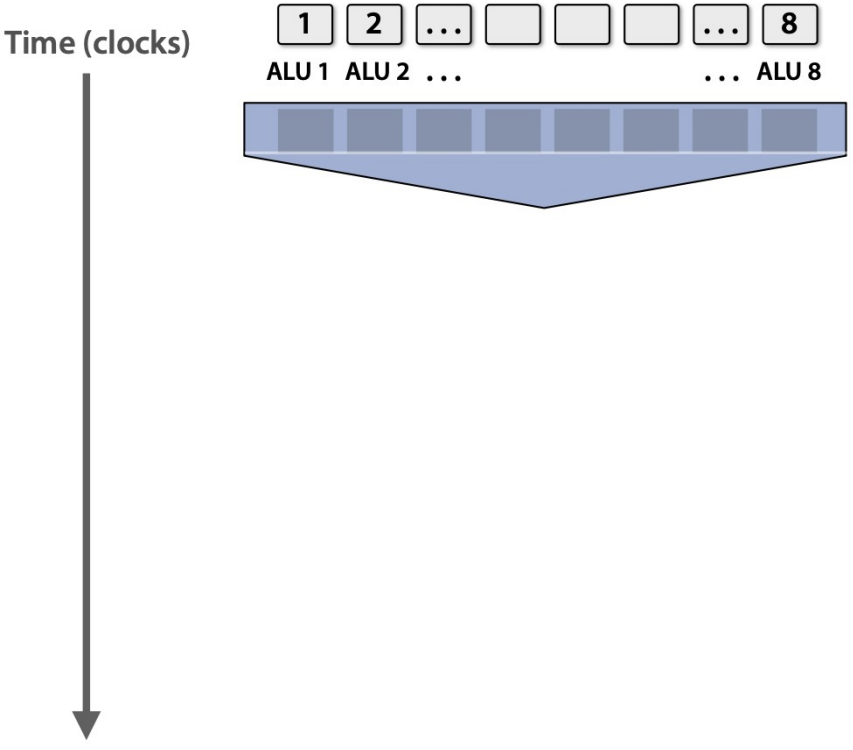
Constrain of SIMD: every SIMD ALUs share the same program counter (PC)

- But conditional execution have 2 (or more) branches

Solution: masked instruction

- Basic idea: execute all branches sequentially
- Each ALU has a mask to indicate whether the instruction will be executed

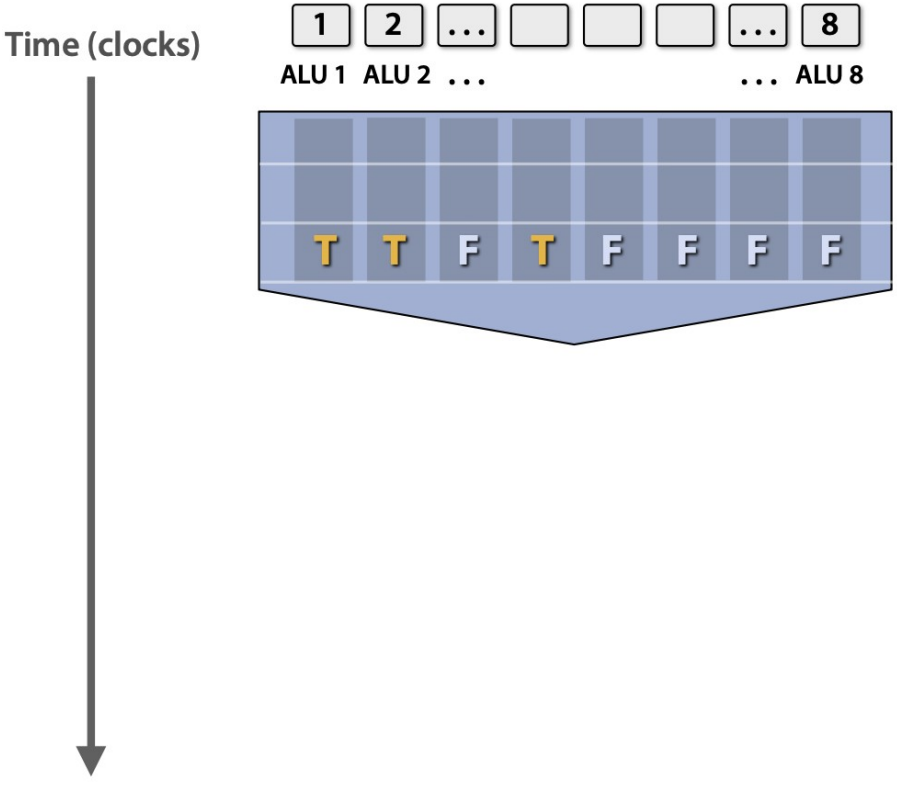
Conditional execution under SIMD



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);

   tmp *= kMyConst1;

   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;

   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

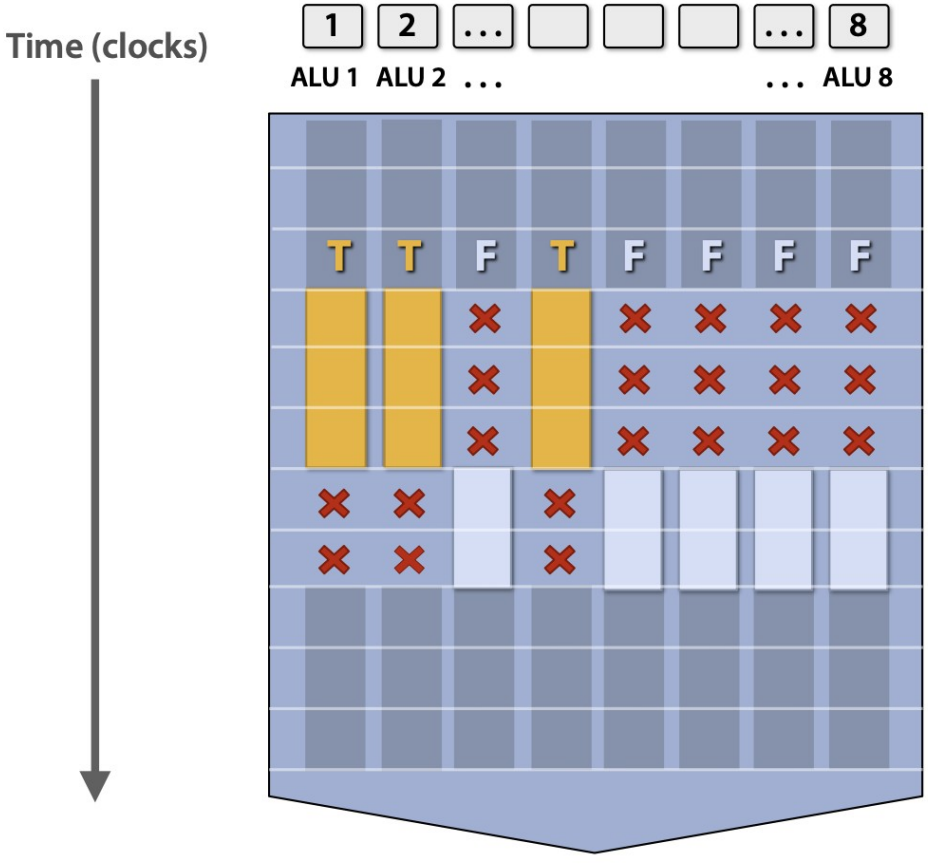
Conditional execution under SIMD



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);

   tmp *= kMyConst1;

   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;

   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

Conditional execution under SIMD



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

Conditional execution under SIMD

Challenging due to the lack of individual PC for each ALU

- Masks are needed to selectively enable ALU execution

Cons

- Waste performance: at worst case, no ALUs are enabled during execution (when all the elements take one branch)

Drawback of masked SIMD & the introduction of SIMT

Drawback: hard to express the conditional execution

- Programmers cannot directly call the simple vector add ops ⏪

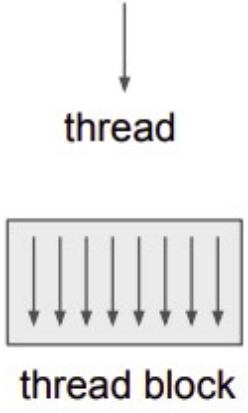
SIMT: single instruction multiple threads

- Hardware vendors provide development kit to simplify developing SIMD programs

SIMT is not a hardware implementation, but a parallel abstraction

- The programs write code assuming an abstracted SIMT model
- The compiler do necessary transformation (out of the scope of this course) to the underlying (conditional) SIMD operations
- But developers still need to consider hardware details in order to best utilize it

Example: SIMT and CUDA



Hardware model: SIMT

- Single instruction, multiple threads

Abstraction: C code

- Programmer writes code for a single thread using C
- All threads executes the same code
    - But can take different path (control flow supported)

Threads are grouped in a block

- Threads within the same block can synchronize execution
- Each block is mapped to a GPU core for execution (we will omit details right now)

Example: vector add

Compute vector sum

- C = A + B

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

How to parallel it with SIMT?

- Each thread only compute the sum of one entry. There is no need to manually add the masks

Example: vector add

Compute vector sum

– C = A + B

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

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

Performance of the vector add on GPU

Execution time: 28.32us (NVIDIA V100 GPU)

- In comparison: CPU time 2.54 ms w/ AVX

Note that SIMT handles many things for us

- Work distribution to multiple cores (CPU needs to use pthread)
- Conditional branches
- No need to call low-level AVX instructions
- Etc.

The comparison is not so fair, but shows the overall trend

- GPU has high bandwidth memory
- GPU has no cache coherence
- Etc.

Summary: parallelism on a single device

3 parallelism strategies on a single device

- Single core+: pipeline + super scalar with instruction level parallelism (ILP)
- Single core++: added SIMD support
- Multiple core: a single core (single core, single core+, single core++) can be glued together !

Question: what's next? Single core+++++?

- Solution (Out of the scope of this lecture): domain-specific accelerators !

The era of domain-specific accelerators

Accelerators (even on general-purpose computing devices)

- Hardware designed to fulfill a single task
- Typically are not general-purpose, e.g., not programmable

CPU:

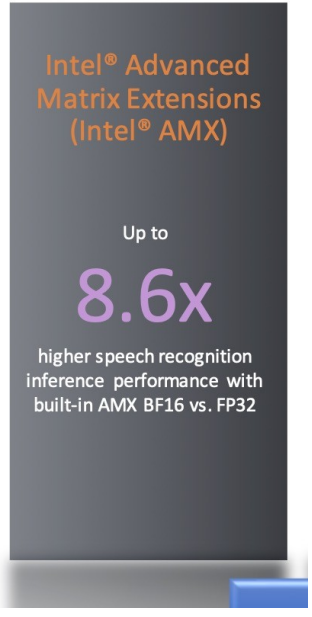
- SIMD + Matrix accelerators (e.g., Intel's AMX accelerators)

GPU:

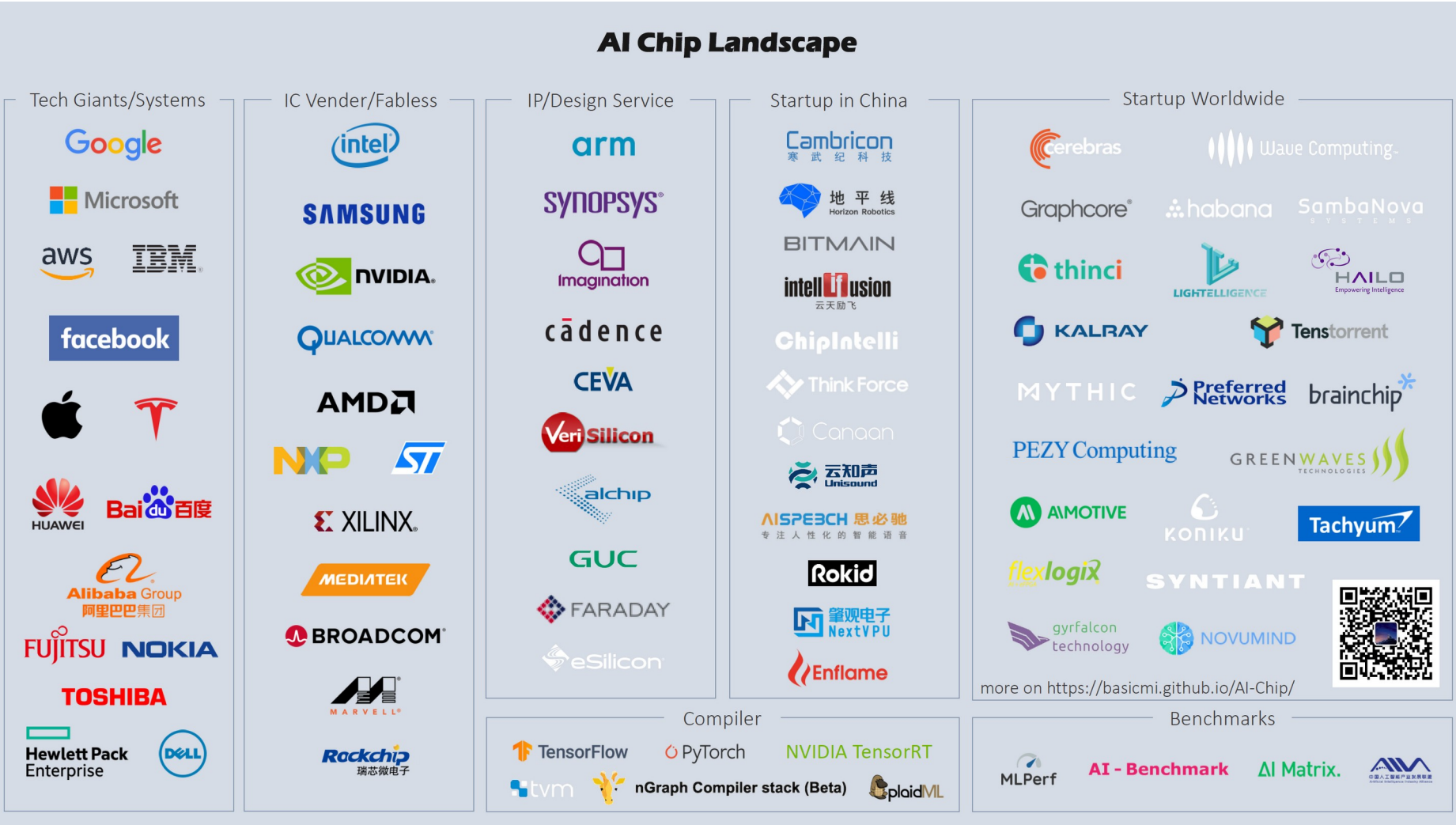
- Tensorcore: accelerators for matrix operations

TPU (Tensor processing unit):

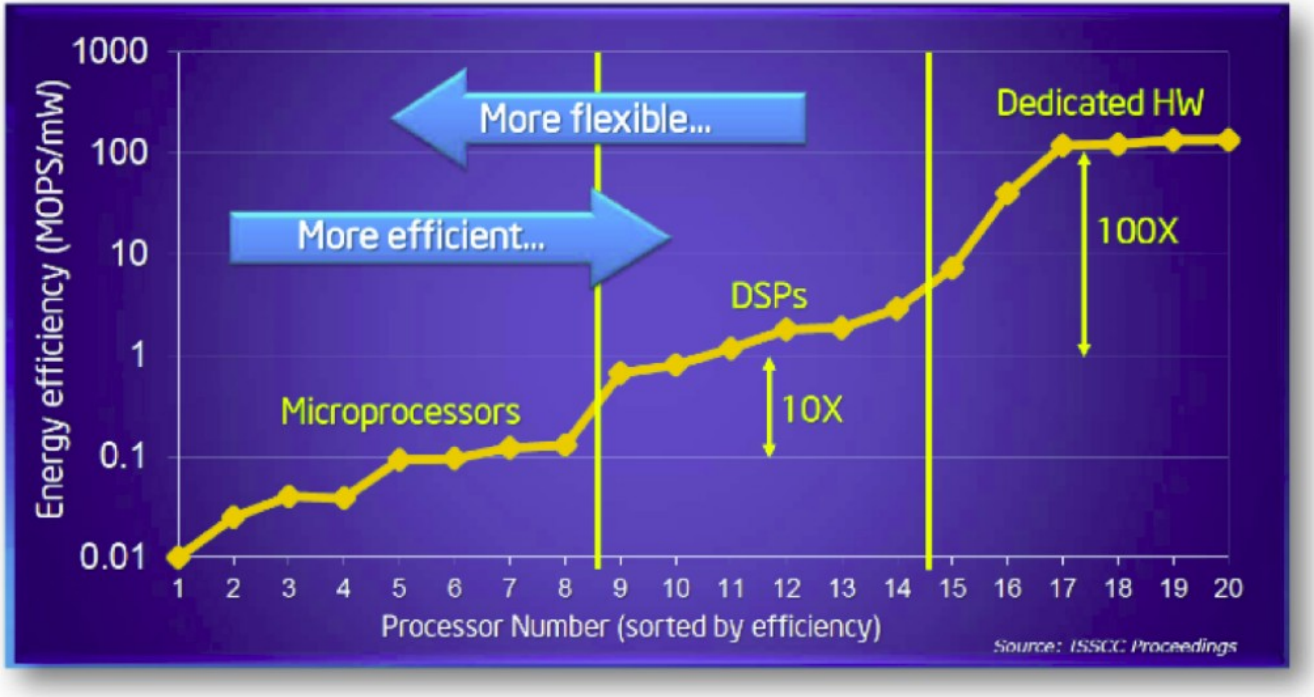
- tensor process (optimized for large matrix operations)



AI Chip Landscape



Accelerators: Flexibility vs. Efficiency Tradeoffs

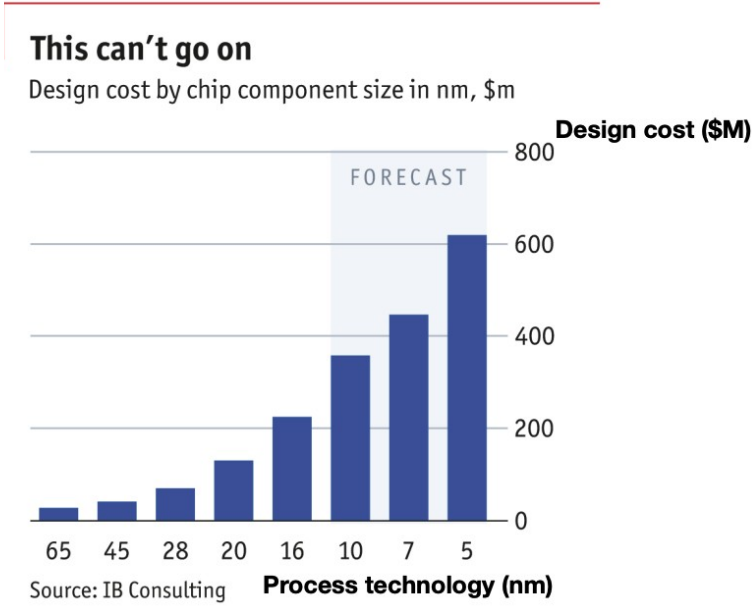


Source: Bob Broderson, Berkeley Wireless group

Specialization Challenge

Tape-out costs for accelerators ASICs is exorbitant 10x cost gap between 16nm and 65nm

- Risky bet to design hardware accelerators for ever-changing applications



Remaining question: is a single device sufficient?

A device has limited physical capacity to store “cores” (chip size)

- Our cores are generalized, e.g., can either be CPU cores, GPU cores (cores w/o cache coherence + many SIMD ALUs, etc.), domain-specific cores

How to make a single device faster?

- Increasing clock rate (has a limit)
- Put more cores on a single chip, but has physical limits, e.g., chip size

Question: is a single device sufficient?

A device has limited physical capacity to store “cores” (chip size)

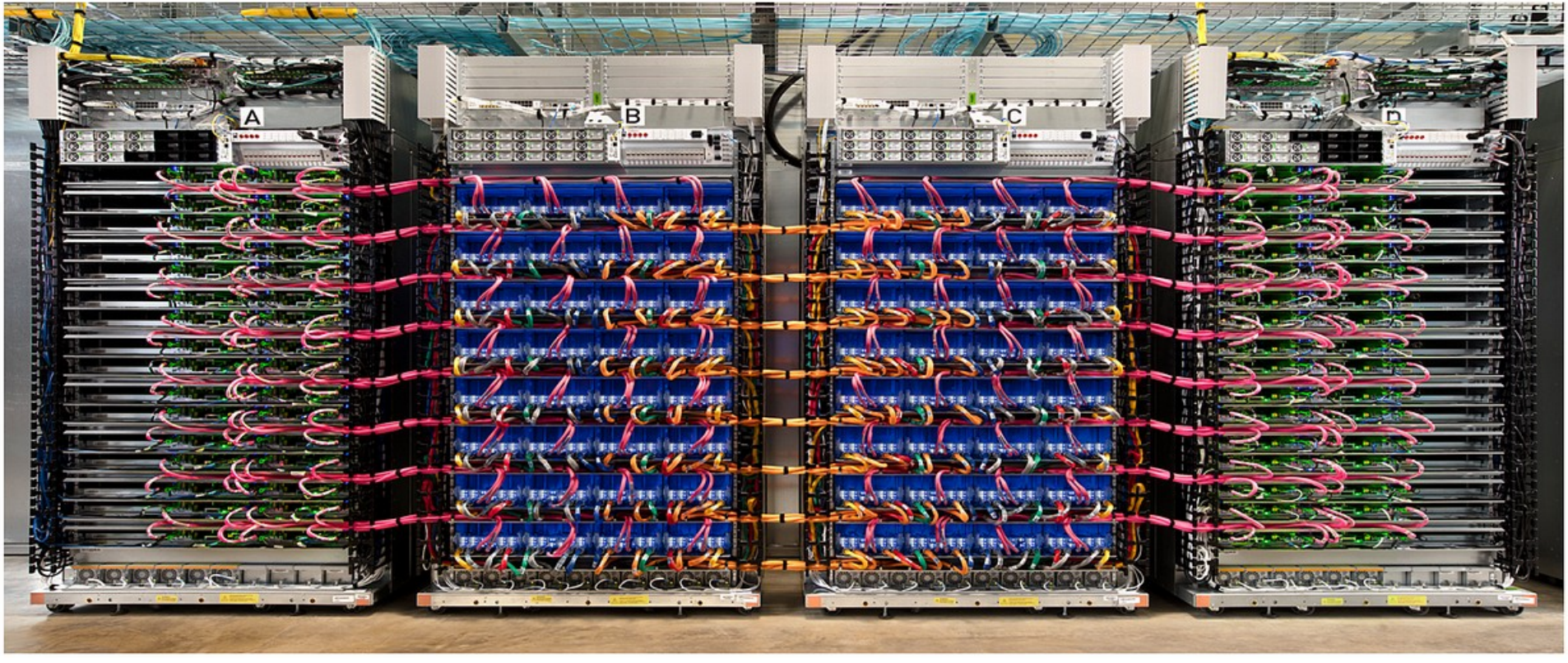
- Our cores are generalized, e.g., can either be CPU cores, GPU cores (cores w/o cache coherence + many SIMD ALUs, etc.), domain-specific cores

Why? Recall our previous calculation

- Basically, a A100 needs 30 seconds for an A100 GPU to finish an iteration on a single input (a.k.a, token) in the optimal case
- How many tokens are trained? 13T tokens! [1]
- To use one A100 to train GPT-4, we need about 412 years to finish the training

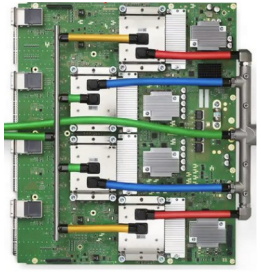
[1] https://the-decoder.com/gpt-4-has-a-trillion-parameters/

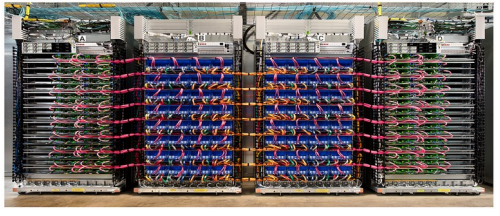
The case for distributed computing



Example: Google’s TPU v4 cluster

Spectrum of computation device available





GPU
OPs

Apple M2
GPU 3.6 TFLOPs

NVIDIA A100
GPU 19.5 TFLOPs

Google TPUv4
275 TFLOPS

A cluster
? TFLOPS

Performance

Not so easy: many tedious things to cope with

e.g., programmers need to manually do the following:

- Partition the data
- Deploy programs on multiple machines
- Schedule the works, e.g., what if the jobs have dependencies?
- Fault tolerance
- Network communication is much slower than local memory accesses