CME 213

SPRING 2019

Eric Darve

Before we start

Download from class web page
https://stanford-cme213.github.io/

the file in `Code/`
`create_vm_hw3.sh`

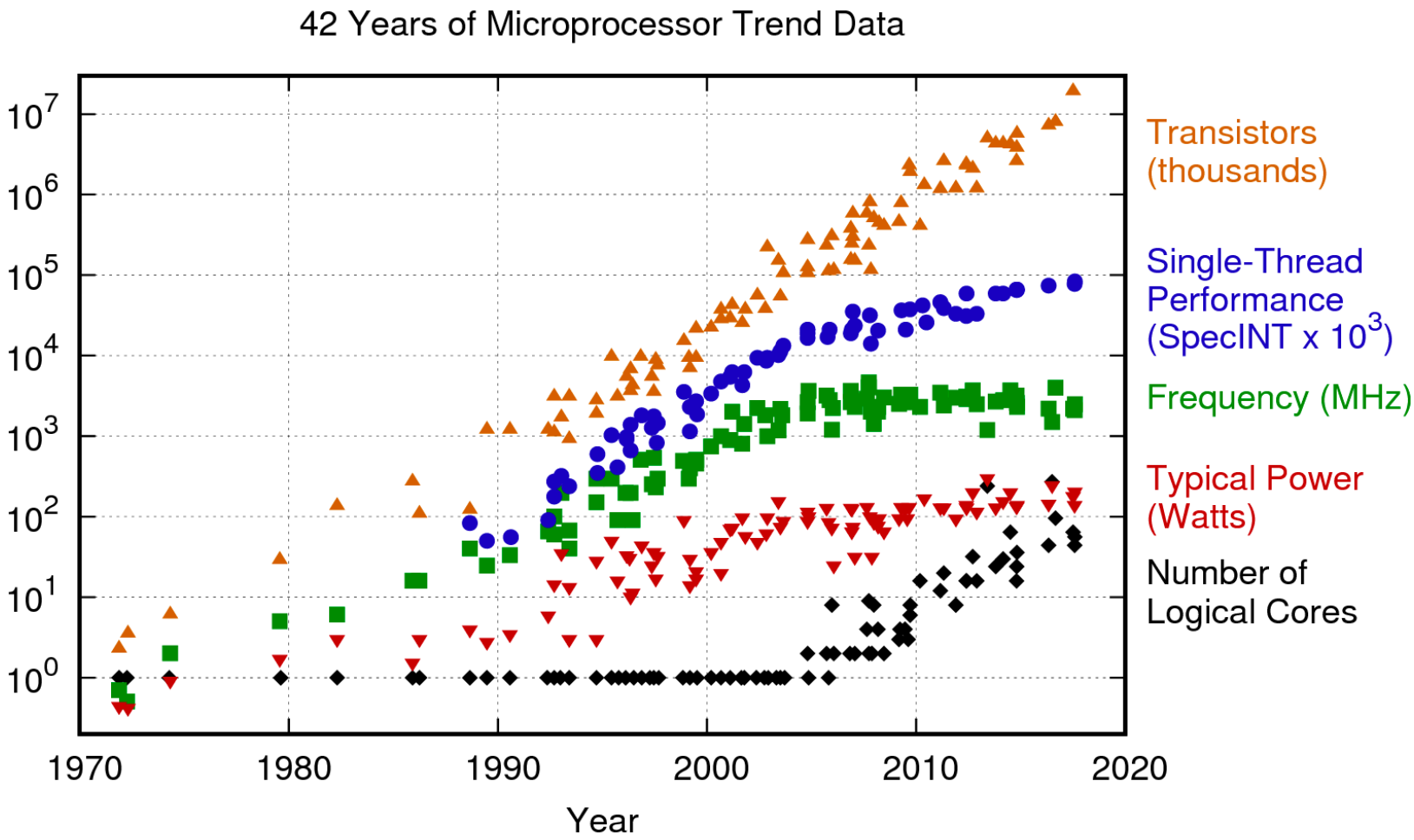
And run the script:
```
$ ./create_vm_hw3.sh
```

Summary of previous lectures

- C++ threads, C Pthreads: low-level multi-threaded programming
- OpenMP: simplified interface based on #pragma, adapted to scientific computing
- OpenMP `for` and scheduling (static, dynamic)
- OpenMP `tasks`
- `Reduction, atomic, critical`

GPU computing!

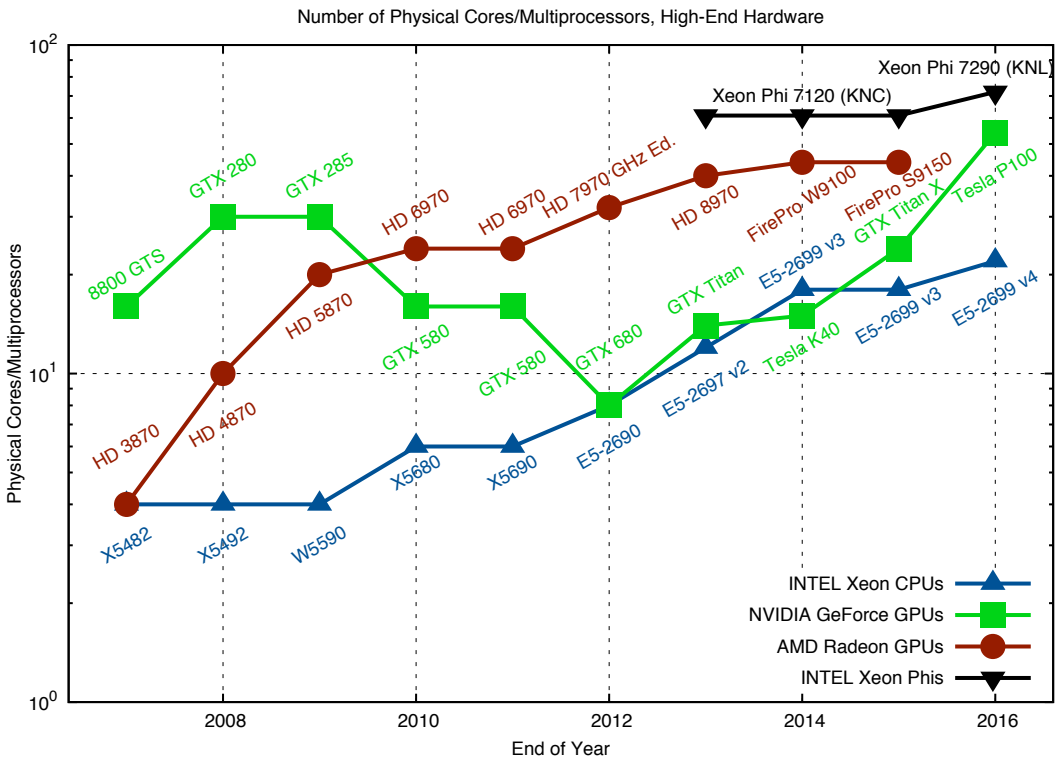
Performance numbers (Karl Rupp)



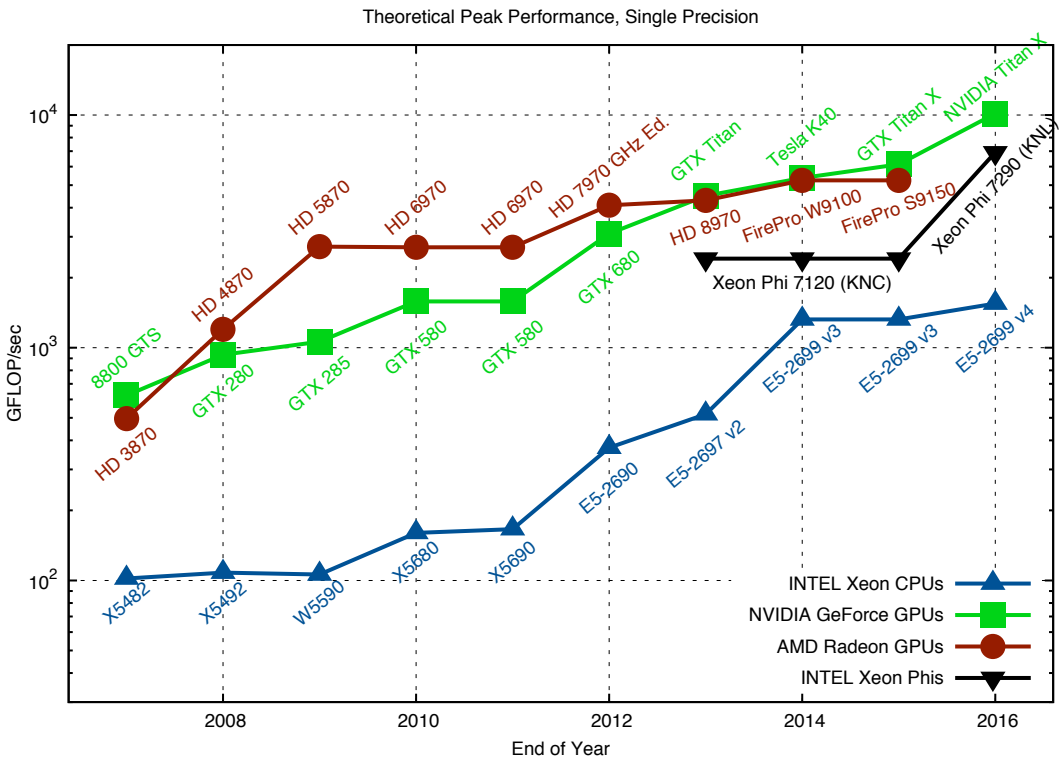
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/
https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

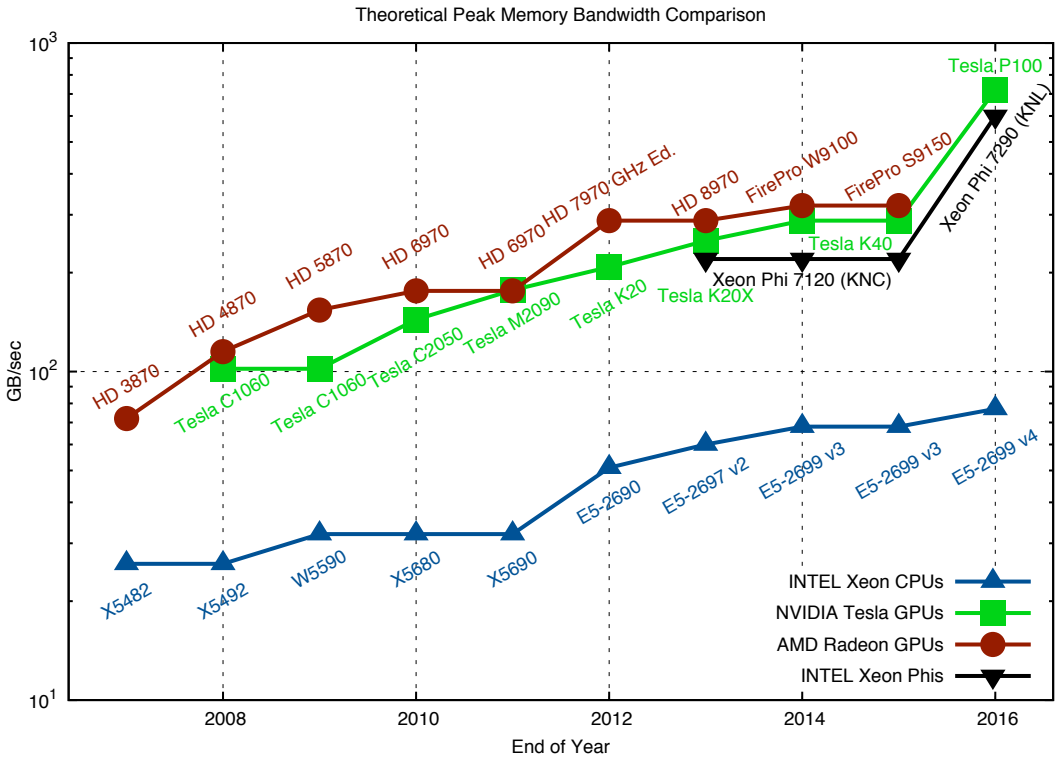
Key to performance: lots of cores!



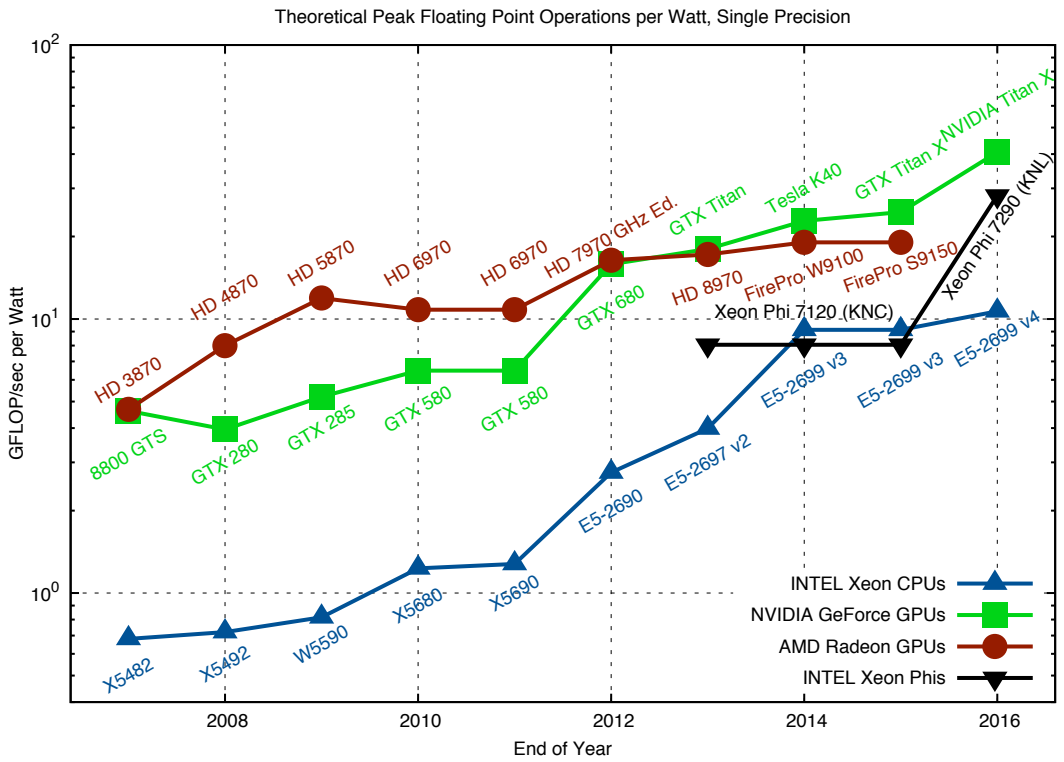
Peak performance: single precision



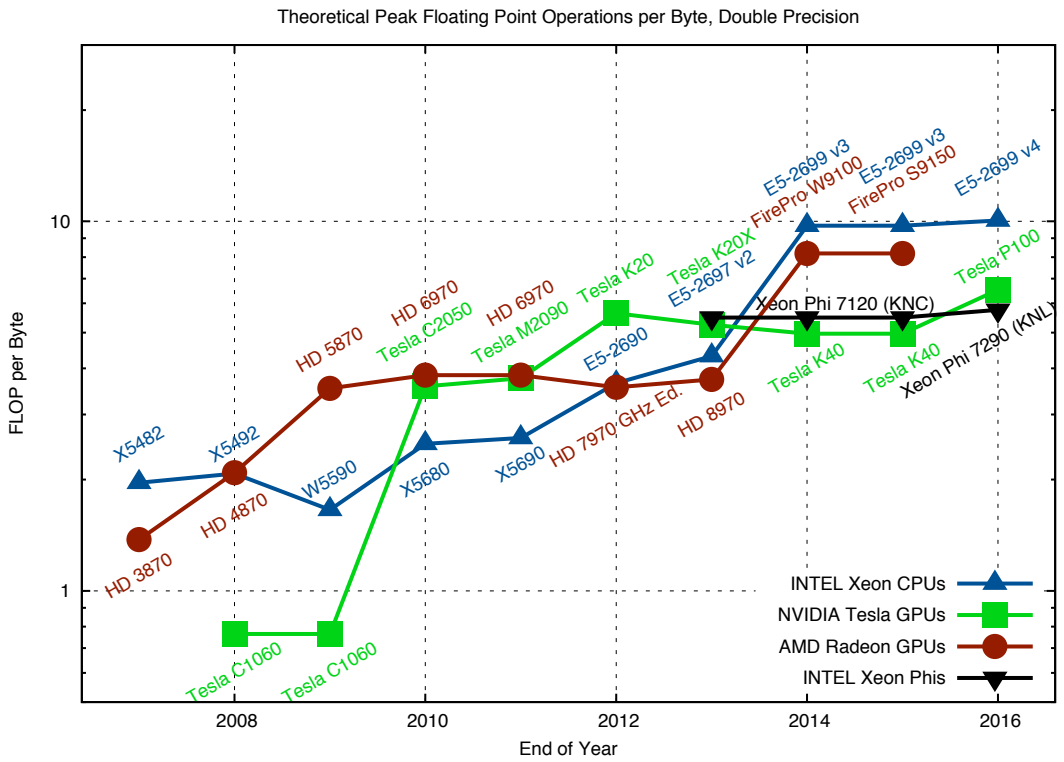
Memory bandwidth



Perf/Watt: key for server farms



Arithmetic intensity: memory/compute bound?

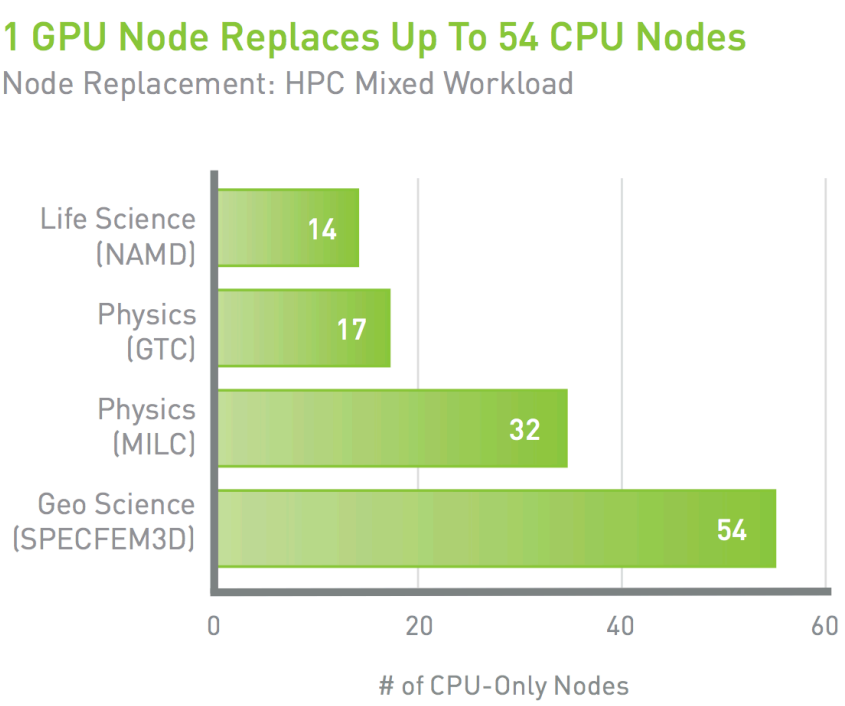


Performance for scientific applications

Example: Volta V100

- 7.45 teraflops double-precision performance
- 14.9 teraflops single-precision performance
- 134 teraflops for tensor (deep learning)
- 870 GB/sec memory bandwidth
- 250 W power

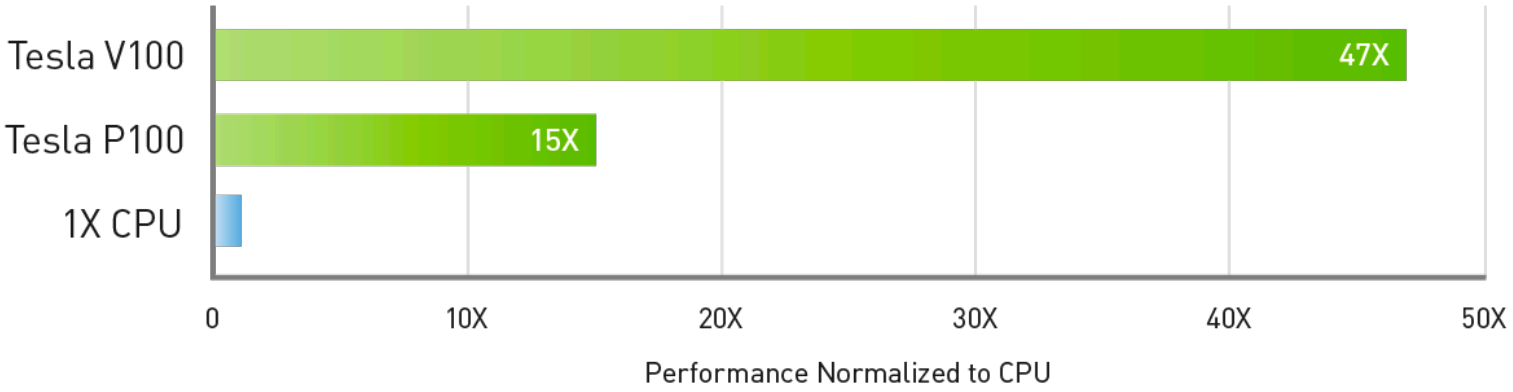
Performance of V100



CPU Server: Dual Xeon Gold 6140@2.30GHz, GPU Servers: same CPU server w/ 4x V100 PCIe | CUDA Version: CUDA 9.x| Dataset: NAMD (STMV), GTC (mpi#proc.in), MILC (APEX Medium), SPECFEM3D (four_material_simple_model) | To arrive at CPU node equivalence, we use measured benchmark with up to 8 CPU nodes. Then we use linear scaling to scale beyond 8 nodes.

V100 Deep Learning

47X Higher Throughput than CPU Server on Deep Learning Inference



Workload: ResNet-50 | CPU: 1X Xeon E5-2690v4 @ 2.6GHz | GPU: add 1X NVIDIA® Tesla® P100 or V100

Alphago

Configuration and performance

Configuration	Search threads	No. of CPU	No. of GPU	Elo rating
Single[6] p.10-11	40	48	1	2,151
Single	40	48	2	2,738
Single	40	48	4	2,850
Single	40	48	8	2,890
Distributed	12	428	64	2,937
Distributed	24	764	112	3,079
Distributed	40	1,202	176	3,140
Distributed	64	1,920	280	3,168

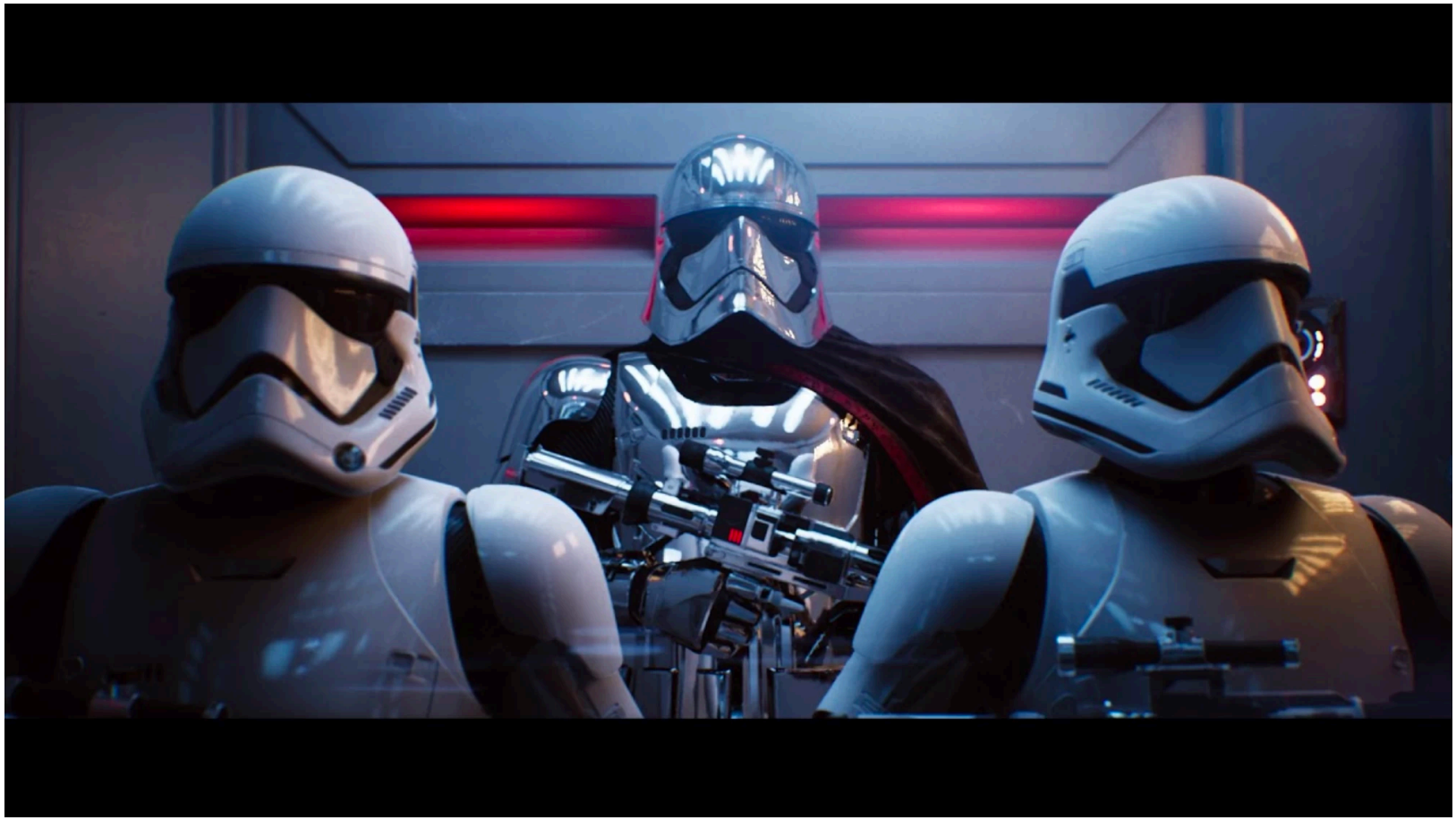
AlphaGo Zero with TPUs

Configuration and strength[21]

Versions	Playing hardware[22]	Elo rating	Matches
AlphaGo Fan	176 GPUs,[2] distributed	3,144[1]	5:0 against Fan Hui
AlphaGo Lee	48 TPUs,[2] distributed	3,739[1]	4:1 against Lee Sedol
AlphaGo Master	4 TPUs,[2] single machine	4,858[1]	60:0 against professional players; Future of Go Summit
AlphaGo Zero (40 days)	4 TPUs,[2] single machine	5,185[1]	100:0 against AlphaGo Lee 89:11 against AlphaGo Master
AlphaZero (34 hours)	4 TPUs, single machine[6]	4,430 (est.)[6]	60:40 against a 3-day AlphaGo Zero

What is the technology behind GPU processors?

It's started with 3D graphics



Parallelism

For graphics rendering, GPUs can reach amazing performance.

Characteristics of calculations:

- Simple arithmetic operations at each pixel
- Large amounts of parallelism
- Same operation needs to be performed on many different elements

Graphics processing sounds a lot like scientific computing!

Multicore processors

The mean giant.

A fast processor because:

- Pipelining of execution
- Out of order execution
- Branch prediction
- Pre-fetching
- Large amount of multilevel cache
- Hyperthreading
- …

Fast and general; does not assume anything about the computation.

Streaming processors, e.g., GPUs

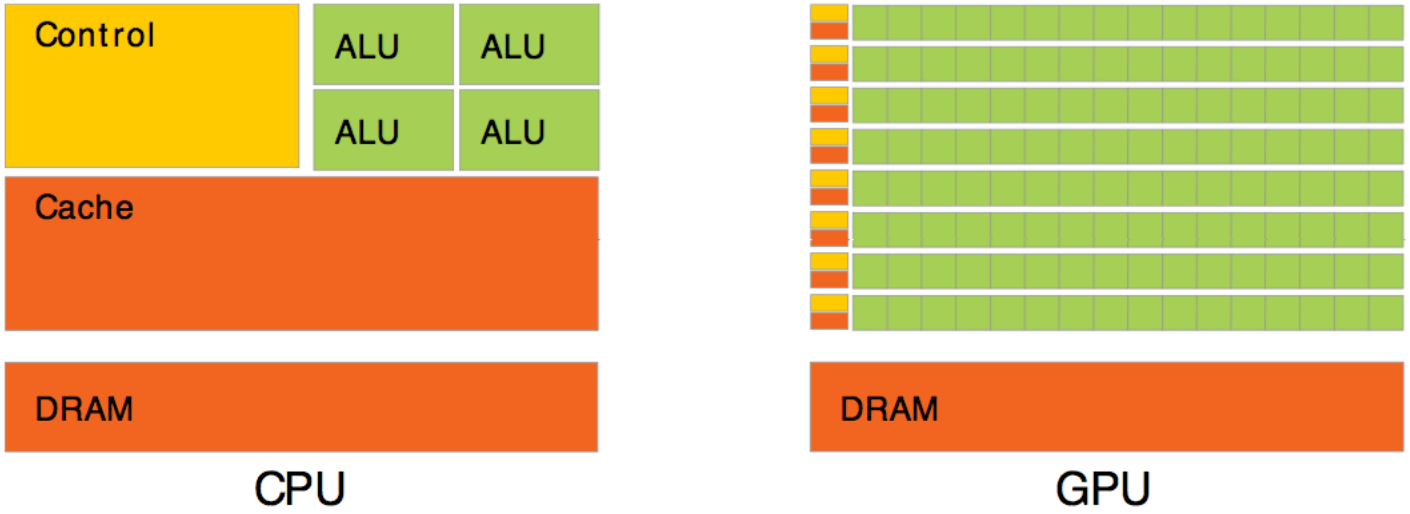
- The ants model
- Many computing units operating in parallel
- Ideal for simple but repetitive tasks.
- Example:
  - Dense linear algebra
  - Finite-difference
  - Finite-element
  - Neural network
  - Large data set processing
- Bad when calculation involves significant branching, e.g., each thread is doing something different

How is this possible?

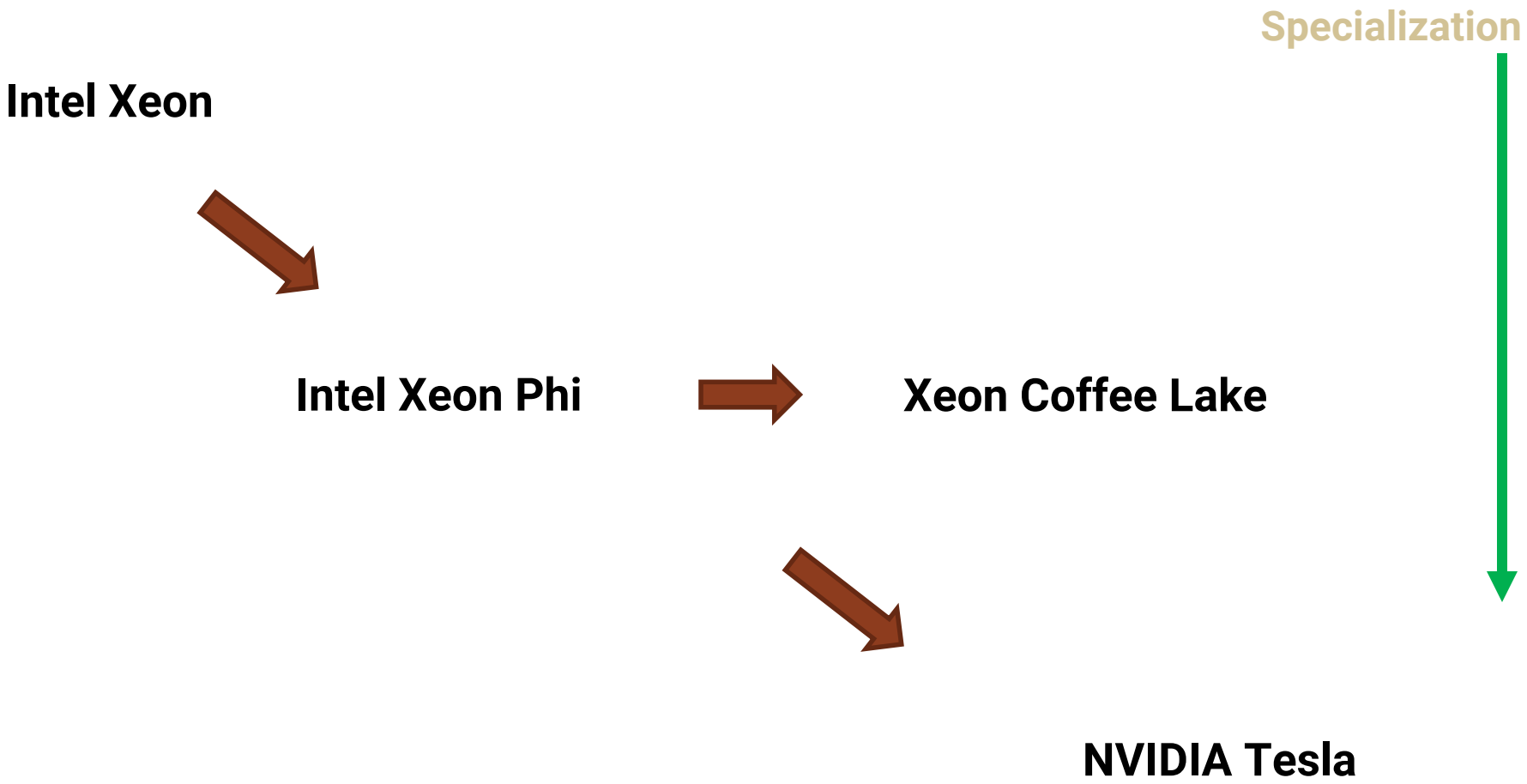
There has to be a trade-off.

More computing units means you need to give up something:

- No processor space dedicated to complex optimizations, e.g., out-of-order execution
- Cache/memory is limited because it has to be shared amongst the threads; trade memory for parallel performance
- Light threads: hardware supports the ability to switch threads every cycle; this allows a large number of threads in-flight (i.e., live)
- Limited logic for program control: 32 threads are grouped into warps; **all threads in warp execute the same instruction at the same time.**
- Each computing unit is less powerful but there are more of them.


Control
ALU
ALU
ALU
ALU
Cache
DRAM
CPU
DRAM
GPU

Modern processors

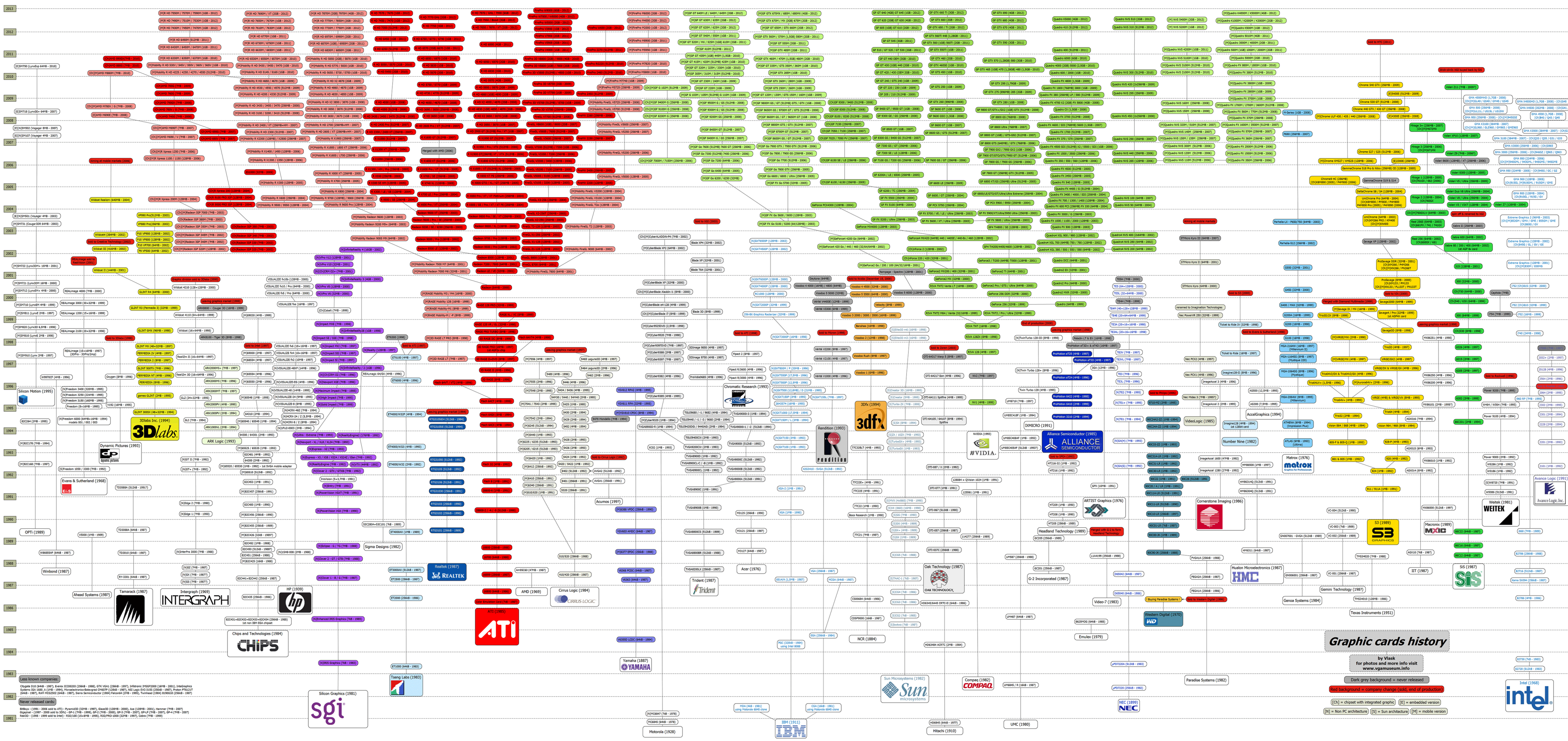


Intel micro-architecture code names

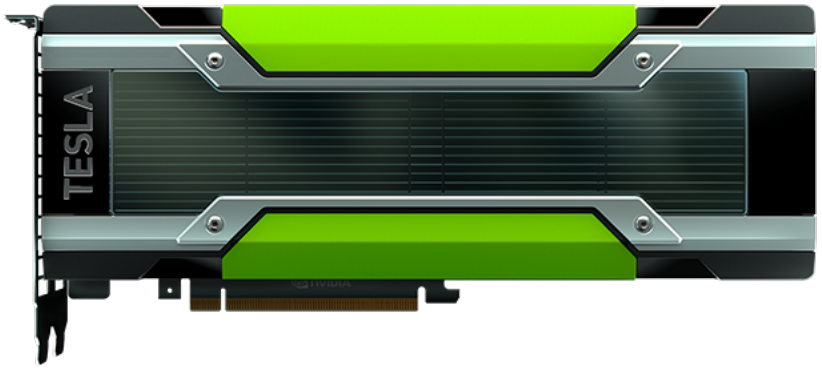
		Intel Skylake microarchitecture (Supermicro X11-Boards)				Intel Icelake microarchitecture	
code name		Skylake	Kaby Lake	Coffee Lake	Cannonlake	Icelake	Tigerlake
tick/tock or Process/Architecture/Optimization		new micro-architekture **Tock**	**Optimization**	**Optimization**	**Process**	**Architecture**	**Optimization**
process technology		14nm	14nm	14nm	10nm	10nm	10nm
CPU code name							
Desktop	Extreme / High-End / Performance	Skylake i7-67xx (LGA 1151)	Kaby Lake i7-77xx (LGA 1151)	Coffee Lake i7-87xx (LGA 1151)			
	Mainstream	Skylake i5-66xx i5-65xx i5-64xx i3-63xx i3-61xx	Kaby Lake i5-76xx i5-75xx i5-74xx i3-73xx i3-71xx	Coffee Lake i5-86xx i5-84xx i3-83xx i3-81xx			

Short History of NVIDIA GPU architectures

Micro-architecture	Release	Compute Capability	GPU code name
G70	2005		
Tesla	2006	1.0–1.3	GXX, GT2XX
Fermi	2010	2.0–2.1	GFXXX
Kepler	2012	3.0–3.7	GKXXX
Maxwell	2014	5.0–5.3	GMXXX
Pascal	2016	6.0–6.2	GPXXX
Volta	2017	7.0–7.2	GVXXX
Turing	2018	7.5	TUXXX


Graphic cards history
by Vlask
for photos and more info visit
www.vgamuseum.info

K80 on Google Cloud Platform

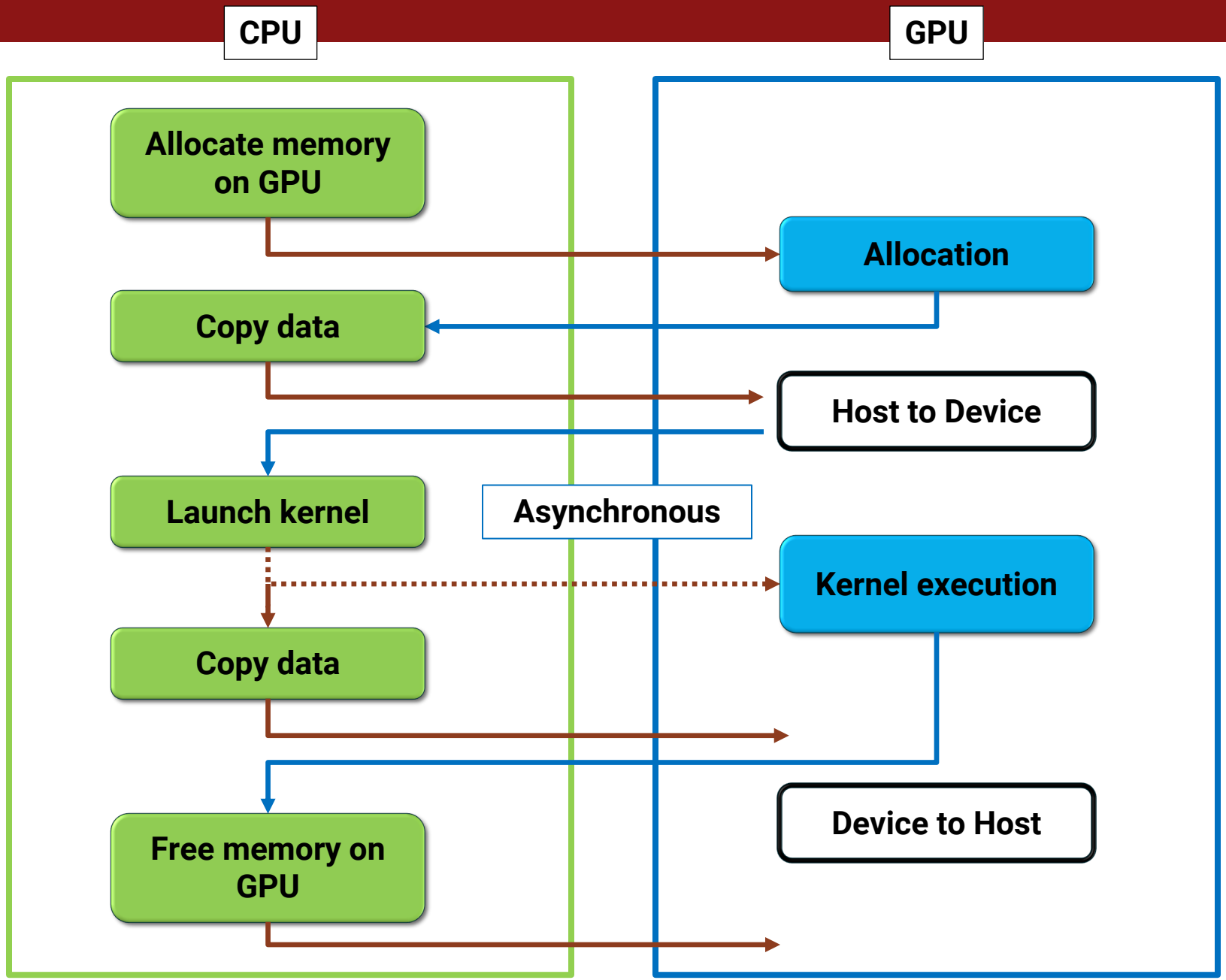


What we have

- Tesla K80
- 13 SMX
- 2,496 cores (192 per SMX)
- Total global memory: 12 GB
- Bandwidth: 241 GB/sec
- Peak performance: single 4,113 Gflop/sec; double 1,371 Gflop/sec
- Compute capability: 3.7
- Power: 300 W
- Boost clock: 824 MHz
- Architecture: Kepler 2.0
- Release: Nov 17th, 2014
- L1 Cache: 16 KB (per SMX)
- L2 Cache: 1,536 KB

Host and coprocessor

- NVIDIA GPUs are different from conventional processors.
- They only work as **co-processors.**
- This means you need a host processor (e.g., Intel Xeon).
- Your program runs on the host and uses the CUDA API to move data back and forth to the GPU and run programs on the GPU.
- You cannot log on the GPU directly or run an OS on the GPU.


CPU
GPU
Allocate memory on GPU
Allocation
Copy data
Host to Device
Launch kernel
Asynchronous
Kernel execution
Copy data
Device to Host
Free memory on GPU

GPU architecture

You cannot program a GPU without understanding the architecture of the processor.

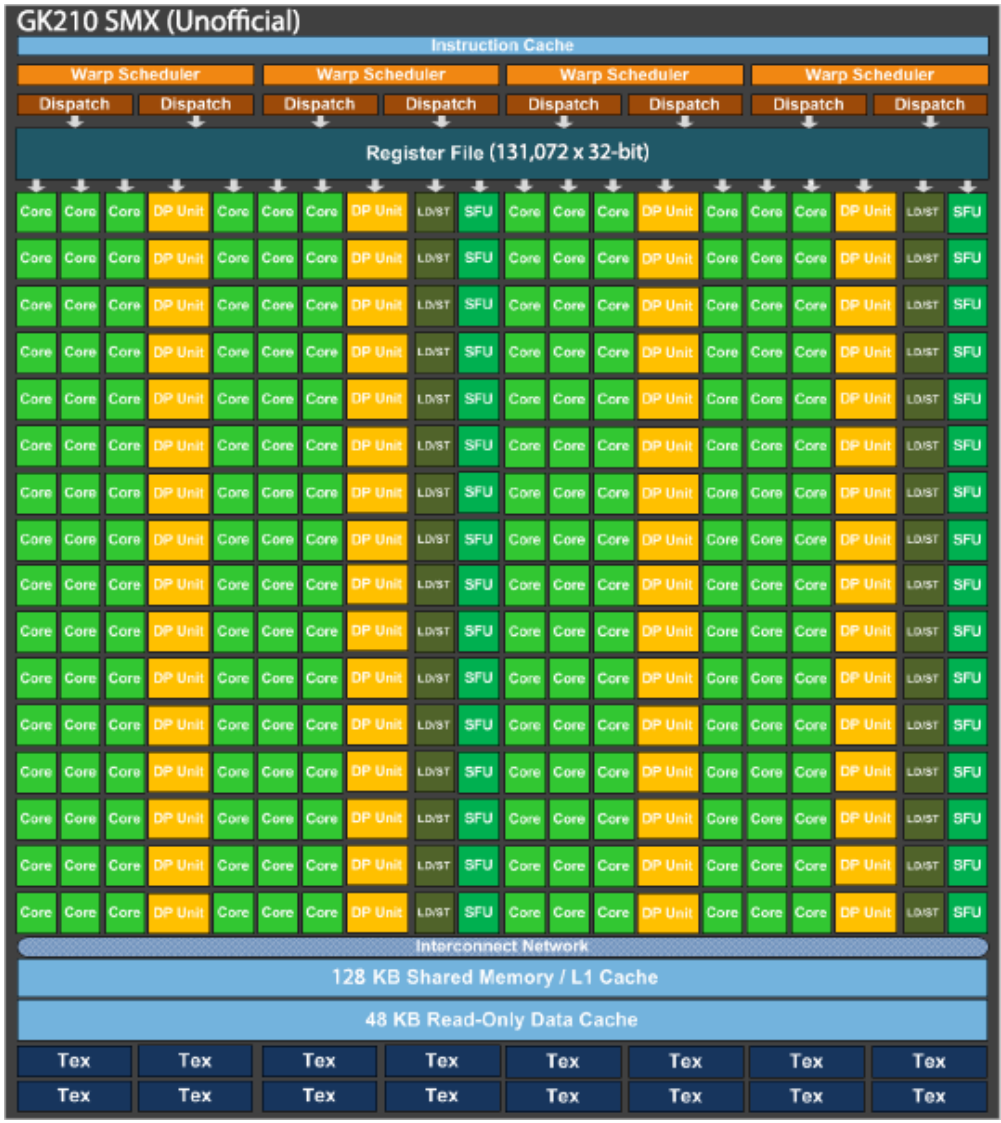
1 instruction buffer + warp scheduler

Large register file

192 cores

Kepler Streaming Multiprocessor (SM)

Shared memory



Global view

Common L2 cache

No. of cores:
(16x12) x (8+7) = 2,880 cores!

To memory

To memory

Volta!
More, Better!


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

80 SM; 5,120 CUDA cores; 640 tensor cores

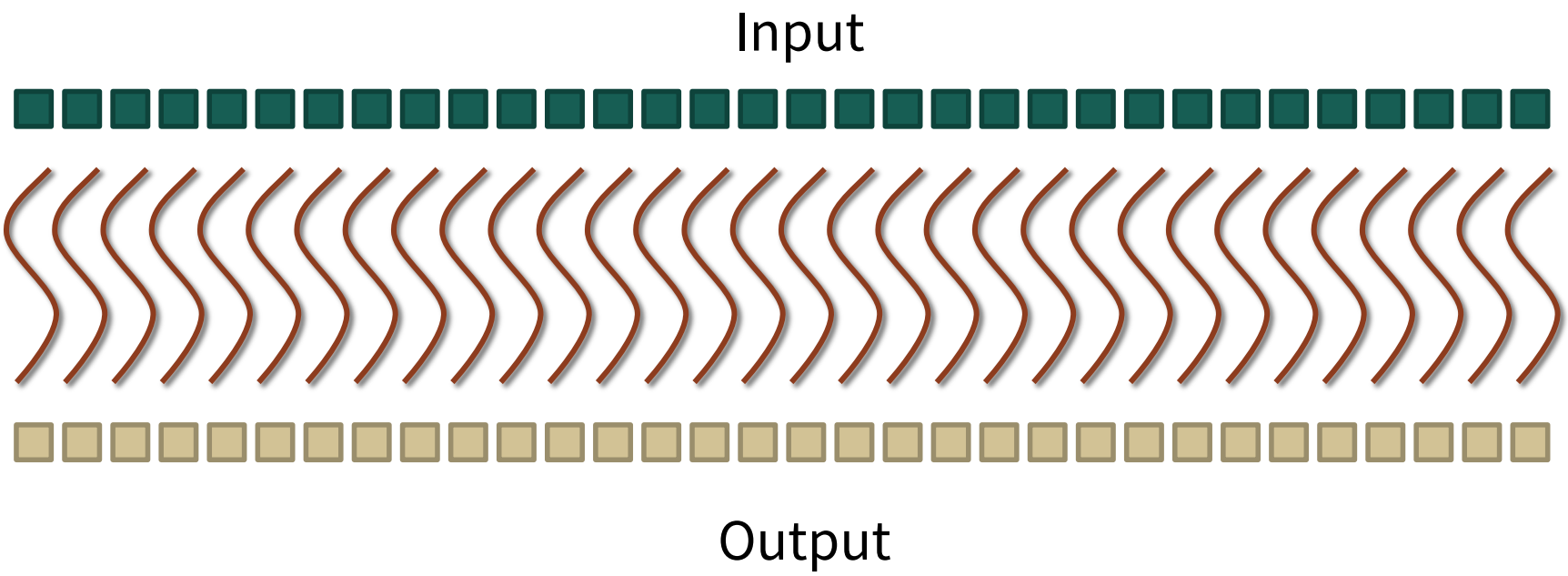
Basic breakdown

To program GPUs you need to understand the basic breakdown, at least as a programming model:

- Bottom-most level: 32 cores = 32 threads, grouped in a warp. The hardware is optimized to have all threads in a warp execute the same instruction at the same time. SIMT (single instruction multiple threads) model.
- Groups of warps form a block. A block executes on 1 SM (streaming multiprocessor). Threads within a block have some ability to exchange data and synchronize.
- Collection of many blocks constitute the “entire dataset” that will be operated on by a kernel.

Execution model

- Calculations on a GPU always follow the same model.
- This is very constrained.
- GPUs run kernels that are designed to execute calculations in the following way.



Code example

- Let's start with a very basic example
  `firstProgram.cu`

- Log onto GCP!

```
$ gcloud compute ssh hw3
$ gcloud compute scp * hw3:~/.
```

Compile with:

```
$ make
```

or:

```
$ nvcc --compiler-options -Wall -arch=sm_35 -o test test.cu
```

Code overview

```
int* d_output;
cudaMalloc(&d_output, sizeof(int) * N);
kernel<<<1, N>>>(d_output);
vector<int> h_output(N);
cudaMemcpy(&h_output[0], d_output, sizeof(int) * N, cudaMemcpyDeviceToHost);
for(int i = 0; i < N; ++i)
    printf("Entry %3d, written by thread %2d\n", h_output[i], i);
cudaFree(d_output);
```

Function to execute

Function arguments

```
kernel<<<1, N>>>(d_output);
```

Number of threads to use

Device kernels

```
__device__ __host__
int f(int i) {
    return i*i;
}

__global__
void kernel(int* out) {
    out[threadIdx.x] = f(threadIdx.x);
}
```

`__global__ __host__ __device__`

What are these mysterious keywords?

`__global__` kernel will be
- Executed on the device
- Callable from the host

`__host__` kernel will be ← Compiled for host
- Executed on the host
- Callable from the host

`__device__` kernel will be ← Compiled for device
- Executed on the device
- Callable from the device only

Run the following!

```
./firstProgram 1
./firstProgram 32
./firstProgram 1024
./firstProgram 1025
```

```
darve@hw3:~$ ./getProperties
CUDA Device Query...
There are 1 CUDA devices.

CUDA Device #0
Major revision number:          3
Minor revision number:          7
Name:                           Tesla K80
Total global memory:            11996954624
Total shared memory per block: 49152
Total registers per block:      65536
Warp size:                      32
Maximum memory pitch:           2147483647
Maximum threads per block:      1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                     823500
Total constant memory:          65536
Texture alignment:              512
Concurrent copy and execution: Yes
Number of multiprocessors:      13
Kernel execution timeout:       Yes
ECC enabled:      Yes
```

> 1024 ?

We will see on Wednesday how to use more threads and run larger cases.