CME 213

Spring 2017

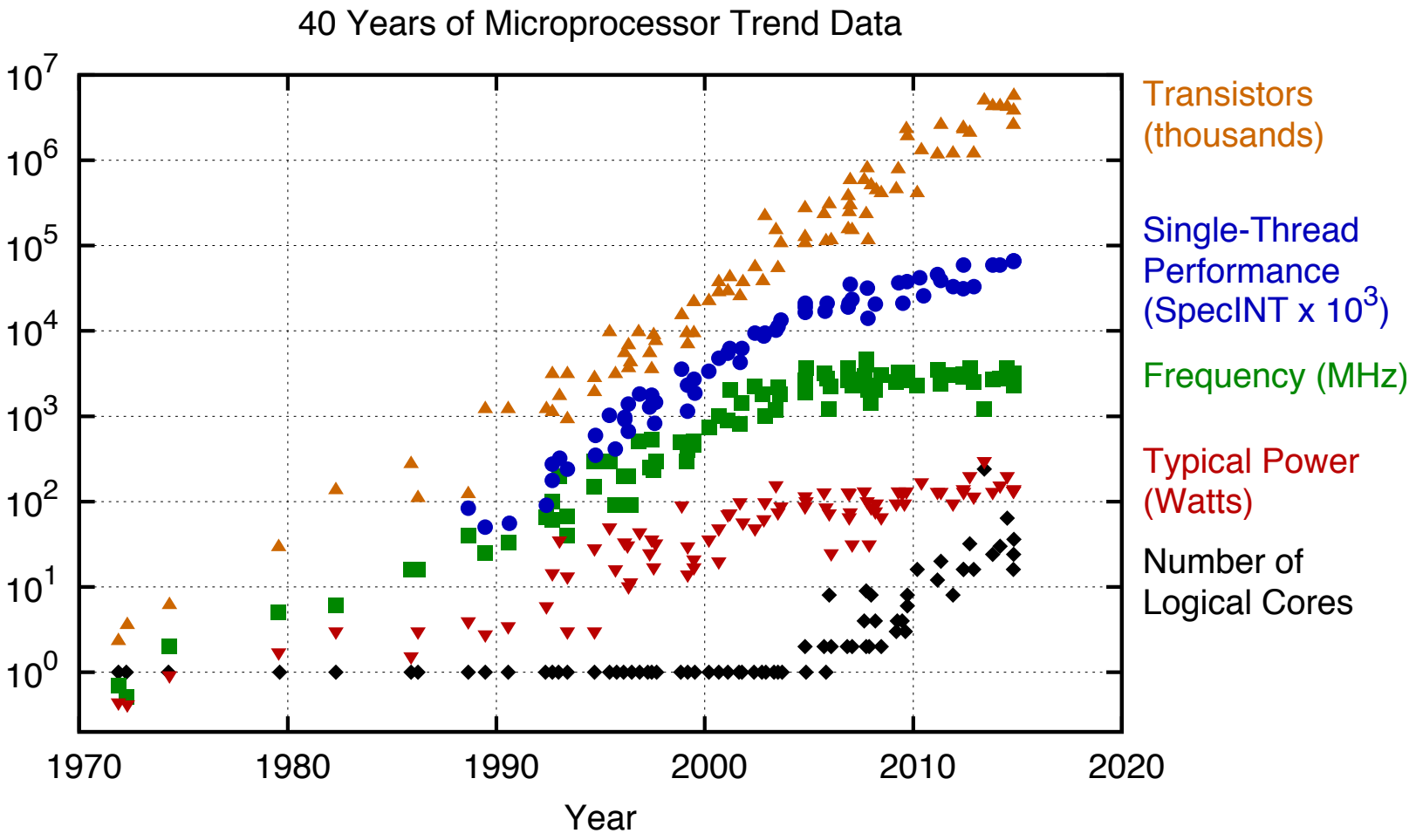
Eric Darve

Summary of previous lectures

- Pthreads: low-level multi-threaded programming
- OpenMP: simplified interface based on `#pragma`, adapted to scientific computing
- OpenMP `for` and scheduling (`static`, `dynamic`)
- Reduction and atomic
- Data layout and vectorization (SoA vs AoS)
- Arithmetic intensity and roofline model
- NUMA and first-touch allocation

GPU computing!

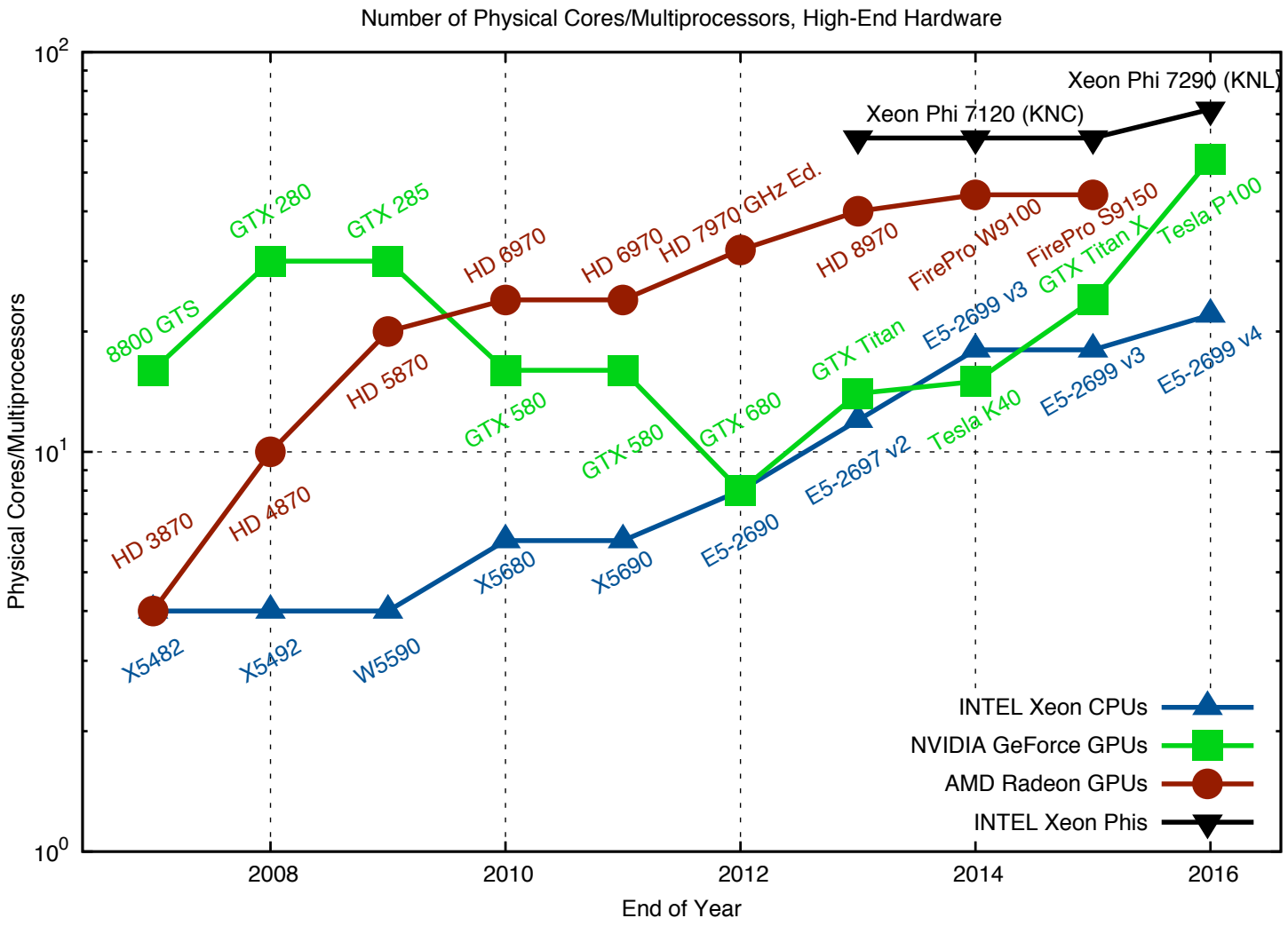
Performance numbers (Karl Rupp)



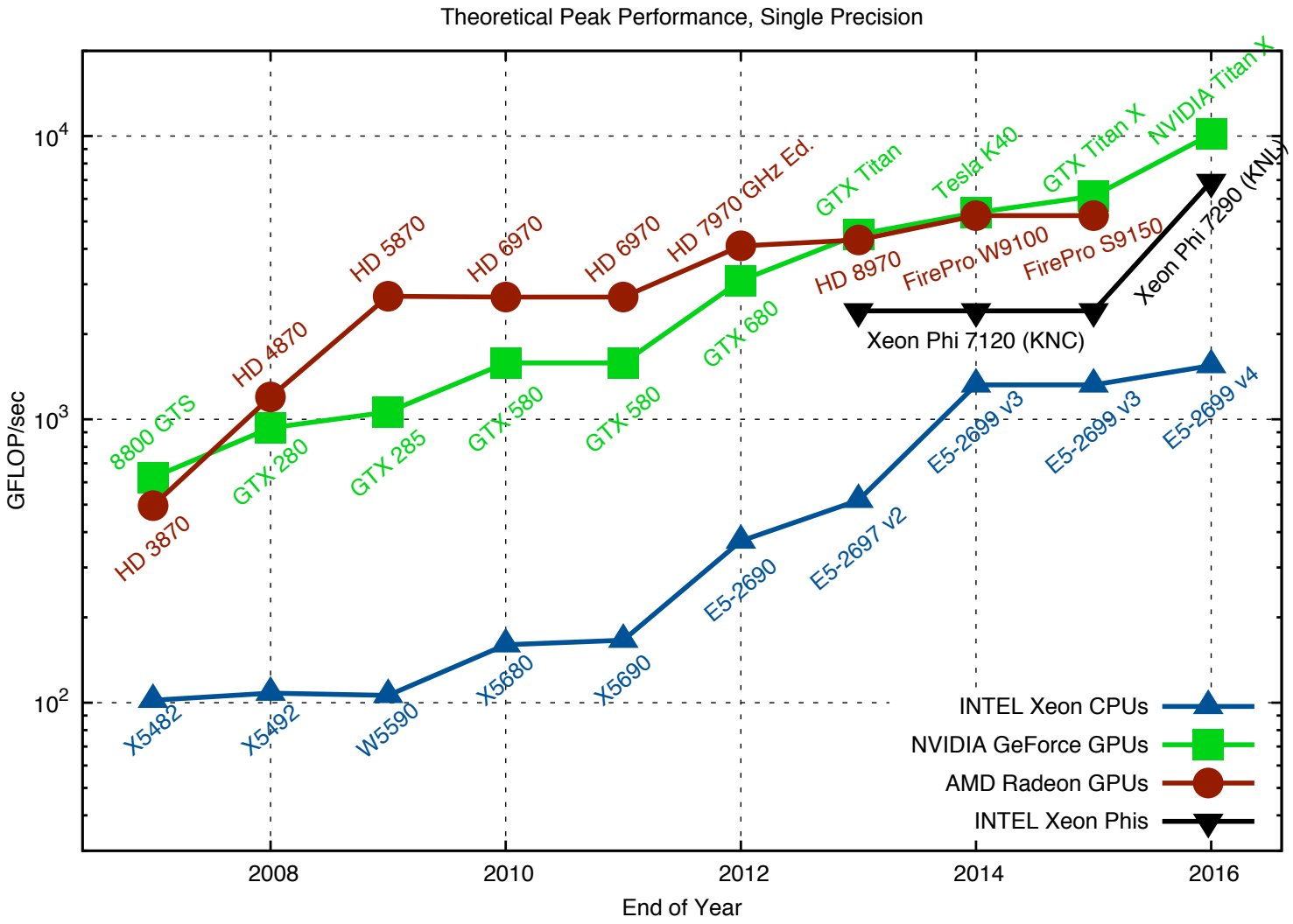
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/
https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

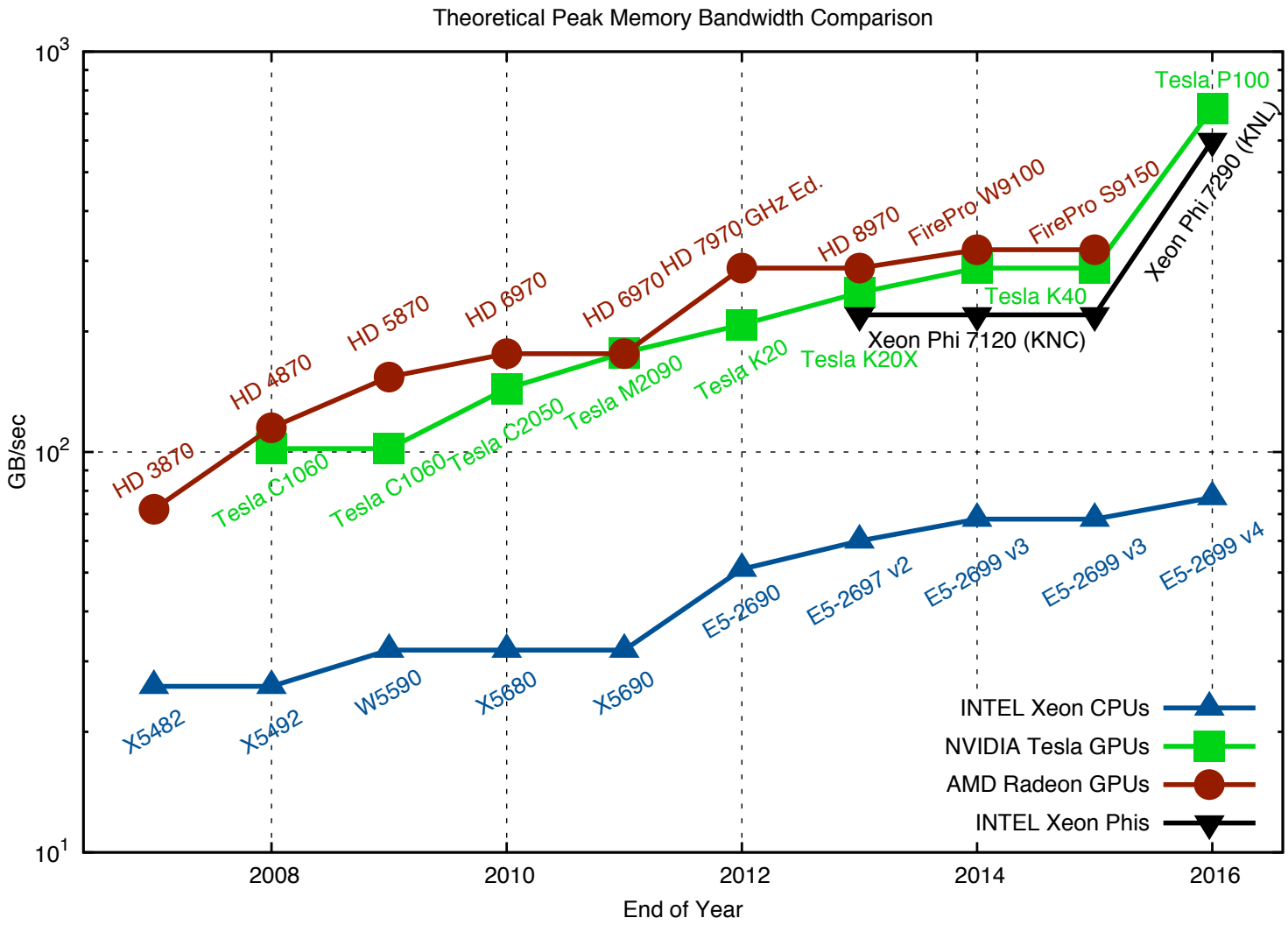
Key to performance: lots of cores!



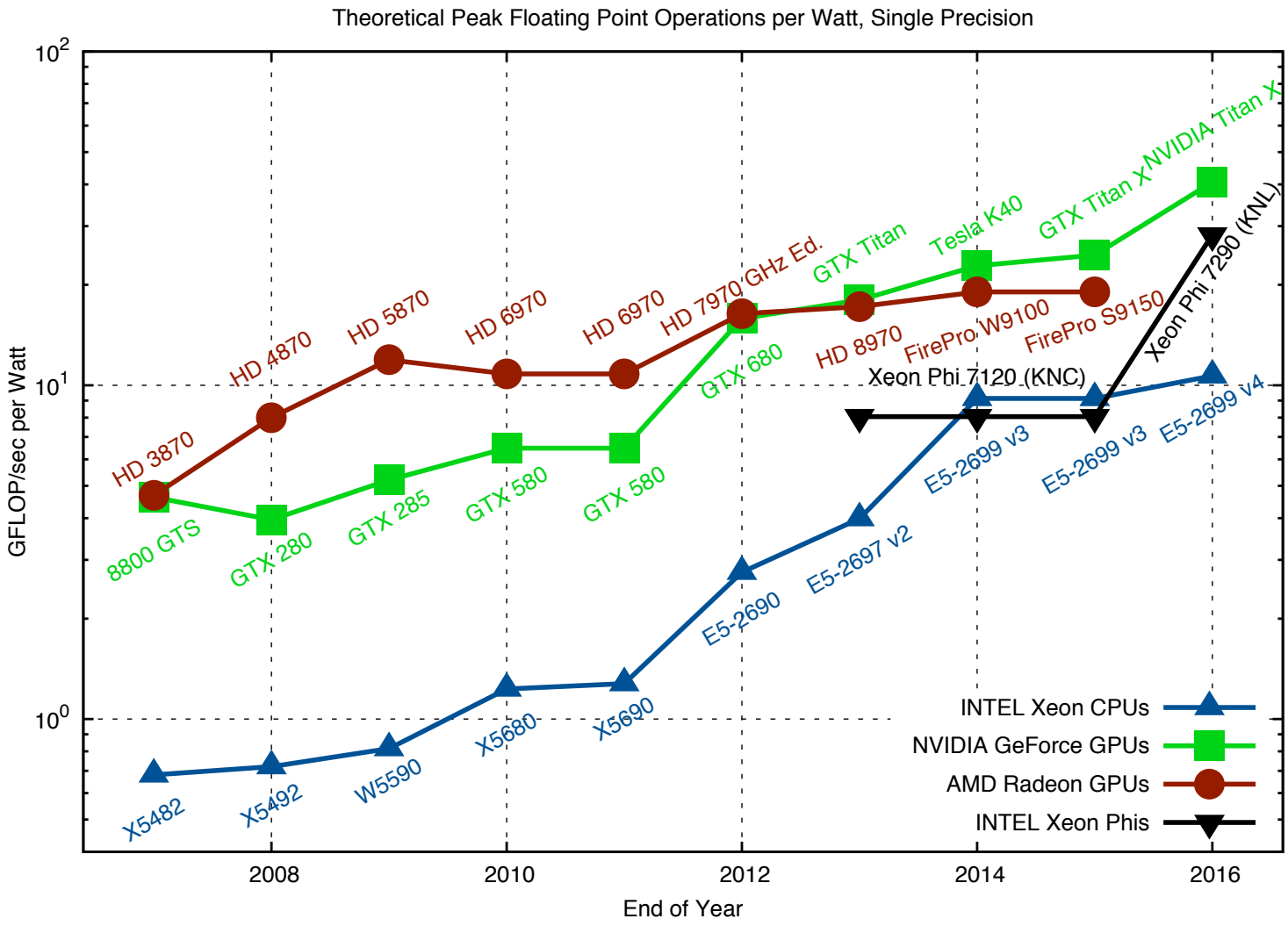
Peak performance: single precision



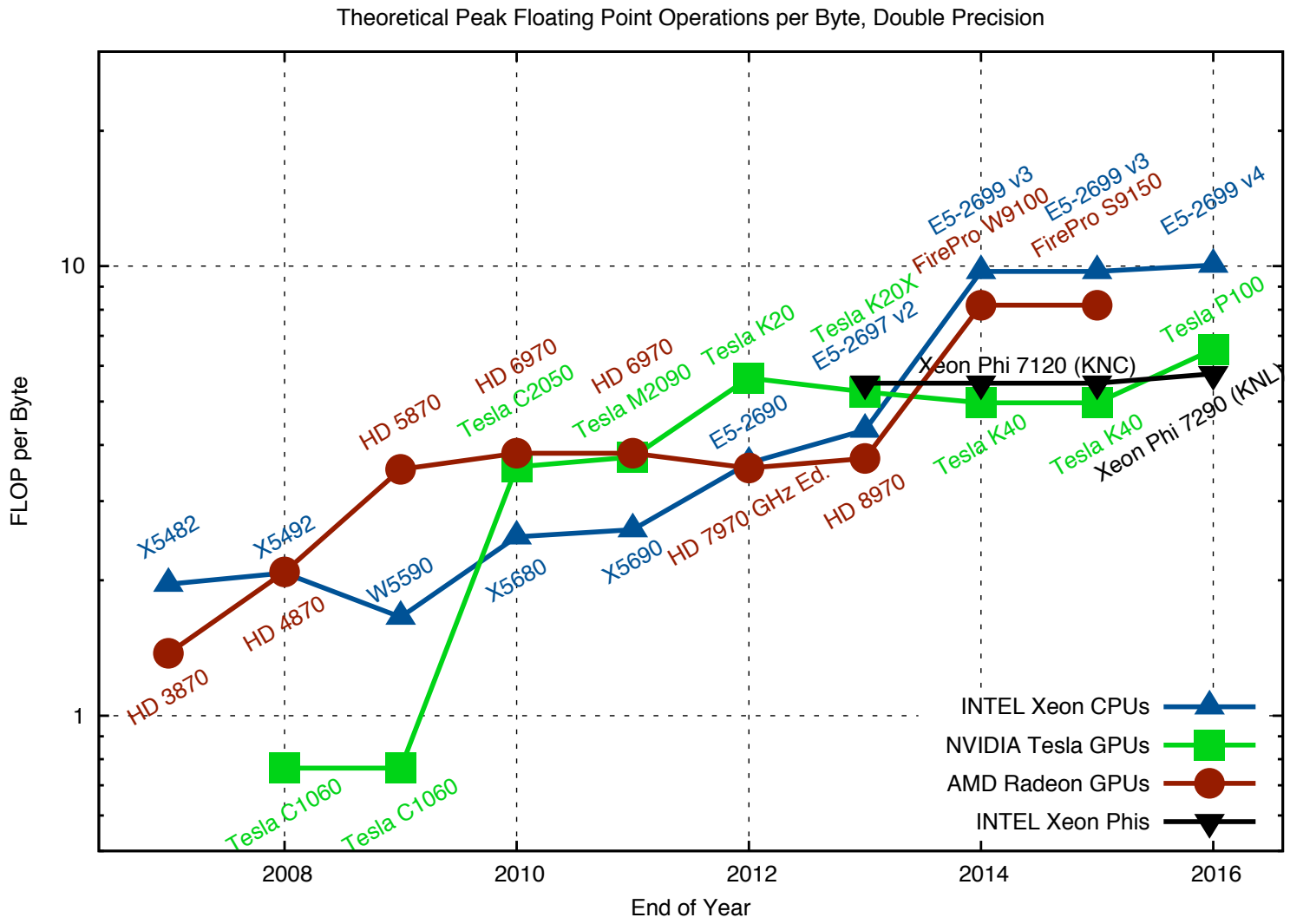
Memory is key for many applications



To build supercomputers, perf/watt is the relevant metric



Arithmetic intensity: are you memory or compute bound?



t's the main bottleneck in most scientific co

Number of flops

Data traffic

Concu...

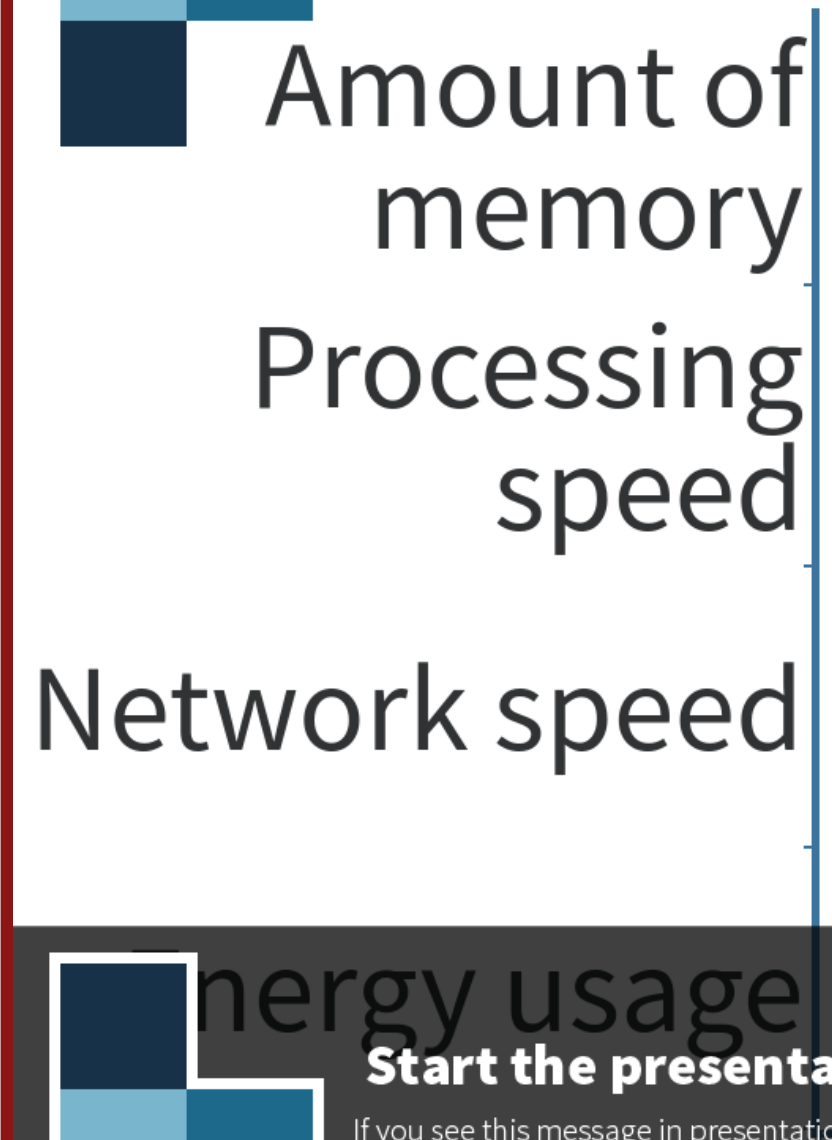
0

Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

hat is a major issue in building a supercompu



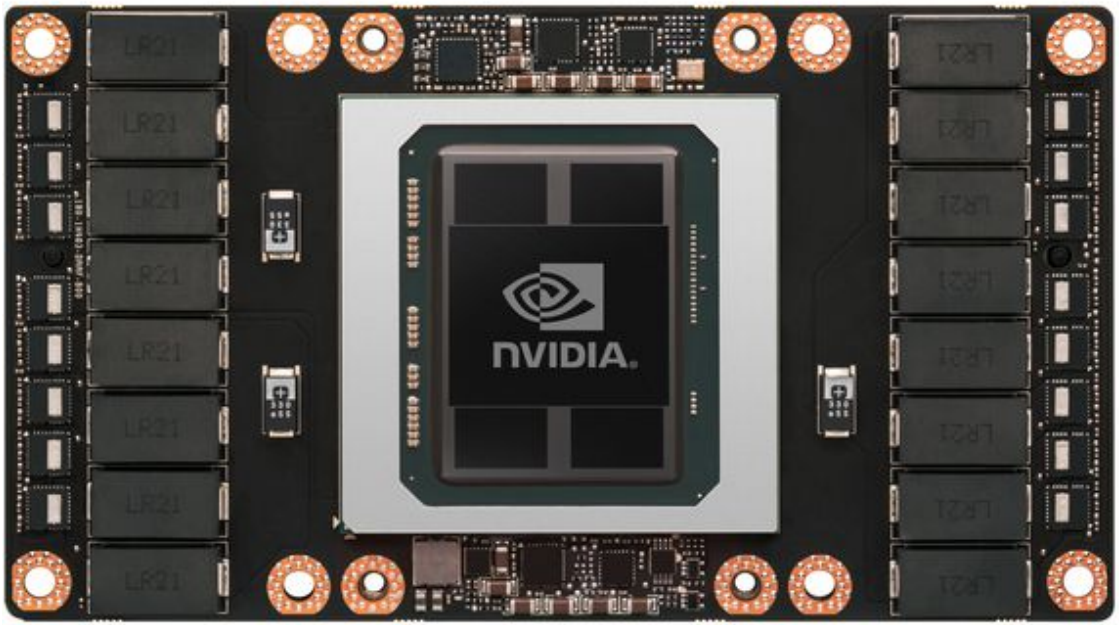
Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

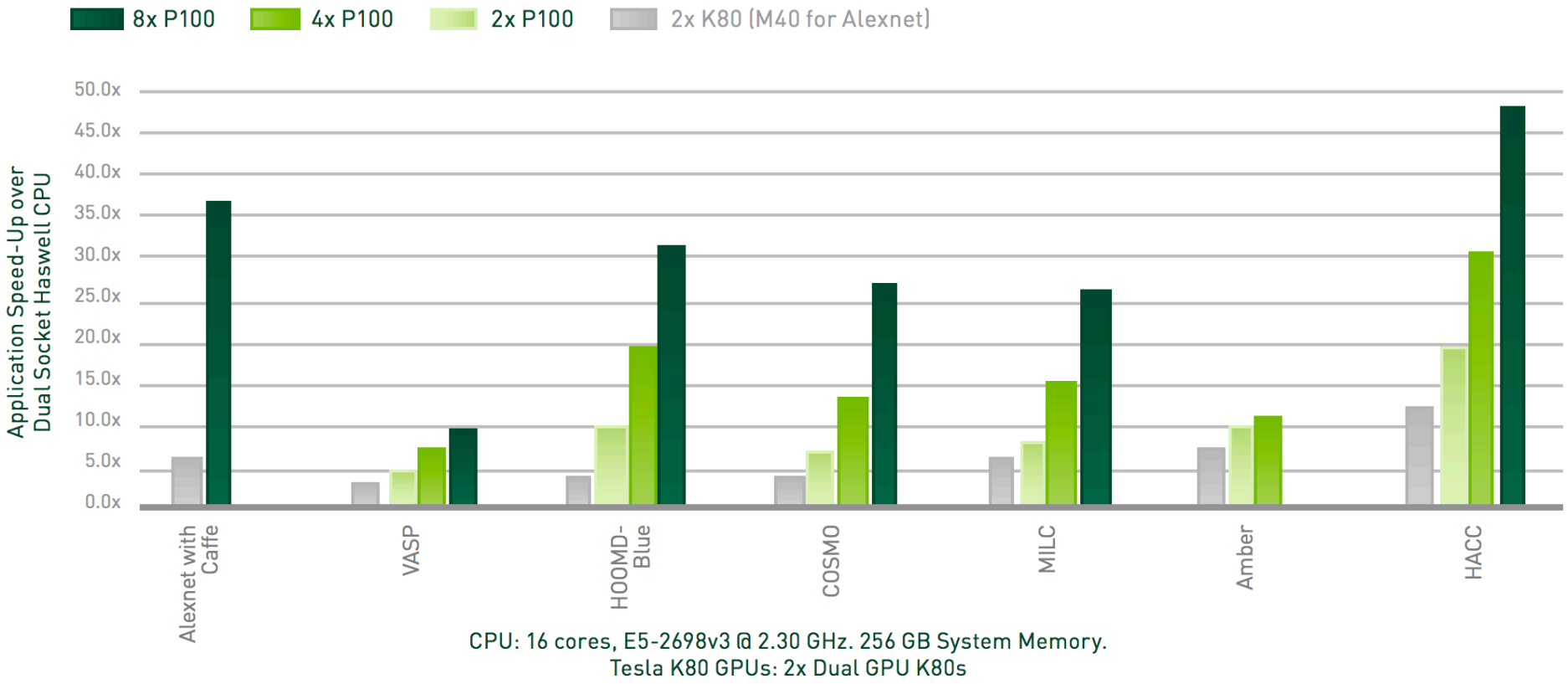
Total Results: 0

Performance for scientific applications

- Example: Tesla P100
- 5.3 teraflops double-precision performance
- 10.6 teraflops single-precision performance
- 732 GB/sec memory bandwidth



Performance of P100



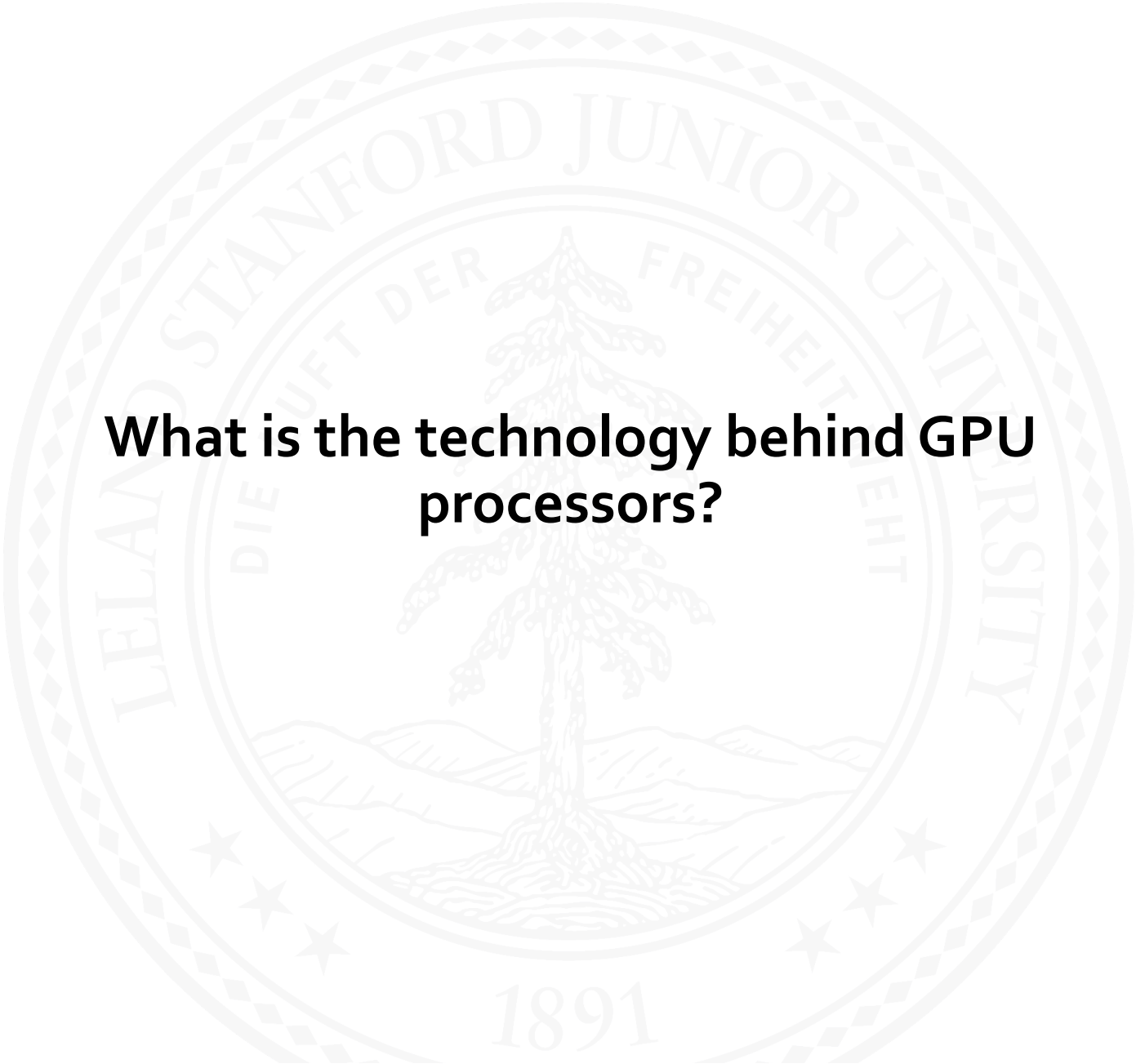
Deep Learning | Quantum Mechanics | Particle | Weather | Quantum ChromoDynamics | Molecular Dynamics | Cosmology

Alphago

1,202 CPUs and 176 GPUs

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



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.

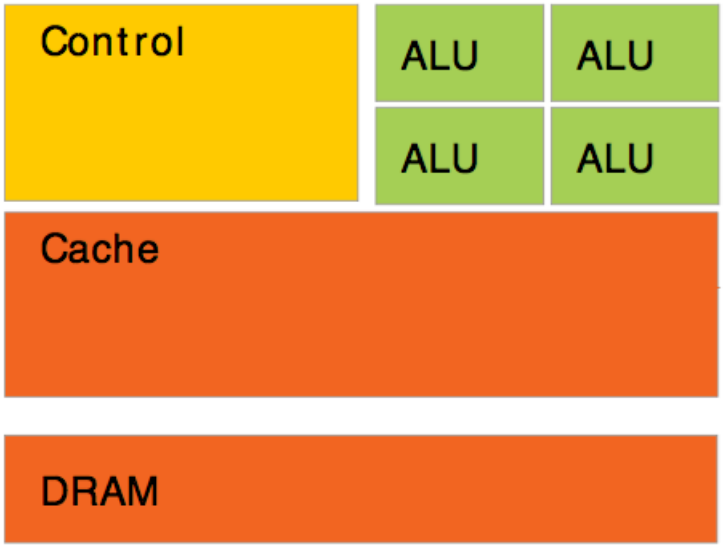
Streaming processors, e.g., GPUs

- 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

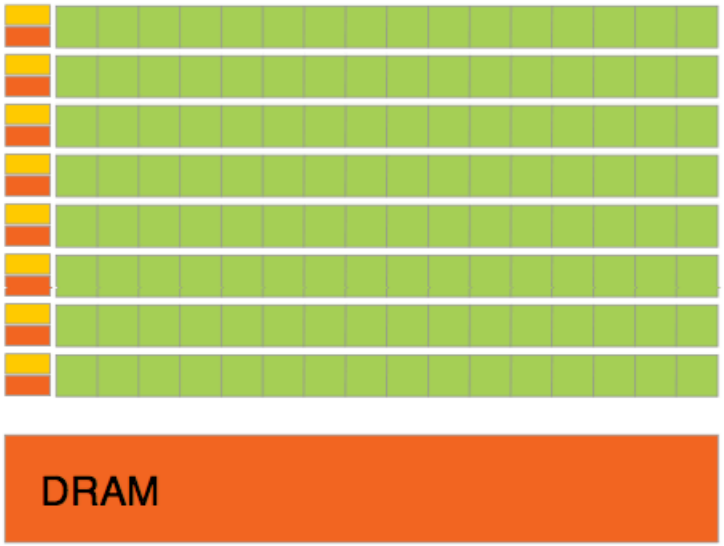
How is this possible?

- There has to be a trade-off.

- More computing units means you need to give up something:
- No processor space is dedicated to complex optimizations, e.g., out-of-order execution
- Cache space is limited because it has to be shared amongst the threads
- Light threads: hardware supports the ability to switch threads every cycle; this allows a large number of threads in-flight or 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.

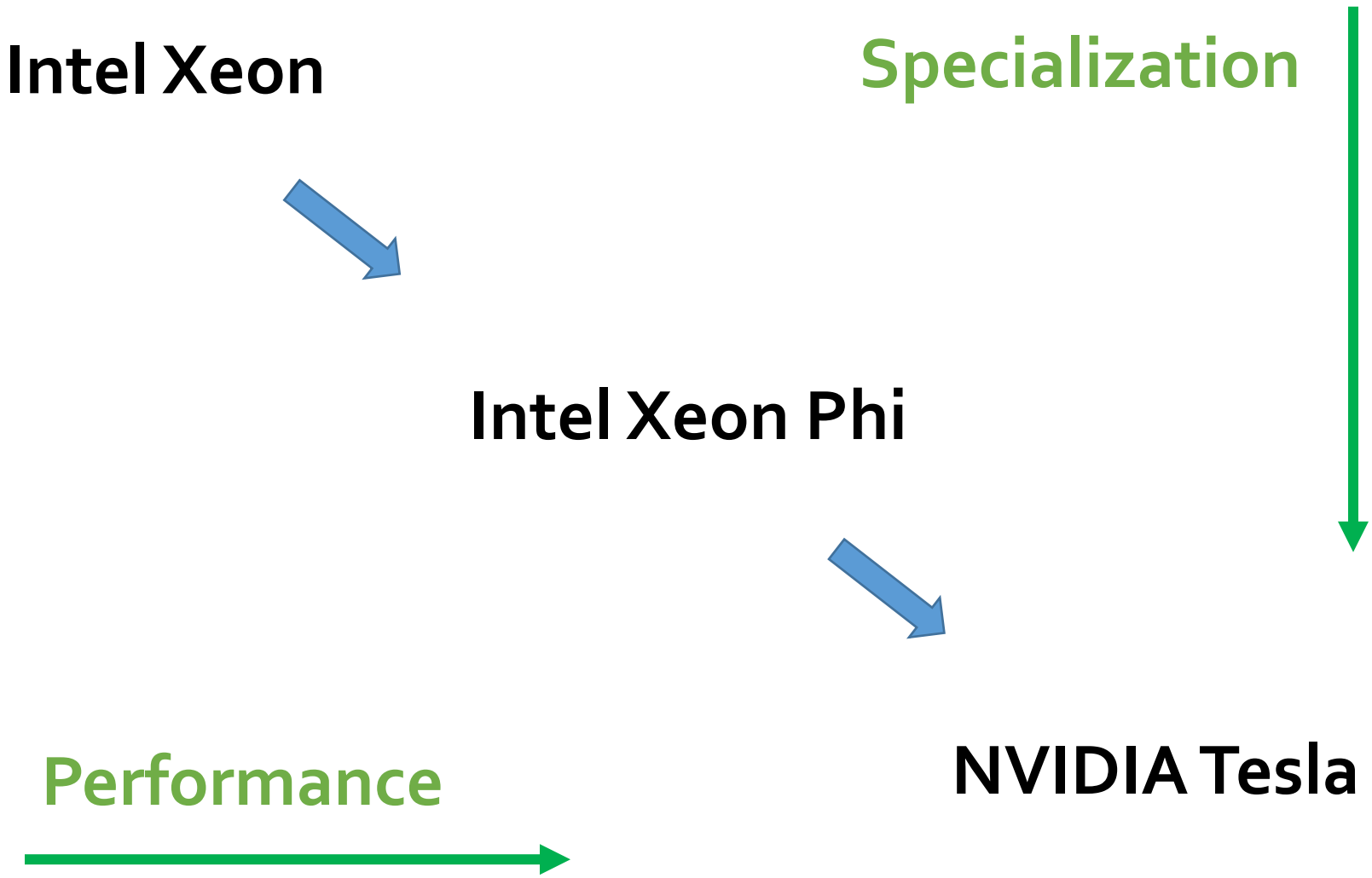


CPU



GPU

Modern processors



Short History of NVIDIA GPU architectures

- G70: 2006
- Tesla: 2006
- Fermi: 2010
- Kepler: 2012
- Maxwell: 2014
- Pascal: 2016

Graphic cards history
by Vlask
for photos and more info visit
www.vgamuseum.info
Dark grey background = never released
Red background = company change (sold, end of production)
[Ch] = chipset with integrated graphic
[E] = embedded version
[N] = Non PC architecture
[S] = Sun architecture
[M] = mobile version

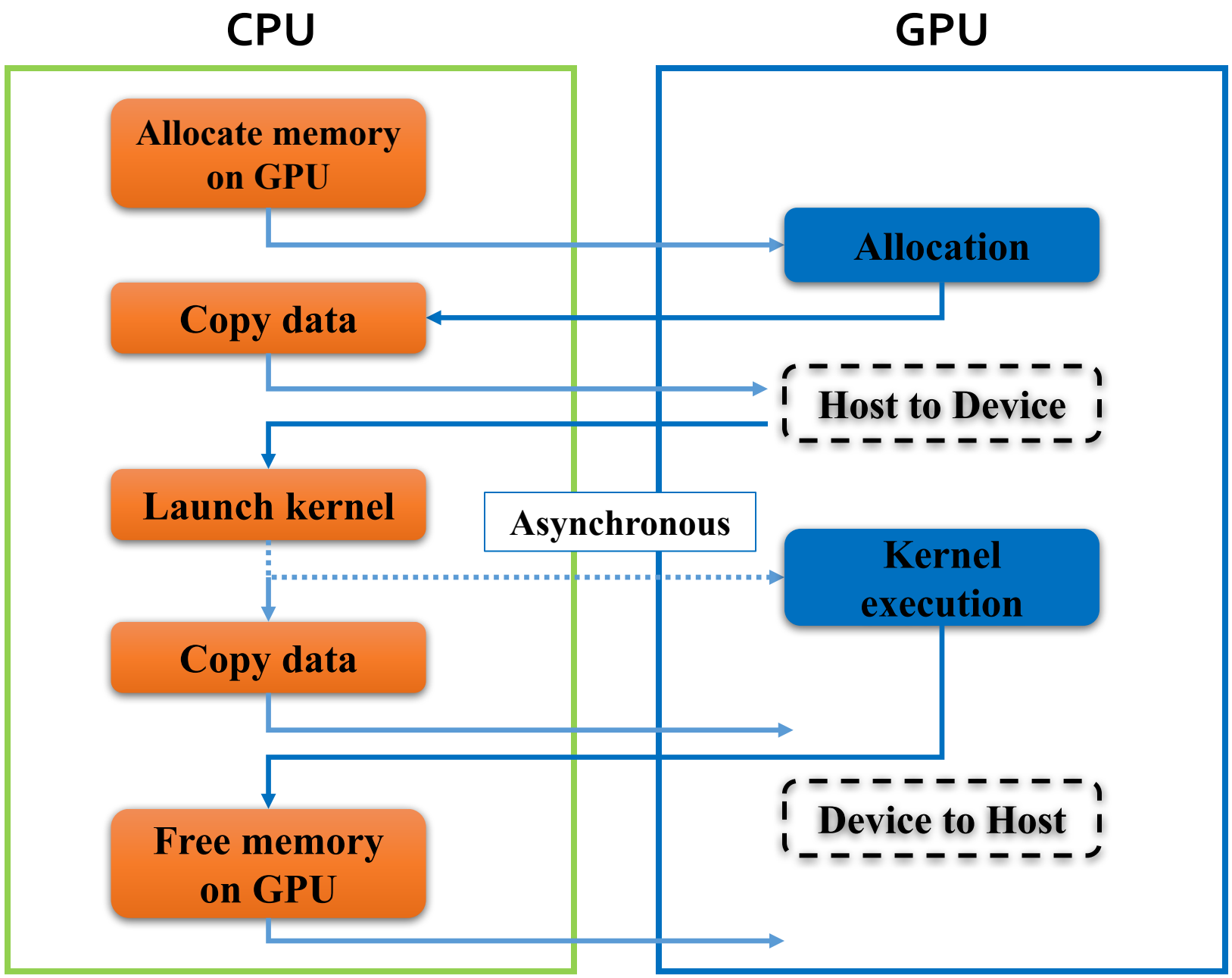
Certainty

What we have:

- Fermi M2050; 4 GPUs/node
- Release date: July 2011
- Clock rate: 1.147 GHz
- 14 multiprocessors
- 448 cores
- Total global memory: 2.8 GB
- Bandwidth: 148 GB/sec
- Peak performance: single 1.03 Tflop/sec; double 515 Gflop/sec
- Compute capability: 2.0

Host and coprocessor

- Xeon Phi accelerators support a full-fledged operating system like Linux.
- NVIDIA GPUs are different. They only work as a co-processor.
- 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.

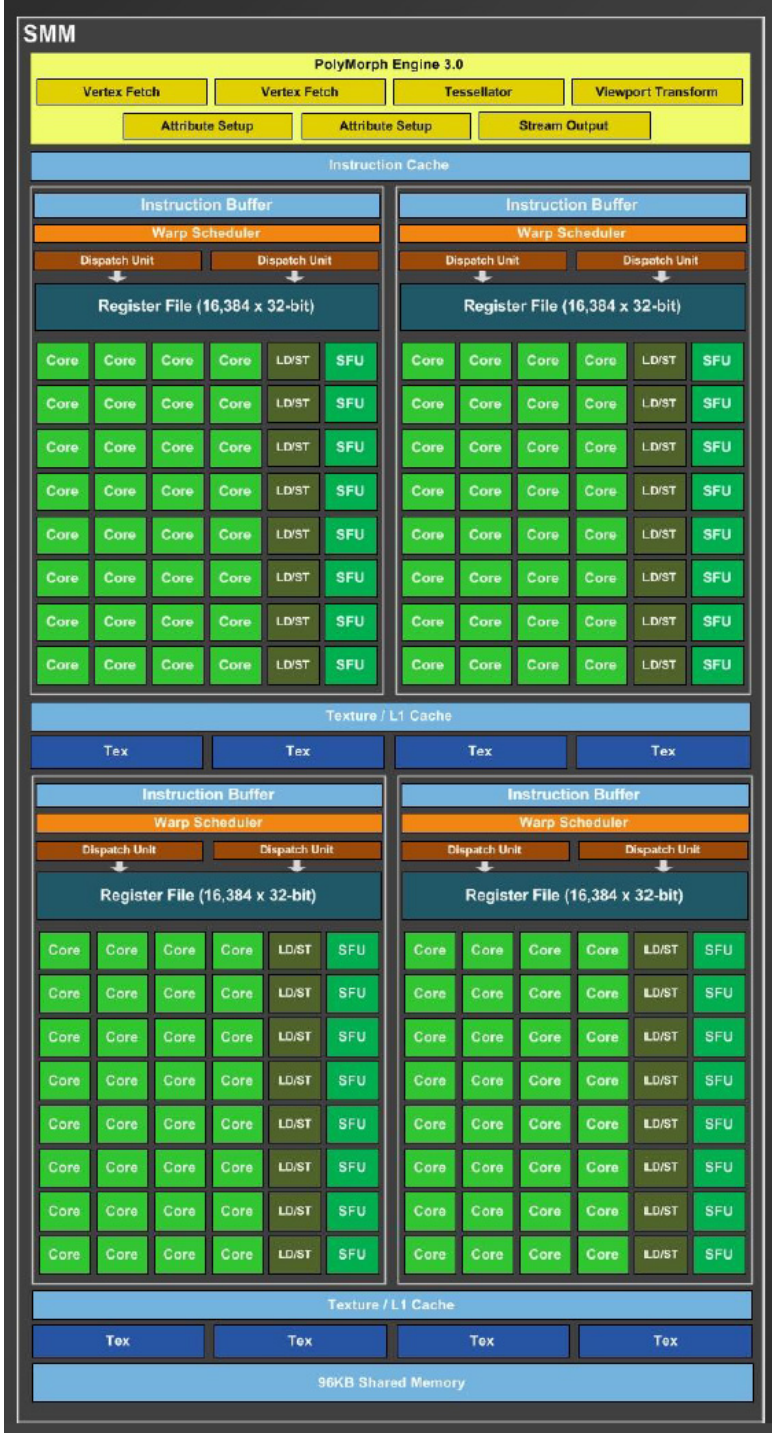
1 instruction buffer + warp scheduler

Large register file

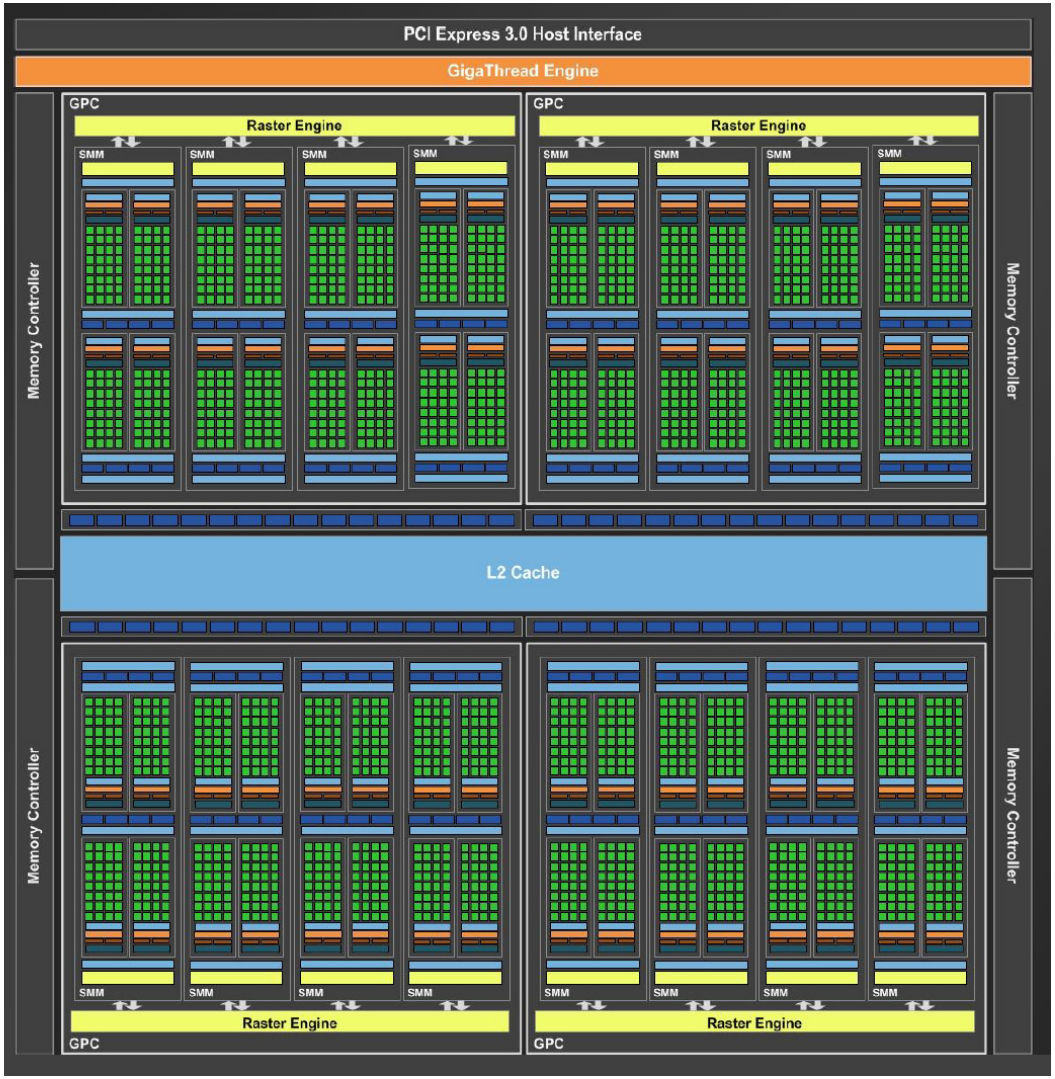
32 cores

Maxwell architecture
Maxwell Streaming Multiprocessor (SM)

Shared memory



Global view



Common L2 cache

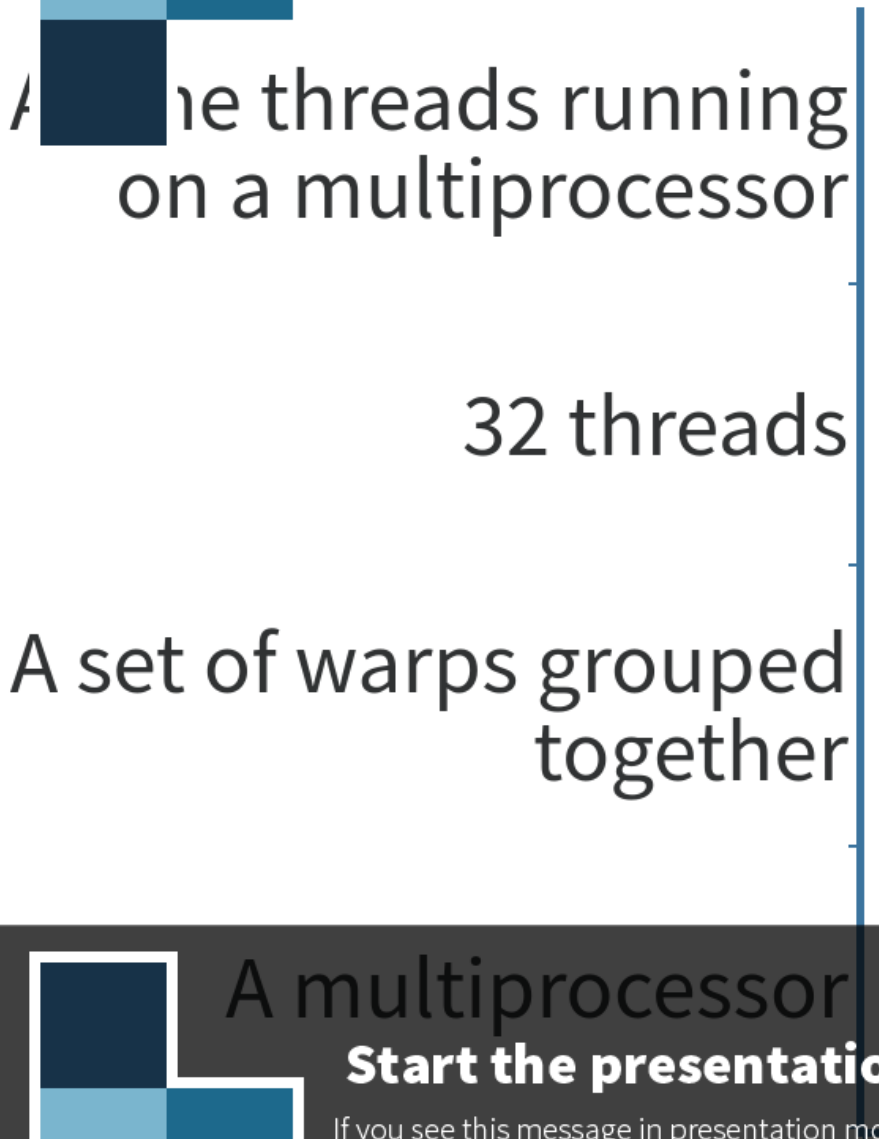
No. of cores:
(8x4x2) x 8 x 4
= 2,048 cores!

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.
- All the blocks constitute the "entire dataset" that will be operated on by a kernel.

What's a CUDA thread block?



Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

What does SIMT mean?

Full efficiency is reached when 32 threads execute the same instruction

All threads execute the same instruction

Threads can execute vector instructions efficiently

0

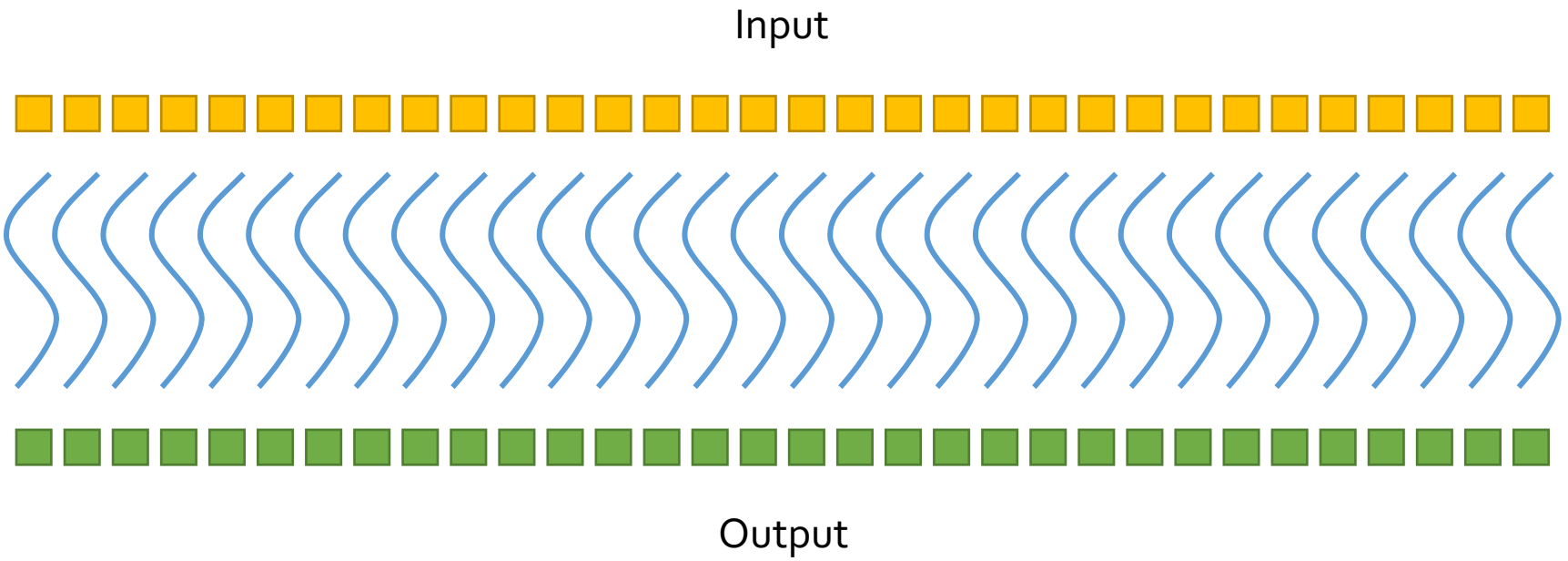
Start the presentation to activate live content

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app

Total Results: 0

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

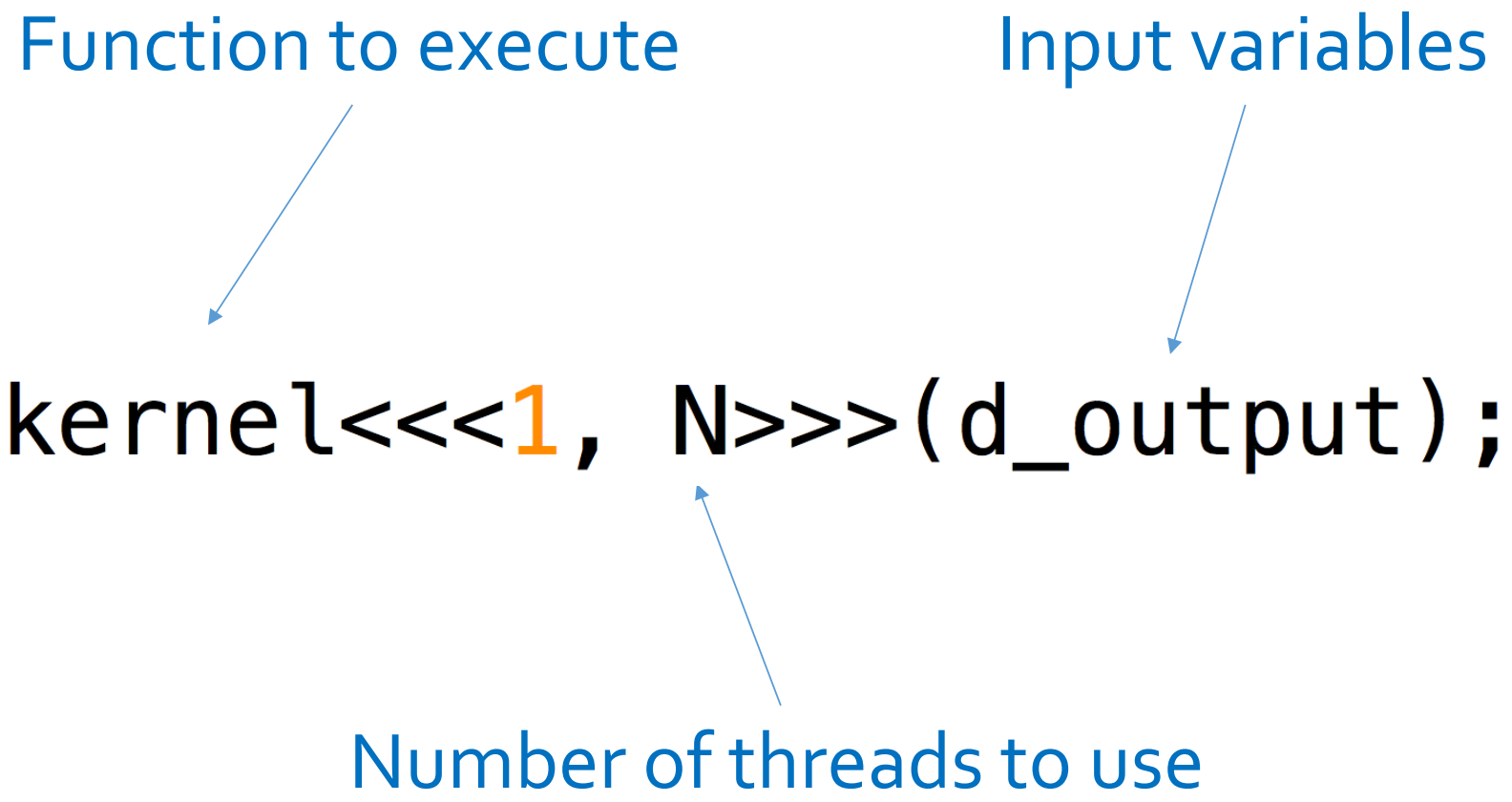
`firstProgram.cu`

`.bashrc:`

```
# User specific aliases and functions
module load null
module load gnu/4.9.2
module load cuda/7.5.18
```

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
Input variables
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
- Executed on the host
- Callable from the host

Compiled for host

__device__ kernel will be
- Executed on the device
- Callable from the device only

Compiled for device

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

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