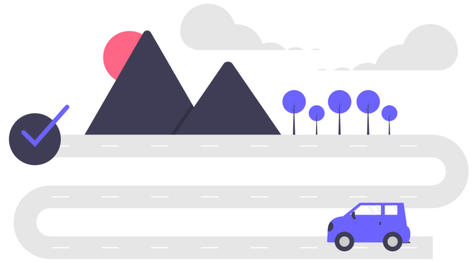
CME 213, ME 339 Spring 2023
Introduction to parallel computing using MPI, openMP, and CUDA

Eric Darve, ICME, Stanford

"There are two ways to write error-free programs; only the third one works." (Alan J. Perlis)

Stanford University

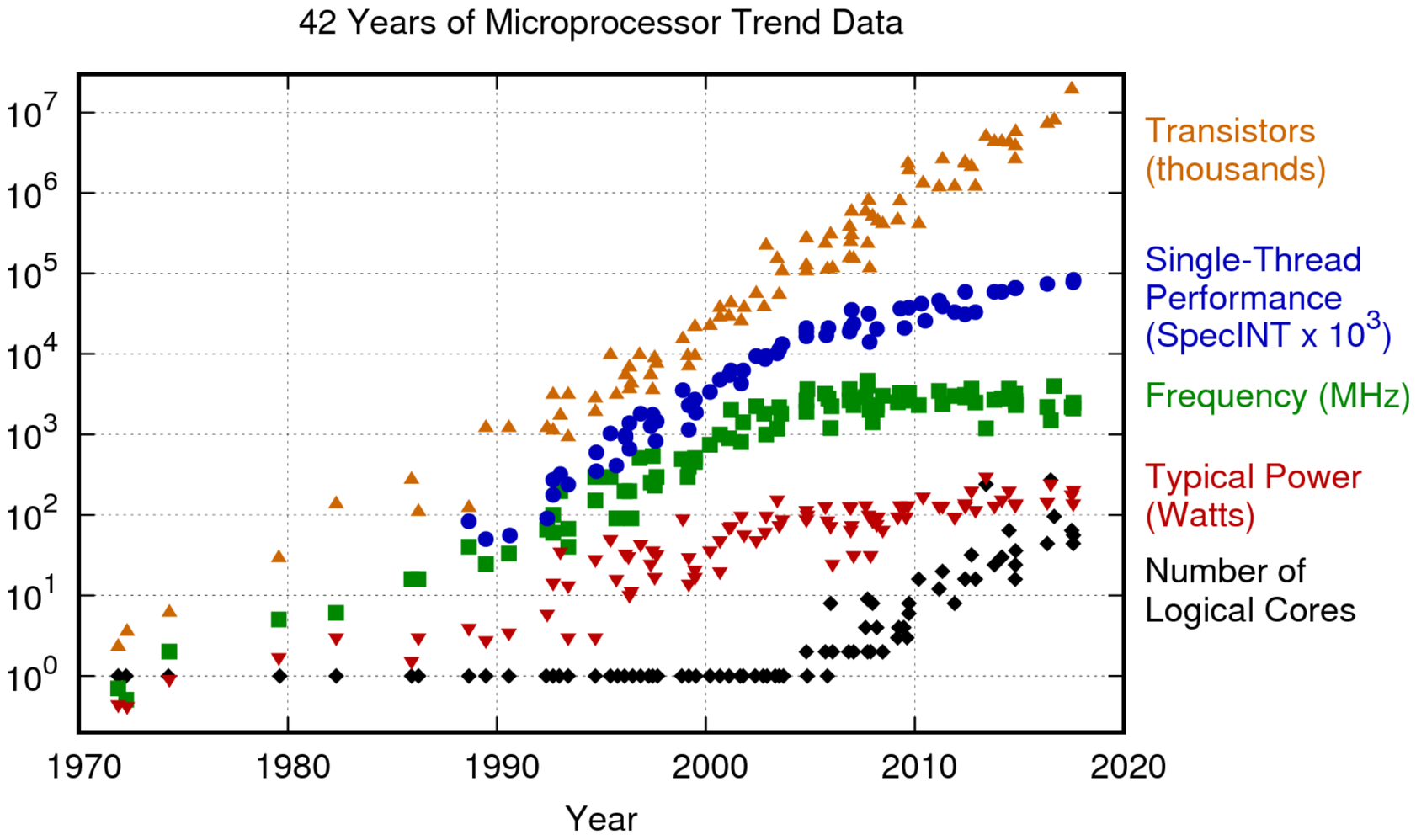
Recap



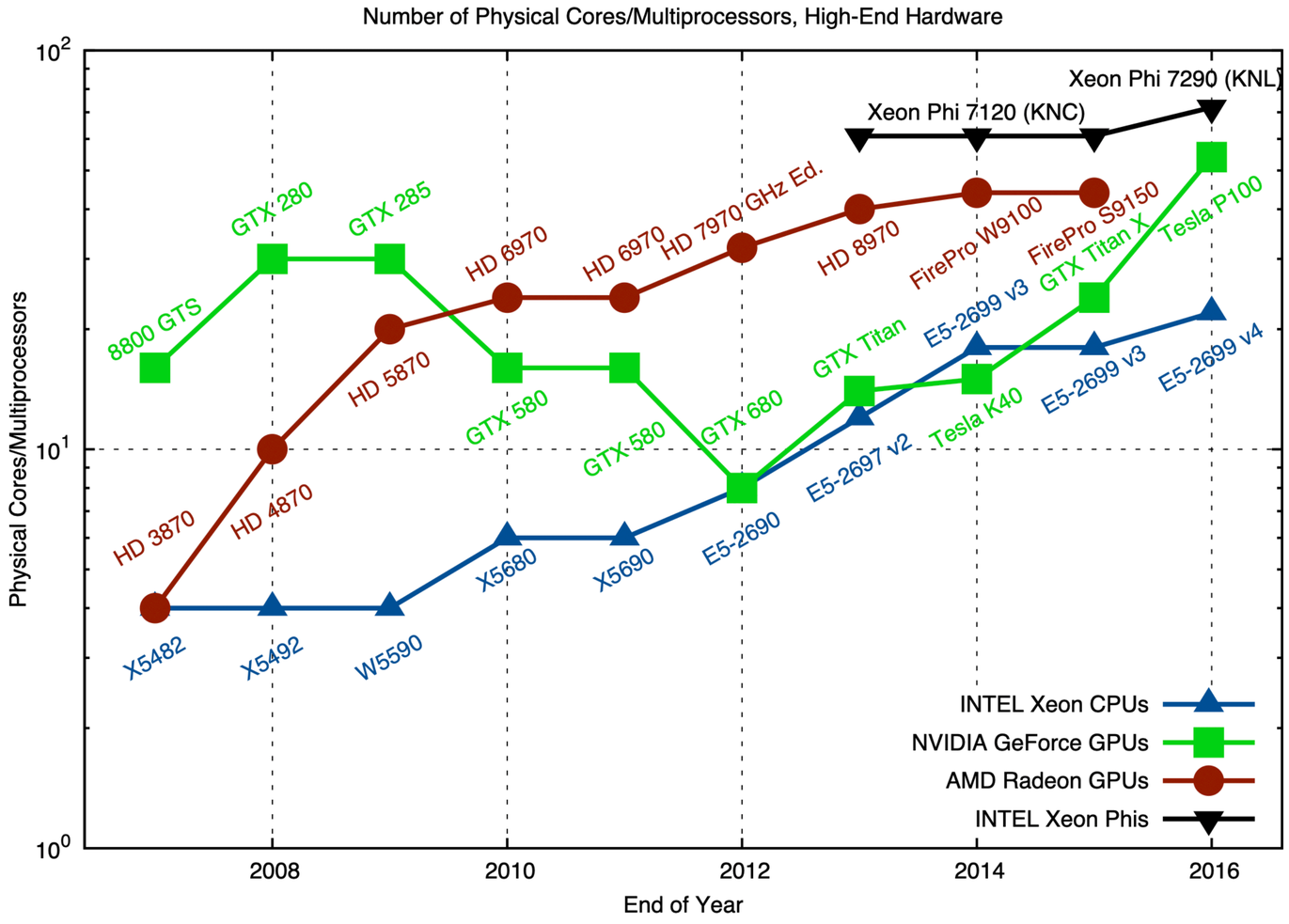
- OpenMP: a simpler way to program multicore processors
- Parallel constructs: `for` loop, `omp task`
- `taskwait, taskgroup, barrier`
- Data sharing clauses
- Synchronization constructs: `reduction, atomic, critical`

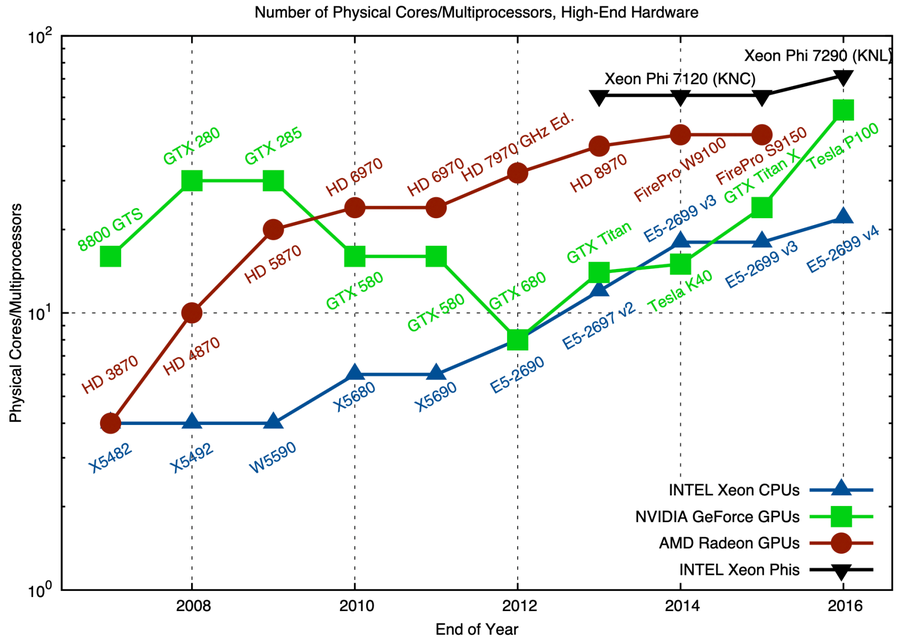
GPU processors
History and Performance

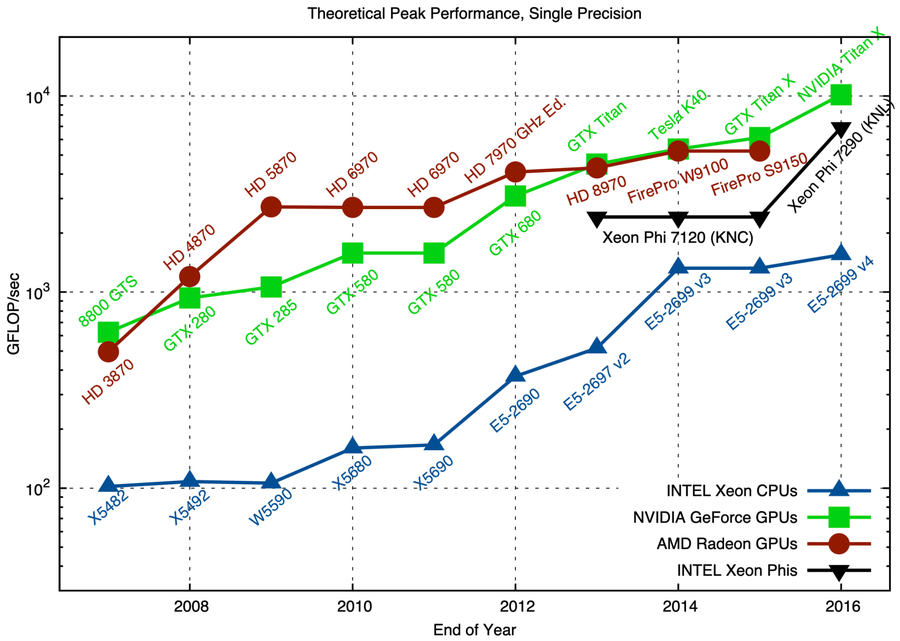
GPU computing

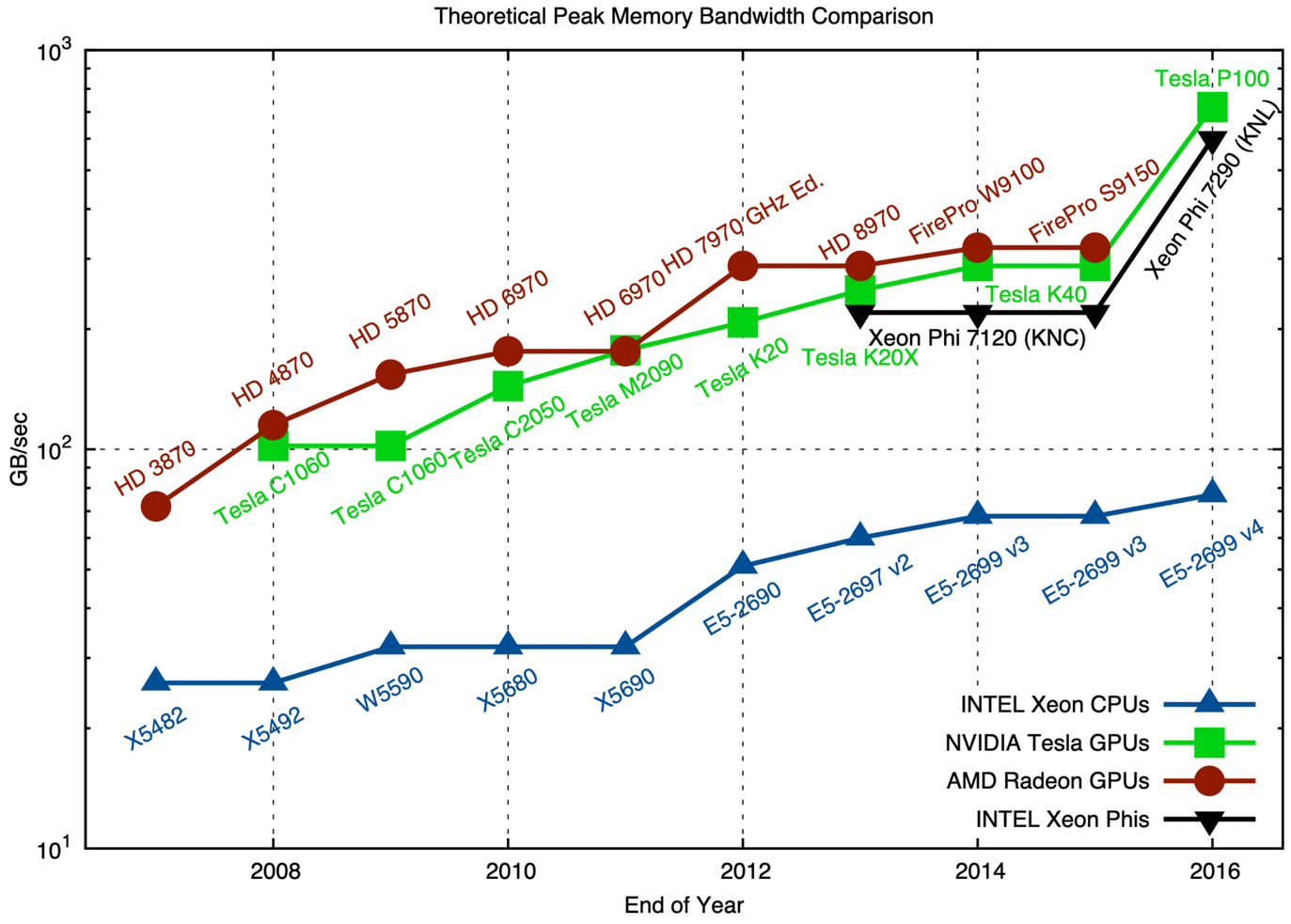


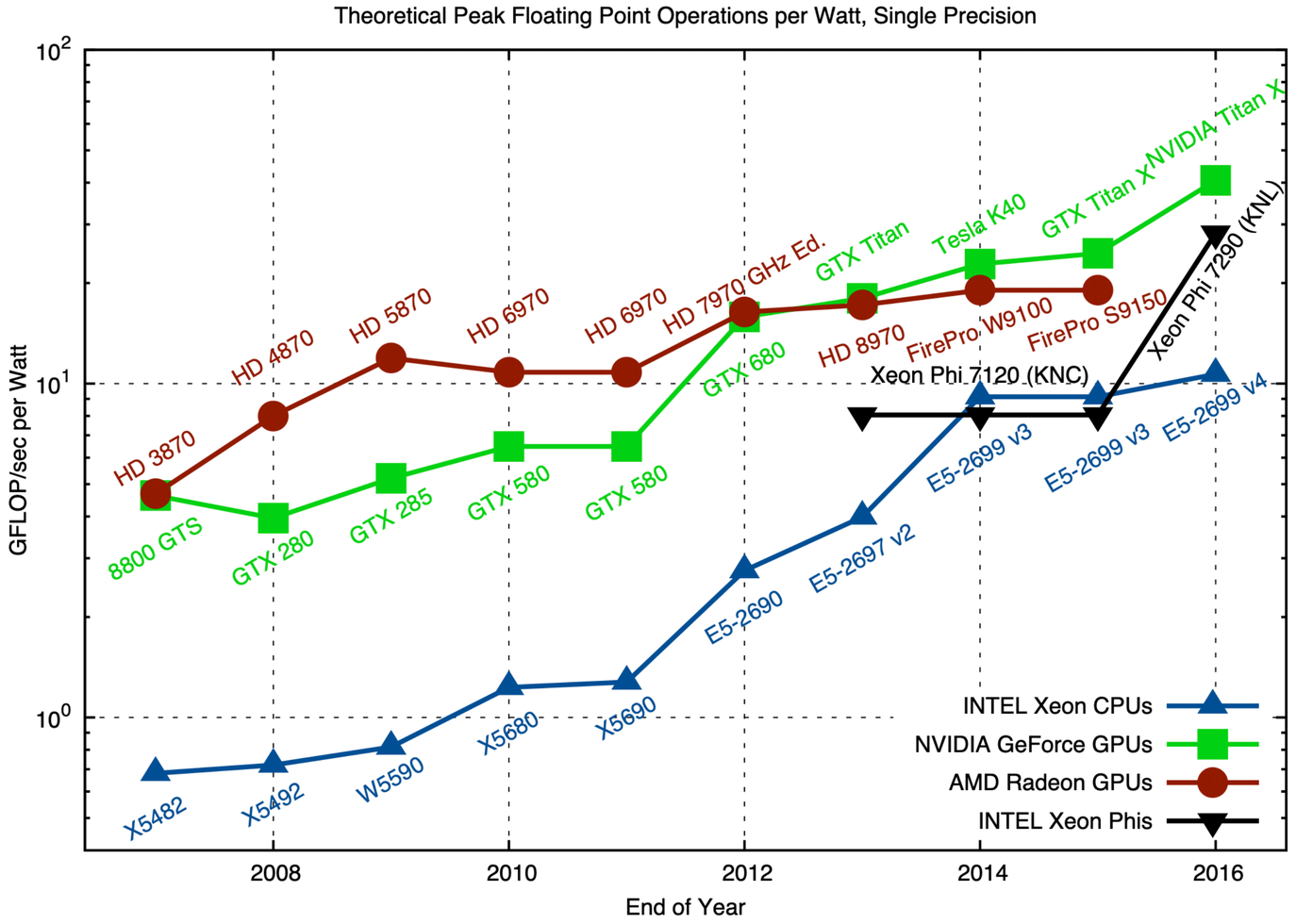
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

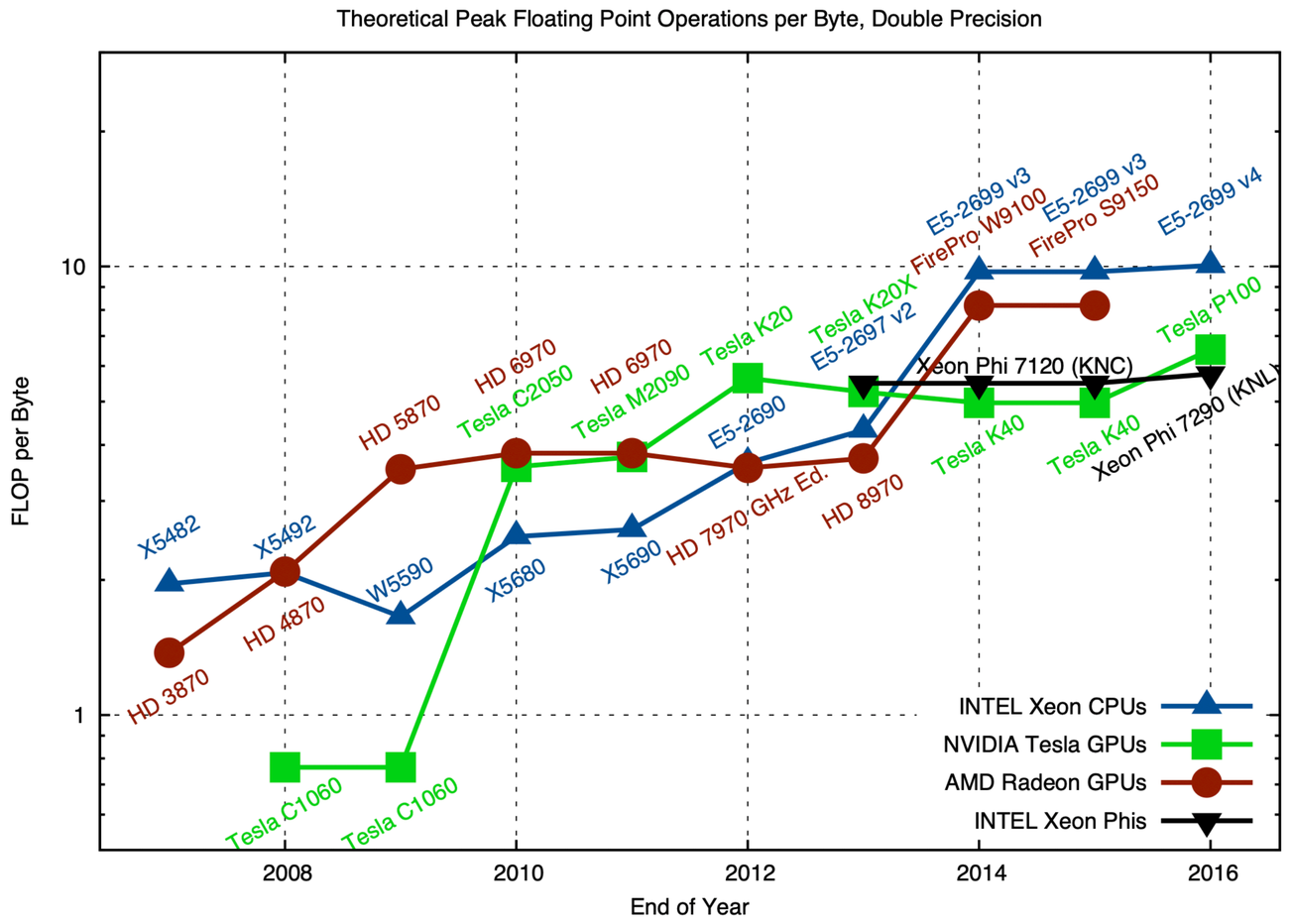

Number of Physical Cores/Multiprocessors, High-End Hardware
Physical Cores/Multiprocessors
End of Year
10^2
10^1
10^0
2008
2010
2012
2014
2016
Xeon Phi 7290 (KNL)
Xeon Phi 7120 (KNC)
GTX 280
GTX 285
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
GTX Titan X
Tesla P100
8800 GTS
GTX 580
GTX 580
GTX 680
GTX Titan
Tesla K40
HD 5870
HD 3870
HD 4870
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis


Number of Physical Cores/Multiprocessors, High-End Hardware
Physical Cores/Multiprocessors
End of Year
Xeon Phi 7290 (KNL)
Xeon Phi 7120 (KNC)
8800 GTS
GTX 280
GTX 285
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
GTX Titan X
Tesla P100
HD 3870
HD 4870
HD 5870
GTX 580
GTX 580
GTX 680
GTX Titan
E5-2699 v3
Tesla K40
E5-2699 v3
E5-2699 v4
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis

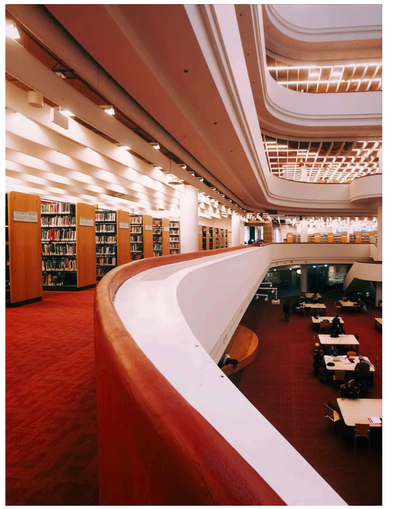

Theoretical Peak Performance, Single Precision
GFLOP/sec
End of Year
Xeon Phi 7290 (KNL)
Xeon Phi 7120 (KNC)
8800 GTS
GTX 280
GTX 285
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
GTX Titan
Tesla K40.
GTX Titan X
NVIDIA Titan X
HD 8970
FirePro W9100
FirePro S9150
HD 3870
HD 4870
GTX 580
GTX 580
GTX 680
E5-2699 v3
E5-2699 v3
E5-2699 v4
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis


Theoretical Peak Memory Bandwidth Comparison
GB/sec
10^3
10^2
10^1
2008
2010
2012
2014
2016
End of Year
HD 3870
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
Tesla P100
Xeon Phi 7290 (KNL)
Tesla K40
Xeon Phi 7120 (KNC)
Tesla C1060
Tesla C1060
Tesla C2050
Tesla M2090
Tesla K20
Tesla K20X
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA Tesla GPUs
AMD Radeon GPUs
INTEL Xeon Phis


Theoretical Peak Floating Point Operations per Watt, Single Precision
GFLOP/sec per Watt
10^2
10^1
10^0
2008
2010
2012
2014
2016
End of Year
HD 3870
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
8800 GTS
GTX 280
GTX 285
GTX 580
GTX 580
GTX 680
GTX Titan
Tesla K40
GTX Titan X
NVIDIA Titan X
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
Xeon Phi 7120 (KNC)
Xeon Phi 7290 (KNL)
INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis


Theoretical Peak Floating Point Operations per Byte, Double Precision
FLOP per Byte
10
1
2008
2010
2012
2014
2016
End of Year
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v4
HD 3870
HD 4870
HD 5870
HD 6970
HD 7970 GHz Ed.
HD 8970
Tesla C1060
Tesla C1060
Tesla C2050
Tesla M2090
Tesla K20
Tesla K20X
Tesla K40
Tesla K40
Tesla P100
FirePro W9100
FirePro S9150
Xeon Phi 7120 (KNC)
Xeon Phi 7290 (KNL)
INTEL Xeon CPUs
NVIDIA Tesla GPUs
AMD Radeon GPUs
INTEL Xeon Phis

References



https://github.com/karlrupp/cpu-gpu-mic-comparison

https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

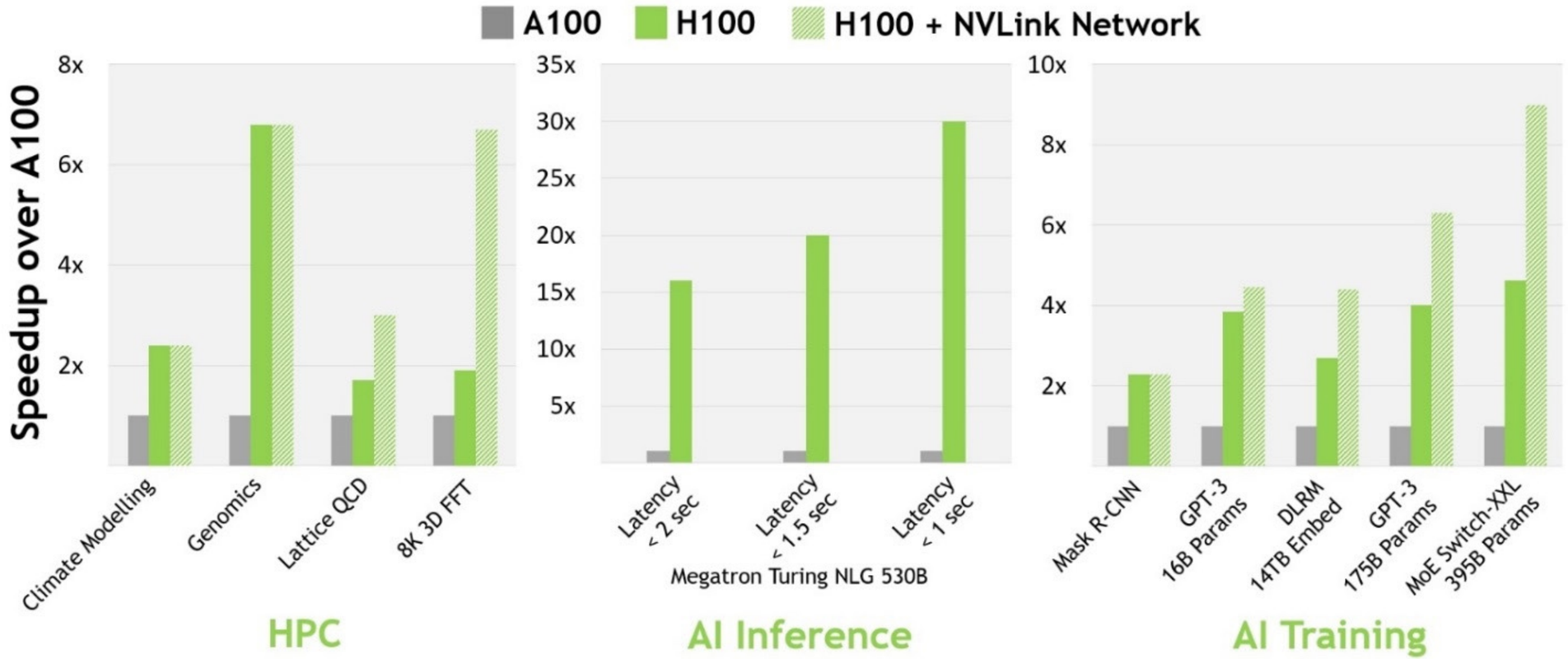
https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Example: Hopper H100

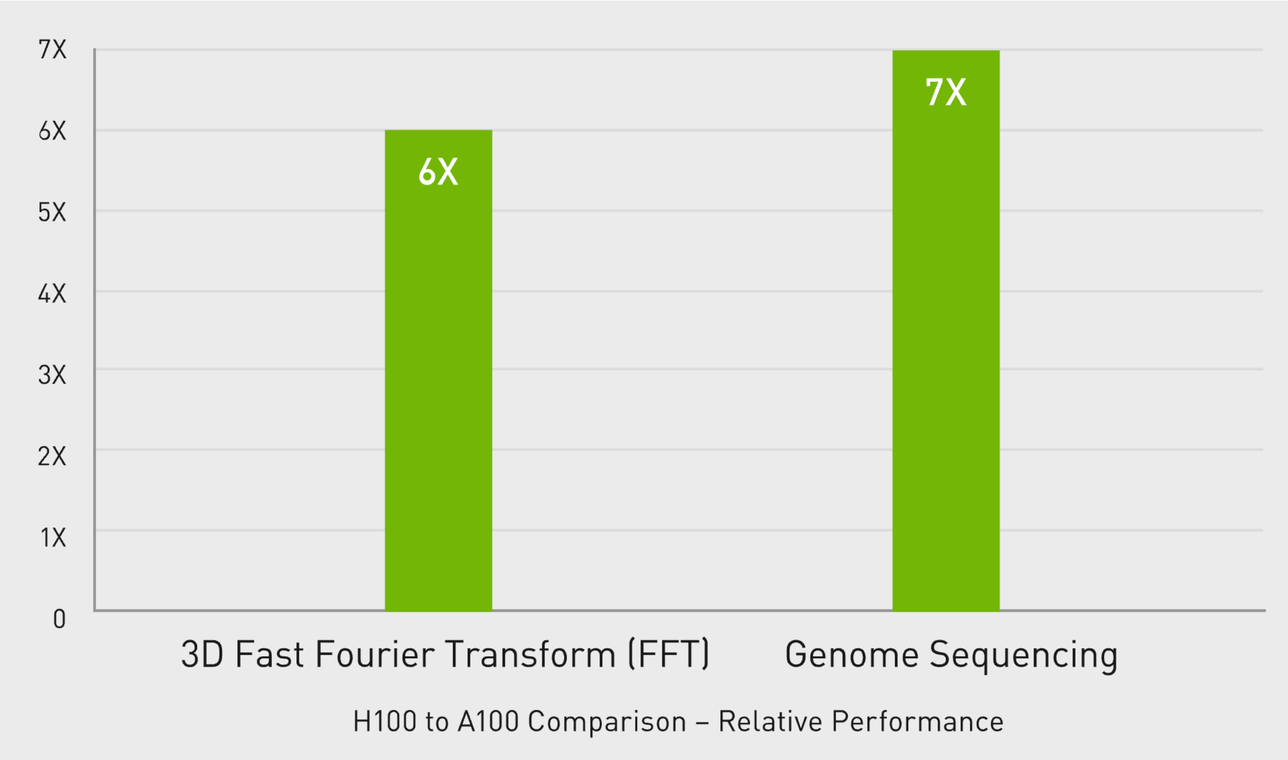
Form factor	H100 SXM	H100 PCIe
FP64	30 teraFLOPS	24 teraFLOPS
FP64 Tensor Core	60 teraFLOPS	48 teraFLOPS
FP32	60 teraFLOPS	48 teraFLOPS
TF32 Tensor Core	1,000 teraFLOPS* | 500 teraFLOPS	800 teraFLOPS* | 400 teraFLOPS
BFLOAT16 Tensor Core	2,000 teraFLOPS* | 1,000 teraFLOPS	1,600 teraFLOPS* | 800 teraFLOPS
FP16 Tensor Core	2,000 teraFLOPS* | 1,000 teraFLOPS	1,600 teraFLOPS* | 800 teraFLOPS
FP8 Tensor Core	4,000 teraFLOPS* | 2,000 teraFLOPS	3,200 teraFLOPS* | 1,600 teraFLOPS
INT8 Tensor Core	4,000 TOPS* | 2,000 TOPS	3,200 TOPS* | 1,600 TOPS
GPU memory	80GB	80GB
GPU memory bandwidth	3TB/s	2TB/s
Max thermal design power (TDP)	700W	350W
Interconnect	NVLink: 900GB/s PCIe Gen5: 128GB/s	NVLINK: 600GB/s PCIe Gen5: 128GB/s

- * With sparsity
- SXM is a high bandwidth socket solution for connecting Nvidia Compute Accelerators to a system
- `bfloat16` (Brain Floating Point) floating-point format is a computer number format occupying 16 bits in computer memory

H100 AI and HPC benchmarks



H100 to A100 comparison—Relative performance



What is the technology behind GPU processors?

History

- GPU were initially focused on 3D graphics = computing the color of each pixel on the screen based on a 3D model of a scene.

- Featured example: ray tracing global illumination (RTXGI)

GPGPU

General purpose GPU computing:

- Extension to general scientific computations.
- Solving equations on a grid is similar to rendering: perform the same regular calculations on a very large dataset.

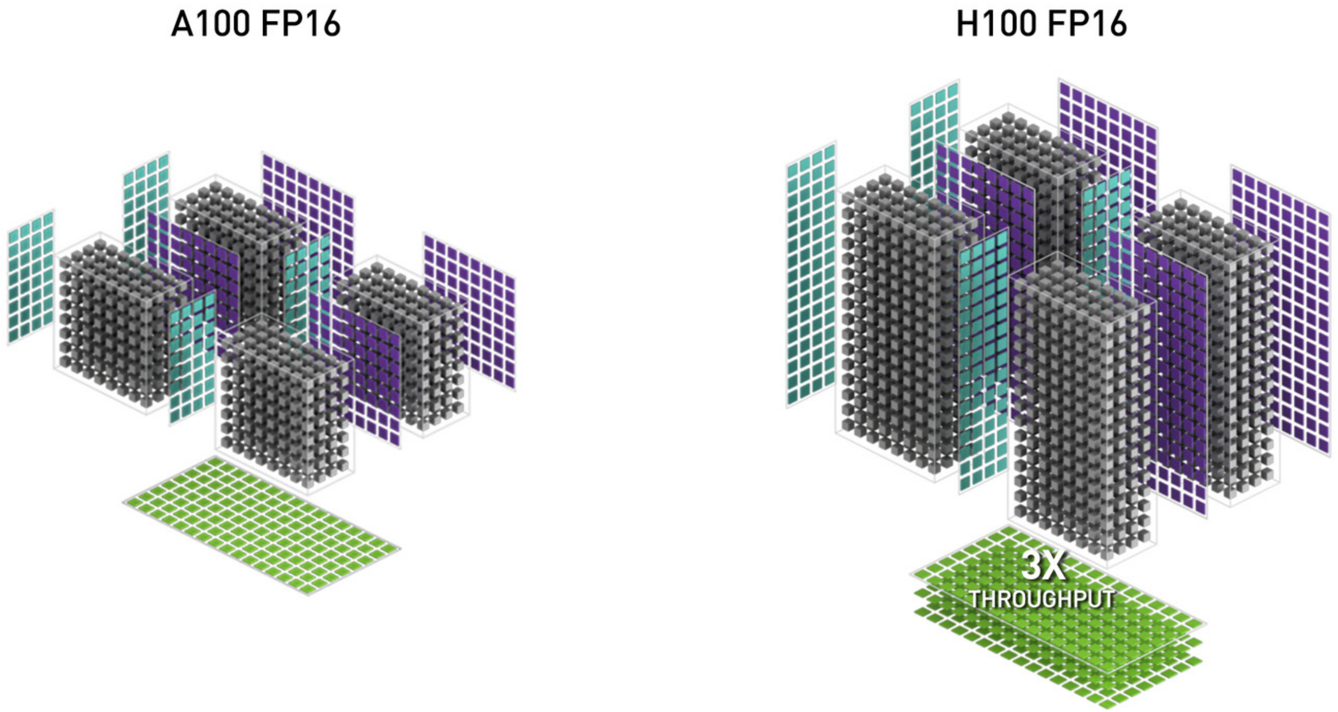
Deep learning

Recent advances in GPU computing target deep learning.

1. Linear algebra: matrix-matrix multiplications.

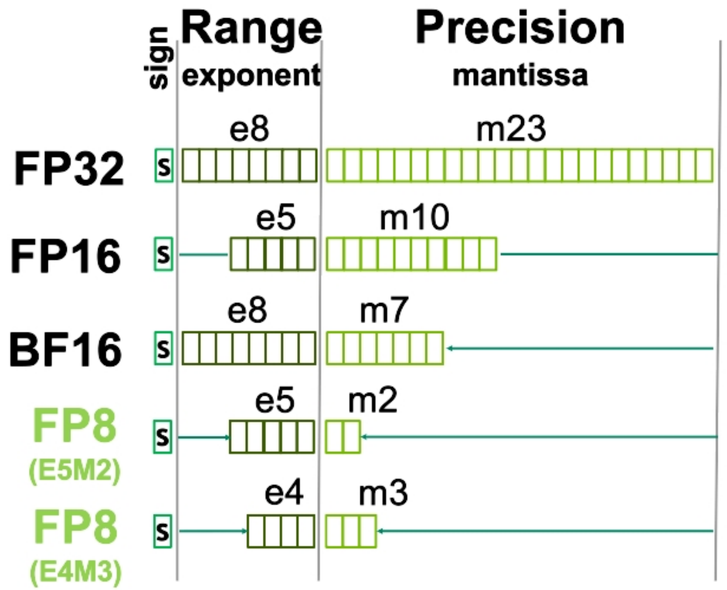
2. Varying precision.

Tensor Cores

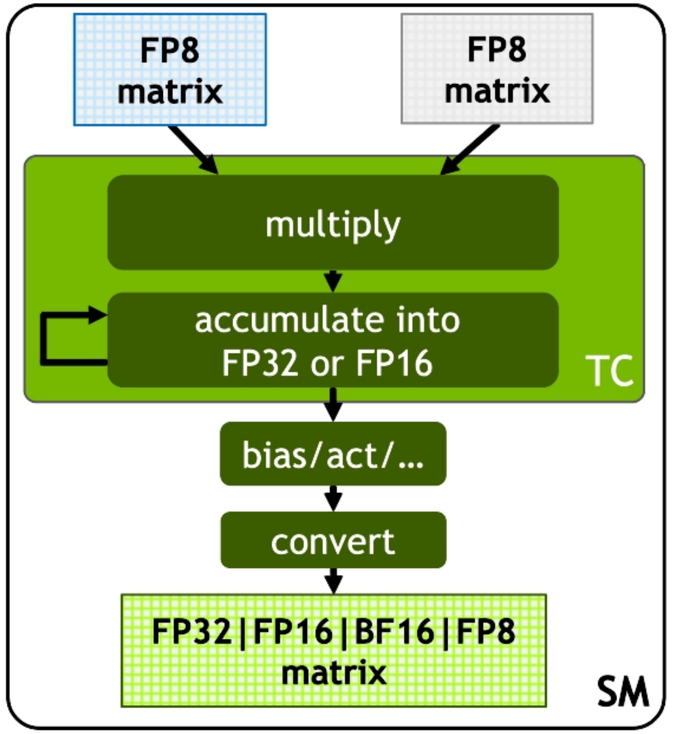


H100 FP16 Tensor Core has 3x throughput compared to A100 FP16 Tensor Core

Performance through varying arithmetic precision



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types

New Hopper FP8 Precisions

GPUs are great for

- Dense linear algebra with massive amount of flops
- Finite-difference and regular grid calculations
- Deep neural networks.

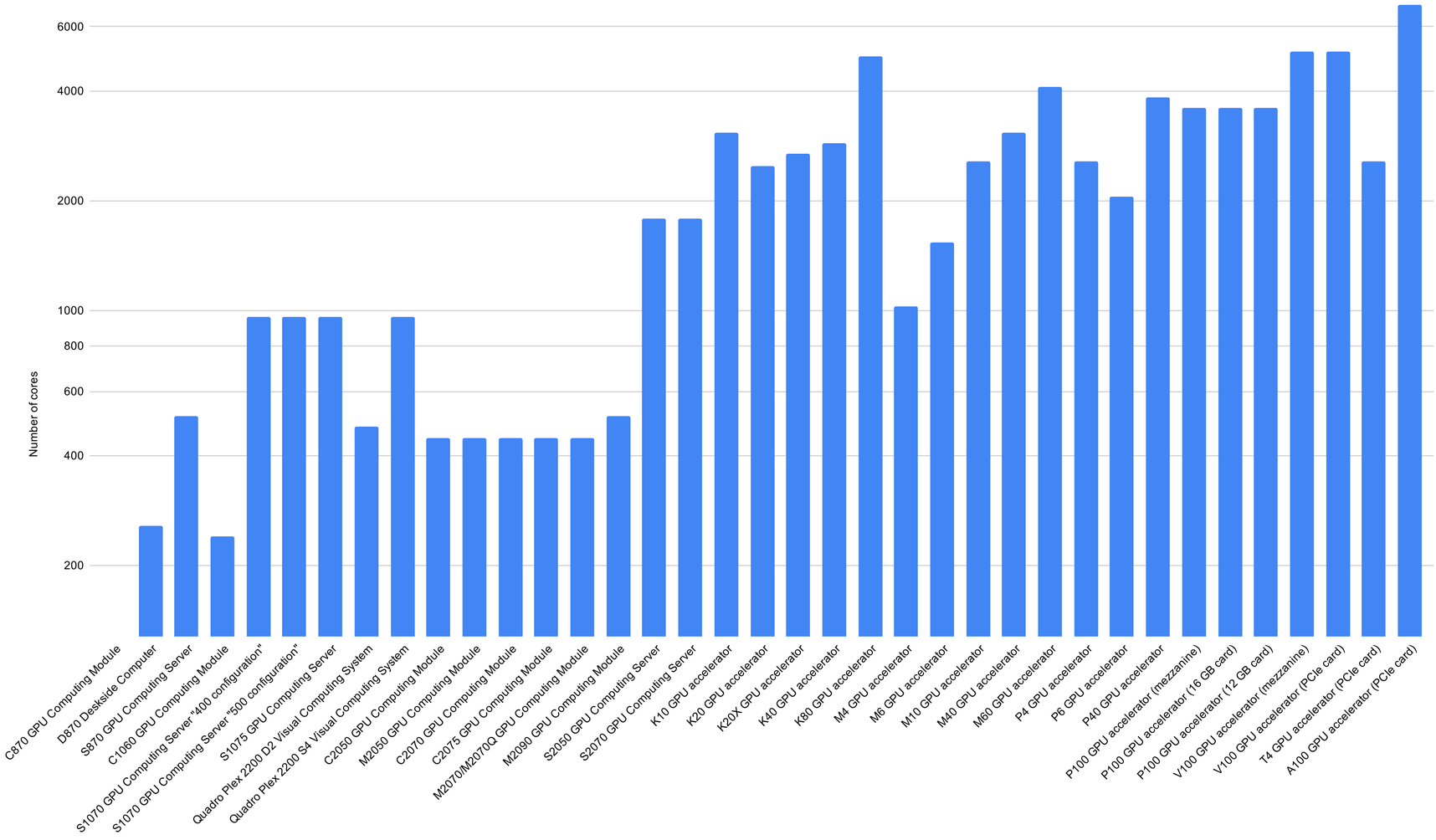
Less suitable for:

- Irregular calculations with branching and irregular workloads
- Long series of sequential operations

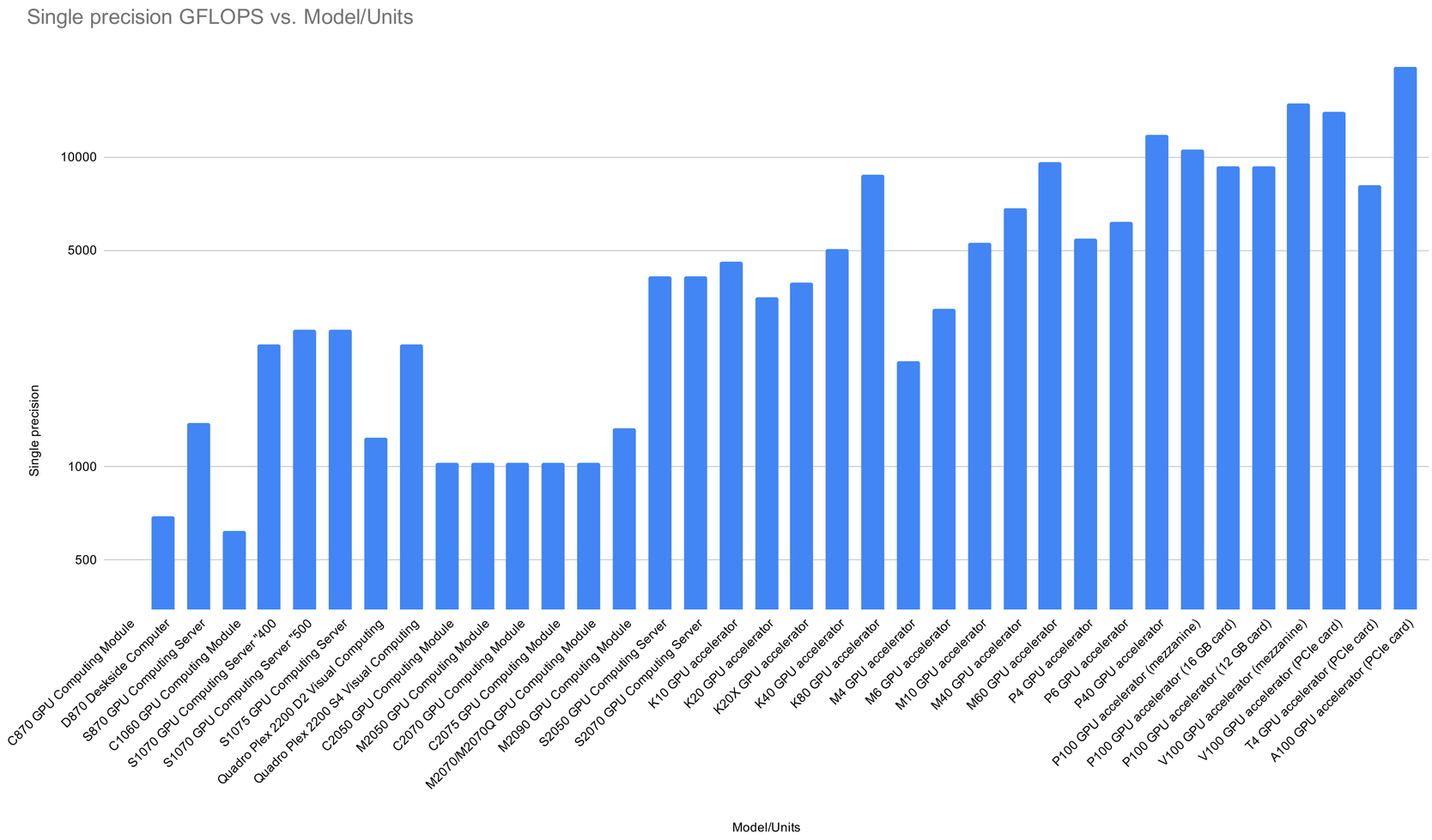
The secret behind performance

- “Simpler” cores
- More cores for computing, but less space for optimization
- Compared to multicore processors:
  - ~~Out of order execution~~
  - ~~Branch prediction~~
  - ~~Pre-fetching~~
  - ~~Large amount of multilevel cache~~

Tesla GPUs: number of cores

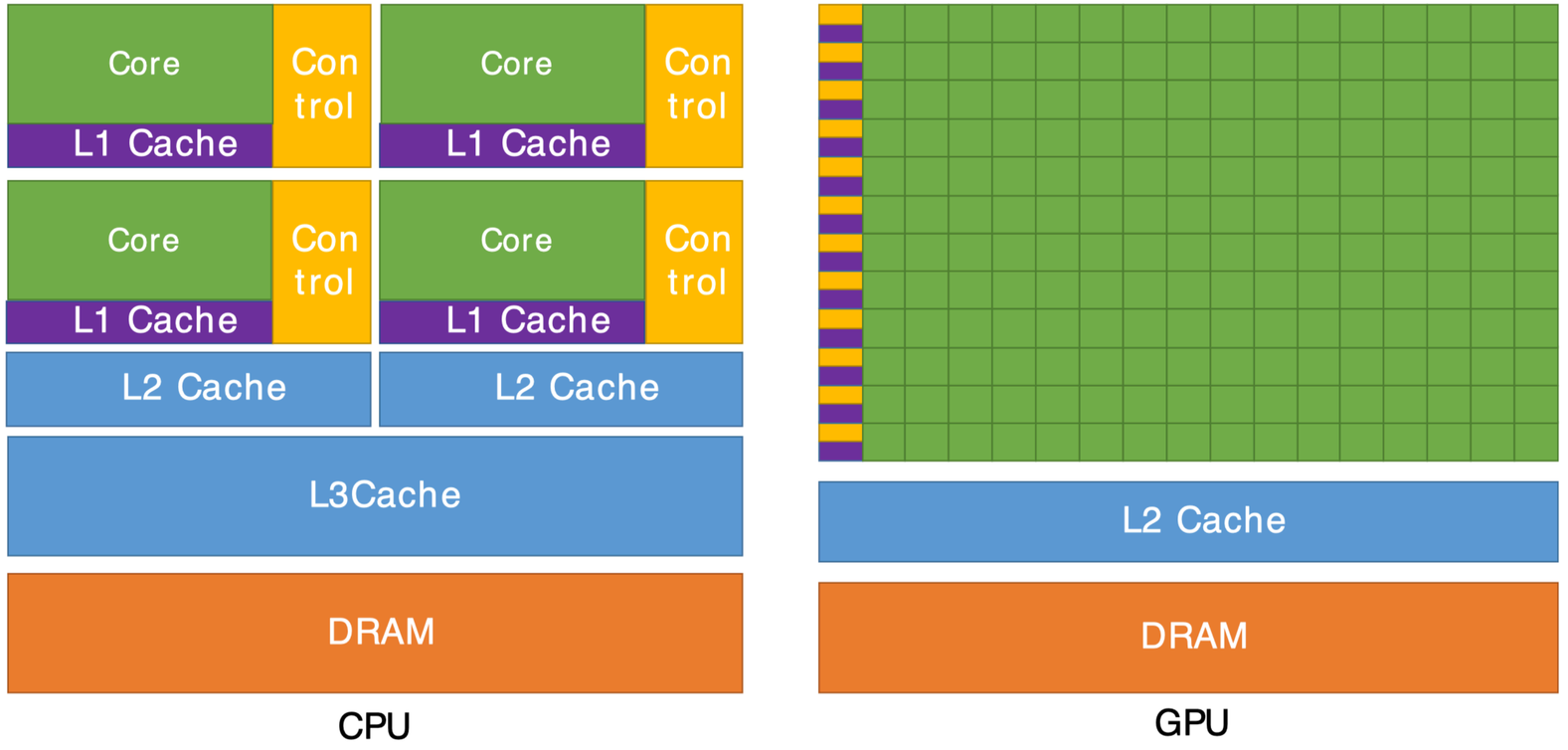


Single precision performance



What does a GPU processor look like?

Schematic organization



The GPU devotes more transistors to data processing

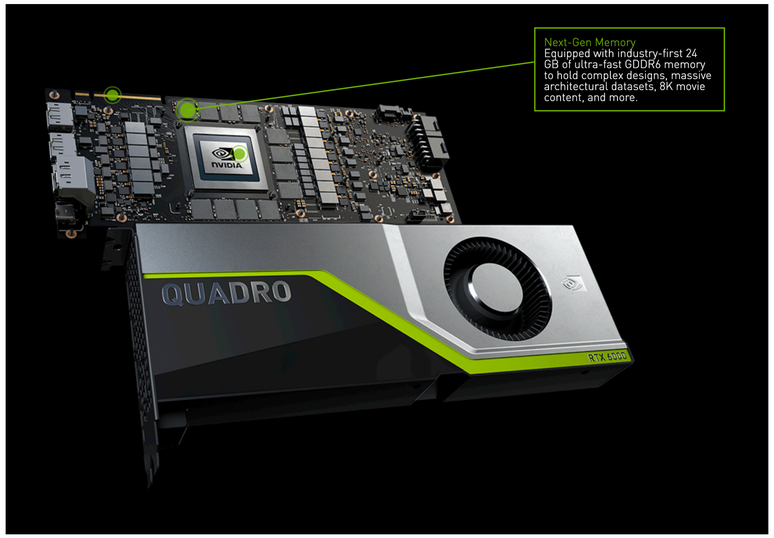
GH100 with 144 Streaming Multiprocessors (SM)

NVLink allows GPU processors to communicate without using the CPU

GH100 Streaming Multiprocessor (SM)

- Special Functions Units (SFUs): execute transcendental instructions such as sin, cosine, reciprocal, and square root.
- Dispatch Unit: instruction dispatch

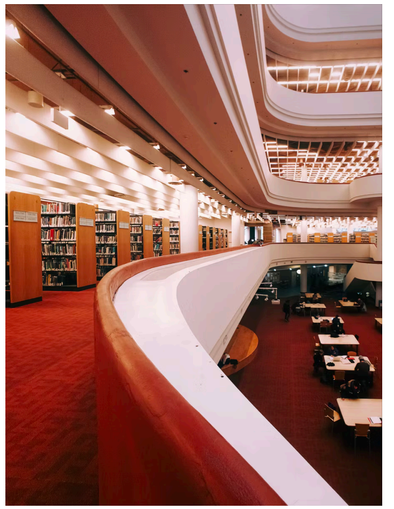
Quadro RTX 6000 in icme-gpu



4,608 parallel processing cores; Turing architecture

- 72 Streaming multiprocessors (6 graphics processing clusters)
- 4,608 cores (64 per SM)
- Peak performance: single 16.3 TFLOPS; double 510 GFLOPS (1/32)
- Total global memory: 24 GB GDDR6
- Bandwidth: 672 GB/sec
- L1 Cache: 96 KB (per SMX); L2 Cache: 6,291,456 B
- Power: 295 W
- Architecture: Turing
- Compute capability: 7.5

References



- Quadro RTX 600
- Turing architecture
- Whitepaper
- Specs

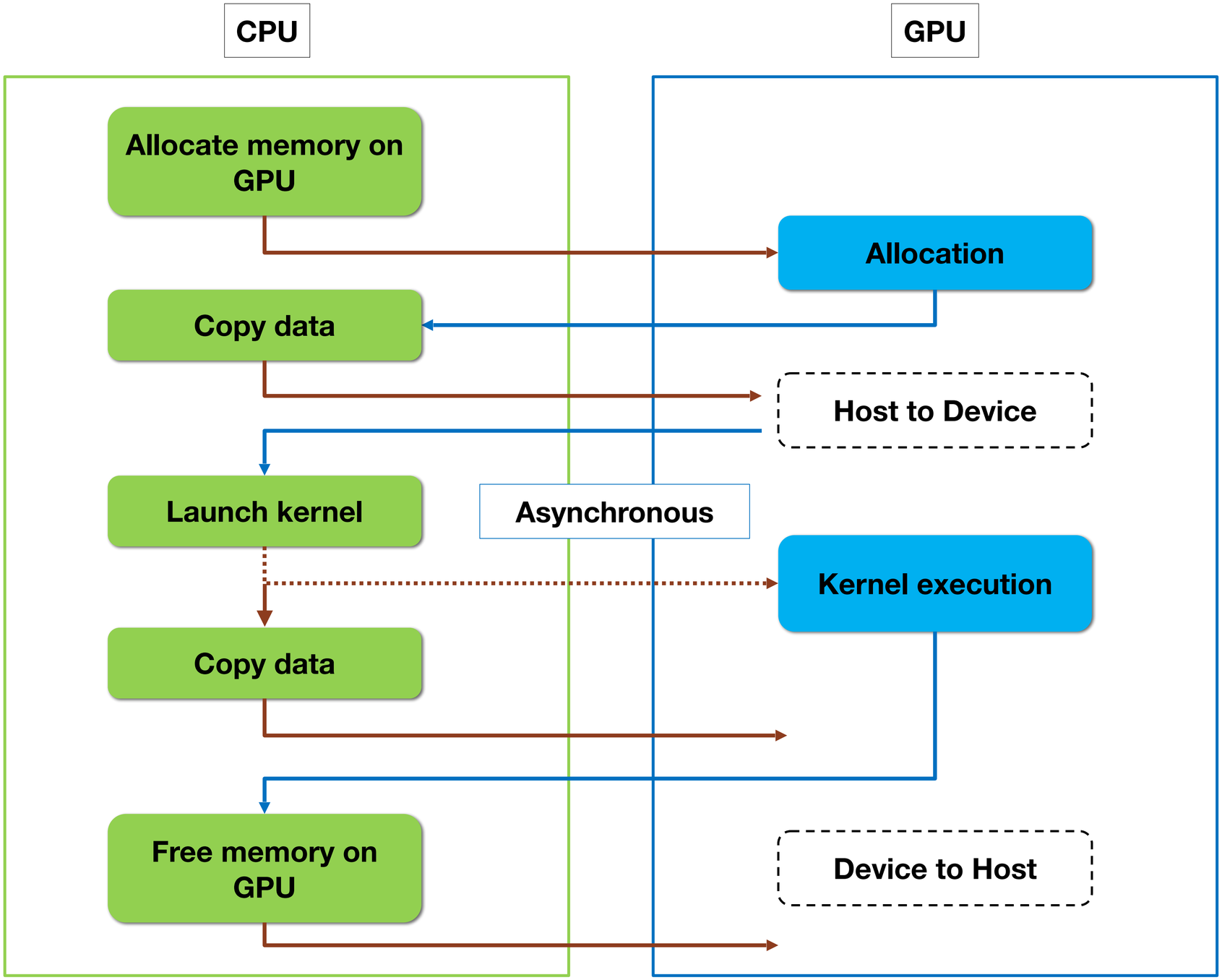
How to program GPUs
Introduction

GPU processors are co-processors

- 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
Free memory on GPU
Device to Host

GPU programming is different from CPU programming

- We will look at this in more details in future lectures but the “logic” behind GPU programming is very different from CPU programming.
- **Light threads:** hardware supports the ability to switch threads every cycle
  - Threads stay live, so switching threads has **no cost**
  - This is different from CPUs where processes and threads constantly switch in and out
  - Threads are organized **hierarchically** for efficient thread management
- **Limited logic capabilities for program control:**
  - 32 threads are grouped into warps
  - Threads in warp execute the same instruction at the same time

Organization of CUDA threads

CUDA warp

- 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.

CUDA thread block

- Groups of warps form a block.
- A block executes on one streaming multiprocessor (SM).
- Threads within a block have some ability to exchange data and synchronize.
- Multiple blocks may reside on the same SM.

GH100 SMs

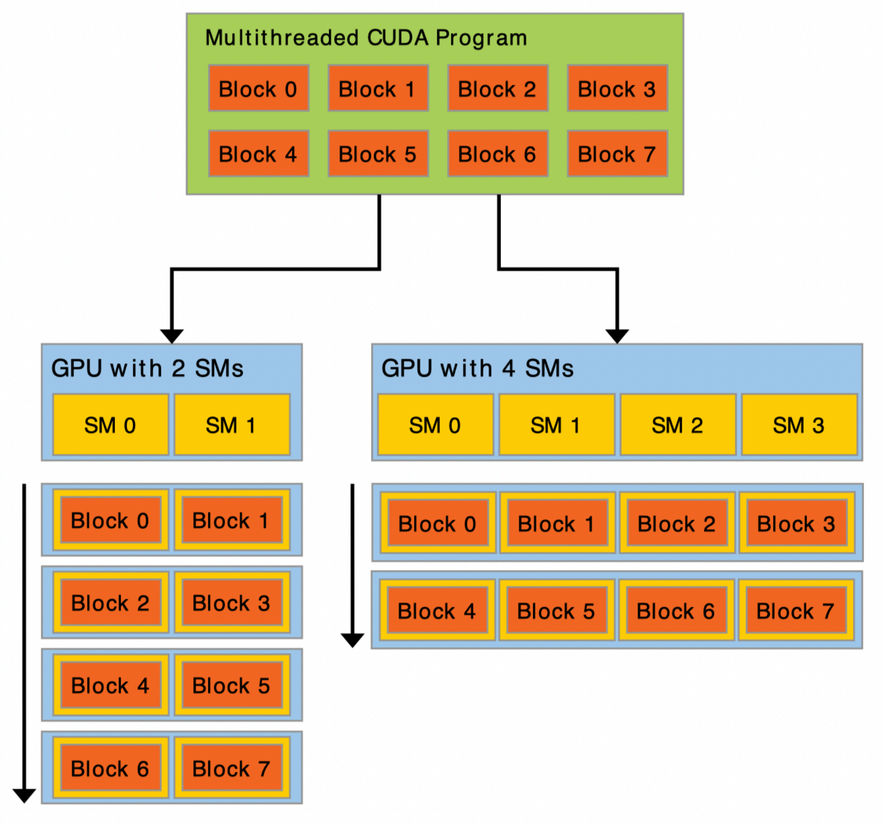

PCI Express 5.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
L2 Cache
HBM3
Memory Controller
High-Speed Hub
NVLink

Grid

A collection of many blocks constitute the entire “dataset” that will be operated on by a kernel.

A grid of thread blocks

Execution model of blocks



Hardware execution model

- When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity.

- The threads of a thread block execute concurrently on **one multiprocessor,** and multiple thread blocks can execute concurrently on one multiprocessor.

- As thread blocks terminate, **new blocks are launched** on the vacated multiprocessors.

Hardware is reflected in CUDA
Example of CUDA kernel

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}
```

- Each thread is identified by an ID.
- Work is assigned to thread based on thread ID.

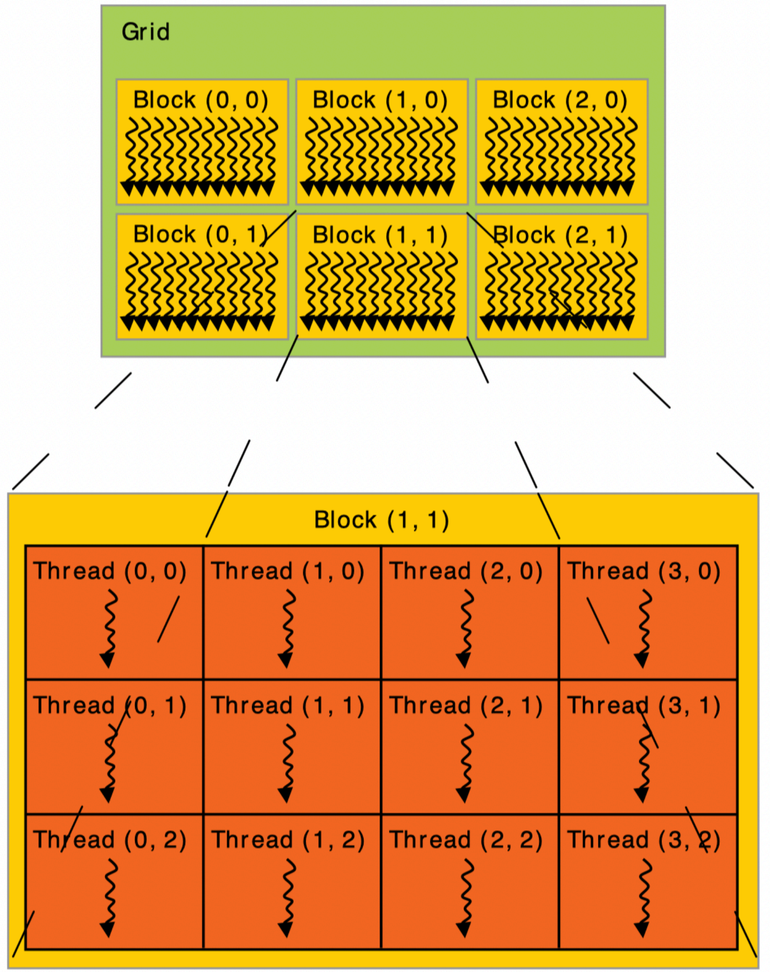
Calling a CUDA kernel

```
int main() {
  // ...
  // Kernel invocation with one block of N * N * 1 threads
  int numBlocks = 1;
  dim3 threadsPerBlock(N, N);
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
  // ...
}
```

- `numBlocks` : number of thread blocks in a grid.
- `threadsPerBlock` : number of threads in a block.

Grid of blocks and threads in a block

- Grid of thread blocks
- Team of threads in a block
- Indexing uses a `dim3` : 1D, 2D, or 3D indexing is possible as needed for the application.



Scalability of model

- Thread blocks are required to execute independently.
- It must be possible to execute them in any order, in parallel or in series.
- This independence requirement allows thread blocks to be scheduled in any order across any number of cores.
- This enables programmers to write code that scales with the number of cores.

SIMT architecture

SIMT architecture details

- A streaming multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called **warps**.
- Individual threads composing a warp start together at the same program address, but they have **their own instruction address counter** and register state and are therefore free to branch and execute independently.
- But, best performance is achieved when threads in a warp execute the same instruction.
- Unlike CPU cores, instructions are issued in order and there is no branch prediction or speculative execution.

SIMT vs SIMD

- SIMT: CUDA model; single-instruction multiple threads
- SIMD: vector processors; single-instruction multiple data.
- The SIMT architecture is akin to SIMD vector organizations in that a **single instruction controls multiple processing elements.**
- A key difference is that SIMD vector organizations expose the vector width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread.
- In contrast with SIMD vector machines, **SIMT enables programmers to write data-parallel code for coordinated threads (best for performance), as well as thread-level parallel code for independent, scalar threads.**
- Each of the SIMT cores has a different stack pointer (and thus performs computation on different data sets), whereas SIMD lanes are part of an ALU with no knowledge of memory per se.