

# Code of the Day – Power of Compiler Optimization

```
#include <array>

int main() {
    std::array<int, 5> numbers = {1, 2, 3, 4, 5};
    int sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        sum += numbers[i];
    }
    return sum;
}
```

g++ -O0

```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 48
    mov     dword ptr [rbp - 12], 0
    mov     rax, qword ptr [rip + .L__const.main.numbers]
    mov     qword ptr [rbp - 32], rax
    mov     rax, qword ptr [rip + .L__const.main.numbers+8]
    mov     qword ptr [rbp - 24], rax
    mov     eax, dword ptr [rip + .L__const.main.numbers+16]
    mov     dword ptr [rbp - 16], eax
    mov     dword ptr [rbp - 36], 0
    mov     dword ptr [rbp - 40], 0
.LBB0_1:
    movsxd  rax, dword ptr [rbp - 40]
    lea     rcx, [rbp - 32]
    mov     qword ptr [rbp - 8], rcx
    cmp     rax, 5
    jae     .LBB0_4
    movsxd  rsi, dword ptr [rbp - 40]
    lea     rdi, [rbp - 32]
    call    std::array<int, 5ul>::operator[](unsigned long)
    mov     eax, dword ptr [rax]
    add     eax, dword ptr [rbp - 36]
    mov     dword ptr [rbp - 36], eax
```

>40 lines of assembly code

```
#include <array>

int main() {
    std::array<int, 5> numbers = {1, 2, 3, 4, 5};
    int sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        sum += numbers[i];
    }
    return sum;
}
```

g++ -O3

```
main:
    mov     eax, 15
    ret
```

2 lines of assembly code

# ECE 455

High Performance Computing for Applications in Engineering

[Fall 2025]

Introduction to GPU and CUDA

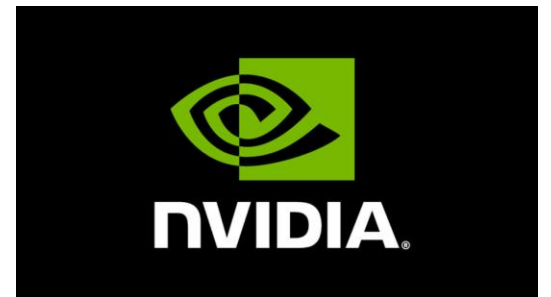
09/29/2025

# Before we get started...

- Quick overview, things discussed last time
  - Parallel programming using OpenMP
- In today's lecture:
  - GPU architecture
  - GPU programming using Nvidia CUDA
- Miscellaneous
  - Assignment #3 – due on **10/03 at 23:59 PM**

# Guest Lecture by Matt@Nvidia in 10/3 Friday's Seminar (Virtual)

- **Title:** AI for Decision-Making in Modern Chip Design
- **Abstract:** Semiconductor design is colliding with unprecedented complexity—chipllets, tight PPA targets, and compressed schedules—making rigorous, data-driven decision-making essential. My presentation shows how AI augments engineering judgment across the silicon lifecycle: from architectural exploration and RTL/verification to physical design, sign-off, and manufacturing. I'll frame AI as a set of roles—predictor, optimizer, and assistant—that narrow search spaces, surface risks earlier, and accelerate trade-off analysis with a couple of examples. I'll also share insight into the general impact of AI in the Chip Design industry.
- **Bio:** Matt is currently the AI for Chip Design Inference lead at NVIDIA, responsible for managing and deploying new AI models into the company for GPU Chip Design Engineers to utilize for improving our chips and the design flows. Matt started his career at Hewlett-Packard working on the physical design for the early Intel Itanium 64-bit data center processors, moved to Intel and continued work on XEON processor design. He transitioned from physical hardware design to software and Machine Learning and developed ML solutions for design automation at Intel before moving to NVIDIA where he works today. Matt earned a BSECE from the UW in 2000.



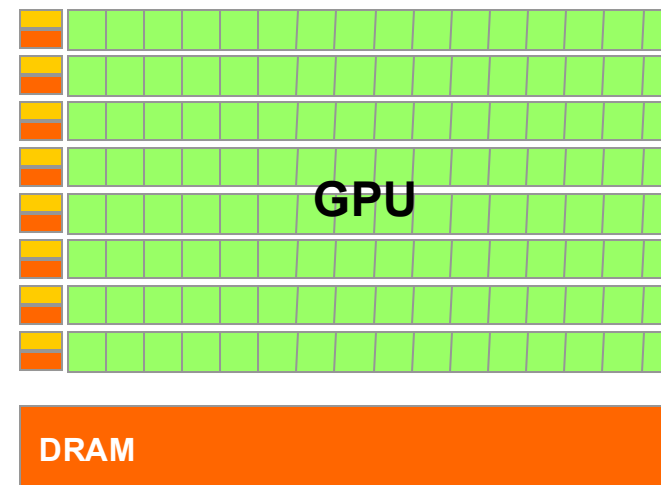
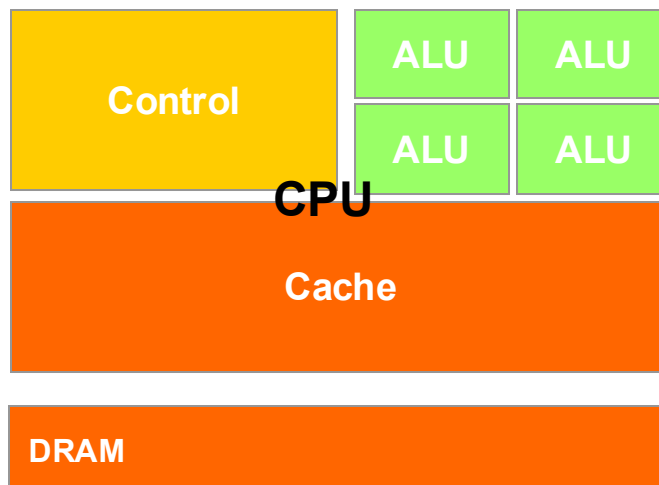
# GPU Computing with CUDA

# Overview of GPU Programming to be Covered in the Course

- Goal: Understand how GPUs can be used in Scientific Computing
- Topics
  - Hardware considerations
  - “Hello World” with GPU computing
  - Execution configuration issues
  - The memory model in CUDA
  - Matrix multiplication algorithm using GPU
  - Reduction algorithm using GPU
  - Scheduling optimization

# The CPU and GPU: showing their true colors

- The CPU is specialized for general-purpose computation
  - Ex: Decision making, general data structure, recursion, caching, control flow, etc.
- The GPU is specialized for compute-intensive, highly data-parallel computation
  - Ex: Graphics rendering, matrix operations, linear algebra, etc.
  - More transistors devoted to data processing rather than data caching and control flow



*“If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?” – Seymour Cray*

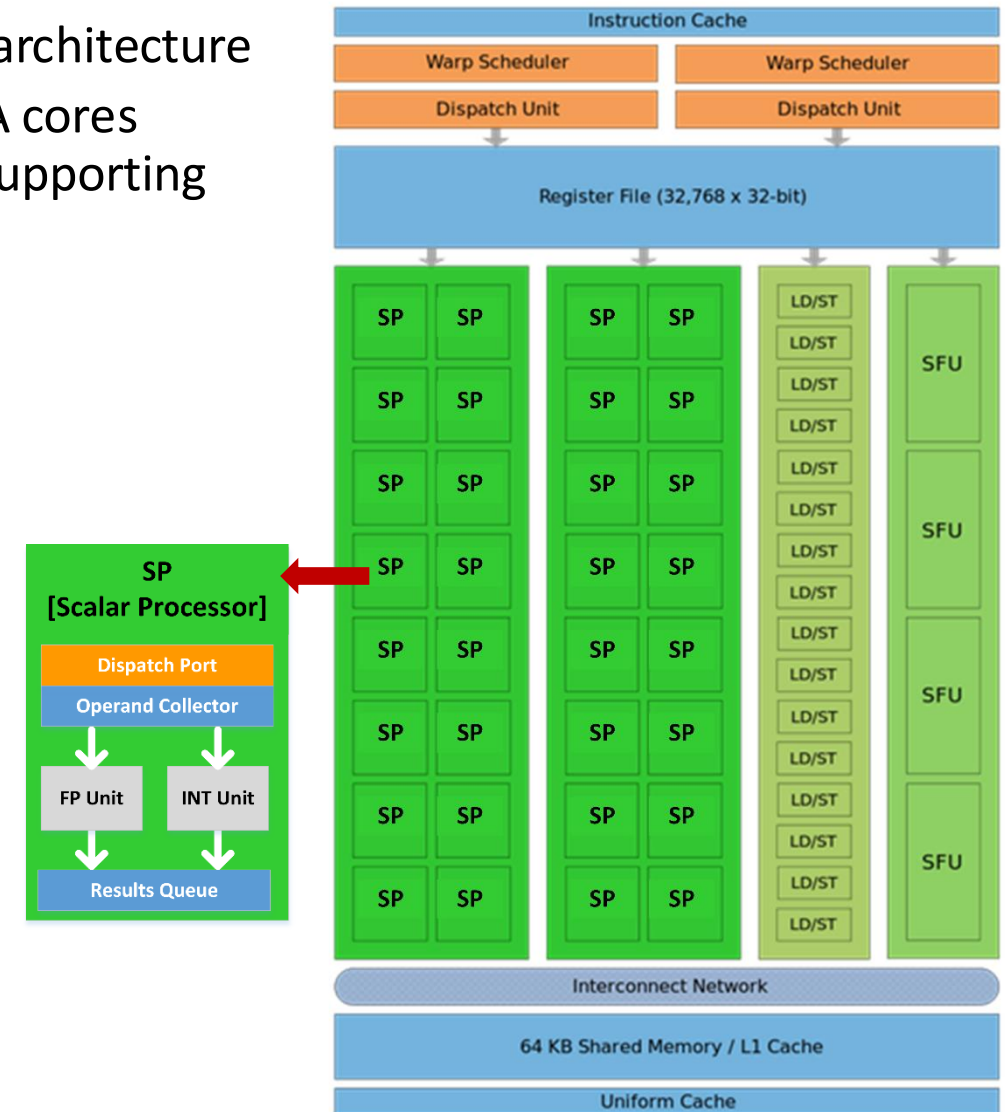
# Euler Node, Nvidia GeForce GTX GPU



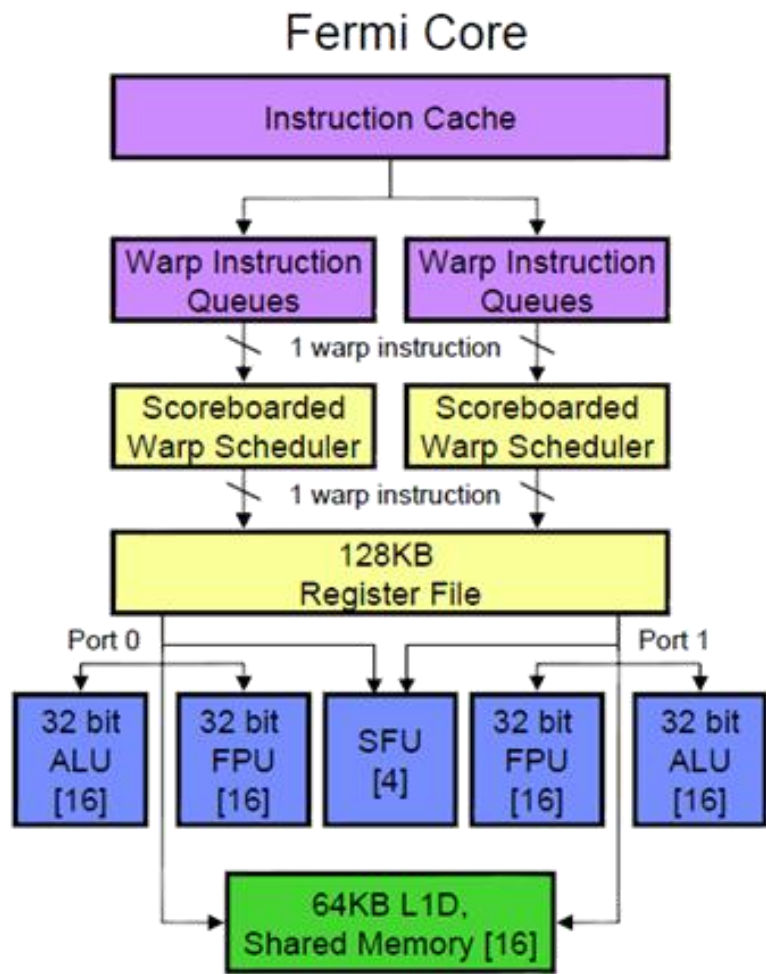


# Streaming Multiprocessor (SM) [Nvidia Fermi Architecture]

- SM is a fundamental unit of processing power in NVIDIA's GPU architecture
  - It's a collection of smaller processing elements called CUDA cores (scalar processors), along with shared memory and other supporting hardware
- Ex: Nvidia Fermi Architecture (block diagram on the right)
  - Released in late 2009 to early 2010
  - 40 nm technology
  - Three billion transistors
  - 512 Scalar Processors (SPs), or “shaders” (chickens)
  - L1 and L1 caches (no L3 cache)
  - 6 GB of global memory
    - DRAM (like the main memory for CPU)
  - Operates at low clock rate: 1.5 GHz
  - High memory bandwidth (for 2010,  $\approx 200$  GB/s)

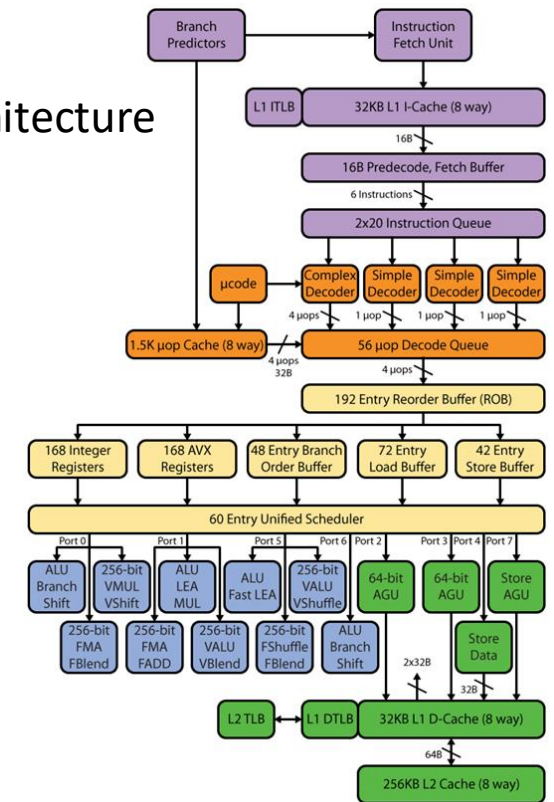


# Architectural Diagram of One Fermi SM (CUDA Core)



Fermi Stream Multiprocessor

Haswell CPU architecture

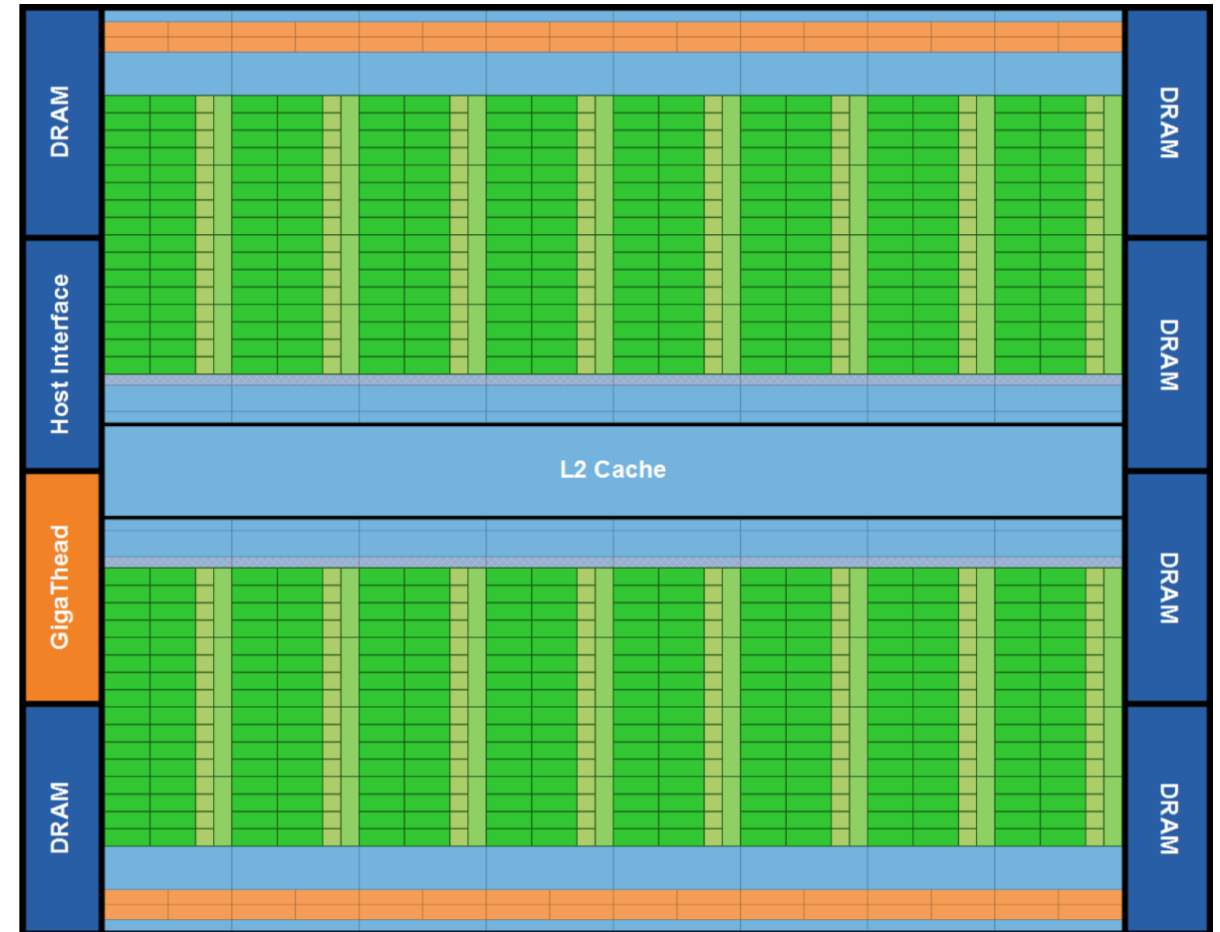


Not a whole lot of brain here, compared to CPU. Specifically, no orange boxes, which have to do with ILP.

[NOTE: color code is the same for SM & CPU]

# Overview of the Entire Fermi Chip

- Lots of ALU (green), not much of CU (orange)
- Explains why GPUs are fast for high arithmetic intensity applications
  - Linear algebra, machine learning, scientific computing
- Arithmetic intensity: the number of operations performed per byte of data



# Beefy GPU vs. beefy CPU comparison, 2019

	GPU – NVIDIA Tesla V100 32GB \$8,995	CPU – Intel® Xeon® Platinum 8180 \$10,009
Processing Cores	5120 CUDA Cores 640 Tensor Cores	28 cores (56 threads, SMT)
Memory	32GB HBM2	- 64 KB L1 (I+D) cache / core - 1 MB L2 cache / core - 38.5 MB L3 cache (1.375 MB / core), average
Clock speed	1.29 GHz	2.5 GHz (3.80 GHz Turbo; Throttles to 2.3 GHz when all FP units are active)
Memory bandwidth	900 GB/s	119.21 GB/s (19.87 GB / channel / s)
Floating Point Throughput	7.8 Double Precision TFLOP/s 15.6 Single Precision TFLOP/s	2.0608 Double Precision TFLOP/s
TDP	300 Watts	205 Watts

[https://en.wikipedia.org/wiki/Nvidia\\_Tesla](https://en.wikipedia.org/wiki/Nvidia_Tesla)

<https://www.techpowerup.com/gpu-specs/tesla-v100-sxm2-32-gb.c3185>

[https://en.wikichip.org/wiki/intel/xeon\\_platinum/8180](https://en.wikichip.org/wiki/intel/xeon_platinum/8180)

<https://ark.intel.com/products/120496/Intel-Xeon-Platinum-8180-Processor-38-5M-Cache-2-50-GHz->

# Beefy GPU vs. beefy CPU comparison, 2021

	GPU – NVIDIA Tesla A100 40 GB \$9895	CPU – Intel® Xeon® Platinum 9282 \$20K
Processing Cores	6912 CUDA Scalar Processors 432 Tensor Cores	56 cores (112 threads, SMT)
Memory	40GB HBM2	- 32+32 KB L1 (I+D) cache / core (8-way assoc.) - 1 MB L2 cache / core (16-way assoc.) - 77 MB L3 cache (1.375 MB / core, average)
Clock speed	1.41 GHz (max)	2.6 GHz (3.80 GHz Turbo)
Memory bandwidth	1555 GB/s	262 GB/s
Floating Point Throughput	9.7 Double Precision TFLOP/s 19.5 Single Precision TFLOP/s	3.2 Double Precision TFLOP/s
TDP	250 Watts	400 Watts

[https://en.wikipedia.org/wiki/Nvidia\\_Tesla](https://en.wikipedia.org/wiki/Nvidia_Tesla)

[https://en.wikichip.org/wiki/intel/xeon\\_platinum/9282](https://en.wikichip.org/wiki/intel/xeon_platinum/9282)

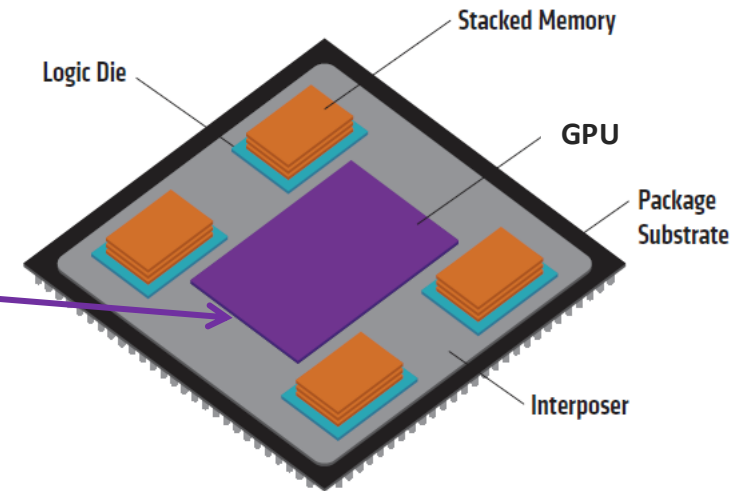
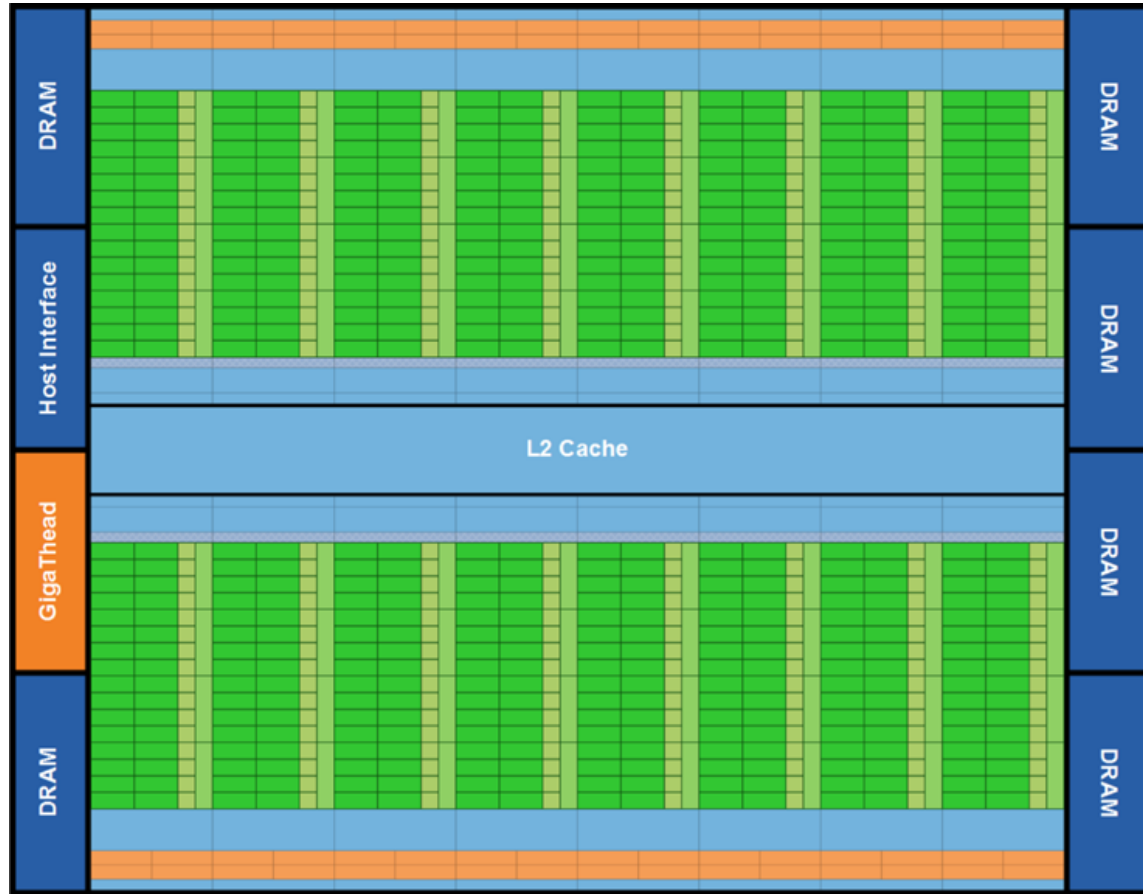
# Beefy GPU vs. beefy CPU comparison, 2023

	<b>GPU – NVIDIA Tesla H100 80 GB \$44,000</b>	<b>CPU – Intel® Xeon® Max 9480 \$12,980</b>
Processing Cores	14592 CUDA Scalar Processors 640 Tensor Cores & 128 RT Cores	56 cores (112 threads, SMT)
Memory	80GB HBM2	<ul style="list-style-type: none"><li>- 80 KB cache / core (8-way assoc.)</li><li>- 2 MB L2 cache / core (16-way assoc.)</li><li>- 112 MB L3 cache (2 MB / core, average)</li><li>- 64 GB HBM on package</li></ul>
Clock speed	1.75 GHz (max)	1.9 GHz (3.50 GHz Turbo)
Memory bandwidth	2039 GB/s [HBM]	307 GB/s [HBM]
Floating Point Throughput	25.6 Double Precision TFLOP/s 51.2 Single Precision TFLOP/s	??? Double Precision TFLOP/s
TDP	350 Watts	350 Watts

[https://en.wikipedia.org/wiki/Nvidia\\_Tesla](https://en.wikipedia.org/wiki/Nvidia_Tesla)

<https://www.intel.com/content/www/us/en/products/sku/232592/intel-xeon-cpu-max-9480-processor-112-5m-cache-1-90-ghz/specifications.html>

# GPU's Main Memory, "global memory," is off-chip





# NVIDIA Volta GV100: Microarchitecture Schematic

- 21.1 billion transistors
- GPC – 6 of them
  - GPU Processing Cluster
- TPC – 42 of them
  - Texture Processing Cluster
- SM – 84 of them
  - Stream multiprocessor
- A full GV100 has
  - Six GPCs, each of which has
    - Seven TCP, each with
      - Two SMs
- $6 \times 7 \times 2 = 84$  SMs



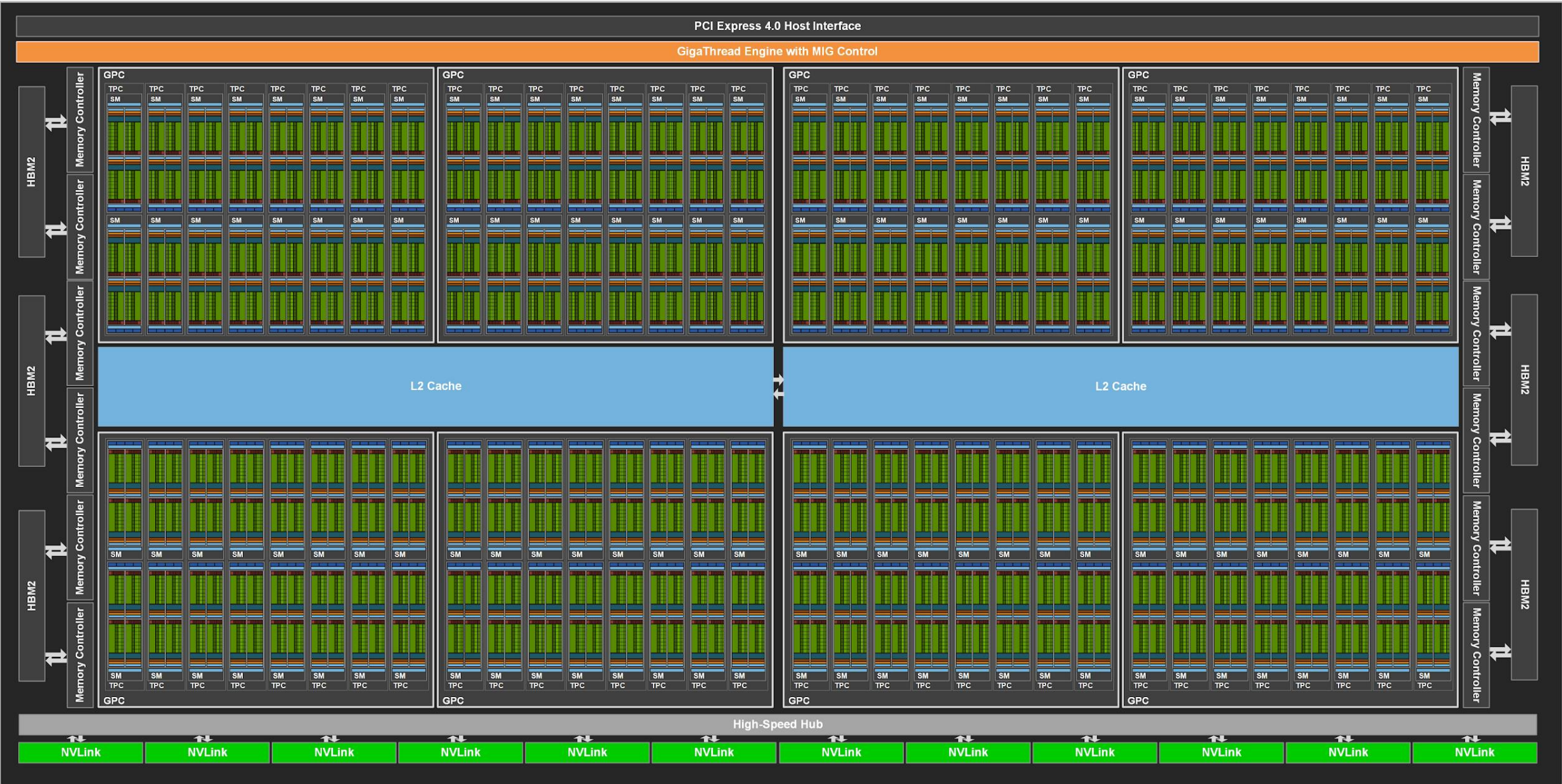


# NVIDIA Volta GV100: Microarchitecture Schematic (cont'd)

- Each SM has
  - 64 FP32 scalar processors (SPs)
  - 64 INT32 scalar processors (SPs)
  - 32 FP64 scalar processors (SPs)
  - 8 Tensor scalar processors (SPs)
  - Four texture units
  - Its own L0 Instruction Cache
  - A warp scheduler
  - A dispatch unit
  - 16,384 registers



# Ampere A100 (108 SMs)



[<https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/>]→

# NVIDIA Ampere A100: microarchitecture schematic [1/2]

- Each SM (out of the 108) partitioned into four processing blocks, called SM sub-partitions
- Each sub-partition contains the following:
  - Warp scheduler
  - Register file
  - Functional units
    - Integer Execution units
    - Floating Point Execution units
    - Memory Load/Store units
    - Special Function unit
    - Tensor Cores





# NVIDIA Ampere A100: microarchitecture schematic [2/2]

- There are several things shared amongst the four sub-partitions
  - Unified L1 Data Cache / Shared Memory
  - Texture units
  - RT Cores, if available (RT: ray tracing)



# Performance overview, over four NVIDIA GPU generations

- What lots of transistors buys you →

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS <sup>1</sup>	5	6.8	10.6	15.7
Peak FP64 TFLOPS <sup>1</sup>	1.7	.21	5.3	7.8
Peak Tensor TFLOPS <sup>1</sup>	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm <sup>2</sup>	601 mm <sup>2</sup>	610 mm <sup>2</sup>	815 mm <sup>2</sup>
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

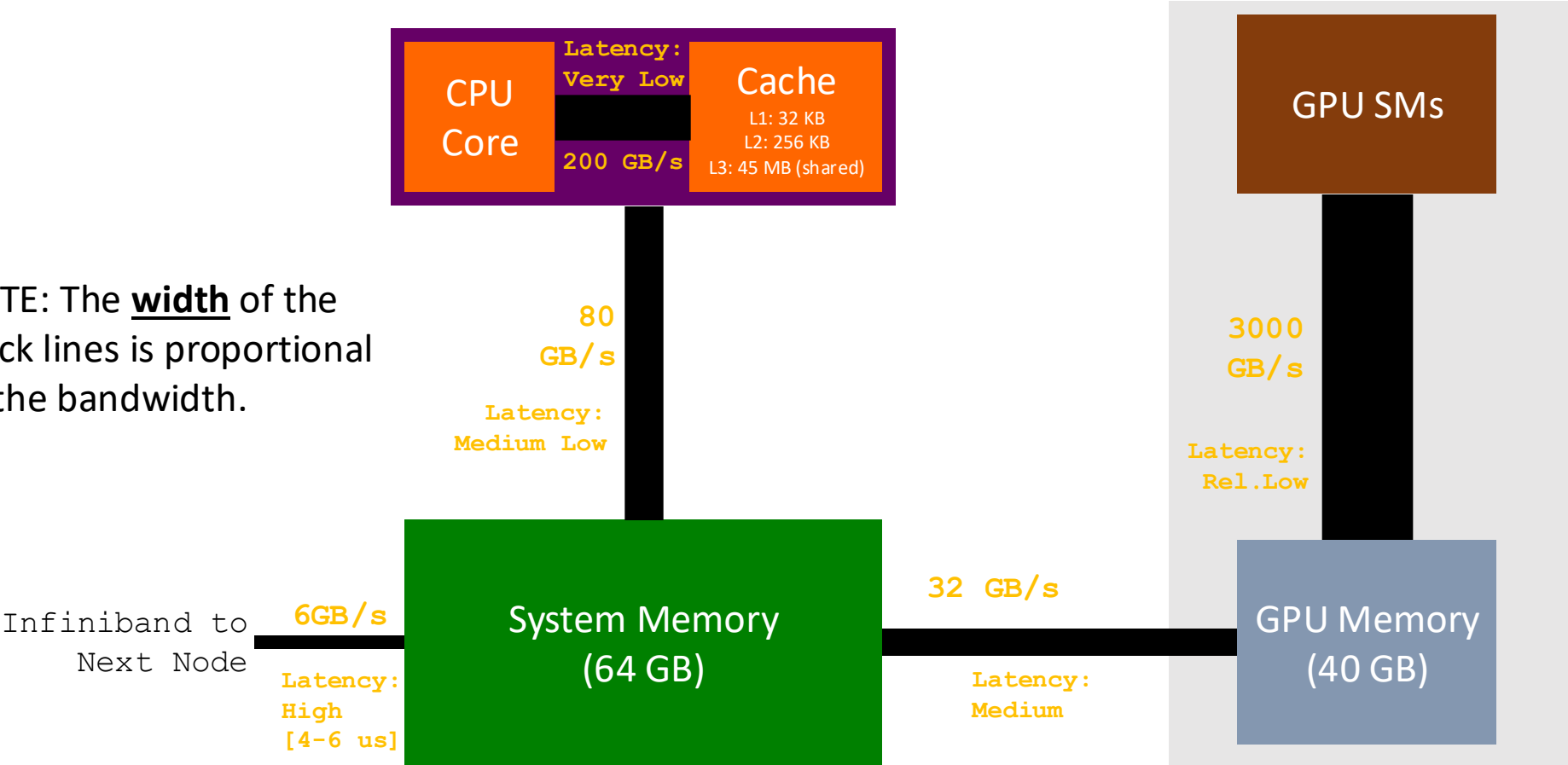
<sup>1</sup> Peak TFLOPS rates are based on GPU Boost Clock

# High Speed Requires High Bandwidth

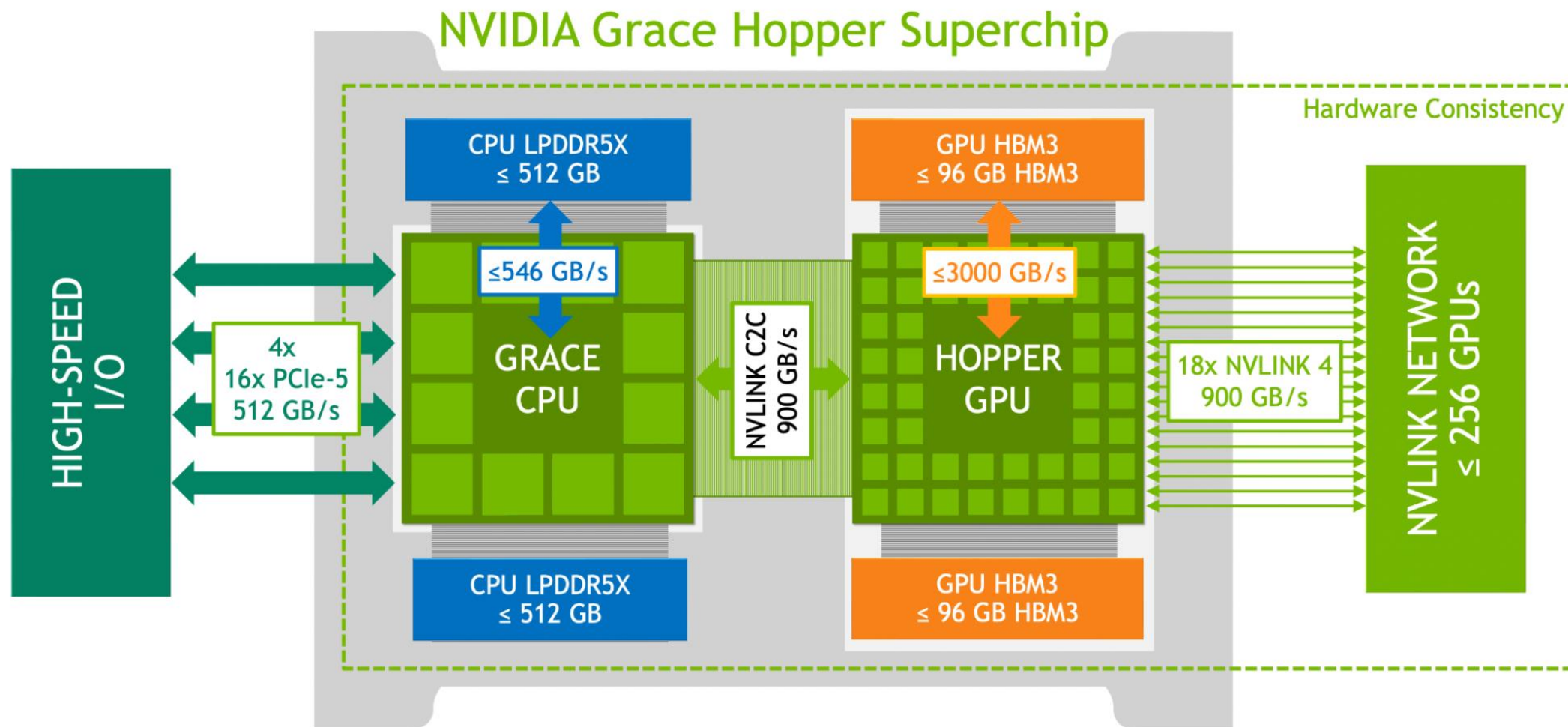
- Assume that you want to add two arrays and reach 1 Tflops:  $10^{12}$  ops/second
  - You need to load from memory  $2^{12}$  operands per second...
  - Assume each number is stored using  $4=2^2$  bytes (float or integer)
  - You need to fetch  $2^{12} \times 2^2$  bytes in a second. This is  $4^{12}$  B/s, which is 4 TB/s...
- Today's global memory bandwidth on the GPU
  - **NVIDIA GeForce RTX 4090:** Up to 1 TB/s with 24 GB GDDR6X memory
  - **NVIDIA GeForce RTX 4080:** Up to 716.8 GB/s with 16 GB GDDR6X memory
  - **NVIDIA A100 (Data Center):** Up to 1.5 TB/s with 40 GB or 80 GB HBM2e memory
  - **NVIDIA H100 (Data Center):** Up to 3 TB/s with HBM3 memory
  - **Intel Arc A770:** Up to 560 GB/s with 16 GB GDDR6 memory.

# Bandwidth in a CPU-GPU System: NVIDIA H100

NOTE: The width of the black lines is proportional to the bandwidth.



# NVIDIA Grace Hopper superchip





# Why “Grace Hopper”?

- Back when we didn’t have compilers:
  - Writing a program required knowledge of assembly and machine code
  - Programming in a human-readable language was an exotic concept
- 1955: Navy Rear Admiral Grace Hopper developed the prototype for the first compiled language, FLOW-MATIC
  - FLOW-MATIC influenced COBOL which in turn influenced many modern languages



Commodore Grace Hopper in 1985  
[1906-1992]

# Speed-ups, GPU Computing: What's reasonable to expect?

- Since you have higher bandwidth on a GPU, it is likely that your code will run faster on the right problem
  - Right problem: applications that exhibit highly data-parallel patterns with a large demand on data processing
- How much faster?
  - Compare the bandwidth/latency of mem accesses on GPU and CPU
  - Ball park: 5-7X speedup for data-parallel applications
  - Depends \*heavily\* on the nature of the problem solved: ML jobs sped up by a whole lot more
- Other things might come into play
  - Caches and cache size, for instance
  - The very nature of the problem you're trying to solve
    - Good for data parallelism, **fine grain parallelism**

# GPGPU Computing: when it started, why it started

- GPGPU: General Purpose GPU Computing
  - Done in early 2000s using graphics libraries
  - Before 2000, GPU are very much limited to gaming applications
- Why was GPGPU attractive?
  - GPUs had high bandwidths
  - Meet the need of machine learning applications: large amount of data needs to be moved into the GPU to process the inference and training workloads
    - Ex: LLM contains billions of parameters (and still keeps growing)

# “Hello World!”, on the GPU

- GPU is good for **fine grain** parallelism
  - More precisely, **data parallelism**


```
#include <cuda.h>
#include <stdio.h>

__global__ void simpleKernel() { printf("Hello World!\n"); }

int main() {
    const int numThreads = 4;

    // invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1, numThreads>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

The NVIDIA compiler, available on Euler



```
$ nvcc helloWorld.cu -o hello
$ ./hello
Hello World!
Hello World!
Hello World!
Hello World!
```



This is a bash shell, running under Linux

**NOTE:** do not do this on Euler, you'll get your account suspended if you don't use Slurm to run it.

# General-purpose GPU (GPGPU) Programming

- Now you see we can use GPU to do general-purpose programming
- The idea of GPGPU is to use the GPU as a co-processor
  - Farm out big parallel jobs to the GPU
  - CPU stays busy with the control of the execution and “corner” tasks
  - Data moved down into the GPU, and then results fetched back – idea works ok when data transfer overhead overshadowed by the number of operations applied to that data
- As a programmer, we should be aware of how data is moved between CPU and GPU to ensure the throughput performance can outweigh the data transfer cost

# The Cornerstone of GPGPU: CUDA

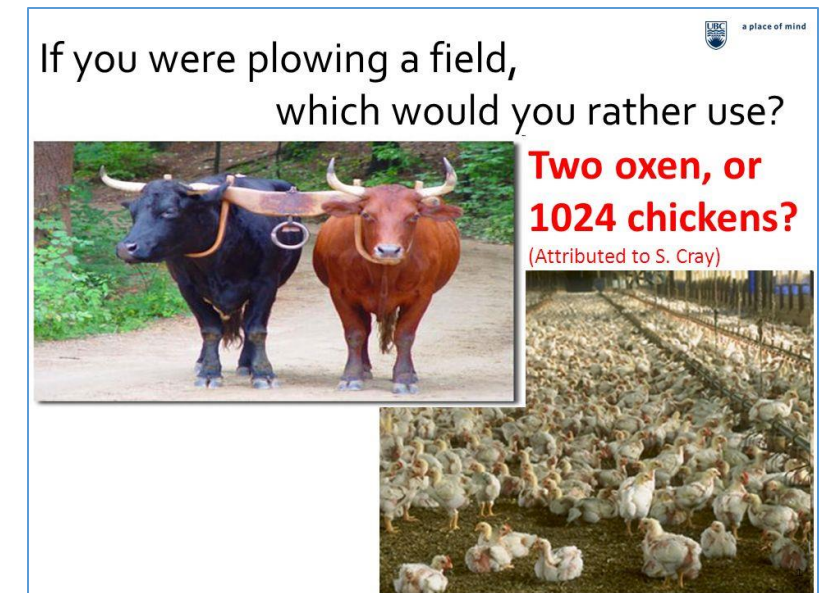
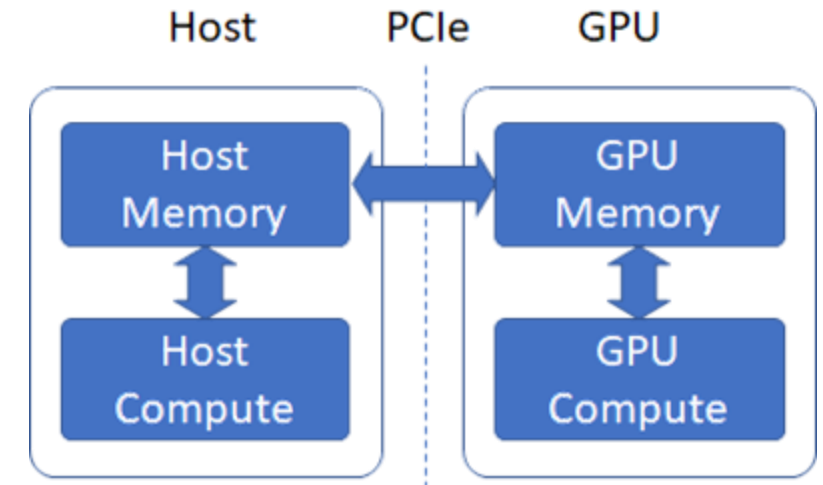
- CUDA: “Compute Unified Device Architecture” – freely distributed by NVIDIA
  - Now at version 12++
- When introduced (in 2006) it eliminated the graphics-constraints associated with GPGPU
  - Students learning CUDA need to sign NDA when I was in University of Illinois at Urbana-Champaign ...
- CUDA enables a general-purpose programming model for GPU programming
  - User kicks off batches of threads on the GPU to execute a function, namely *kernel*
- Targeted software stack
  - Scientific computing-oriented drivers, language, and tools
  - Interface designed for compute (i.e., graphics-free API)
  - Explicit GPU memory management

# Historical Milestones of CUDA

Year	Event
2003-2006	NVIDIA researchers explore GPGPU potential, leading to CUDA development.
2006	CUDA 1.0 is released with <b>GeForce 8800 GTX</b> , enabling general-purpose programming.
2007-2010	CUDA gains widespread adoption in <b>HPC, AI, and scientific computing</b> .
2012	Kepler GPUs introduce <b>dynamic parallelism</b> (GPUs launching their own kernels).
2017	Volta GPUs introduce <b>Tensor Cores</b> for deep learning acceleration.
2020 – now	CUDA dominates <b>HPC, AI, and ML</b> , with libraries like <b>cuBLAS, cuDNN</b> , and <b>TensorRT</b> .

# CUDA Programming Model: GPU as a Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a co-processor to the CPU or host
  - Has its own memory (device memory, in CUDA' language)
  - Runs many threads in parallel
  - Data transferred through PCIe between CPU and GPU
- Data-parallel portions of an app. run on the device as kernels executed in parallel by many threads
  - Kernel is the fundamental execution unit for Nvidia GPU
- Differences between GPU and CPU threads
  - GPU threads are **extremely lightweight**
    - Very little creation overhead
  - GPU **needs 1000s of threads** for full efficiency
    - Multi-core CPU needs only a few heavy ones



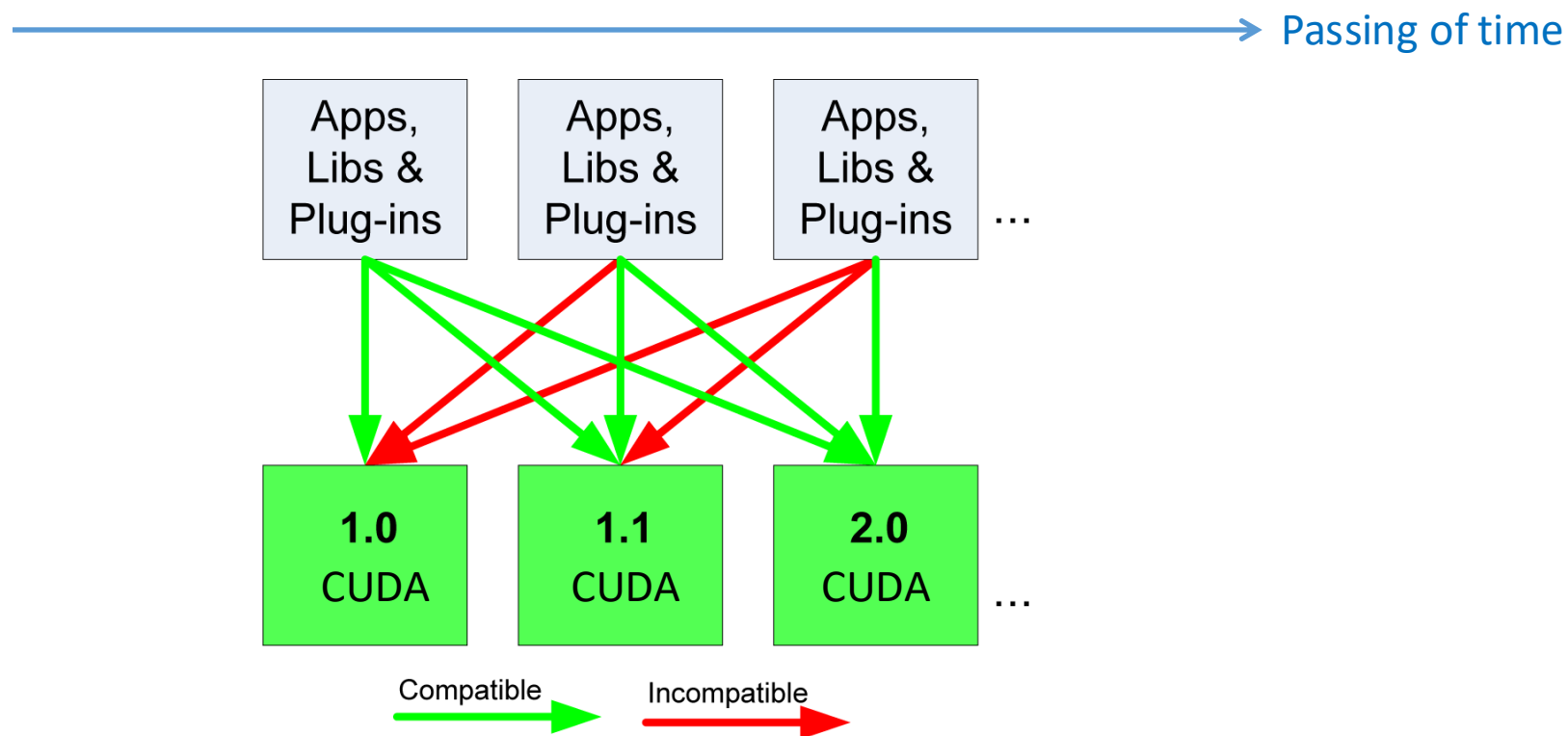


# Compute Capability [of a Device] vs. CUDA Version

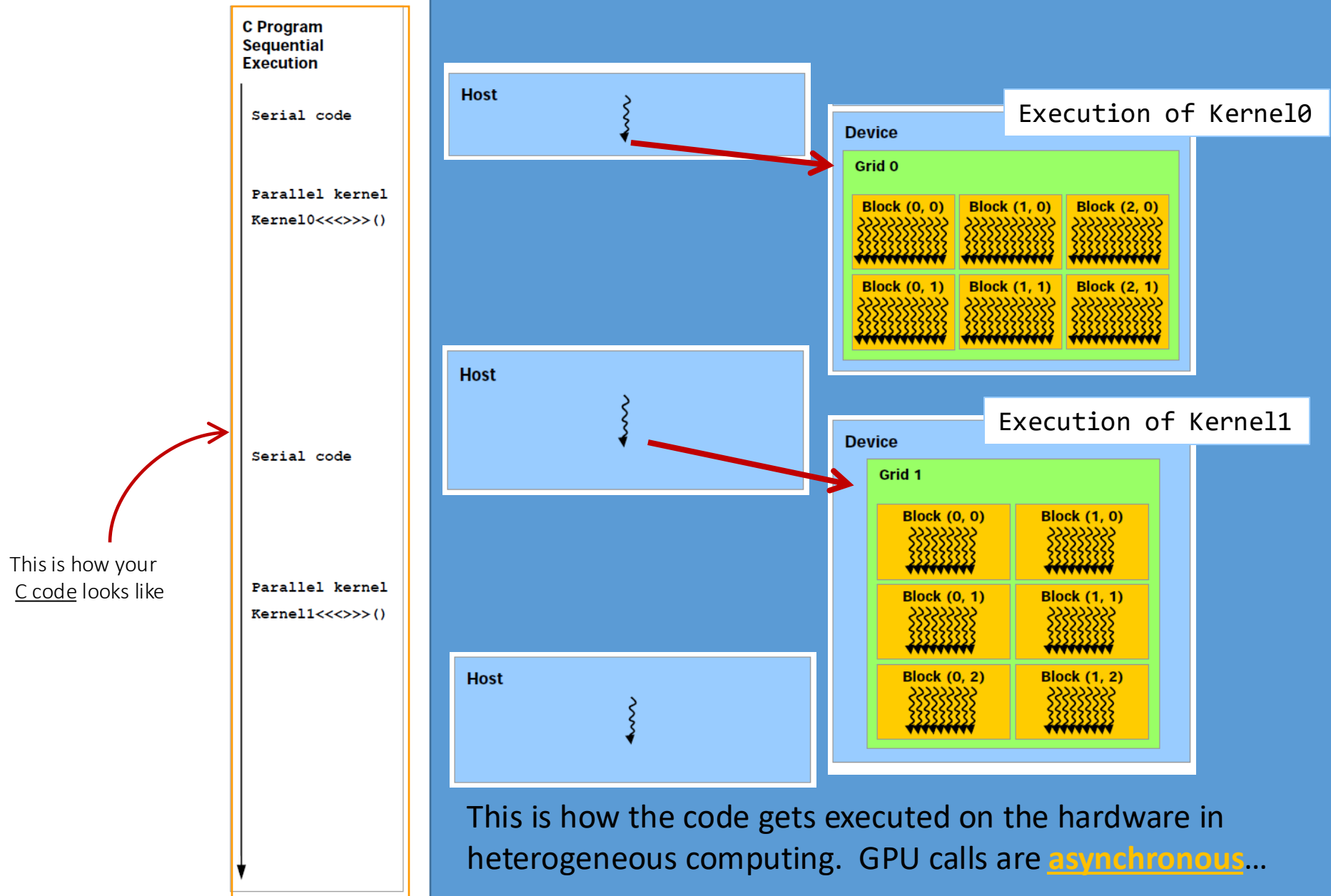
- “**Compute Capability**” is a term used to describe the feature set and capabilities of NVIDIA GPUs
  - Defined by a major revision number and a minor revision number
    - Tesla C1060 is compute capability 1.3
    - Fermi architecture is of capability 2
    - Kepler architecture is of capability 3
    - Maxwell architecture is of capability 5
    - Pascal architecture is of capability 6
    - Turing and Volta are of compute capability 7
    - Ampere is of compute capability 8
    - Hopper is of compute capability 9
  - A higher compute capability indicates a more able piece of Nvidia GPU
- The “**CUDA Version**” indicates what version of the software is used to manage on the hardware
- In a perfect world
  - You would run the most recent CUDA software release
  - You would use the most recent architecture (compute capability 9.0 for Hopper GPU)

# Compatibility Issues, software-wise

- The basic rule: the CUDA Driver API is backward, but not forward compatible
  - Code you ran ok w/ CUDA 8.0 will work w/ CUDA 10.0 as well (at least in theory)
    - Not the other way around though



# The CUDA Execution Model



# CUDA, First Example

```
#include<cuda.h>
#include<iostream>

__global__ void simpleKernel(int* data)
{
    //this adds a value to a variable stored in global memory
    data[threadIdx.x] += 2*(blockIdx.x + threadIdx.x);
}

int main()
{
    const int numElems = 4;
    int hostArray[numElems], *devArray;

    //allocate memory on the device (GPU); zero out all entries in this device array
    cudaMalloc((void**)&devArray, sizeof(int) * numElems);
    cudaMemset(devArray, 0, numElems * sizeof(int));

    //invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1,numElems>>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int) * numElems, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: " << std::endl;
    for (int i = 0; i < numElems; i++)
        std::cout << hostArray[i] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);
    return 0;
}
```



The NVIDIA compiler, available on Euler

```
$ nvcc firstExample.cu -o firstExample
$ ./firstExample
Values stored in hostArray:
0
2
4
6
```

This is a bash shell, running under Linux (ubuntu distro), via WSL2, on a Windows Laptop

NOTE: Do not run like this on Euler, you can get your account suspended. Use Slurm

# CUDA Function Declarations: (the “C with extensions” part)

	Executed on the:	Only callable from the:
<b>__device__</b> float myDeviceFunc()	device	device
<b>__global__ void</b> myKernelFunc()	device	host
<b>__host__</b> float myHostFunc()	host	host

- **\_\_global\_\_** defines a kernel function, launched by host, executed on the device
  - Must return **void**
- NOTE: can combine **\_\_device\_\_** with **\_\_host\_\_**
- See CUDA Reference Manual <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

# The concept of Execution Configuration

- A kernel function must be called with <<< execution configuration parameters >>>:
  - A three-dimensional grid to configure blocks – *how many blocks?*
  - A three-dimensional block to configure threads – *how many threads per block*
  - The shared memory size used by each block
  - Execution stream (job queue) to which this kernel is inserted
    - The CUDA stream is an in-order, first-come-first-server queue

```
__global__ void kernelFoo(...); // declaration

dim3 DimGrid(100, 50);           // 2D grid structure, w/ total of 5000 thread blocks
dim3 DimBlock(4, 8, 8);          // 3D block structure, with 256 threads per block

kernelFoo<<<DimGrid, DimBlock, shm, stream>>>(... arg list); // 5000x256 threads
```

# Example

- The number of threads running a kernel is co-decided by “grid” and “block”
- The host call below asks the GPU to execute the kernel “foo” using 25,600 threads
  - One dimensional grid of 100 blocks
  - One dimensional block of 256 threads

```
foo<<<100, 256>>>(p_matrixD, p_vectorD);
```

- The above execution configuration instructs the GPU to run 100 blocks each of 256 threads

# The CUDA Execution Model

- Data needs to be copied to GPU memory and the results need to be sent back to CPU
  - This happens over a PCI-Express 4.0/5.0 connection
  - Can also happen over NVLink (introduced by NVIDIA; not cheap to purchase)
- IMPORTANT:
  - For GPU computing to pay off, the data transfer overhead should be overshadowed by the GPU operations that draw on that transferred data
    - `cudaMalloc`: typical cost is 10–50  $\mu$ s, depending on memory size and GPU architecture
    - `cudaMemcpy`: typical cost is 10–100  $\mu$ s (host to device), depending on PCIe bandwidth
    - `Kernel<<<...>>>`: 2–10  $\mu$ s per launch
- The CUDA kernel calls and copying to/from GPU are managed by the CUDA runtime in a *separate job queue or stream* associated w/ the GPU execution



# GPU Job Queue: CUDA Stream

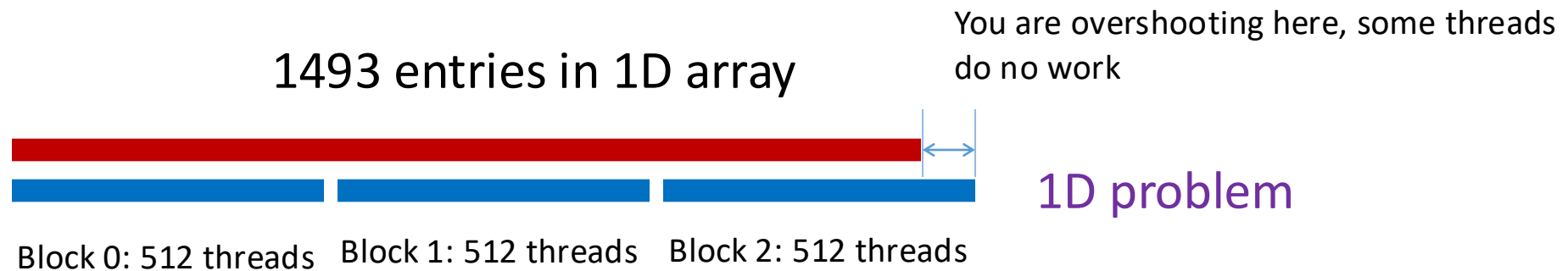
- CUDA stream has **nothing** to do with the SM (Streaming Multiprocessor)
  - SM is a piece of hardware
- The CUDA runtime places all calls that invoke the GPU in a stream of calls
  - This stream is an *ordered*, FIFO (first-in-first-out) stream
  - The first operation inserted into the stream gets executed first and so on
- More importantly, CUDA stream enables asynchronous execution between CPU and GPU
  - Host: minds its business, continuing execution right away after launching a kernel
  - Device: minds its business, taking on the next task in the sequence of tasks in the stream

# Further comments, on the “asynchronous” aspect of GPU computing

- Not everything related to GPU computing is asynchronous to the execution on the host
  - Example: A H2D or D2H memory data copy that is sufficiently large is synchronous
- You can even force everything to be synchronous during the execution of the code
  - Simply set `CUDA_LAUNCH_BLOCKING` environment variable to 1 to make everything synchronous
    - When you launch a CUDA operation (kernel, data movement, etc.), the program blocks until that CUDA operation finishes – *synchronous execution*
- If you force kernel calls to be synchronous, you lose the opportunity to do something on the host while the device is busy running your kernel

# Discussion on Execution Configuration

- Assume arrays of 1493 elements
  - Largest CUDA block that you can get has 512 threads
  - You'll have to use three CUDA blocks of threads, i.e.,  $\text{ceil}(1493/512) = 3$
  - You'll have some threads that do NO work (there'll be an `if` statement to check the boundary condition)



- Fundamental question that `a thread` asks in GPU computing: **What work do I have to do?**
  - In our 1D example, we have the following 1D mapping where index indicates the work of each thread

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

# CUDA, Another Example (1/2)

[Highlighted aspect: a thread figuring out its work order]

- Multiply, elementwise, two arrays of 3 million integers (i.e., element by element)

```
1.  int main(int argc, char* argv[])
2.  {
3.      const int arraySize = 3000000;
4.      int *hA, *hB, *hC;
5.      setupHost(&hA, &hB, &hC, arraySize);
6.
7.      int *dA, *dB, *dC;
8.      setupDevice(&dA, &dB, &dC, arraySize);
9.
10.     cudaMemcpy(dA, hA, sizeof(int) * arraySize, cudaMemcpyHostToDevice);
11.     cudaMemcpy(dB, hB, sizeof(int) * arraySize, cudaMemcpyHostToDevice);
12.
13.     const int threadsPerBlock = 512;
14.     const int blocksPerGrid = (arraySize + threadsPerBlock - 1)/threadsPerBlock;
15.     multiply_ab<<<blocksPerGrid, threadsPerBlock>>>(dA, dB, dC, arraySize);
16.     cudaMemcpy(hC, dC, sizeof(int) * arraySize, cudaMemcpyDeviceToHost);
17.
18.     cleanupHost(hA, hB, hC);
19.     cleanupDevice(dA, dB, dC);
20.     return 0;
21. }
```

For positive integer numbers where you want to find the ceiling of  $x$  divided by  $y$ :

$$\text{ceil}(x/(\text{float})y) = (x + y - 1) / y;$$

# CUDA, Another Example (2/2)

```
1.  __global__ void multiply_ab(int* a, int* b, int* c, int size)
2.  {
3.      int whichEntry = threadIdx.x + blockIdx.x * blockDim.x;
4.      if( whichEntry < size )
5.          c[whichEntry] = a[whichEntry] * b[whichEntry];
6.  }
```

The if statement is to prevent thread from going beyond the array boundary

```
1.  void setupDevice(int** pdA, int** pdB, int** pdC, int arraySize)
2.  {
3.      cudaMalloc((void**) pdA, sizeof(int) * arraySize);
4.      cudaMalloc((void**) pdB, sizeof(int) * arraySize);
5.      cudaMalloc((void**) pdC, sizeof(int) * arraySize);
6.  }
7.
8.  void cleanupDevice(int *dA, int *dB, int *dC)
9.  {
10.     cudaFree(dA);
11.     cudaFree(dB);
12.     cudaFree(dC);
13. }
```

# Forgetting `if( whichEntry < size )` is a Common Source of Bug

- The `if` is needed to prevent out of bounds indexing
  - Sometimes the product of the number of blocks  $\times$  number of threads per block is NOT exactly how many jobs need to be done
  - Size of the array to process may vary and not always a multiple of the block size

```
1.  __global__ void multiply_ab(int* a, int* b, int* c, int size)
2.  {
3.      int whichEntry = threadIdx.x + blockIdx.x * blockDim.x;
4.      // the code below may go beyond the boundary of the array - seg fault
5.      c[whichEntry] = a[whichEntry] * b[whichEntry];
6.  }
```

# More specifics: GPU operations that are **asynchronous**

- The following calls made on the host return the control to host right away
  - CUDA kernel launches
  - CUDA memory copies within a single device's memory
    - On the host, you ask the GPU to copy bytes from the GPU's device memory to some other GPU mem location
  - CUDA memory copies from host to device of a memory block of 64 KB or less
  - CUDA memory copies performed by functions that are suffixed with Async
  - CUDA “memory set” function calls

# The CUDA Execution Model: Three Opportunities for Asynchronous Execution

- Three opportunities, listed in increasing order of use/engagement complexity
  - Opportunity 1: The GPU and CPU work in asynchronous mode
    - The CPU moves on **right away** after a kernel launch
    - The GPU works at the same time the CPU works
  - Opportunity 2: the GPU has three engines that can work at the same time
    - copy-in engine + copy-out engine + execution engine
      - So as to overlap data movement with kernel execution
    - More later when we talk about streams
  - Opportunity 3: Multiple GPUs can work at the same time on one host