

# CIS 431/531

# Intro to Parallel Computing

GPUs and CUDA

# CPU vs. GPU

## CPU

- Instruction-level parallelism (ILP)
  - few “brawny” cores
  - general-purpose computing
- Reduce latency
  - large, complex memory hierarchy
  - hardware prefetching

## GPU

- Data-level parallelism (DLP)
  - many “wimpy” cores
  - high peak performance
- Increase throughput
  - large number of hardware threads
  - user-managed “scratchpad” cache

# System Comparison (2019)

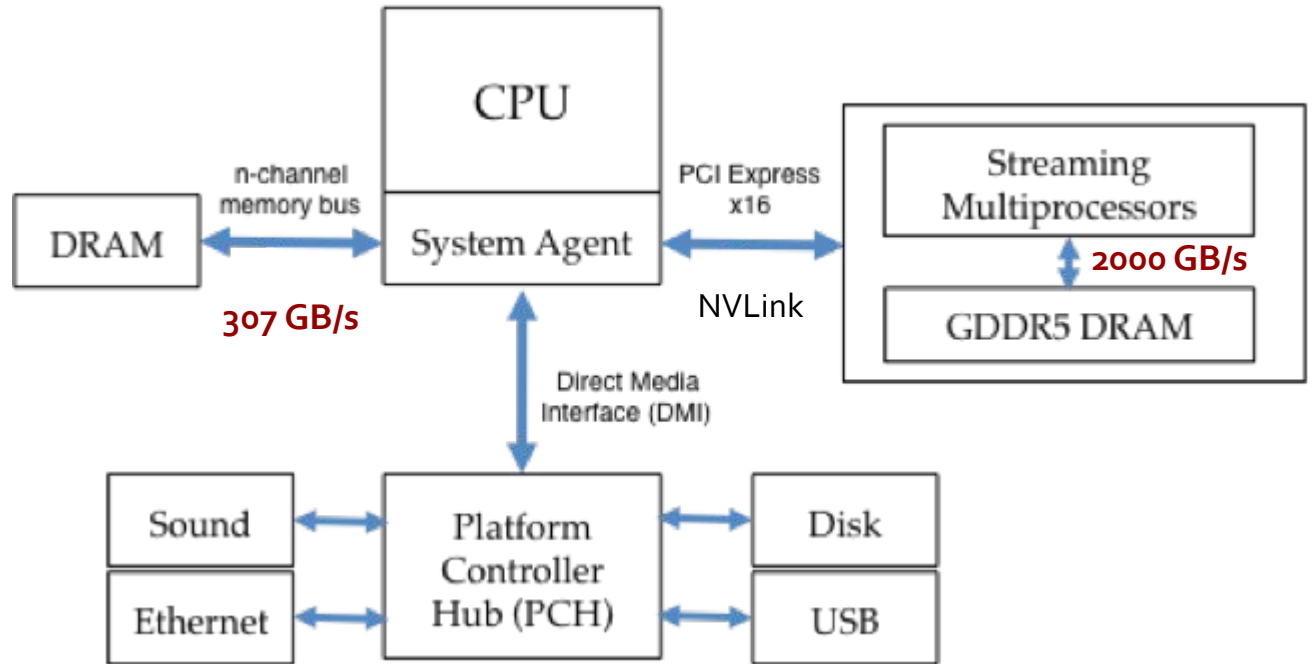
	Sapphire Rapids Platinum 8490H	Hopper H100 PCIe	Difference
# Cores/SM	60	114	1.90x
Clock (max)	3.5 GHz	1.71 GHz	0.49x
SIMD width	512 Bits	N/A	
CUDA cores	N/A	14592	
Performance (single-precision)	13.44 TFLOPS*	51 TFLOPS	3.80x
Performance (double-precision)	6.72 TFLOPS*	26 TFLOPS	3.86x
Bandwidth	307.2 GB/s	2000 GB/s	6.51x
TDP	350 Watts	350 Watts	1.00x

\*Difficult to achieve in practice - when using 2 AVX-512, CPU typically runs at a lower frequency due to power limitations

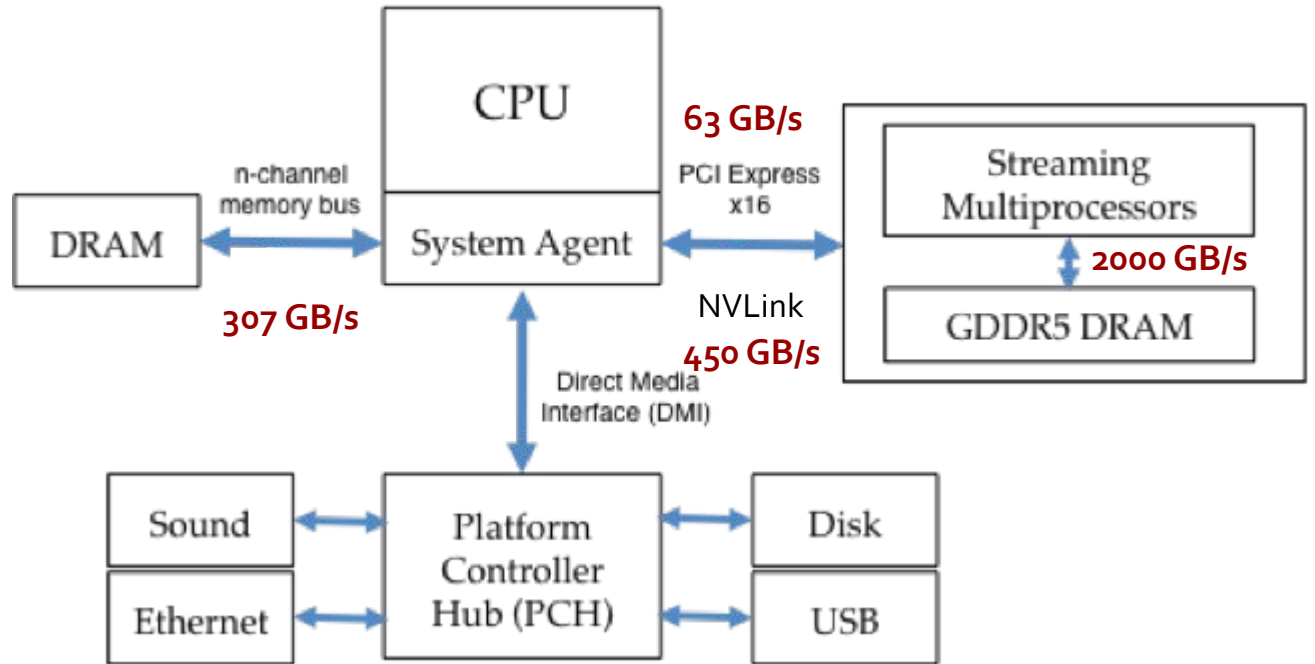
# System Comparison (2013)

	Intel Xeon E5-2687W	NVIDIA K20X	Difference
# Cores/SMX	8	14	1.75x
Clock frequency (max)	3.8 GHz	735 MHz	0.20x
SIMD Width	256-bits		
Thread processors		2688 SP + 896 DP	
Performance (single precision)	8 cores × 3.8 GHz × (8 Add + 8 Mul) = <b>486.4 GFLOPS</b>	2688 × 735 MHz × 2 (FMA) = <b>3.95 TFLOPS</b>	<b>8.12x</b>
Performance (double precision)	8 cores × 3.8 GHz × (4 Add + 4 Mul) = <b>243.2 GFLOPS</b>	896 × 735 MHz × 2 (FMA) = <b>1.32 TFLOPS</b>	<b>5.42x</b>
Memory bandwidth	51.2 GB/s	250 GB/s	4.88x
TDP	150 W	235 W	1.57x

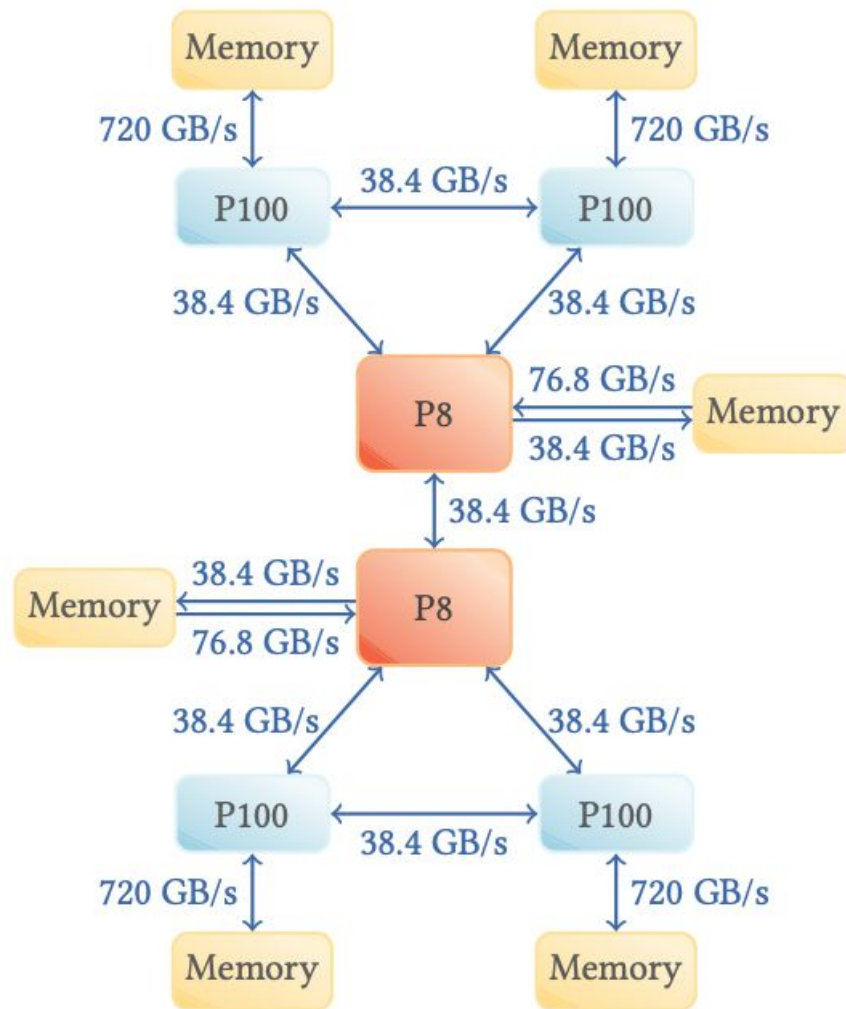
# Typical Compute Node



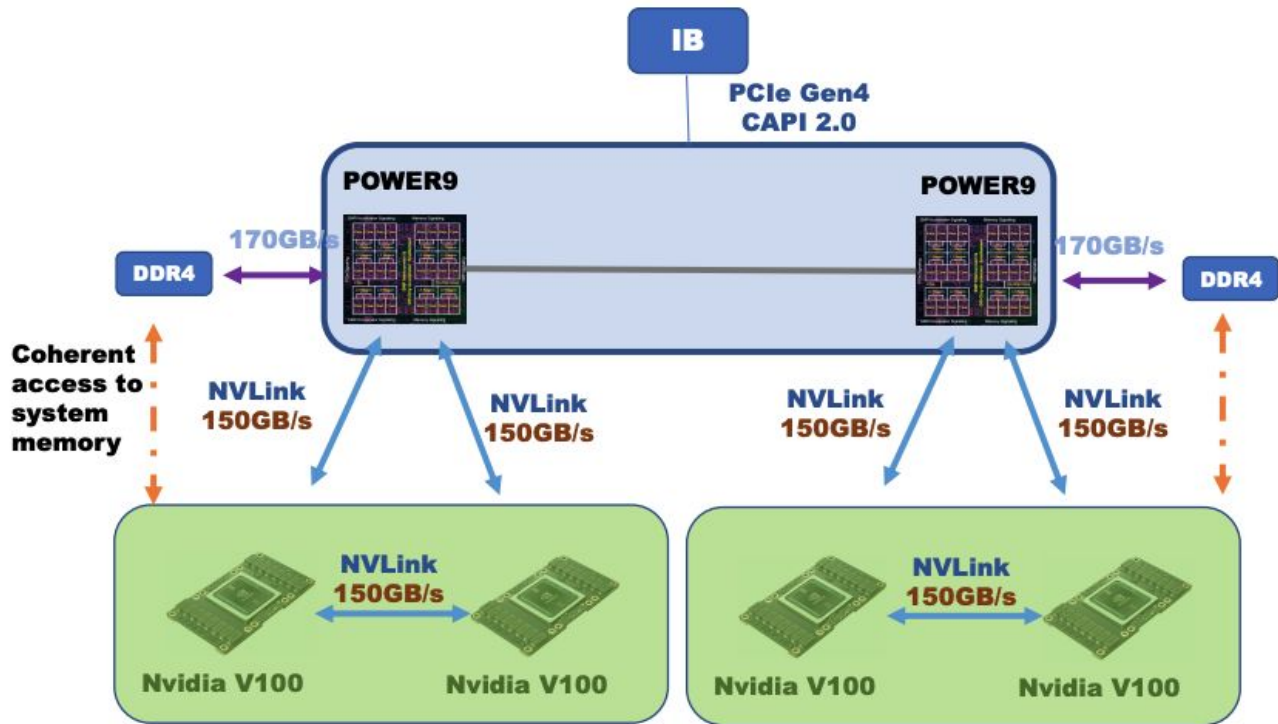
# Typical Compute Node



# NVLink 1.0



# NVLink 2.0





# Is GPU the Right Solution?

Given an application, is it a good idea to put it on the GPU?

If so, what is the expected speedup?

# Amdahl's Law Revisited

Maybe. Let's see what Amdahl says:

Consider

- Two systems: **Sapphire Rapids** and **Hopper**
- Workload  $W$  that uses **12GB of data** and takes  $T_{\text{CPU}} = 2 \text{ seconds}$  to run on the CPU
- 30% of  $W$  is serial and 70% is perfectly parallelizable
- What is  $T_{\text{GPU}}$ ?


$$\begin{aligned} T_{\text{GPU}} &= (12 \text{ GB} / 63 \text{ GB/s}) + 0.3 \times 2 \text{ seconds} + (0.7 \times 2 \text{ seconds}) / 3.86x \\ &= 0.96 \text{ seconds} \end{aligned}$$

$$\text{Speedup} = 2.1x$$

# Amdahl's Law Revisited

Maybe. Let's see what Amdahl says:

Consider

- Two systems: **Sapphire Rapids** and **Hopper**
- Workload  $W$  that uses **12GB of data** and takes  $T_{\text{CPU}} = 2$  seconds to run on the CPU
- 30% of  $W$  is serial and 70% is perfectly parallelizable
- What is  $T_{\text{GPU}}$ ?  **NVLink overhead**

$$T_{\text{GPU}} = (12 \text{ GB} / 63 \text{ GB/s}) + 0.3 \times 2 \text{ seconds} + (0.7 \times 2 \text{ seconds}) / 3.86x$$
$$= 0.96 \text{ seconds}$$

$$\text{Speedup} = 2.1x$$

# Amdahl's Law Revisited

Maybe. Let's see what Amdahl says:

Consider

- Two systems: **Sapphire Rapids** and **Hopper (with PCIe)**
- Workload  $W$  that uses **12GB of data** and takes  $T_{\text{CPU}} = 2$  seconds to run on the CPU
- 30% of  $W$  is serial and 70% is perfectly parallelizable
- What is  $T_{\text{GPU}}$ ?

$$\begin{aligned}T_{\text{GPU}} &= (12 \text{ GB} / 63 \text{ GB/s}) + 0.3 \times 2 \text{ seconds} + (0.7 \times 2 \text{ seconds}) / 3.86x \\ &= 1.15 \text{ seconds}\end{aligned}$$

$$\text{Speedup} = 1.73x$$

# Is GPU the Right Solution?

- Answer: **Not always**
  - Computation needs to be **sufficiently large** to hide/amortize PCIe transfer times
  - There needs to be **sufficient amount of parallelism** in the computation
- Of course, there are many other factors...

# CUDA

Programming model for GPUs (i.e., many-core architecture)

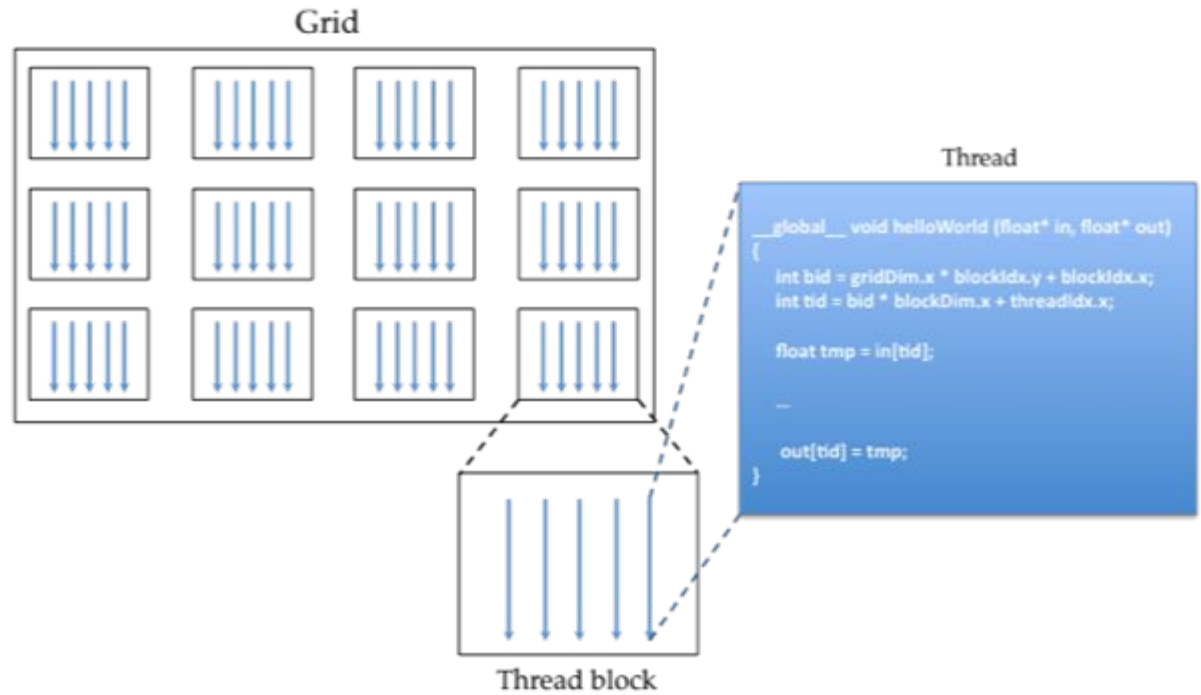
Single instruction multiple thread (SIMT)

- Large number of threads that all execute the same sequence of instructions
- Hardware multithreading allows fast switching between idle and active threads to hide latency

Designed to scale to different number of GPU cores

- Three key abstractions:
  - Thread hierarchy
  - Memory hierarchy, and
  - Synchronization

# Thread Hierarchy



# Example

## Naïve

```
for(i = 0; i < N; i++)  
{  
    A[i] += 2;  
}
```



# Example

## Naïve

```
for(i = 0; i < N; i++)  
{  
    A[i] += 2;  
}
```

## OpenMP

```
#pragma omp parallel for  
for(i = 0; i < N; i++) {  
    A[i] += 2;  
}
```

# Example

## Naïve

```
for(i = 0; i < N; i++)  
{  
    A[i] += 2;  
}
```

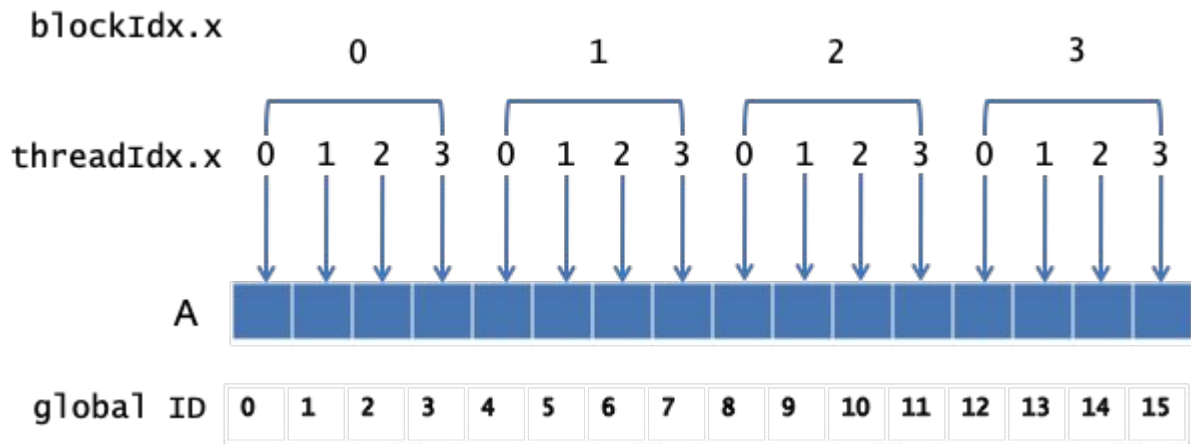
## OpenMP

```
#pragma omp parallel for  
for(i = 0; i < N; i++) {  
    A[i] += 2;  
}
```

## CUDA

```
int threadID = blockIdx.x * blockDim.x + threadIdx.x  
A[threadID] += 2;
```

# Mapping Threads



# Thread Blocks

Given a 3-D grid of thread blocks

- There are  $(\text{gridDim.x} * \text{gridDim.y} * \text{gridDim.z})$  thread blocks in the grid
- Each thread block's position is identified by `blockIdx.x`, `blockIdx.y`, and `blockIdx.z`

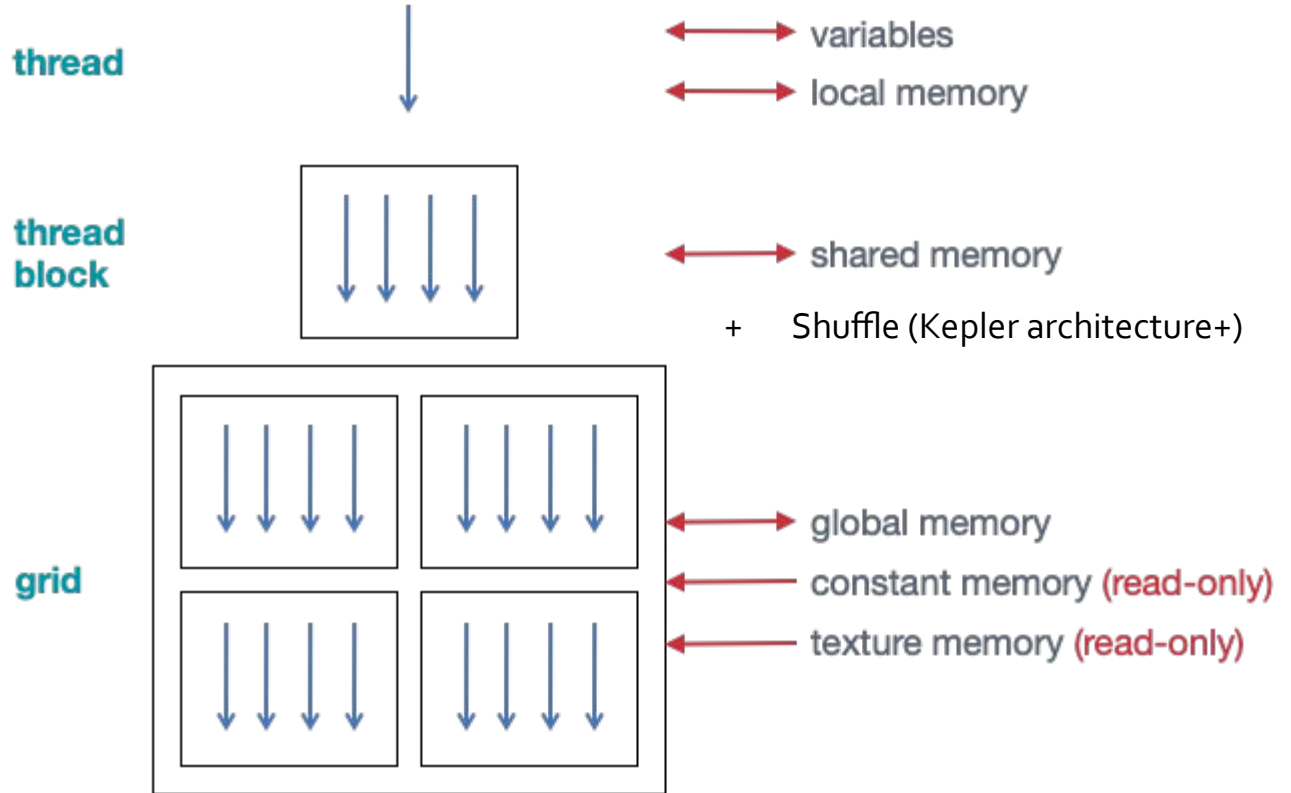
Similarly for a 3-D thread block

- `blockDim.x`, `blockDim.y`, `blockDim.z`
- `threadIdx.x`, `threadIdx.y`, `threadIdx.z`

Thread-to-data mapping depends on how the work is divided amongst the threads

- You can choose a “natural” mapping (i.e., one thread per data) or assign multiple data to a single thread (i.e., thread processes the assigned data serially)

# Memory Hierarchy



# Synchronization

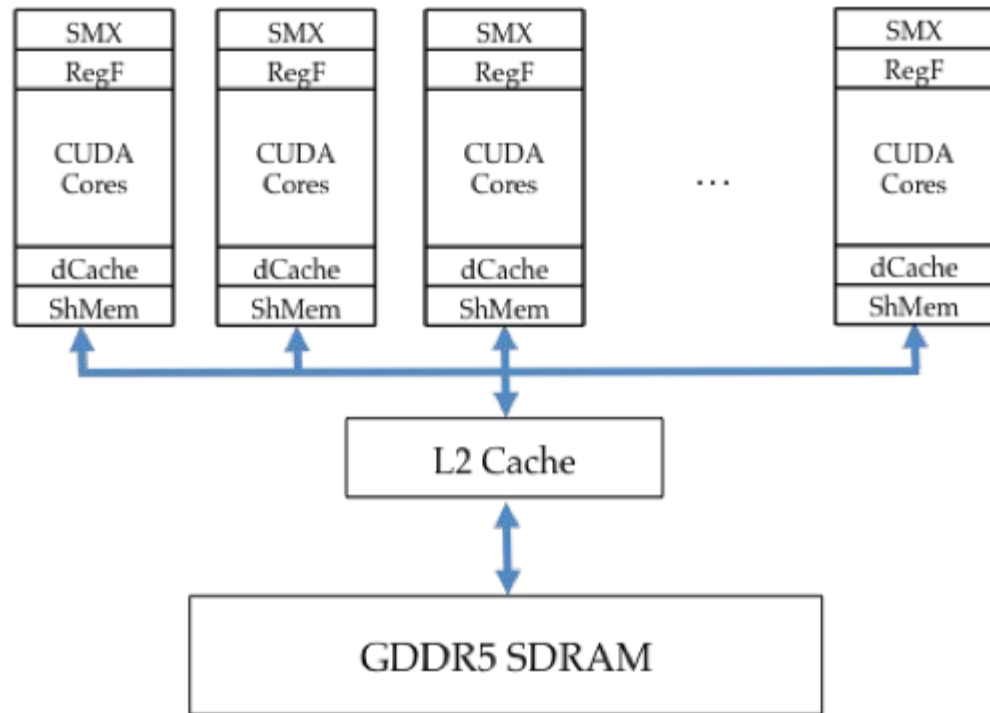
Within a thread block

- via `__syncthreads () ;`

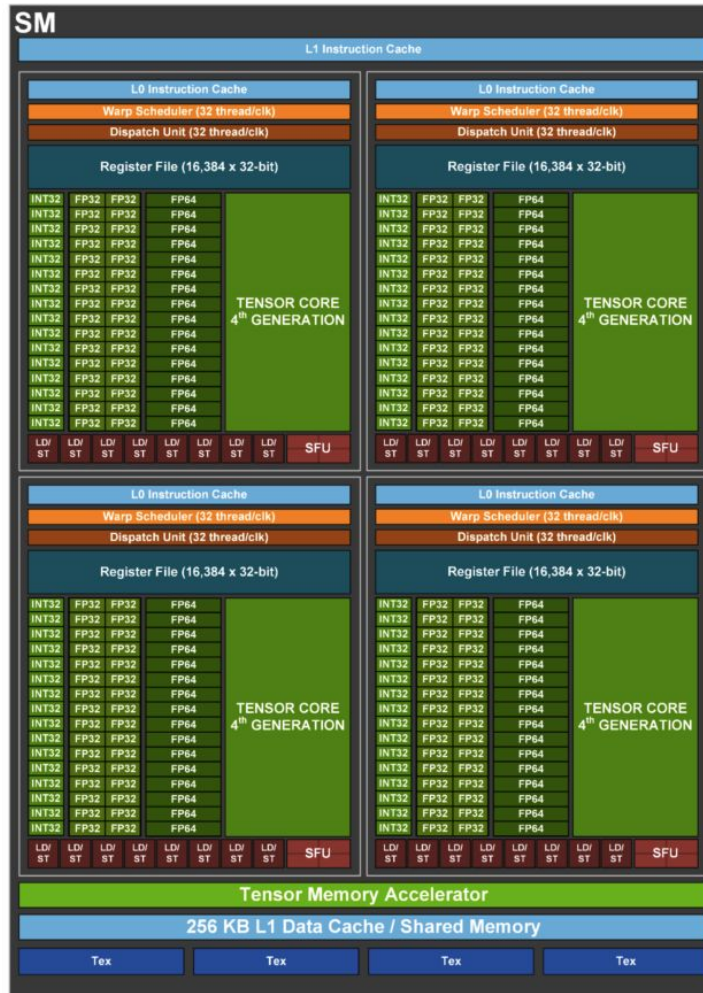
Global synchronization

- Implicit synchronization between **grids**
- Only way to synchronize globally is to finish the **grid** and start another **grid**

# GPU Architecture



# Volta Streaming Multiprocessor





# Memory Hierarchy

On CPUs

- **Registers** << **L1 (2.81MB)** << **L2 (120MB)** ~ = **L3 (112.5MB)**

On GPUs

- Register file:  $256 \text{ KB/SM} * 114 = 28.5 \text{ MB}$
- L1 cache:  $256 \text{ KB/SM} * 114 = 28.5 \text{ MB}$
- L2 cache: 50 MB
- **Registers == L1 (28.5MB) < L2 (50MB)**

CPU has few threads -> use cache to hide latency

GPU has many threads -> use threads to hide latency (but need a lot of registers to do hardware multithreading)

# Scheduling

Each thread blocks get scheduled on a multiprocessor (SM) by the GigaThread Engine

- There is **no guarantee in the order** in which they get scheduled
- Thread blocks run independently to each other

Multiple thread blocks can reside on a single SM simultaneously (occupancy)

- The number of thread blocks residing in a SM is determined by the resource usage and availability (typically shared memory and registers)

Once scheduled, each thread blocks runs to completion

# Execution

Minimum unit of execution: **warp**

- 32 threads
- Many optimizations are based on the behavior at warp level

At any given time, multiple warps will be executing

- Could be from the same or different thread blocks

A warp of threads could be either

- Executing
- Waiting (for data or their turn)

When a warp gets stalled, they could be switched out “instantaneously” so that another warp can start executing

- Hardware multithreading (thus SIMT)

# Performance Notes

On a branch, threads in a warp can diverge

- Execution is serialized – threads taking one branch executes while others idle (those cores are not working).

Avoid divergence!!!

- Use bitwise operation when possible
- Diverge at granularity of warps (no penalty)

\*\*On Volta (new with Volta)

- Each thread gets its own PC and call stack
- Threads can now diverge and converge at sub-warp granularity
- Idle threads can yield cores to other threads (cores are working)

Nevertheless, try to keep divergence to a minimum - there are other hardware bottlenecks that can reduce performance

# Performance Notes

Occupancy = # resident warps / max # warps

- # resident warps is determined by per-thread register and per-block shared memory usage
- Max # warps is specific to the hardware generation

More warps means more threads with which to hide latency

- Increases the chance of keeping the GPU busy at all times
- **Does not necessarily mean better performance**

# Performance Notes

Reading from the DRAM occurs at the granularity of 128 Byte transactions

- Requests are further decomposed to aligned cache lines
- L1 constant: 64 Bytes (Volta)
- L1 data: 32 Bytes (Volta)
- L2 cache: 64 Bytes (Volta)

Minimize loading redundant cache lines to maximize bandwidth utilization

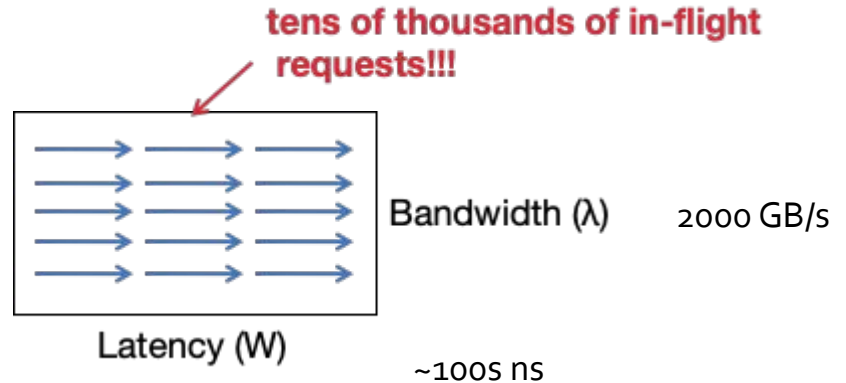
- Aligned access to memory
- Sequential access pattern

# Performance Notes

## Little's Law

- $L = \lambda W$
- $L$  = average number of customers in a store
- $\lambda$  = arrival rate
- $W$  = average time spent

## Memory Bandwidth



# Performance Notes

In summary...

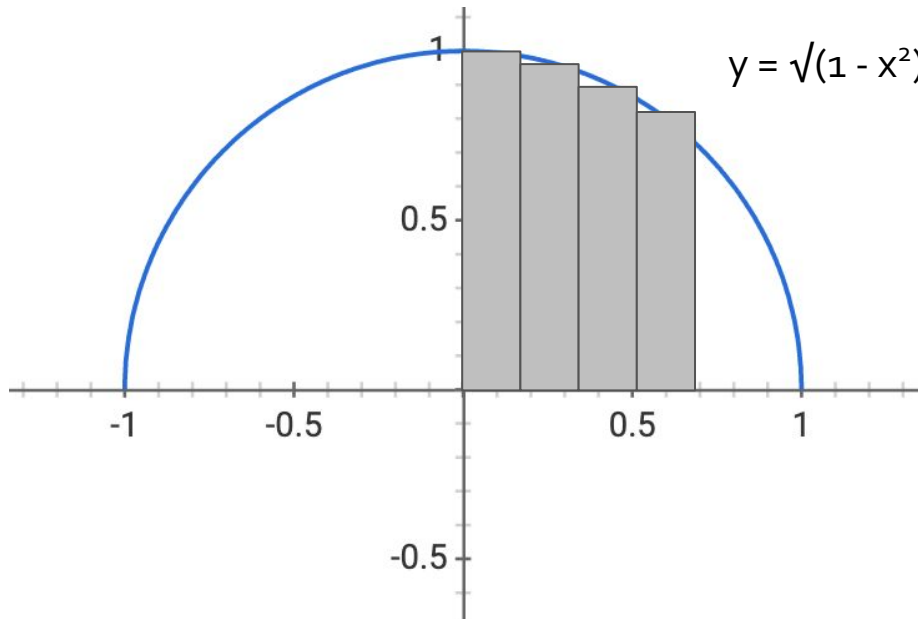
- Use as many “cheap” threads as possible
  - Maximizes occupancy
  - Increases the number of memory requests (to maximize bandwidth utilization)
- Avoid thread divergence
  - If unavoidable, diverge at the warp level
- Use aligned and sequential data access pattern
  - Minimize redundant data loads



Questions?

# Exercise - Pi

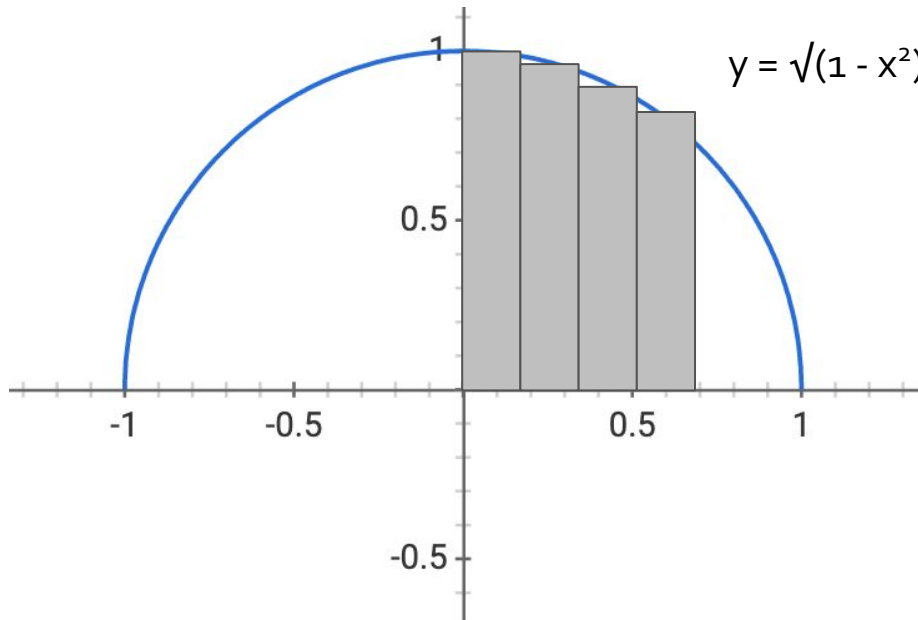
Estimate Pi using an integral of  $(\sqrt{1-x^2})$  from  $-1$  to  $1 = \pi/2$



## Exercise - Pi

Estimate Pi using an integral of  $(\sqrt{1-x^2})$  from  $-1$  to  $1 = \pi/2$

Estimate as sum of rectangular areas - more rectangles  $\rightarrow$  more accurate



# Exercise - Pi

```
long num_steps = 100000;
if(argc > 1) {
    num_steps = atoi(argv[1]);
}

double step;
double x;
double y;
double pi;
double sum = 0.0;

step = 1.0/(double) num_steps;
for (int i = 0; i < num_steps; i++) {
    x = i * step;
    y = sqrt(1 - x * x);
    sum = sum + y * step;
}
pi = 4 * sum;

printf("Pi is %1.10g\n", pi);
```

# Exercise - Pi

```
long num_steps = 100000;
if(argc > 1) {
    num_steps = atoi(argv[1]);
}

double step;
double x;
double y;
double pi;
double sum = 0.0;

step = 1.0/(double) num_steps;
#pragma omp parallel for private(x,y)
for (int i = 0; i < num_steps; i++) {
    x = i * step;
    y = sqrt(1 - x * x);
    sum = sum + y * step;
}
pi = 4 * sum;

printf("Pi is %1.10g\n", pi);
```

# Exercise - Pi

```
long num_steps = 100000;
if(argc > 1) {
    num_steps = atoi(argv[1]);
}

double step;
double x;
double y;
double pi;
double sum = 0.0;

step = 1.0/(double) num_steps;
#pragma omp parallel for private(x,y) reduction(+:sum)
for (int i = 0; i < num_steps; i++) {
    x = i * step;
    y = sqrt(1 - x * x);
    sum = sum + y * step;
}
pi = 4 * sum;

printf("Pi is %1.10g\n", pi);
```

# Exercise - Pi

```
Time to calculate Pi with 1.000000e+06 steps is: 0.0172998  
Pi is 3.141594652  
Time to calculate Pi with 1.000000e+06 steps is: 0.0871445  
Pi is 3.141594652
```

Why?

## Exercise - Pi

Using 56 thread and OMP\_PROC\_BIND=true

Time to calculate Pi with 1.000000e+06 steps is: 0.0172998

Pi is **3.141594652**

Time to calculate Pi with 1.000000e+06 steps is: 0.0871445

Pi is **3.141594652**

Time to calculate Pi with 1.000000e+07 steps is: 0.173032

Pi is **3.141592854**

Time to calculate Pi with 1.000000e+07 steps is: 0.78545

Pi is **3.141592854**

Time to calculate Pi with 1.000000e+08 steps is: 1.73137

Pi is **3.141592674**

Time to calculate Pi with 1.000000e+08 steps is: 8.03387

Pi is **3.141592674**

Why?

- Dynamic scheduling with chunk size 1 - each thread only does 1 iteration before being assigned another.
- Same issue as Fibonacci - more overhead than benefit



# Exercise - Pi

```
long num_steps = 100000;
if(argc > 1) {
    num_steps = atoi(argv[1]);
}

double step;
double x;
double y;
double pi;
double sum = 0.0;

step = 1.0/(double) num_steps;
#pragma omp parallel for private(x,y) reduction(+:sum)
schedule(static)
for (int i = 0; i < num_steps; i++) {
    x = i * step;
    y = sqrt(1 - x * x);
    sum = sum + y * step;
}
pi = 4 * sum;

printf("Pi is %1.10g\n", pi);
```

# Exercise - Pi

Using 56 thread and OMP\_PROC\_BIND=true

Time to calculate Pi with 1.000000e+06 steps is: 0.0174838

Pi is 3.141594652

Time to calculate Pi with 1.000000e+06 steps is: 0.0151599(**1.15x**)

Pi is 3.141594652

Time to calculate Pi with 1.000000e+07 steps is: 0.175225

Pi is 3.141592854

Time to calculate Pi with 1.000000e+07 steps is: 0.0201986(**8.68x**)

Pi is 3.141592854

Time to calculate Pi with 1.000000e+08 steps is: 1.67194

Pi is 3.141592674

Time to calculate Pi with 1.000000e+08 steps is: 0.0622832(**26.84x**)

Pi is 3.141592674

Time to calculate Pi with 1.000000e+09 steps is: 16.8294

Pi is 3.141592656

Time to calculate Pi with 1.000000e+09 steps is: 0.429524(**39.18x**)

Pi is 3.141592656

Time to calculate Pi with 1.410065e+09 steps is: 23.7939

Pi is 3.141592655

Time to calculate Pi with 1.410065e+09 steps is: 0.595115(**39.98x**)

Pi is 3.141592655

## Exercise - Pi

```
long num_steps = 100000;
if(argc > 1) {
    num_steps = atoi(argv[1]);
}

double step;
double x;
double y;
double pi;
double sum = 0.0;

step = 1.0/(double) num_steps;
#pragma omp parallel for private(x,y) reduction(+:sum)
schedule(static)
for (int i = 0; i < num_steps; i++) {
    x = i * step;
    y = sqrt(1 - x * x);
    sum = sum + y * step;
}
pi = 4 * sum;

printf("Pi is %1.10g\n", pi);
```

How would you implement this without reduction?

## Exercise - Pi

```
int nThreads = omp_get_max_threads();
double* sum_t = (double*) malloc(sizeof(double) * nThreads);
for(int i = 0; i < nThreads; i++) {
    sum_t[i] = 0.0;
} /* in main function */
```

```
double step;
double x;
double y;
double pi;
double sum = 0.0;
```

```
step = 1.0/(double) num_steps;
#pragma omp parallel for private(x,y) schedule(static)
for (int i = 0; i < num_steps; i++) {
    int tid = omp_get_thread_num();
    x = i * step;
    y = sqrt(1 - x * x);
    sum_t[tid] = sum_t[tid] + y * step;
}
```

```
for(int i = 0; i < omp_get_max_threads(); i++) {
    sum += sum_t[i];
}
pi = 4 * sum;
```

```
return pi;
```

Would this be faster, slower, or the same as  
Implementation using reduction, and why?

# Exercise - Pi

```
step = 1.0/(double) num_steps;
#pragma omp parallel private(x,y,sum)
{
    sum = 0.0;
    #pragma omp for schedule(static)
    for (int i = 0; i < num_steps; i++) {
        int tid = omp_get_thread_num();
        x = i * step;
        y = sqrt(1 - x * x);
        sum = sum + y * step;
    }
    #pragma omp atomic
    total_sum += sum;
}

pi = 4 * total_sum;
```