

Applications of Parallel Computers

Introduction to Graphics Processing Units (GPUs)

<https://sites.google.com/lbl.gov/cs267-spr2021>



Acknowledgements

- Slides from today's lecture are derived from
 - John Owens, UC Davis
 - Bill Dally, Stanford / NVIDIA
 - Kayvon Fatahalian, Stanford
 - Kurt Keutzer, Berkeley
 - Brian Catanzaro, NVIDIA (Berkeley grad)

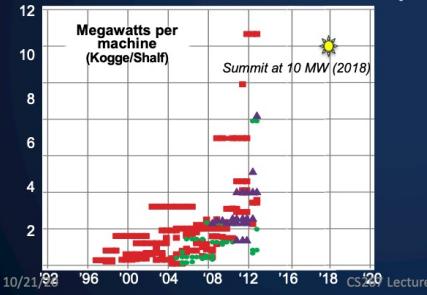
10/21/20

CS267 Lecture

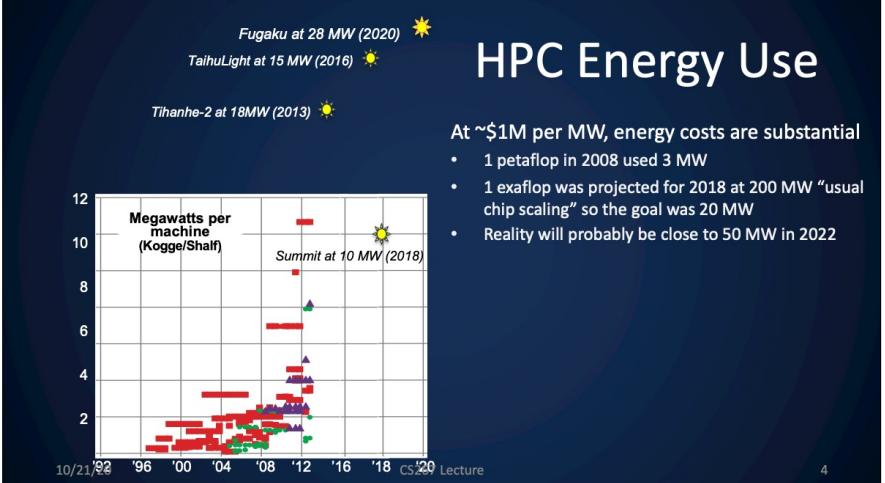
2

HPC Energy Use

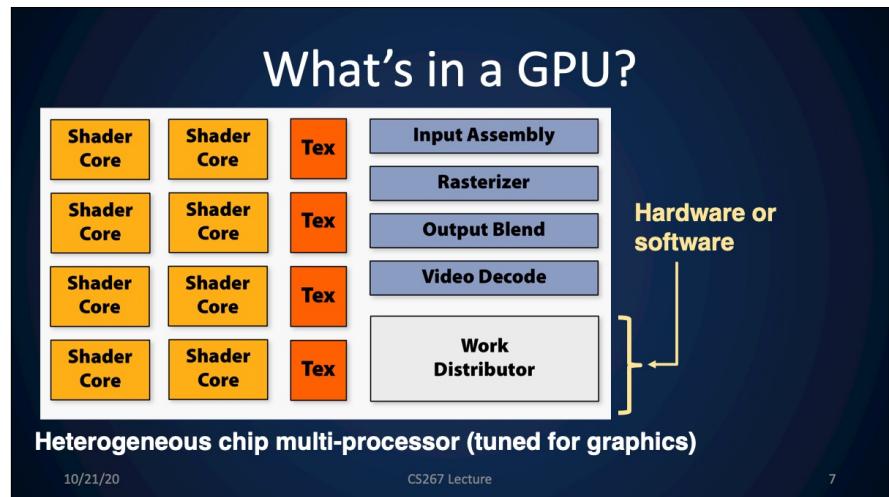
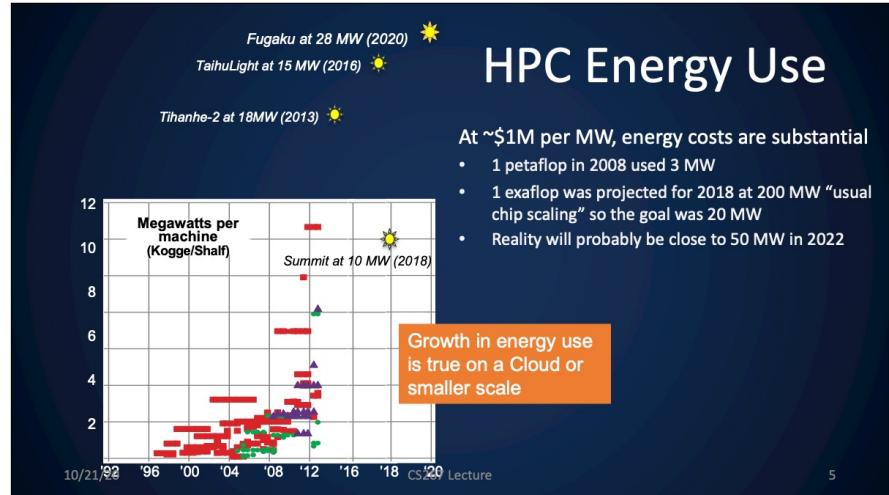
- At ~\$1M per MW, energy costs are substantial
- 1 petaflop in 2008 used 3 MW
 - 1 exaflop was projected for 2018 at 200 MW “usual chip scaling” so the goal was 20 MW



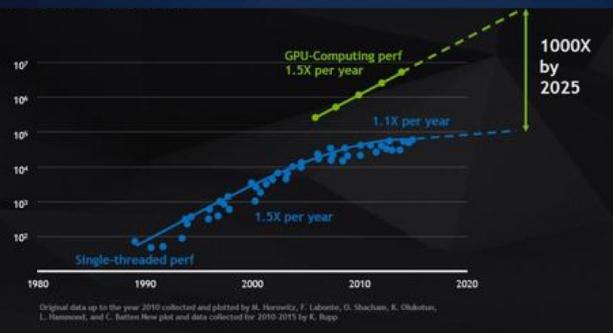
3



4



GPUs: The Hype

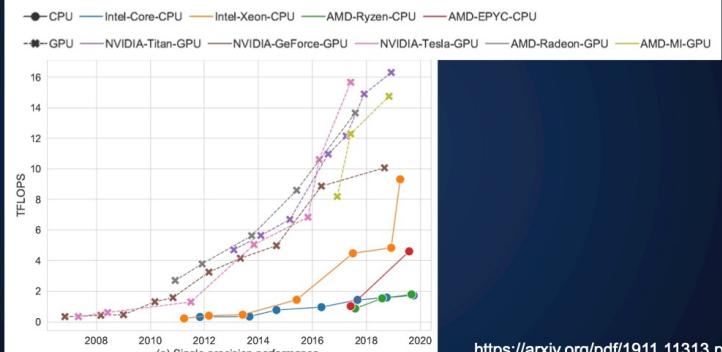


10/21/20

CS267 Lecture

30

GPUs: A More Balanced Analysis

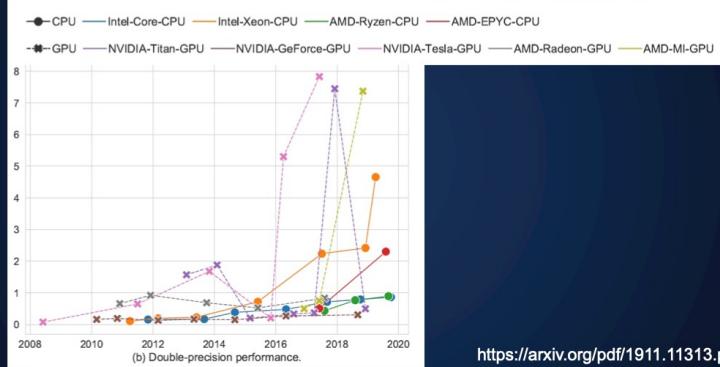


10/21/20

CS267 Lecture

31

GPUs: A More Balanced Analysis

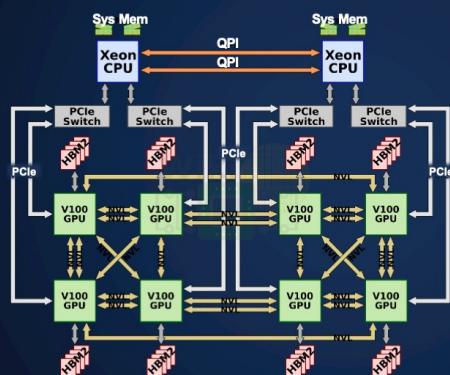


10/21/20

CS267 Lecture

32

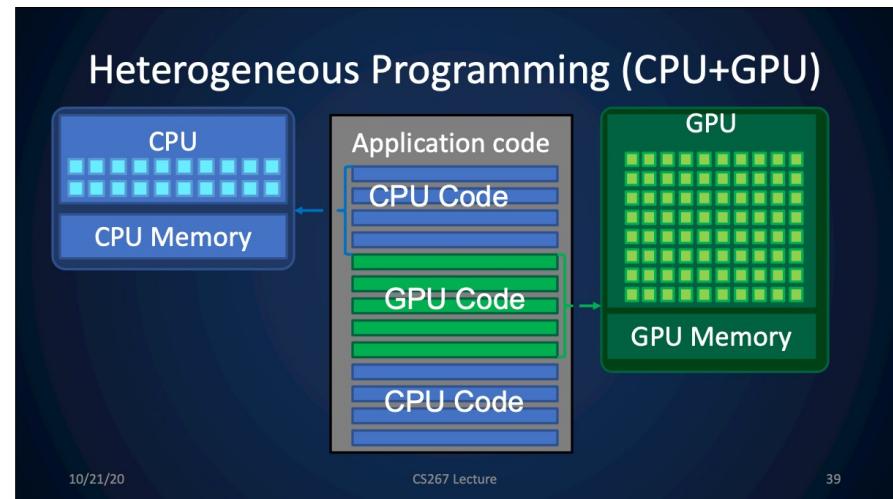
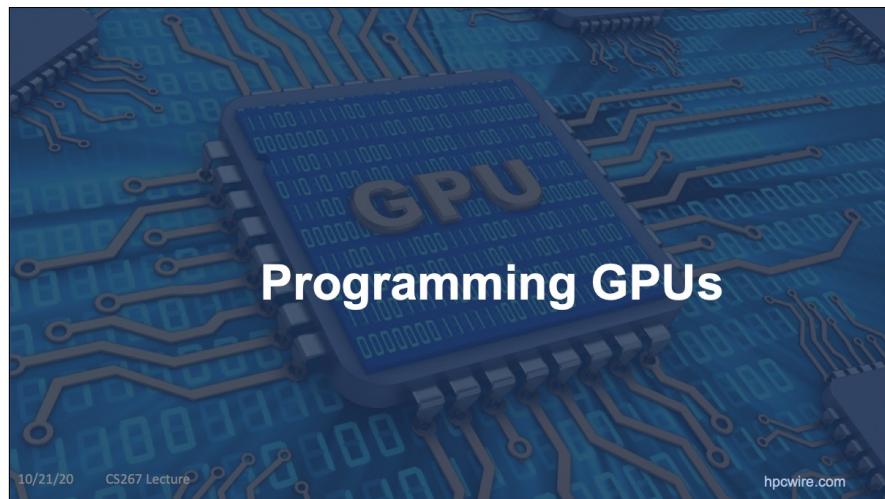
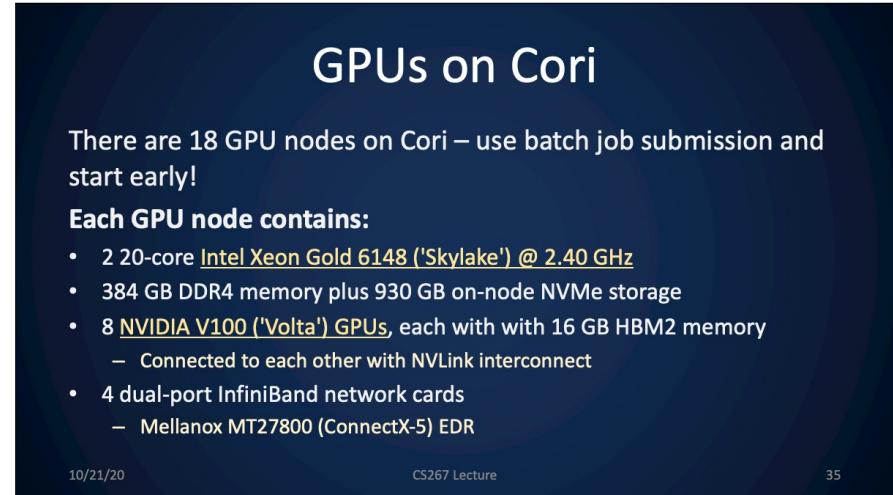
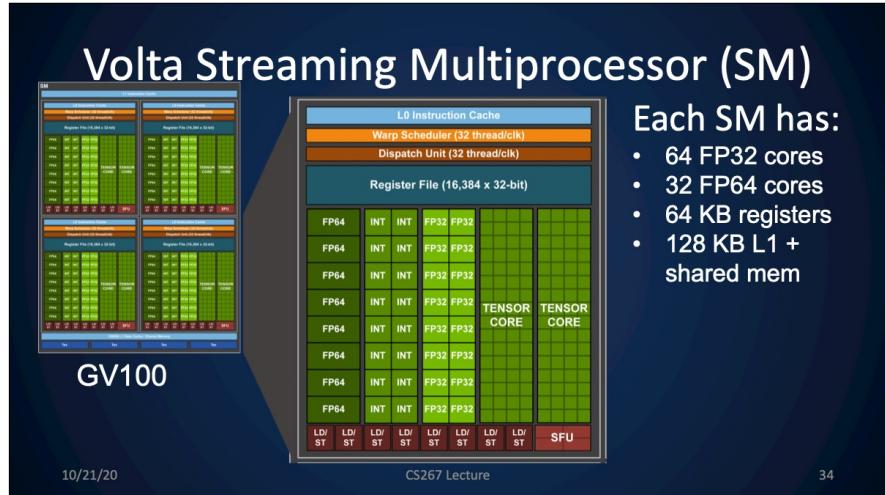
Cori GPU Node Architecture



10/21/20

CS267 Lecture

33



Example: Vector Addition (CPU)

```
#include <iostream>
int main(void) {
    int N = 1<<20; // 1M elements
    float *x = new float[N]; // Allocate memory
    float *y = new float[N];

    // Initialize x and y on the CPU
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f; y[i] = 2.0f;
    }

    // Free memory
    delete [] x; delete [] y;
    return 0;
}
```

10/21/20

CS267 Lecture

40

Example: Vector Addition (CPU)

```
#include <iostream>
int main(void) {
    int N = 1<<20; // 1M elements
    float *x = new float[N]; // Allocate memory
    float *y = new float[N];
    // Initialize x and y on the CPU
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f; y[i] = 2.0f;
    }

    // Free memory
    delete [] x; delete [] y;
    return 0;
}
```

10/21/20

CS267 Lecture

41

Example: Vector Addition (CPU)

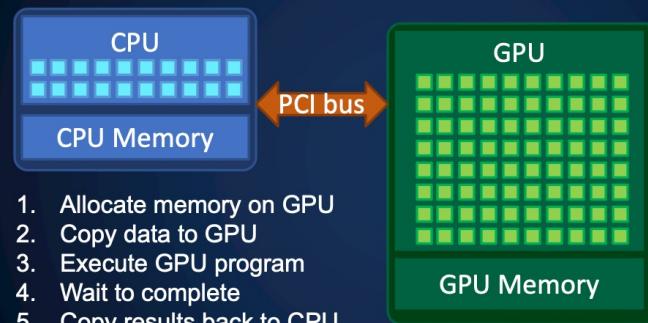
```
#include <iostream>
int main(void) {
    int N = 1<<20; // 1M elements
    float *x = new float[N];
    float *y = new float[N];
    // Initialize x and y on the CPU
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f; y[i] = 2.0f;
    }
    // Run on the CPU
    add(N, x, y);
    // Free memory
    delete [] x; delete [] y;
    return 0;
}
```

10/21/20

CS267 Lecture

42

Running GPU Code (Kernel)



10/21/20

CS267 Lecture

43

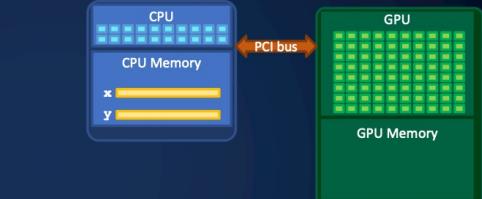
Example: Vector Addition (GPU Serial)

```
float *x = new float[N];
float *y = new float[N];
// initialize x and y on the CPU
for (int i = 0; i < N; i++) {
    x[i] = 1.0f; y[i] = 2.0f;
}
```

```
// Run on the CPU
add(N, x, y);
```

```
// Free memory
delete [] x; delete [] y;
```

10/21/20



CS267 Lecture

44

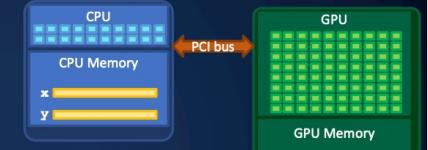
Example: Vector Addition (GPU Serial)

```
float *x = new float[N]; initialization
float *y = new float[N]; not show
```

```
// Run on the CPU
add(N, x, y);
```

```
// Free memory
delete [] x; delete [] y;
```

10/21/20



CS267 Lecture

45

Example: Vector Addition (GPU Serial)

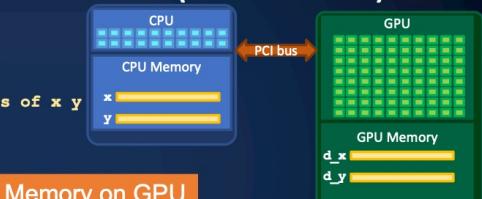
```
float *x = new float[N];
float *y = new float[N];
int size = N*sizeof(float);
float *d_x, *d_y; // device copies of x y
cudaMalloc((void **) &d_x, size);
cudaMalloc((void **) &d_y, size);
```

1) Allocate Memory on GPU

```
// Run on the CPU
add(N, x, y);
```

```
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;
```

10/21/20



And free on GPU

CS267 Lecture

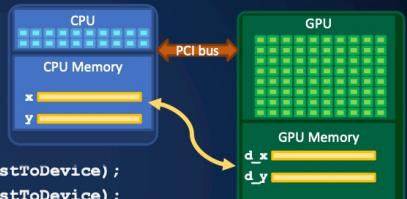
46

Example: Vector Addition

```
float *x = new float[N];
float *y = new float[N];
int size = N*sizeof(float);
float *d_x, *d_y; // device copies of x y
cudaMalloc((void **) &d_x, size);
cudaMalloc((void **) &d_y, size);
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
// Run kernel on GPU
add<<<1,1>>>(d_x, d_y);
```

```
// Copy result back to host
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;
```

10/21/20



2. Copy data to GPU

CS267 Lecture

47

5. Copy results back to CPU

Example: Vector Addition

```

float *x = ...;
float *y = ...;
int size = ...;

float *d_x, *d_y; // device copies of x y
cudaMalloc((void **) &d_x, size);
cudaMalloc((void **) &d_y, size);
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);

// Run kernel on GPU
add<<<1,1>>>(N, d_x, d_y);
// Copy result back to host
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;

```

Function runs on GPU, callable from CPU

// GPU function to add two vectors
`_global_ void add(int n, float *x, float *y) { for (int i = 0; i < n; i++) y[i] = x[i] + y[i]; }`

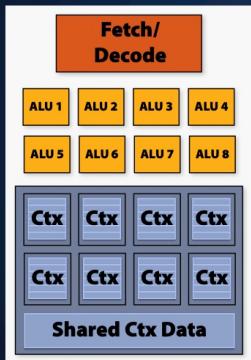
3. Run on GPU
 4. Wait to complete

10/21/20

CS267 Lecture

48

Programming as Threads



Program with multiple copies of program

Have each thread compute on different elements

Ctx = Context

10/21/20

CS267 Lecture

53

Example: Vector Addition

```

float *x = ...;
float *y = ...;
int size = ...;

float *d_x, *d_y; // device copies of x y
cudaMalloc((void **) &d_x, size);
cudaMalloc((void **) &d_y, size);
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);

// Run kernel on GPU
add<<<1,1>>>(N, d_x, d_y);
// Copy result back to host
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;

```

Function runs on GPU, callable from CPU

// GPU function to add two vectors
`_global_ void add(int n, float *x, float *y) { for (int i = 0; i < n; i++) y[i] = x[i] + y[i]; }`

3. Run on GPU
 4. Wait to complete

10/21/20

CS267 Lecture

49

Example: Vector Addition (GPU)

```

// Run kernel on GPU
add<<<1,256>>>(N, d_x, d_y);

```

```

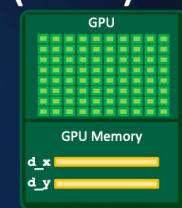
// GPU function to add two vectors
_global_ void add(int n, float *x, float *y) { int index = threadIdx.x; y[index] = x[index] + y[index]; }

```

CUDA's get thread id

1 block of 256 threads

Works iff N = 256!!



10/21/20

CS267 Lecture

54

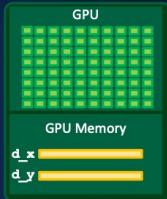
Example: Vector Addition (GPU)

```
// Run kernel on GPU
add<<<1,256>>>(N, d_x, d_y);
```

```
// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i+=stride)
        y[i] = x[i] + y[i];
}
```

1 block of
stride threads

CUDA's # threads



10/21/20

CS267 Lecture

55

Example: Vector Addition (GPU)

```
// Run kernel on GPU
add<<<1,256>>>(N, d_x, d_y);
```

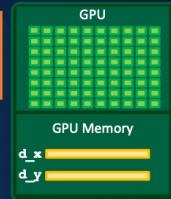
Threads (256 here)
must be ≤ 1024 on
Volta (previously 512)

```
// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
    int index = threadIdx.x;
    int stride = blockDim.x;
    for (int i = index; i < n; i+=stride)
        y[i] = x[i] + y[i];
}
```

1 block of
stride threads

CUDA's # threads

Works for arbitrary N and #
threads / block, only one block



10/21/20

CS267 Lecture

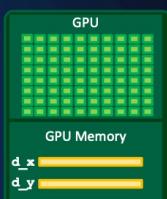
56

Example: Vector Addition

```
// Run kernel on GPU
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

```
// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    int stride = gridDim.x;
    for (int i = index; i < n; i+=stride)
        y[i] = x[i] + y[i];
}
```

Works for arbitrary N
and # threads / block



10/21/20

CS267 Lecture

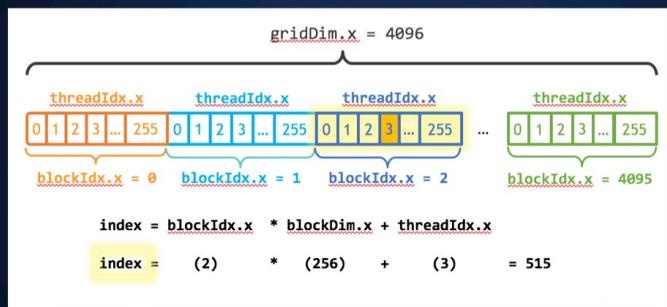
57

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6	7
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	2551
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB

10/21/ https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

58

Blocks and Threads (and Grids)



<https://developer.nvidia.com/blog/even-easier-introduction-cuda/>

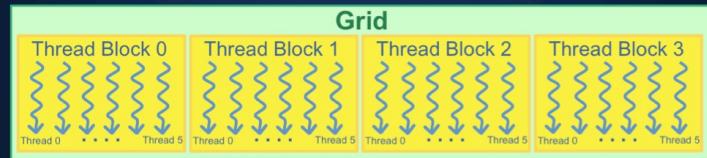
10/21/20

CS267 Lecture

59

Grids and Thread Blocks

- A 1D Grid of 1D Blocks



- $\text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$

<https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf>

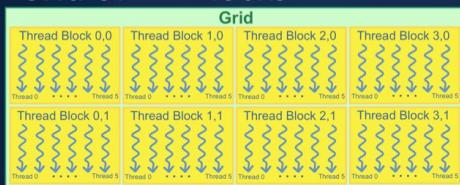
10/21/20

CS267 Lecture

60

Grids and Thread Blocks

- A 2D Grid of 1D Blocks



```
int blockIdx = blockIdx.y * gridDim.x + blockIdx.x;  
int threadId = blockIdx * blockDim.x + threadIdx.x;
```

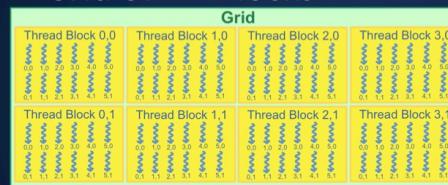
10/21/20

CS267 Lecture

61

Grids and Thread Blocks

- A 2D Grid of 2D Blocks

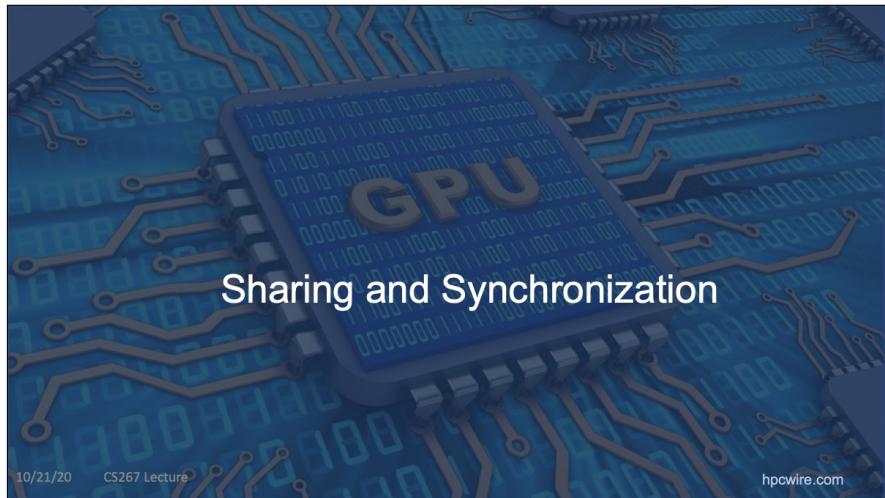


```
int blockIdx = blockIdx.y * gridDim.x + blockIdx.x;  
int threadId = blockIdx * (blockDim.x * blockDim.y)  
              + (threadIdx.y * blockDim.x) + threadIdx.x;
```

10/21/20

CS267 Lecture

62

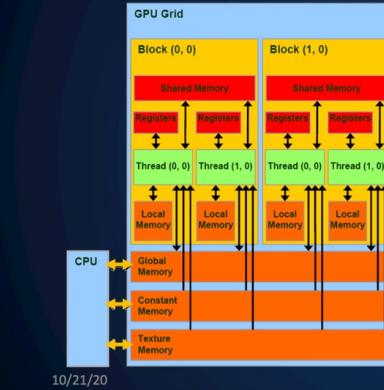


10/21/20

CS267 Lecture

hpcwire.com

Memory types on NVIDIA GPUs



10/21/20

CS267 Lecture

64

- Registers per thread
- Local cached memory per thread
- Shared memory (shared in block)
- Global device level shared
- Constant cache shared by threads
- Texture cache shared by all block

Cost: local/global
100x slower

Hierarchical Parallelism Strategy

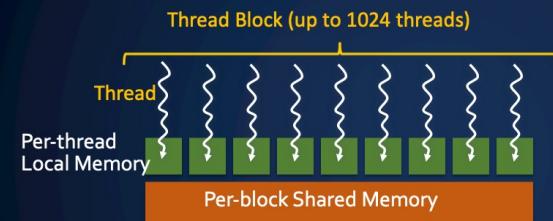
- Use both blocks and threads – Why?
 - `kernel_call<<<blocks,threads>>>`
 - Limit on maximum number of threads/block
 - Threads alone won't work for large arrays
 - Fast shared memory only between threads
 - Blocks alone are slower

10/21/20

CS267 Lecture

65

Shared (within a block) Memory



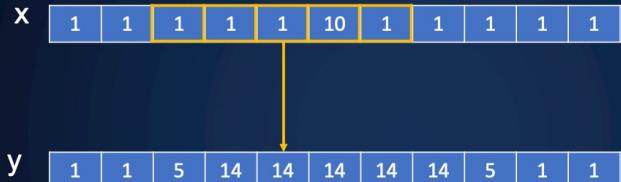
- Declare using `__shared__`, allocated per block
- Fast on-chip memory, user-managed
- Not visible to threads in other blocks

10/21/20

CS267 Lecture

66

1D Stencil Example



1D 5-point stencil (with a “radius” of 2)

$$y[i] = x[i] + x[i-2] + x[i-1] + x[i+2] + x[i+1]$$

10/21/20

CS267 Lecture

67

1D Stencil Example



- Each thread processes one output element
 - blockDim.x elements per block
- Input elements are read several times
- Reuse of inputs:
 - Radius of 2, each input element is read 5 times
 - Radius of 3, each input element is read 7 times

10/21/20

CS267 Lecture

68

output

GPU thread strategy

Divide output array into blocks, each assigned to a thread block

- Each element within is assigned to a thread **“owner computes”**
- Compute blockDim.x output elements
- Write blockDim.x output elements to global memory

What about the input?

10/21/20

CS267 Lecture

69

temp

output

GPU thread strategy



Shared mem
copy of input

Cache (manually) input data in shared memory

- Have each block read (blockDim.x + 2 * radius) input elements from global memory to shared memory
- Each block needs a **halo** of radius elements at each boundary
- (halos are also called **ghost regions**)

10/21/20

CS267 Lecture

70



10/21/20

CS267 Lecture

71

Stencil Kernel

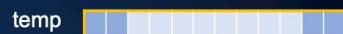
```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) { // fill in halos
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // to be continued
}
```

10/21/20

CS267 Lecture

72

Stencil Kernel



```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) { // fill in halos
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // to be continued
}
```

10/21/20

CS267 Lecture

73

Stencil Kernel



```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) { // fill in halos
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // to be continued
}
```

10/21/20

CS267 Lecture

74

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {  
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];  
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;  
    int lindex = threadIdx.x + RADIUS;  
    // Read input elements into shared memory  
    temp[lindex] = in[gindex];  
    if (threadIdx.x < RADIUS) { // fill in halos  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
    }  
    // to be continued
```

10/21/20

CS267 Lecture

75



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {  
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];  
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;  
    int lindex = threadIdx.x + RADIUS;  
    // Read input elements into shared memory  
    temp[lindex] = in[gindex];  
    if (threadIdx.x < RADIUS) { // fill in halos  
        temp[lindex - RADIUS] = in[gindex - RADIUS];  
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
    }  
    // to be continued
```

10/21/20

CS267 Lecture

76



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {  
    // code from previous slide...  
  
    // Apply the stencil  
    int result = 0;  
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)  
        result += temp[lindex + offset];  
    // Store the result  
    out[gindex] = result;  
}
```

10/21/20

CS267 Lecture

77



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {  
    // code from previous slide...  
  
    // Apply the stencil  
    int result = 0;  
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)  
        result += temp[lindex + offset];  
    // Store the result  
    out[gindex] = result;  
}
```

10/21/20

CS267 Lecture

78



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {  
    // code from previous slide...  
  
    // Apply the stencil  
    int result = 0;  
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)  
        result += temp[lindex + offset];  
    // Store the result  
    out[gindex] = result;  
}
```

10/21/20

CS267 Lecture

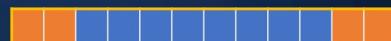
79



Problem: Race Condition!

Suppose thread 7 (of 8) reads the halo before thread 0 has filled it in.

```
temp[lindex] = in[gindex];  
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
int result = 0;  
result += temp[lindex + 1];
```



10/21/20

CS267 Lecture

80

Problem: Race Condition!

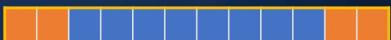
Suppose thread 7 (of 8) reads the halo before thread 0 has filled it in.

```
temp[lindex] = in[gindex];  
if (threadIdx.x < RADIUS) {  
    temp[lindex - RADIUS] = in[gindex - RADIUS];  
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];  
}  
int result = 0;  
result += temp[lindex + 1];  
T0 writes temp[2]  
T0 writes temp[0]  
T7 reads temp[11]  
T0 writes temp[11]
```

10/21/20

CS267 Lecture

81



Thread Synchronization

- Synchronizes all threads within a block
`void __syncthreads();`
- Used to prevent RAW / WAR / WAW hazards
- All threads in the block must reach the barrier
- If used inside a conditional, the condition must be uniform across the block

10/21/20

CS267 Lecture

82

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    // code from earlier slide to setup temp halos...
    // Synchronize (ensure all the data is available)
    __syncthreads();
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];
    // Store the result
    out[gindex] = result;
}
```

10/21/20

CS267 Lecture

83

Synchronization

- Threads within a block may synchronize with **barriers**
... Step 1 ...
`__syncthreads();`
... Step 2 ...
- Blocks **coordinate** via atomic memory operations
 - e.g., increment shared pointer with `atomicInc()`
 - Or use cooperative thread groups
- Implicit barrier between **kernels**

```
vec_minus<<<nblocks, blksize>>>(a, b, c);
vec_dot<<<nblocks, blksize>>>(c, c);
```

10/21/20

CS267 Lecture

84

Blocks must be independent

- Any possible interleaving of blocks should be valid
 - presumed to run to completion without pre-emption (not fairly scheduled)
 - can run in any order
 - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
 - shared queue pointer: **OK**
 - shared lock: **BAD** ... can easily deadlock
- Independence requirement gives **scalability**

10/21/20

CS267 Lecture

85

CUDA: Extensions to C/C++

- Declaration specifiers to indicate where things live
`__global__ void KernelFunc(...);` // kernel callable from host
`__device__ void DeviceFunc(...);` // function callable on device
`__device__ int GlobalVar;` // variable in device memory
`__shared__ int SharedVar;` // in per-block shared memory
- Extend function invocation syntax for parallel kernel launch
`KernelFunc<<<500, 128>>>(...);` // 500 blocks, 128 threads each
- Special variables for thread identification in kernels
`dim3 threadIdx; dim3 blockIdx; dim3 blockDim;`
- Intrinsics that expose specific operations in kernel code
`__syncthreads();` // barrier synchronization

10/21/20

CS267 Lecture

86

CUDA: Features available on GPU

- Double and single precision
- Standard mathematical functions
 - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.
- Atomic memory operations
 - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
- These work on both global and shared memory

10/21/20

CS267 Lecture

87

CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
 - `cudaMalloc()`, `cudaFree()`
 - `cudaMallocManaged()`;
- Explicit memory copy for host \leftrightarrow device, device \leftrightarrow device
 - `cudaMemcpy()`, `cudaMemcpy2D()`, ...

10/21/20

CS267 Lecture

88

Mapping CUDA to Nvidia GPUs

- Threads:
 - Each thread is a SIMD lane (ALU)
- Warps:
 - A warp executed as a logical SIMD instruction (sort of)
 - Warp width is 32 elements: **LOGICAL** SIMD width
 - (Warp-level programming also possible)
- Thread blocks:
 - Each thread block is scheduled onto an SM
 - Peak efficiency requires multiple thread blocks per processor
- Kernel
 - Executes on a GPU (there is also multi-GPU programming)



10/21/20

CS267 Lecture

89

Summary

- GPUs gain efficiency from simpler cores and more parallelism
 - Very wide SIMD (SIMT) for parallel arithmetic and latency-hiding
- Heterogeneous programming with manual offload
 - CPU to run OS, etc. GPU for compute
- Massive (mostly data) parallelism required
 - Not as strict as CPU-SIM (divergent addresses, instructions)
- Threads in block share faster memory and barriers
 - Blocks in kernel share slow device memory and atomics

10/21/20

CS267 Lecture

90