




CUDA Memory Model

A large orange circle is positioned on the left side of the slide, partially cut off by the edge.

Learning Objectives

- **Describe** Different Levels of Nvidia GPU Memory hierarchy
 - **Use** Shared Memory
 - **Examples** GPU programs
 - Histogram
- 
- A series of four yellow dashed line segments are arranged in a curved, upward-sloping path in the bottom right corner of the slide.

A large orange circle is positioned on the left side of the slide, partially cut off by the edge.

Expected Time of the Activity:

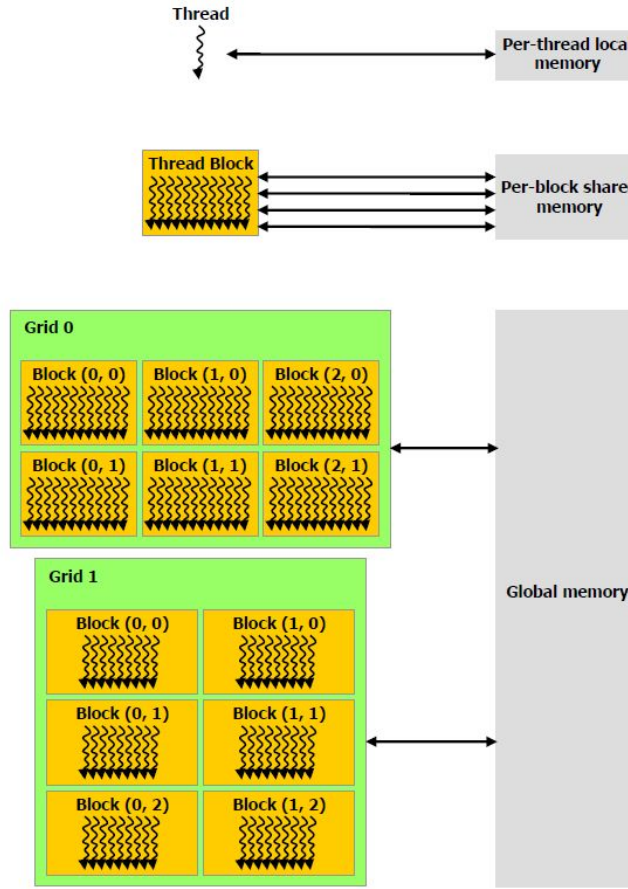
Module Approximate Timing

- Memory Architecture Description: 10 min
- CUDA Histogram 14 min
 - Sequential
 - NO Shared Memory 2 min
 - Shared Memory 10 min
- Compare Sequential vs Parallel CUDA implementation: 1 min

Total time for the module: 25 minutes

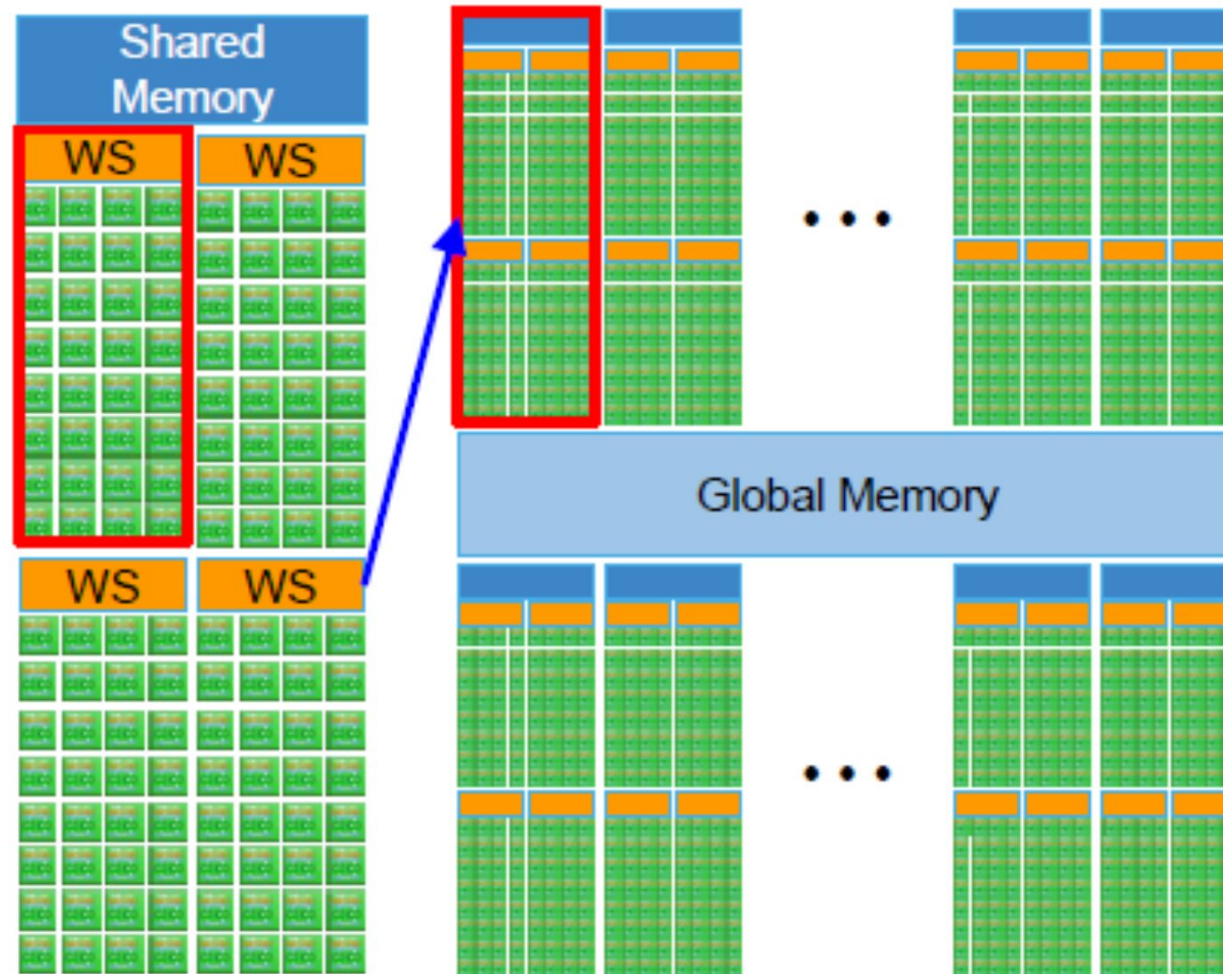
A series of yellow dashed line segments are arranged in a curved path at the bottom right of the slide.

CUDA Memory Hierarchy



From CUDA 10.2 programming model document

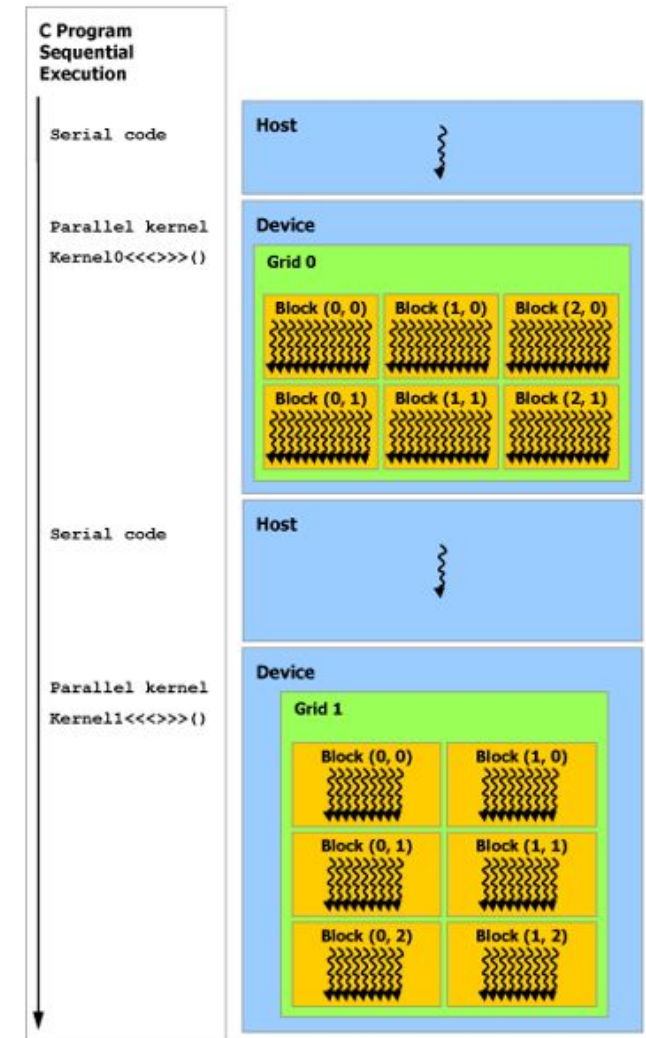
CUDA Memory Hierarchy



From CUDA 10.2 programming model document

CUDA Memory Hierarchy

- The CUDA programming model assumes that the CUDA threads execute on a physically separate *device* that operates as a coprocessor to the *host* running the C/C++ program.
- The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as *host memory* and *device memory*, respectively.
- Unified Memory provides *managed memory* to bridge the host and device memory spaces.



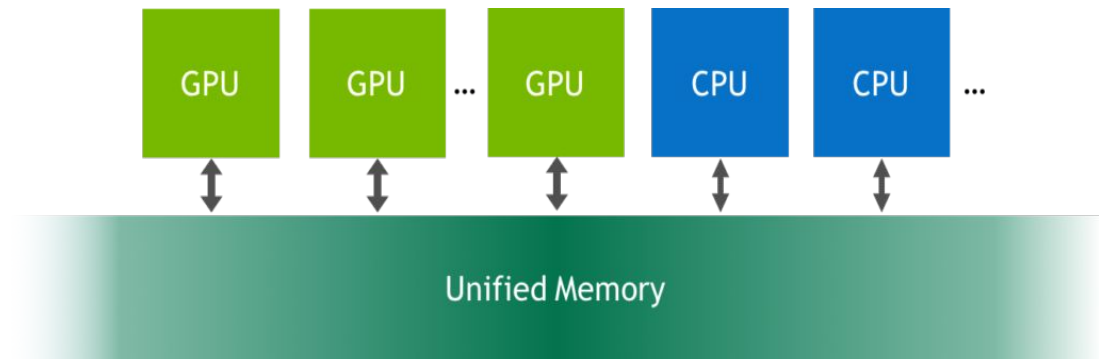
From CUDA 10.2 programming model document

CUDA Unified Memory Model

Managed memory is accessible from all CPUs and GPUs in the system as a single, coherent memory image with a common address space.

Managed memory is accessible to both the CPU and GPU using a single pointer.

The key is that the system automatically *migrates* data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU.



Example: Matrix Addition

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

int main(void){
    float *x, *y;
    // Allocate Unified Memory -- accessible from CPU or GPU
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));

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

    // Launch kernel on 1M elements on the GPU
    int blockSize = 256; int numBlocks = (N + blockSize - 1) / blockSize;
    add<<<numBlocks, blockSize>>>(N, x, y);
    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();

    // Free memory
    cudaFree(x); cudaFree(y);
    return 0;
}
```


Shared Memory

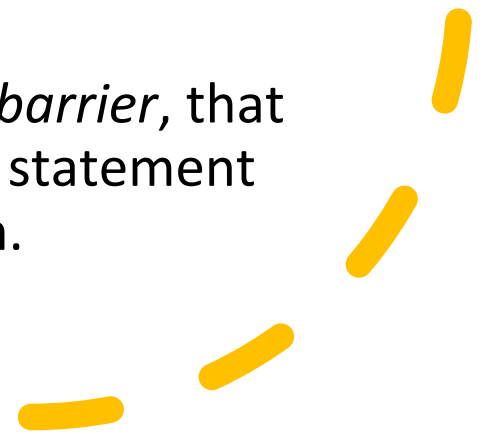
- The RAM used for shared memory can be also used as L1-cache.
- A programmer can specify the balance between cache and shared memory space by using the `cudaFuncSetCacheConfig` function.

Shared Memory

- **Shared memory** provides a fast, on-chip data repository that can be accessed by all the threads in a block.
- Example:

```
__global__ void foo(...) {  
    __shared__ counter[BLOCKSIZE];  
    int myID = threadIdx.x;  
    counter[myID] = 0;  
    __syncthreads ();  
}
```

- The `__syncthreads` is a *block-wide barrier*, that ensures that all warps have reached that statement before the threads can resume execution.



vector dot product Sequential

```
for (int i=0; i<N; i++)  
    c+=a[i]+b[i]
```

$$AB = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \end{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{bmatrix}$$

$$= a_{11}b_{11} + a_{12}b_{21} + \cdots + a_{1n}b_{n1}$$

vector dot product CUDA

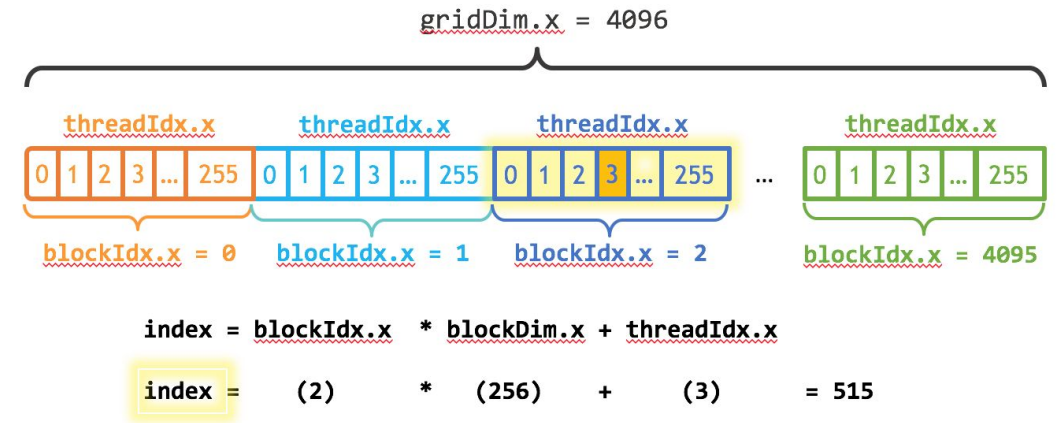
```
__global__ void dot( Lock lock, float *a, float *b, float *c ) {
```

```
    __shared__ float cache[threadsPerBlock];  
    int tid = threadIdx.x + blockIdx.x * blockDim.x;  
    int cacheIndex = threadIdx.x;
```

```
    float temp = 0;  
    while (tid < N) {  
        temp += a[tid] * b[tid];  
        tid += blockDim.x * gridDim.x;  
    }
```

```
    // set the cache values  
    cache[cacheIndex] = temp;
```

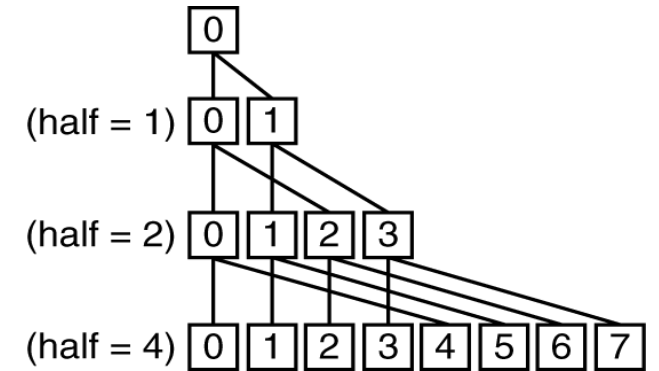
```
    // synchronize threads in this block  
    __syncthreads();
```



Vector Dot Product CUDA

```
// for reductions, threadsPerBlock must be a power of 2  
// because of the following code
```

```
int i = blockDim.x/2;  
while (i != 0) {  
    if (cacheIndex < i)  
        cache[cacheIndex] += cache[cacheIndex + i];  
    __syncthreads();  
    i /= 2;  
}  
  
if (cacheIndex == 0)  
    atomicAdd(&c, cache[0]); //Race Condition accumulates all results from blocks  
}
```



Example: Histogram of ASCII Characters

```
int main(void){
    /*Read a file of ASCII characters if size SIZE*/

    /* histogram will have only 256 different values*/
    unsigned int hist[256];
    for(i=0; i<256;i++)
        hist[i]=0;

    for(i=0;i<SIZE;i++)
        hist[buffer[i]]++;

    /*clean after ourself as always*/
    free(buffer);
}
```

CPU Profiling

Compile : `gcc -g -Wall -O3 -o Hist HistCPU.c`

CPU: Intel i7-7820HQ 8 cores @2.90GHz

gcc -version: 7.5.0

command output:

Elapsed time 0.18 sec (for a file of size
 $100 \cdot 2^{20}$)



Atoms Example: Histogram

```
__global__ void histoNoSharedMem( unsigned char *buffer,
                                   long size,
                                   unsigned int *histo ) {
    // calculate the starting index and the offset to the next
    // block that each thread will be processing
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < size)
        atomicAdd( &histo[buffer[i]], 1 );
}
```

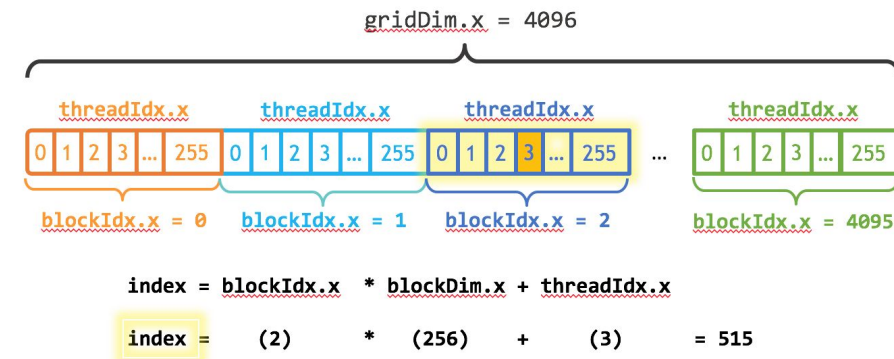
Each thread performs one operation: Read one character from file and increase the corresponding bucket for the character

Since more than one thread may need to update the same histogram bucket we need an atomicAdd to guarantee correctness

Execution time: 0.10 Sec

Atoms Example: Histogram

```
__global__ void histoNoSharedStride( unsigned char *buffer,
                                     long size,
                                     unsigned int *histo ) {
    // calculate the starting index and the offset to the next
    // block that each thread will be processing
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x*gridDim.x;
    while (i < size){
        atomicAdd( &histo[buffer[i]], 1 );
        i+=stride;
    }
}
```



Each thread performs x operations, where $x = \text{size} / (\text{numberBlocks} * \text{numberThreadsperBlock})$. The operation is: Read one character from file and increase the corresponding bucket for the character

Since more than one thread may need to update the same bucket we need an atomicAdd to guarantee correctness

Execution time: 0.10 Sec NO improvement but it doesn't depend on the size of N so will not hit a HW limit

Histogram Shared Memory

```
__global__ void histo_kernel( unsigned char *buffer,
                               long size, unsigned int *histo ) {

    // clear out the accumulation buffer temp
    __shared__ unsigned int temp[256];
    temp[threadIdx.x] = 0;
    __syncthreads();
    // calculate the starting index and the offset to the next
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &temp[buffer[i]], 1 );//atomic on shared mem
        i += stride;
    }
    // sync the data from the above writes to shared memory
    __syncthreads();
    //256 threads and 256 bins
    atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
}
```

GPU Profiling with Share Memory

Compile : gcc -g -Wall -o3 -o Hist HistCPU.c

CPU: Intel i7-7820HQ 8 cores @2.90GHz

gcc -version: 7.5.0

GPU Nvidia: Quadro M12

Nvidia driver: 396.26

NVCC compiler CUDA 9.2

Compile : nvcc HistGPU.cu

command output:

Elapsed time 0.07 sec (for a file of size
 100×2^{20})

Module Evaluation

STUDENT ACTIVITY

Change the histogram to:

- Histogram with number of buckets bigger than the max number of threads per block allowed
- Histogram of words

