Advanced Topics in CUDA

Sudsanguan Ngamsuriyaroj
Ekasit Kijsipongse
Ittipon Rassameeroj

Semester 1/2022

Topics

- Sorting in CUDA
- CUDA reduction
- Error checking on CUDA operations
- CUDA Memory

Rank Sort

- Number of numbers that are smaller than each selected number counted. This count provides the position of selected number in sorted list; that is, its "rank."

```
for (i = 0; i < n; i++) {          /* for each number */
   x = 0;
   for (j = 0; j < n; j++)          /* count number less than it */
         if (a[i] > a[j]) x++;
   b[x] = a[i];                     /* copy number into correct place */
}
```

- Assume no duplicated numbers.
- Overall sequential sorting time complexity of $O(n^2)$.
- Not a good sequential sorting algorithm!

Parallel Rank Sort Using n Threads

- In parallel version, one thread allocated to each number. Finds final index in O(n) steps.
- With all thread operating in parallel, parallel time complexity O(n), given n threads.

```
For each thread i,  do counting for a[i] in parallel
    x = 0;
    for (j = 0; j < n; j++)        /* count number less than it */
        if (a[i] > a[j]) x++;
    b[x] = a[i];                   /* copy no into correct place */
```

- Where i is the global ID = blockIdx.x*blockDim.x + threadIdx.x

CUDA Rank Sort When n = T (small n)

```
__global__ void ranksort(int *dA, int *dB) {
    int i = threadIdx.x;      // one thread block
    int self = dA[i];
    int x = 0;
    for (int j=0; j < n; j++) {
        if (self > dA[j])
            x++;
    }
    dB[x] = self;
}

...
ranksort<<<1,T>>>(dA,dB);    // one thread block
...
```

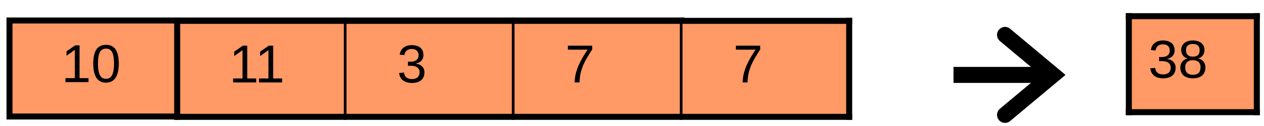
CUDA Rank Sort (n/T is an integer)

```
#include<stdio.h>

#define n 64
#define T 16

__global__ void ranksort(int *dA, int *dB) {
  int i,j, x;
  i = blockIdx.x*blockDim.x + threadIdx.x;
  int self = dA[i];

  x = 0;
  for (j=0; j < n; j++) {
     if (self > dA[j])
        x++;
  }
  dB[x] = self;
}

int main() {
    int A[n], *dA, *dB;
    int i;
    int size = n * sizeof(int);
```

```
    //fill the host_array randomly (no duplicated)
    for(i = 0; i < n; i++)
        A[i] = n-i;

    cudaMalloc( (void**)&dA,size );
    cudaMalloc( (void**)&dB,size );

    cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice);

    dim3 dimBlock(T);
    dim3 dimGrid(n/T );
    ranksort<<<  dimGrid , dimBlock  >>>(dA,dB);

    cudaMemcpy(A,dB,size,cudaMemcpyDeviceToHost);

    cudaFree(dA);
    cudaFree(dB);

    //printf the result after sorting
    for(i = 0; i < n; i++) {
         printf("%d ",A[i]);
    }
    printf("\n");

}
```

Reduction Operation

- Reduce all of the data in an array to a single value that contains some information from the entire array.
  - Sum, maximum element, minimum element, etc.

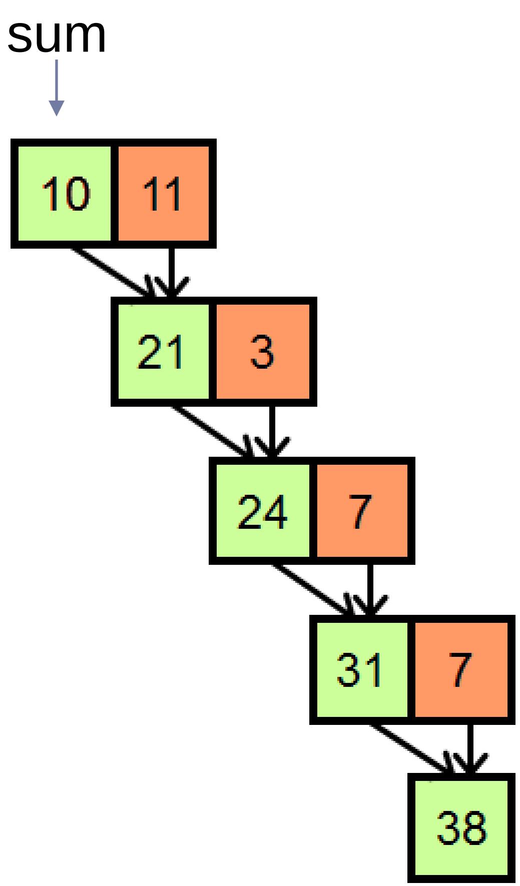


- Useful primitive used in lots of applications

Sequential Reduction

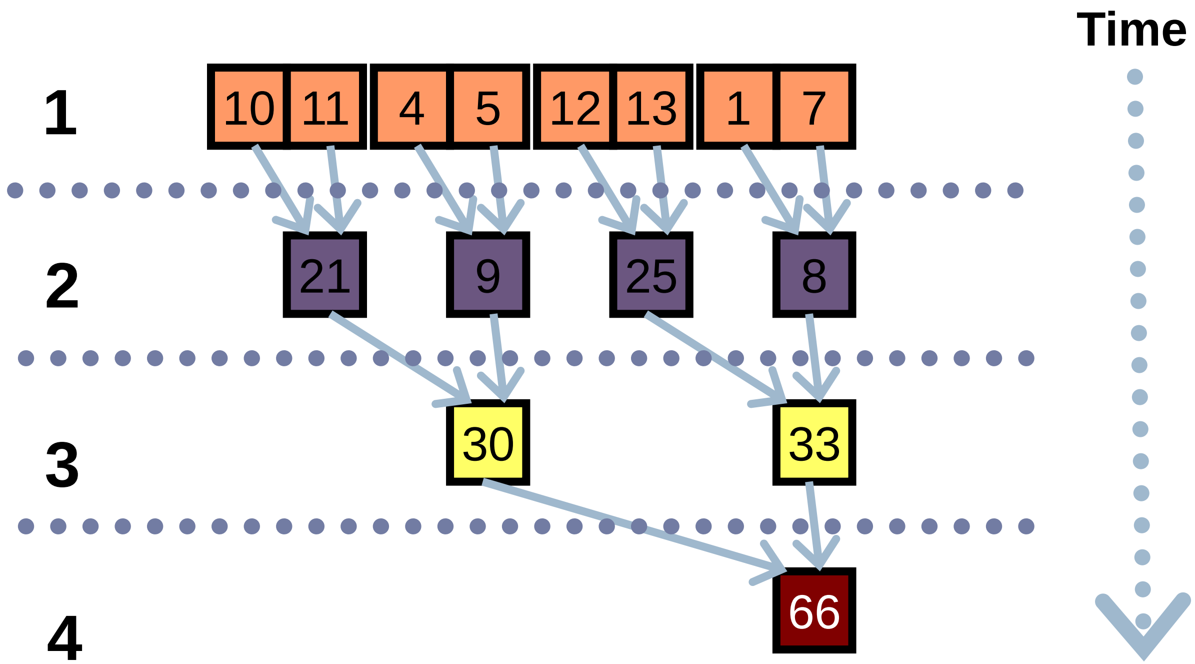
- Start with the first element --> partial result
- Process the next element
- O(*N*)

```
int sum = data[0];
for (i = 1; i < N; i++) {
      sum = sum + data[i];
}
```

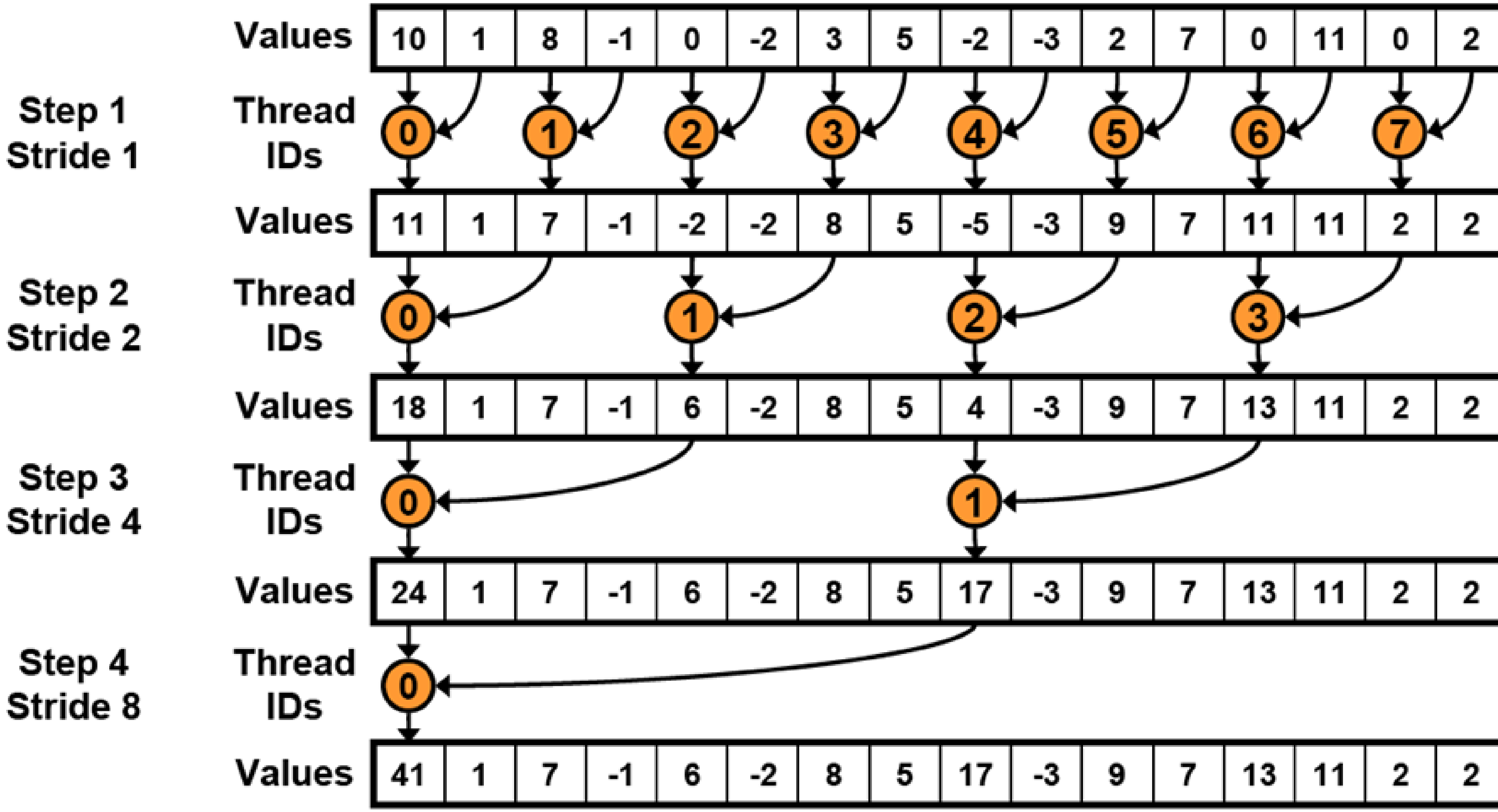


Parallel Reduction

- Pair-wise reduction in steps – Tree-like structure
- $\log_2 N$ steps
- Assume that the data size is a power of 2 and the operator used in the reduction is associative, e.g. +, *



Reduction in CUDA



Simple CUDA Reduction

```
#define n 1024      // power of 2
#define T 256      // n/T must be an integer

__global__ void reduction(int *data, int stride) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    int idx = 2*stride*tid;
    if (idx < n) {
       data[idx] = data[idx]+data[idx+stride];
    }
}

int main () {
    int size = n *sizeof(int);
    int a[n], sum = 0;
    int *dA;

    /* Put random numbers in a[] */

    cudaMalloc( (void**)&dA, size);
    cudaMemcpy( dA, a, size, cudaMemcpyHostToDevice);
```

Simple CUDA Reduction

```
    for (int s=1; s < n; s = s*2) {
        reduction<<<n/T, T>>>(dA,s);
    }

    cudaMemcpy(&sum, dA, sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(dA);

    printf("%d\n",sum);

}
```

The first element in dA contains sum.
Other elements are partial sum.

Error Checking

```
cudaError_t error;
…
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

…
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
error = cudaGetLastError();
if (error != cudaSuccess) {
    printf("CUDA Error: %s\n", cudaGetErrorString(error));
    return 1;
}
…
```

Read-modify-write problem in parallel computation

- Multiple customers booking air tickets
- Each
  - Brings up a flight seat map
  - Decides on a seat
  - Update the seat map, mark the seat as taken

- A bad outcome
  - Multiple passengers ended up booking the same seat

Race Condition in Concurrent Threads

- Threads can access (read/write) shared memory.
- Consider two threads each of which is to add one to a shared data item, x. If x was initially 0, what would the value of x be after threads 1 and 2 have completed?
- Suppose that Old and New are registers, and Mem[x] = 0 initially

	Instruction	Thread 1	Thread 2
Time	`x = x + 1;`	Old ← Mem[x]	Old ← Mem[x]
		New ← Old + 1	New ← Old + 1
		Mem[x] ← New	Mem[x] ← New

Timing Scenario #1

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence

Timing Scenario #2

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence

Timing Scenario #3

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Timing Scenario #4

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Need Mechanism To Ensure Good Outcomes

Time

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Or

Time

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Synchronization

- Synchronization is the coordination of threads/processes to operate a system in unison
- Mutual exclusion is a way to ensure only one thread/process accesses a particular resource at a time.
  - Lock and atomic operations are mutual exclusion implementations
- The sections of code, called critical sections, must not be concurrently executed by more than one thread.
- Barrier synchronizes multiple threads so that any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.
- This concept also appears in operating systems.

Atomic Operations

- Performed by a single instruction on a memory location address
  - Read the old value, calculate a new value, and write the new value to the location
- The hardware ensures that no other threads can access the location until the atomic operation is complete
  - Any other threads that access the location will typically be held in a queue until its turn
  - All threads perform the atomic operation serially
- The advantage of atomic operations is that they are relatively quick compared to locks

AtomicAdd

- CUDA provides different types of atomic operations
  - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
- Atomic Add

atomicAdd(&x, 10) => Lock x
x = x + 10
Unlock x

int atomicAdd(int **address**, int **val**);*

reads the 32-bit word **old** pointed to by **address** in global or shared memory, computes **(old + val)**, and stores the result back to memory at the same address. The function returns **old**.

- Single-precision floating-point atomic add

float atomicAdd(float* address, float val);

Compiler Options for Atomic Operations

- In emulation mode only, use option **sm_11** to enable atomic operations

nvcc **-deviceemu -arch=sm_11** -o *progname progname*.cu

- In real GPU mode, atomic operations have already been enabled by default in the current NVCC compiler

A Simple AtomicAdd() Example

```
__device__ int sum = 0;   // see note below

__global__ void addAll(int *data) {
    int localVal = data[blockIdx.x * blockDim.x + threadIdx.x];
    atomicAdd(&sum, localVal);
}

int main() {
    int *dA,result;
    …
    addAll<<<n/T, T>>>(dA);
    …
}
```

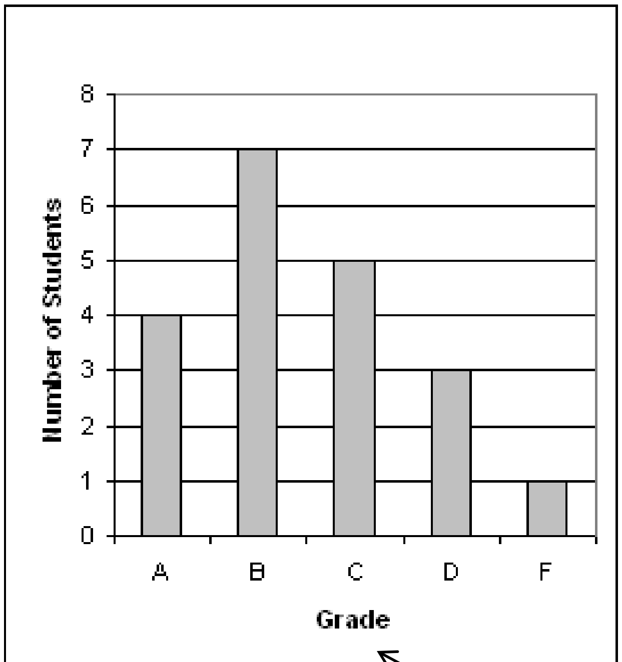
Low performance since all threads sequentially add numbers to sum!!

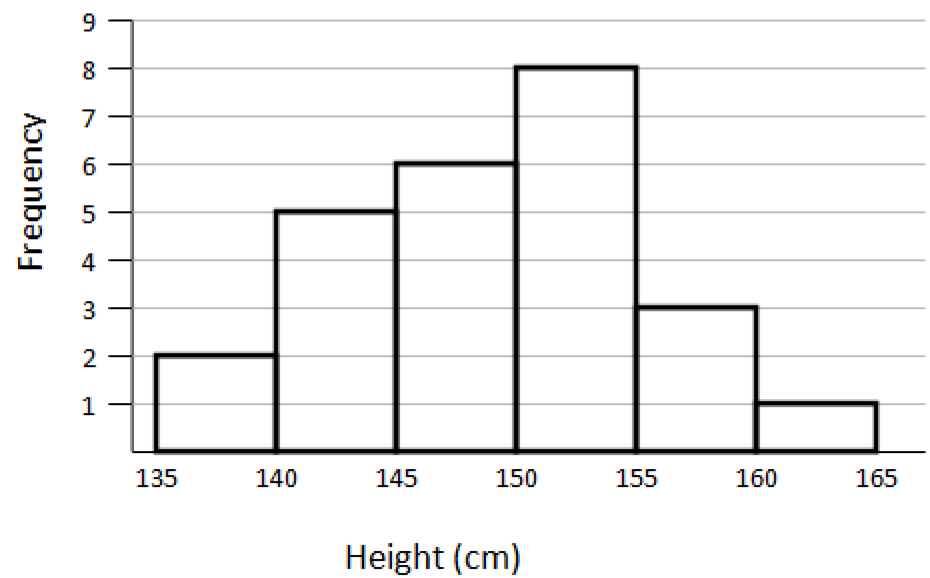
Use reduction operation instead.

To read sum, use cudaMemcpyFromSymbol(&result, “sum", sizeof(int), 0, cudaMemcpyDeviceToHost);

Case Study: Histogram

- Histogram is a representation showing the frequency distribution of data, e.g. student grade or height

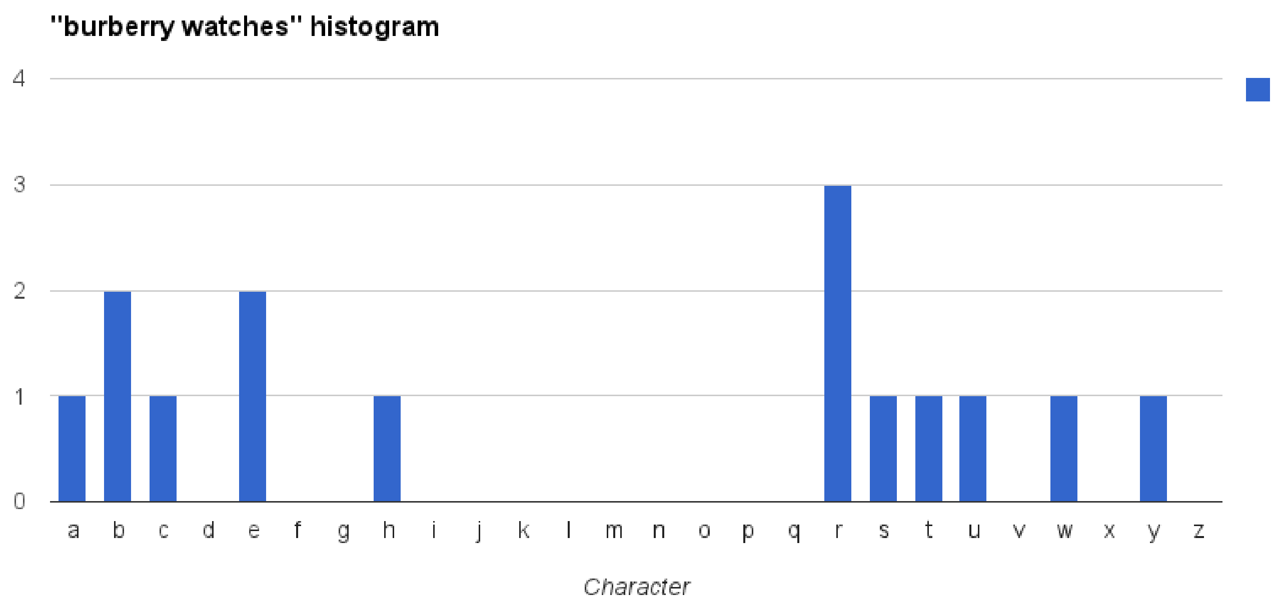




Bin (could be interval)

Example: Frequency of Letter

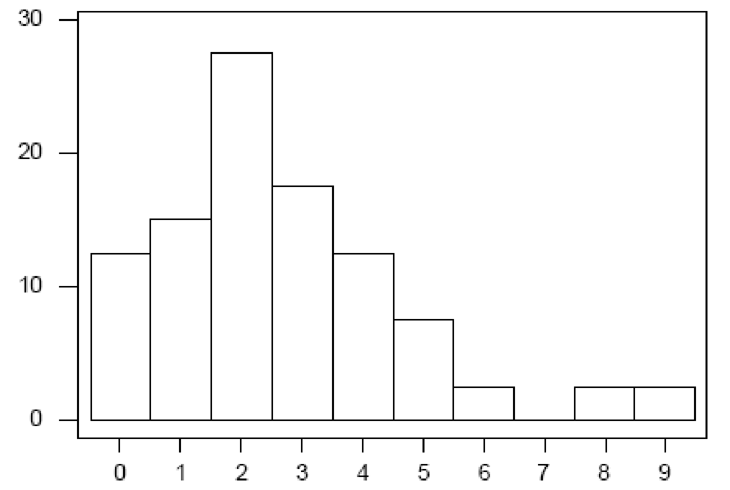
- In sentence "burberry watches" build a histogram of frequencies of each letter
  - freq(a) = 1, freq(b) = 2, freq(c) = 1, …



Histogram: Sequential Code

- For each element in the data set, use the value to identify a "bin" to increment

```
// assume an array of student scores from 0 to 9
int data[DATA_SIZE] = {2, 1, 0, 6, 2, 1, 4, 0, …} ;
// Counters for 10 different scores
int bin[10];

for(int i = 0; i < BIN_SIZE; i++)
    bin[i] = 0;
for(int i = 0; i < DATA_SIZE; i++)
    bin[data[i]]++;
```



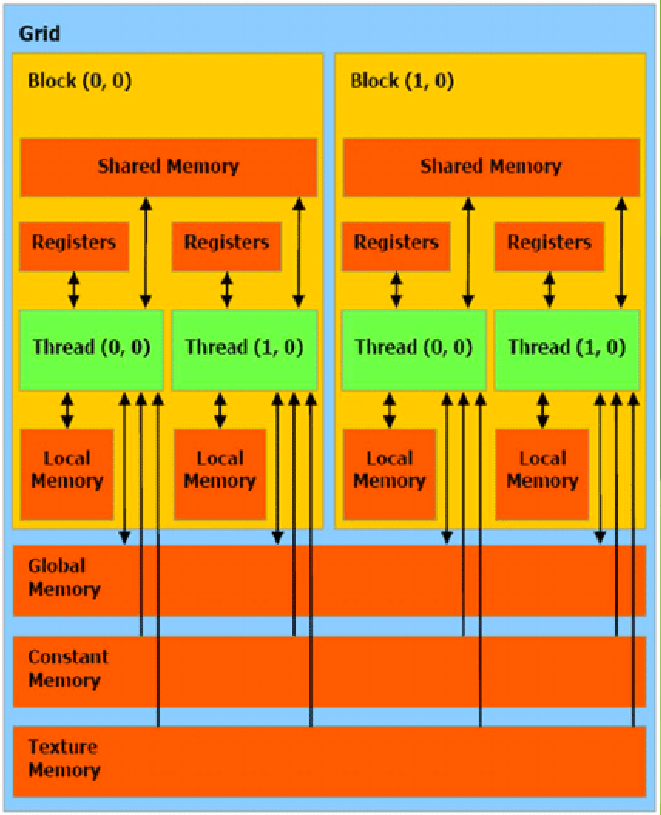
histogram.cu

```
#include <stdio.h>

#define n 1024
#define NUMTHREADS 256

__global__ void histogram_kernel( unsigned int *data, unsigned int *bin)  {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        atomicAdd( &(bin[data[i]]), 1 );
    }
}
```

```
int main (int argc, char *argv[] ) {

    int i;
    int size = n *sizeof(int);
    unsigned int a[n];
    unsigned int bin[10];
    unsigned int *dA, *dBin;

    for (i=0; i < n; i++) {
        a[i] = i % 10;
    }
    cudaMalloc( (void**)&dA, size);
    cudaMalloc( (void**)&dBin, 10*sizeof(int));

    cudaMemcpy( dA, a, size, cudaMemcpyHostToDevice);
    cudaMemset( dBin,0, 10*sizeof(int));

    int nblocks = (n+NUMTHREADS-1)/NUMTHREADS;
    histogram_kernel<<<nblocks, NUMTHREADS>>>(dA,dBin);

    cudaMemcpy(bin, dBin, 10*sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree( dA); cudaFree( dBin);

    int count = 0;
    for (i=0; i < 10; i++) {
      printf("Freq %d = %d\n",i,bin[i]);
      count = count + bin[i];
    }
    printf("#elements = %d\n",count);
}
```

CUDA Memories

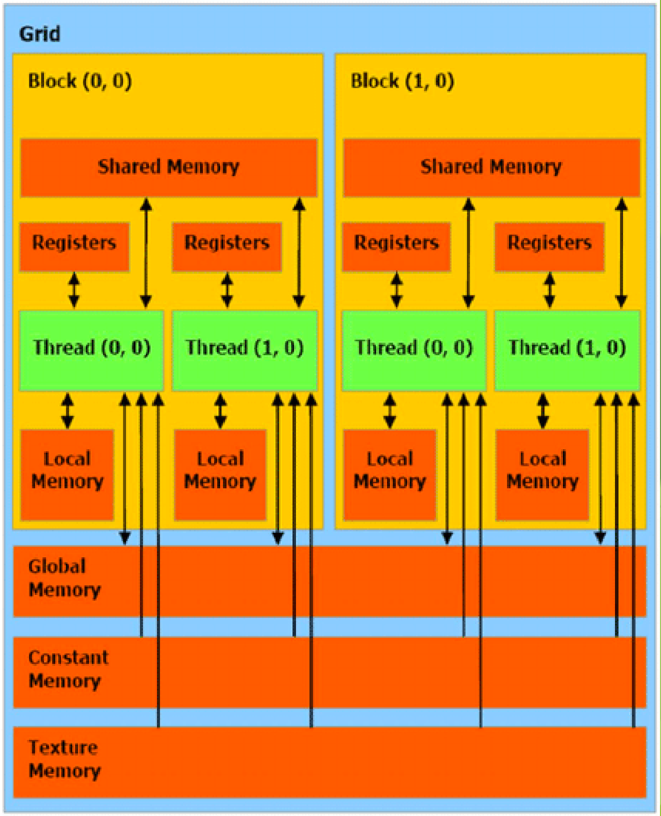
- GPUs have multiple memory spaces
- How to make the best use of the GPU memory system?
- How to deal with hardware limitation?



CUDA Memory Hierarchy

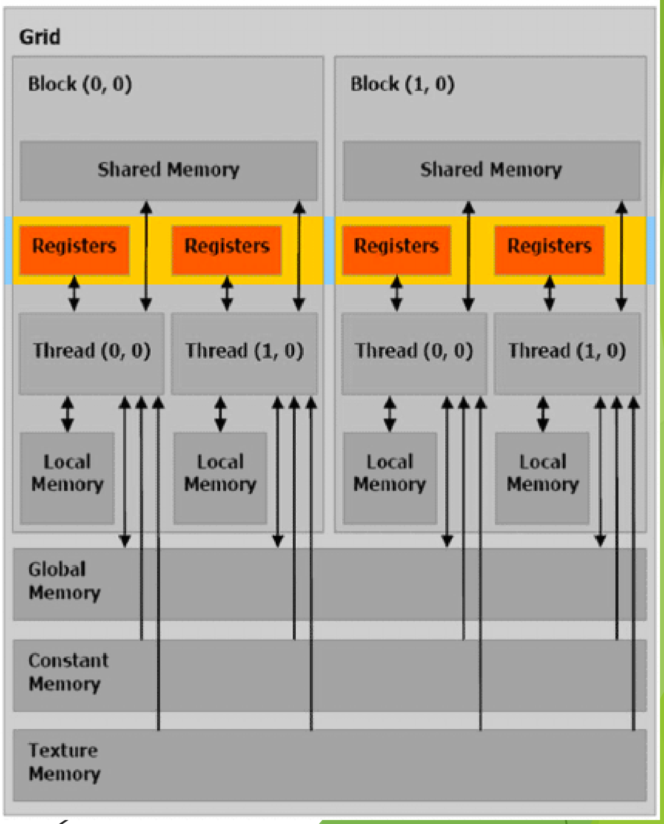
- CUDA threads may access data from multiple memory spaces
  - On-chip memories => inside SM
  - Off-chip memories => device memory

Access Level	Memory Type	Location
Thread-private	register	on-chip
	local Memory	off-chip
Thread block	shared memory	on-chip
Grid	global memory	off-chip
	constant memory	
	texture memory	



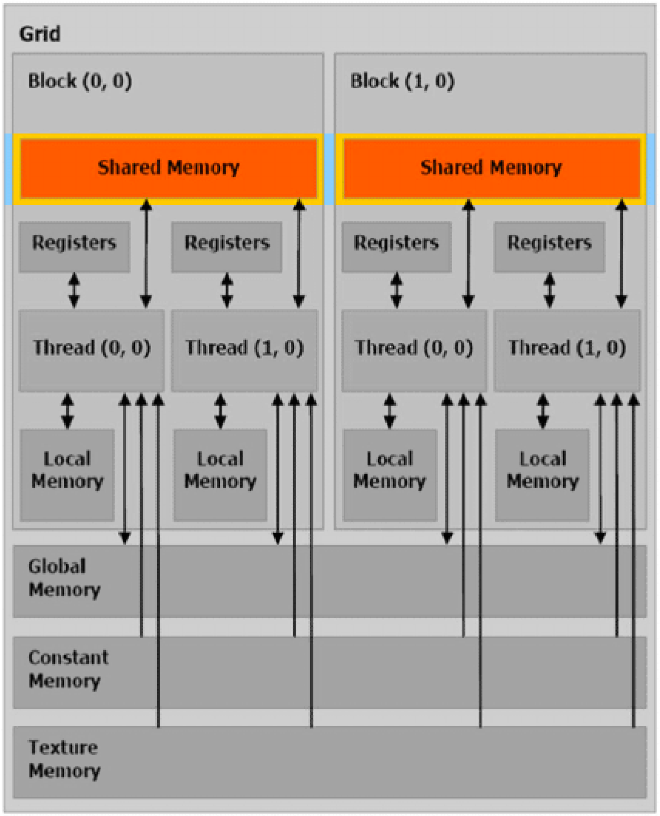
Registers

- Fastest memory
- Only accessible by a thread
- Lifetime of a thread
- Automatically allocated
  - Typically for scalar variables
- Number of registers is very limited
- Resides in register files
  - Shared across all active threads in an SM
- Cannot access by host



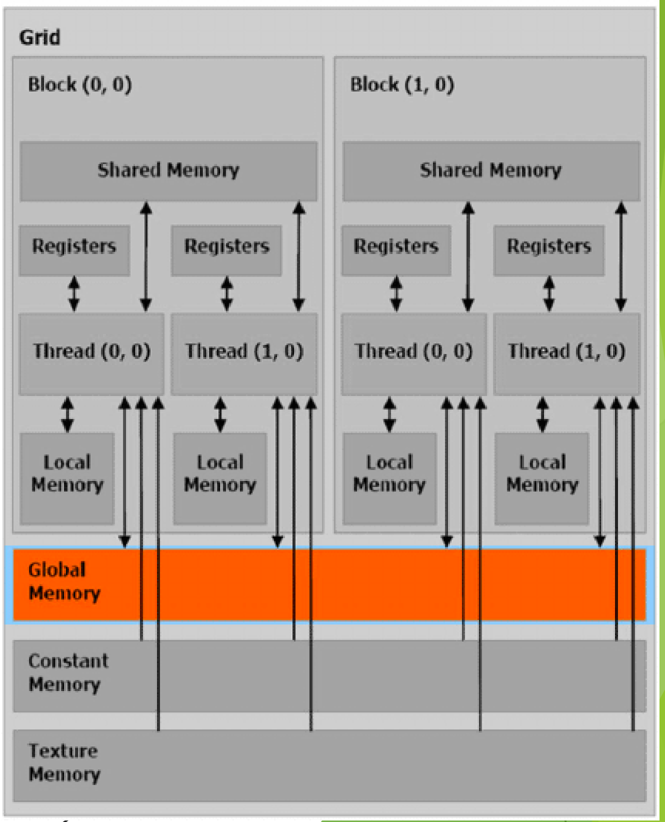
Shared Memory

- Extremely fast
- Shared across threads in the **same thread block**
- Lifetime of a kernel
- User-managed
  - Programmers must explicitly allocate shared memory in the kernel
  - __shared__ specifier
  - Will discuss later
- Cannot access by host



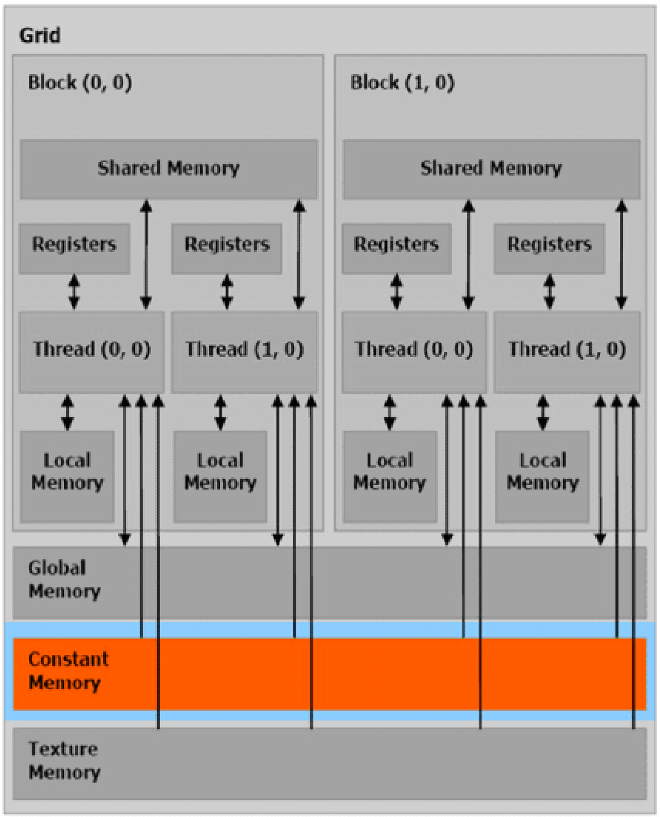
Global Memory

- Also called **"device memory"**
- Accessible by **all threads**
- Lifetime of a program
- Very high access latency
  - 400-800 clock cycles
- Potential of traffic congestion
- Can be allocated and modified by host



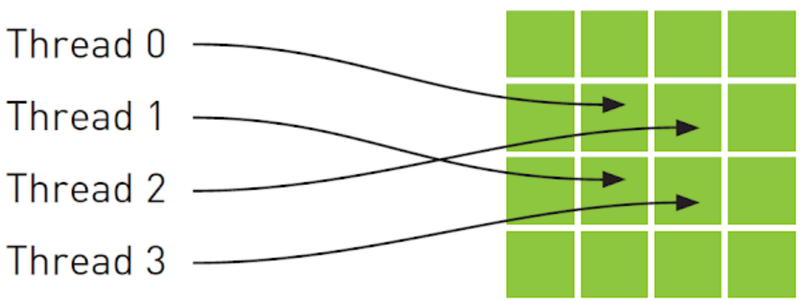
Constant Memory

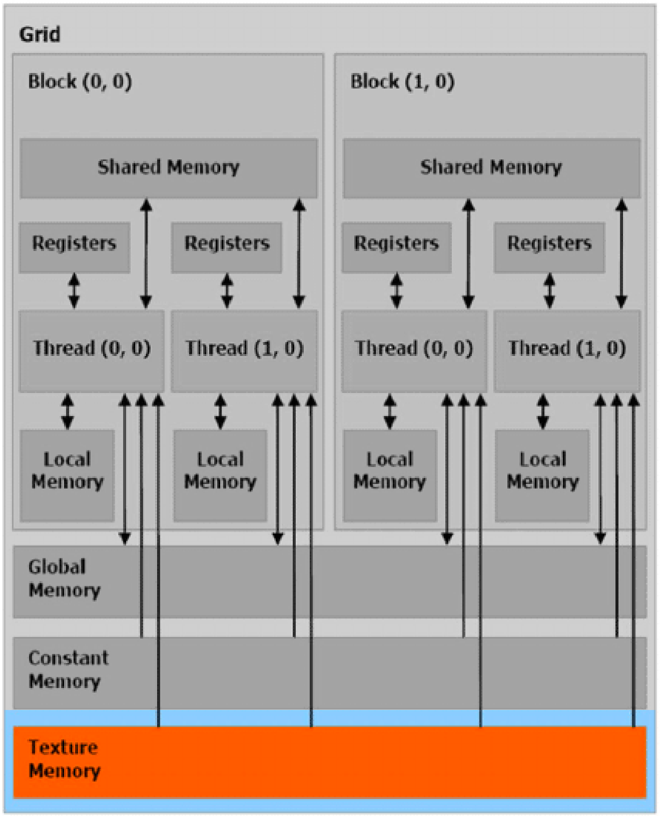
- Special type of global memory
- Read-only
- Cached
- Short latency and high bandwidth when all threads access the same location



Texture Memory

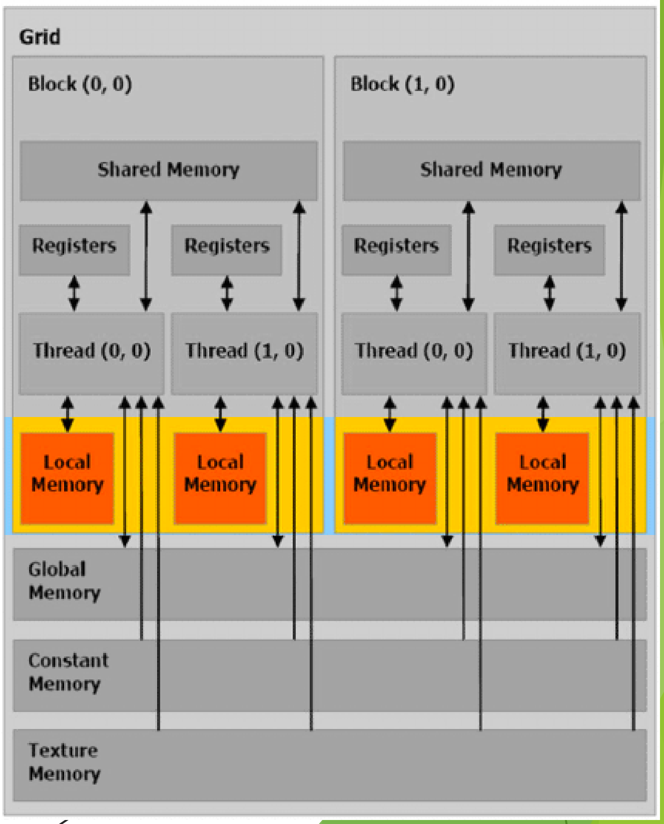
- Special type of global memory
- Read-only
- Cached
- Optimized for 1D, 2D, or 3D spatial locality in graphic programs
  - Threads read memory is nearby location





Local Memory

- Not an actual memory space
- Similar to register
  - Only accessible by **a thread**
  - Lifetime of a thread
  - Automatically allocated
- Resides in **global memory**
- Used for
  - Storing arrays declared inside a kernel
  - Register spilling



Reducing Global Memory Traffic

- Global memory is slow
  - Can be **performance bottleneck**.
- Reducing global memory access enhances performance
- Using the memory space with caching could improve performance
  - Constant value => constant memory
  - 2D/3D memory accesses => texture memory
- What else?

Reducing Global Memory Traffic using Shared Memory

- Extremely fast, user-managed, on-chip memory.
  - ~100x faster than global memory
- Could be used as user-managed cache
  - Reduce global memory traffic
- Shared across all threads in a thread block.
  - Data could be loaded from global memory **once** and shares across all threads in the same block
  - Data will stay in the shared memory as long as
    - The thread block ends its executions
    - Programmer explicitly replace it
- Declare using `__shared__` specifier

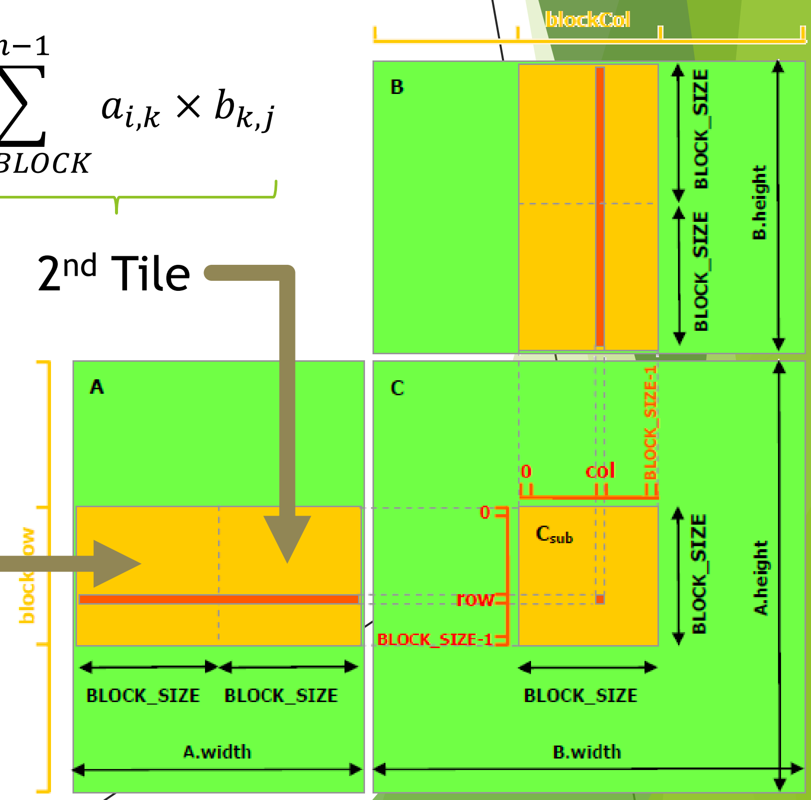
Back to Matrix Multiplication

- **Strategy**: partition the data into subsets called “tiles”, such that each tile fits into the shared memory.
  - Data loaded inside a tile could be reused
  - Make threads that use common elements collaborate
- Each thread can load a submatrix into the shared memory before calculations
- The submatrix will be used by the thread that loaded them and other threads that share them

Matrix Multiplication with Shared Memory

- Looking at matrix multiplication equation

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} \times b_{k,j} = \underbrace{\sum_{k=0}^{BLOCK-1} a_{i,k} \times b_{k,j}}_{1^{\text{st}}\ \text{Tile}} + \underbrace{\sum_{k=BLOCK}^{n-1} a_{i,k} \times b_{k,j}}_{2^{\text{nd}}\ \text{Tile}}$$



Matrix Multiplication with Shared Memory

```
__global__ void matrixMul(float* A, float* B, float* C, int width)
{
      __shared__ float As[TILE_WIDTH] [TILE_WIDTH];
      __shared__ float Bs[TILE_WIDTH] [TILE_WIDTH];

      int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
      int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

      float c_val = 0.0f;

      for(int i = 0; i < width/TILE_WIDTH; i++){
            As[threadIdx.y][threadIdx.x] = A[row * width + (i * TILE_WIDTH + threadIdx.x)];
            Bs[threadIdx.y][threadIdx.x] = B[(i * TILE_WIDTH + threadIdx.y) * width + col ];
            __syncthreads();

            for(int k = 0; k < TILE_WIDTH; k++)
                  c_val += As[threadIdx.y][k] * Bs[k][threadIdx.x];
            __syncthreads();
      }
      C[row * width + col] = c_val;
}
```

Matrix Multiplication with Shared Memory

```
__global__ void matrixMul(float* A, float* B, float* C, int width)
{
      __shared__ float As[TILE_WIDTH] [TILE_WIDTH];
      __shared__ float Bs[TILE_WIDTH] [TILE_WIDTH];

      int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
      int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

      float c_val = 0.0f;

      for(int i = 0; i < width/TILE_WIDTH; i++){
            As[threadIdx.y][threadIdx.x] = A[row * width + (i * TILE_WIDTH + threadIdx.x)];
            Bs[threadIdx.y][threadIdx.x] = B[(i * TILE_WIDTH + threadIdx.y) * width + col ];
            __syncthreads();

            for(int k = 0; k < TILE_WIDTH; k++)
                  c_val += As[threadIdx.y][k] * Bs[k][threadIdx.x];
            __syncthreads();
      }
      C[row * width + col] = c_val;
}
```

Allocate As and Bs in shared memory

Matrix Multiplication with Shared Memory

```
__global__ void matrixMul(float* A, float* B, float* C, int width)
{
      __shared__ float As[TILE_WIDTH] [TILE_WIDTH];
      __shared__ float Bs[TILE_WIDTH] [TILE_WIDTH];

      int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
      int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

      float c_val = 0.0f;

      for(int i = 0; i < width/TILE_WIDTH; i++){
            As[threadIdx.y][threadIdx.x] = A[row * width + (i * TILE_WIDTH + threadIdx.x)];
            Bs[threadIdx.y][threadIdx.x] = B[(i * TILE_WIDTH + threadIdx.y) * width + col ];
            __syncthreads();

            for(int k = 0; k < TILE_WIDTH; k++)
                  c_val += As[threadIdx.y][k] * Bs[k][threadIdx.x];
            __syncthreads();
      }
      C[row * width + col] = c_val;
}
```

Iterate through each tile

Matrix Multiplication with Shared Memory

```
__global__ void matrixMul(float* A, float* B, float* C, int width)
{
      __shared__ float As[TILE_WIDTH] [TILE_WIDTH];
      __shared__ float Bs[TILE_WIDTH] [TILE_WIDTH];

      int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
      int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

      float c_val = 0.0f;

      for(int i = 0; i < width/TILE_WIDTH; i++){
            As[threadIdx.y][threadIdx.x] = A[row * width + (i * TILE_WIDTH + threadIdx.x)];
            Bs[threadIdx.y][threadIdx.x] = B[(i * TILE_WIDTH + threadIdx.y) * width + col ];
            __syncthreads();

            for(int k = 0; k < TILE_WIDTH; k++)
                  c_val += As[threadIdx.y][k] * Bs[k][threadIdx.x];
            __syncthreads();
      }
      C[row * width + col] = c_val;
}
```

Make sure that all elements are loaded

Matrix Multiplication with Shared Memory

```
__global__ void matrixMul(float* A, float* B, float* C, int width)
{
      __shared__ float As[TILE_WIDTH] [TILE_WIDTH];
      __shared__ float Bs[TILE_WIDTH] [TILE_WIDTH];

      int row = blockIdx.y * TILE_WIDTH + threadIdx.y;
      int col = blockIdx.x * TILE_WIDTH + threadIdx.x;

      float c_val = 0.0f;

      for(int i = 0; i < width/TILE_WIDTH; i++){
            As[threadIdx.y][threadIdx.x] = A[row * width + (i * TILE_WIDTH + threadIdx.x)];
            Bs[threadIdx.y][threadIdx.x] = B[(i * TILE_WIDTH + threadIdx.y) * width + col ];
            __syncthreads();

            for(int k = 0; k < TILE_WIDTH; k++)
                  c_val += As[threadIdx.y][k] * Bs[k][threadIdx.x];
            __syncthreads();
      }
      C[row * width + col] = c_val;
}
```

Make sure calculation are completed before loading new values

Matrix Multiplication with Shared Memory

- GPU on-chip resources are limited
- Each SMs has limited shared memory space
- Shared memory space is also shared by thread blocks
- What if a thread block allocate large shared memory ?
  - Maxwell SM can execute 32 thread blocks maximum
  - Maxwell SM has 64KB shared memory space
  - If a thread block requires 8KB of shared memory
    - E.g. TILE_WIDTH is 32
    - 32x32x4 = 4K for As and Bs
  - Only 8 thread blocks can reside in an SM at a time

Q & A