Advanced Topics in CUDA

Sudsanguan Ngamsuriyaroj
Ekasit Kijsipongse
Putt Sakdhnagool

Semester 1/2019

Topics

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

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).

For each thread i, do counting 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 */
```

ของบล็อก   มีกี่บล็อก   หา ID ในblock

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

CUDA Rank Sort When n = T (small n)

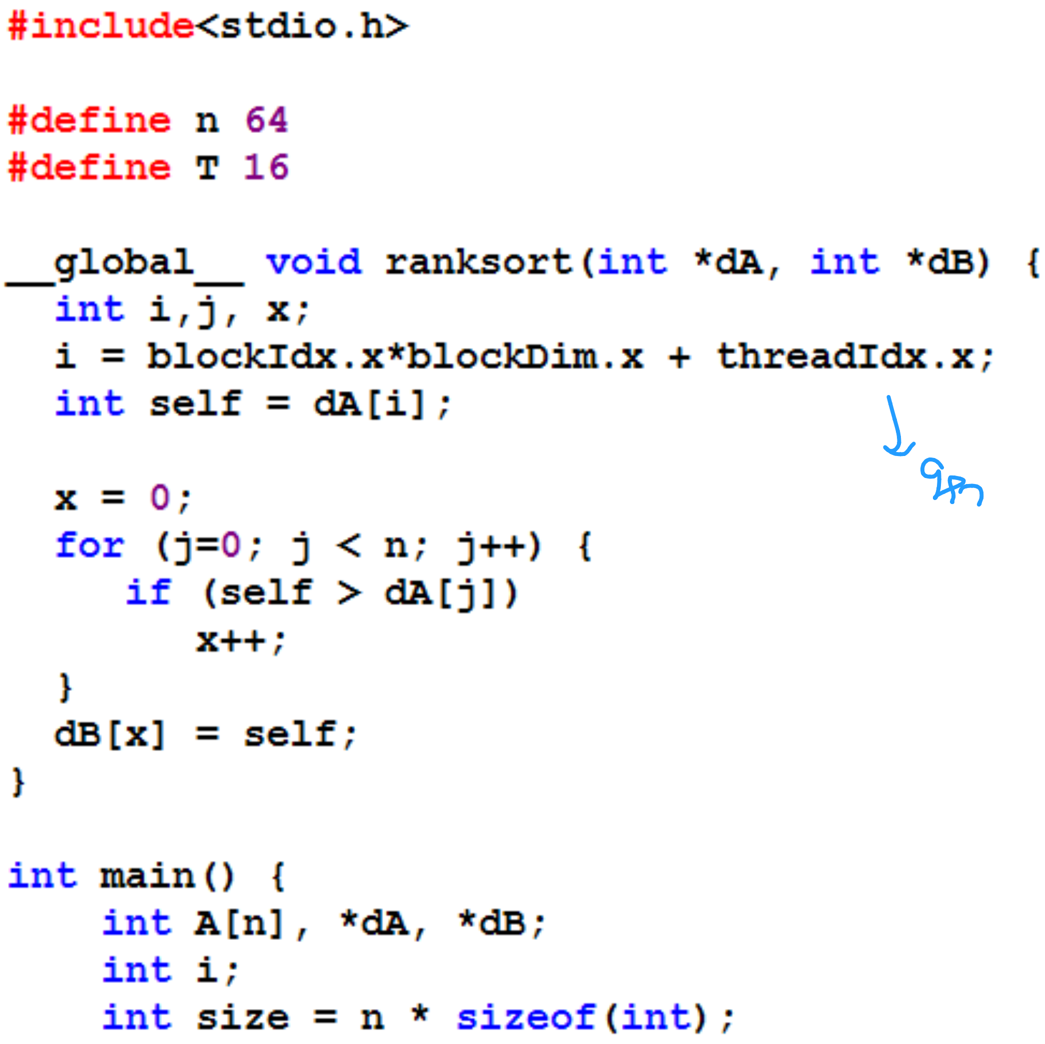
```
__global__ void ranksort(int *dA, int *dB) {
  2  int i = threadIdx.x;เลขที่ใช้ เพราะตอนนี้มีแค่ block.   // one thread block
     int self = dA[i];ตำแหน่ง ในอาร์เรย์
     int x = 0;
     for (int j=0; j < n; j++) {
          if (self > dA[j])
               x++;มากกว่ากี่ตำแหน่ง
     }
     dB[x] = self;
}         1    =  1

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

blo c k.Id. x

0	7	1	3	2	4	5

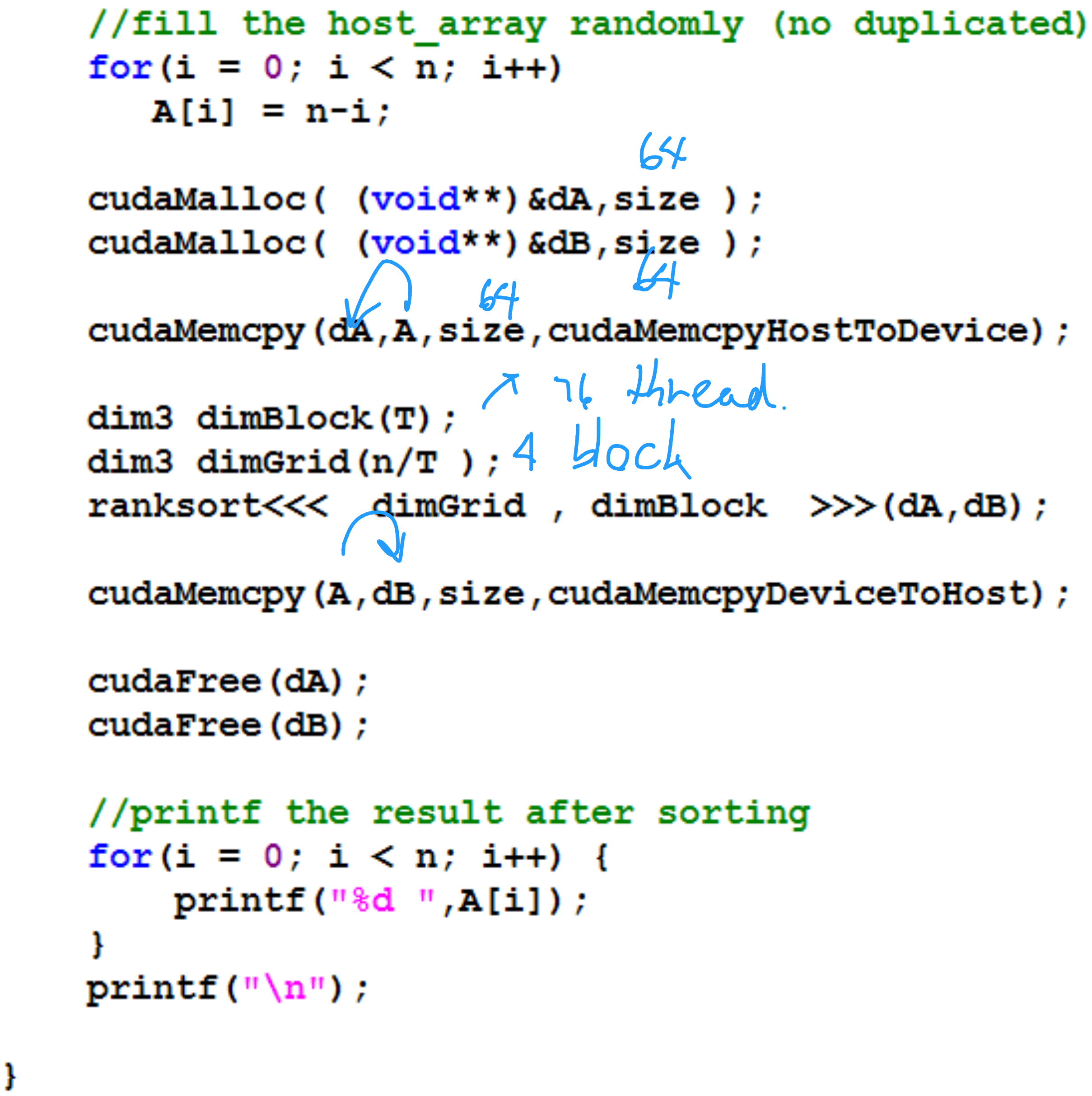
0 1 2 3 4 5 6

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);
```

↓ am global ID



```
    //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);
```

64

64

64

dA = A

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

16 thread.

4 block

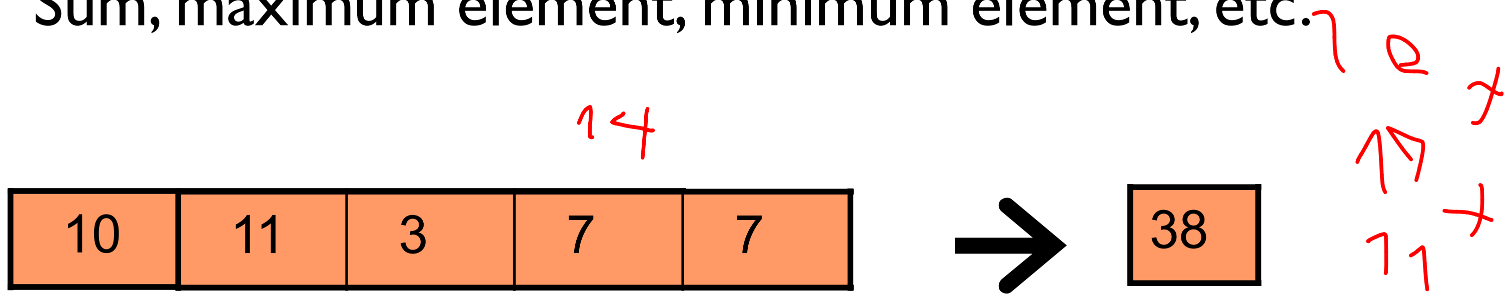
```
    cudaMemcpy(A,dB,size,cudaMemcpyDeviceToHost);
```

A = dB

```
    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.



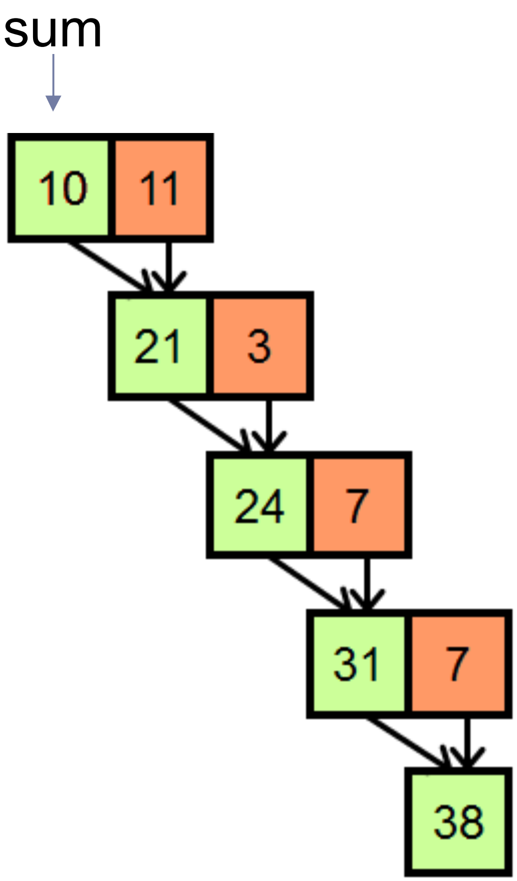
- Useful primitive used in lots of applications

Sequential Reduction

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

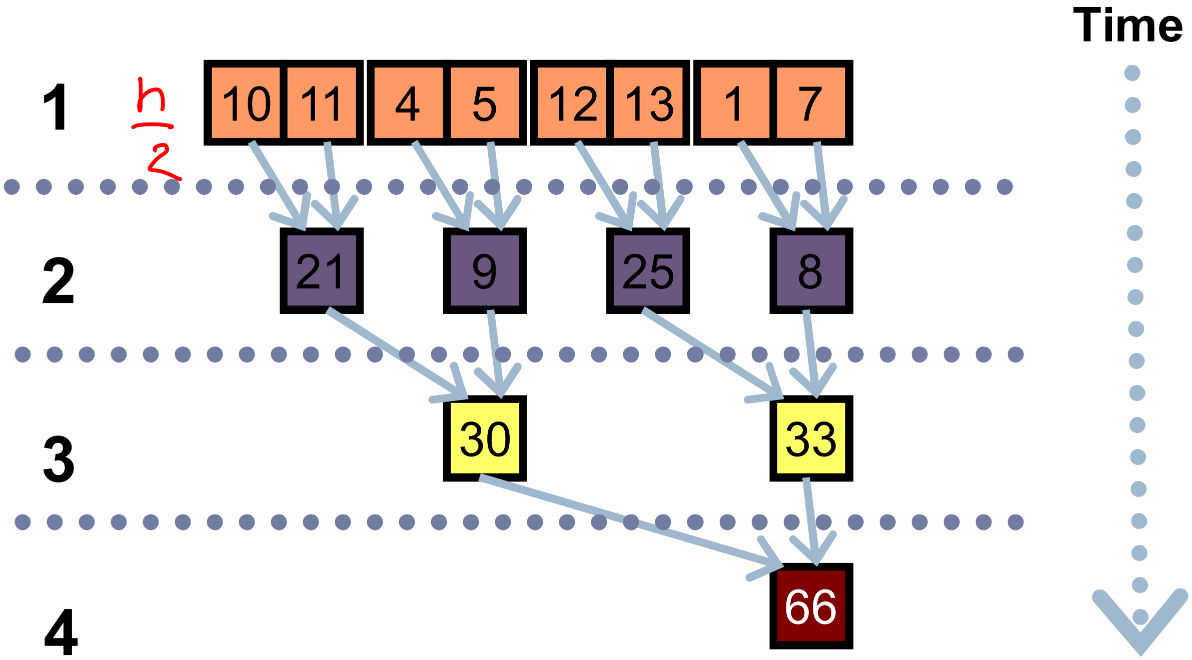
[illegible]

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

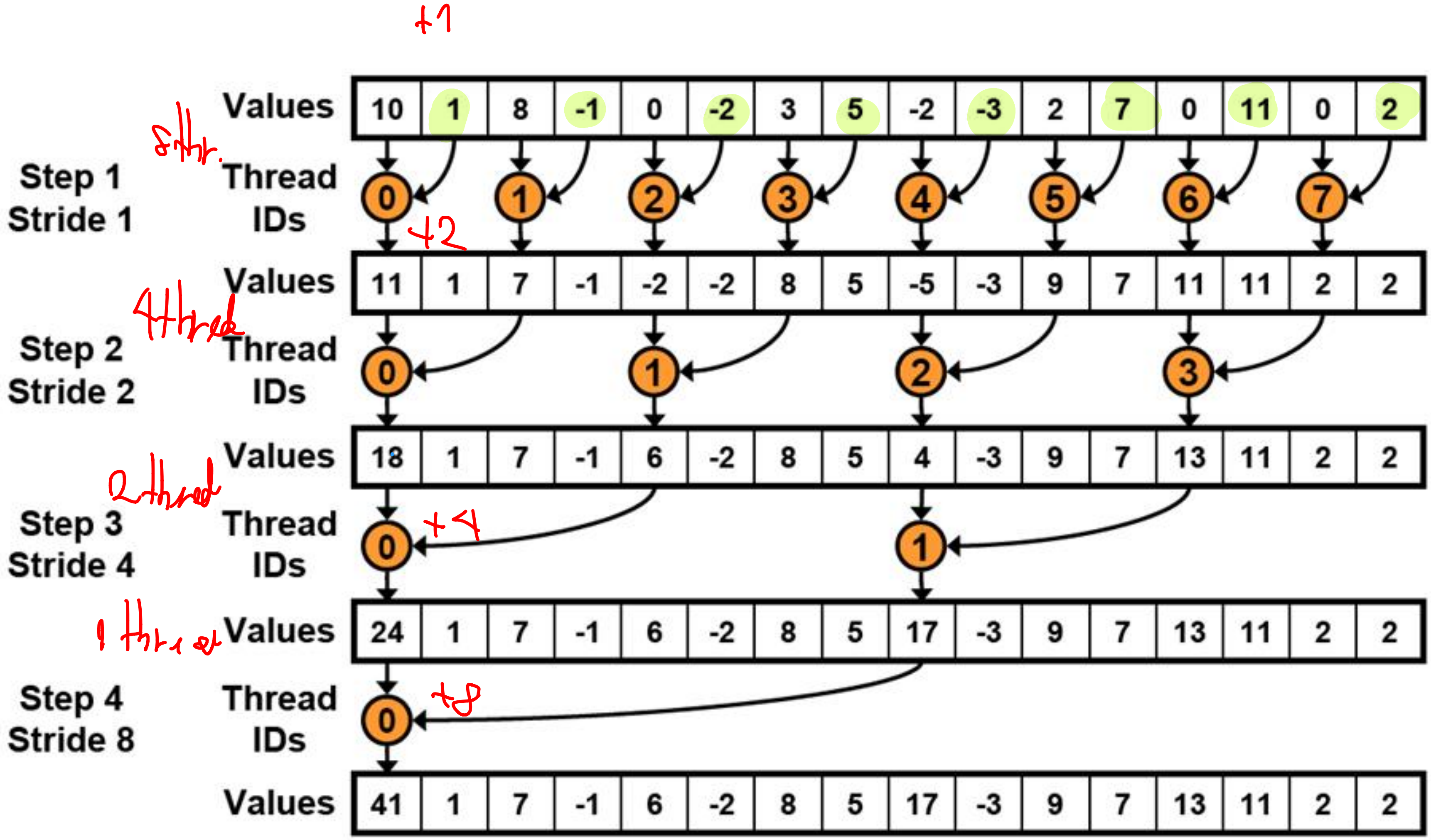


Parallel Reduction

- 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

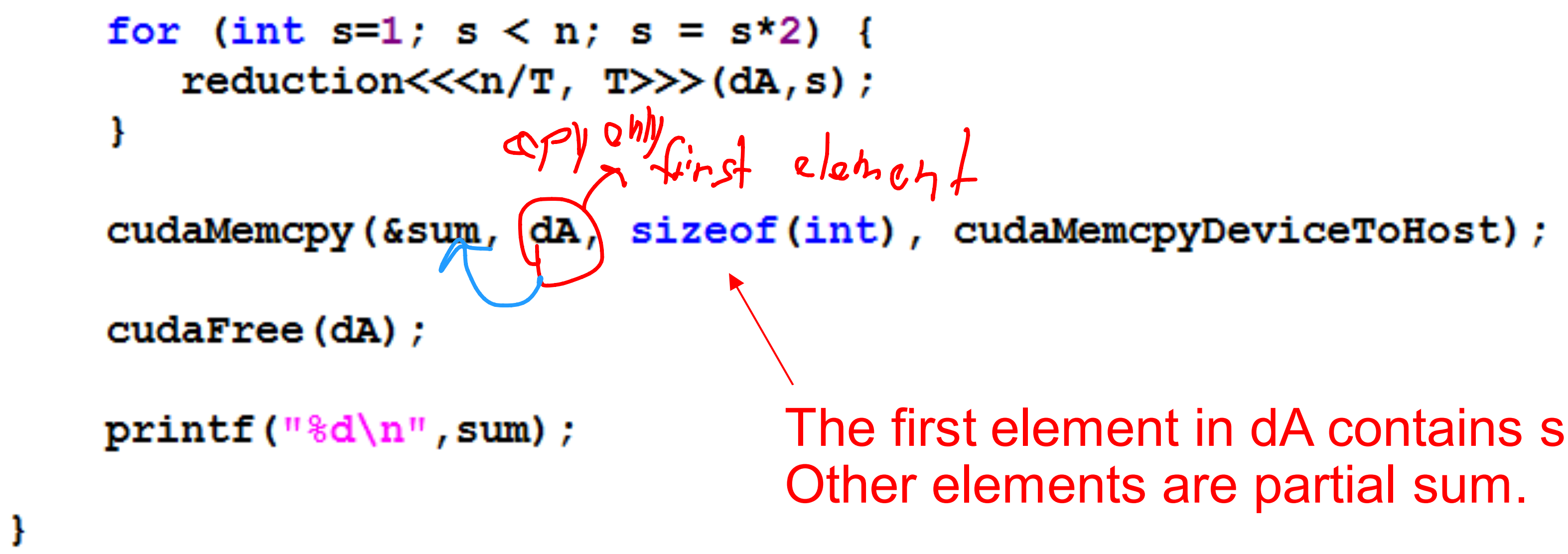


Values	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 1 Thread IDs	0		1		2		3		4		5		6		7	
Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Step 2 Stride 2 Thread IDs	0				1				2				3			
Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Step 3 Stride 4 Thread IDs	0								1							
Values	24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
Step 4 Stride 8 Thread IDs	0															
Values	41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

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;   0, 1, 2, 3, 4, 5
                          1+2+4+8
    int idx = 2*stride*tid;  2*  {1,2,4,8} *1
    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);

}
```

copy only first element

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

Error Checking

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

…
```

when copy result จะทำ 1 รอบ

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

return error ออกมา

Read-modify-write problem in parallel computation

race condition in openMP

- 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	share memory x = x + 1;	Old ← Mem[x]	Old ← Mem[x]
	try to update	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) Old ← 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

X

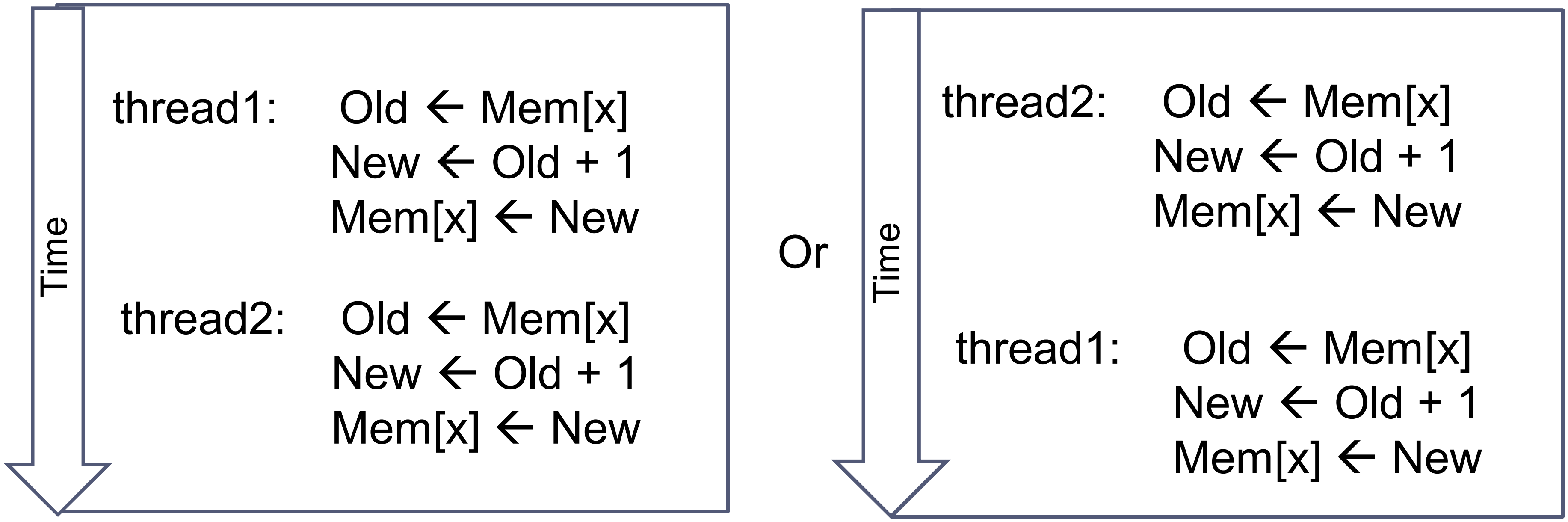
race condition

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



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

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

Or

```
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.

lock

ehs

wire [illegible]

ค่อยไปต่อ

Atomic Operations

perfor

- 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

Soft

A = 10

AtomicAdd

int B = atomic Add (&A, 1)

old value of A

B = 10, A = 11

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

  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);

A Simple AtomicAdd() Example

```
__device__ int sum = 0;

__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);
    …
}
```

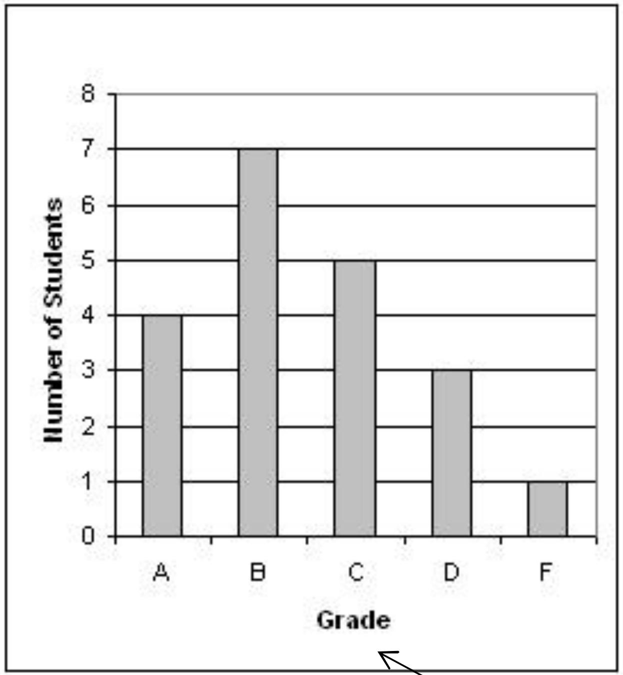
Low performance since all threads sequentially add numbers to sum!!

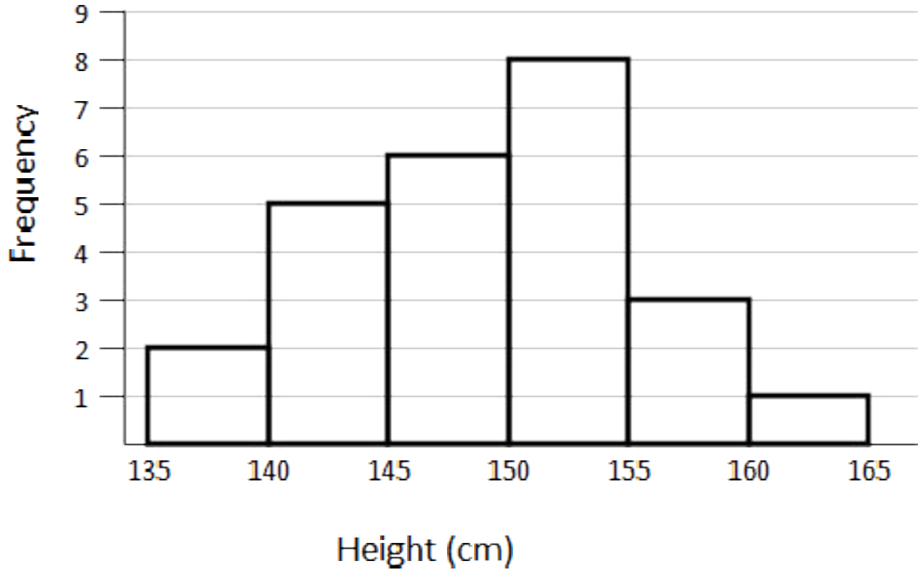
Use reduction operation instead.

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

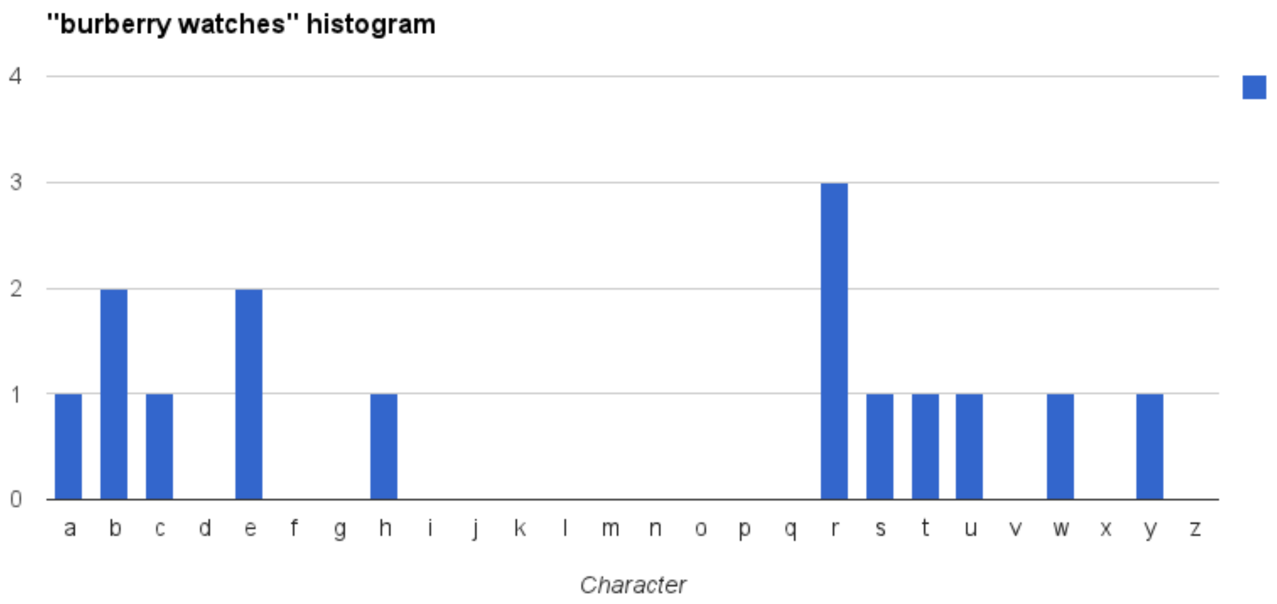




Bin (could be interval)

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

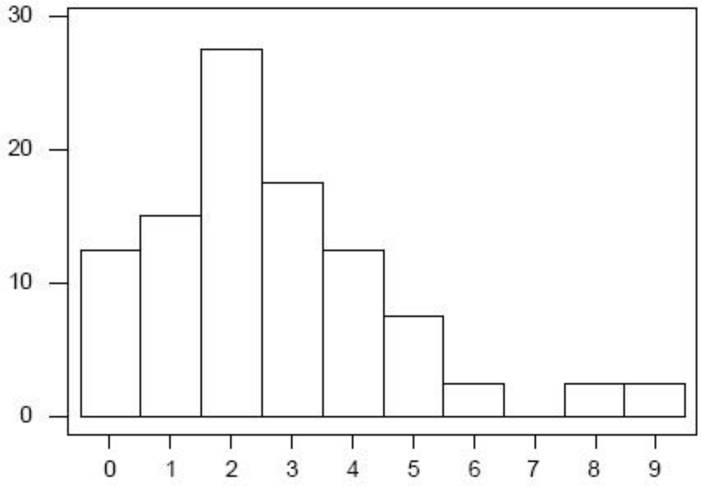
- For each element in the data set, use the value to identify a “bin” to increment

```
// 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]]++;
```

data[i]

0

bin[2]++



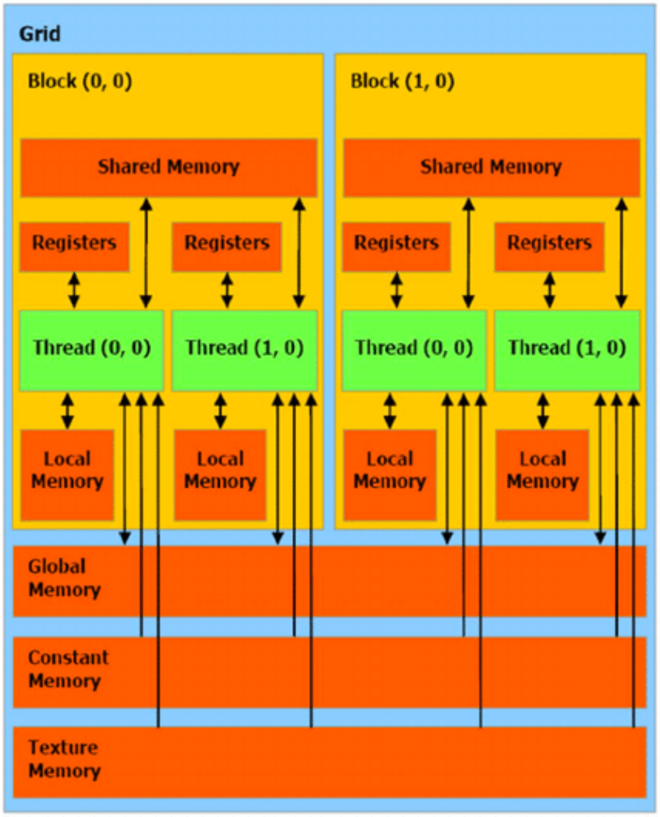
histogram.cu

```
#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; ID
    if (i < n) {                                 +1
        atomicAdd( &(bin[data[i]]), 1 );                    incorrect
    }                            ✓
}
```

```
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));
                                             dA ← a
    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

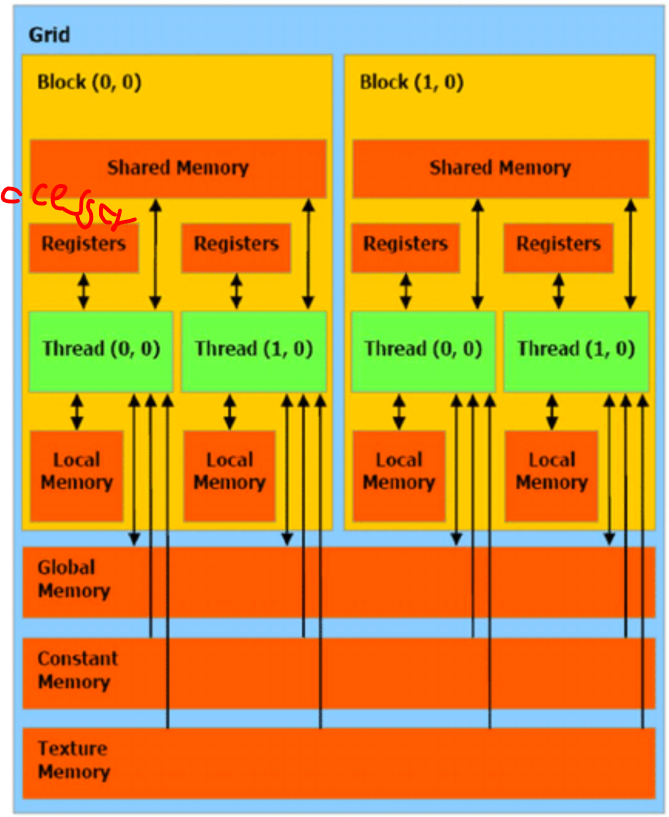
- GPUs have multiple memory spaces
- How to make the best use of the GPU memory system?
- How to deal with hardware limitation?



CUDA Memory Hierarchy

- CUDA threads may access data from multiple memory spaces ใกล้กว่า หมาย = closer to the processor
  - 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	



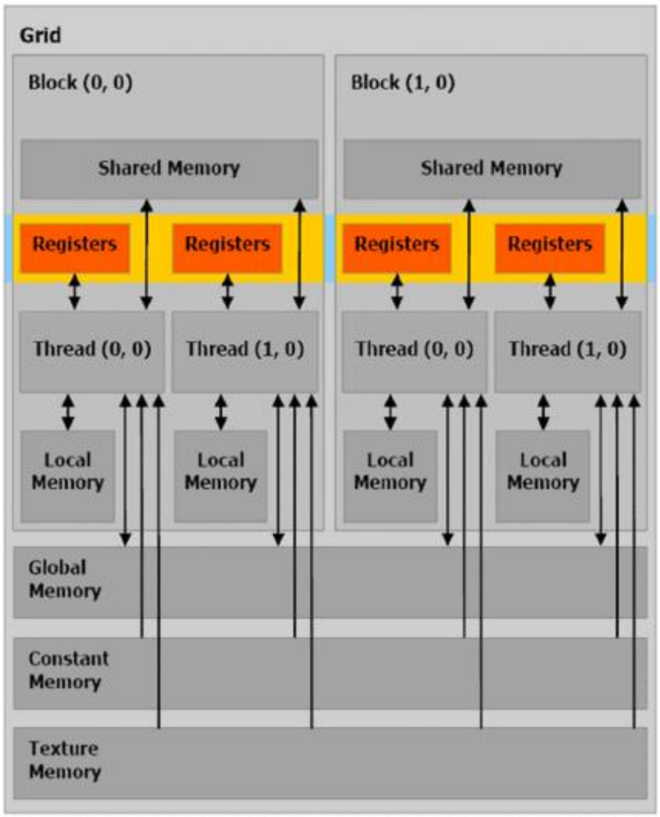
Registers

1024

[illegible] 32 register

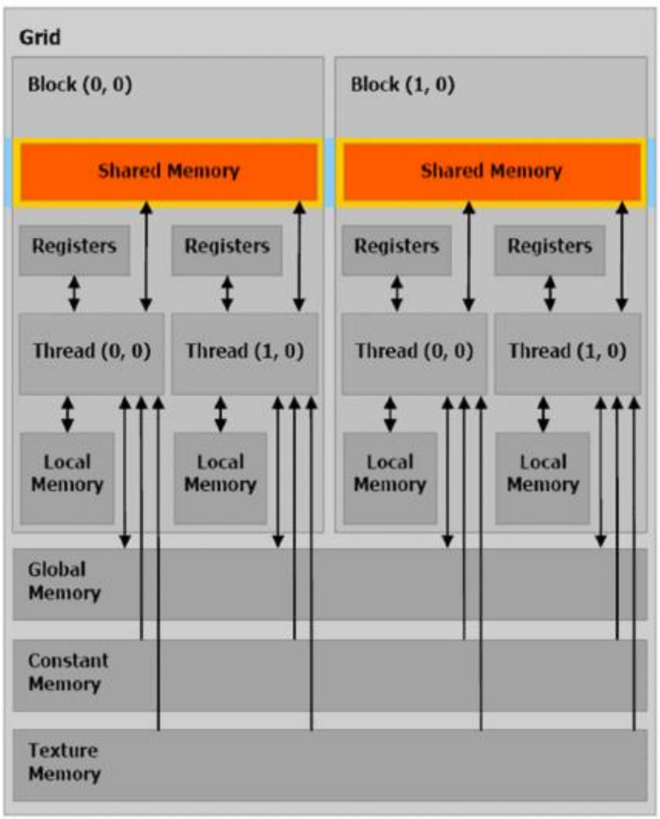
- Fastest memory
- Only accessible by a thread
- Lifetime of a thread
- Automatically allocated
  - Typically for scalar variables
- Number of registers is very limited
- Resides in register files
  - Shared across all active threads in an SM
- Cannot access by host

directly



Shared Memory in sid SM.

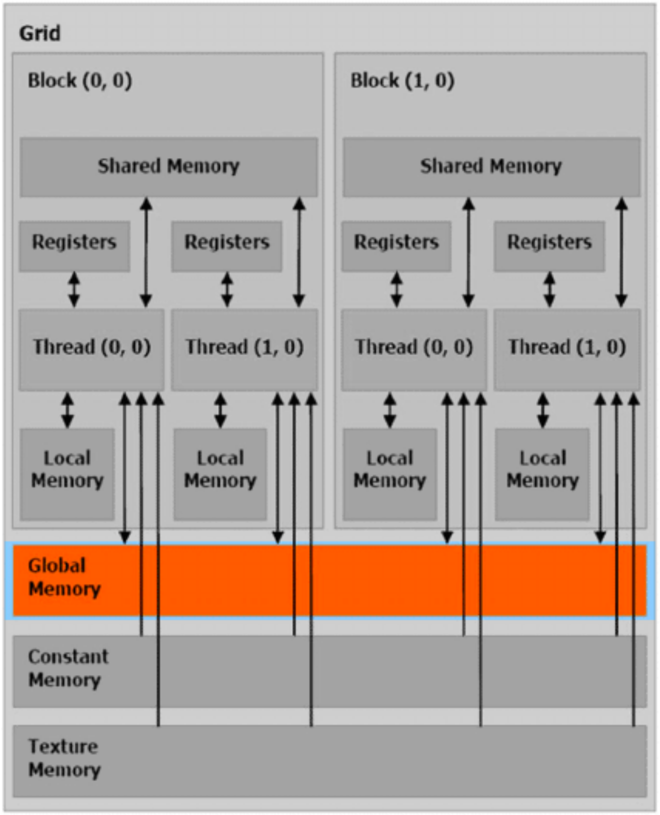
- Extremely fast
- Shared across threads in the **same thread block**
- Lifetime of a kernel share memory
- User-managed
  - Programmers must explicitly allocate shared memory in the kernel
  - __shared__ specifier
  - Will discuss later
- Cannot access by host



Global Memory

ช้ากว่า register and share 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



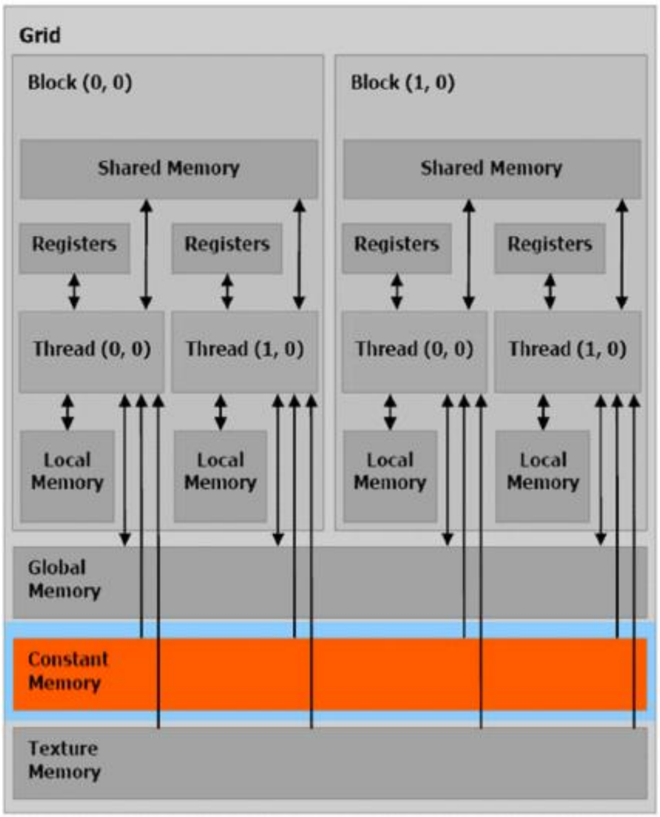
Constant Memory

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

int tid = —

c[tid]= a[tid]+b[tid]

c[tid] =a[tid]~~x~~x +b[tid]



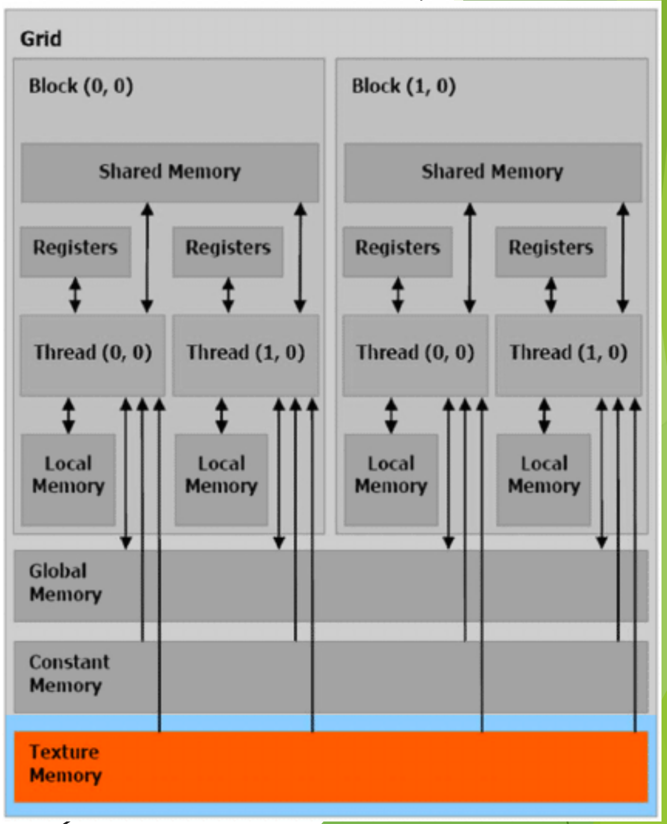
temporal locality ~ time

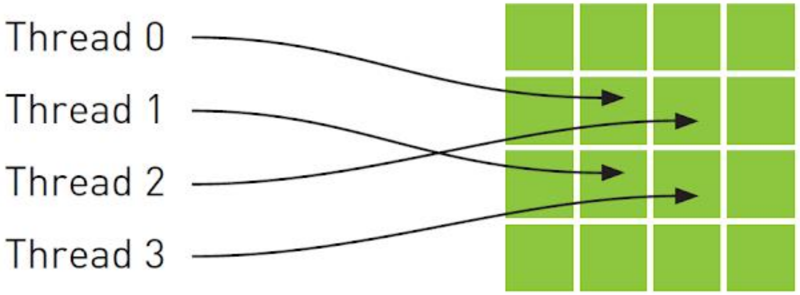
spatial locality ~ space → ข้อมูลอยู่ใกล้ cache อยู่แล้ว

temporal = ใกล้กันใน term of time.

Texture Memory

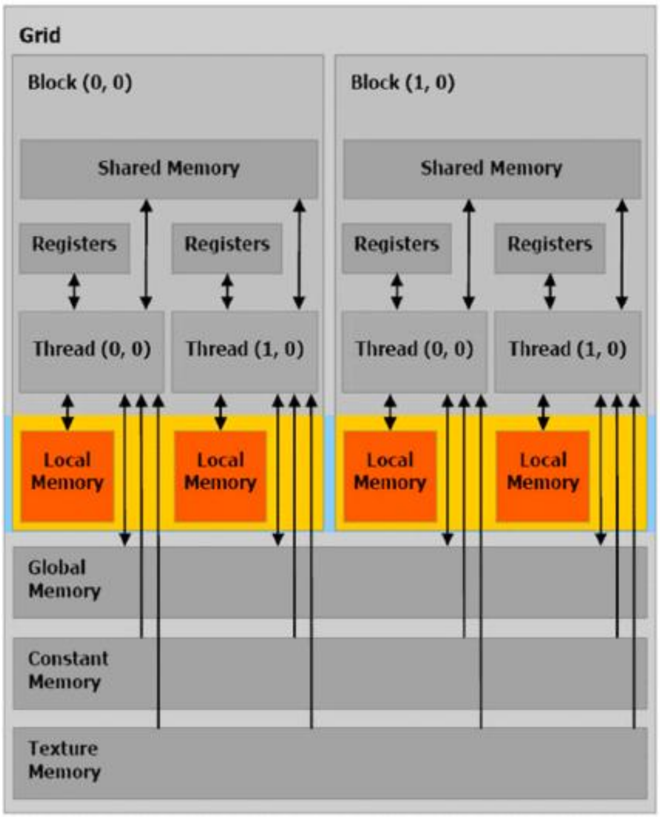
- Special type of global memory
- **Read-only**
- Cached → จะไม่ให้ต้องรอข้อมูลที่โหลดมาแล้ว ไว้ใช้ซ้ำ
- Optimized for 1D, 2D, or 3D spatial locality in graphic programs
  - Threads read memory is nearby location





Local Memory

- Not an actual memory space
- Similar to register
  - Only accessible by a thread
  - Lifetime of a thread อยู่ทีละ [illegible]
  - 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

data cohand share হয়

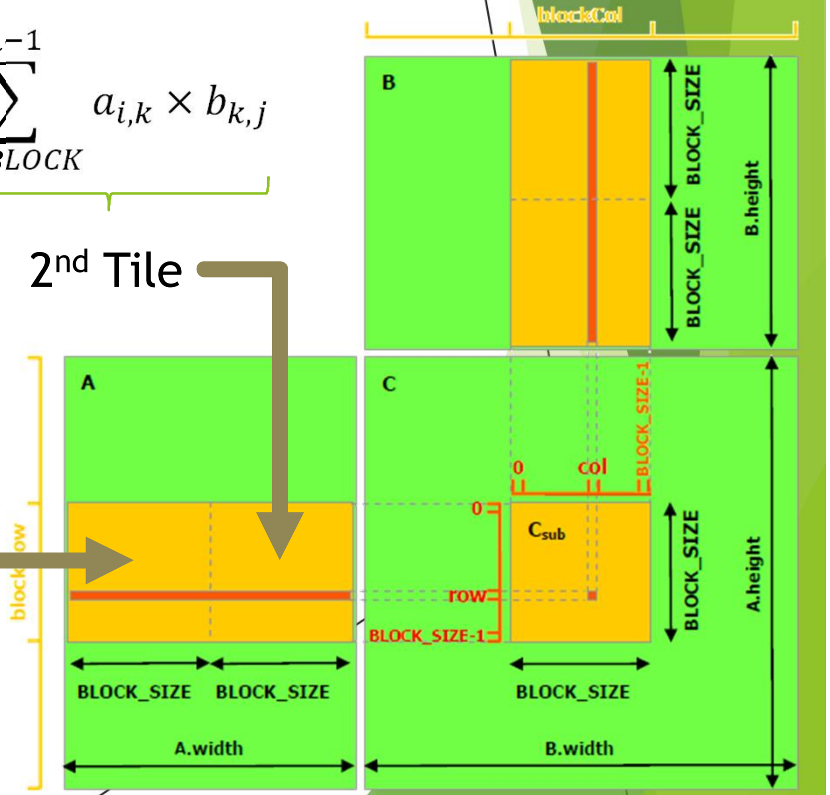
Back to Matrix Multiplication

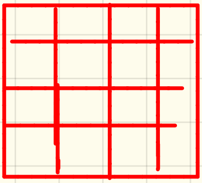
- **Strategy:** partition the data into subsets called "tiles", such that each tile fits into the shared memory.
  - Data loaded inside a tile could be reused
  - Make threads that use common elements collaborate
- Each thread can load a submatrix into the shared memory before calculations
- The submatrix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

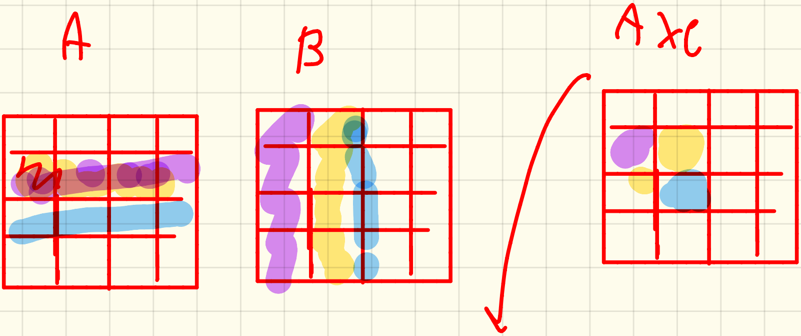
Matrix Multiplication with Shared Memory

- Looking at matrix multiplication equation

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







we share 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;
}
```

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

load data to sm [illegible]

we share memory instead

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

update cval

ith matx at

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

```
__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_

    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;
}
```

Any remaining problem?

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 concurrently execute 32 thread blocks
  - Maxwell SM has 64KB shared memory space
  - If a thread block requires 8KB of shared memory
    - E.g. TILE_WIDTH is 64
  - Only 8 thread blocks can reside in an SM at a time

Occupancy

- A ratio of active warps to the maximum number of active warps supported by the SM
- A ratio of active threads to the maximum number of active threads supported by the SM
  - Simpler definition but not 100% accurate
- Indicate how well the GPU is utilized
- Usually, higher occupancy leads to better performance
  - 60-80% is a reasonable goal for occupancy
  - But not always
- **Occupancy Factors**
  - Register used per thread
  - Shared memory used per thread block
  - Thread block size

Occupancy: Thread block size

- Indicate a units of threads to be executes on an SM
- For example, if a kernel has a thread block size of 80 thread
  - Maxwell SM can execute 2048 threads concurrently
  - Only 2000 threads (25 blocks) can be executed
  - The remaining 48 threads are idle. [illegible]
  - Maximum occupancy is 2000/2048 = 98%

0.25

80)2000

16

40

Occupancy: Register

- Each SM can store a limited number of registers
- Register usage could limit number of threads
- For example, Maxwell GPU has 64K registers per SM
  - Maxwell SM can execute 2048 threads concurrently
  - If a kernel requires 64 registers per thread
  - Only 64K/64 = 1024 threads can be executed on SM
  - Maximum occupancy is 1024/2048 = 50%
- How can we know the number of register requires per threads ?
  - Compile program with --resource-usage flags

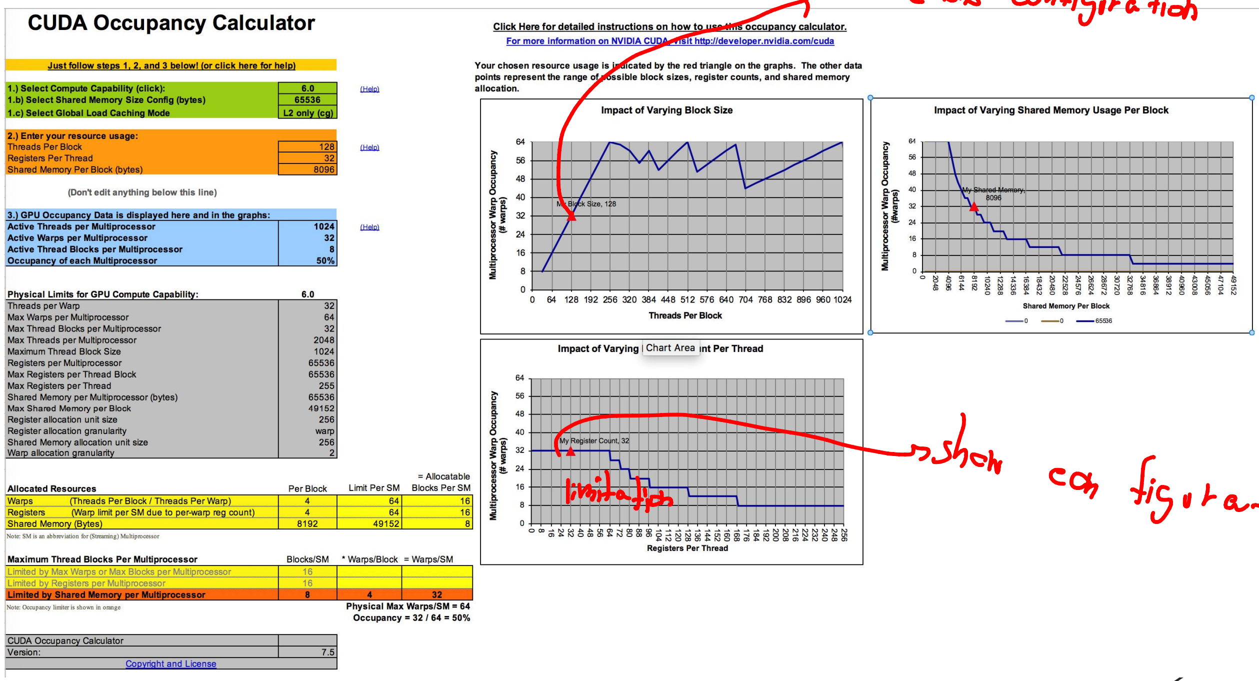
```
$ nvcc --resource-usage acos.cu
ptxas info : 1536 bytes gmem, 8 bytes cmem[14]
ptxas info : Compiling entry function 'acos_main' for 'sm_30’
ptxas info : Function properties for acos_main
                 0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info : Used 6 registers, 1536 bytes smem, 32 bytes cmem[0]
```

Occupancy: Shared Memory

- Each SM has limited shared memory space
- Shared memory space is shared across thread blocks in the SM
- Shared memory could limit number of thread blocks => limiting number of threads
- For example, Maxwell GPU has 64KB shared memory per SM
  - Maxwell SM can execute 2048 threads concurrently
  - If a kernel requires 8KB of shared memory i.e. 8 thread block
  - Only 64KB/8KB = 8 thread blocks can be executed on SM
  - If thread block size is 128 threads => 8*128 = 1024 threads
  - Maximum occupancy is 1024/2048 = 50%

Occupancy: How to Compute?

- Nvidia provide a tool called Occupancy Calculator
- Excel calculator for computing maximum occupancy



CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	6.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
1.c) Select Global Load Caching Mode	L2 only (cg)	

2.) Enter your resource usage:		
Threads Per Block	128	(Help)
Registers Per Thread	32	
Shared Memory Per Block (bytes)	8096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	50%	

Physical Limits for GPU Compute Capability:	6.0
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	2

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	4	64	16
Registers	(Warp limit per SM due to per-warp reg count)	4	64	16
Shared Memory (Bytes)		8192	49152	8

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	16		
Limited by Registers per Multiprocessor	16		
Limited by Shared Memory per Multiprocessor	8	4	32

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 32 / 64 = 50%

CUDA Occupancy Calculator	
Version:	7.5
Copyright and License	

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

If chans configiration

limitation

show configuration

Q & A