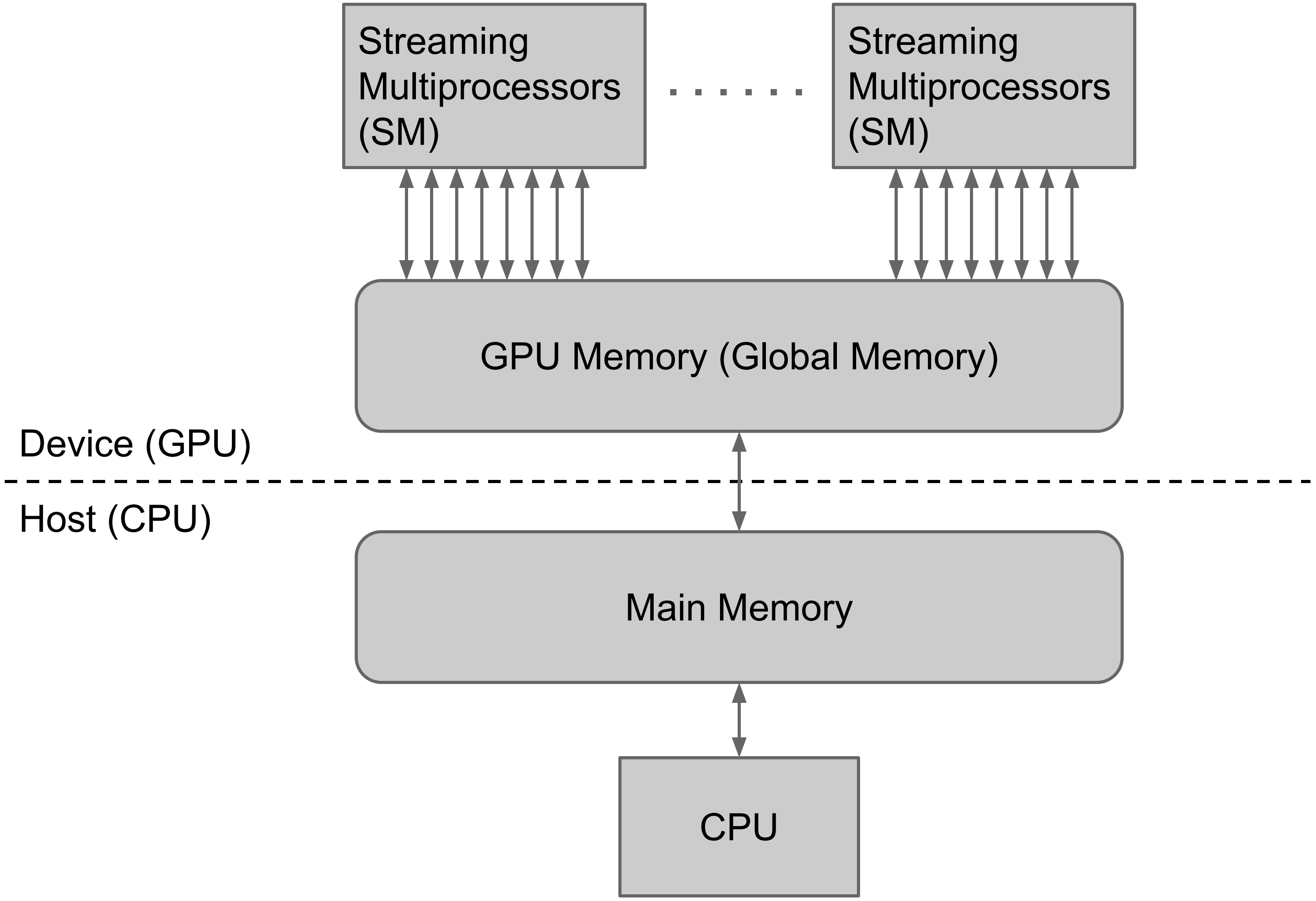
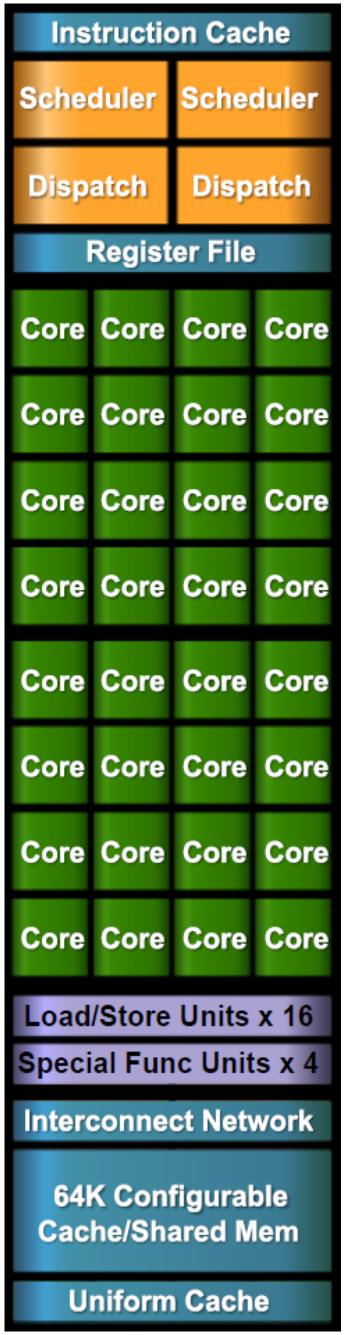
Introduction to GPU Programming with CUDA

Shengren Li
shrli@ucdavis.edu

- Jan 13 and 20
- Chapter 5
- CUDA-enabled GPU


Streaming Multiprocessors (SM)
Streaming Multiprocessors (SM)
GPU Memory (Global Memory)
Device (GPU)
Host (CPU)
Main Memory
CPU

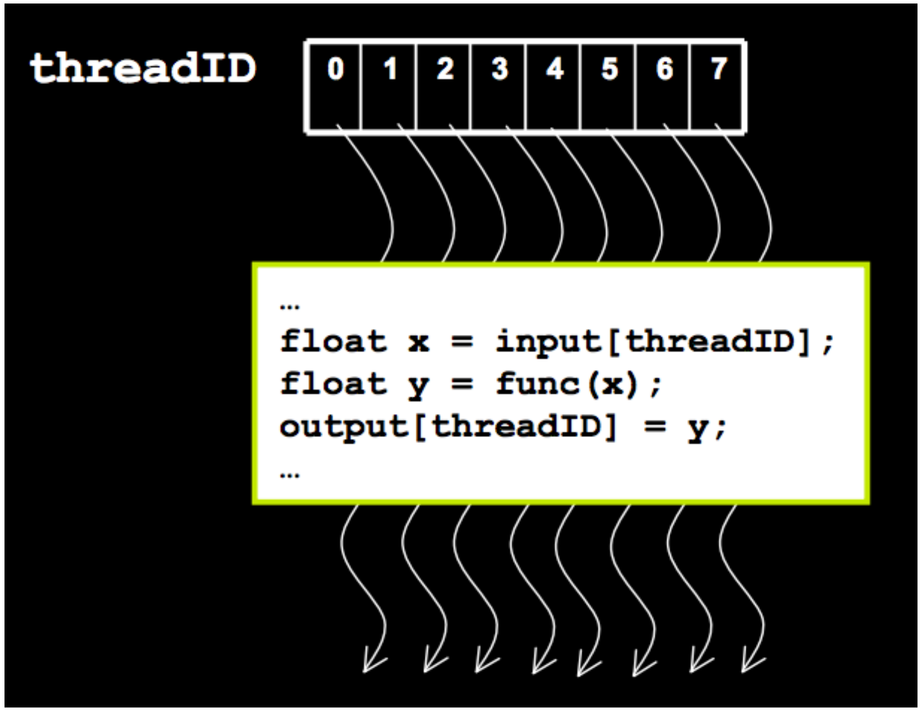
Streaming Multiprocessors (SM)



- Core = Streaming Processor (SP)
- One SP runs one thread at a time slice
- 32 SPs runs a warp of 32 threads
- Threads in a warp share instruction unit
- Single Instruction Multiple Thread (SIMT)
- Single Instruction Multiple Data (SIMD)
- All the threads execute the same program, kernel function

CUDA Threads

- All threads run the same code
- Each thread has an ID that it uses to compute memory addresses and make control decisions



```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

Typical CUDA Program - Host

1. Allocate host memory (input & output)
2. Put input data into host memory
3. Allocate device memory (input & output)
4. Copy input data from host memory to device memory
5. Set launch configuration
6. Launch kernel function
7. Copy output data from device memory to host memory
8. Release memory (host & device)

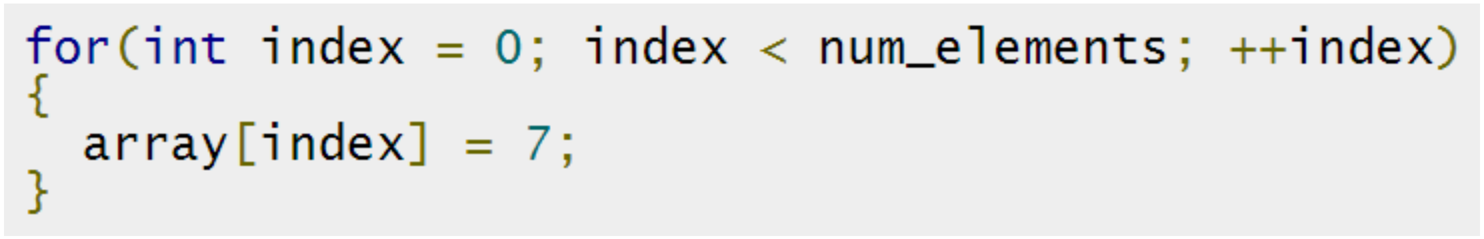
Typical CUDA Program - Device

Kernel = function that runs on the device

1. Figure out the piece of input data assigned to the current thread
2. Read input data from device memory
3. Computation (input -> output)
4. Write output data to device memory

Example

Set all 256 elements in an array to 7



```
for(int index = 0; index < num_elements; ++index)
{
  array[index] = 7;
}
```

```
int main(void)
{
  int num_elements = 256;

  int num_bytes = num_elements * sizeof(int);

  // pointers to host & device arrays
  int *device_array = 0;
  int *host_array = 0;

  // malloc a host array
  host_array = (int*)malloc(num_bytes);

  // cudaMalloc a device array
  cudaMalloc((void**)&device_array, num_bytes);

  int block_size = 128;
  int grid_size = num_elements / block_size;

  kernel<<<grid_size,block_size>>>(device_array);

  // download and inspect the result on the host:
  cudaMemcpy(host_array, device_array, num_bytes, cudaMemcpyDeviceToHost);

  // print out the result element by element
  for(int i=0; i < num_elements; ++i)
  {
    printf("%d ", host_array[i]);
  }

  // deallocate memory
  free(host_array);
  cudaFree(device_array);
}
```

```
__global__ void kernel(int *array)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;

  array[index] = 7;
}
```

GPU Memory Allocation / Release

- cudaMalloc(void **pointer, size_t nbytes)
- cudaMemset(void *pointer, int value, size_t count)
- cudaFree(void *pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes,
  enum cudaMemcpyKind direction);
  - `direction` specifies locations (host or device) of `src` and `dst`
  - Blocks CPU thread: returns after the copy is complete
  - Doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

Launching kernels

- Modified C function call syntax:

```
kernel<<<dim3 grid, dim3 block>>>(…)
```

- Execution Configuration ("<<< >>>"):
  - grid dimensions: x and y
  - thread-block dimensions: x, y, and z

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Can only access GPU memory
  - Must have `void` return type
  - No variable number of arguments (“varargs”)
  - Not recursive
  - No static variables
- Function arguments automatically copied from CPU to GPU memory

Function Qualifiers

- `__global__` : invoked from within host (CPU) code,
  cannot be called from device (GPU) code
  must return void

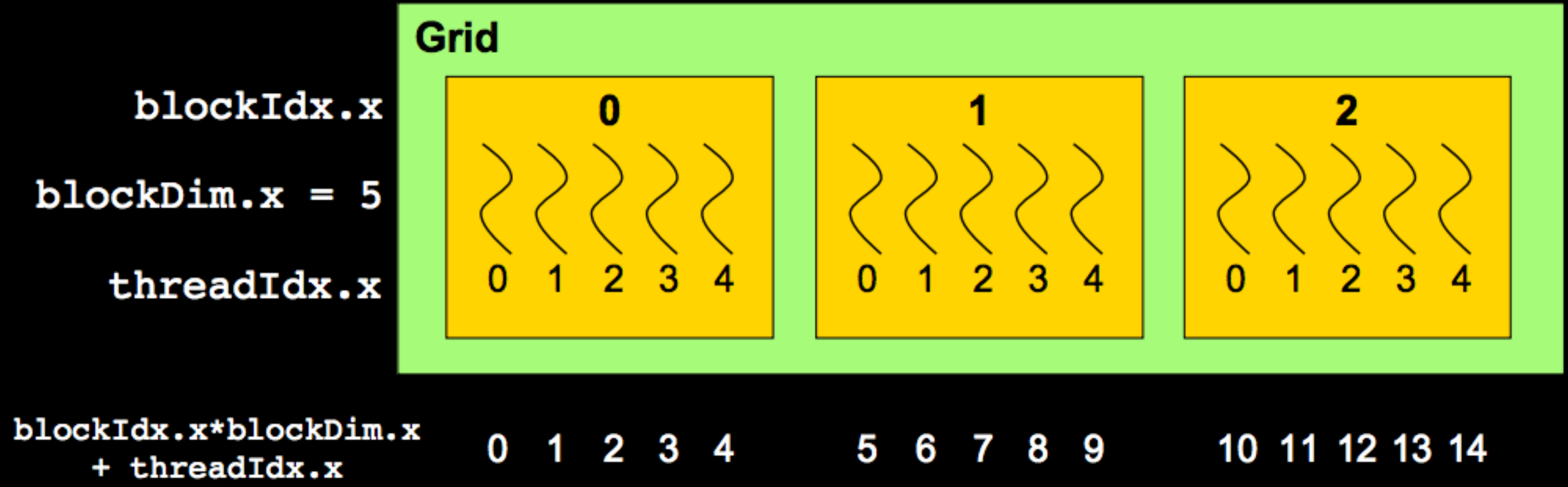
- `__device__` : called from other GPU functions,
  cannot be called from host (CPU) code

CUDA Built-in Device Variables

- All `__global__` and `__device__` functions have access to these automatically defined variables
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

Data Decomposition

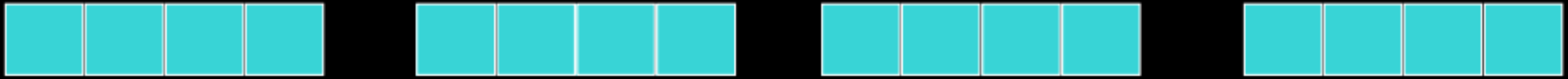
- Often want each thread in kernel to access a different element of an array



Example: Increment Array Elements

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks



blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```
will map from local index threadIdx to global index

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}


void main()
{
  .....
      increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx < N)
            a[idx] = a[idx] + b;
}


void main()
{
   .....
      dim3 dimBlock (blocksize);
      dim3 dimGrid( ceil( N / (float)blocksize)  );
      increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Example on page 112

Calculate row sums

$$\boxed{\begin{matrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{matrix}} \qquad \boxed{\begin{matrix} 6 \\ 22 \\ 38 \\ 54 \end{matrix}}$$

4 x 4

Example on page 112

Calculate row sums

$$\begin{array}{l|ccccccc|c|c|}
\text{thread 0} & 0 & & 1 & & 2 & & 3 & & 6 \\
\text{thread 1} & 4 & + & 5 & + & 6 & + & 7 & = & 22 \\
\text{thread 2} & 8 & & 9 & & 10 & & 11 & & 38 \\
\text{thread 3} & 12 & & 13 & & 14 & & 15 & & 54
\end{array}$$

4 x 4

Example on page 112

- line 23:26 Pointers to host/device input/output
- line 29, 43 Allocate host memory
- line 31:36 Generate input data
- line 38, 44 Allocate device memory
- line 40 Copy input from host to device
- line 46:47 Set launch configuration
- line 49 Launch kernel function
- line 53 Copy output from device to host
- line 57:60 Release memory

Example on page 112

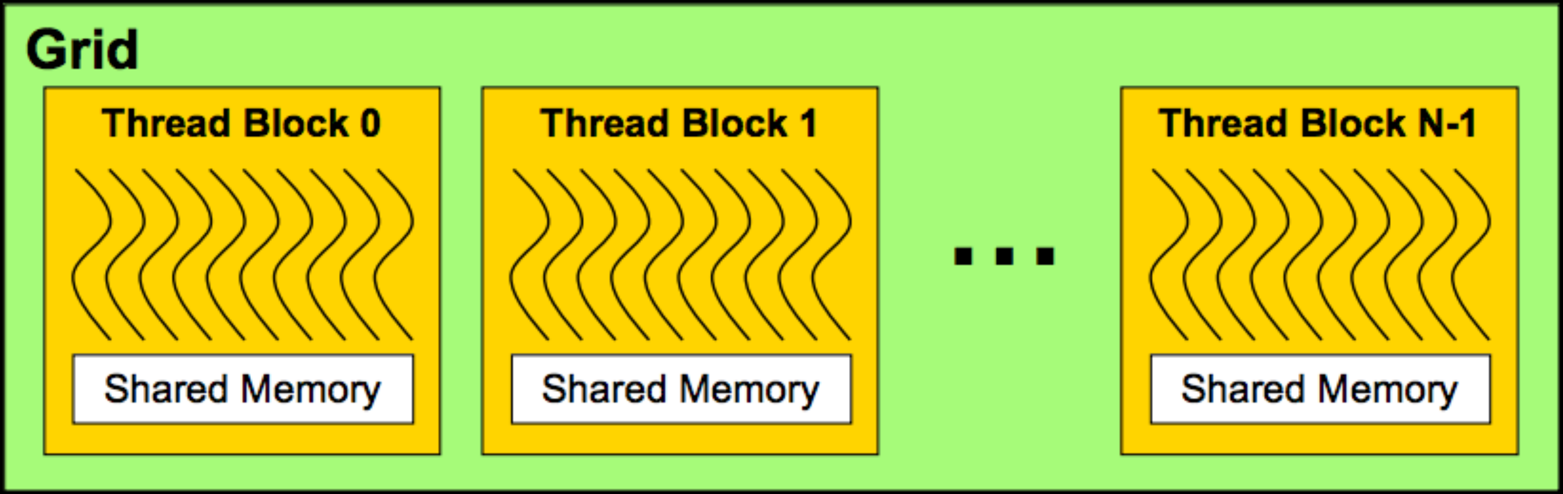
- line 13 Figure out the piece of data assigned to this thread
    - m[rownum * n + k] k = 0:n-1
- line 14:16 Read input from device memory and compute output
- line 17 Write output to device memory

Host Synchronization

- All kernel launches are *asynchronous*
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- cudaMemcpy() is *synchronous*
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete

Thread Batching

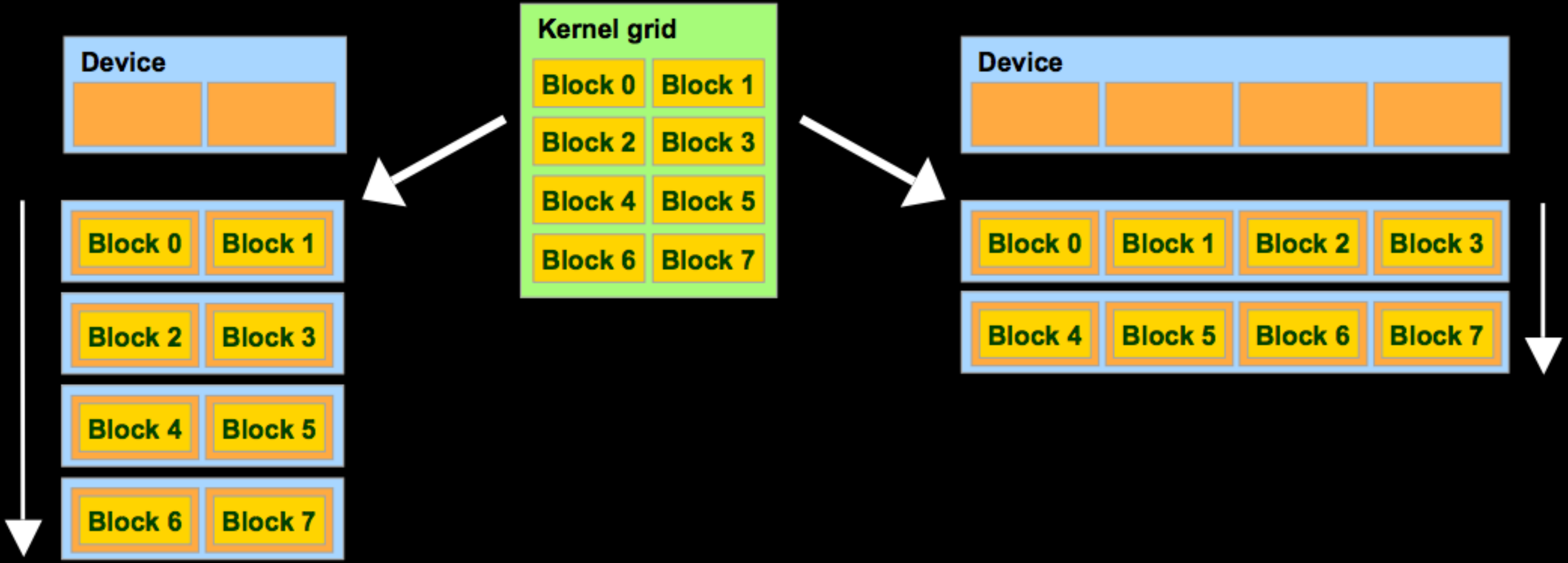
- Kernel launches a grid of thread blocks



  - Threads within a block cooperate via shared memory
  - Threads in different blocks cannot cooperate
- Allows programs to *transparently scale* to different GPUs

Transparent Scalability

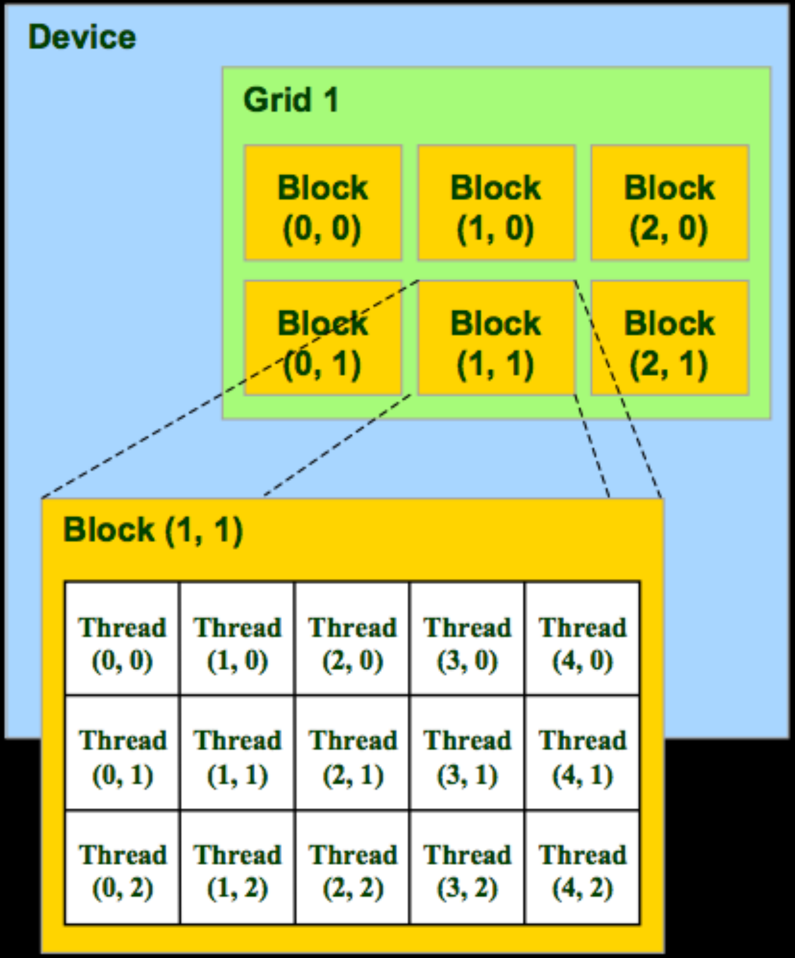
- Hardware is free to schedule thread blocks on any processor



Multidimensional IDs

- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D

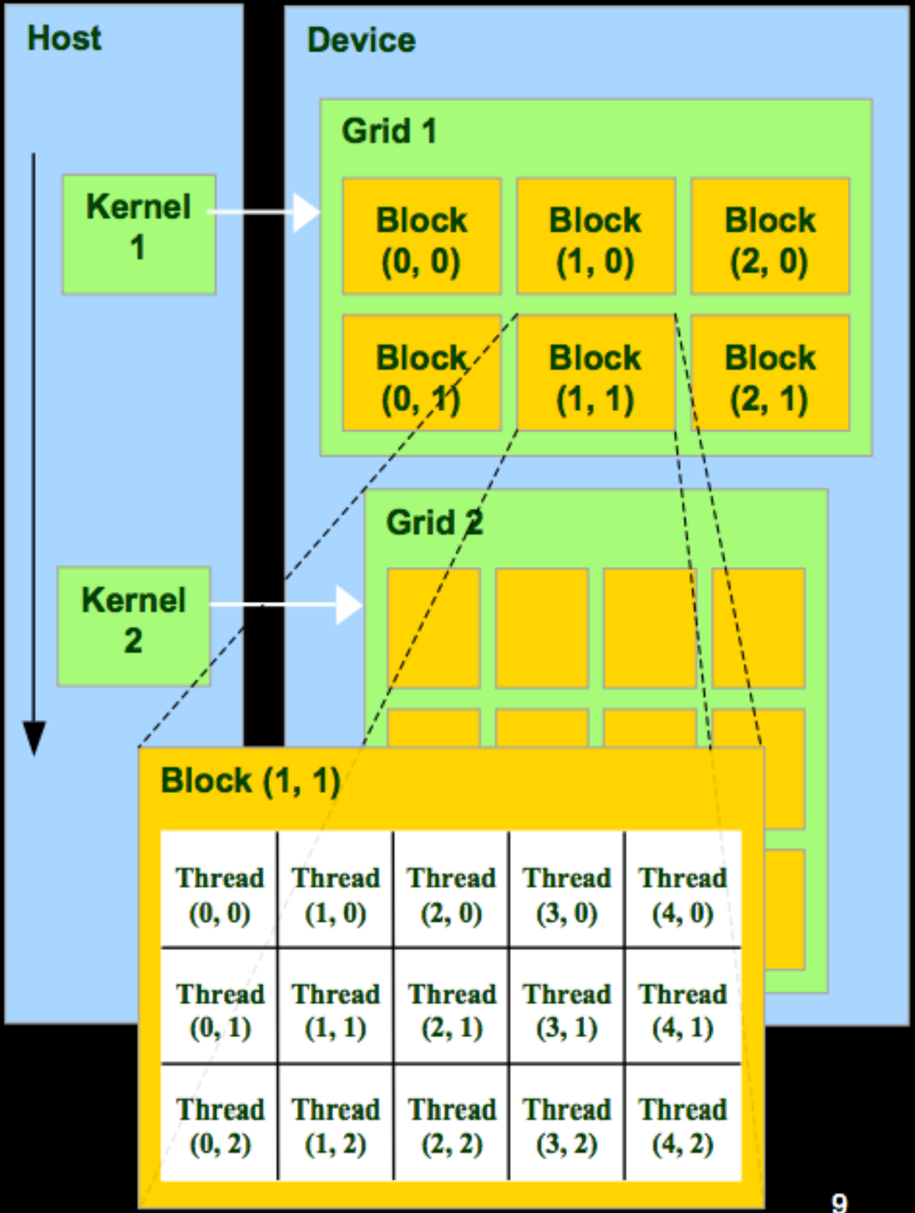
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes



CUDA Programming Model

A kernel is executed by a grid of thread blocks

- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate



Execution Model

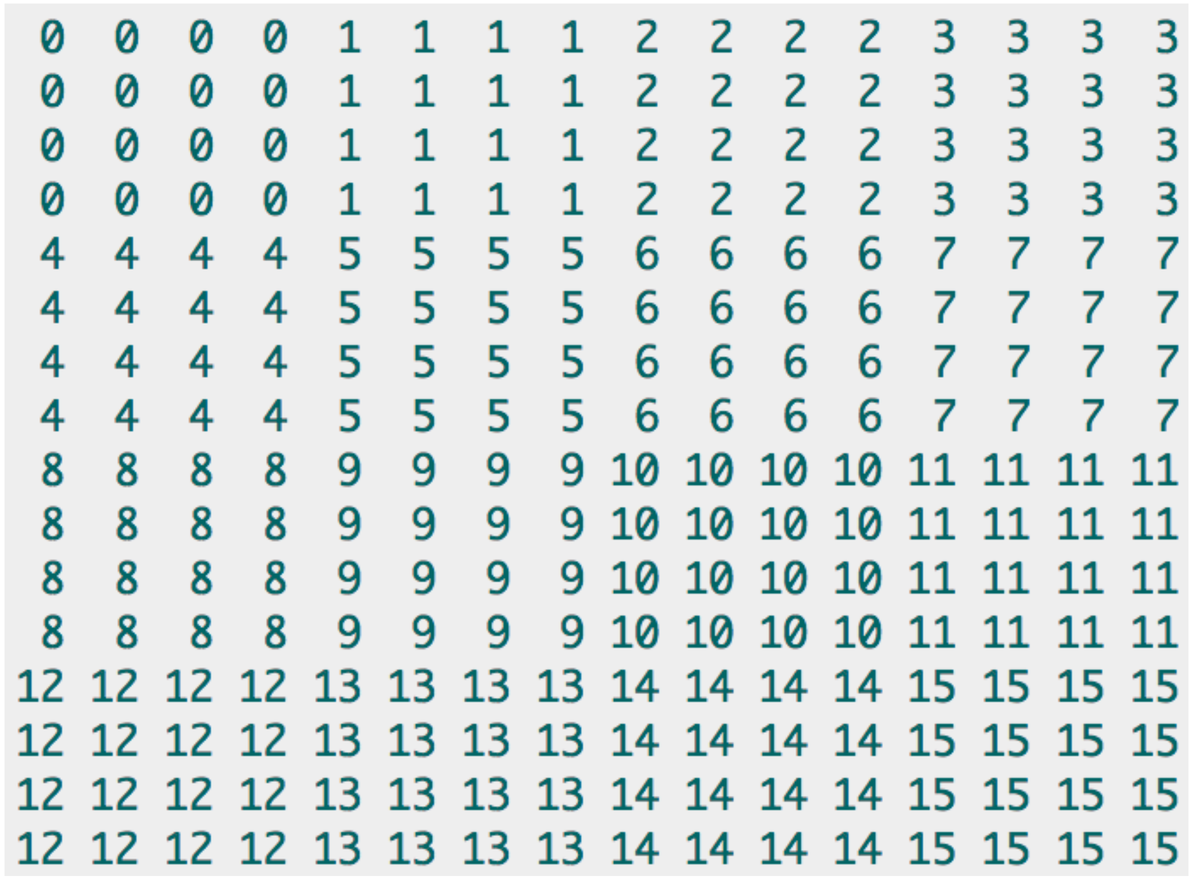
- Kernels are launched in grids
  - One kernel executes at a time
- A thread block executes on one multiprocessor
  - Does not migrate
- Several blocks can reside concurrently on one multiprocessor
  - Number is limited by multiprocessor resources
    - Registers are *partitioned* among all resident threads
    - Shared memory is *partitioned* among all resident thread blocks

CUDA Kernels and Threads

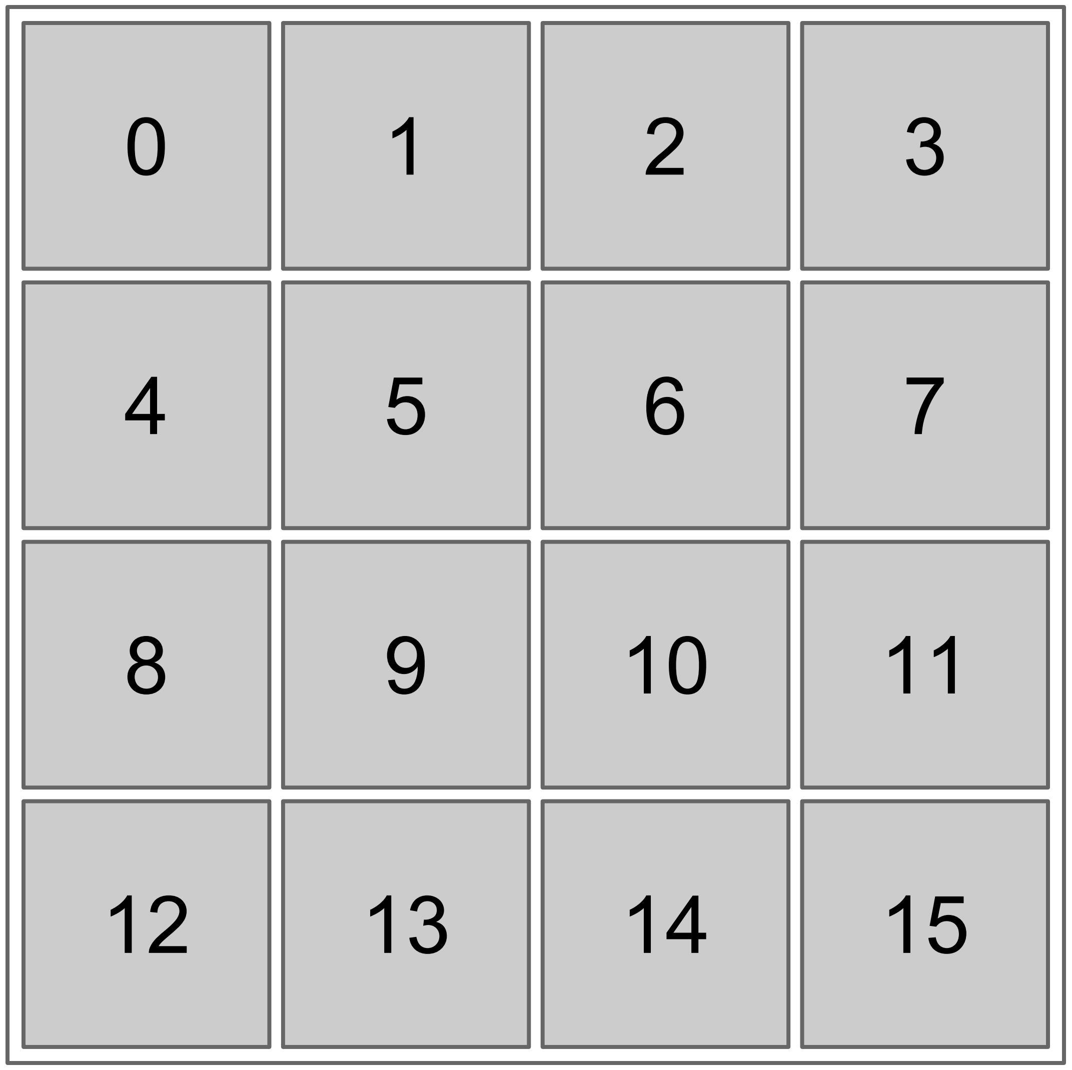
- Parallel portions of an application are executed on the device as **kernels**
  - One **kernel** is executed at a time
  - Many threads execute each **kernel**
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Fast switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can only use a few

Example

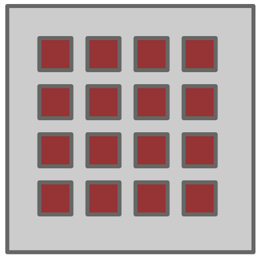
Create a matrix that looks like this:



```
 0  0  0  0  1  1  1  1  2  2  2  2  3  3  3  3
 0  0  0  0  1  1  1  1  2  2  2  2  3  3  3  3
 0  0  0  0  1  1  1  1  2  2  2  2  3  3  3  3
 0  0  0  0  1  1  1  1  2  2  2  2  3  3  3  3
 4  4  4  4  5  5  5  5  6  6  6  6  7  7  7  7
 4  4  4  4  5  5  5  5  6  6  6  6  7  7  7  7
 4  4  4  4  5  5  5  5  6  6  6  6  7  7  7  7
 4  4  4  4  5  5  5  5  6  6  6  6  7  7  7  7
 8  8  8  8  9  9  9  9 10 10 10 10 11 11 11 11
 8  8  8  8  9  9  9  9 10 10 10 10 11 11 11 11
 8  8  8  8  9  9  9  9 10 10 10 10 11 11 11 11
 8  8  8  8  9  9  9  9 10 10 10 10 11 11 11 11
12 12 12 12 13 13 13 13 14 14 14 14 15 15 15 15
12 12 12 12 13 13 13 13 14 14 14 14 15 15 15 15
12 12 12 12 13 13 13 13 14 14 14 14 15 15 15 15
12 12 12 12 13 13 13 13 14 14 14 14 15 15 15 15
```



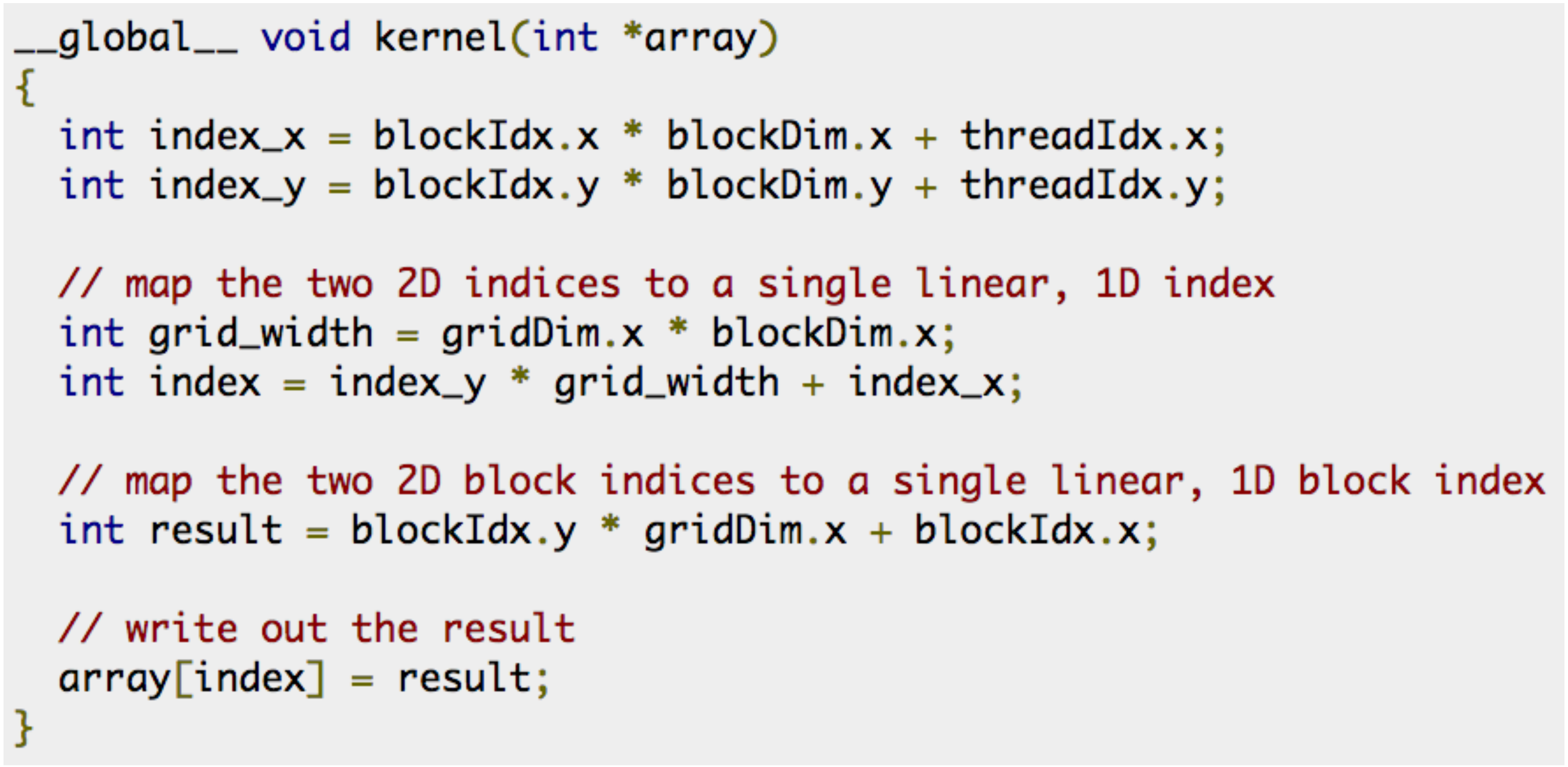
4x4 Grid



4x4 Block

```
int main(void)
{
  int num_elements_x = 16;
  int num_elements_y = 16;

  int num_bytes = num_elements_x * num_elements_y * sizeof(int);

  int *device_array = 0;
  int *host_array = 0;

  // allocate memory in either space
  host_array = (int*)malloc(num_bytes);
  cudaMalloc((void**)&device_array, num_bytes);

  // create two dimensional 4x4 thread blocks
  dim3 block_size;
  block_size.x = 4;
  block_size.y = 4;

  // configure a two dimensional grid as well
  dim3 grid_size;
  grid_size.x = num_elements_x / block_size.x;
  grid_size.y = num_elements_y / block_size.y;

  // grid_size & block_size are passed as arguments to the triple chevrons as usual
  kernel<<<grid_size,block_size>>>(device_array);
```

```
gridDim.x = 4      gridDim.y = 4
blockIdx.x = 0:3   blockIdx.y = 0:3
blockDim.x = 4     blockDim.y = 4
threadIdx.x = 0:3  threadIdx.y = 0:3
grid_width         number of columns of the matrix (16)
index_x, index_y   row/column index in the matrix (0:15, 0:15)
index              matrix element index in 1D array (0:255)
result             submatrix index (0:15)
```



```
__global__ void kernel(int *array)
{
  int index_x = blockIdx.x * blockDim.x + threadIdx.x;
  int index_y = blockIdx.y * blockDim.y + threadIdx.y;

  // map the two 2D indices to a single linear, 1D index
  int grid_width = gridDim.x * blockDim.x;
  int index = index_y * grid_width + index_x;

  // map the two 2D block indices to a single linear, 1D block index
  int result = blockIdx.y * gridDim.x + blockIdx.x;

  // write out the result
  array[index] = result;
}
```

Compilation

```
MyProgram.cu

nvcc MyProgram.cu -o MyProgram

-V              // CUDA toolkit version
-arch=sm_20     // Enable printf
-Xptxas -v      // Kernel memory usage
```

CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - Except for kernel launches
  - cudaError_t type
- cudaError_t cudaGetLastError(void)
  - Returns the code for the last error (no error has a code)
  - Can be used to get error from kernel execution
- char* cudaGetErrorString(cudaError_t code)
  - Returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int *ptr = 0;

  // gimme!
  cudaError_t error = cudaMalloc((void**)&ptr, UINT_MAX);
  if(error != cudaSuccess)
  {
    // print the CUDA error message and exit
    printf("CUDA error: %s\n", cudaGetErrorString(error));
    exit(-1);
  }

  return 0;
}
```

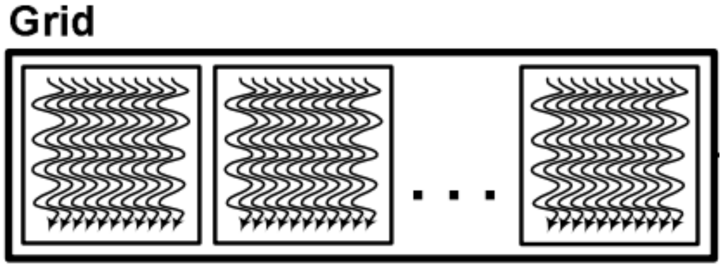
This program outputs a different error message:

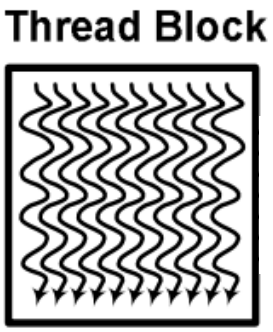
```
$ nvcc big_malloc.cu -o big_malloc
$ ./big_malloc
CUDA error: out of memory
```

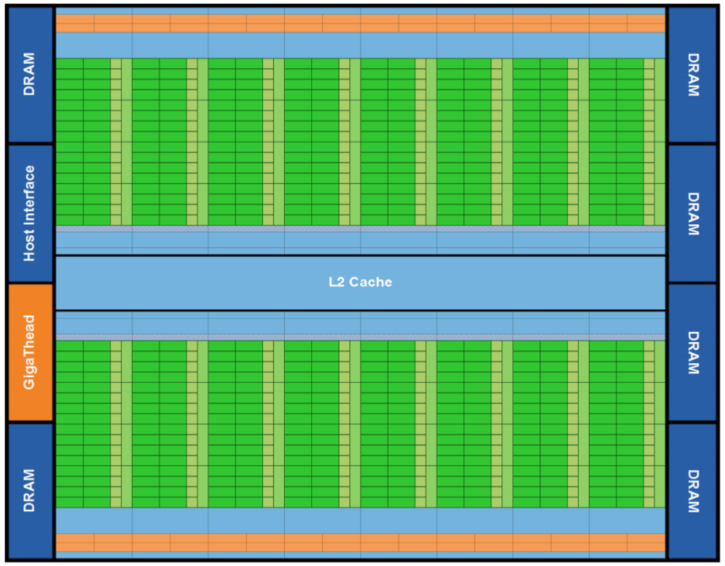
```
__global__ void foo(int *ptr)
{
  *ptr = 7;
}

int main(void)
{
  foo<<<1,1>>>(0);

  // make the host block until the device is finished with foo
  cudaThreadSynchronize();

  // check for error
  cudaError_t error = cudaGetLastError();
  if(error != cudaSuccess)
  {
    // print the CUDA error message and exit
    printf("CUDA error: %s\n", cudaGetErrorString(error));
    exit(-1);
  }

  return 0;
}
```

This example, when compiled, produces the following output:

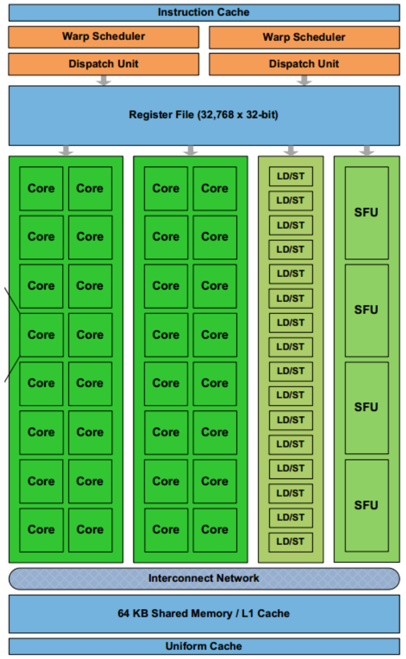
```
$ nvcc check_for_error.cu -o check_for_error
$ ./check_for_error
CUDA error: unspecified launch failure
```



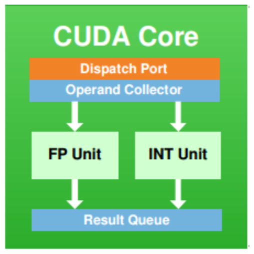




16 SMs



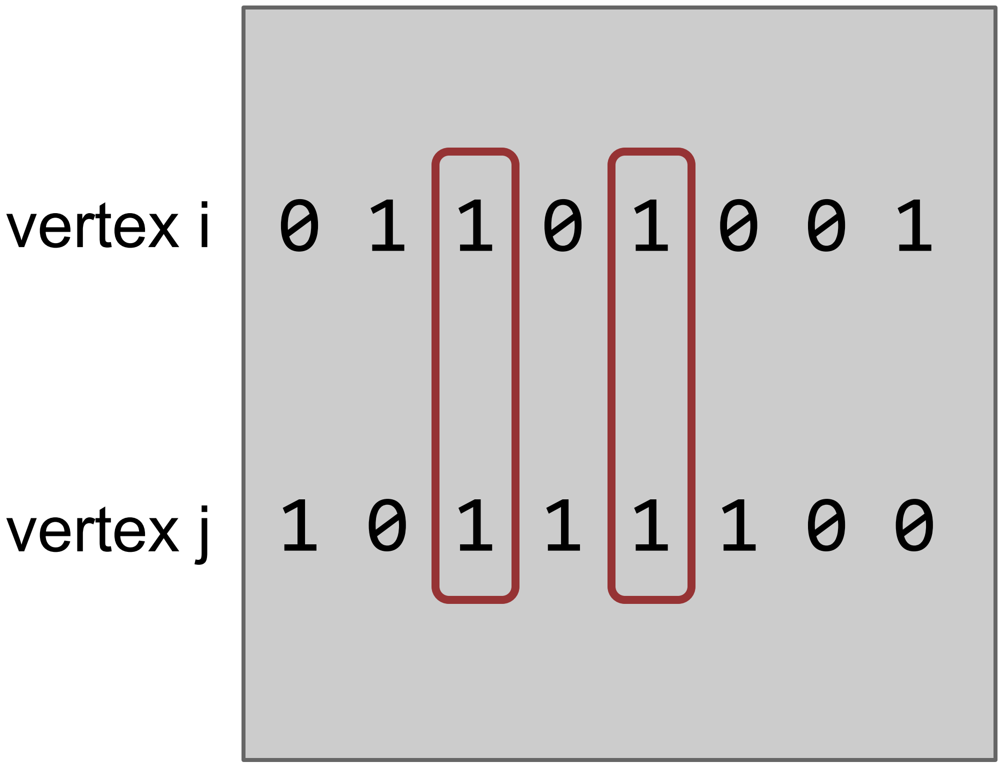
1 SM 32 SPs



1 SP

Example on page 132

Finding the mean number of mutual outlinks



- Adjacency matrix
- `m[i][j] = 1` if there is an edge (outlink) from vertex i to j
- e.g., Vertex i and j (pair(i, j)) have 2 mutual outlinks

Example on page 132

- The thread with global thread index i will count mutual outlinks for all the pairs (i, j) where i < j. i.e., between vertex i and all the vertices with a larger index
- e.g.,
  thread 0 processes (0, 1), (0, 2), …, (0, n-1)
  thread 5 processes (5, 6), (5, 7), …, (5, n-1)

Example on page 132

- line 24:30 Declarations
- line 28, 32 Allocate host memory
- line 34:40 Generate input
- line 42, 47 Allocate device memory
- line 44, 48 Copy input from host to device
- line 50:51 Set launch configurations
- line 53 Launch kernel
- line 57 Copy output from device to host
- line 66 `htot / (n * (n - 1) / 2.0)`
- line 68:70 Release memory

Example on page 132

- line 11:12 Identify the thread itself
- line 14 Set vertex i
- line 15 Set vertex j (j > i)
- line 16:17 Enumerate common vertex k, increment 'sum' if there are i->k and j->k
- line 20 tot += sum using atomic operation

Example on page 132

- number of threads < number of vertices
- (line 14) When totth < n, thread i works for vertex i, i + totth, i + totth * 2, …
- e.g., totth=4, n=10,
  thread 1 works for vertex 1, 5 and 9
  thread 2 works for vertex 2 and 6

Example on page 132

mean = total number of mutual outlinks / number of pairs

- (line 20) All the threads add their counts to the total count, an integer variable in global memory
- atomicAdd (page 127)
- Atomic operation is guaranteed to be performed without interference from other threads
- Avoid race conditions
- Serialization can be quite expensive

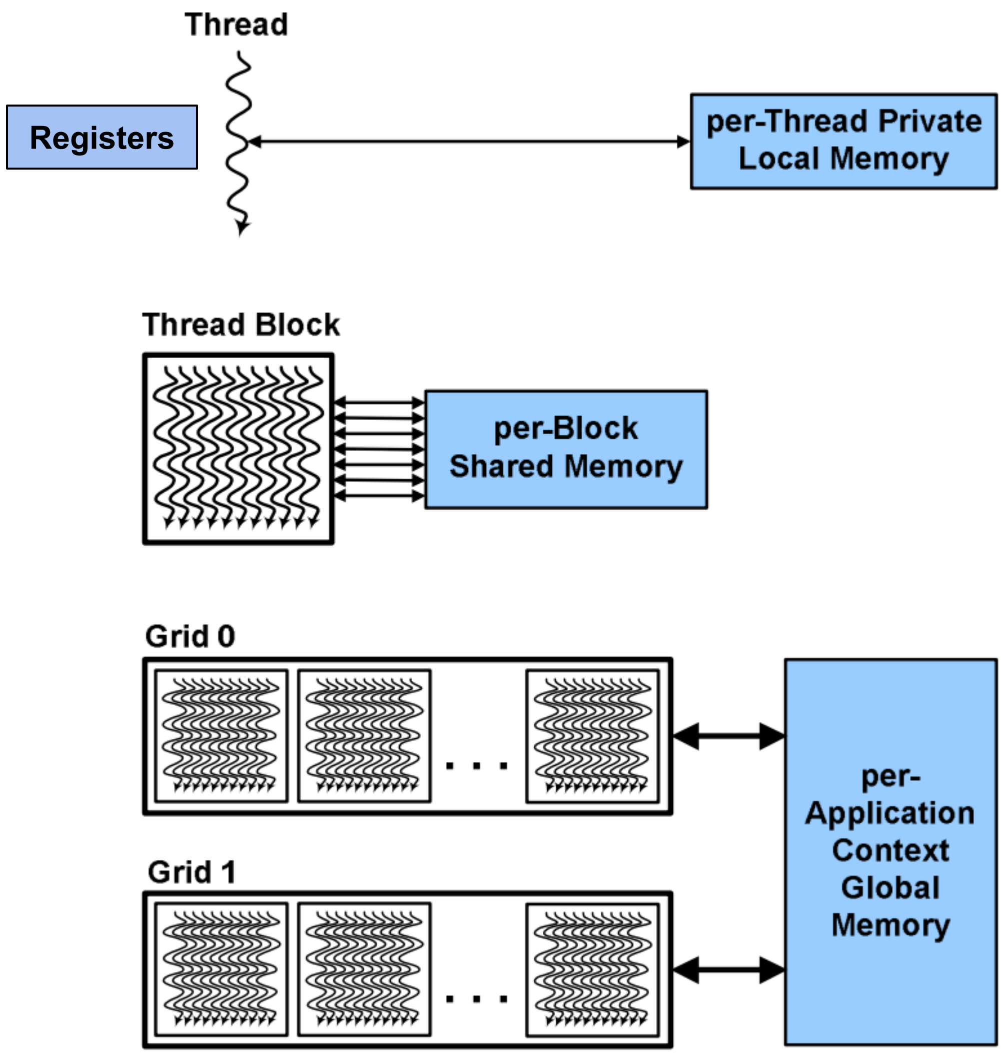
Example of __device__ function

```
__device__ int get_global_index(void)
{
  return blockIdx.x * blockDim.x + threadIdx.x;
}

__device__ int get_constant(void)
{
  return 7;
}

__global__ void kernel1(int *array)
{
  int index = get_global_index();
  array[index] = get_constant();
}

__global__ void kernel2(int *array)
{
  int index = get_global_index();
  array[index] = get_global_index();
}
```

Memory Model

- Registers
  - Per thread
  - Data lifetime = thread lifetime
- Local memory
  - Per thread off-chip memory (physically in device DRAM)
  - Data lifetime = thread lifetime
- Shared memory
  - Per thread block on-chip memory
  - Data lifetime = block lifetime
- Global (device) memory
  - Accessible by all threads as well as host (CPU)
  - Data lifetime = from allocation to deallocation
- Host (CPU) memory
  - Not directly accessible by CUDA threads

Comparisons on page 118 and 123 speed and lifetime


Thread
Registers
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory

Example on page 119-120

Double the values in an array

One value per thread

- Declare, allocate and use shared memory
- Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory

Example on page 119-120

- line 21:24 Declarations **vsize**
- line 26 Allocate host memory
- line 28:30 Generate input
- line 32 Allocate device memory
- line 34 Copy input from host to device
- line 36:37 1 block with n threads
- line 39 Launch kernel <<<dimGrid, dimBlock, **vsize**>>>
- line 43 Copy output from device to host
- line 45 Print output
- line 47:48 Release memory

Example on page 119-120

- line 11 Declare the shared memory array (sv) using an unsized extern array syntax. The size is determined from the third launch configuration parameter (vsize).
- line 12 Identify the thread itself
- line 15 Read the value assigned to this thread from global memory (dv), double it, and write the result to shared memory (sv)
- line 16 Read the result from shared memory and write it to global memory

Two ways to declare/allocate arrays in shared memory

- Dynamic shared memory
  - Previous example
  - Use the optional third kernel launch configuration parameter to specify the size
- Static shared memory
  - Size is known at compile time
  - (page 119) `__shared__ int abcsharedmem[100];`

Using shared memory

Size known at compile time

```
__global__ void kernel(…)
{
   …
   __shared__ float sData[256];
   …
}

int main(void)
{
  …
  kernel<<<nBlocks,blockSize>>>(…);
  …
}
```

Size known at kernel launch

```
__global__ void kernel(…)
{
   …
   extern __shared__ float sData[];
   …
}

int main(void)
{
   …
   smBytes = blockSize*sizeof(float);
   kernel<<<nBlocks, blockSize,
       smBytes>>>(…);
   …
}
```

GPU Thread Synchronization

-
- `void __syncthreads();`
- Synchronizes all threads in a block
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory

Example on page 137

Finding Cumulative Sums (inclusive prefix sum)

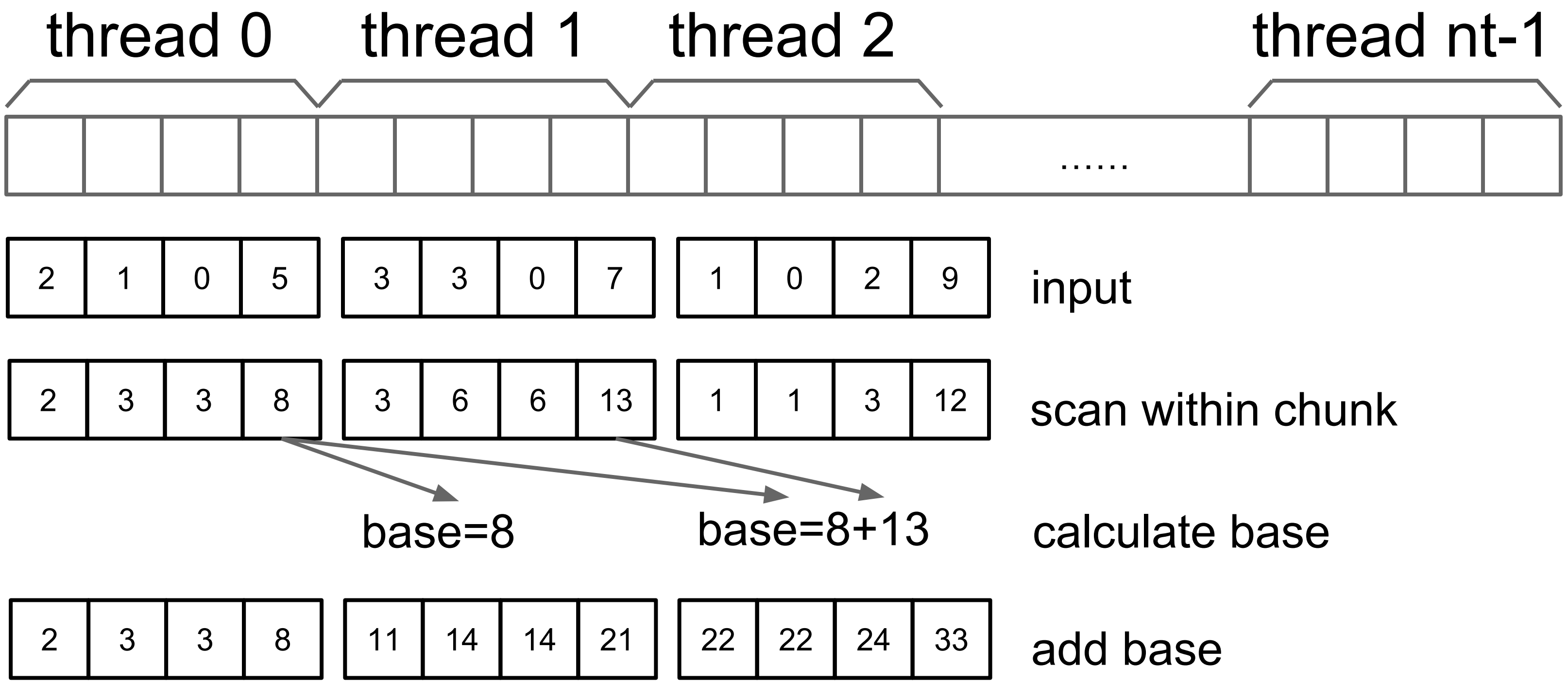
output[k] = input[0] + input[1] + … + input[k]

e.g.,

```
input:  3  1  2  0  3  0  1  2
output: 3  4  6  6  9  9 10 12
```

Example on page 137

Launch a single block and csize=4



Example on page 137

- line 15 identify the thread itself
- line 16 'csize' chunk size
- line 17 starting position of the chunk assigned to this thread
- line 19:22 Calculate prefix sum within chunk
- line 24:28 Calculate 'base', which is the sum of the chunks in front of this chunk
- line 30:33 Add 'base' to this chunk
- line 23, 29 Sync among threads to eliminate Read After Write hazards

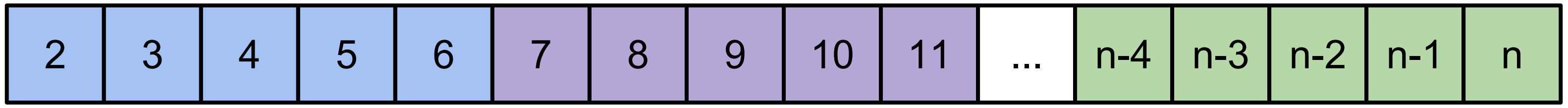
Example on page 134

Finding Prime Numbers

Build a table where
isprime[k] = 1 if k is a prime number
isprime[k] = 0 otherwise

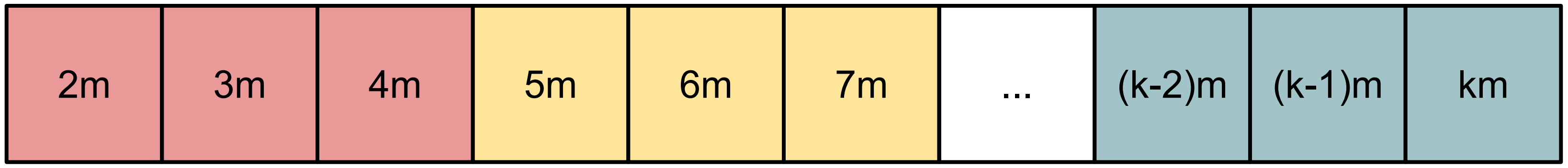
Example on page 134

Initialization (1 for odds, 0 for evens)

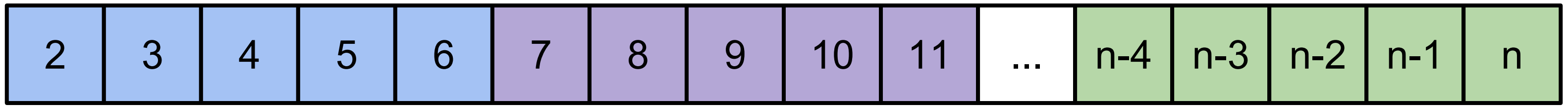


Cross out multiples for each prime

For prime 'm'



Copy the table from shared memory to global memory



Example on page 134

- line 83 Find primes among 1, …, 'n'
- line 84 'nth' Number of threads
- line 85:86 'hprimes' and 'dprimes' are isprime tables on host and device
- line 87 Shared memory size 'psize'
- line 92:95 Launch the kernel with a single block having 'nth' threads and 'psize' bytes shared memory
- line 97:100 cudaThreadSynchronize should be called before cudaGetLastError
- sprimes -> dprimes -> hprimes

Example on page 134

- line 54 Declare 'sprimes'
- line 58 Call 'initsp' (jump to line 17)
- line 22 'chunk' chunk size
- line 23 'startsetsp' starting position of this chunk
- line 24:25 'endsetsp' ending position of this chunk
- line 27:31 Initialize within chunk, 1 for odds and 0 for evens
- line 33 Sync (then jump to line 59)

Example on page 134

- line 63:64 Get the next prime number 'm'
- line 66 'maxmult' number of multiples of 'm'
- line 68 'chunk' chunk size
- line 69 'startmult' starting position
- line 70:71 'endmult' ending position
- line 74 Cross out multiples assigned to this thread
- line 72 ?
- line 76 ?
- line 78 Call 'cpytoglb' (jump to line 38)

Example on page 134

- line 41 'chunk' chunk size
- line 42 'startcpy' starting position
- line 43:44 'endcpy' ending position
- line 45 Copy from shared memory to global memory
- line 46 ?

Example on page 138

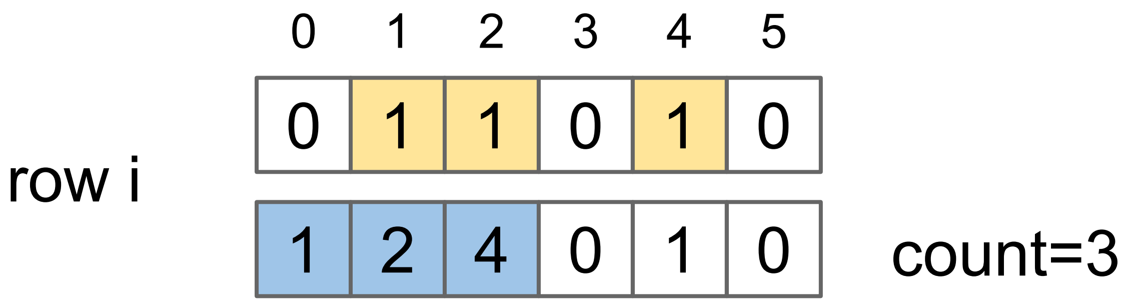
Transforming an Adjacency Matrix

input: n by n adjacency matrix where
`adjm[n*i+j] = 1` if edge i->j exists
`adjm[n*i+j] = 0` otherwise

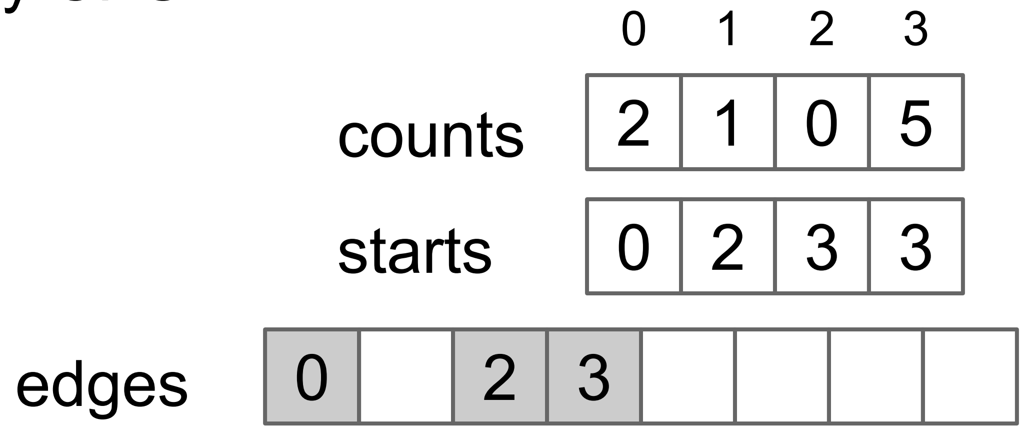
output: nout by 2 matrix where 'nout' is the number of edges and the edge x is from vertex `outm[2*x]` to `outm[2*x+1]`

Example on page 138

Count the number of edges started from each vertex
One thread per vertex(row)

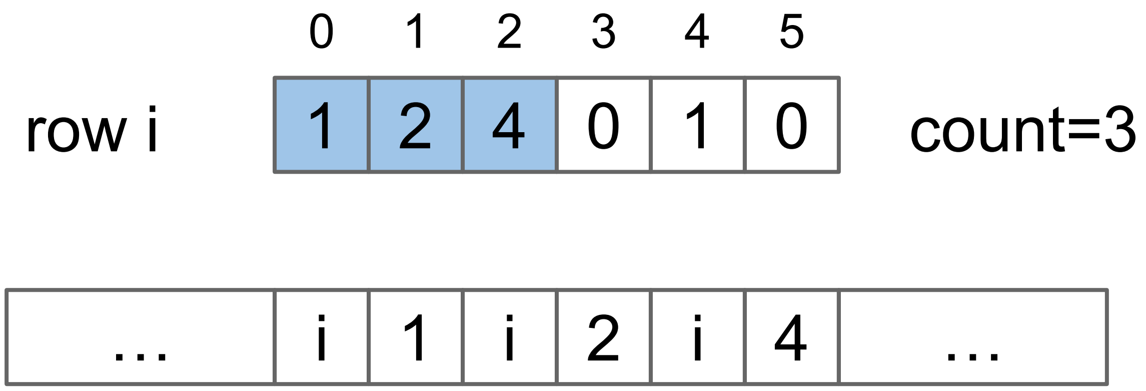


Compute starting positions in output array for each vertex
Done by CPU



Example on page 138

Write the starting vertex and ending vertex for each edge
One thread per vertex(row)



Example on page 138

- line 127 Call 'transgraph' (jump to line 68)
- line 84:85 `gsize * bsize = n`
- line 87 Launch tgkernel1 (jump to line 31)
- line 33 row 'me', vertex 'me'
- line 34:38 Count the number of edges and write target vertex indices (overwrite 'dadjm')
- line 39 Write 'dcounts[me]', the number of edges started from vertex 'me' (jump to line 89)
- line 90 Call 'cumulcounts' (jump to line 60)

Example on page 138

- line 61:65 Compute the exclusive prefix sums of 'hcounts' and save them as 'hstarts'
- line 91 '*nout' the total number of edges
- line 93 Launch tgkernel2 (jump to line 43)
- line 49 'outrow' starting position in 'doutm' for edges started from vertex 'me'
- line 50 'num1si' number of edges started from vertex 'me'
- line 51:56 Write the edges `(me, dadjm[n*me+j])`

GPU Spec

```
./deviceQuery Starting...
...
Detected 1 CUDA Capable device(s)
...
Device 0: "GeForce GTX 460"
  CUDA Driver Version / Runtime Version          6.5 / 6.5
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1023 MBytes (1072889856 bytes)
  ( 7) Multiprocessors, ( 48) CUDA Cores/MP:     336 CUDA Cores
...
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
...
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
...
```

Reference

- http://docs.nvidia.com/cuda/
- http://www.sdsc.edu/us/training/assets/docs/NVIDIA-02-BasicsOfCUDA.pdf
- https://code.google.com/p/stanford-cs193g-sp2010/wiki/ClassSchedule
- http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf