Lecture 11: Programming on GPUs (Part 1)

Overview

- GPGPU: General purpose computation using graphics processing units (GPUs) and graphics API
- GPU consists of multiprocessor element that run under the shared-memory threads model. GPUs can run hundreds or thousands of threads in parallel and has its own DRAM.
  - GPU is a dedicated, multithread, data parallel processor.
  - GPU is good at
    - Data-parallel processing: the same computation executed on many data elements in parallel
    - with high arithmetic intensity

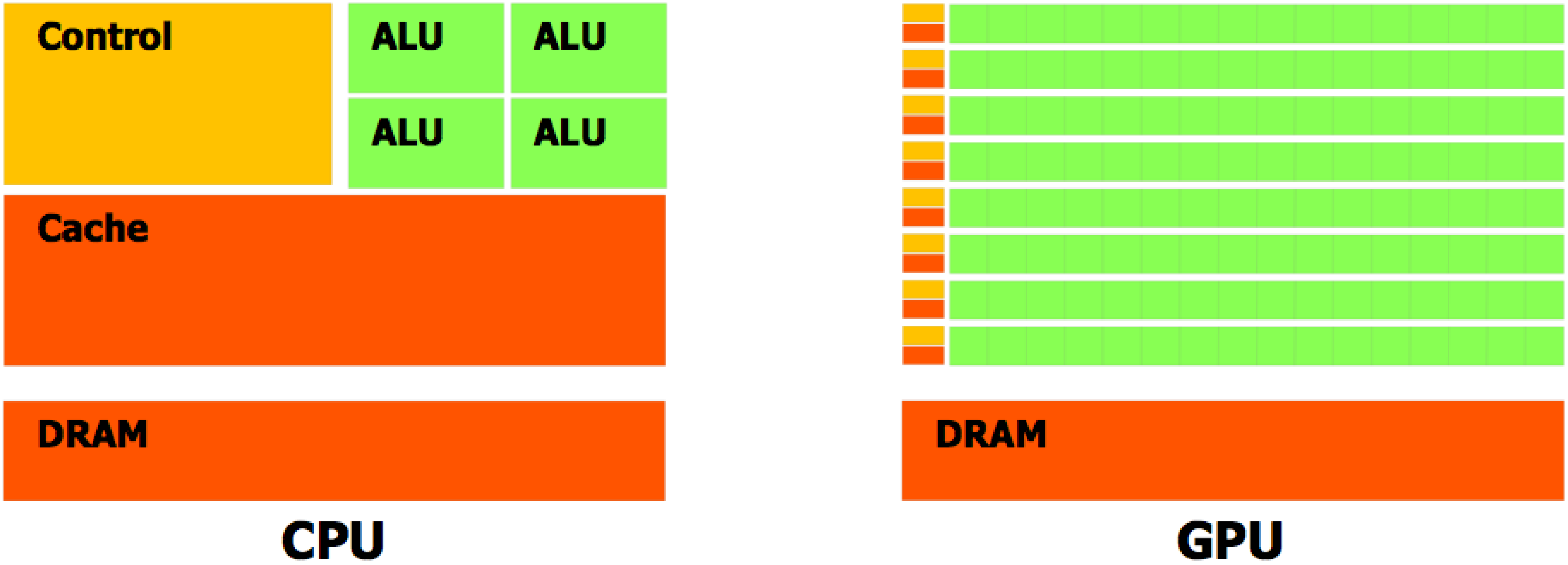
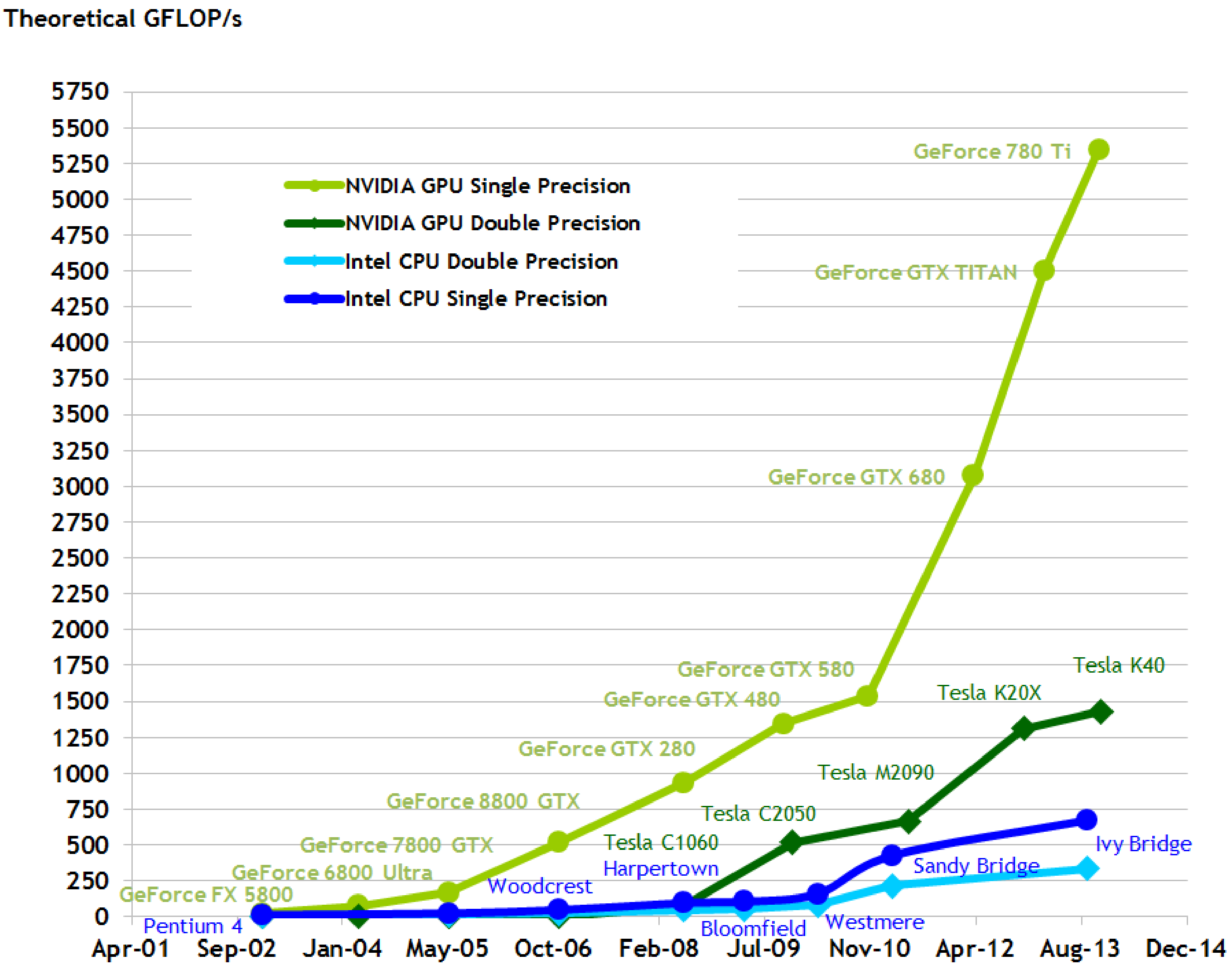


Figure 1-2. The GPU Devotes More Transistors to Data Processing

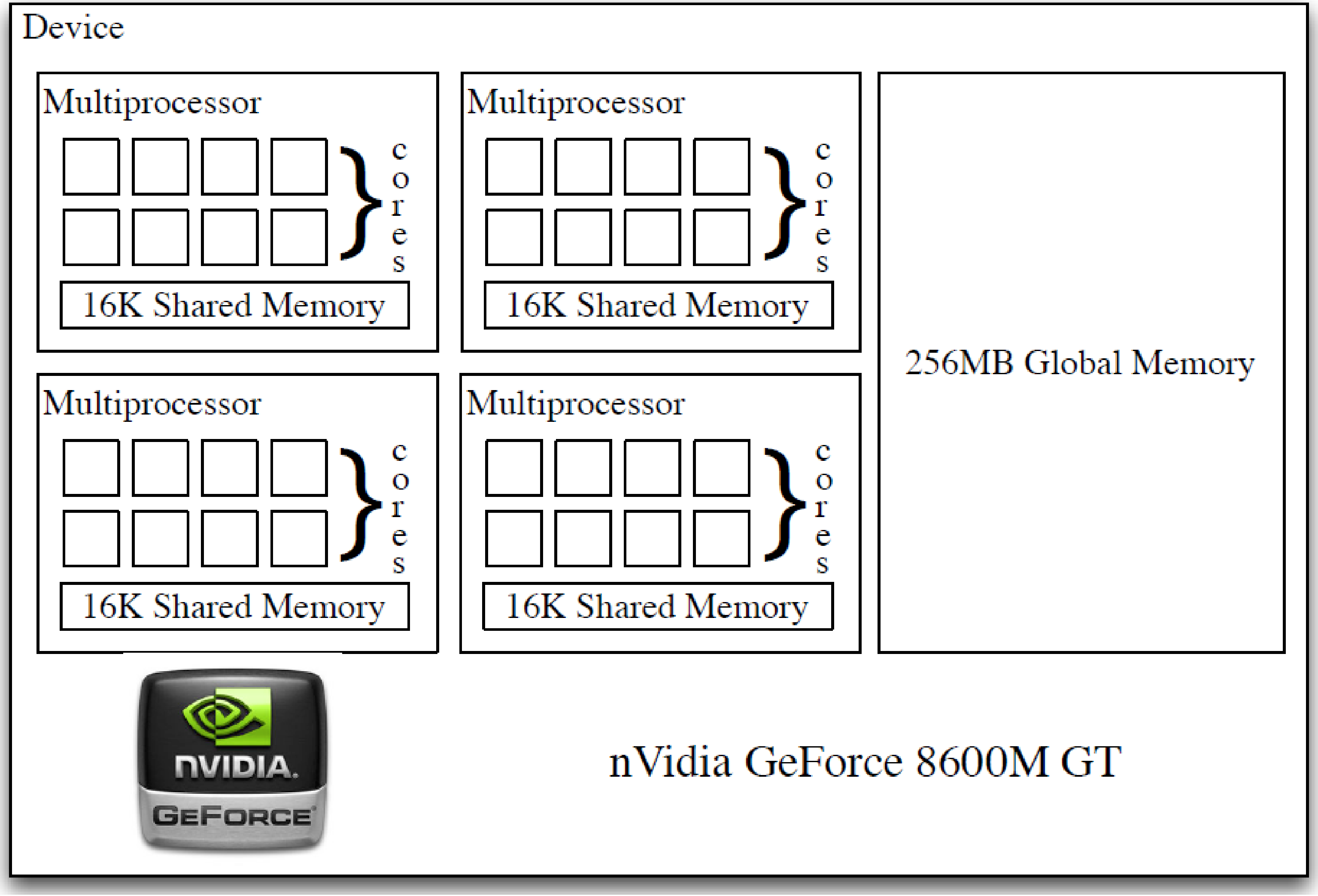
- Performance history: GPUs are much faster than CPUs



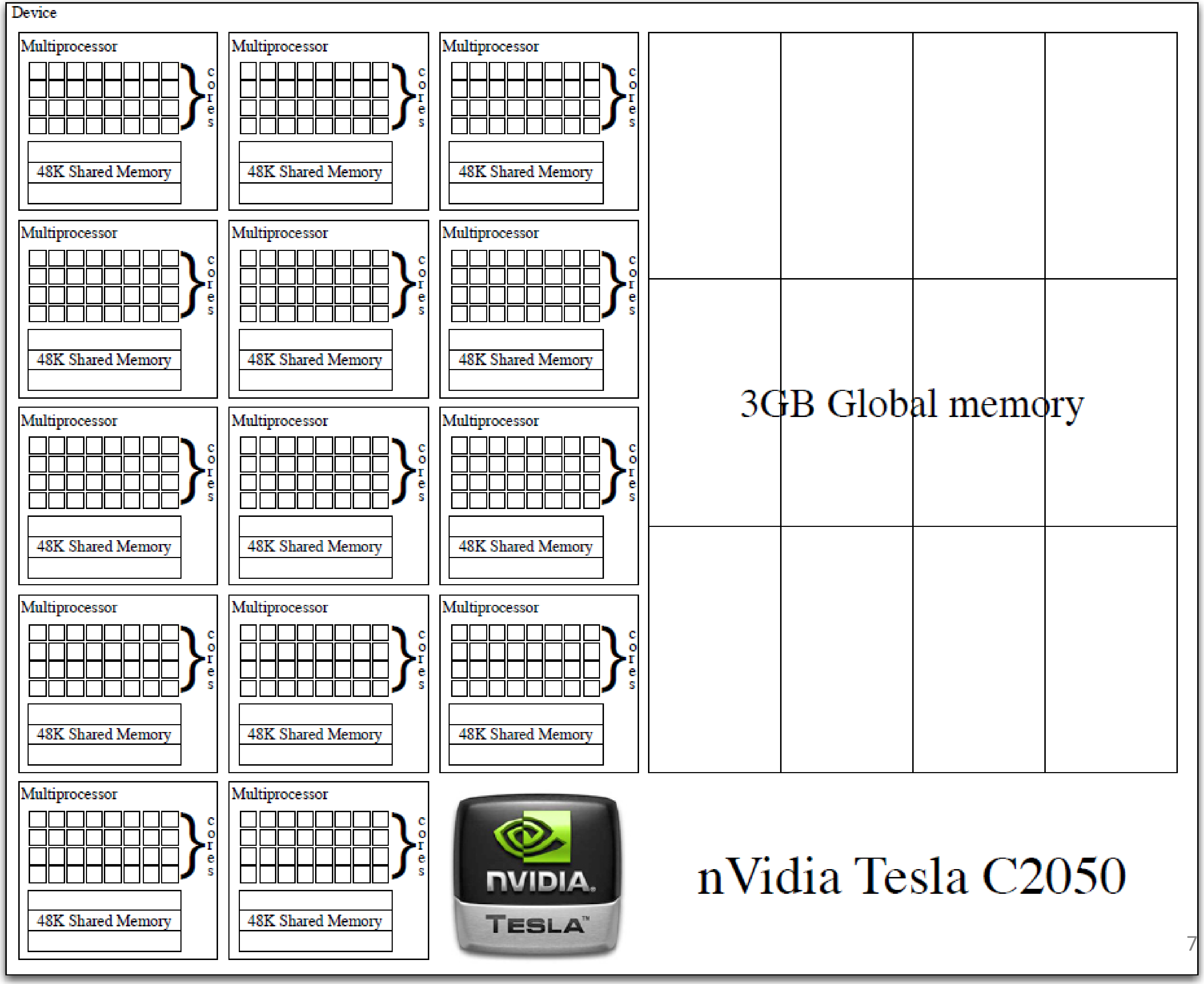
http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3X9Fwos00

- ✓ CUDA: Compute unified device architecture
  - – A new hardware and software architecture for issuing and managing computations on the GPU
  - – CUDA C is a programming language developed by NVIDIA for programming on their GPUs. It is an extension of C.
- OpenGL (Open Graphics Library)

nVidia GPU Architecture



- Many processors are striped together
- Small, fast shared memory



nVidia Tesla C2050

Hardware Overview

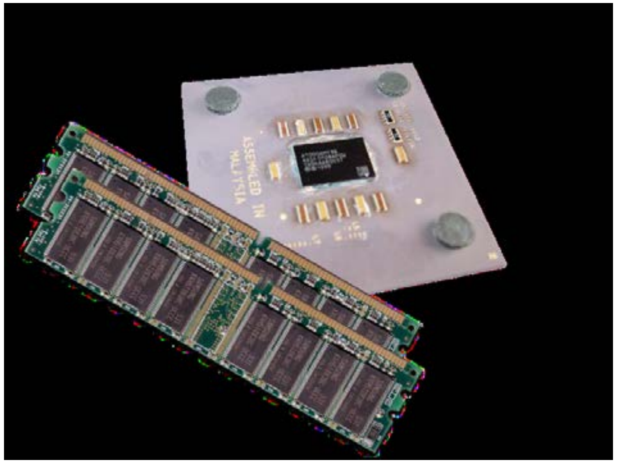
- Basic building block is a “streaming multiprocessor” (SM) with:
  - 32 cores, each with 1024 registers
  - up to 48 threads per core
  - 64KB of shared memory / L1 cache
  - 8KB cache for constants held in device memory
- C2050: 14 SMs, 3/6 GB memory
- Geforce GTX 780: 2,304 cores, 3GB memory

GPU Computing at CRC

- http://wiki.crc.nd.edu/wiki/index.php/Developmental_Systems
- gpu1.crc.nd.edu
- gpu2.crc.nd.edu
- gpu3.crc.nd.edu
- gpu4.crc.nd.edu
- gpu5.crc.nd.edu
- CUDA compiler is nvcc
- To compile and run GPU code:
  - module load cuda
  - module show cuda
  - nvcc hello.cu

Heterogeneous Computing

- Host: The CPU and its memory (host memory)
- Device: The GPU and its memory (device memory)



Things to learn:

1. Write code for the host and code for the device
2. Run device code from the host
3. Use device memory (transfer data between host and device)

A First Program

```
/* Cuda Hello, World, hello.cu"
#include <stdio.h>

__global__ void mykernel(void) {
}

int main(void){
   mykernel<<<1,1>>>();
   printf("Hello, World\n");
   return 0;
}
```

__global__ :

1. A qualifier added to standard C. This alerts the compiler that a function should be compiled to run on a device (GPU) instead of host (CPU).
2. Function mykernel() is called from host code.

Compile: nvcc hello.cu

nvcc separates source code into host and device components

Device functions (e.g. mykernel()) processed by NVIDIA compiler

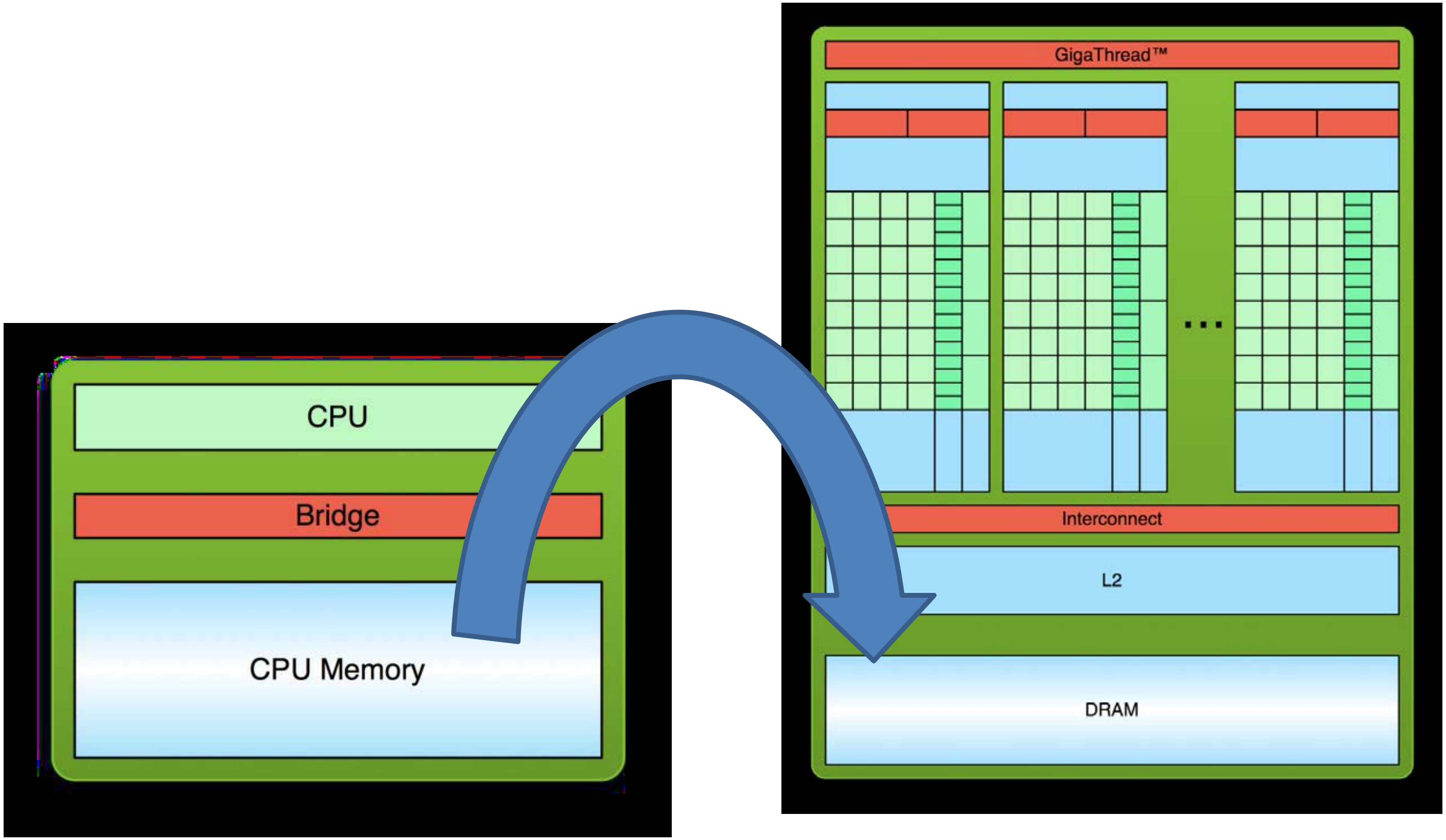
Host functions (e.g. main()) processed by standard host compiler like gcc

CUDA Concepts and Terminology

mykernel<<<1,1>>>();

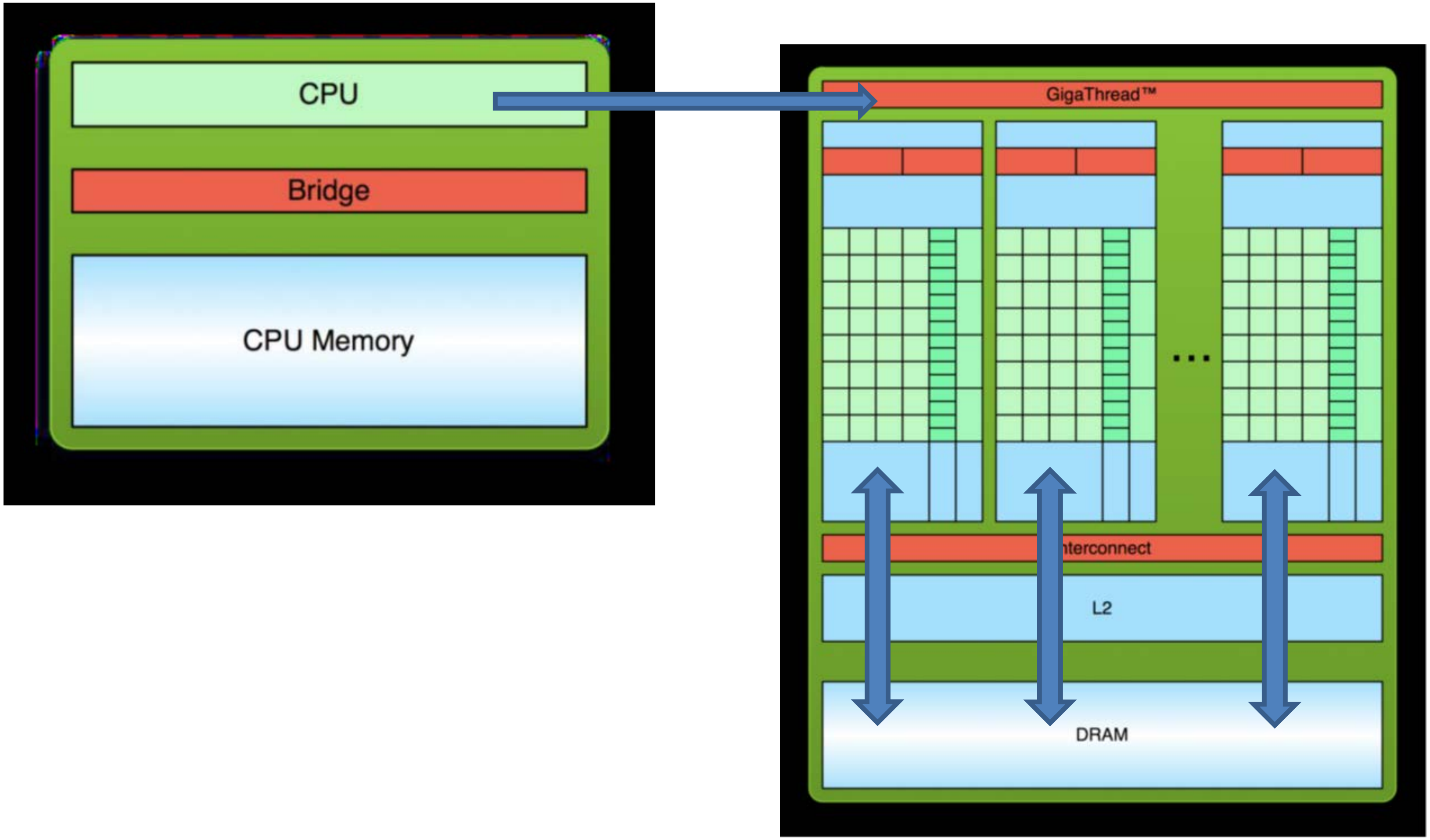
- **Kernel:** a C function which is flagged to be run on a GPU (or a device).
- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
  - The parameters (1,1) will be explained in a moment

Processing Flow



1. Copy input data from CPU memory to GPU memory and allocate memory

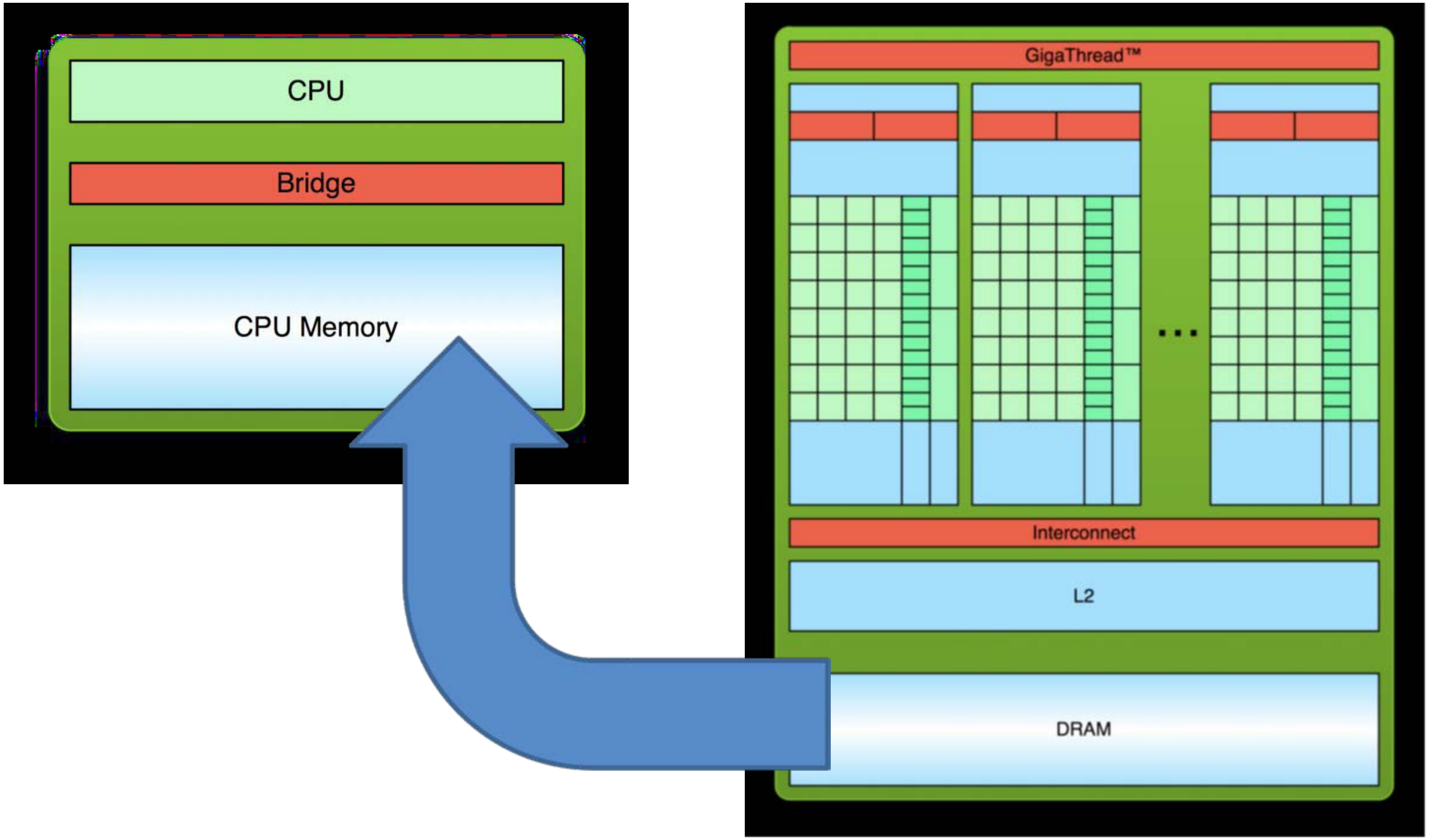
// cudaMalloc((void**)&device_c, sizeof(int));



2. Load GPU program and execute,

Caching data on chip for performance

//add<<<1, 1>>>(2, 7, device_c);



3. Copy results from GPU memory to CPU memory

```
//cudaMemcpy(&c, device_c, sizeof(int), cudaMemcpyDeviceToHost);
```

Passing Parameters & Data Transfer

```
// File name: add.cu
#include <stdio.h>

__global__ void add(int a, int b, int *c){
    *c = a+b;
}

int main(void){
    int c;
    int  *device_c;

    cudaMalloc((void**)&device_c, sizeof(int));
    add<<<1, 1>>>(2, 7, device_c);
    cudaMemcpy(&c, device_c, sizeof(int), cudaMemcpyDeviceToHost);

    printf(“2+7 = %d\n”, c);
    cudeFree(device_c);
    return 0;
}
```

- Can pass parameters to a kernel as with C function
- Need to allocate memory to do anything useful on a device, such as return values to the host.

- add() runs on the device, so device_*c* must point to the device memory
  - This is why we call cudaMalloc() to allocate memory on the device
- Do not deference the pointer returned by cudaMalloc() from code that executes on the host. Host code may pass this pointer around, perform arithmetic on it. But we can not use it to read or write from memory.
  - Its C equivalent is malloc().
- We can access memory on a device through calls to cudaMemcpy() from host code.
  - Its C equivalent is memcpy().

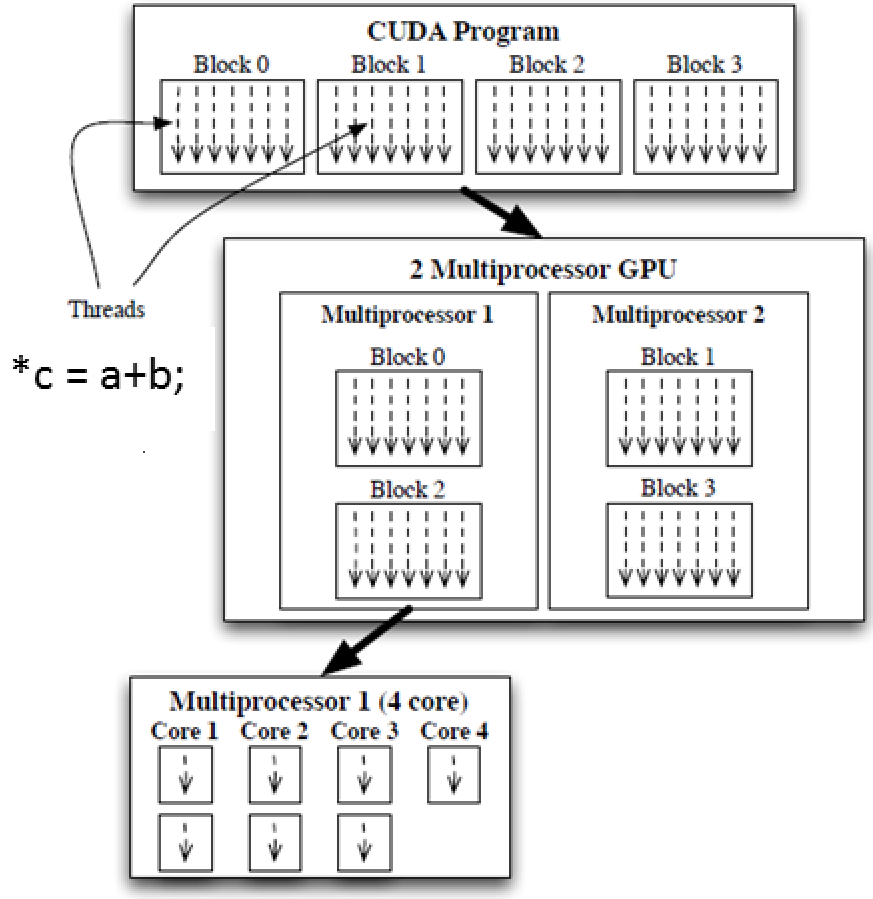
Parallel Computing

- How do we run code in parallel on the device?

add<<< 256, 1>>>(2, 7, device_c);

  - Instead of executing add() once, execute 256 times in parallel

- <<<N,1>>>();
  - The number “N” represents the number of parallel blocks (of threads) in which we would like the GPU to execute our kernel.
  - add<<< 256, 1>>>() can be thought as that the runtime creates 256 copies of the kernel and runs them in parallel.



*c = a+b;

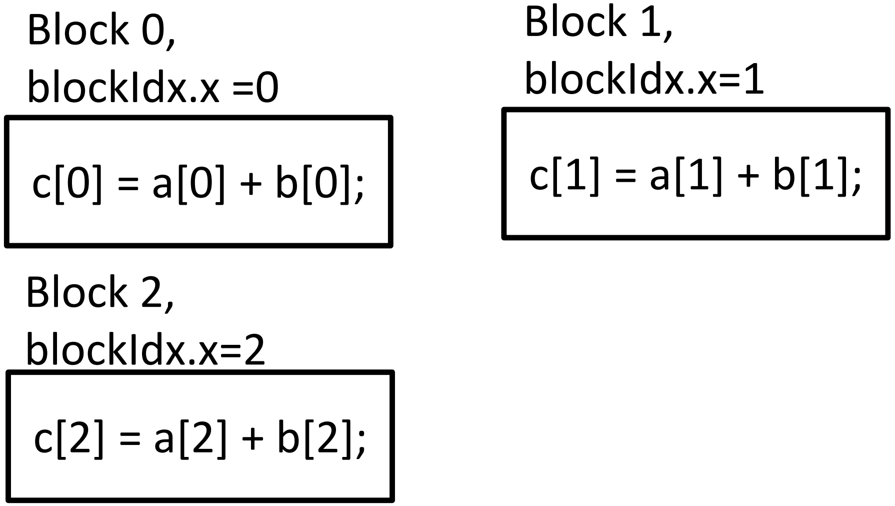
- A kernel is executed by an array of threads
- Threads are organized into blocks; blocks are organized into grids.

Built-in variable “blockIdx”

- How to tell within the code which block is currently running?
- Suppose we add vectors a[] and b[].

```
__global__ void add(int *a, int *b, int *c){
   c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

1. The set of blocks is referred to as a grid.
2. Each invocation can refer to its block index using blockIdx.x
3. By using blockIdx.x to index into the array, each block handles a different index
4. On the device, each block executes in parallel and looks like the following:



Block 0,
blockIdx.x =0

```
c[0] = a[0] + b[0];
```

Block 1,
blockIdx.x=1

```
c[1] = a[1] + b[1];
```

Block 2,
blockIdx.x=2

```
c[2] = a[2] + b[2];
```

GPU Vector Sums (Block Version)

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>

#define N 512

__global__ void add(int *a, int *b, int *c){
    int tid = blockIdx.x;  // handle the data at this index

    if(tid < N)
        c[tid] = a[tid] + b[tid];
}

int main()
{
    int a[N], b[N], c[N], i;
    int *dev_a, *dev_b, *dev_c;

    cudaMalloc((void**)&dev_c, N*sizeof(int));
    cudaMalloc((void**)&dev_b, N*sizeof(int));
    cudaMalloc((void**)&dev_a, N*sizeof(int));
    for(i=0; i < N; i++)
    {
        a[i] = -i;
        b[i] = i*i*i;
    }
    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

    add <<<N, 1>>>(dev_a, dev_b, dev_c);
    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
    for(i=0; i < N; i++)
        printf("%d + %d = %d\n", a[i], b[i], c[i]);

    cudaFree(dev_c);
    cudaFree(dev_b);
    cudaFree(dev_a);
    return 0;
}
```

- CUDA built-in variable: blockIdx
  - CUDA runtime defines this variable.
  - It contains the value of the block index for whichever block is currently running the device code.
  - CUDA C allows to define a group of blocks in one-, two- or three-dimensions (version 2.x above).
- $N -$ specified as the number of parallel blocks per dimension
  - A collection of parallel blocks is called a **grid**.
  - This example specifies to the runtime system to allocate a one-dimensional grid of N blocks.
  - Threads will have different values for blockIdx.x, from 0 to $N - 1$.
  - $N \leq 65{,}535 -$ a hardware-imposed limit ($N \leq 2^{31}$-1 from version 3.x and above).
- if(tid< N)
  - Avoid potential bugs – what if # threads requested is greater than N?

CUDA Threads

- A block can be split into parallel threads
- Using blocks:

```
__global__ void add(int *a, int *b, int *c){
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Using threads all in one block:

```
__global__ void add(int *a, int *b, int *c){
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

main(){
    …
    add<<<1, 100>>>(dev_a, dev_b, dev_c);
}
```

Combining Blocks and Threads

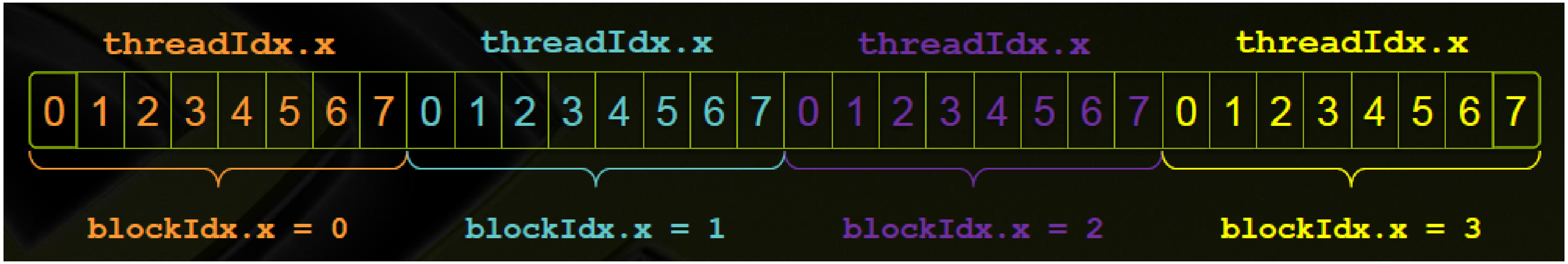
- Hardwire limits the number of blocks in a single launch to 65,535.
- Hardwire also limits the number of threads per block with which we can launch a kernel.
  - For many GPUs, maxThreadsPerBlock = 512 (or 1024, version 2.x above).
- Blocks and threads are often combined.

To parallelize a for loop:

for(int i=0; i < 1000000; i++) {a[i]=x[i];}

- In block/thread, we would like to have a single block/1000000 thread ($i = 0, j = 0, \ldots, 999999$) kernels containing: a[thread_index] = x[thread_index];
- In real implementation, the exact same kernel is called blocks × threads times with the block and thread indices changing.
  - To use more than one multiprocessor, say $i = 0, \ldots, 19, j = 0, \ldots, 49$ and kernel: a[block_index+thread_index]=x[block_index+thread_index];

- Vector addition to use both blocks and threads
  - We no longer simply use either blockIdx.x or threadIdx.x
  - Consider indexing an array with one element per thread
  - We also use 8 threads per block.



1. With "M" threads/block a unique index for each thread is given by:
   int index = threadIdx.x + blockIdx.x*M;
2. Use the built-in variable blockDim.x for threads/block
   int index = threadIdx.x + blockIdx.x*blockDim.x;

- New version of add() to use both threads and blocks

```
__global__ void add(int *a, int *b, int *c) {
int index = threadIdx.x + blockIdx.x * blockDim.x;
c[index] = a[index] + b[index];
}
```

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512

int main(void) {
   …
   // Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(dev_a, dev_b, dev_c);
}
```

For Vector with Arbitrary Sizes

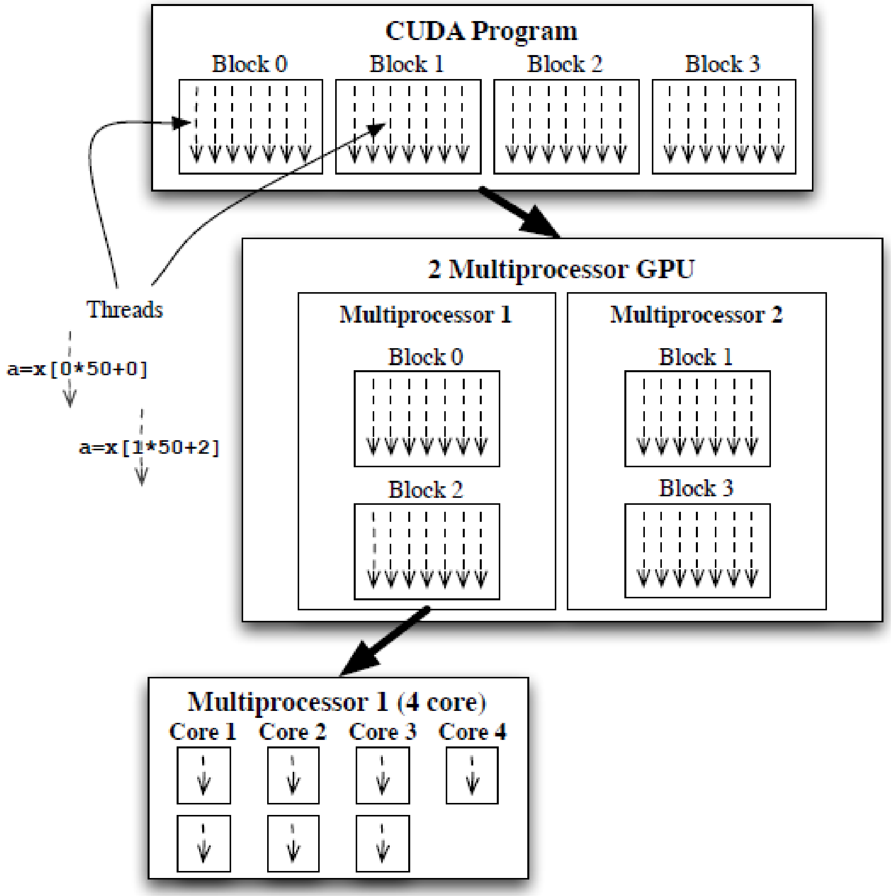
- Problems often are not multiples of blockDim.x
- To avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- See code vec_add_ver2.cu
- Update the kernel launch:

```
    Add<<<(N+M-1)/M, M>>>(dev_a, dev_b, dev_c, N);
```

- Remark:
  - Threads add a level of complexity, why we need them?
  - Unlike parallel blocks, threads have mechanisms to:
    - Communicate
    - Synchronize

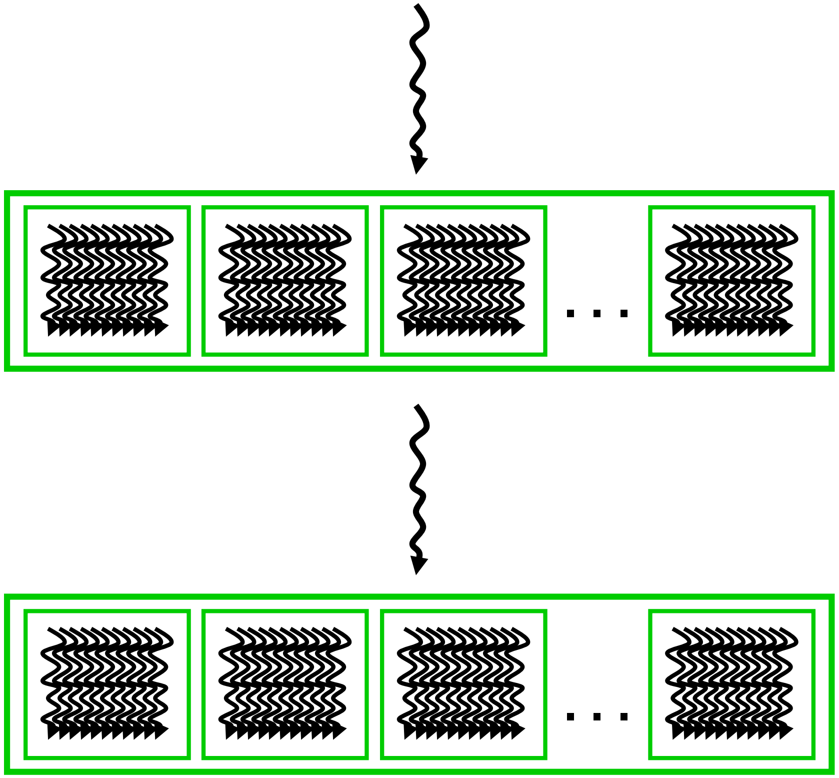


- We can not assume threads will complete in the order they are indexed.
- We can not assume blocks will complete in the order they are labeled.
- To deal with data/task dependency:
  - Use synchronization: __syncthreads();
  - Split into kernels and call consecutively from C
- Shared memory model: do not write to same memory location from different threads

Review: CUDA Programming Model

- A CUDA program consists of code to be run on the **host**, i.e. the CPU, and the code to be run on the **device,** i.e. the GPU.
  - Device has its own DRAM
  - Device runs many threads in parallel
- A function that is called by the host to execute on the device is called a **kernel**.
  - Kernels run on many threads which realize data parallel portion of an application
- Threads in an application are grouped into **blocks**. The entirety of blocks is called the **grid** of that application.

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SIMD kernel C code



Serial Code (host)

Parallel Kernel (device)

KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)

KernelB<<< nBlk, nTid >>>(args);

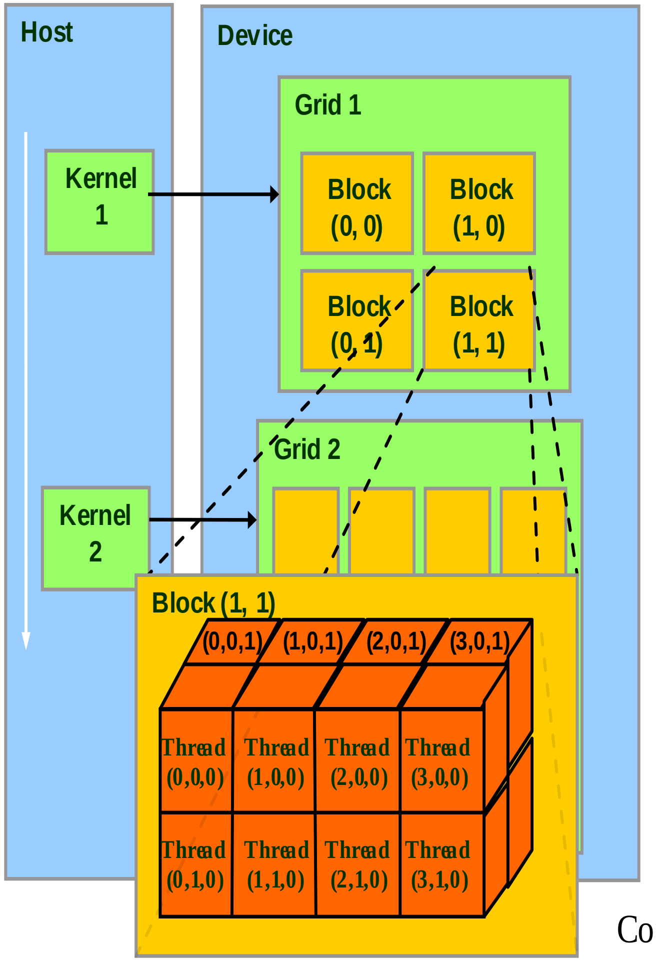
Extended C

- **Type Qualifiers**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function(kernel) launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

Thread Batching

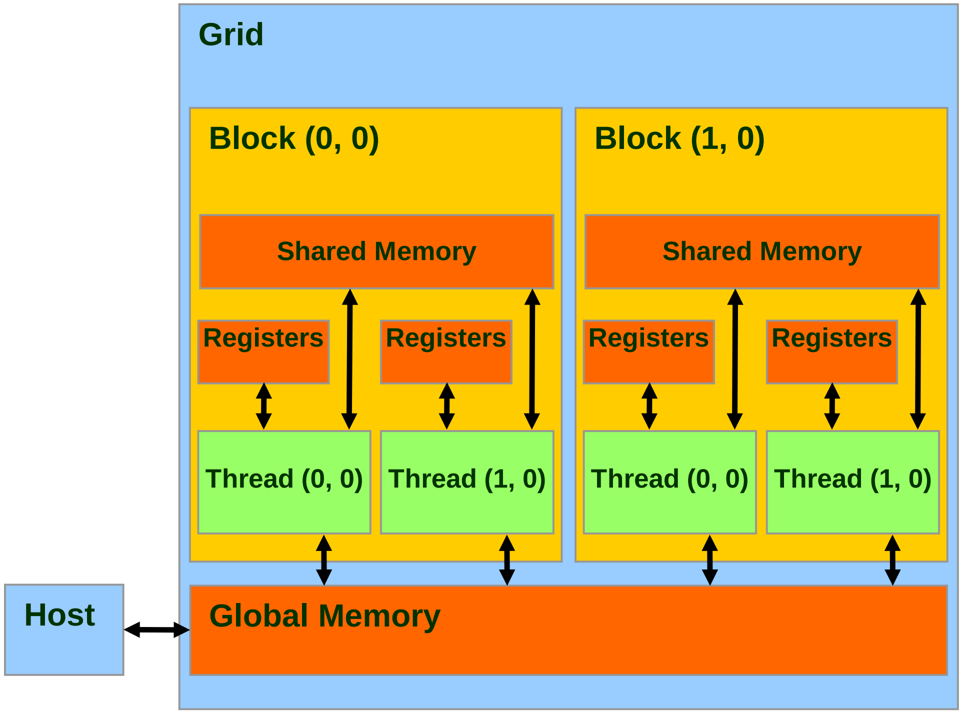
- A kernel is executed as a **grid of thread blocks**
- A **thread block** is a batch of threads that can cooperate.
- Each thread uses **ID** to decide what data to work on
  - Block ID: 1D or 2D (or 3D from version 2.x)
  - Thread ID: 1D, 2D or 3D
- Threads within a block coordinate by shared memory, atomic operations and barrier synchronization.
- Threads in different blocks can not cooperate.
- Convenient for solving PDEs on grid cells.



Courtesy: NDVIA

CUDA Memory Model

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access



Device Memory Allocation

- cudaMalloc(void **devPtr, size_t size)
  - Allocate space in device Global Memory
- cudaFree()
  - Free allocated space in device Global Memory
- Example. Allocate 64 by 64 single precision float array. Attached the allocated storage to *Md.

```
TILE_WIDTH = 64;
float* Md;
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);


cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

Host-Device Data Transfer

- cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous transfer

- Example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);


cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```

Device Memory Allocation – MultiD Case

- Linear memory can also be allocated through cudaMallocPitch() and cudaMalloc3D() etc.
  - Recommended for allocations of 2D or 3D arrays as it makes sure that the allocation is appropriately padded to meet the alignment requirements imposed by the device.
  - It ensures best performance when accessing the row addresses or performing copies between 2D arrays and other regions of device memory (using the cudaMemcpy2D() and cudaMemcpy3D() functions)
  - The returned pitch (or stride) must be used to access array elements.

cudaError_t cudaMallocPitch(void **devPtr, size_t *pitch, size_t width, size_t height)

- Allocates at least width (in bytes) * height bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row. The pitch returned in *pitch by cudaMallocPitch() is the width in bytes of the allocation.
- Parameters:

devPtr	- Pointer to allocated pitched device memory
pitch	- Pitch for allocation
width	- Requested pitched allocation width (in bytes)
height	- Requested pitched allocation height

- Returns: cudaSuccess, cudaErrorMemoryAllocation
- Given the row and column of an array element of type T, the address is computed as:

```
T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
```

```
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch, width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code
__global__ void MyKernel(float* devPtr, size_t pitch, int width, int height)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c > width; ++c) {
            float element = row[c];
        }
    }
}
```

See also pitch_sample.cu

CUDA Function Declarations

- `__global__` defines a kernel function
  - Must return `void`
  - `Example: __global__ void  KernelFunc()`
  - `Executed on the device, only callable from the host`
- `__device__` defines a function called by kernels.
  - Example: `__device__ float DeviceFunc()`
  - `Executed on the device, only callable from the device`
- `__host__` defines a function running on the host
  - `Example: __host__   float HostFunc()`
  - `Executed on the host, only callable from the host`

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Querying Device

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <string.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>

int main(int argc, char** argv)
{
  int gpuDevice;
  int devNum = 0;
  int c, count;
  int cudareturn;

  cudaGetDeviceCount(&count);
  while ((c = getopt (argc, argv, "d:")) != -1)
  {
    switch (c)
    {
    case 'd':
      devNum = atoi(optarg);
    break;
    case '?':
      if (isprint (optopt))
        fprintf (stderr, "Unknown option `-%c'.\n", optopt);
      else
        fprintf (stderr,
          "Unknown option character `\\x%x'.\n",
          optopt);
      return 1;
    default:
      printf("GPU device not specified using device 0 ");
   }
  }
  cudareturn = cudaSetDevice( devNum );
  printf("device count = %d\n", count);
  if (cudareturn == 11)
  {
   printf("cudaSetDevice returned  11, invalid device number ");
   exit(-1);
  }
  cudaGetDevice(&gpuDevice);
  return 0;
}
```

Lecture 11: Programming on GPUs (Part 2)

Thread Creation

- Threads are created when program calls kernel functions.
- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);  // 5000 thread blocks
dim3   DimBlock(4, 8, 8); // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
   >>>(...);
```

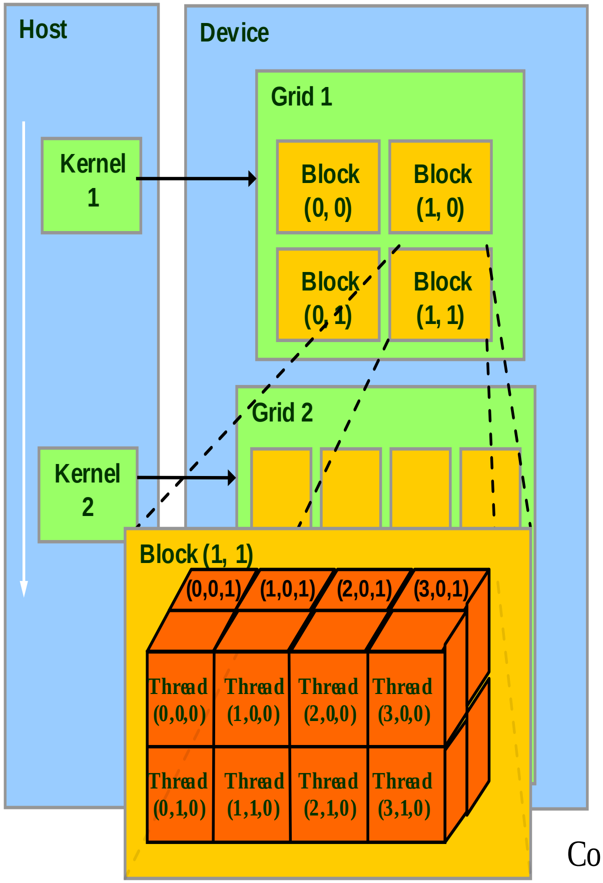
- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

dim3 is a special CUDA datatype with 3 components **.x, .y, .z** each initialized to 1.

kernel_routine<<<gridDim, blockDim>>>(args);

- A collection of blocks from a **grid** (1D, 2D or 3D)
  - Built-in variable **gridDim** specifies the size (or dimension) of the grid.
  - Each copy of the kernel can determine which block it is executing with the built-in variable **blockIdx**.

- Threads in a block are arranged in 1D, 2D, or 3D arrays.
  - Built-in variable **blockDim** specifies the size (or dimensions) of block.
  - **threadIdx** index (or 2D/3D indices) thread within a block
  - maxThreadsPerBlock: The limit is 512 threads per block



Courtesy: NDVIA

Language Extensions: Built-in Variables

- `dim3 gridDim;`
  - Dimensions of the grid in blocks (`gridDim.z` unused below version 2.x)
- `dim3 blockDim;`
  - Dimensions of the block in threads
- `dim3 blockIdx;`
  - Block index within the grid
- `dim3 threadIdx;`
  - Thread index within the block

dim3 is a special CUDA datatype with 3 components **.x, .y, .z** each initialized to 1.

Specifying 1D Grid and 1D Block

```
/// host code
int main(int argc, char **argv) {
   float *h_x, *d_x; // h=host, d=device
   int nblocks=3, nthreads=4, nsize=3*4;

   h_x = (float *)malloc(nsize*sizeof(float));
   cudaMalloc((void **)&d_x,nsize*sizeof(float));
   my_first_kernel<<<nblocks,nthreads>>>(d_x);
   cudaMemcpy(h_x,d_x,nsize*sizeof(float),
   cudaMemcpyDeviceToHost);
   for (int n=0; n<nsize; n++)
      printf(" n, x = %d %f \n",n,h_x[n]);
   cudaFree(d_x); free(h_x);
}
```

```
/// Kernel code
__global__ void my_first_kernel(float *x)
{
int tid = threadIdx.x + blockDim.x*blockIdx.x;
x[tid] = (float) threadIdx.x;
}
```

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3

Within each block of threads, **threadIdx.x** ranges from 0 to **blockDim.x-1**, so each thread has a unique value for tid

GPU SUMs of a Long Vector

- Assume 65,535*512 >> N > 512, so we need to launch threads across multiple blocks.
- Let’s use 128 threads per block. We need N/128 blocks.
  - N/128 is integer division. If N were < 128, N/128 would be **0**.
  - Actually compute (N+127)/128 blocks.
- **add <<<(N+127)/128, 128>>>(dev_a, dev_b, dev_c);**

```
#define N 4000
__global__ void add(int *a, int *b, int *c){
   int tid = threadIdx.x + blockDim.x*blockIdx.x;  // handle the data at this index

   if(tid < N)    c[tid] = a[tid] + b[tid]; // launch too many treads when N is not exact
}                                           // multiple of 128
```

GPU Sums of Arbitrarily Long Vectors

- Neither dimension of a grid of blocks may exceed 65,535.
- Let’s use 1D grid and 1D block.

```
__global__ void add(int *a, int *b, int *c){
   int tid = threadIdx.x + blockIdx.x*blockDim.x;  // handle the data at this index

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

Principle behind this implementation:

- Initial index value for each parallel thread is:
  int tid = threadIdx.x + blockIdx.x*blockDim.x;
- After each thread finishes its work at current index, increment each of them by the total number of threads running in the grid, which is **blockDim.x*gridDim.x**

```
#define N (55*1024)
__global__ void add(int *a, int *b, int *c){
   int tid = threadIdx.x + blockIdx.x*blockDim.x;  // handle the data at this index

   while(tid < N){
       c[tid] = a[tid] + b[tid];
       tid += blockDim.x*gridDim.x;
    }
}
int main()
{
…
   add <<<128, 128>>>(dev_a, dev_b, dev_c);
…
}

//see vec_arb_len_add.cu
```

Specifying 1D Grid and 2D Block

If we want to use a 1D grid of blocks and 2D set of threads, then **blockDim.x, blockDim.y** give the block dimensions, and **threadIdx.x, threadIdx.y** give the thread indices.

```
Main()
{
   int nblocks = 2;
   dim3  nthreads(16, 4);
   my_second_kernel<<<nblocks, nthreads>>>(d_x);
}
```

dim3 is a special CUDA datatype with 3 components **.x, .y, .z** each initialized to 1.

```
/// Kernel code
__global__ void my_second_kernel(float *x)
{
int tid = threadIdx.x + blockDim.x* threadIdx.y +blockDim.x*blockDim.y*blockIdx.x;
x[tid] = (float) threadIdx.x;
}
```

- In a 3D block of threads, thread ID is computed by:

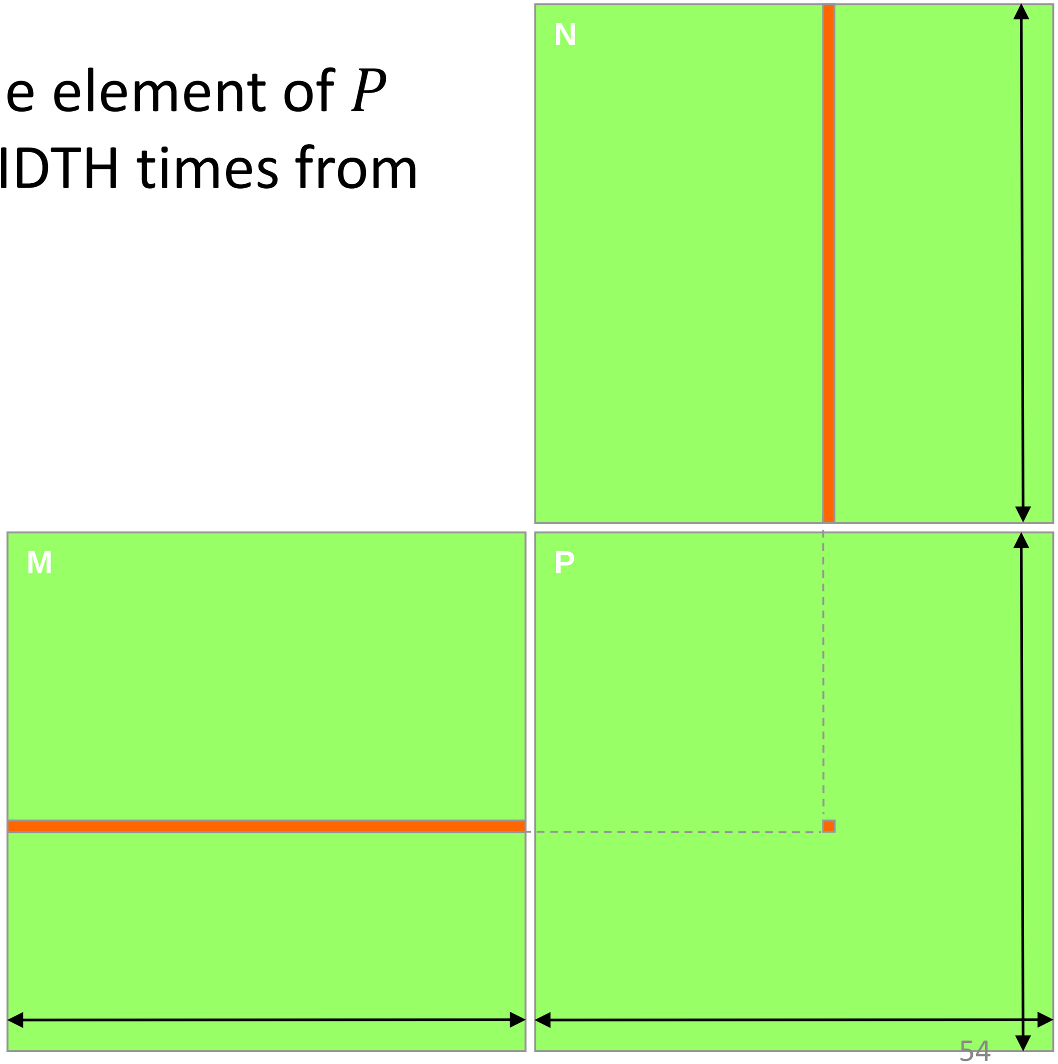
threadIdx.x +threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockDim.y

```
__global__ void KernelFunc(...);

main()
{
  dim3   DimGrid(100, 50);   // 5000 thread blocks
  dim3   DimBlock(4, 8, 8);  // 256 threads per block
  KernelFunc<<< DimGrid, DimBlock>>>(...);
}
```

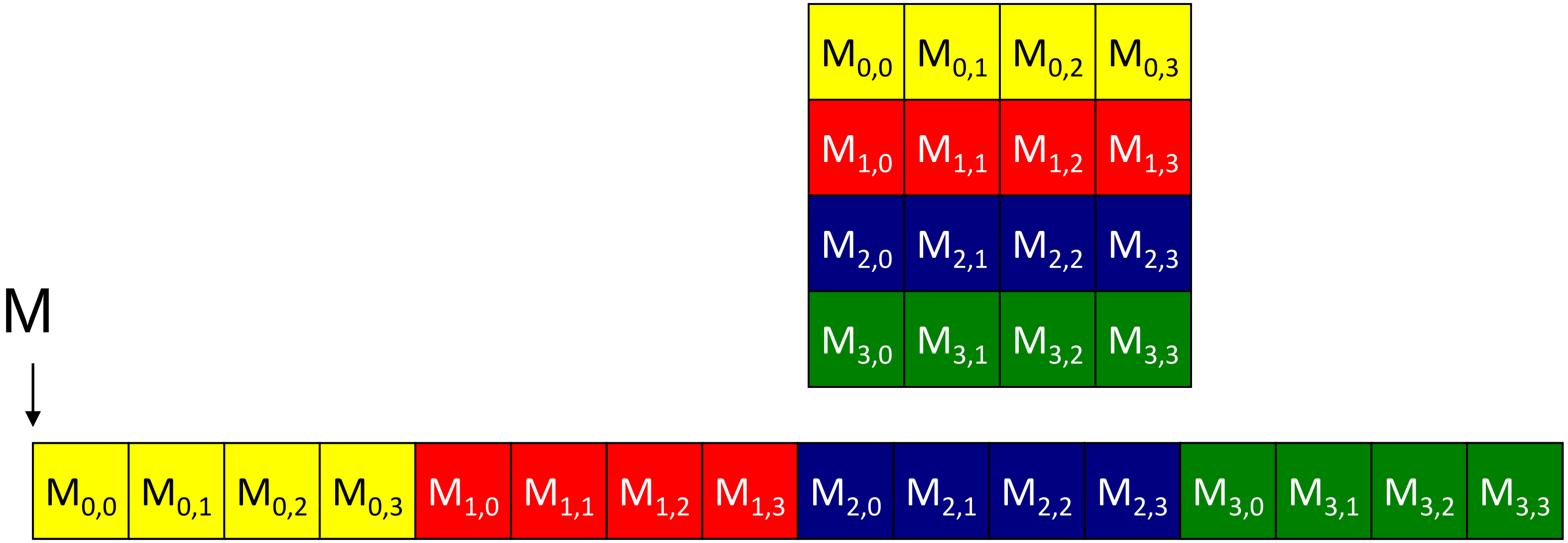
Matrix Multiplication

- Demonstrate basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity

- $P = M \times N$ of size WIDTH×WIDTH
- **Without blocking**:
  - One thread handles one element of P
  - M and N are loaded WIDTH times from global memory



C Language Implementation



$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

M

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

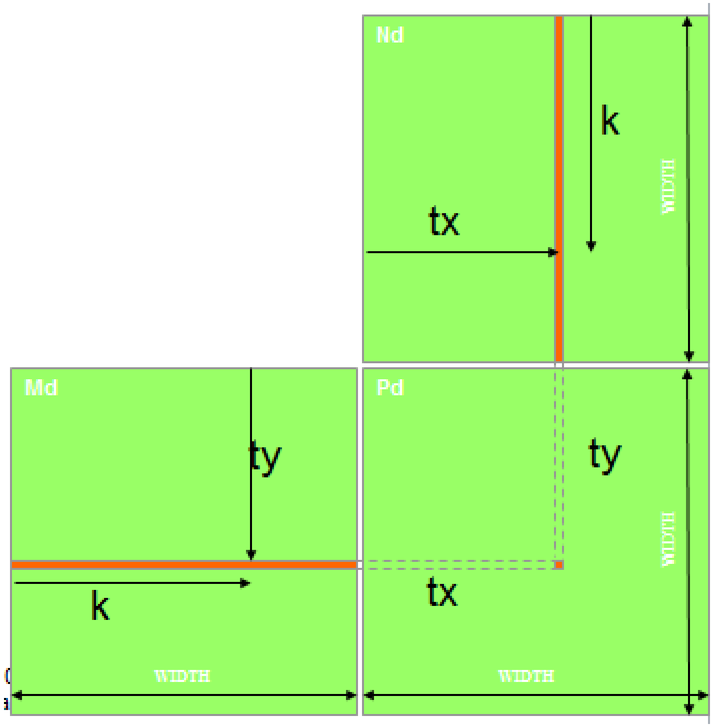
```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

Data Transfer (Host/Device)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
    float *Md, *Nd, *Pd;
   …
    //1. Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

     cudaMalloc(&Nd, size);
     cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

     // Allocate P on the device
    cudaMalloc(&Pd, size);
   //2.  Kernel invocation code –
   …

    // 3. Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

     // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

Kernel Function

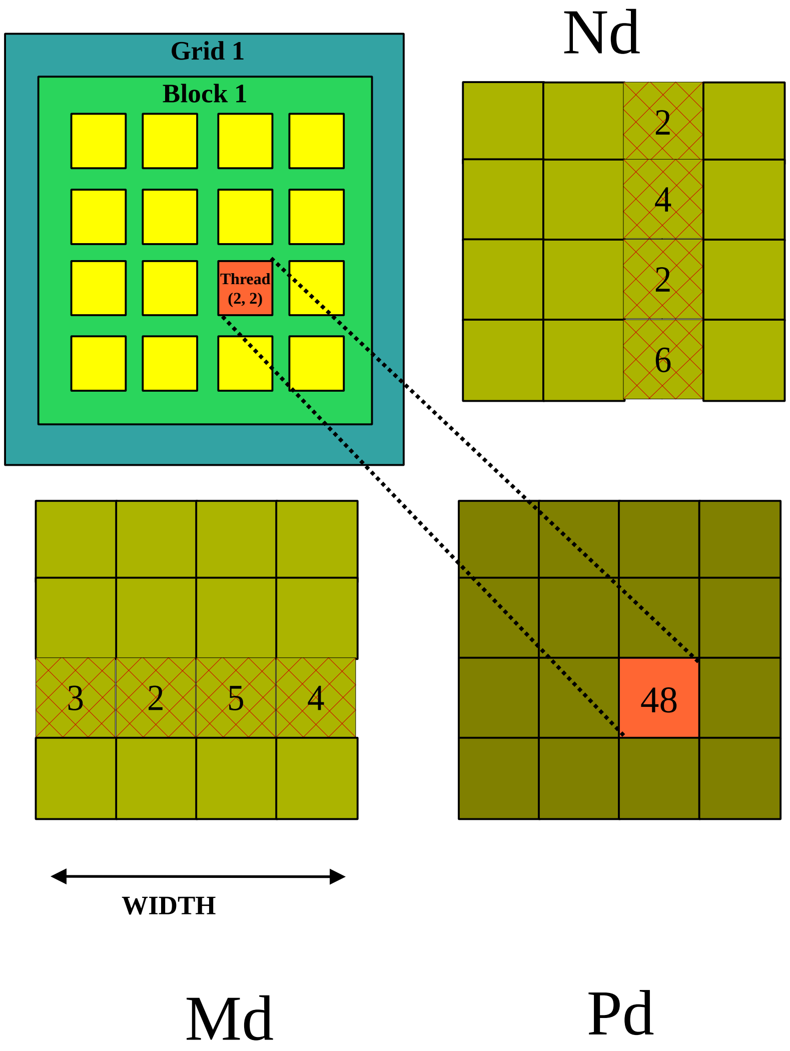
```
// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
       float Melement = Md[threadIdx.y*Width+k];
       float Nelement = Nd[k*Width+threadIdx.x];
       Pvalue += Melement * Nelement;
   }

  Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



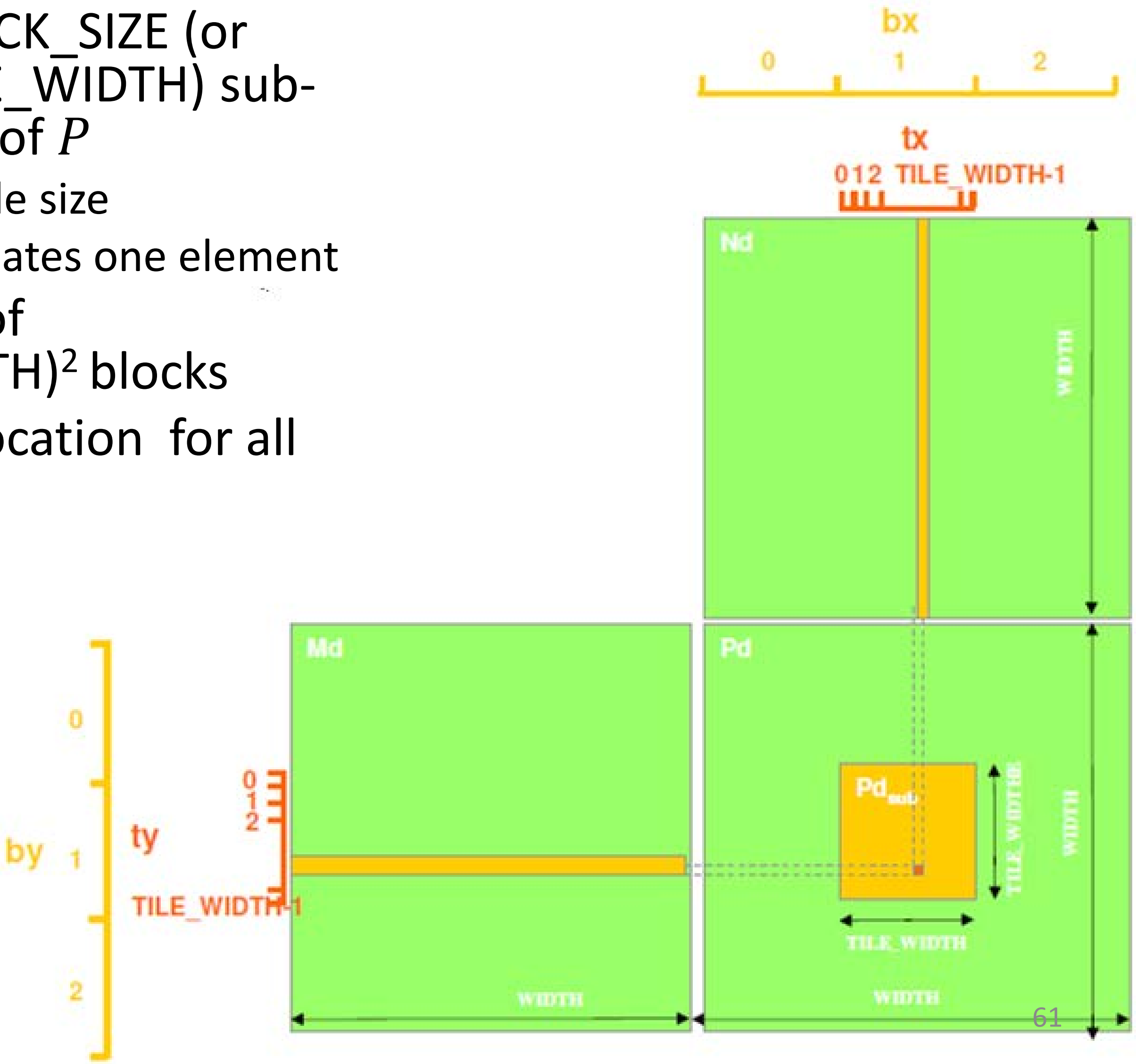
Kernel Invocation

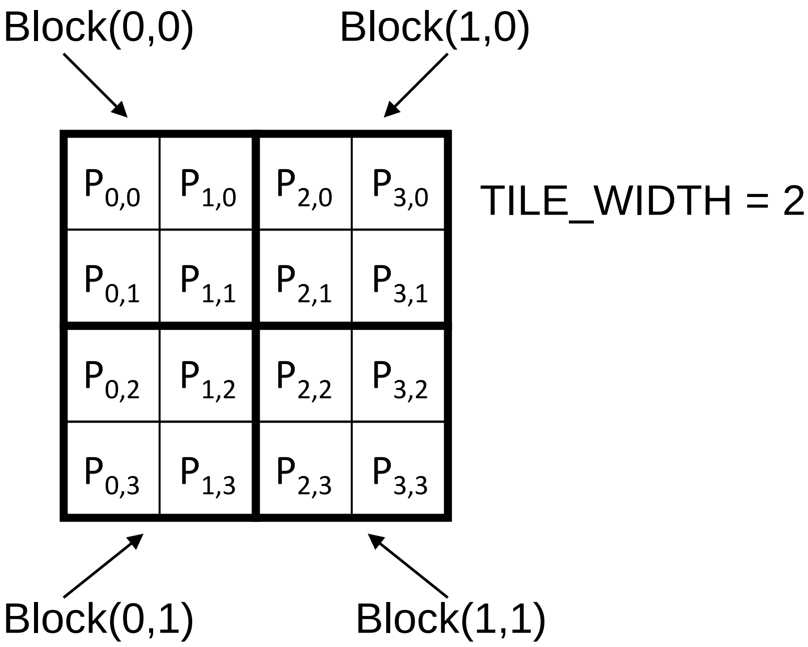
```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    …
   //2. Kernel invocation code – to be shown later
    // Setup the execution configuration
       dim3 dimGrid(1, 1);
      dim3 dimBlock(Width, Width);

    // Launch the device computation threads
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd,
    Width);
    …
}
```

- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



- $P = M \times N$ of size WIDTH×WIDTH
- **With blocking**:
  - One **thread block** handles one BLOCK_SIZE × BLOCK_SIZE (or TILE_WIDTH × TILE_WIDTH) sub-matrix (tile) Pd_{sub} of P
    - Block size equal tile size
    - Each thread calculates one element
  - Genrate a 2D grid of (WIDTH/TILE_WIDTH)^2 blocks
  - Linear memory allocation for all matrices is used





Block(0,0)

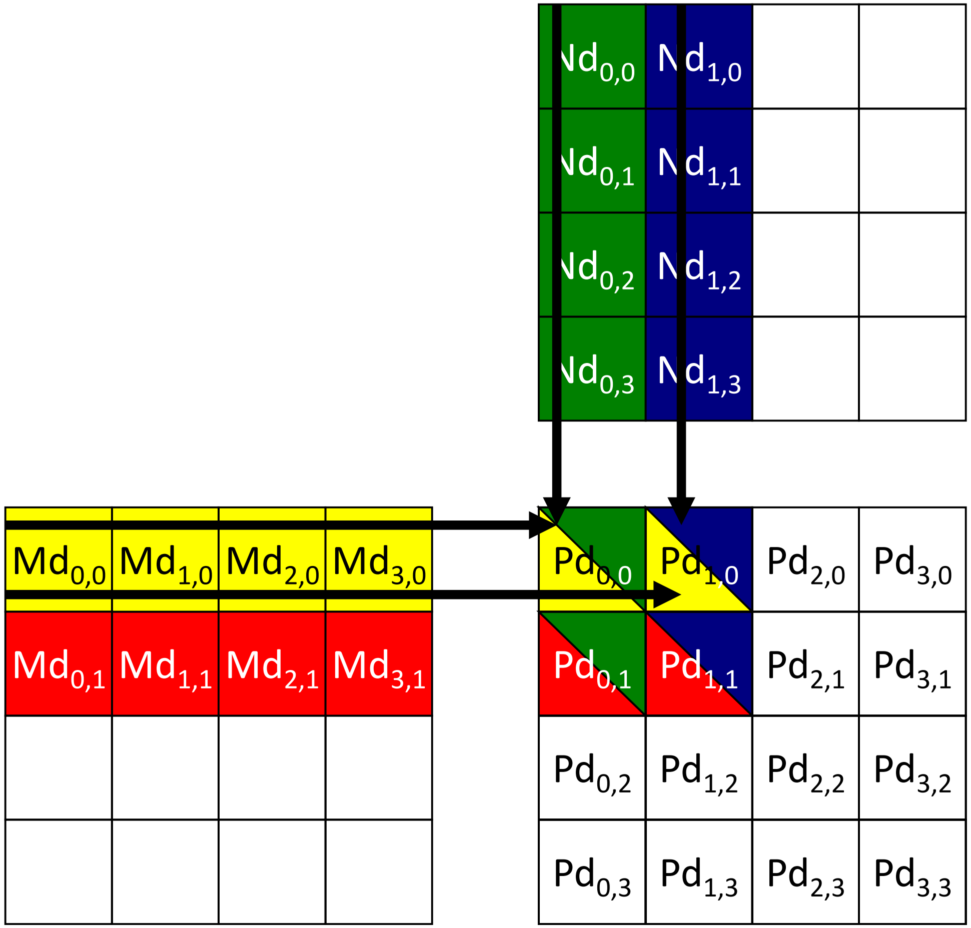
Block(1,0)

$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$
$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$
$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$
$P_{0,3}$	$P_{1,3}$	$P_{2,3}$	$P_{3,3}$

TILE_WIDTH = 2

Block(0,1)

Block(1,1)



$Nd_{0,0}$	$Nd_{1,0}$		
$Nd_{0,1}$	$Nd_{1,1}$		
$Nd_{0,2}$	$Nd_{1,2}$		
$Nd_{0,3}$	$Nd_{1,3}$		

$Md_{0,0}$	$Md_{1,0}$	$Md_{2,0}$	$Md_{3,0}$
$Md_{0,1}$	$Md_{1,1}$	$Md_{2,1}$	$Md_{3,1}$

$Pd_{0,0}$	$Pd_{1,0}$	$Pd_{2,0}$	$Pd_{3,0}$
$Pd_{0,1}$	$Pd_{1,1}$	$Pd_{2,1}$	$Pd_{3,1}$
$Pd_{0,2}$	$Pd_{1,2}$	$Pd_{2,2}$	$Pd_{3,2}$
$Pd_{0,3}$	$Pd_{1,3}$	$Pd_{2,3}$	$Pd_{3,3}$

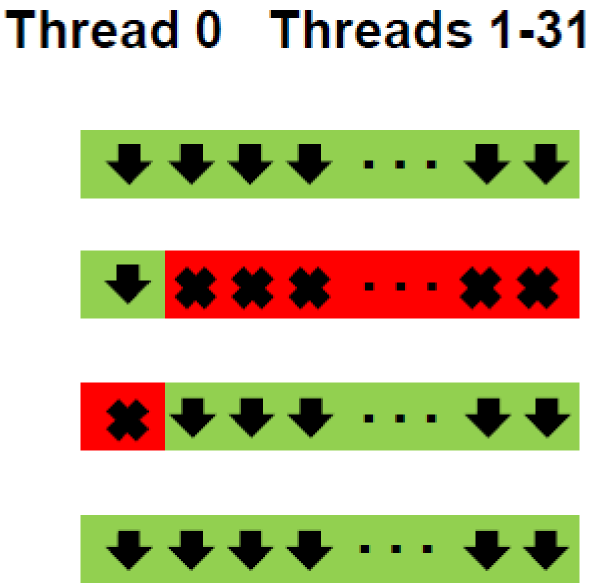
Revised Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int
     Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

Multithreading

- Cores in a streaming multiprocessor (SM) are Single Instruction Multiple Threads (SIMT) cores:
  - Maximum number of threads in a block depends on the compute capability (1024 on Fermi)
    - all cores execute the same instructions simultaneously, but with different data.
  - GPU multiprocessor creates, manages, schedules and executes threads in warps of 32*
    - minimum of 32 threads all doing the same thing at (almost) the same time (Warp executes one common instruction at a time).
    - no “context switching”; each thread has its own registers, which limits the number of active threads
    - Threads are allowed to branch, but branches are serialized
  - threads on each SM execute in groups of 32 called “warps”
  - execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data

- Thread Branching



Program

```
int tid = threadIdx.x;
 if (tid==0) {var1++}
 else {var1 = var1+3;}
 var2 = 3*5 + var1;
```

- Suppose we have 1000 blocks, and each one has 128 threads – how does it get executed?
- On current Fermi hardware, would probably get 8 blocks running at the same time on each SM, and each block has 4 warps =) 32 warps running on each SM
- Each clock tick, SM warp scheduler decides which warp to execute next, choosing from those not waiting for
  - data coming from device memory (memory latency)
  - completion of earlier instructions (pipeline delay)
- Programmer doesn’t have to worry about this level of detail (can always do profiling later), just make sure there are lots of threads / warps

Spatial Locality

```
__global__ void good_kernel(float *x)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   x[tid] = threadIdx.x;
}
```

- 32 threads in a warp address neighboring elements of array x.
- If the data is correctly “aligned” so that x[0] is at the beginning of a cache line, then x[0]-x[31] will be in the same cache line.
  - Cache line is the basic unit of data transfer, 128 bytes cache line (32 floats or 16 doubles).
- Good spatial locality.

```
__global__ void bad_kernel(float *x)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   x[1000*tid] = threadIdx.x;
}
```

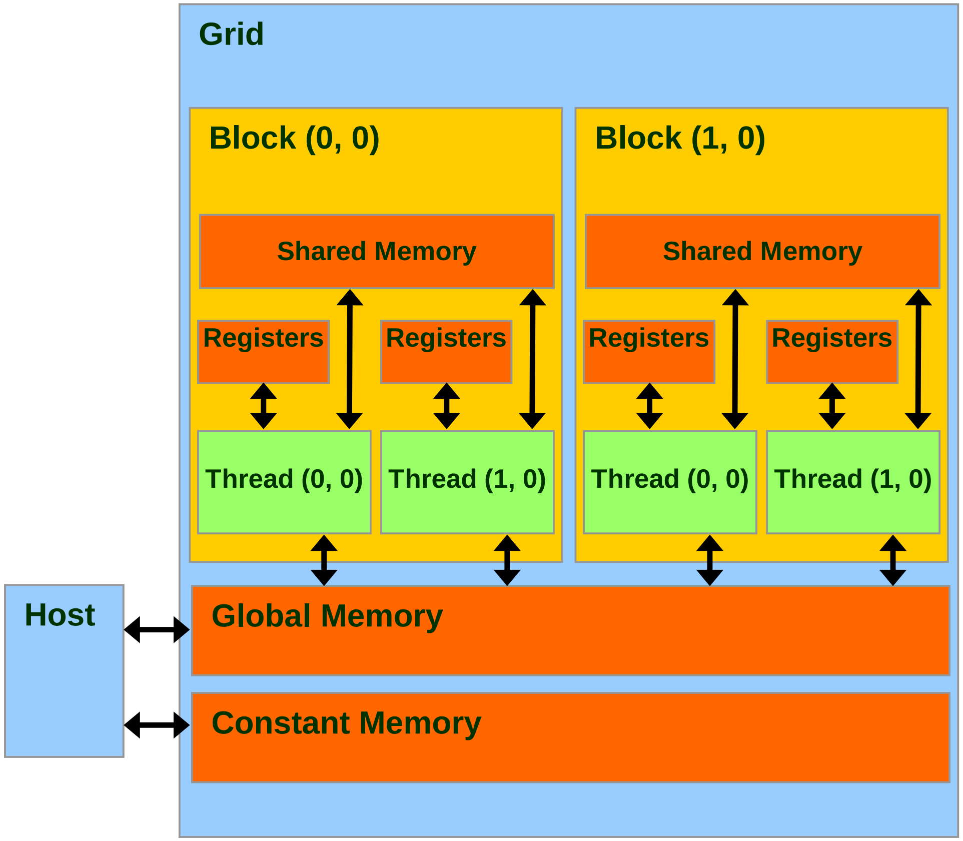
- Different threads within a warp access widely spaced elements of array x.
- Each access involves a different cache line, so performance is poor.

CUDA Memories

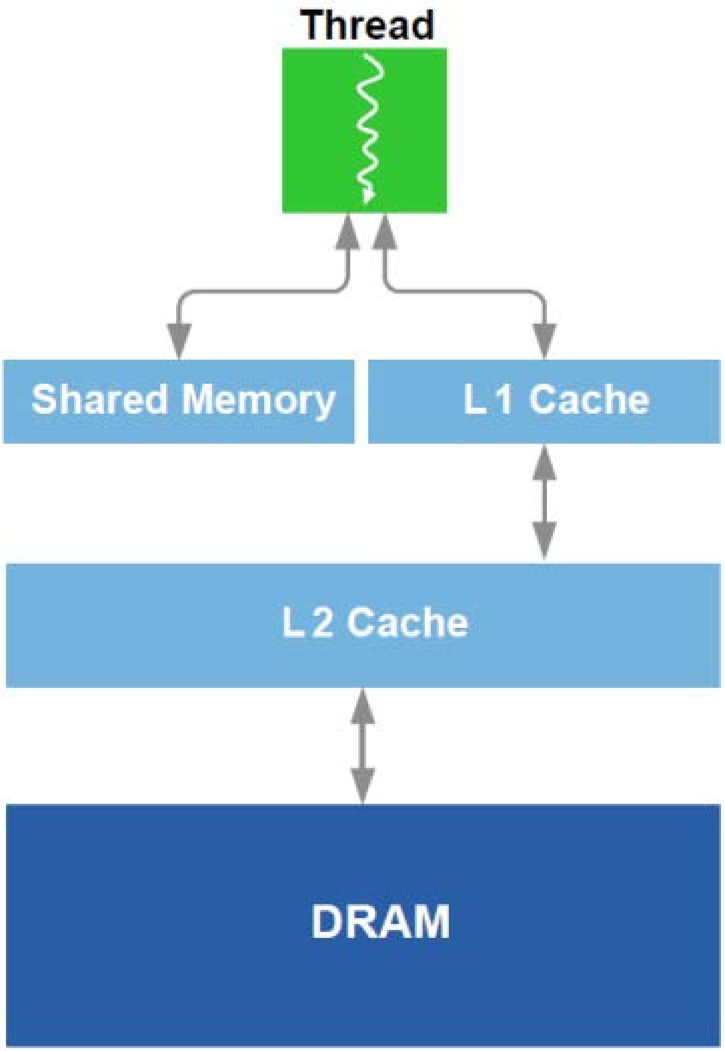
- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread local memory
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read/only per-grid **constant memory**

Local Memory

- Usually used when one runs out of SM resources
- "Local" because each thread has its own private area
- Not really a "memory" – bytes are stored in global memory
  - Stores are cached in L1



Fermi Memory Hierarchy



NVIDIA 2011

Access Times

- Register – dedicated HW – single cycle
- Shared Memory – dedicated HW – single cycle
- Local Memory – DRAM – slow
- Global Memory – DRAM – slow
- Constant Memory – DRAM, cached,
  1…10s … 100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached,
  1…10s … 100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

Variable Types

Variable declaration	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- the __device__ indicates this is a global variable in the GPU
  - the variable can be read and modified by any kernel
  - its lifetime is the lifetime of the whole application
  - can also declare arrays of fixed size
  - can read/write by host code using standard cudaMemcpy
- **`__device__`** is optional when used with **`__local__`**, **`__shared__`**, or **`__constant__`**

- Constant variables
  - Very similar to global variables, except that they can’t be modified by kernels
  - defined with global scope within the kernel file using the prefix __constant__
  - initialized by the host code using cudaMemcpyToSymbol, cudaMemcpyFromSymbol or cudaMemcpy in combination with cudaGetSymbolAddress
  - Only 64KB of constant memory
- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:

    ```
    __global__ void KernelFunc(float* ptr)
    ```

  - Obtained as the address of a global variable:

    ```
    float* ptr = &GlobalVar;
    ```

- **Automatic variables** without any qualifier reside in a **register**
  - Except arrays that reside in local memory

```
__global__ void lap(int I, int J,float *u1, float *u2) {
int i = threadIdx.x + blockIdx.x*blockDim.x;
int j = threadIdx.y + blockIdx.y*blockDim.y;
int id = i + j*I;
if (i==0 || i==I-1 || j==0 || j==J-1) {
   u2[id] = u1[id]; // Dirichlet b.c.’s }
else {
u2[id] = 0.25f * ( u1[id-1] + u1[id+1]
+ u1[id-I] + u1[id+I] );}
}
```

Accessing Global Variables via the Runtime API

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
cudaMemcpyFromSymbol(data, constData, sizeof(data));
```

```
__device__ float devData;
float value = 3.14f;
cudaMemcpyToSymbol(&devData, &value, sizeof(float));
```

```
__device__ float* devPointer;
float* ptr;
cudaMalloc(&ptr, 256 * sizeof(float));
cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));
```

Shared Memory

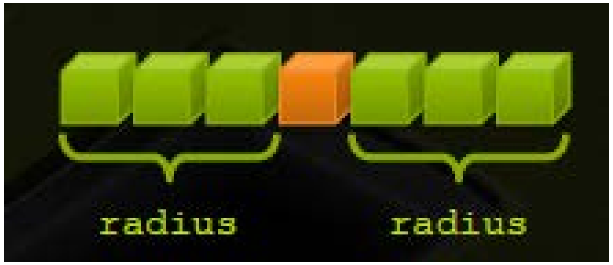
```
__shared__ int x_dim;
__shared__ float x[128];
```

- declares data to be shared between all of the threads in the thread block – any thread can set its value, or read it.
- Advantages of using shared memory
  - essential for operations requiring communication between threads
  - useful for data re-use
  - alternative to local arrays in device memory
  - reduces use of registers when a variable has same value for all threads

Cooperating Threads

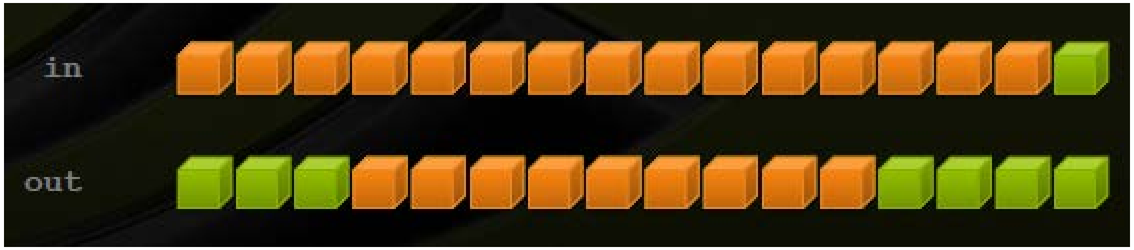
1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
  - If radius = 3, then each output element is the sum of 7 input elements
  - Assume we use 1D block grid and 1D block of threads



Implementing within a Block

- Each thread processes one output element
  - – blockDim.x elements per block
- Input elements are read several times
  - – With radius 3, each input element is read 7 times

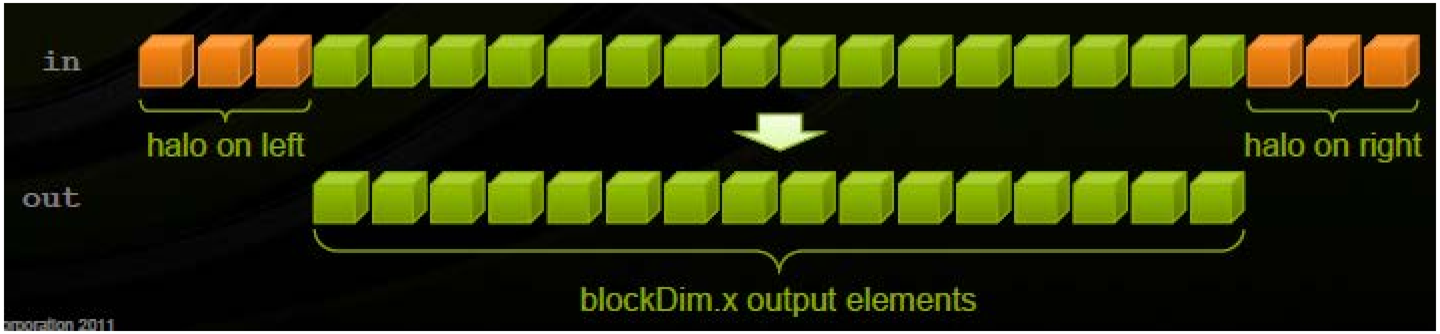


Sharing Data Between Threads

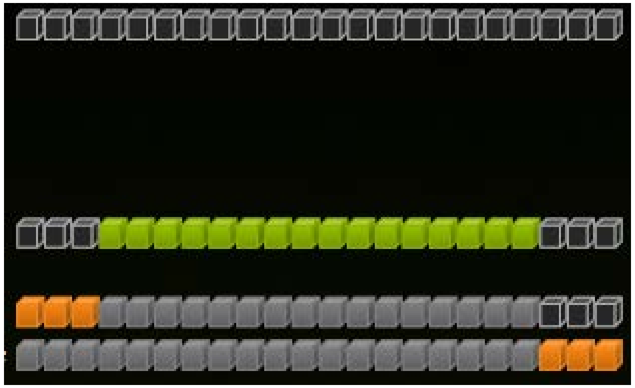
- Within a block, threads share data by shared memory
- Extremely fast on-chip memory, user-managed
- Declare using __shared__, allocated per block
- Data is not visible to threads in other blocks

Implementation

- Cache data in shared memory
  - Read (blockDim.x + 2*radius) input elements from device global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory

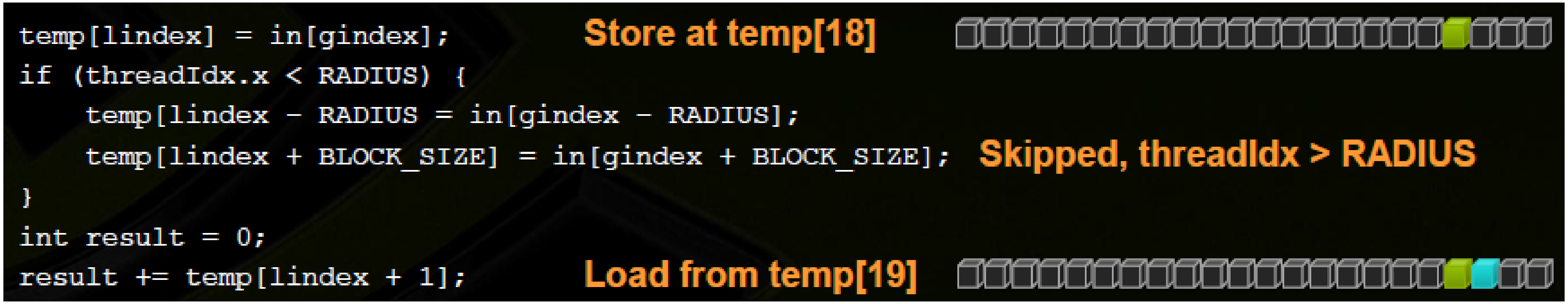


```
__global__ void stencil_1d(int *in, int *out)
{
   __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
   int gindex = threadIdx.x + blockIdx.x * blockDim.x;
   int lindex = threadIdx.x + RADIUS;
   // Read input elements into shared memory
   temp[lindex] = in[gindex];
   if (threadIdx.x < RADIUS) {
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

   // Apply the stencil
   int result = 0;
   for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
      result += temp[lindex + offset];
   // Store the result
   out[gindex] = result;
}
```



Data Race

- The stencil example will not work
- Suppose thread 15 reads the halo before thread 0 has fetch it



```
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];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

Load from temp[19]

void __syncthreads()

- Synchronizes all threads within a block
  - Used to prevent data races
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

```
__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) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    __syncthreads();
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];
    // Store the result
    out[gindex] = result;
}
```

- If a thread block has more than one warp, it’s not pre-determined when each warp will execute its instructions – warp 1 could be many instructions ahead of warp 2, or well behind.

- Consequently, almost always need thread synchronization to ensure correct use of shared memory.

- Instruction
  - __syncthreads();

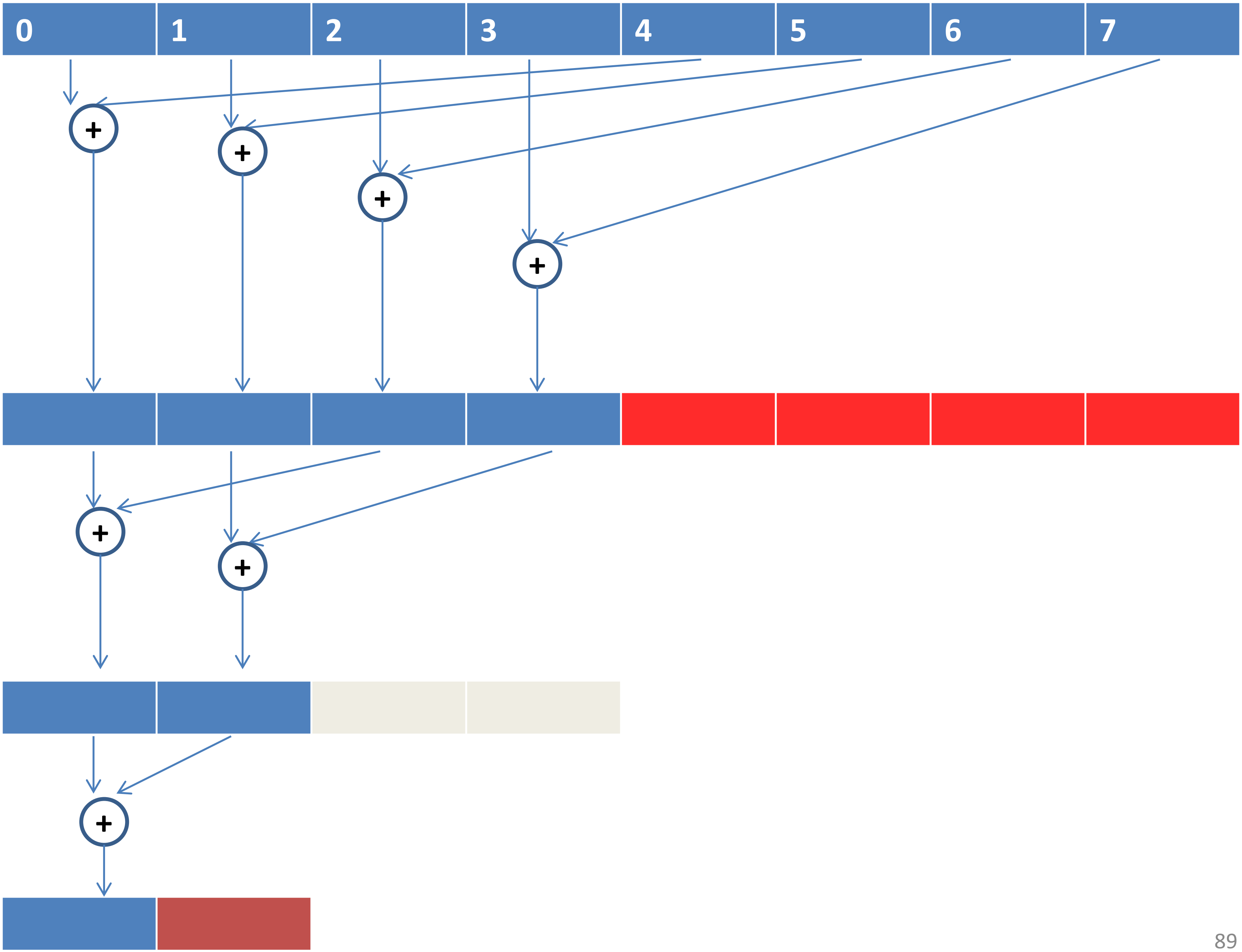
- inserts a “barrier”; no thread/warp is allowed to proceed beyond this point until the rest have reached it
- Total size of shared memory is specified by an optional third arguments when launching the kernel:
  - kernel<<<blocks,threads,shared_bytes>>>(...)

- Global memory resides in device memory (DRAM) - much slower access than shared memory
- So, a profitable way of performing computation on the device is to tile data to take advantage of fast shared memory:
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory
- Constant memory also resides in device memory (DRAM) - much slower access than shared memory
  - But… cached!
  - Highly efficient access for read-only data
- Carefully divide data according to access patterns
  - R/Only → constant memory (very fast if in cache)
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very fast)
  - R/W inputs/results → global memory (very slow)

Shared Memory and Synchronization for Dot Product

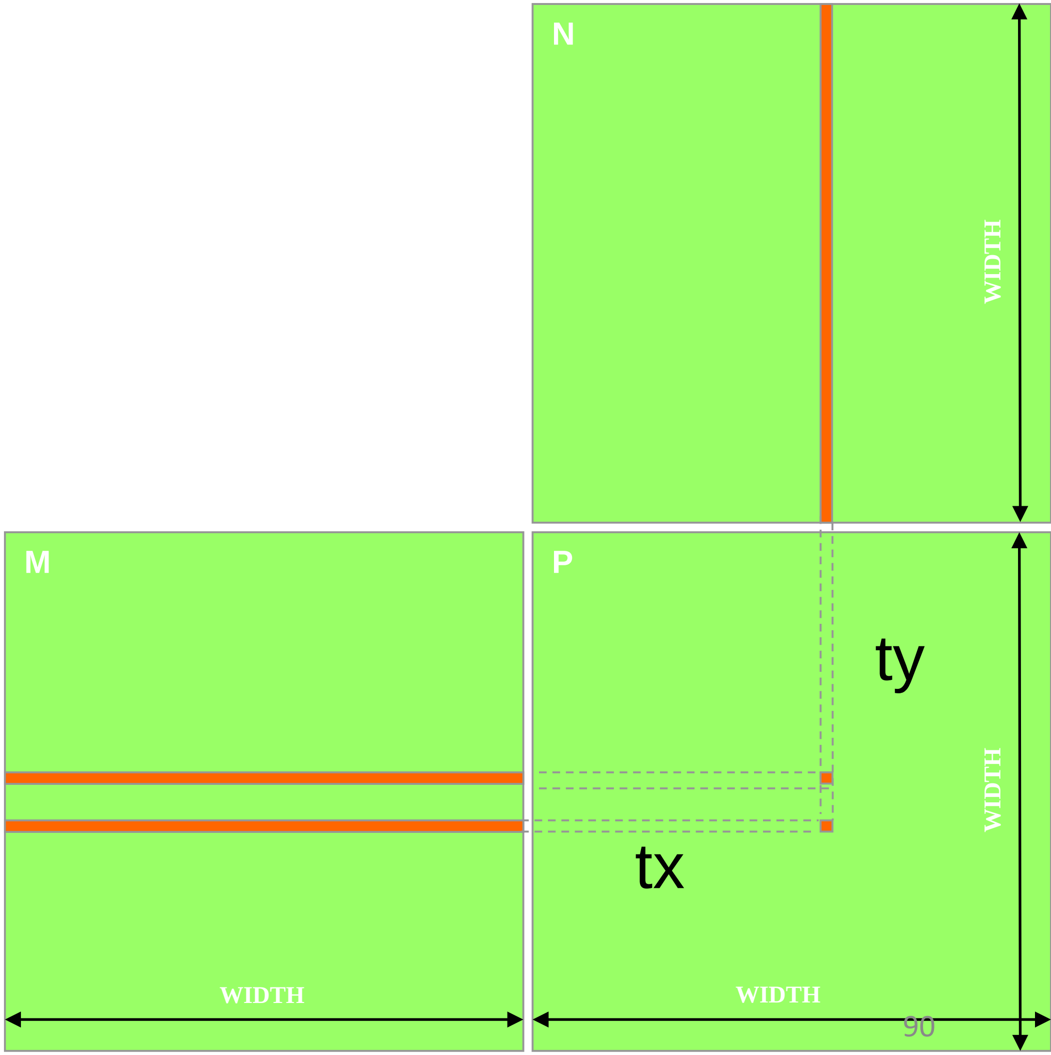
```
#define imin(a,b) ((a)<(b)?(a):(b))
const int N = 33*1024;
const int threadsPerBlock = 256;
const int blocksPerGrid = imin(32, (N+threadsPerBlock-1)/threadsPerBlock);
int main(){
   float *a, *b, c, *partial_c;
   float *dev_a, *dev_b, *dev_partial_c;
   a = (float*)malloc(N*sizeof(float));  b = (float*)malloc(N*sizeof(float));
   partial_c = (float*)malloc(blocksPerGrid*sizeof(float));
   cudaMalloc((void**)&dev_a,N*sizeof(float));
   cudaMalloc((void**)&dev_b,N*sizeof(float));
   cudaMalloc((void**)&dev_partial_c,blocksPerGrid*sizeof(float));
   // initialize a[] and b[] …
   cudaMemcpy(dev_a,a,N*sizeof(float),cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b,b,N*sizeof(float),cudaMemcpyHostToDevice);
   dot<<< blocksPerGrid, threadsPerBlock>>>(dev_a,dev_b,dev_partial_c);
   cudaMemcpy(partial_c,dev_partialc,blocksPerGrid*sizeof(float),cudaMemcpyDeviceToHost);
   c = 0;
   for(int i=0; i<blocksPerGrid;i++)    c += partial_c[i];
   // cuda memory free, etc.
}
```

```
__global__ void dot(float *a, float*b, float *c){
   __shared__ float cache[threadsPerBlock];
   //this buffer will be used to store each thread's running sum
   // the compiler will allocate a copy of shared variables for each block
   int tid = threadIdx.x + BlockIdx.x*blockDim.x;
   int cacheIndex = threadIdx.x;
   float temp = 0.0;
   while(tid < N){
      temp += a[tid]*b[tid];
      tid += blockDim.x*gridDim.x;
   }
   // set the cache values
   cache[cacheIndex]=temp;

   // we need to sum all the temporary values in the cache.
   // need to guarantee that all of these writes to the shared array
   // complete before anyone to read from this array.

   // synchronize threads in this block
   __syncthreads();    // This call guarantees that every thread in the block has
                        // completed instructions prior to __syncthreads() before the
                        // hardware will execute the next instruction on any thread.
// each thread will add two of values in cache[] and
// store the result back to cache[].
// We continue in this fashion for log_2(threadsPerBlock)
//steps till we have the sum of every entry in cache[].
// For reductions, threadsPerBlock must be a power of 2
    int i=blockDim.x/2;
    while(i!=0){
        if(cacheIndex <i)
            cache[cacheIndex] += cache[cacheIndex+i];
         __syncthreads();
         i/=2;
     }
     if(cacheIndex==0)
         c[blockIdx.x]=cache[0];
}
```


0
1
2
3
4
5
6
7
+
+
+
+
+
+
+

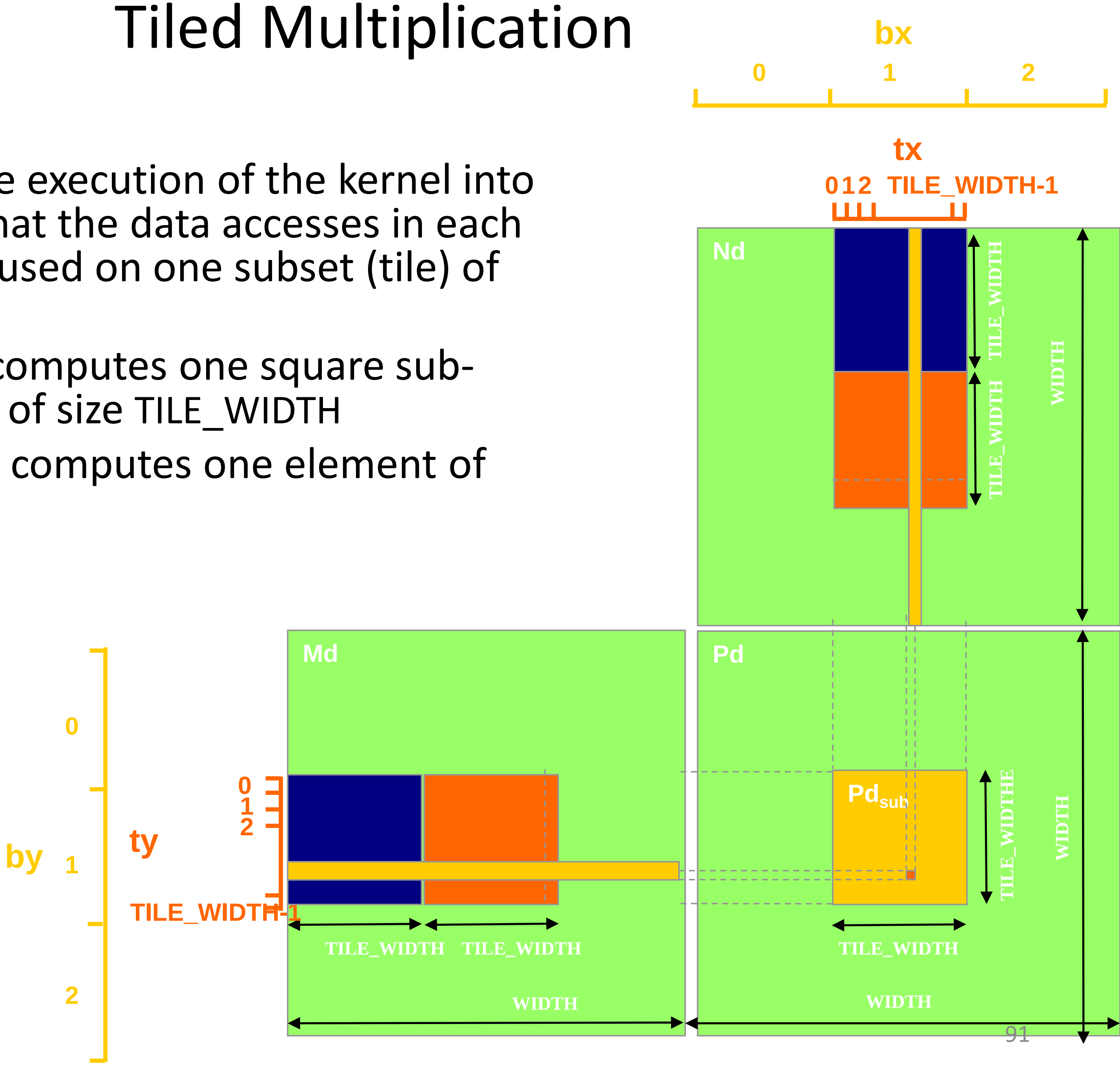
Shared Memory to Reuse Global Memory Data

- Each input element is read by Width threads.
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth
  - **Tiled algorithms**

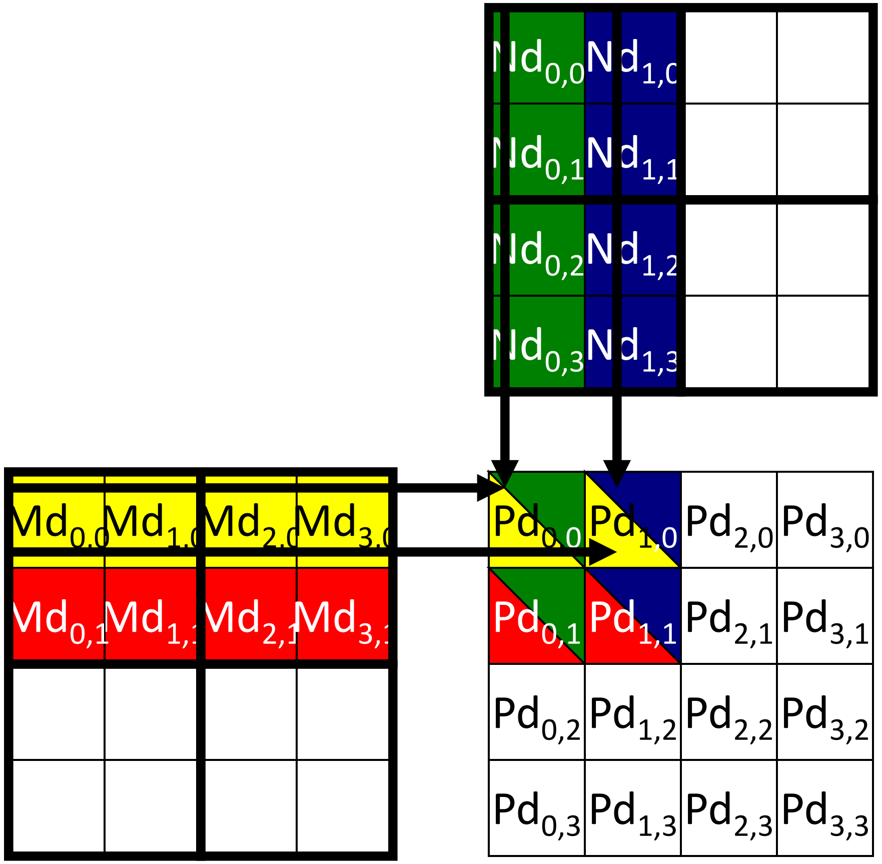


Tiled Multiplication

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd
- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



Breaking Md and Nd into Tiles


$Nd_{0,0}$ $Nd_{1,0}$
$Nd_{0,1}$ $Nd_{1,1}$
$Nd_{0,2}$ $Nd_{1,2}$
$Nd_{0,3}$ $Nd_{1,3}$
$Md_{0,0}$ $Md_{1,0}$ $Md_{2,0}$ $Md_{3,0}$
$Md_{0,1}$ $Md_{1,1}$ $Md_{2,1}$ $Md_{3,1}$
$Pd_{0,0}$ $Pd_{1,0}$ $Pd_{2,0}$ $Pd_{3,0}$
$Pd_{0,1}$ $Pd_{1,1}$ $Pd_{2,1}$ $Pd_{3,1}$
$Pd_{0,2}$ $Pd_{1,2}$ $Pd_{2,2}$ $Pd_{3,2}$
$Pd_{0,3}$ $Pd_{1,3}$ $Pd_{2,3}$ $Pd_{3,3}$

Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P

Access order

$P_{0,0}$ $thread_{0,0}$	$P_{1,0}$ $thread_{1,0}$	$P_{0,1}$ $thread_{0,1}$	$P_{1,1}$ $thread_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

Each Phase of a Thread Block Uses One Tile from Md and One from Nd

	Phase 1			Phase 2		
$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

- Each thread block should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
- There should be many thread blocks
  - A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks
- Each thread block perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  - Memory bandwidth no longer a limiting factor

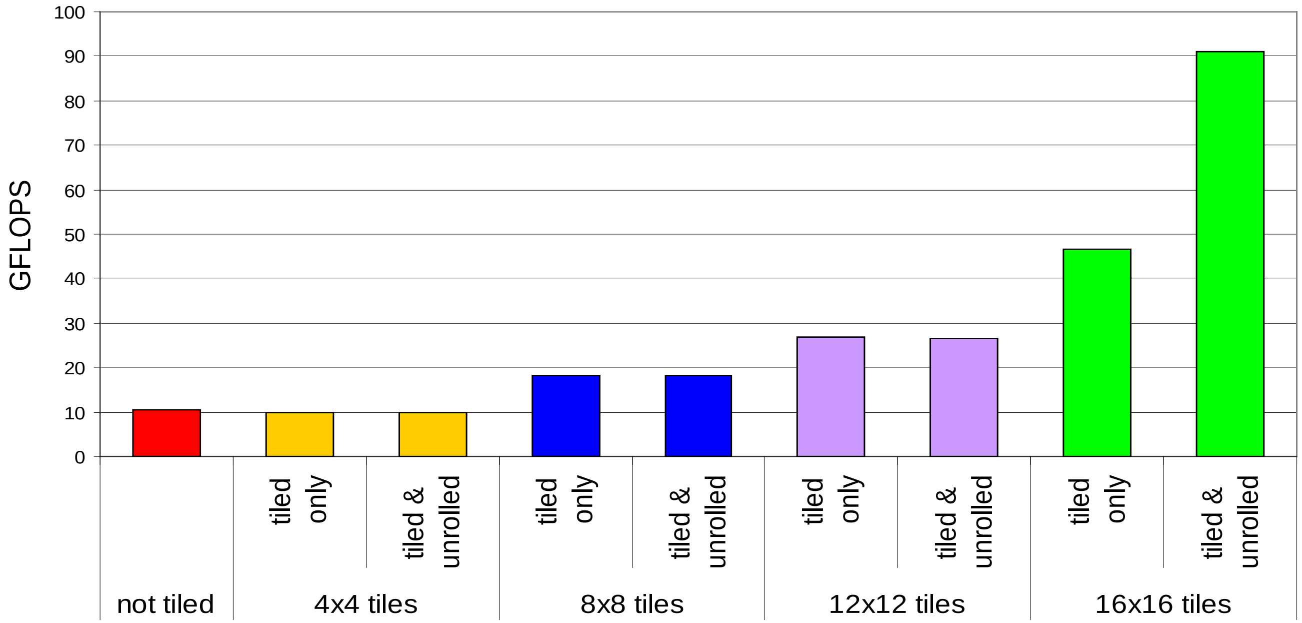
Kernel Execution Configuration

```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width/TILE_WIDTH,
            Width/TILE_WIDTH);
```

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

   float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Coolaborative loading of Md and Nd tiles into shared memory
       Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
       Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
       __syncthreads();

       for (int k = 0; k < TILE_WIDTH; ++k)
          Pvalue += Mds[ty][k] * Nds[k][tx];
       Synchthreads();
   }
   Pd[Row*Width+Col] = Pvalue;
}
```

Performance on G80

- Each SM in G80 has 16KB shared memory
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B= 8KB shared memory usage per thread block, allowing only up to two thread blocks active at the same time
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 86.4B/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS



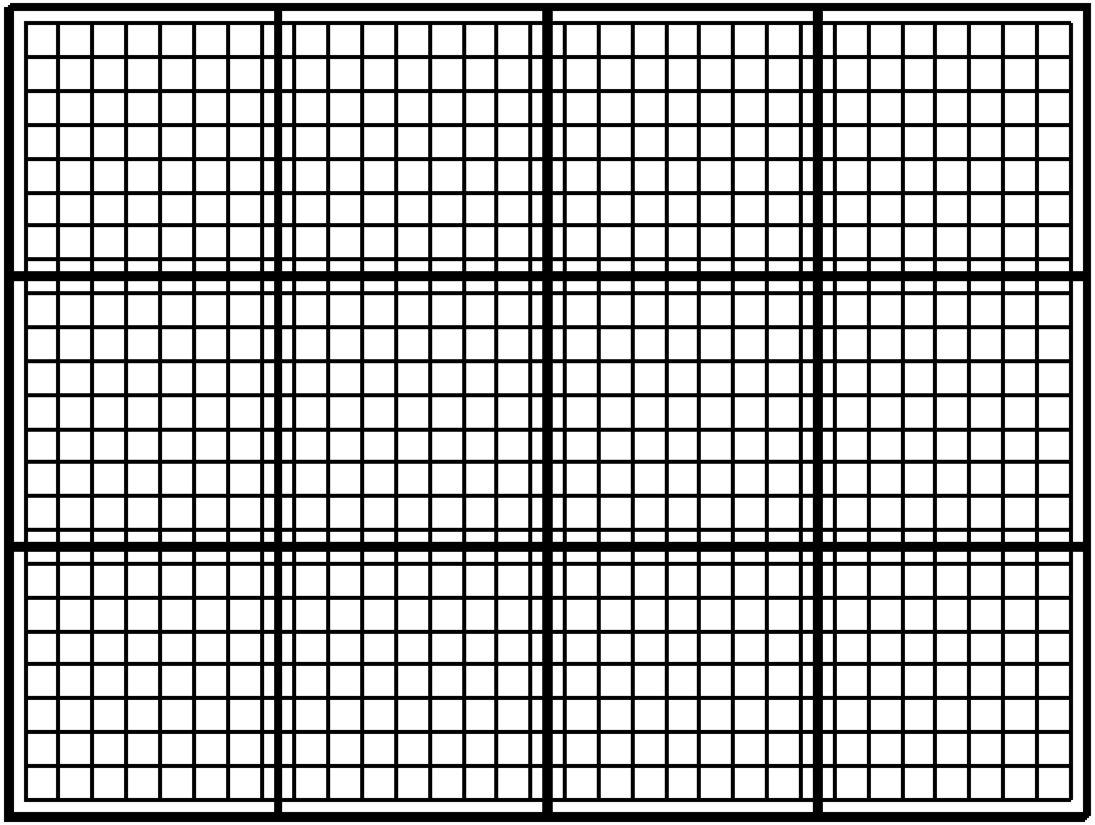
2D Laplace Solver

- Jacobi iteration to solve discrete Laplace equation on a uniform grid

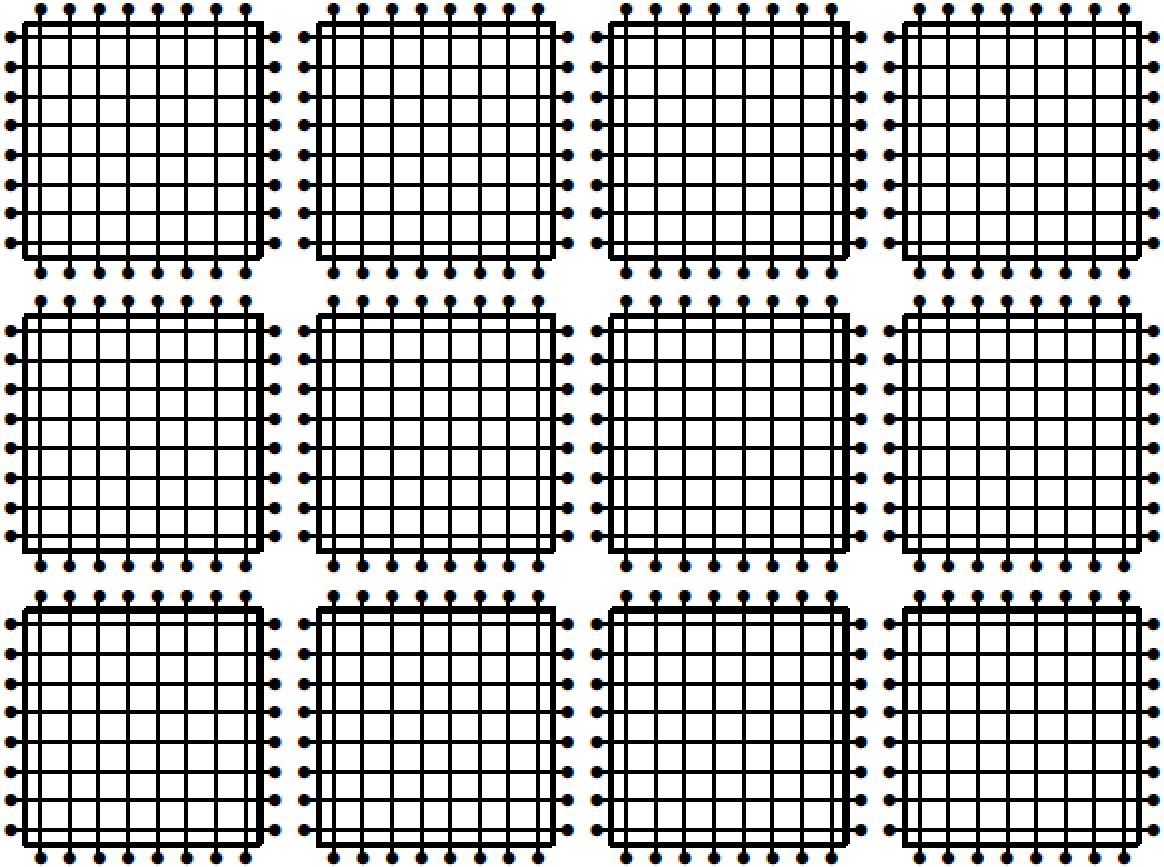
```
for (int j=0; j<J; j++) {
  for (int i=0; i<I; i++) {
    id = i + j*I; // 1D memory location
    if (i==0 || i==I-1 || j==0 || j==J-1)
      u2[id] = u1[id];
    else
    u2[id] = 0.25*( u1[id-1] + u1[id+1]
                  + u1[id-I] + u1[id+I] );
   }
}
```

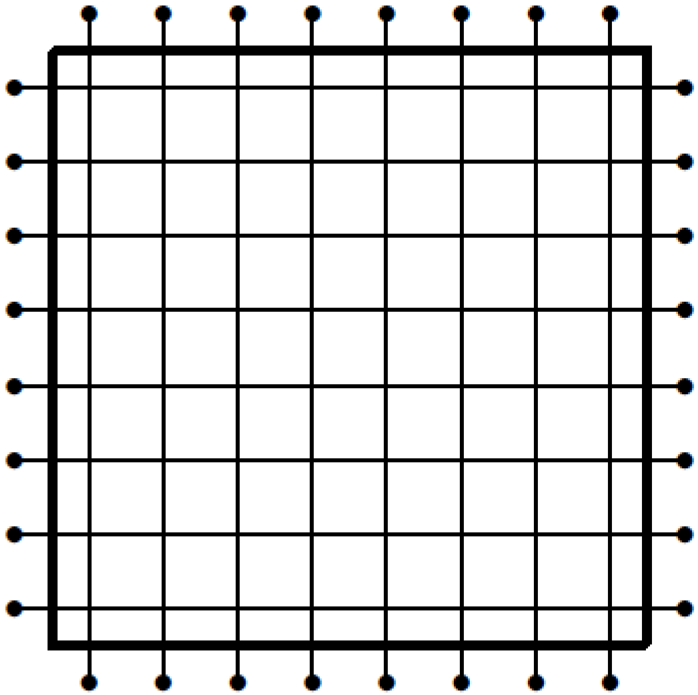
2D Laplace Solver Using CUDA

- each thread responsible for one grid point
- each block of threads responsible for a block of the grid
- conceptually very similar to data partitioning in MPI distributed-memory implementations, but much simpler



→





- Each block of threads processes one of these grid blocks, reading in old values and computing new values.

```
__global__ void lap(int I, int J, float *u1, float *u2) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;
    int id = i + j*I;
    if (i==0 || i==I-1 || j==0 || j==J-1) {
        u2[id] = u1[id]; // Dirichlet b.c.’s
    }
    else {
        u2[id] = 0.25 * ( u1[id-1] + u1[id+1]
                          + u1[id-I] + u1[id+I] );
    }
}
```

Assumptions:

- I is a multiple of blockDim.x
- J is a multiple of blockDim.y

  grid breaks up perfectly into blocks

- I is a multiple of 32

Can remove these assumptions by

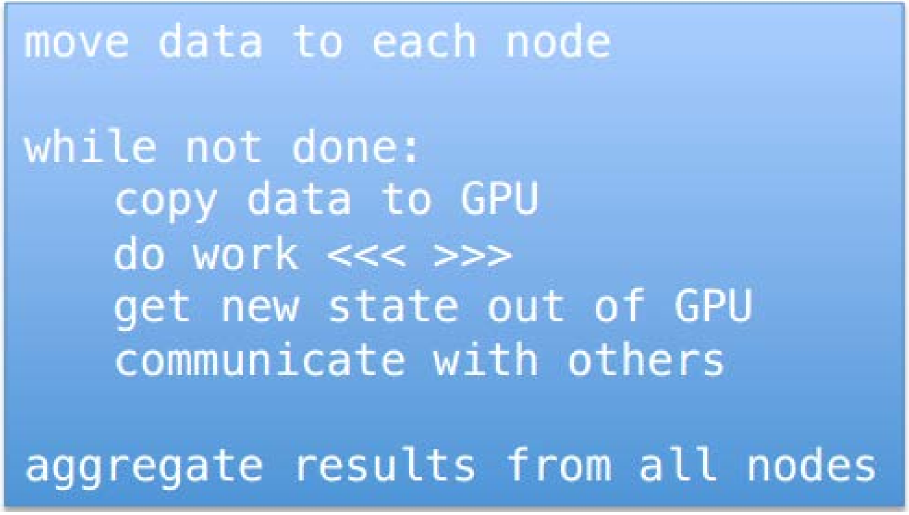
- testing if i, j are within grid
- padding the array in x to make it a multiple of 32, so each row starts at the beginning of a cache line – this uses a special routine cudaMallocPitch

Lecture 11: Programming on GPUs (Part 3)

Hybrid MPI/CUDA

1. One MPI process per GPU
   - GPU handling is straight forward
   - Wastes the other cores of the processor
2. Many MPI processes per GPU, only one uses it
   - Poses difficult load balancing problems
3. Many MPI processes share a GPU
   - Two processes cannot share the same GPU context, per process memory on GPU
   - Sharing may not always be possible
     - Limited memory on GPU
     - If GPUs are in exclusive mode

- CUDA will be:
  - Doing the computational heavy lifting
  - Dictating your algorithm & parallel layout (data parallel)
- Therefore:
  - Design CUDA portions first
  - Use MPI to move work to each node



```
move data to each node

while not done:
    copy data to GPU
    do work <<< >>>
    get new state out of GPU
    communicate with others

aggregate results from all nodes
```

Multi-GPU Programming

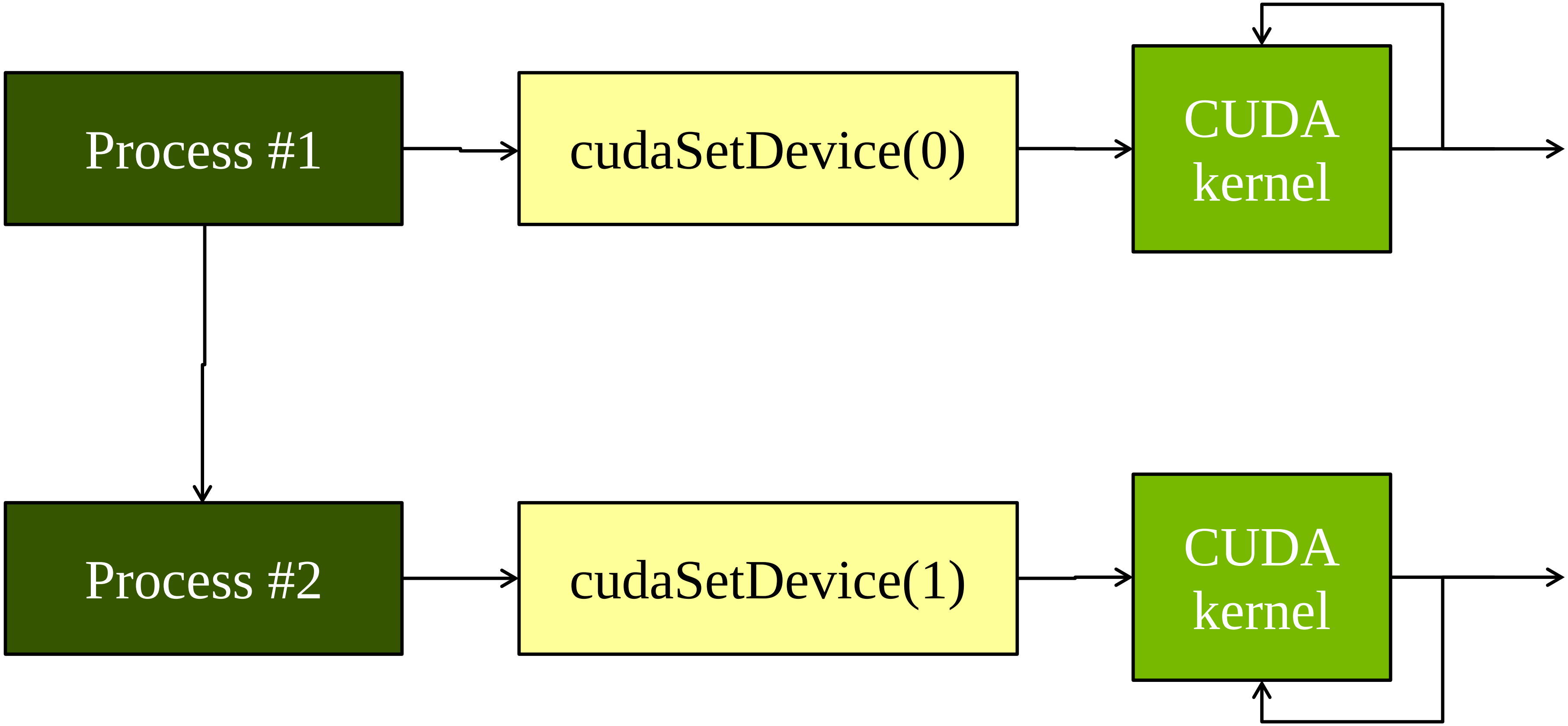
- Selecting GPUs

  The number of active GPUs visible to the rank is

  ```
  cudaGetDeviceCount(&deviceCount);
  ```

- One GPU per process (strategy 1)

  ```
  if(processesPerNode==deviceCount){
     id= nodeRank%deviceCount;
     cudaSetDevice(id);
  }
  else //ERROR
  ```


Process #1
cudaSetDevice(0)
CUDA kernel
Process #2
cudaSetDevice(1)
CUDA kernel

Compiling

- Kernel, kernel invocation, cudaMalloc, are all best off in a .cu file somewhere
- MPI calls should be in .c files
- nvcc processes .cu files to generate objective files
- mpicc/mpicxx processes .c/.cpp files to generate objective files
- If we need to call CUDA kernels from within an MPI task, we can wrap the appropriate CUDA-compiled functions with the “extern” keyword.

CUDA RNG

- RNG: random number generator
- CURAND
  - NVIDIA's library for random number generation in CUDA
  - CURAND can be called from the host and the device
  - CURAND Host API provides functions callable on the host to generate random data in GPU global memory
  - Can create multiple pseudorandom generators using different algorithms

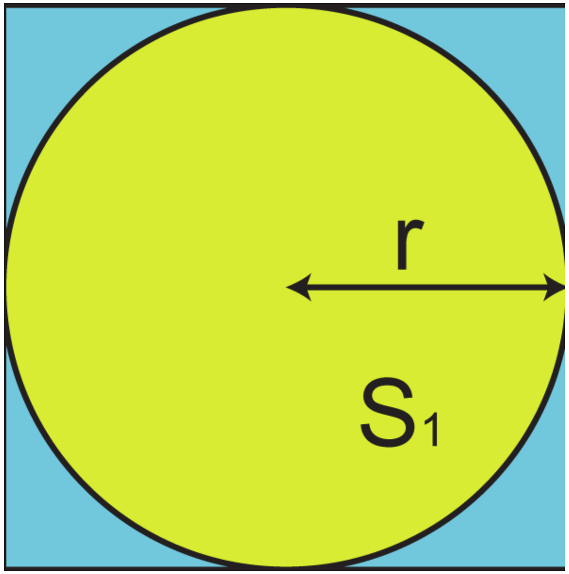
- Example:

```
curandGenerator_t r;
// argument tells which algorithm to use
curandCreateGenerator(&r, CURAND_RNG_PSEUDO_DEFAULT);
curandSetStream(r, stream); // optional
curandSetPseudoRandomGeneratorSeed(r, seed);
curandGenerateUniform(r, data, numElems);
curandDestroyGenerator(r);
```

Using CURAND in the Host

```
#include <curand.h>
int main()
{
. . .
curandGenerator_t gen;
float *devNum, *hostNum;
hostNum = new float[n];
cudaMalloc((void **)&devNum, n*sizeof(float));
. . .
curandCreateGenerator(&gen,CURAND_RNG_PSEUDO_DEFAULT);
curandSetPseudoRandomGeneratorSeed(gen, 12345);
curandGenerateUniform(gen, devNum, n);
cudaMemcpy(hostNum, devNum, n*sizeof(float),cudaMemcpyDeviceToHost);
. . .
curandDestroyGenerator(gen);
cudaFree(devNum);
. . .
}
```

PI Calculation



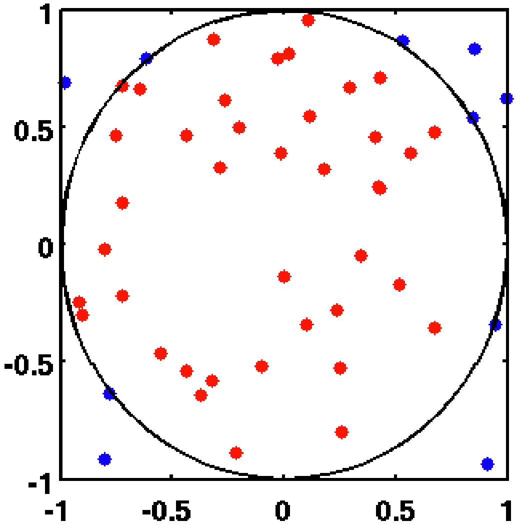
- Disk: $S_1 = \pi r^2$
- Square: $S_2 = 4r^2$
- $\pi = \frac{4S_1}{S_2}$

- To generate random numbers on the GPU memory:
  1. Include curand_kernel.h
  2. Allocate a memory space on device to store CURAND state.
  3. Initialize the state with a “seed”
  4. Generate random number sequences

```
#include <stdio.h>
#include <stdlib.h>
#include <curand_kernel.h> // CURAND lib header file
#define TRIALS_PER_THREAD 2048
#define BLOCKS 256
#define THREADS 256

int main(int argc, char *argv[]) {
   float host[BLOCKS * THREADS];
   float *dev;
   curandState *devStates;

   cudaMalloc((void **) &dev, BLOCKS * THREADS * sizeof(float));
   cudaMalloc( (void **)&devStates, BLOCKS*THREADS*sizeof(curandState) );
   …
}
```

```
__global__ void pi_mc(float *estimate, curandState *states) {
   unsigned int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int points_in_circle = 0;
   float x, y;
   // Initialize CURAND
   curand_init(tid, 0, 0, &states[tid]);
   for(int i = 0; i < TRIALS_PER_THREAD; i++) {
      x = curand_uniform(&states[tid]);
      y = curand_uniform(&states[tid]);
      // count if x & y is in the circule.
      points_in_circle += (x*x + y*y <= 1.0f);
   }
   estimate[tid] = 4.0f * points_in_circle / (float) TRIALS_PER_THREAD;
}
```

- __device__ void curand_init (unsigned long long seed, unsigned long long sequence, unsigned long long offset, curandState *state)
  - The curand_init() function sets up an initial state allocated by the caller. thread will use its own curandState to generate its own random number sequence
- __device__ float curand_uniform (curandState *state)
  - This function returns a sequence of pseudorandom floats uniformly distributed between 0.0 and 1.0
- __device__ float curand_normal (curandState *state)
  - This function returns a single normally distributed float with mean 0.0 and standard deviation 1.0.



- Generate many randomly distributed points within the square
- The area of the circle can be approximately obtained from the ratio of points inside of the circle and the total number of points.

References

- An Introduction to GPU Computing and CUDA Architecture, S. Tariq, NVIDIA Corporation
- CUDA C Programming Guide, NVIDIA Corporation
- CUDA by Example, An Introduction to General-Purpose GPU Programming, J. Sanders, E. Kandrot